SWE Data Construction, Automatically!

LIANGHONG GUO, Sun Yat-sen University, China
YANLIN WANG[*], Sun Yat-sen University, China
CAIHUA LI, Sun Yat-sen University, China
WEI TAO[*], Independent Researcher, China
PENGYU YANG, Sun Yat-sen University, China
JIACHI CHEN, Sun Yat-sen University, China
HAOYU SONG, Huawei Technologies Co, Ltd, China
DUYU TANG, Huawei Technologies Co, Ltd, China
ZIBIN ZHENG, Sun Yat-sen University, China

Constructing large-scale datasets for the GitHub issue resolution task is crucial for both training and evaluating the software engineering capabilities of Large Language Models (LLMs). However, the existing GitHub issue resolution data construction pipeline is challenging and labor-intensive. We identify three key limitations in existing pipelines: (1) test patches collected often omit binary file changes; (2) the manual construction of evaluation environments is labor-intensive; and (3) the fail2pass validation phase requires manual inspection of test logs and writing custom parsing code to extract test status from logs. In this paper, we propose SWE-Factory, a fully automated issue resolution data construction pipeline, to resolve these limitations. First, our pipeline automatically recovers missing binary test files and ensures the correctness of test patches. Second, we introduce SWE-Builder, a LLM-based multi-agent system that automates evaluation environment construction. Third, we introduce a standardized, exit-code-based log parsing method to automatically extract test status, enabling a fully automated fail2pass validation. Experiments on 671 real-world GitHub issues across four programming languages show that our method can effectively construct valid evaluation environments for GitHub issues at a reasonable cost. For example, with GPT-4.1 mini, our SWE-Builder constructs 337 valid task instances out of 671 issues, at $0.047 per instance. Our ablation study further shows the effectiveness of different components of SWE-Builder. We also demonstrate through manual inspection that our exit-code-based fail2pass validation method is highly accurate, achieving an F1 score of 0.99. Additionally, we conduct an exploratory experiment to investigate whether we can use SWE-Factory to enhance models' software engineering ability. After training five models on 2,809 Python task instances collected by our method, all models show improved software engineering ability. For example, the resolve rate of a trained Qwen2.5-Coder-14B-Instruct on SWE-bench Verified increases from 5.8% to 21.0%. We hope our method can accelerate the construction of large-scale, high-quality GitHub issue resolution datasets for both training and evaluation.

[*]Corresponding author.

Authors' Contact Information: Lianghong Guo, guolh8@mail2.sysu.edu.cn, Sun Yat-sen University, Zhuhai City, China; Yanlin Wang, yanlin-wang@outlook.com, Sun Yat-sen University, Zhuhai City, China; Caihua Li, 1471890903@qq.com, Sun Yat-sen University, Zhuhai City, China; Wei Tao, wtao@ieee.org, Independent Researcher, Shenzhen City, China; Pengyu Yang, 2454700600@qq.com, Sun Yat-sen University, Zhuhai City, China; Jiachi Chen, chenjch86@mail.sysu.edu.cn, Sun Yat-sen University, Zhuhai City, China; Haoyu Song, songhaoyu1@huawei.com, Huawei Technologies Co, Ltd, Beijing City, China; Duyu Tang, tangduyu@huawei.com, Huawei Technologies Co, Ltd, Beijing City, China; Zibin Zheng, zhzibin@mail.sysu.edu.cn, Sun Yat-sen University, Zhuhai City, China.

Conference acronym 'XX, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

Additional Key Words and Phrases: Github Issue Resolution, Benchmark, Large Language Models

ACM Reference Format:
Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. 2018. SWE Data Construction, Automatically!. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The GitHub issue resolution task, which involves addressing real-world software issues like bug fixing and feature enhancements [4, 40], is a crucial aspect of software maintenance [15, 22]. Given its practical importance, the task has become a key benchmark for evaluating the software engineering capabilities of Large Language Models (LLMs) [3, 7, 25, 29, 41, 46, 49, 52, 61]. A prominent example is SWE-bench [22], a large-scale benchmark for evaluation, whose success has inspired many subsequent benchmarks that extend coverage to more languages and issue types [2, 15, 16, 52, 56, 57]. More recently, researchers have begun to construct training datasets to improve the software engineering ability of models [20, 35, 53]. For example, SWE-Gym [35] collects a large-scale GitHub issue resolution dataset from Python repositories for agent-based model training, and achieves significant performance improvements on this task.

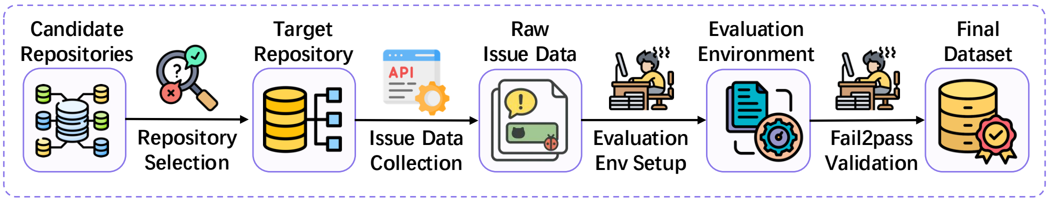


Fig. 1. Traditional pipeline of GitHub issue resolution data construction.

Existing works [22, 52, 53, 57, 59] typically construct GitHub issue resolution datasets following the pipeline of SWE-bench [22]. As shown in Figure 1, this pipeline consists of four stages. First, in the Repository Selection stage, target repositories are selected, typically based on popularity metrics like GitHub stars. Next, during Issue Data Collection, predefined patterns and the GitHub API are used to collect pairs of issues and their related pull requests as raw task instances. In the next stage, an evaluation environment is constructed for each task instance, consisting of a runtime environment and a script for running the relevant tests. Finally, the Fail2pass Validation stage validates each instance. Following the method from SWE-bench, tests are run both before and after applying the ground truth patch, and an instance is retained only if the test status is "fail" before and "pass" after the grount truth patch is applied. While this pipeline is widely adopted, the process is challenging and labor-intensive due to several problems, which we summarize as follows:

- **P1 Binary Test Files Missing in Issue Collection Stage.** We find that for task instances where the test patch includes changes to binary test files, the content of these changes is empty when collected via the GitHub API. This causes errors in building the test environment for this instance. This problem is especially prevalent in image processing and visualization repositories, such as Pillow (30.43% of instances affected) and matplotlib (22.10%).
- **P2 Manual Evaluation Environment Construction.** The existing pipeline requires significant manual effort to construct an evaluation environment for each task instance. This complexity

stems from the diversity of programming languages and repository configurations, resulting in highly varied dependencies and test commands. Furthermore, repositories often have multiple versions, and dependencies or test frameworks can change between them, making the environment setup particularly complex.

P3 Manual Fail2Pass Validation. This stage relies on manual effort to inspect test logs and write custom parser code to extract the final test status. The lack of consistency in test log formats complicates this task. Different repositories use different testing commands, resulting in varied log formats. Furthermore, the log format can change as the project evolves, even within the same repository requiring constant rewriting of parsers.

In this paper, we propose **SWE-Factory**, an automatic issue resolution dataset construction pipeline to address these problems. First, to resolve the binary test file missing issue, we automatically download the missing binary files and then remove their code changes from the test patch, to ensure the correctness of the test environment **(addressing P1)**. Second, we introduce **SWE-Builder**, a multi-agent framework that automates evaluation environment construction. Through the collaboration among different agents, SWE-Builder automatically collects information from the repository to create an environment construction script and corresponding test scripts. It also automatically executes tests and refines the environment based on the feedback. To further improve efficiency, the framework uses an Evaluation Environment Memory Pool to reuse previously successful setups **(addressing P2)**. Third, we propose an exit-code-based log parsing method to automate the fail2pass validation stage. Inspired by the observation that mainstream test frameworks report test outcomes via exit codes [1, 23, 32, 38], we standardize test status collection by capturing the exit code from test commands. Using this method, we can simply extract test status from logs without checking logs and writing parser code manually. **(addressing P3)**.

To investigate the effectiveness of SWE-Factory, we evaluate our method on 671 issues sampled from 12 repositories across four languages: Python, Java, JavaScript, and TypeScript. Through these experiments in Section 5, we identify several key findings: (1) First, experiments in Section 5.1 show that SWE-Builder can effectively construct evaluation environments for issues at a reasonable cost. For example, with GPT-4.1 mini, SWE-Builder successfully constructs 337 valid task instances out of 671 issues (50.2%) at an average cost of $0.047 per instance. (2) Second, ablation experiments in Section 5.2 demonstrate effectiveness of different components of SWE-Builder. (3) Third, through manual verification in Section 5.3, we demonstrate that our exit-code-based fail2pass validation method is highly accurate, achieving an F1 score of 0.99. Besides, we also verify that the exit code is a reliable indicator to determine the test status.

To investigate whether SWE-Factory can be used to enhance the software engineering ability of models, we conduct an exploratory experiment in Section 6. First, we use our method to collect 2,809 task instances automatically. Second, following previous studies [20, 35, 53], we use Kimi-K2 [42] as the agent model and the DeepSWE agent framework [26] to generate 2,809 corresponding agent trajectories. Finally, we finetune five LLMs on the collected trajectories based on the collected task instances. The evaluation results demonstrate that the performance of all models improves after finetuning. For example, the resolve rate of a finetuned Qwen2.5-Coder-14B-Instruct [19] on SWE-bench Verified [33] increases from 5.8% to 21.0%. These findings suggest the potential of SWE-Factory for enhanching the software engineering ability of models.

Our main contributions are summarized as follows:

- We present SWE-Factory, the first open-source fully automatic pipeline for constructing GitHub issue resolution benchmarks across multiple languages. Our code, datasets, and model weights are released at https://github.com/DeepSoftwareAnalytics/swe-factory.

- We propose SWE-Builder, a multi-agent framework that automates evaluation environment construction. Experiments show this method can efficiently construct valid evaluation environments for issue across different langauges.
- We propose an exit-code-based log parsing method to automate the fail2pass validation stage. Our human evaluation demonstrates the high accuracy of this method.
- Our exploratory experiment demonstrates SWE-Factory can effectively increase the software engineering ability of models. For example, after agent training using 2809 task instances collected by our method, the resolve rate of Qwen2.5-Coder-14B-Instruct on SWE-bench Verified increases from 5.8% to 21.0%.

2 Background of GitHub Issue Resolution Data Construction Pipeline

Following the method of SWE-bench [22], existing works [15, 52, 56, 57] often use a four-stage pipeline for constructing GitHub issue resolution datasets, including: (1) Repository Selection, (2) Issue Data Collection, (3) Evaluation Environment Construction, and (4) Task Validation. We introduce these stages in detail as follows.

Repository Selection. In this stage, researchers decide which GitHub repositories to use for collecting issue resolution data. According to previous studies [15, 22, 52, 53, 57], researchers often prioritize popular repositories, as measured by GitHub stars or download counts (e.g., via pip). These repositories generally feature a higher volume of issue-resolving activity, leading to a larger and more diverse collection of issue data, and possess comprehensive contribution documentation, which is beneficial for constructing the evaluation environment.

Issue Data Collection. This stage aims to collect issue-pull request pairs, which constitute the task instances for the GitHub issue resolution task. Existing works typically use the GitHub API and predefined patterns to collect issues and their corresponding pull requests. In these pairs, the issue's description serves as the task input. The pull request provides two key components: a "test patch" containing test cases to verify the resolution, and a "gold patch" that represents the ground-truth solution. A common filtering step is to discard pairs where the pull request does not include changes to test files. Additionally, metadata such as the base commit hash is collected to allow the repository to be checked out to the state it was in before the fix was applied. While this collection process is largely automated, we have identified a critical problem: for issues involving binary test files, the binary files retrieved via the GitHub API are often empty. This leads to errors during the evaluation environment construction, which we discuss further in Section 3.1.

Evaluation Environment Construction. This stage is aimed at building a runnable environment and preparing test scripts for each task instance collected in the previous step. To construct the runtime environment, the process begins by cloning the issue's repository and using the collected base commit to revert it to the state before the fix was applied. Next, all necessary software dependencies are installed to ensure the code can be executed properly. For the testing script, the test patch is first applied to introduce the tests that verify the issue's resolution. Then, the specific command required to run these target tests is created. This entire process is highly dependent on manual effort, as it often requires researchers to consult documentation for different repositories and their various versions to install the correct dependencies and determine the right test commands, making it a time-consuming phase. In our approach, we automate this process by generating a Dockerfile to define the runtime environment and a bash script to run the tests. We accomplish this generation using an LLM-based multi-agent system, introduced in Section 3.2.

Fail2pass Validation. This stage aims to filter the collected task instances to retain only the valid ones. Following the rules from SWE-bench, a task is considered valid based on a two-step testing process. First, the tests for the issue are run before applying the gold patch, and they must fail. Second, the same tests are run again after applying the gold patch, and this time they must pass.

This "fail-to-pass" change shows that the patch correctly fixes the issue. The most time-consuming part of this stage is getting the test results from the logs. This is because researchers need to carefully check many different log file styles and write specific code to parse them. In Section 3.3, we introduce our method to automate this task validation step.

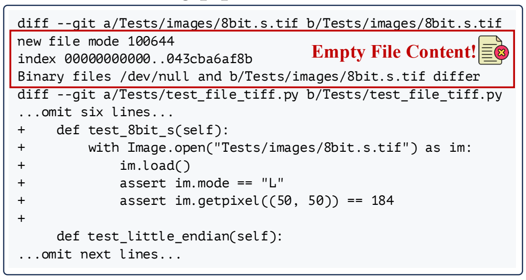
3 Methodology

In this section, we introduce our framework, SWE-Factory, designed to automate the GitHub issue resolution data construction pipeline. Our framework enhances this process across three main stages. First, for issue collection, we identify and resolve an issue within the SWE-bench script where it failed to collect certain binary test resources (Section 3.1). Second, to automate environment setup, we introduce SWE-Builder, a multi-agent system detailed in Sections 3.2. Third, to automate fail2pass validation, we present our exit-code-based fail2pass validation in Section 3.3.

3.1 Augmenting Raw Issue Data Collection to Include Binary Test Files

Following the methodology of SWE-bench, existing works typically rely on GitHub APIs and predefined patterns to collect issue–PR pairs for constructing issue resolution datasets. However, when a pull request involves binary files modification (e.g., .png, .tif, .zip), the patch directly downloaded via the GitHub API does not contain the actual binary content. As illustrated in Fig. 2, the test patch only includes a placeholder message such as “Binary files ... differ”, leaving the file body empty. We term this phenomenon the Binary Test Files Missing Issue. To the best of our knowledge, we are the first to identify this problem in the existing pipeline.

This issue has two major consequences. We present an example[1] in Fig. 2. First, it causes the target test cases to lack essential inputs. We can find that the “8bit.s.tif” is directly used as the test input of the test function “test_8bit_s”. When their contents are missing, the corresponding test code cannot obtain the required inputs and thus fails during execution, even if the test patch is applied successfully. Second, it leads to the failure of applying the test patch itself. Since the binary file is empty, the git apply operation cannot be completed, which also prevents subsequent modifications (e.g., changes in test_file_tiff.py) from being incorporated. As a result, the evaluation environment remains stuck at the base commit, without the intended test configuration. Consequently, these instances cannot pass the fail-to-pass validation step and are silently excluded from the final dataset. We also find that this problem is particularly frequent in libraries for image processing and visualization, such as Pillow (30.43% of instances affected), Manim (27.34%), and matplotlib (22.10%). This issue can limit the diversity of the collected data, especially for tasks that may require multi-modal understanding.



```
diff --git a/Tests/images/8bit.s.tif b/Tests/images/8bit.s.tif
new file mode 100644
index 00000000000..043cba6af8b
Binary files /dev/null and b/Tests/images/8bit.s.tif differ
diff --git a/Tests/test_file_tiff.py b/Tests/test_file_tiff.py
...omit six lines...
+    def test_8bit_s(self):
+        with Image.open("Tests/images/8bit.s.tif") as im:
+            im.load()
+            assert im.mode == "L"
+            assert im.getpixel((50, 50)) == 184
+
     def test_little_endian(self):
...omit next lines...
```

Fig. 2. Test patch of python-pillow__Pillow-7111.

To address this problem, we adopt a simple yet effective strategy. We first use predefined patterns to detect such binary files and automatically generate direct download commands (e.g., via wget) to recover their contents. Then, we clean the test patch by removing incomplete binary-related hunks, ensuring that the remaining patch can be applied successfully. Although lightweight, this approach proves highly effective in Section 5.2, enabling us to collect task instances that would otherwise be discarded by existing methods.

[1]https://github.com/python-pillow/Pillow/pull/7111.diff

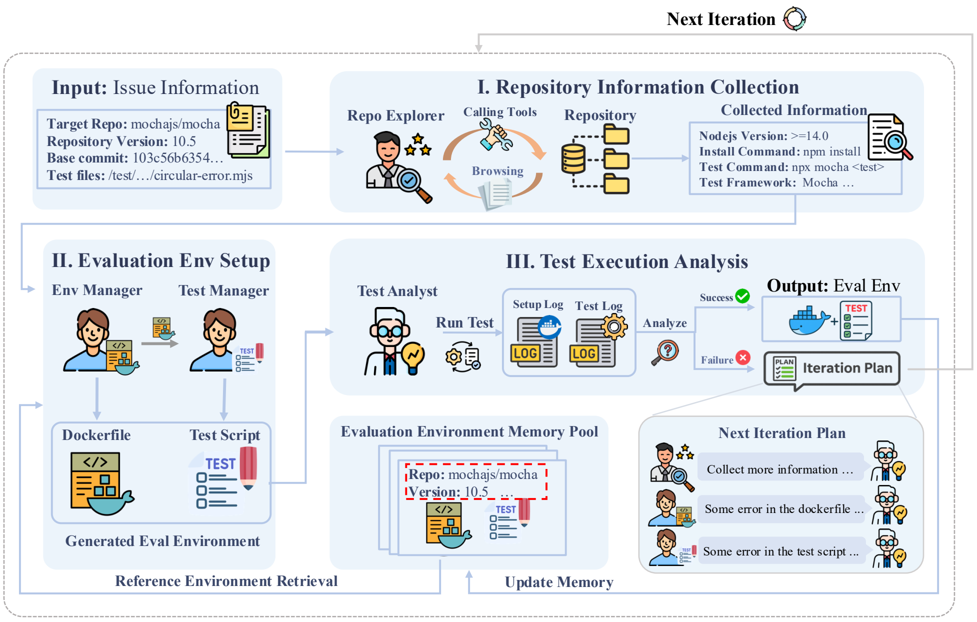


Fig. 3. Framework overview of SWE-Builder.

Table 1. List of tools used in SWE-Builder.

Tool Signature	Description	Output
`browse_file(fp, q)`	Extract content from file path `fp` relevant to query `q`.	Relevant snippets.
`browse_dir(fp, d)`	Return directory tree of `fp` up to depth `d`.	Directory structure.
`search_keyword(kw)`	Find file paths containing keyword `kw`.	Matched file paths.
`build_image(df)`	Build Docker image from `df` (Dockerfile).	Image ID or build logs.
`start_container(img)`	Launch container from image `img`.	Container ID or error logs.
`run_eval(ct, es)`	Run evaluation script `es` in container `ct`.	Evaluation logs.

3.2 Automatic Environemnt Setup Using SWE-Builder

Inspired by success of previous works [5, 8, 14, 21, 24, 25, 28, 39, 45, 49–51, 61] in using LLM-based agent to resolve software engineering tasks automatically, we propose **SWE-Builder**, a LLM-based multi-agent system to automate the evaluation environment construction for issues. We first introduce the roles of the four collaborative LLM-based agents designed to simulate the data construction process: (1) the Repository Explorer in Section 3.2.1, (2) the Environment Manager in Section 3.2.2, (3) the Test Manager in Section 3.2.3, and (4) the Test Analyst in Section 3.2.4. Following that, we describe SWE-Builder's memory mechanism to reuse previous experience, the Evaluation Environment Memory Pool, in Section 3.2.5. Finally, we detail the complete workflow of SWE-Builder in Section 3.2.6. The overview of SWE-Builder overview is presented in Figure 3. We provide the prompts for all proposed agents in our online Appendix[2].

[2]https://github.com/DeepSoftwareAnalytics/swe-factory/Appendix.pdf

3.2.1 Repository Explorer. The repository explorer is responsible for automatically collecting the information essential for constructing an evaluation environment for each issue. For each target repository, it autonomously extracts (1) environment dependencies from configuration files (e.g., `requirements.txt` for Python, `pom.xml` for Java); (2) the relevant test commands (such as `pytest` for Python, `mvn test` for Java); and (3) additional setup details from documentation files (e.g., `CONTRIBUTING.md`) or installation scripts. By automating this process, the Repository Explorer efficiently handles the environment diversity across different projects without manual intervention.

Algorithm 1: The workflow of Repository Explorer

```
Input: Repository $R$, LLM $\mathcal{L}$, Max
       Rounds $N$, Initial Context $C$ ,
Output: Summary of setup information $S$
1  Initialize: $S \leftarrow$ None;
2  for $i = 1$ to $N$ do
3      $actions, status, summary \leftarrow \mathcal{L}(C)$;
4      if $status$ is true then
5          $S \leftarrow summary$;
6          return $S$
7      for each API call $a_j$ in actions do
8          $o_j \leftarrow$ Execute $a_j$ on $R$;
9          Update $C$ with $o_j$;
10     $S \leftarrow summary$;
11 return $S$
```

Algorithm 1 demonstrates the workflow of repository explorer. This agent begins by initializing its context with a task-specific prompt. In each iteration, the agent queries the LLM with the current context to decide which tools to use and whether sufficient setup information has been collected. Three tools from Table 1 are available: `browse_file(fp, q)`, which calls the LLM to extract information relevant to the custom query from a given file; `browse_directory(fp, d)`, which returns the structure of the specified directory up to a given depth; and `search_keyword(kw)`, which retrieves file paths containing the target keyword. After executing the selected API calls and updating the context with the new observations, the agent summarizes the collected information. This process repeats until either the agent determines all necessary setup information has been obtained or the maximum number of rounds is reached, at which point the most recent summary is returned.

3.2.2 Environment Manager. The environment manager is responsible for constructing a reliable runtime environment that ensures all tests related to the target issue can be executed correctly. To achieve this, the agent outputs a `Dockerfile`, which scripts the complex environment configuration and installation commands into a reproducible and portable format. The Environment Manager operates in a stepwise manner: after receiving comprehensive environment information from the Repo Explorer, it automatically generates or updates the `Dockerfile` to match the requirements of the repository and the specific issue. Importantly, the agent preserves its generation history across iterations, ensuring that all previous modifications are retained for robust and reproducible environment setup. In cases where generation fails, the previous `Dockerfile` is used as a fallback to guarantee continuity.

3.2.3 Test Manager. The primary responsibility of the test manager is to automatically generate a shell script designed to execute tests relevant to a given issue. Its workflow commences only after its prerequisite agents—the Repository Explorer and Environment Manager—have completed their tasks. Leveraging the repository setup information from the Repository Explorer and the finalized Dockerfile from the Environment Manager, the test manager crafts the script to run within the specified containerized environment. Like the environment manager, it maintains a persistent context across iterations, ensuring that all script generation history is preserved.

Furthermore, to support other components in Section 3.1 and Section 3.3, the Test Manager's prompt includes specific instructions. First, to ensure binary file changes from test patches are correctly applied, the agent is prompted to insert the necessary commands into the test script, such as downloading new or modified files and deleting removed ones. Second, to ensure each test script provides a standardized interface for its final status, the agent is instructed to append commands that explicitly report the test's exit code. Following the convention where zero indicates success, the script captures the exit code of the main test command using `rc=$?` and then prints a unique marker: `echo "OMNIGRIL_EXIT_CODE=$rc"`. This allows other components to reliably and automatically parse the test status.

3.2.4 Test Analyst. The test analyst evaluates the quality of the generated evaluation environment (including the Dockerfile and evaluation script), and based on the results, orchestrates multi-agent iterations to refine it. Its workflow begins after it receives these outputs from the Environment Manager and Test Manager. The agent operates on the fundamental assumption that a well-constructed environment must allow the issue's ground-truth patch to pass all relevant tests. To verify this, the agent follows a standardized procedure using a sequence of tools in Table 1. It first builds the Docker image with `build_image(df)`, launches a container with `start_container(img)`, and finally executes the evaluation script with `run_eval(ct, es)` to apply the ground-truth patch and validate the tests. If the environment is successfully built, the agent analyzes the test logs to determine the test status. If the tests pass, the task is complete. If the tests fail, it analyzes the error information to generate an optimization plan. Similarly, if the environment construction fails, the agent analyzes the corresponding image build logs. In case of any failure, it generates a targeted plan and dispatches the relevant agents to initiate the next optimization iteration.

3.2.5 Evaluation Environment Memory Pool. We propose the evaluation environment memory pool, a component designed for SWE-Builder to reuse previously successful environment setups. This approach is motivated by a key observation: for issues within the same code repository, their required dependency environments and testing frameworks are often highly similar, especially for those from nearby versions. Building an evaluation environment from scratch for each issue is, therefore, both inefficient and prone to inconsistency. The memory pool addresses this challenge by archiving every successfully validated evaluation configuration, including both the dockerfile and the test script. Subsequently, when the Environment Manager and Test Manager handle a new issue, they first query the pool to find setups from the same repository and, from these, retrieve a reference environment setup from a nearby software version to use as a baseline. This strategy of reusing a pre-existing setup aims to accelerate the generation process and improve consistency across evaluation environments.

3.2.6 Orchestration of Different Agents . The workflow of SWE-Builder is an iterative process that orchestrates the four agents to construct and refine a valid evaluation environment, as detailed in Algorithm 2. The process begins with a comprehensive initial iteration. In this first step, the Repository Explorer collects setup information while the system retrieves a relevant reference from the Evaluation Environment Memory Pool. This combined information serves as the initial context for the Environment Manager and Test Manager to generate the first versions of the Dockerfile and evaluation script, which are then passed to the Test Analyst for validation.

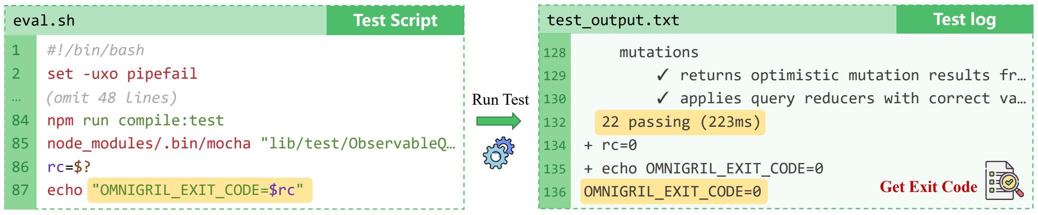
In contrast, subsequent iterations are not exhaustive but are instead targeted based on feedback from the Test Analyst. If a validation attempt fails, the Test Analyst identifies the cause of the error and provides specific guidance to the agent responsible for the flawed component. For example, if only the test script contains an error, the workflow invokes only the Test Manager to generate a revised script, bypassing the other agents. The corrected script is then passed back for another round of validation. This refinement cycle continues until the environment is successfully validated or the maximum number of iterations is reached.

Algorithm 2: Workflow of SWE-Builder

```
Input: Issue $\mathcal{I}$; evaluation environment memory pool $\mathcal{M}$; Max iteration number $\mathcal{N}$;
Repo Explorer Agent$_{\text{repo}}$; Environment Manager Agent$_{\text{docker}}$;
Test Manager Agent$_{\text{script}}$;Test Analyst Agent$_{\text{analyst}}$;
Output: Dockerfile $\mathcal{D}$, evaluation script $\mathcal{S}$
1  Initialization: $S_{\text{repo}}, S_{\text{dockerfile}}, S_{\text{script}} \leftarrow$ False;
2      // $S_{\text{repo}}$: collected information ready; $S_{\text{dockerfile}}$: Dockerfile generated; $S_{\text{script}}$: test script generated
3  for $i = 1$ to $\mathcal{N}$ do
4      if $!S_{repo}$ then
5          collected_information = Agent$_{\text{repo}}$.run();
6          if collected_information then
7              $S_{\text{repo}} \leftarrow$ True; Agent$_{\text{docker}}$.update_context(collected_information);
8              Agent$_{\text{script}}$.update_context(collected_information);
9      reference_val_env = $\mathcal{M}$.retrieve_closet_version($\mathcal{I}$);
10     Agent$_{\text{docker}}$.update_context(reference_val_env.dockerfile);
11     Agent$_{\text{script}}$.update_context(reference_val_env.eval_script);
12     if $S_{repo}$ and $!S_{dockerfile}$ then
13         $\mathcal{D}$, ok = Agent$_{\text{docker}}$.run();
14         if ok then
15             $S_{\text{dockerfile}} \leftarrow$ True;
16     if $S_{repo}$ and $S_{dockerfile}$ and $!S_{script}$ then
17         $\mathcal{S}$, ok = Agent$_{\text{script}}$.run($\mathcal{D}$);
18         if ok then
19             $S_{\text{script}} \leftarrow$ True;
20     if $S_{repo}$ and $S_{dockerfile}$ and $S_{script}$ then
21         analysis = Agent$_{\text{analyst}}$.run($\mathcal{D}$, $\mathcal{S}$);
22         if analysis.is_finish then
23             $\mathcal{M}$.update($\mathcal{I}$, $\mathcal{D}$, $\mathcal{S}$);
24             return $\mathcal{D}$, $\mathcal{S}$;
25         if analysis.guidance_collected_information then
26             $S_{\text{repo}} \leftarrow$ False; Agent$_{\text{repo}}$.update_context(analysis.guidance_retrieval);
27         if analysis.guidance_docker then
28             $S_{\text{dockerfile}} \leftarrow$ False; Agent$_{\text{docker}}$.update_context(analysis.guidance_docker);
29         if analysis.guidance_eval_script then
30             $S_{\text{script}} \leftarrow$ False; Agent$_{\text{script}}$.update_context(analysis.guidance_eval_script);
```

3.3 Fail2Pass Validation Using Exit-Code-Based Test Log Parser

Fail2pass validation is a critical step for ensuring the quality of benchmarks for the GitHub issue resolution task, following the methodology established by SWE-bench [22]. After the successful construction of the evaluation environment for a given issue, the validation workflow for the



eval.sh

```
#!/bin/bash
set -uxo pipefail
(omit 48 lines)
npm run compile:test
node_modules/.bin/mocha "lib/test/ObservableQ…
rc=$?
echo "OMNIGRIL_EXIT_CODE=$rc"
```

test_output.txt

```
128     mutations
129         ✓ returns optimistic mutation results fr…
130         ✓ applies query reducers with correct va…
132   22 passing (223ms)
134  + rc=0
135  + echo OMNIGRIL_EXIT_CODE=0
136  OMNIGRIL_EXIT_CODE=0
```

Fig. 4. An example of capturing test command's exit code in script to reflect test status.

instance involves three key stages: (1) applying test patche related to the given issue, and then executing the related tests on the codebase twice, once before and once after the gold patch is applied, to collect test logs from both runs; (2) manually inspecting these logs and developing custom parsers to extract the test status from them; and (3) retaining the instance for the final benchmark only if it exhibits a clear fail-to-pass transition.

However, the primary bottleneck in this phase is the manual process of inspecting logs and developing custom parsers to determine test status. This is a very challenging task because test log formats vary substantially across different programming languages, testing frameworks (e.g., pytest, JUnit, Jest), and repository configurations. Even within a single repository, testing framework versions and their output formats can change over time. Consequently, researchers often need to manually inspect a vast number of verbose test reports to understand their structure and then write repository-specific and version-specific parsing logic, typically relying on complex regular expressions. This process is very labor-intensive and severely limits the scalability and efficiency of constructing large-scale GitHub issue resolution benchmarks.

To automate the fail-to-pass validation pipeline, we replace complex log parsing with a standardized, exit-code-based approach. Our method leverages the common software engineering convention where a process exit code of zero signifies success and a non-zero value indicates failure. Instead of parsing diverse log formats, we modify the test execution script to report the outcome directly. This approach involves inserting two key commands into the test script immediately after the primary test command, as shown in the example in Figure 4. The first, `rc=$?`, captures the exit code of the test process. The second, `echo "OMNIGRIL_EXIT_CODE=$rc"`, prints this code prefixed with a unique, searchable identifier. This design establishes a simple and reliable interface for validation: the test status is determined by using a regular expression to match the OMNIGRIL_EXIT_CODE pattern, where an exit code of 0 indicates success and any non-zero value indicates failure. We implement this by providing a specific instruction in the prompt for our Test Manager agent (detailed in Section 3.2.3), which directs it to automatically include these commands to capture and report the exit code in the test scripts it generates.

The validation workflow using this exit-code approach is straightforward. After executing the test script for a given issue and obtaining the test log, our system parses the log to find the unique identifier, `OMNIGRIL_EXIT_CODE`. If the pattern is found, the captured exit code determines the outcome: a value of 0 signifies a successful test run (pass), while any non-zero value indicates a failure. If the identifier is not present in the log, the status of test is marked with an error. For an instance to be considered valid, it must demonstrate a clear fail-to-pass transition, yielding a non-zero exit code before the gold patch is applied and a zero exit code after.

4 Evaluation Setup

In this paper, we evaluate SWE-Factory from two aspects. First, we evaluate the effectiveness of SWE-Factory in Section 5, and the experimental setup is detailed in Section 4.1. Furthermore,

Table 2. Statistics of selected base models.

Model	Input Cost	Output Cost	Release Date
GPT-4.1-mini-2025-04-14	\$0.40 / 1M tokens	\$1.60 / 1M tokens	April 14, 2025
Kimi-K2	\$0.60 / 1M tokens	\$2.50 / 1M tokens	July 11, 2025
DeepSeek-V3-0324	\$0.27 / 1M tokens	\$1.10 / 1M tokens	March 24, 2025

Table 3. Task instances statistics of SweSetupBench.

Repository Name	Language	# Instances	# Versions	Time Span	# Stars
pallets/click	Python	32	11	2014–2025	16.5k
python-attrs/attrs	Python	34	24	2016–2025	5.5k
python-pillow/Pillow	Python	132	48	2013–2025	12.8k
assertj/assertj	Java	39	32	2013–2025	2.7k
checkstyle/checkstyle	Java	77	49	2015–2025	8.6k
eclipse-vertx/vert.x	Java	70	14	2016–2024	14.5k
mochajs/mocha	JavaScript	60	45	2012–2025	22.8k
iamkun/dayjs	JavaScript	28	7	2018–2023	47.9k
nodejs/undici	JavaScript	23	17	2024–2025	6.9k
apollographql/apollo-client	TypeScript	78	29	2016–2025	19.6k
tailwindlabs/tailwindcss	TypeScript	72	25	2017–2025	88.3 k
reduxjs/redux-toolkit	TypeScript	26	13	2021–2025	11.0k

we investigate whether SWE-Factory can enhance a model's software engineering capabilities in Section 6, with the corresponding setup described in Section 4.2.

4.1 Experiment Details

Base Model Selection. We select three models : *GPT-4.1-mini-2025-04-14* [34], *Kimi-K2* [42], and *DeepSeek-V3-0324* [10] as base models of SWE-Builder. Considering the high cost of running LLM-based agents, we do not use some of the most advanced models such as GPT-4.1 and Claude-4. The statistics of the selected base models are shown in Table 2.

Evaluation Dataset Construction. To evaluate the effectiveness of SWE-Factory, we construct **SweSetupBench**, a dataset containing 671 raw issues data from 12 well-recognized open-source repositories spanning four programming languages. First, we build a dataset of 2,441 issues from 12 open-source repositories. All selected repositories are well-recognized open-source projects, each with over 2.5k GitHub stars. These repositories span four programming languages—Python, Java, TypeScript, and JavaScript—which are among the most popular languages according to GitHub statistics. All issues included in the dataset are created before March 1st, 2025. Given the substantial computational cost of running LLM-based agent tools, we construct a smaller dataset for evaluation, refered to as SweSetupBench, by performing stratified sampling based on the repository version. Specifically, we randomly sample 20% of the issues from each version. For versions with very few issues, we ensure that at least one issue is included to maintain comprehensive coverage. As a result, SweSetupBench contains 671 issues from 12 repositories across four languages. The detailed statistics of SweSetupBench are presented in Table 3.

Hyperparameter Settings. In our experiments, we set the maximum number of iterations for SWE-Builder to 5. The temperature of the base models is set to 0.1. We also set the maximum number of retrieval rounds for the context retrieval agent to 10. In addition, we run the experiments with 20 parallel subprocesses to improve efficiency. We empirically set these hyperparameters based on our preliminary experiments.

Evaluation Metrics. In evaluation, we use evaluation metrics as follows:

- Output Rate (%): The proportion of tasks where our method successfully outputs a result.
- F2P Rate (%): The proportion of instances for which manual inspection confirms that the generated instance passes fail-to-pass validation. This process involves three experienced researchers manually labeling the outcome of each test log as "fail" or "pass" and resolving any disagreements through discussion.
- Time (minute): The average time required to process each task.
- Cost ($): The average LLM API cost per task.

4.2 Exploratory Experiment Details

Base Models for Training. For the choice of base models for training, we select three code-specific LLMs: Qwen-2.5-Coder-instruct-3B, 7B, and 14B [19], as well as two general LLMs: Llama-3.1-8B-instruct [31] and Qwen3-8B-instruct [43]. Considering that Qwen3 features a Hybrid Thinking mode and our collected agent trajectories do not contain thinking mode information, we switch Qwen3 to a non-thinking mode for the subsequent training and inference phases.

Training Details. We train our models on a cluster of 8 A800 GPUs. We utilize the MS-Swift framework[3] to perform full-parameter supervised fine-tuning (SFT) on the base models for 3 iterations. The training is conducted with a maximum sequence length of 65,536, bf16 precision, a learning rate of 1e-5, and a warmup ratio of 0.05. To enhance training efficiency, we enable FlashAttention [9]. Furthermore, to optimize for long sequence training, we activate packing with the YaRN [36] method and employ sequence parallelism, setting sequence parallel size to 2 for the Qwen2.5-Coder-3B-instruct, 8 for the Qwen2.5-Coder-14B-instruct, and 4 for the other three LLMs.

Evaluation Settings. We evaluate performances of our trained models on a single A100 GPU. We use vLLM as the inference engine, with a maximum sequence length set to 32,768, bf16 precision, and prefix caching enabled. We use the DeepSWE agent [26] as the agent framework. This agent framework equips the model with four custom-define tools to interact with the code environment: (1) *Execute Bash* for running shell commands and capturing their output; (2) *Search* for locating specific code snippets within files or directories; (3) *File Editor* for viewing, creating, and modifying file contents; and (4) *Finish/Submit* to signal the successful completion of the task. When running this agent, we set the model's temperature to 0 and the maximum number of agent iterations to 40.

Benchmarks and Metrics. To evaluate model performance, we use two popular GitHub issue resolution benchmarks: SWE-bench-verified [33] and SWE-bench-lite [22], containing 500 and 300 task instances, respectively. We use the following four metrics:

- **Resolve Rate (%):** the proportion of issues successfully resolved by the model.
- **Empty Patch Rate (%):** the proportion of tasks where the model submits an empty patch.
- **Tool Call Failure Rate (%):** the rate of failed tool calls made by the model.
- **Turns:** the average number of interaction rounds with the environment per task.

5 Evaluation

We summarize the following research questions (RQs) to evaluate SWE-Factory:

- **RQ1: What is the Effectiveness of SWE-Builder?**
- **RQ2: How Much Do Different Components of SWE-Builder Contribute?**
- **RQ3: What is the Correctness of Exit-Code-Based Fail2pass Validation?**

[3]https://github.com/modelscope/ms-swift

Table 4. SWE-Builder's performance on the SweSetupBench, with three different models. In this table, GPT-4.1-mini is short for GPT-4.1-mini-2025-04-14, DeepSeek-V3 is short for DeepSeek-V3-0324, and "min" is short for minute. ↑ means higher is better, ↓ means lower is better.

Model	Dataset	F2P Rate (%)↑	Output Rate (%)↑	Time (min)↓	Cost ($)↓
GPT-4.1 mini	Full	**50.2** (337/671)	**64.8** (435/671)	26.3	0.047
	Python	54.0 (107/198)	**73.7** (146/198)	19.0	0.040
	Java	**43.5** (81/186)	**50.5** (94/186)	31.8	0.061
	TS	55.1 (97/176)	**68.2** (120/176)	29.6	0.042
	JS	**46.8** (52/111)	**67.6** (75/111)	25.2	0.042
Kimi-K2	Full	47.8 (321/671)	63.2 (424/671)	30.2	0.056
	Python	54.0 (107/198)	70.7 (140/198)	26.5	0.051
	Java	33.9 (63/186)	48.4 (90/186)	38.9	0.073
	TS	**56.3** (99/176)	68.2 (120/176)	27.0	0.047
	JS	**46.8** (52/111)	66.7 (74/111)	27.7	0.050
DeepSeek-V3	Full	42.0 (282/671)	53.4 (358/671)	**23.0**	**0.037**
	Python	**54.5** (108/198)	70.2 (139/198)	16.2	0.030
	Java	30.1 (56/186)	39.8 (74/186)	26.4	0.047
	TS	42.0 (74/176)	49.4 (87/176)	28.6	0.035
	JS	39.6 (44/111)	52.3 (58/111)	20.9	0.035

5.1 RQ1: Effectiveness of SWE-Builder

In this section, we evaluate the effectiveness of SWE-Builder on the SweSetupBench dataset. We focus on two primary metrics: **Output Rate** and **Fail-to-Pass (F2P) Rate**. The Output Rate measures the proportion of tasks for which SWE-Builder successfully generates a complete evaluation environment (i.e., a Dockerfile and a test script). The F2P Rate, following the strict criteria of SWE-bench, measures the percentage of tasks that are manually verified to correctly transition from "fail" to "pass" after applying the gold patch, representing the portion of high-quality and valid instances for benchmarking.

The experiment results, presented in Table 4, demonstrate that SWE-Builder is effective at creating evaluation environments. Overall, all three models consistently achieve F2P rates over 42% and Output Rates exceeding 53%, with tasks completed in under 30 minutes on average and for less than \$0.06. For example, with GPT-4.1 mini, SWE-Builder achieves the best F2P Rate at 50.2%, successfully producing 337 valid instances out of 671 issues, and also attains the highest Output Rate of 64.8%. The performance of Kimi-K2 is comparable, with a strong F2P Rate of 47.8%, though at a slightly higher cost of \$0.056 per task. In contrast, DeepSeek-V3 stands out as the most cost-effective option, requiring the least amount of time (23.0 minutes) and the lowest cost (\$0.037) per task. In terms of efficiency, DeepSeek-V3 generates approximately 11.4 valid instances per dollar, significantly higher than GPT-4.1 mini (10.7) and Kimi-K2 (8.5), making it a superior choice for large-scale processing under budget constraints.

We also observe that the performance of SWE-Builder varies across programming languages when paired with different models. For Python tasks, all three models achieve great and comparable F2P rates, with DeepSeek-V3 showing a slight advantage. On Java tasks, GPT-4.1 mini performs significantly better than the other two models. For TypeScript and JavaScript, both Kimi-K2 and GPT-4.1 mini show strong and comparable results, outperforming DeepSeek-V3 by a notable margin.

Table 5. Ablation study of Binary Test File Detecting method moudule in SWE-Builder. The BTFD is short for Binary Test File Detecting method. ↑ means higher is better, ↓ means lower is better.

Model	Setting	F2P Rate (%)↑	Output Rate (%)↑	Time (min)↓	Cost ($)↓
GPT-4.1 mini	Baseline	45.5 (20/44)	72.7 (32/44)	22.0	0.042
	w/o BTFD	0 (0/44)	63.6 (28/44)	23.6	0.049
Kimi-K2	Baseline	50.0 (22/44)	63.6 (28/44)	29.9	0.061
	w/o BTFD	0 (0/44)	59.1 (26/44)	27.0	0.062
DeepSeek-V3	Baseline	50.0 (22/44)	50.0 (22/44)	27.5	0.042
	w/o BTFD	0 (0/44)	50.0 (22/44)	26.6	0.039

RQ1 Summary: SWE-Builder can effectively construct valid evaluation environments at a reasonable cost for issues from repositories in different programming languages.

5.2 RQ2: Ablation Studies of SWE-Builder

In this section, we conduct a series of ablation studies to investigate the contribution of different components within SWE-Builder. These experiments target four key modules: (1) the Binary Test File Detecting method (Section 3.1), (2) the Repository Explorer (Section 3.2.1), (3) the Evaluation Environment Memory Pool (Section 3.2.5), and (4) Execution Feedback (Section 3.2.4). For the first three components, we directly remove these components to measure their impacts. For the fourth component, Execution Feedback, we alter the evaluation method. In our standard process, the Test Analyst provides guidance based on dynamic feedback from building the environment with the Dockerfile and executing the test script. For the ablation study, we replace this with a static analysis approach: the agent generates refinement suggestions by only inspecting the code of the Dockerfile and test script, without any actual execution.

First, We conduct an ablation study to evaluate the impact of our Binary Test File Detecting (BTFD) method. In SweSetupBench, there are 44 task instances where the test patch includes binary test files. To isolate the impact of our method, this study focuses exclusively on these 44 instances. As shown in Table 5, removing the BTFD method (w/o BTFD) causes the F2P Rate of SWE-Builder to drop to 0% across all models, with a general decline in the Output Rate as well. This is because the original test patch does not contain the actual content of the binary files, causing the test patch application to fail. Consequently, the codebase remains at the base commit, preventing the successful setup of a valid test environment. These results show that this component enables SWE-Builder to construct evaluation environments for issues that involve binary test files.

We then evaluate the remaining three components on the full set of 671 issues in SweSetupBench, with results presented in Table 6. First, removing the Repository Explorer leads to a decline in both F2P and Output rates for all models, demonstrating its positive contribution. The performance drop for Kimi-K2 is less significant, and we believe this is because its training data may already contain information from these repositories. We also observe a substantial reduction in API costs with this removal, which we attribute to the elimination of frequent tool calls required for repository exploration. Second, removing the Memory Pool leads to an average decrease of 5.2% in the F2P rate across all three models, demonstrating the value of reusing previous experiences. This removal also increases the average time, as more iterations are needed without prior experience to leverage. Third, removing the Execution Feedback component results in a significant performance collapse, with the F2P Rate decreasing to near zero. This highlights its critical role in guiding the construction

Table 6. Ablation studies of three components in SWE-Builder. Exec Feedback is short for Execution Feedback, Memory Pool is short for Evaluation Environment Memory pool. min is short for minute. DeepSeek-V3 is short for DeepSeek-V3-0324 ↑ means higher is better, ↓ means lower is better.

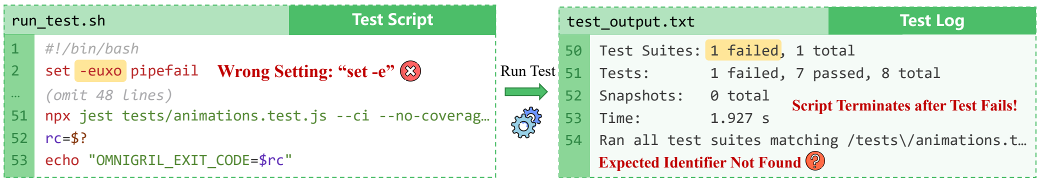
Model	Setting	F2P Rate (%)↑	Output Rate (%)↑	Time (min)↓	Cost ($)↓
GPT-4.1 mini	Baseline	50.2 (337/671)	64.8 (435/671)	26.3	0.047
	w/o Repo Explorer	42.9 (288/671)	55.3 (371/671)	25.9	0.021
	w/o Memory Pool	45.0 (302/671)	58.3 (391/671)	30.0	0.050
	w/o Exec Feedback	0.3 (2/671)	3.8 (26/671)	7.25	0.096
Kimi-K2	Baseline	47.8 (321/671)	63.2 (424/671)	30.2	0.056
	w/o Repo Explorer	47.1 (316/671)	60.4 (405/671)	32.6	0.032
	w/o Memory Pool	41.7 (280/671)	54.7 (367/671)	35.3	0.055
	w/o Exec Feedback	2.7 (18/671)	11.5 (77/671)	6.1	0.077
DeepSeek-V3	Baseline	42.0 (282/671)	53.4 (358/671)	23.0	0.037
	w/o Repo Explorer	33.5 (225/671)	44.3 (297/671)	20.0	0.016
	w/o Memory Pool	37.7 (253/671)	45.8 (307/671)	27.5	0.038
	w/o Exec Feedback	0.7 (5/671)	16.4 (110/671)	15.8	0.065

of a valid environment. Although task time shortens by skipping the environment construction and test execution, costs increase because the models become uncertain and repeatedly attempt refinements, increasing the average number of iterations. Overall, these experiments confirm that all three components contribute effectively to the performance of SWE-Builder.

RQ2 Summary: All components contribute to the effectiveness of SWE-Builder. First, the Binary Test File Detecting method is essential for building test environments for issues that include binary files. Second, the Repository Explorer improves results by providing helpful context, but at the cost of more API calls. Third, the Evaluation Environment Memory Pool boosts performance and shortens run time by reusing past solutions. Finally, Execution Feedback is the most critical part. Without it, performance collapses and the F2P Rate drops to nearly zero, making it almost impossible to build the test environment, even though it runs faster.

5.3 RQ3: Correctness of Exit-Code-Based Fail2pass Validation

In this section, we conduct a human evaluation to investigate the correctness of our exit-code-based fail2pass validation method. First, we collect 1217 task instances generated from GPT-4.1 mini, Kimi-K2, and DeepSeek-V3-0324 as described in Section 5.1. For each instance, we execute the test scripts before and after applying the gold patch to obtain a pair of test logs. We successfully obtained test log pairs for 1201 instances, after excluding those that failed due to environment build errors or test timeouts. We then employ our exit-code-based log parser to automatically determine the test status. Following SWE-bench [22], if the status changes from "fail" to "pass", the instance is labeled as "valid"; otherwise, it is labeled "invalid". Finally, to validate our method, we establish the ground truth through manual evaluation. Three experienced researchers manually inspect each test log, labeling its outcome as "fail" or "pass". If there are any disagreement in their labels, the three researchers will discuss to resolve them. Based on these consensus labels, we then determine the validity of each instance. We then compare the labels generated by our method against this ground-truth. The results are presented in Table 7.



```
#!/bin/bash
set -euxo pipefail
(omit 48 lines)
npx jest tests/animations.test.js --ci --no-coverag…
rc=$?
echo "OMNIGRIL_EXIT_CODE=$rc"
```

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 7 passed, 8 total
Snapshots:   0 total
Time:        1.927 s
Ran all test suites matching /tests\/animations.t…
```

Fig. 5. An example of early test script termination phenomenon.

Table 7. Human evaluation results on the correctness of exit-code-based fail2pass validation method.

Data Sources From	# Task Instances	# TP	# FP	# TN	# FN	Precision (%)	Recall (%)	F1
GPT-4.1 mini	434	337	0	97	0	100	100	1.000
DeepSeek-V3	352	277	0	71	4	100	98.6	0.993
Kimi-K2	415	309	0	94	12	100	96.3	0.981
Total	**1,201**	**923**	**0**	**262**	**16**	**100**	**98.3**	**0.991**

From experiment results from Table 7, we can find that our method shows a high accuracy in judging valid task instances, with a F1 score of 0.99. The small number of false negatives (FN) stem from 16 task instances that are classified as invalid due to the failure to parse a test status from their logs. Further analysis revealed that this is due to issues in the test scripts generated by LLM, which can be divided into two types: (1) Incorrect Identifier (4/16). This type means the test script fails to use our expected unique identifier (`OMNIGRIL_EXIT_CODE`), for example, using a typo like `OMNIGRIL_CODE`, which prevents our parser from extracting the status. (2) Early Script Termination (12/16). As shown in Figure 5, the test script incorrectly includes a “set -e” command (a shell option to exit immediately on error). This causes the test script to terminate upon test failure before it could execute the `echo “OMNIGRIL_EXIT_CODE=$rc”` line, thus preventing us from parsing the test status. We attribute these script generation errors to the stochastic nature of the LLM’s inference, which sometimes does not guarantee strict adherence to instructions. Fortunately, the resulting issues can be reliably detected through simple pattern matching and corrected manually.

Furthermore, to investigate whether the exit code itself accurately reflects the test status, we examine the individual logs from the aforementioned fail-to-pass runs. From the instances without script errors, we collect all 2,380 test logs that correctly outputted an exit code, labeling logs with an exit code of 0 as “pass” and non-zero as “fail”. Manual inspection confirm the accuracy of this method, which achieves a high F1 score of 1.0 and demonstrates its reliability to determine the status of individual tests.

RQ3 Summary: Our human evaluation demonstrates the high accuracy of the exit-code-based fail2pass validation method. Besides, we also verify that the exit code is a reliable indicator for determining the test status.

6 Exploratory Experiment: Enhance SE Ability of LLMs Using SWE-Factory

In this section, we explore whether SWE-Factory can enhance the software engineering ability of models. Following previous studies [27, 30, 48, 58], we construct agent training data to improve models’ performance. In Section 6.1, we will describe details about how to construct agent training data using SWE-Factory. And in Section 6.2, we will evaluate the performance of our trained model on two popular benchmarks: SWE-bench-verified [33] and SWE-bench-lite [22]. The detailed experiment settings are introduced in Section 4.2.

Table 8. Performances of base models and fintuned models on **SWE-Bench Verified** (top) and **SWE-Bench Lite** (bottom). Arrows indicate the preferred direction: ↑ means higher is better, ↓ means lower is better. Qwen2.5-Coder denotes *Qwen2.5-Coder-Instruct*. EP is the Empty Patch Rate (%), TF is the Tool Call Failure Rate (%), RR is the Resolve Rate (%), and "Turns" is the average number of interaction rounds.

Model	Size	EP (%, ↓)			TF (%, ↓)			Turns			RR (%, ↑)		
		Base	SFT	Δ	Base	SFT	Δ	Base	SFT	Δ	base	SFT	Δ
SWE-Bench Verified (500 instances)													
Qwen2.5-Coder	3B	90.8	16.4	−74.4	92.8	8.9	−83.9	28.10	39.26	+11.2	0.0	3.4	+3.4
Qwen2.5-Coder	7B	91.0	4.4	−86.6	82.4	1.01	−81.4	15.02	38.91	+23.9	0.0	13.6	+13.6
Qwen2.5-Coder	14B	4.6	6.6	+2.0	5.1	0.5	−4.6	37.22	37.50	+0.3	5.8	21.0	+15.2
Qwen-3-instruct	8B	78.0	17.4	−60.6	0.2	4.1	+3.9	38.64	39.33	+0.7	3.4	16.2	+12.8
Llama-3.1-instruct	8B	41.0	19.60	−21.4	43.2	5.6	−37.6	38.76	39.34	+0.6	1.2	5.2	+4.0
SWE-Bench Lite (300 instances)													
Qwen2.5-Coder	3B	94.0	14.0	−80.0	91.07	9.13	−81.9	27.74	39.34	+11.6	0.0	3.3	+3.3
Qwen2.5-Coder	7B	94.33	3.3	−91.0	82.62	0.83	−81.8	13.72	38.70	+25.0	0.0	8.0	+8.0
Qwen2.5-Coder	14B	6.33	6.67	+0.34	5.80	0.47	−5.3	36.59	37.62	+1.0	4.3	14.7	+10.4
Qwen-3-instruct	8B	78.67	19.33	−59.3	0.25	4.14	+3.9	38.68	39.40	+0.7	3.0	11.3	+8.3
Llama-3.1-instruct	8B	45.0	20.0	−25.0	43.32	5.46	−37.9	39.06	39.31	+0.3	1.0	4.3	+3.3

6.1 Agent Training Data Construction using SWE-Factory

Training Task Instances Collection Using SWE-Factory. We select 10 popular Python repositories and leveraged SWE-Factory to automatically construct 2877 task instances using GPT-4.1 mini. . To prevent data leakage during evaluation on SWE-bench, we ensure that the repositories selected for data construction are distinct from those in the SWE-bench benchmark. Each task instance includes basic issue information, the Docker-based evaluation environment, and associated test scripts. In our experiments, we employ the DeepSWE agent [26] as our agent framework. To ensure the agent could interact correctly within the environment of each task instance, we used GPT-4.1 mini to install the four custom-defined tools the agent framework relies on. This process ultimately yielded 2,809 fully prepared agent training environments for training. The final dataset statistics is present in our online Appendix[4]

Agent Trajectory Collection for LLM Training. Following previous studies [20, 35, 53], we collect agent trajectories—defined as the multi-turn interaction logs between a capable agent model and the environment—to train the model's agent capabilities. For this step, we select Kimi-K2 as the agent model because of its strong performance on SWE-bench and its reasonable cost. Subsequently, we use the DeepSWE agent [26] as the agent framework, setting the model's temperature to 0.2 and the maximum number of agent iterations to 40. For each of the 2,809 agent training environments we construct, we sample a single trajectory, resulting in a total of 2,809 trajectories for our agent training dataset for models.

6.2 Performances of Trained Models

We evaluate the software engineering ability of the base models and the finetuned models on both SWE-bench-verified and SWE-bench-lite. The evaluation settings are detailed in Section 4.2.

From the results in Table 8, we can find some key observations. First, all LLMs show an improved resolve rate on both datasets after fine-tuning, which demonstrates the effectiveness of the training

[4]https://github.com/DeepSoftwareAnalytics/swe-factory/Appendix.pdf

data collected by our method. Second, we find that the performance improvement after finetuning is more significant for models with larger parameter sizes. We attribute this to the stronger foundational coding abilities of larger models, allowing finetuning to unlock greater potential. Third, we notice that finetuned models tend to have more interaction turns with the environment, show a low empty patch rate, and show a lower tool call failure rate. We believe that SFT improves model performance by enhancing its capabilities in multi-turn interaction, code editing, and tool calling. Overall, our experiments demonstrate the potential of SWE-Factory for enhancing the software engineering capabilities of models.

Exploratory Experiment Summary: SWE-Factory can effectively enhance the software engineering ability of models by constructing training datasets automatically.

7 Related Work

7.1 Datasets for GitHub Issue Resolution

Recently, many benchmarks have been proposed to evaluate the ability of large language models (LLMs) to resolve real-world GitHub issues. Among them, SWE-bench [22] is the most widely used, providing 2,294 Python issues with paired post-PR test suites for automated evaluation. Subsequently, some works such as SWE-bench-Verified [33] improved SWE-bench to enhance its reliability [2, 18, 33, 47, 54]. Following that, other works such as Multi-SWE-bench [56] propose new issue resolution datasets covering more languages and modalities [2, 11, 12, 15, 52, 56, 57, 60]. Besides, to improve the issue resolution ability of models by training, researchers propose a number of training datasets [35] that support large-scale training and automated evaluation. For example, SWE-Gym [35] provides 2,438 Python tasks with Docker-based environments and unit tests for agent training. Additionally, other researchers [20, 37, 53] use LLM to synthesize issue data to train the issue resolution ability of models. For example, R2E-Gym [20] uses LLM to generate 8,700 task instances using automated test synthesis and back-translation.

7.2 Automatic Environment Setup

Recently, many works [6, 13, 17, 55] try to use LLM to automate repository environment construction. For example, ExecutionAgent [6] is an LLM-based agent capable of constructing repository environments across many languages. However, while these works address general environment setup, their objectives and methods are not directly applicable to the construction of issue resolution datasets, which has more specialized requirements. Subsequently, some researchers [44, 59] have focused specifically on automated environment setup for the issue resolution task. For example, SetupAgent [44] shows a strong ability to automate environment construction for issue resolution, but this work is not open-source. Compared to our method, these approaches still rely on manually written test log parsers, which require manual fail-to-pass (F2P) validation. Furthermore, these existing works are only applicable to Python repositories.

8 Threats to Validity and Limitations

A potential threat to validity is the scope of our evaluation dataset. To enhance the diversity of our data, we collect issues in four different programming languages from 12 repositories and ensure variety across the issue versions. In the future, we plan to conduct a more extensive evaluation on a more diverse dataset to enhance the generalizability of our conclusions. Another threat to validity is the selection of models for training in Section 6. To ensure model diversity, we select models with various parameter scales, from different model families and versions. However, due to computational resource constraints, our experiments are limited to models with up to 14B parameters. In the future, we plan to extend our experiments to a broader range of models.

9 Conclusion

In this paper, we propose SWE-Factory, an automatic benchmark construction pipeline for the GitHub issue resolution task. First, we identify a binary test file missing issue in the current data construction pipeline and propose methods to resove it. Second, we introduce a multi-agent system called SWE-Builder to automate evaluation environment construction with collaboration among LLM-based agents. To automate the fail2pass validation stage, we propose an exit-code-based log parsing method to robustly extract test status from test log without the need for manual parsing or inspection. Our experiments show that SWE-Builder can effectively construct evaluation environment for issues. For example, using GPT-4.1 mini, SWE-Factory successfully constructs evaluation environment for 50.2% issues. And our ablation studies show that all components contribute to the effectiveness of SWE-Builder. Besides, our human evaluation show that our exit-code-based fail2pass method achieves a high accuracy. Finally, our exploratory demonstrates that SWE-Factory can effectively improve the software engineering of models. We hope SWE-Factory can accelerate the collection of large-scale, high-quality GitHub issue resolution datasets for training and evaluation.

10 Data Availability

We have released our code, task instances, agent trajectories, and trained models at https://github.com/DeepSoftwareAnalytics/swe-factory.

References

[1] 2024. Exit status. https://en.wikipedia.org/wiki/Exit_status. Accessed: 2025-06-11.
[2] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992* (2024).
[3] Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular Architecture for Software-engineering AI Agents. *arXiv preprint arXiv:2406.11638* (2024).
[4] Tegawendé F. Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillère, Jacques Klein, and Yves Le Traon. 2013. Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. In *ISSRE*. IEEE Computer Society, 188–197.
[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).
[6] Islem Bouzenia and Michael Pradel. 2024. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *arXiv preprint arXiv:2412.10133* (2024).
[7] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *arXiv preprint arXiv:2406.01304* (2024).
[8] Zimin Chen, Yue Pan, Siyu Lu, Jiayi Xu, Claire Le Goues, Martin Monperrus, and He Ye. 2025. Prometheus: Unified Knowledge Graphs for Issue Resolution in Multilingual Codebases. *arXiv preprint arXiv:2507.19942* (2025).
[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., 16344–16359. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf
[10] DeepSeek AI. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024). arXiv:2412.19437 [cs.CL]
[11] Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. 2025. NoCode-bench: A Benchmark for Evaluating Natural Language-Driven Feature Addition. *arXiv preprint arXiv:2507.18130* (2025).
[12] Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. 2025. SWE-Dev: Evaluating and Training Autonomous Feature-Driven Software Development. *arXiv preprint arXiv:2505.16975* (2025).
[13] Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. 2025. EnvBench: A Benchmark for Automated Environment Setup. *arXiv preprint arXiv:2503.14443* (2025).
[14] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling.

arXiv preprint arXiv:2507.23370 (2025).

[15] Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. OmniGIRL: A Multilingual and Multimodal Benchmark for GitHub Issue Resolution. *arXiv preprint arXiv:2505.04606* (2025).

[16] Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. 2025. SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories? *arXiv preprint arXiv:2507.12415* (2025).

[17] Ruida Hu, Chao Peng, Xinchen Wang, and Cuiyun Gao. 2025. An llm-based agent for reliable docker environment configuration. *arXiv preprint arXiv:2502.13681* (2025).

[18] Yanxian Huang, Wanjun Zhong, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, Zibin Zheng, and Yanlin Wang. 2024. Agents in Software Engineering: Survey, Landscape, and Vision. *arXiv preprint arXiv:2409.09030* (2024).

[19] 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* (2024). arXiv:2409.12186 [cs.CL]

[20] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164* (2025).

[21] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. CoSIL: Software Issue Localization via LLM-Driven Code Repository Graph Searching. *arXiv preprint arXiv:2503.22424* (2025).

[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[23] JUnit. 2024. User Guide: Launcher API Exit Codes. https://junit.org/junit5/docs/current/user-guide/#launcher-api-exit-codes. Accessed: 2025-06-11.

[24] Yalan Lin, Yingwei Ma, Rongyu Cao, Binhua Li, Fei Huang, Xiaodong Gu, and Yongbin Li. 2024. Llms as continuous learners: Improving the reproduction of defective code in software issues. *arXiv preprint arXiv:2411.13941* (2024).

[25] Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024. MarsCode Agent: AI-native Automated Bug Fixing. *arXiv preprint arXiv:2409.00899* (2024).

[26] Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. 2025. DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL. Blog post, Together AI. Available at: https://www.together.ai/blog/deepswe.

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

[28] Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Rongyu Cao, Jue Chen, Fei Huang, and Binhua Li. 2025. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute. *arXiv preprint arXiv:2503.23803* (2025).

[29] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? *arXiv preprint arXiv:2406.01422* (2024).

[30] Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. 2025. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *arXiv preprint arXiv:2502.20127* (2025).

[31] Meta AI. 2024. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/.

[32] Mocha. 2024. Mocha Exit Code. https://mochajs.org/#mocha-exit-code. Accessed: 2025-06-11.

[33] OpenAI. 2024. SWE-bench Verified: A Human-Validated Subset for AI Model Evaluation. https://openai.com/index/introducing-swe-bench-verified. Accessed: 2024-10-21.

[34] OpenAI. 2025. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/.

[35] 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* (2024).

[36] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. YaRN: Efficient Context Window Extension of Large Language Models. *arXiv preprint arXiv:2309.00071* (2023). arXiv:2309.00071 [cs.CL]

[37] Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. 2025. SWE-Synth: Synthesizing Verifiable Bug-Fix Data to Enable Large Language Models in Resolving Real-World Bugs. *arXiv preprint arXiv:2504.14757* (2025).

[38] pytest. 2024. pytest exit codes. https://docs.pytest.org/en/latest/reference/exit-codes.html. Accessed: 2025-06-11.

[39] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. SpecRover: Code Intent Extraction via LLMs. *arXiv preprint arXiv:2408.02232* (2024).

[40] Wei Tao, Yucheng Zhou, Yanlin Wang, Hongyu Zhang, Haofen Wang, and Wenqiang Zhang. 2024. KADEL: Knowledge-Aware Denoising Learning for Commit Message Generation. *ACM Trans. Softw. Eng. Methodol.* 33, 5 (2024), 133:1–133:32.

[41] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. In *NeurIPS*.
[42] Kimi Team. 2025. Kimi K2: Open Agentic Intelligence. *arXiv preprint arXiv:2507.20534* (2025). arXiv:2507.20534 [cs.LG]
[43] Qwen Team. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025). arXiv:2505.09388 [cs.CL]
[44] Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. 2025. Automated Benchmark Generation for Repository-Level Coding Tasks. *arXiv preprint arXiv:2503.07701* (2025).
[45] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2024. AEGIS: An agent-based framework for general bug reproduction from issue descriptions. *arXiv preprint arXiv:2411.18015* (2024).
[46] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).
[47] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are" Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *arXiv preprint arXiv:2503.15223* (2025).
[48] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449* (2025).
[49] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
[50] Boyang Yang, Haoye Tian, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, and Bach Le. 2025. Enhancing Repository-Level Software Repair via Repository-Aware Knowledge Graphs. *arXiv preprint arXiv:2503.21710* (2025).
[51] 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* (2024).
[52] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. 2024. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859* (2024).
[53] John Yang, Kilian Leret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798* (2025).
[54] Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. 2025. Utboost: Rigorous evaluation of coding agents on swe-bench. *arXiv preprint arXiv:2506.09289* (2025).
[55] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2618–2640.
[56] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605* (2025).
[57] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. 2024. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. *arXiv preprint arXiv:2408.14354* (2024).
[58] Kechi Zhang, Huangzhao Zhang, Ge Li, Jinliang You, Jia Li, Yunfei Zhao, and Zhi Jin. 2025. SEAlign: Alignment training for software engineering agent. *arXiv preprint arXiv:2503.18455* (2025).
[59] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. 2025. SWE-bench Goes Live! *arXiv preprint arXiv:2505.23419* (2025).
[60] Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. 2025. SWE-Flow: Synthesizing Software Engineering Data in a Test-Driven Manner. *arXiv preprint arXiv:2506.09003* (2025).
[61] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 1592–1604.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009