Skywork-SWE: Unveiling Data Scaling Laws for Software Engineering in LLMs

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li,

Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and Yahui Zhou

Skywork AI, Kunlun Inc

Software engineering (SWE) has recently emerged as a crucial testbed for next-generation LLM agents, demanding inherent capabilities in two critical dimensions: sustained iterative problem-solving (e.g., >50 interaction rounds) and long-context dependency resolution (e.g., >32k tokens). However, the data curation process in SWE remains notoriously time-consuming, as it heavily relies on manual annotation for code file filtering and the setup of dedicated runtime environments to execute and validate unit tests. Consequently, most existing datasets are limited to only a few thousand GitHub-sourced instances. To this end, we propose an incremental, automated data-curation pipeline that systematically scales both the volume and diversity of SWE datasets. Our dataset comprises 10,169 real-world Python task instances from 2,531 distinct GitHub repositories, each accompanied by a task specified in natural language and a dedicated runtime-environment image for automated unit-test validation. We have carefully curated over 8,000 successfully runtime-validated training trajectories from our proposed SWE dataset. When fine-tuning the Skywork-SWE model on these trajectories, we uncover a striking data scaling phenomenon: *the trained model's performance for software engineering capabilities in LLMs continues to improve as the data size increases, showing no signs of saturation.* Notably, our Skywork-SWE model achieves 38.0% pass@1 accuracy on the SWE-bench Verified benchmark without using verifiers or multiple rollouts, establishing a new state-of-the-art (SOTA) among the Qwen2.5-Coder-32B-based LLMs built on the OpenHands agent framework. Furthermore, with the incorporation of test-time scaling techniques, the performance further improves to 47.0% accuracy, surpassing the previous SOTA results for sub-32B parameter models. Finally, we distill a set of practical guidelines aimed at further advancing LLM-driven software engineering in both academic research and industrial practice. We release the Skywork-SWE-32B model checkpoint to accelerate future research.

Keywords: Software engineering, Data scaling laws, LLMs

1. Introduction

> *Talk is cheap. Show me the code.*
>
> — Linus Torvalds

Two core capabilities define the emerging potential of Large Language Model (LLM) agents: the ability to engage in multi-turn interactions and to reason over long-context inputs (OpenAI, 2025, Team et al., 2023, Guo et al., 2025, Weng, 2023). Among real-world applications, software engineering (SWE) tasks (Jimenez

Figure 1: (Top) The Skywork-SWE model achieves 38.0% pass@1 accuracy on the SWE-bench Verified benchmark, outperforming previous open-source SoTA Qwen2.5-Coder-32B-based LLMs built on the OpenHands agent framework. Moreover, Skywork-SWE clearly demonstrates the data scaling law phenomenon for software engineering capabilities in LLMs, with no signs of saturation at $8,209$ collected training trajectories. *All evaluations are conducted in-house using a single attempt per instance, without verifiers or multiple rollouts, based on the OpenHands v0.32.0 framework.* **(Bottom)** Performance comparison among recent advanced approaches using OpenHands on SWE-bench Verified. With the incorporation of test-time scaling (TTS) techniques, Skywork-SWE achieves a further improvement to 47.0% accuracy, surpassing the latest Qwen and DeepSeek model series.

et al., 2024), which involve localizing bugs, modifying source codes, and validating fixes on real-world software issues collected from GitHub, stand out as a critical evaluation domain. Unlike conventional code generation tasks, which produce simple code snippets to solve competitive programming problems (Jain et al., 2024, Zhuo et al., 2024), SWE tasks require iterative problem-solving over extended interaction rounds to utilize code tools and the ability to manage long-context dependencies within code files to address the real-world software engineering challenges (Pan et al., 2024, Yang et al., 2025). The increasing prominence of benchmarking datasets such as SWE-bench (Jimenez et al., 2024), SWE-bench Verified (OpenAI, 2024e) reflects both the rising research interest and the inherent challenges of LLM-driven SWE.

Despite these advances, existing datasets still suffer from key limitations that hinder progress in the field:

- **Insufficient environment and validation support.** As detailed in Table 1, existing benchmarks typically lack comprehensive mechanisms for configuring executable runtime environments or standardized code

Table 1: Comparison of publicly available benchmark datasets for SWE tasks. *Executable Environment*: whether each instance is provided with an executable environment in which all necessary dependencies are pre-installed. *Verified Unit Tests*: whether the associated unit tests have been validated to ensure the correctness of both `FAIL_TO_PASS` and `PASS_TO_PASS` outcomes. *Code Execution Suite*: whether a unified and automated script is provided to automatically initialize, configure, and execute tests across diverse repositories without manual intervention. The Skywork-SWE dataset encompasses all three critical dimensions of SWE benchmarks and is systematically scaled to highlight both the volume (#Instances) and diversity (#Repos) of SWE datasets.

Dataset	Executable Env.	Verified Unit Tests	Code Exec. Suite	#Instances	#Repos
SWE-bench (Jimenez et al., 2024)	✓	✓	✗	2,294	12
SWE-bench Lite (Jimenez et al., 2024)	✓	✓	✗	300	12
SWE-bench Verified (OpenAI, 2024e)	✓	✓	✗	500	12
SWE-bench Extra (Badertdinov et al., 2024)	✗	✗	✓	6,376	1,974
SWE-Fixer (Xie et al., 2025)	✗	✗	✗	115,406	856
SWE-Smith (Yang et al., 2025)	✓	✓	✗	50,137	128
SWE-Gym (Pan et al., 2024)	✓	✓	✗	2,438	11
Skywork-SWE	✓	✓	✓	10,169	2,531

execution suites to systematically validate the generated code patches across diverse repositories. For example, SWE-bench-extra (Badertdinov et al., 2024) and SWE-Fixer (Xie et al., 2025) either omit executable environments entirely or lack rigorous test validations, resulting in inconsistent and non-reproducible evaluations.

- **Scarcity of high-quality training data.** Although some existing datasets are large in scale, few provide rigorously validated and high-quality training instances. This lack of openly available validated data has led open-source LLMs to consistently underperform compared to proprietary models on SWE tasks. For example, SWE-Dev (Wang et al., 2025) lacks rigorously validated training instances, and SWE-Gym (Pan et al., 2024) suffer from limited repository coverage.

- **Unclear applicability of data scaling laws.** The volume of training data for SWE tasks is notably smaller than that in other LLM domains (Pan et al., 2024). It remains uncertain whether data scaling laws (Kaplan et al., 2020b, Hoffmann et al., 2022a) still hold in the software engineering context. Addressing this question is crucial for guiding future dataset expansion and optimizing model training strategies.

In response to these challenges, we develop an automated data curation pipeline that systematically scales both the volume and diversity of Skywork-SWE datasets. As shown in Table 1, the Skywork-SWE dataset comprises 10,169 real-world Python task instances drawn from 2,531 GitHub repositories. Each instance in the Skywork-SWE dataset includes a detailed natural language description along with an associated executable runtime environment specifically designed to support automated execution and test validation. We propose a three-stage incremental pipeline consisting of data collection and pre-filtering, execution-based validation, and agent trajectory generation for SWE tasks within a unified framework. This approach ensures the high quality and diversity of the Skywork-SWE dataset by combining broad coverage of GitHub repositories with rigorous reproducibility.

Leveraging this dataset, we fine-tune our Skywork-SWE model on over 8,000 successfully validated trajectories, achieving 38.0% pass@1 accuracy on the SWE-bench Verified benchmark *without using verifiers or multiple rollouts*. This establishes a new state-of-the-art among the Qwen2.5-Coder-32B-based LLMs built on the OpenHands agent framework. As shown in Fig. 1, our extensive experiments reveal a clear data-scaling

trend: the trained model's software engineering capabilities consistently improve with increased training data size, validating the applicability of scaling laws to SWE tasks. Moreover, applying test-time scaling techniques boosts performance to 47.0% accuracy, outperforming prior SoTA results for LLMs with fewer than 32 billion parameters.

Skywork-SWE aims to bridge the gap between open-source and proprietary SWE agent models, fostering transparency, reproducibility, and progress in LLM-driven software engineering. Our main contributions can be summarized as follows:

- We propose an efficient and automated pipeline for SWE data collection, resulting in the Skywork-SWE dataset, a large-scale, high-quality dataset featuring comprehensive executable runtime environments.
- We release Skywork-SWE-32B, a powerful open-source code agent model tailored for SWE tasks, establishing a new performance benchmark among the same-scale open-source SWE agents.
- We empirically observe the data scaling law in SWE tasks, demonstrating consistent performance improvements with increased training data size. This finding not only validates the applicability of scaling laws within software engineering but also highlights the need for larger, well-curated datasets to further enhance model performance.

2. Related Work

2.1. Code-related Tasks in LLMs

Large Language Models (LLMs) have achieved remarkable progress across a wide range of code-related tasks (Jiang et al., 2024, Chang et al., 2024), ranging from code snippet generation (Leblond et al., 2023) to complex software engineering tasks (Jimenez et al., 2024). Below, we briefly outline these two major directions of LLMs in code-related tasks.

Code Generation empowers LLMs to synthesize functional programs from natural language descriptions. Early function-level benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) laid the groundwork for this task, spurring the development of code-oriented LLMs including AlphaCode (Leblond et al., 2023), Code Llama (Roziere et al., 2023), WizardCoder (Luo et al., 2023), StarCoder (Li et al., 2023), and DeepSeek-Coder (DeepSeek AI, 2024). Through curated training datasets and specialized fine-tuning techniques, these models exhibit strong performance on code generation tasks and have nearly saturated the performance limits of traditional benchmarks. To advance evaluation, recent benchmarks such as LiveCodeBench (Jain et al., 2024) and BigCodeBench (Zhuo et al., 2024) introduce contamination-free, practical, and challenging programming problems, offering a more rigorous assessment of the code capabilities of modern LLMs (Jaech et al., 2024, OpenAI, 2024d). In response, leading large reasoning models (LRMs), such as OpenAI's o3 (OpenAI, 2025), DeepSeek-R1 (Guo et al., 2025) and Kimi-k1.5 (Team et al., 2025), employ reinforcement learning to incentive chain-of-thought reasoning capability of LLMs to substantially improve performance on these code benchmarks.

Software Engineering (SWE) focuses on resolving real-world GitHub issues within repository environments, where agents are required to localize bugs, modify source code, and validate fixes based on execution results. This practical task marks a shift in the field—from static, single-turn code generation to dynamic, interactive coding workflows—significantly expanding the applicability and capabilities of LLMs in real-world software development scenarios. SWE-bench (Jimenez et al., 2024) and its successor, SWE-bench Verified (OpenAI,

2024e), serve as the canonical benchmark in this domain. They provide thousands of real GitHub issues accompanied by complete codebases, natural language descriptions, and human-filtered regression test suites to reliably evaluate LLM models' ability to solve real-world software issues. Creating training data for SWE agents is a difficult and labor-intensive process. Recently, numerous efforts have been devoted to synthesizing training data for SWE tasks, with each instance required to be validated in its corresponding runtime environment. SWE-Gym (Pan et al., 2024) provide executable environments featuring realistic SWE tasks and corresponding unit tests, albeit with a relatively limited scale of just over 2,000 instances. SWE-bench-extra (Badertdinov et al., 2024) extends the methodology used in constructing the SWE-bench benchmark (Jimenez et al., 2024), resulting in 6,415 Python issue-pull request instances. SWE-Dev (Wang et al., 2025) proposes a test-case construction pipeline using structured descriptions and execution validation to bypass the overhead associated with full runtime verification. To enable LLM-based agents to learn from large-scale SWE data, SWE-fixer (Xie et al., 2025) and SWE-Smith (Yang et al., 2025) synthetically generate `Fail_to_Pass` instances by injecting and validating plausible bugs, resulting in thousands of SWE task instances. Furthermore, reinforcement learning approaches—as exemplified by SkyRL (Cao et al., 2025)—combine asynchronous rollouts with verifier-based feedback, promoting long-horizon resolution strategies in SWE tasks.
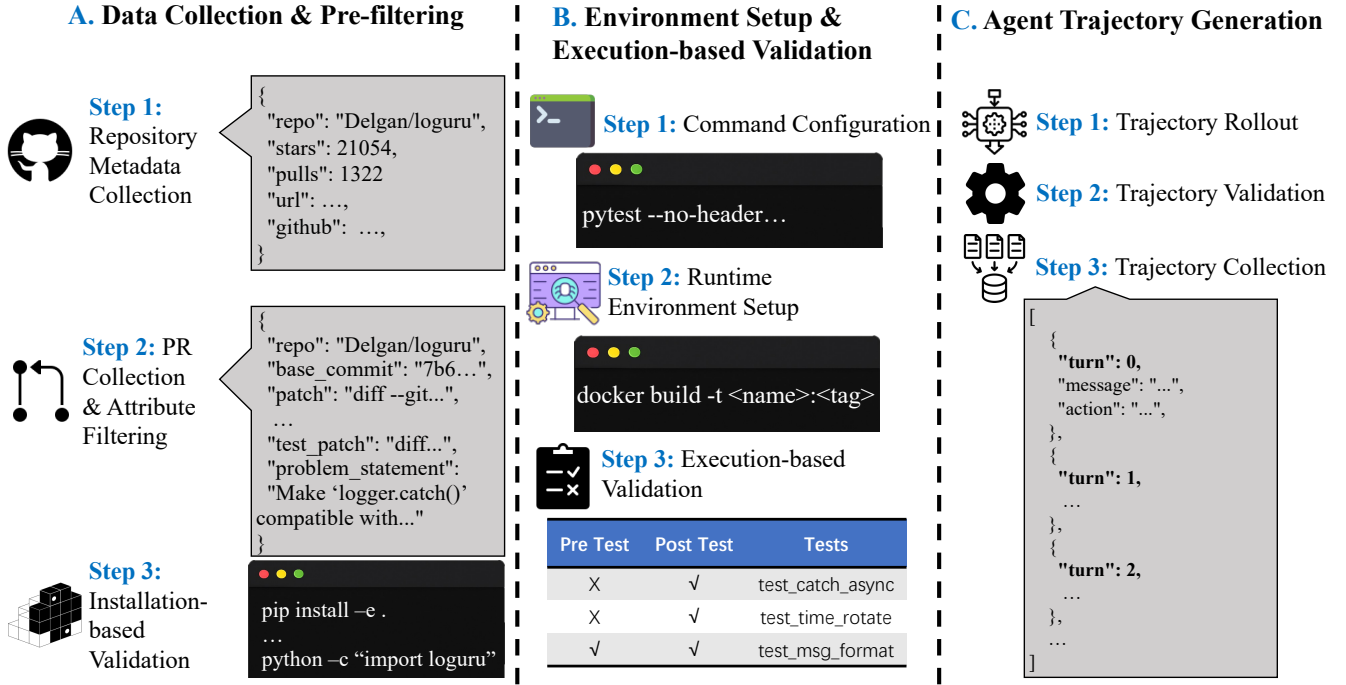
In this work, we propose new SWE data corpus consisting of over 10,000 real-world Python task instances collected from 2,531 distinct GitHub repositories. Each instance is paired with a dedicated runtime environment image to enable automated unit test validation. Leveraging this large-scale dataset, our Skywork-SWE model demonstrates clear evidence of the data scaling law in software engineering tasks.

2.2. Code Agent Framework in Software Engineering

Recent advancements in **software engineering (SWE) agents** have moved beyond generic "toy" code generation toward *repository-scale debugging and build orchestration*. OpenHands (Wang et al., 2024) provides an open, event-driven platform that enables LLM agents iteratively to edit files, execute shell commands and browse the Web inside sandboxed containers, establishing strong and reproducible baselines on SWE-bench. In contrast to interactive multi-step planners, Agentless (Xia et al., 2024) demonstrates that a streamlined *localise–repair–validate* pipeline can outperform many SWE agents on the SWE-bench benchmark while reducing costs by an order of magnitude. Moatless (Orwall, 2024) contends that effective context retrieval, rather than complex reasoning loops, is the key to patch generation; its open-source toolkit showcases scalable prompting and edit application across million-line codebases. Focusing on learning-based approaches, SWE-Fixer (Xie et al., 2025) integrates coarse-to-fine file retrieval and edit disentanglement to train open LLMs for SWE tasks. To enable LLM agents to autonomously operate computer interfaces for solving SWE tasks, *SWE-Agent* (Yang et al., 2024) further formalises an *Agent-Computer Interface* (ACI) that exposes editor, shell and test runners as structured actions. Collectively, these frameworks highlight three emerging trends: (i) *lean pipelines* that replace heavy planning with specialized tools (Agentless, Moatless); (ii) *retrieval-aware fine-tuning* for efficient patch synthesis (SWE-Fixer); and (iii) *rich ACIs* that enable full-stack repository manipulation (OpenHands, SWE-Agent).

In this study, we adopt OpenHands (Wang et al., 2024) as our code agent framework due to its strong empirical performance and widespread adoption in the open-source code agent community. Building upon this foundation, our Skywork-SWE model achieves the SOTA performance on SWE-bench Verified with test-time scaling techniques for sub-32B parameter models in the open-sourced code agent ecosystem.

Figure 2: Overview of our three-stage Skywork-SWE data collection pipeline. (1) **Stage A. Data Collection and Pre-filtering**: Step A.1 scrapes GitHub repo metadata. Step A.2 collects and filters relevant pull requests (PRs). Step A.3 validates PRs via installation checks. (2) **Stage B. Execution-based Validation**: Step B.1 configures unified execution commands. Step B.2 builds Docker-based runtime environments. Step B.3 validates task instances by executing unit tests. (3) **Stage C. Agent Trajectory Generation**: Step C.1 generates agent trajectories. Step C.2 validates trajectories via patch testing. Step C.3 collects validated trajectories for training.
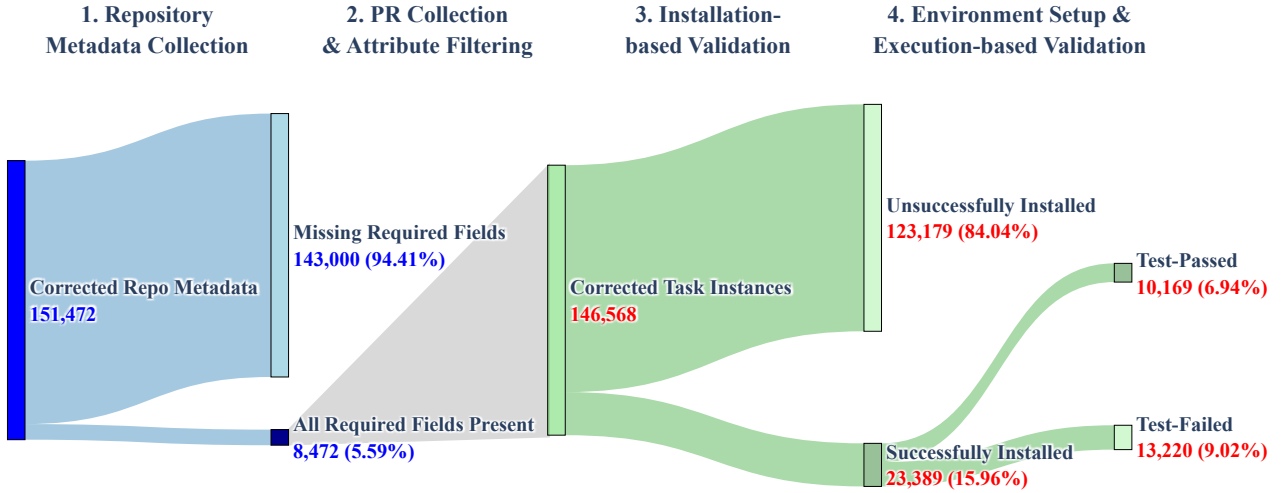
3. Method

In this section, we present the three-stage Skywork-SWE data collection pipeline used to construct our high-quality Skywork-SWE dataset. As shown in Fig. 2, the pipeline consists of: (1) **data collection and pre-filtering** (Sec. 3.1), (2) **environment setup and execution-based validation** (Sec. 3.2), and (3) **agent trajectory generation** (Sec. 3.3). We also provide a flow diagram in Fig. 3 to illustrate how data volume evolves throughout the pipeline. The curated data is subsequently used to train our Skywork-SWE-32B agent model, enhancing its ability to tackle software engineering tasks (Sec. 3.4).

3.1. Data Collection and Pre-filtering

The volume of high-quality training data for SWE tasks remains remarkably smaller than that in other domains (Wang et al., 2025). Previous work has shown that high-quality and diverse training data are crucial for efficiently training code agent models (Pan et al., 2024, Kaplan et al., 2020a). In this stage, we collect a large number of task instances from GitHub repositories and perform an initial round of data filtering.

Repository metadata collection. We collect metadata from a large number of open repositories on GitHub using the GitHub developer API. *To avoid data leakage when collecting GitHub repositories, we exclude those repositories already included in SWE-Bench Verified.* As shown in Fig. 2, the key fields in the metadata include

Figure 3: Visualization of data flow across four key hierarchical filtering steps in our data collection pipeline. The first three steps belong to the Data Collection & Pre-filtering stage in Fig. 2, while the last step corresponds to the Environment Setup & Execution-based Validation stage. The text in Blue and green indicates the number of repositories and task instances, respectively.

`repo`, which denotes the repository's `owner/name` identifier, and `stars`, which represent the number of stars the specific code repository has received. We prioritize GitHub repositories with higher `stars` counts. As depicted in Fig. 3, we scrape metadata for a total of 151,472 repositories. After filtering out the entries with missing required fields, we retain 8,472 valid metadata entries.

Pull request collection and attribute filtering. Using the metadata collected in the previous step, we scrape pull request (PR) data from the corresponding repositories via the GitHub developer API to form the initial task instances. Following the practice of SWE-bench (Jimenez et al., 2024), we select only merged PRs that link to and resolve a GitHub issue using keywords such as "closes/fixes/resolves #...". We retain only those PRs that modify test-related files (i.e., any path or filename containing "test" or "testing"). As shown in Fig. 2, the key fields for these task instances include `base_commit` (the commit hash of the repository representing the HEAD before the solution PR is applied), `patch` (the gold patch generated by the PR that resolved the issue), and `test_patch` (a test-file patch included in the solution PR). As shown in Fig. 3, we scrape a total of 146,568 initial task instances.

Installation-based validation. We filter the task instances by reverting the repositories to their `base_commit` and executing predefined installation commands in a base environment. As shown in Fig. 3, we discard 123,179 instances that fail to install, resulting in 23,389 valid instances.

3.2. Execution-based Validation and Runtime Environment Setup

The execution-based validation of candidate task instances relies on constructing dedicated runtime environments that satisfy the dependencies of a wide range of code libraries. Due to significant heterogeneity across repositories, manual configuration is time-consuming and difficult to scale. At this stage, we implement an automated per-instance pipeline to streamline the process.

```
1  {
2          "python": "3.9",
3          "packages": "requirements.txt",
4          "pip_packages": ["pytest", "hypothesis", "mock", "setuptools", ...],
5          "install": "pip install -e . || true; pip install -e .[test] ...",
6          "pre_install": "apt update && apt install -y make gcc g++ pkg-config",
7          "test_cmd": "pytest --no-header -rA --tb=no ..."
8  }
```

Figure 4: The default configuration snippet specifying environment setup, dependency installation (lines 2-6), and test execution commands (line 7) for all task instances.

3.2.1. Command Configuration

To standardize execution-based validation and runtime environment setup across diverse repositories, we define a unified default setup that includes environment initialization, dependency installation, and test command execution. Traditional approaches, such as that used in SWE-bench (Jimenez et al., 2024), require manually specifying separate configuration commands for each repository. However, due to the substantial variability in test environments across repositories and commits, this manual process is highly resource-intensive and difficult to scale. To address this, we adopt a generalizable configuration strategy inspired by SWE-bench-extra (Badertdinov et al., 2024), designed to maximize compatibility across a broad range of pull-request instances in various GitHub repositories.

The configuration comprises several components, as illustrated in Fig. 4. It sets Python 3.9 as the default runtime and installs essential system packages (e.g., make, gcc, g++, pkg-config) required for building native extensions. Python dependencies are installed via the requirements.txt file, supplemented by commonly used development and testing packages such as pytest, hypothesis, mock, and setuptools. To accommodate varying naming conventions across GitHub repositories, the configuration also includes fallback installation commands for optional extras like [test], [tests], and [dev]. Test execution is standardized using a unified pytest command that disables cache usage and suppresses deprecation warnings to produce consistent and reproducible results.

3.2.2. Runtime Environment Setup

We automate the construction of execution environments for candidate task instances by using Docker to build isolated runtime images based on a unified configuration defined in the command configuration step. To minimize redundant computation, images are incrementally built to enable reuse of base images for each instance. Meanwhile, successfully built images are cached to accelerate subsequent validations. As illustrated in Fig. 4, the image-building process is governed by a default configuration file that specifies core components such as the Python version, system packages, and dependency installation commands. The resulting Docker image follows a three-level hierarchy—base-level, environment-level, and instance-level images—each corresponding to a distinct build stage. Simplified versions of the scripts used for the three-level Docker image construction are listed in Fig. 5.

- The base-level image is built from Ubuntu 22.04 with the linux/x86_64 platform specified for compatibility. It installs essential system packages (e.g., build-essential, git, wget) via apt, followed by the installation of Miniconda and configuration of the conda-forge channel for Python package management.

Base-level image

```
1  FROM --platform=linux/x86_64 ubuntu:22.04
2  RUN apt update && apt install -y build-essential git wget python3-pip curl ...
3  RUN wget 'https://repo.anaconda.com/...' -O miniconda.sh
4  RUN bash miniconda.sh -b -p /opt/miniconda3
5  ENV PATH=/opt/miniconda3/bin:$PATH
6  RUN conda init --all
7  RUN conda config --append channels conda-forge
```

Environment-level image

```
1  FROM --platform=linux/x86_64 base.py.x86_64:latest
2  RUN bash setup_env.sh
```

setup_env.sh

```
1      conda create -n testbed python=3.9 -y
2      conda activate testbed && python -m pip install -r requirements.txt
3      pip install pytest mock ...
```

Instance-level image

```
1  FROM --platform=linux/x86_64 env.py.x86_64:latest
2  RUN bash setup_repo.sh
```

setup_repo.sh

```
1      git clone https://github.com/... /testbed
2      conda activate testbed
3      apt update && apt install -y make gcc g++ pkg-config
4      git reset --hard <commit_id>
5      pip install -e . || true; pip install -e .[test] ...
```

Figure 5: Dockerfile snippets illustrating the construction of a three-level Docker image structure, comprising a base-level image, an environment-level image, and an instance-level image.

- The environment-level image builds upon the base image by executing the `setup_env.sh` script, which creates a Conda environment with Python 3.9 (as specified in the `"python"` field), installs dependencies from the `requirements.txt`, adds additional development and testing packages listed under `"pip_packages"`.

- The instance-level image further extends the environment by executing `setup_repo.sh`, which clones the target repository, checks out a specific commit, installs system dependencies listed under the `"pre_install"` field, and performs an editable installation of the corresponding repository with optional extras defined in the `"install"` key.

3.2.3. Execution-based Validation

We evaluate each candidate task instance by applying its patches and executing the repository's test suite using the corresponding instance-level Docker image constructed during the runtime environment setup step. The validation process consists of two steps:

- **Empty test**: Apply the `test_patch` to the `base_commit` and run the test suite. Failing tests are labeled as `empty-FAIL`, and passing ones as `empty-PASS`.
- **Gold test**: Apply both the `test_patch` and the generated code `patch`, then rerun the test suite. Failing tests are labeled as `gold-FAIL`, and passing ones as `gold-PASS`.
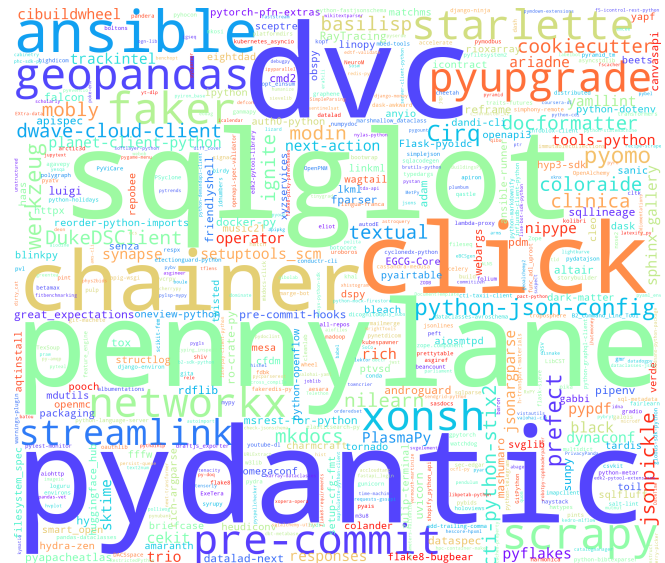
We define `FAIL_TO_PASS` as the set of tests that fail in the empty test (`empty-FAIL`) but pass in the gold test (`gold-PASS`), and `PASS_TO_PASS` as those passing in both stages. Only instances with a non-empty `FAIL_TO_PASS` set are retained, indicating that the applied patch resolves at least one failing test case and successfully corrects previously failing behavior. As shown in Fig. 3, out of 23,389 candidates, 13,220 instances are filtered out during validation, resulting in a final dataset of 10,169 verified instances.
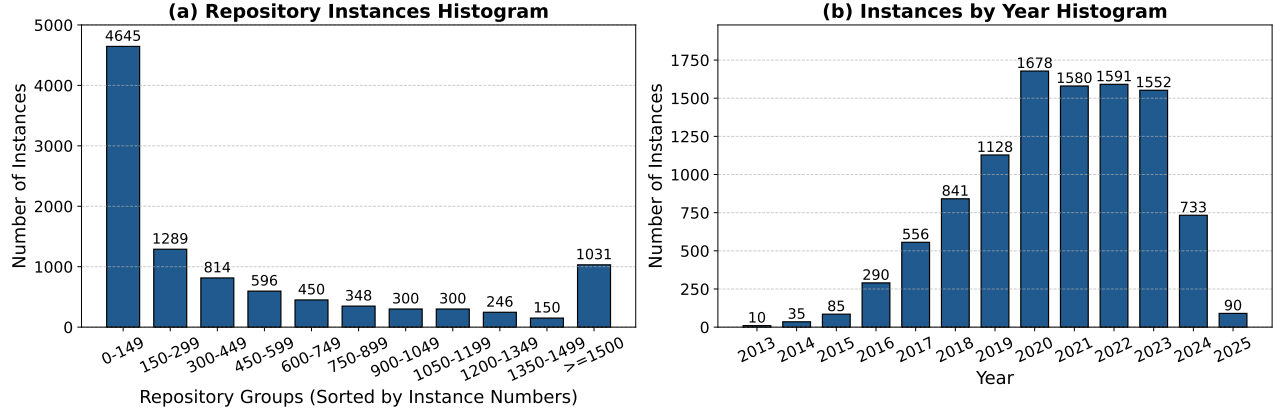
3.2.4. Dataset Statistics

Category	Metric	Skywork-SWE	SWE-Gym Lite	SWE-bench Verified
Size	# Instances	10,169	230	500
	# Repos	2,531	11	12
Issue Text	# Words	140.3	186.2	189.3
Hints Text	# Words	62.2	151.4	151.9
	# Hints	2,459	155	338
Gold Patch	# Files edited	2.5	1.0	1.2
	# Func. edited	2.3	1.4	2.1
	# Hunks edited	6.0	1.6	2.4
	# Lines edited	74.2	10.6	14.3
Tests	# Fail to Pass	10.2	2.0	3.0
	# Pass to Pass	86.2	99.9	120.3
	# Total	96.4	101.9	123.3

Table 2: Dataset statistics for Skywork-SWE, SWE-Gym Lite (Pan et al., 2024), and SWE-bench Verified (OpenAI, 2024e). All metrics are reported as per-instance averages, except for those under the Size category and the #Hints metric.

Figure 6: Word cloud of repository names in the Skywork-SWE dataset. Font size is proportional to the number of instances in each repository. The Skywork-SWE dataset demonstrates high diversity across the collected GitHub repositories.

The final Skywork-SWE dataset comprises 10,169 validated instances collected from 2,531 unique repositories, obtained through the execution-based validation process. Each instance is paired with a dedicated Docker image to support reproducible execution. On average, each image occupies approximately 1.2 GB, resulting in a total storage footprint of around 11.9 TB. We summarize key statistics and characteristics of the Skywork-SWE dataset as below.
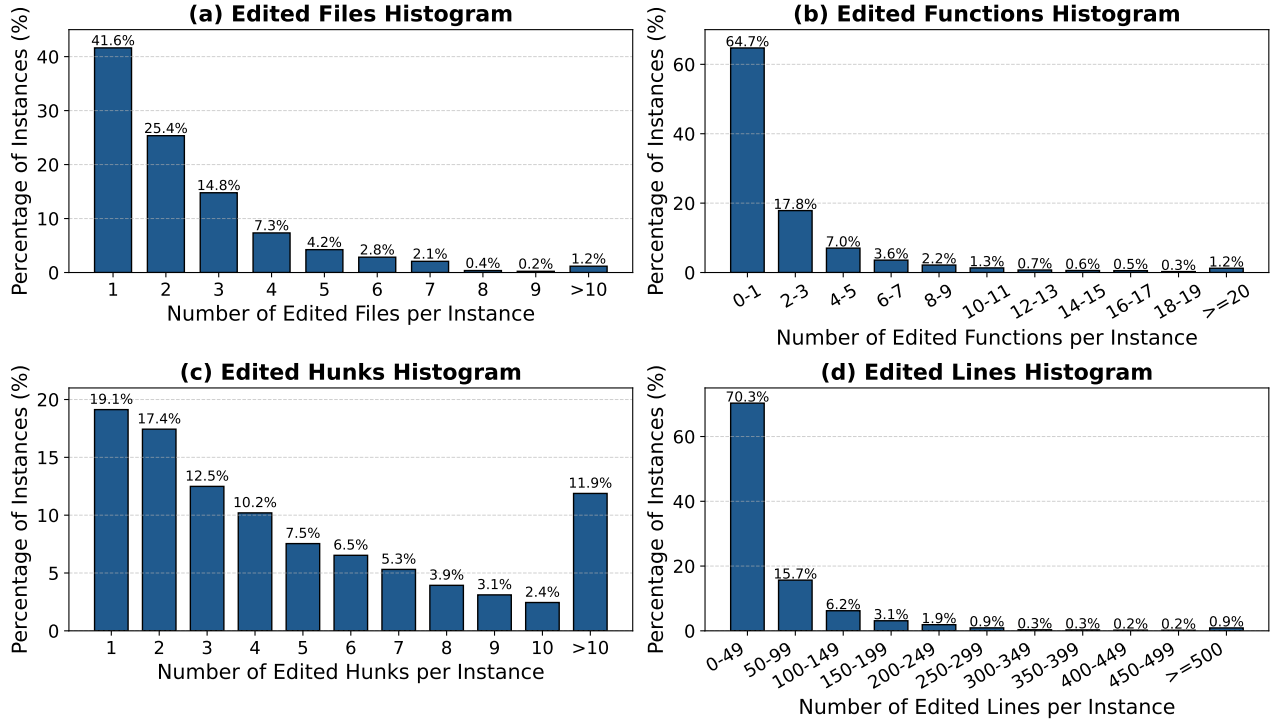
Figure 7: Repository- and year-wise histograms on the Skywork-SWE dataset. (a) The x-axis denotes the number of instances per repository, with every 150 repositories grouped into a single bin for clarity. (b) The x-axis indicates the year in which each issue was created, reflecting the temporal distribution of instance counts. The Skywork-SWE dataset reflects substantial diversity in both temporal coverage and repository sources.

Scale. Skywork-SWE significantly exceeds the scale of existing benchmarks as shown in Table 2. It includes 10,169 validated instances—more than 20 times the size of SWE-Gym Lite and SWE-bench Verified, which contain 230 and 500 instances, respectively. These instances span over 2,500 repositories, whereas previous datasets include at most 12 repositories. This broader coverage introduces a wider range of real-world software engineering scenarios.

Diversity. The dataset exhibits substantial diversity in GitHub repository sources and follows a long-tailed distribution. Fig.6 reflects this diversity, showcasing both well-known projects (e.g., *pydantic*, *dvc*, *sqlglot*, *pennylane*) and a wide array of smaller repositories, indicating a broad representation of software projects. Fig. 7(a) further reveals a pronounced long-tailed distribution of instance counts across repositories. Approximately 450 repositories (roughly 4.4%) account for over 66% of all instances, whereas the remaining 9,719 repositories contribute less than 34%. Notably, more than 9,000 repositories contain fewer than three instances each, significantly enhancing the dataset's diversity.

Temporal Distribution. The histogram in Fig. 7(b) illustrates the annual distribution of dataset instances from 2013 to 2025. In the early years (2013–2015), the number of instances remains relatively low (10, 35, and 85, respectively). A noticeable increase begins in 2016, with the count exceeding 500 in 2017 and reaching a peak of 1,678 in 2020. From 2021 to 2023, the number of instances consistently exceeds 1,500 per year, and 2024 alone contributes over 700 instances. Overall, more than 89.5% of all instances originate from the 2018–2024 period, reflecting a strong focus on recent software development activity.

Edit Complexity. Skywork-SWE instances exhibit varying degrees of structural complexity in the associated patch edits. As shown in Table 2, the average gold patch typically involves more modifications across multiple lines, files, functions, and code hunks than SWE-Gym Lite and SWE-bench Verified. Specifically, Fig. 8 (a) illustrates that 41.6% of instances involve edits to a single file, yet over 80% modify up to three files, demonstrating both focused and distributed complexity. Fig. 8(b) shows that 64.7% of instances affect fewer than two functions, while less than 2% involve changes to more than 15 functions. In Fig. 8(c), nearly 50% of instances comprise one to three hunks, whereas edits spanning more than ten hunks account for less than 12% of cases. Finally, Fig. 8(d) shows that 70.3% of instances involve fewer than 50 edited lines of code, and more than 85% remain within 100 lines. In terms of granularity, nearly half of the instances involve

Figure 8: Statistical analysis of instance-level edits on the Skywork-SWE dataset. (a) Histogram of the number of edited files per instance. (b) Histogram of the number of edited functions per instance. (c) Histogram of the number of edited code hunks per instance. (d) Histogram of the number of edited code lines per instance.

one to three code hunks, and 70.3% involve fewer than 50 edited lines, suggesting a mixture of concise and non-trivial modifications.

Validation Strength. Each instance is validated through unit tests, with many exhibiting multiple cases. As shown in Table 2, the FAIL_TO_PASS coverage per instance is more comprehensive than that of prior benchmarks. This allows for more rigorous verification of patch correctness and encourages robust evaluation of model outputs.

Overall, Skywork-SWE offers a large-scale, diverse, and structurally rich dataset that features rigorous execution-based validation and an emphasis on recent development trends, making it a valuable resource for software engineering tasks.

3.3. Agent Trajectory Generation

High-quality agent trajectories are crucial for agent model training. In this stage, we systematically generate, validate, and collect agent trajectories from the task instances in Skywork-SWE to offer reliable training data for our Skywork-SWE-32B model.

Trajectory Rollout. We leverage multiple high-performing code-focused proprietary LLMs to automatically generate agent trajectories for each task instance using OpenHands v0.32.0 (Wang et al., 2024) code agent framework. Each trajectory is capped at a maximum of 100 rollout turns per instance.

Trajectory Validation. We rigorously validate the generated trajectories by applying their final patches to the corresponding task instances and executing the repository's test suite. A trajectory is considered valid if

its final patch passes all tests, indicating that the underlying issue has been successfully resolved.

Trajectory Collection. We aggregate the validated trajectories as multi-turn supervised fine-tuning data, with only those whose final patches pass all tests in the validation step being retained. This filtering ensures consistently high data quality, directly enhancing the effectiveness of the agent model training.

Following the three aforementioned steps, we collect over $8,000$ successful agent trajectories featuring multi-turn interactions and long-context dependencies. Each trajectory is verified to correctly pass all unit tests within its corresponding runtime environment. Further details on model-specific rollout results and the trajectory composition can be found in Sec. 4.3.

3.4. Training the Skywork-SWE Agent Model

After carefully filtering for format consistency, the final set of successful trajectories was reduced to 8,209 instances, which were used to train our Skywork-SWE model. We fine-tune our Skywork-SWE agent model on the collected trajectories using supervised learning (torchtune maintainers and contributors, 2024), with Qwen-2.5-Coder-32B-Instruct (Hui et al., 2024) as the base model. Our primary goal is to validate the effectiveness of the proposed Skywork-SWE dataset and its associated runtime environments. While reinforcement learning remains a promising direction, it requires instance-level validation within runtime images to obtain verified reward signals (Cao et al., 2025). We leave this engineering-intensive task for future work.

4. Experiments

We structure this section as follows. We first describe the training setup, baseline methods, and evaluation benchmark in Section 4.1. We then present quantitative results on the SWE-bench Verified benchmark and analyze data scaling trends for SWE tasks in Section 4.2. Finally, we provide an in-depth experimental analysis in Section 4.3.

4.1. Experimental Setup

Implementation details   We fine-tuned the Qwen2.5-Coder-32B-Instruct (Qwen, 2024a) model using TorchTune framework (torchtune maintainers and contributors, 2024) on 8 NVIDIA H800 GPUs for 12 hours to obtain Skywork-SWE-32B model. Training was conducted using the AdamW optimizer with a weight decay of 0.01 and a cosine learning rate schedule. The Skywork-SWE-32B model was trained for 3 epochs on over $8,000$ multi-turn, long-context trajectories from our Skywork-SWE dataset, with a peak learning rate of 5e-5. We evaluate our Skywork-SWE-32B model with the OpenHands agent framework with up to 100 interaction rounds under two commonly used inference settings:

- **Skywork-SWE-32B** uses a standard inference strategy with a single rollout (N=1), which is also the recommend setting for SWE-Bench.
- **Skywork-SWE-32B (+ TTS)** employs test-time scaling (TTS) by generating N=8 independent rollouts per instance. The final output is selected from the trajectory with the highest score, as evaluated by the OpenHands critic model (Wang, 2025).

We compare our Skywork-SWE-32B model on SWE-Bench Verified against a broad range of advanced code

Table 3: Model performance across different approaches on SWE-Bench Verified. "TTS" denotes test-time scaling. "–" indicates that the exact parameter count of proprietary LLM is not publicly available. Rows corresponding to both Qwen-2.5-Coder-32B-Instruct and OpenHands are highlighted in a light cyan background. ⋆ Results are based on our own evaluation; all other results are taken from previously published work.

Approach	#Params	Framework	Model	Resolve Rate (%)
Proprietary Models				
OpenHands + Gemini-2.0-Flash (Google DeepMind, 2024)⋆	–	OpenHands	Gemini-2.0-Flash	20.0
OpenHands + OpenAI-GPT-4.1-mini (OpenAI, 2024)	–	OpenHands	OpenAI-GPT-4.1-mini	23.6
OpenHands + Qwen-2.5-Max (Qwen, 2024b)⋆	–	OpenHands	Qwen-2.5-Max	31.4
OpenAI-GPT-4o (OpenAI, 2024a)	–	Internal pipeline	GPT-4o	33.2
AutoCodeRover (Zhang et al., 2024)	–	AutoCodeRover	GPT-4o (@May 13, 2024)	38.4
Agentless-1.5 (GPT-4o) (Xia et al., 2024)	–	Agentless	GPT-4o (@May 13, 2024)	38.8
Moatless Tools (Orwall, 2024)	–	Moatless Tools	Claude-3-5-Sonnet-20241022	39.0
OpenHands + Claude-3-5-Haiku (Anthropic, 2024b)	–	OpenHands	Claude-3-5-Haiku	40.6
OpenAI-o1-preview (OpenAI, 2024c)	–	Internal pipeline	OpenAI-o1-preview	41.3
OpenHands + Claude v3.5 (Anthropic, 2024a) ⋆	–	OpenHands	Claude-3-5-Sonnet	46.0
AutoCodeRover v2.0 (Zhang et al., 2024)	–	AutoCodeRover	Claude-3-5-Sonnet-20241022	46.2
OpenAI-o1 (OpenAI, 2024b)	–	Internal pipeline	OpenAI-o1	48.9
Agentless-1.5 (Claude) (Xia et al., 2024)	–	Agentless	Claude-3-5-Sonnet-20241022	50.8
OpenHands + CodeAct v2.1 (Anthropic, 2024a)	–	OpenHands	Claude-3-5-Sonnet-20241022	53.0
OpenHands + Claude v3.7 (Anthropic, 2025)⋆	–	OpenHands	Claude-3-7-Sonnet	**56.0**
Open-source Models				
SWE-Gym-7B (Pan et al., 2024)	7B	OpenHands	Qwen-2.5-Coder-7B-Instruct	10.6
SWE-SynInfer-7B (Ma et al., 2025)	7B	Agentless	Qwen-2.5-Coder-7B-Instruct	18.2
SWE-Dev-7B (Wang et al., 2025)	7B	OpenHands	Qwen-2.5-Coder-7B-Instruct	23.4
SWE-LLaMA-13B (OpenAI, 2024e)	13B	RAG	Code LLaMA-13B	1.2
Devstral (Mistral AI, 2025)	24B	OpenHands	Mistral-Small-3.1-24B-Base-2503	**46.8**
OpenHands + Qwen (Qwen, 2024a)⋆	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	6.4
SWE-Gym-32B (Pan et al., 2024)	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	20.6
SWE-Dev-32B (Wang et al., 2025)	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	36.6
SWE-smith-LM-32B (Yang et al., 2025)	32B	SWE-Agent	Qwen-2.5-Coder-32B-Instruct	40.2
SWE-SynInfer-72B (Ma et al., 2025)	72B	Agentless	Qwen-2.5-72B-Instruct	30.2
SWE-Fixer-72B (Xie et al., 2025)	72B	Agentless	Qwen-2.5-72B-Instruct	32.8
OpenHands + DeepSeek-V3 (DeepSeek-AI, 2024)	671B	OpenHands	DeepSeek-V3	32.4
OpenHands + DeepSeek-R1 (Guo et al., 2025)	671B	OpenHands	DeepSeek-R1	34.0
OpenHands + DeepSeek-V3-0324 (DeepSeek-AI, 2024)	671B	OpenHands	DeepSeek-V3-0324	38.8
Ours				
Skywork-SWE-32B⋆	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	**38.0**
Skywork-SWE-32B (+ TTS)⋆	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	**47.0**

agent models, which can be categorized into *proprietary* and *open-source* LLMs. *Proprietary models* include OpenAI GPT (OpenAI, 2024a), Gemini (Google DeepMind, 2025), Qwen (Qwen, 2024b) and Claude (Anthropic, 2024a), as well as large reasoning models such as o3-mini (OpenAI, 2025) and Doubao (ByteDance Seed, 2025). These models are integrated with corresponding code agent frameworks, including Open-Hands (Wang et al., 2024), Moatless Tools (Orwall, 2024), AutoCodeRover (Zhang et al., 2024), and Agentless (Xia et al., 2024). *Open-source models* encompass instruction-tuned and retrieval-augmented LLMs, such as Code LLaMA-13B (Roziere et al., 2023), Qwen2.5-Coder (Hui et al., 2024) at various models sizes, DeepSeek (Guo et al., 2025) variants, and the Mistal Code model (Mistral AI, 2025). This broad selection enables a comprehensive comparison across both proprietary and open-source code-augmented LLMs.

Table 4: Summary of rollout results from various proprietary LLMs on our Skywork-SWE dataset. Since the dataset was constructed chronologically, some trajectories were collected on earlier subsets. All rollouts were conducted using the OpenHands framework with up tp 100 interaction rounds. "–" denotes the large reasoning models that do not support the `temperature` parameter. The successful trajectories listed were further filtered down to 8,209 instances used to train our Skywork-SWE model (see Sec. 3.4).

Model	Temperature	Successful Trajectories	Resolve Rate (%)
Gemini-2.0-Flash (Google DeepMind, 2024)	0.0	482	5.59
	1.0	87	3.63
Qwen-2.5-Max (Qwen, 2024b)	0.0	717	8.29
	0.5	153	6.38
	1.0	520	8.32
Doubao-1.5-Thinking-Pro (ByteDance Seed, 2025)	–	23	10.60
DeepSeek-V3 (DeepSeek-AI, 2024)	0.0	1071	12.92
	1.0	760	12.18
DeepSeek-V3-0324 (DeepSeek-AI, 2024)	0.0	684	17.49
	1.0	631	17.10
o3-mini (OpenAI, 2025)	–	1908	15.94
GPT-4.1 (OpenAI, 2024)	0.0	142	18.54
Gemini-2.5-Pro (Google DeepMind, 2025)	0.0	1269	20.23
(Cumulative) Total		**8447**	

Benchmark We adopt SWE-bench Verified (OpenAI, 2024e) as our evaluation benchmark and use resolve rate as our evaluation metric. For the code agent framework, we utilize version 0.32.0 of the Open-Hands (Wang et al., 2024) code agent scaffold, configured with a maximum of 100 interaction rounds.

4.2. Experimental Results

We present the main results in Table 3 and Figure 1. Below, we highlight several key observations.

Skywork-SWE-32B achieve SoTA performance among open-source peers. As shown in Fig. 1 (Top), we report pass@1 resolved accuracy without verifiers or multiple rollouts strategies for open-source LLMs built on the OpenHands agent framework. Skywork-SWE-32B achieves 38.0% pass@1 accuracy, surpassing previous open-source Qwen2.5-Coder-32B-based LLMs. Notably, under the same LLM backbone, our model exceeds SWE-smith-LM-32B by absolute 6.8% points, highlighting the effectiveness of our curated training trajectories from the Skywork-SWE dataset. Furthermore, as shown in Table 3, applying test-time scaling (TTS) techniques further improves accuracy of Skywork-SWE to 47.0%, setting a new state-of-the-art among open-source SWE agent models at the 32B scale. These results suggest a clear data scaling law during training and demonstrate the potential of test-time scaling techniques to further enhance performance on SWE tasks.

Code agent frameworks matter more than model size. As shown in Table 3, while larger open-source models yield slight improvements, the performance gains are relatively modest. For instance, Qwen-2.5-72B Team (2024) and DeepSeek-V3-671B DeepSeek-AI (2024) achieves resolve rates of 30.2% and 38.8%, respectively. These results suggest that model size alone is not the dominant factor driving performance in software engineering tasks. Instead, task-specific high-quality training data and well-designed code agent

frameworks play a more pivotal role. Among these code agent frameworks, OpenHands (Wang et al., 2024) stands out by consistently achieving the highest resolve rates across both proprietary and open-source models. As shown in Table 3, over half of the evaluated approaches adopt OpenHands, underscoring its effectiveness and widespread adoption for complex SWE tasks. In light of these advantages, we also adopt OpenHands as the framework for tackling SWE tasks.

Data scaling laws for SWE tasks. Fig. 1 (Top) illustrates the resolve rate of Skywork-SWE-32B on SWE-bench Verified as a function of the number of training trajectories N_{traj}. The performance exhibits a clear log-linear trend, consistently improving with increasing data volume (Kaplan et al., 2020a, Hoffmann et al., 2022b).We curate high-quality, multi-turn, long-context training trajectories, each ending with a final patch that successfully passes all tests within its corresponding runtime environment. Several key data points illustrate the practical benefits of data scaling: Skywork-SWE-32B outperforms SWE-Dev-32B at $N_{\text{traj}} = 2000$, OpenHands-LM-32B-v0.1 at 6000, and SWE-Agent-LM-32B at 8000. These results highlight that scaling up high-quality training data can match or even surpass the performance gains achieved through more complex agent designs in SWE tasks.
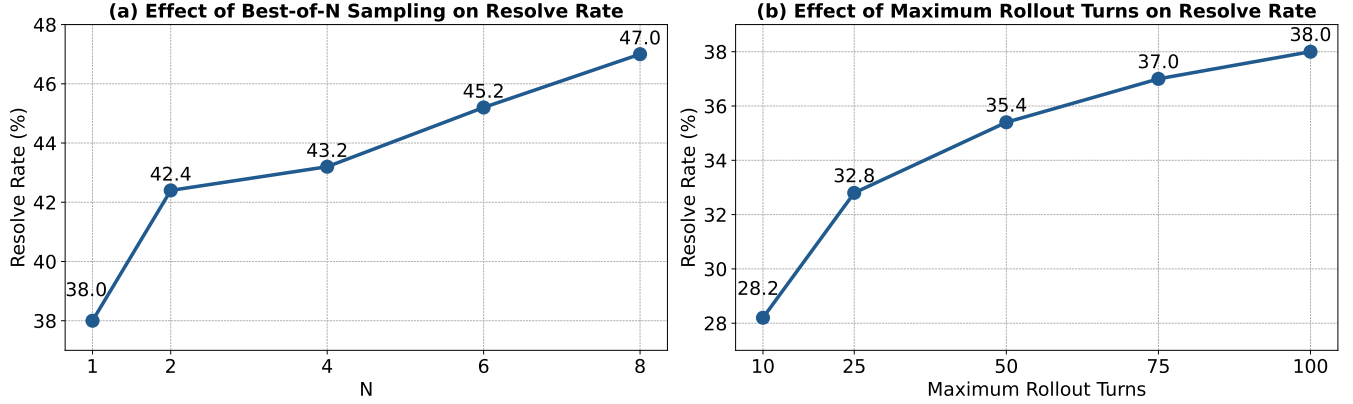
4.3. Experimental Analysis

In this section, we present detailed summary of the rollout results from various proprietary LLMs on our Skywork-SWE dataset. We then examine the impact of two test-time scaling (TTS) evaluation strategies on model performance: Best-of N sampling and the maximum number of rollout rounds.

Data collection efficiency for SWE tasks is low. Table 4 presents the rollout results of various models on our Skywork-SWE dataset. Since the dataset was constructed chronologically, some trajectories were collected from earlier subsets. We observe that even the most advanced proprietary LLMs achieve limited success on Skywork-SWE. Gemini-2.5-Pro yields the highest resolve rate at 20.23%, followed by GPT-4.1 (18.54%) and o3-mini (15.94%). Other models perform even lower, with DeepSeek-V3 and Qwen-2.5-Max achieving 12.92% and 8.29% respectively with `temperature=0` under deterministic decoding. The low performance can be attributed to the extensive diversity of GitHub repository and the large number of unit tests (see Table 2). Overall, executing the SWE tasks is challenging, and the efficiency of data collection is relatively low.

Best-of-N sampling. The effect of varying number of N in Best-of-N sampling on the resolve rate is depicted in Figure 9(a). A clear improvement in the resolve rate is observed as the number of independent rollouts N increases. Specifically, as N is raised to 2, 4, 6, and 8, the resolve rate consistently improves, ultimately reaching 47.0%. This trend indicates that additional rollouts help mitigate output variability, and scaling compute at inference time can significantly enhance model performance, thereby unlocking the model's inherent reasoning capabilities (Brown et al., 2024).

Maximum rollout turns. We evaluate the effect of varying the maximum number of rollout turns on the performance of Skywork-SWE-32B on SWE-bench Verified. As shown in Fig. 9(b), the resolve rate increases from 28.2% at 10 turns to 38.0% at 100 turns. The improvement is most notable in the early stages—from

Figure 9: Resolve rate (%) of Skywork-SWE-32B under different test-time scaling strategies. (a) Effect of Best-of-N Sampling with N ranging from 1 to 8. (b) Effect of maximum rollout turns varying from 10 to 100.

10 to 25 turns yields a 4.6 percentage point gain—while later increases bring smaller benefits, such as a 1.0 point gain from 75 to 100 turns. SWE tasks typically require multiple rounds of interaction to resolve issues in their corresponding GitHub repositories. Therefore, LLMs benefit significantly from extended iteration scaling when the budget for test-time scaling increases with additional interaction rounds.

5. Discussions

Throughout the development of Skywork-SWE, we encountered several engineering challenges and setbacks. Software engineering is inherently complex and labor-intensive, particularly when it comes to tasks such as collecting GitHub repositories, configuring runtime environments, integrating code agent frameworks, and training the agent model. From our development experience, we distill a set of practical insights aimed at advancing LLM-driven software engineering in both academic research and industrial applications.

Data Leakage Issues in Collecting GitHub Repositories. SWE-bench Verified (OpenAI, 2024e) includes 500 instances sourced from pull requests across 12 popular Python GitHub repositories. When curating new SWE datasets from the official PyPI (Python Software Foundation, 2023) collections, it is essential to exclude repositories already included in SWE-bench Verified to prevent potential data leakage. Additionally, training and testing on pull request instances from the same repository can lead to inflated performance. This may occur due to partial contamination at the codebase level or greater similarity between pull requests compared to those from different repositories.

Runtime Environment Configuration. Each instance in an SWE task requires the corresponding runtime environment to verify whether the generated patch passes the unit tests. We adopt a default configuration (see Sec. 3.2), which has been manually verified for reasonable coverage, to filter instances with valid environments. However, this approach inevitably results in significant data loss, as it is impossible to configure all the correct environments for different pull request instances across diverse repositories using a single unified configuration command. Considering the limited number of commonly used GitHub repositories, this high-quality data source warrants a more refined and efficient approach to environment setup. Looking ahead, developing agents capable of automatically setting up environments for the complex and diverse testing setups in real-world repositories represents a promising yet challenging direction for future work.

Runtime Environment Reuse. During agent trajectory generation (see Sec. 3.3), runtime Docker images for each task instance must be built locally to support rollout and validation. This process is highly disk-intensive; the complete set of instance-level images for the 500 SWE-bench Verified tasks requires approximately $1,000$ GB of storage. Due to limited disk capacity, images are deleted in real time after rollout and must be rebuilt for subsequent validation, resulting in redundant overhead. To address this, we divide the instances into mini-batches instead of completing rollout for all instances before validation. For each batch, we perform rollout, then validation, and finally delete the corresponding images to free up disk space. This improved pipeline reuses the built images for both rollout and validation, significantly reducing redundant Docker operations when storage is limited.

Code Agent Framework. In our work, we use version 0.32.0 of the latest OpenHands (Wang et al., 2024) code agent framework. We empirically observe that different versions of OpenHands exhibit variations in system prompts and execution pipelines, which can lead to substantial performance differences. Therefore, we recommend using the latest version of the OpenHands framework to evaluate SWE-bench. It is important to note that switching between OpenHands versions requires updating the corresponding SWE-bench code branch and Docker Hub namespace to ensure compatibility.

Training the Agent Model. Our Skywork-SWE-32B model currently supports a context length of up to $32,768$ tokens. However, we have observed that when the number of interaction rounds exceeds 50, the resulting training trajectories may surpass the 32K-token limit. To accommodate such long sequences, sequence parallelism (Li et al., 2021) is required in the LLM training framework when extending the model context length from 32K to 128K tokens. In future work, we plan to explore LLM training frameworks that support sequence parallelism, such as VeRL (Sheng et al., 2024) and 360-LLaMA-Factory (Haosheng Zou and Zhang, 2024), to enable multi-turn supervised fine-tuning with inputs up to 128K tokens.

6. Conclusion and Future Directions

In this report, we present an automated, execution-aware data curation pipeline used to construct our SWE dataset, comprising $10,169$ execution-validated GitHub issue–fix instances from $2,531$ repositories. Leveraging this dataset, we reveal a clear log–linear data-scaling trend on SWE-bench Verified benchmark. Our Skywork-SWE model achieves 38.0% pass@1 accuracy on SWE-bench Verified—without verifiers or multiple rollouts—setting a new state-of-the-art among Qwen2.5-Coder-32B-based LLMs within the OpenHands agent framework. The integration of test-time scaling techniques boosts performance to 47.0% accuracy, significantly outperforming the previous SOTA results for sub-32B parameter models. These results underscore that high-quality, execution-grounded data remain the primary bottleneck for SWE code agents, and that systematic data expansion can substantially close the gap with proprietary LLMs. We distill practical guidelines to advance LLM-driven software engineering and release the Skywork-SWE-32B checkpoint to accelerate future research.

We outline two promising directions for future work. First, existing benchmarks like SWE-bench focus almost exclusively on Python, limiting their ability to evaluate LLMs in broader software development contexts. Expanding evaluation to multiple programming languages is essential for a more comprehensive assessment of software engineering capabilities, as demonstrated by initiatives like Multi-SWE-Bench (Zan et al., 2025). Second, the SWE task executes and validates unit tests in the runtime environment to provide accurately verified rewards. This setup paves the way for exploring online reinforcement learning methods in the recent LLM community.

7. Acknowledgment

References

Anthropic. Claude 3.5 sonnet announcement. https://www.anthropic.com/news/claude-3-5-sonnet, 2024a. Accessed: 2025-06-04.

Anthropic. Claude haiku 3.5. https://www.anthropic.com/claude/haiku, 2024b. Accessed: 2025-06-03.

Anthropic. Claude 3.7 sonnet and claude code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025. Accessed: 2025-06-04.

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.

Ibragim Badertdinov, Maria Trofimova, Yuri Anapolskiy, Sergey Abramov, Karina Zainullina, Alexander Golubev, Sergey Polezhaev, Daria Litvintseva, Simon Karasik, Filipp Fisin, Sergey Skvortsov, Maxim Nekrashevich, Anton Shevtsov, and Boris Yangel. Scaling data collection for training software engineering agents. *Nebius blog*, 2024.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

ByteDance Seed. Doubao-1.5-pro. https://seed.bytedance.com/zh/doubao_1_5_pro, 2025. Accessed: 2025-06-04.

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

Chi In Chang, Wan Chong Choi, and Iek Chong Choi. A systematic literature review of the opportunities and advantages for aigc (openai chatgpt, copilot, codex) in programming course. In *Proceedings of the 2024 7th International Conference on Big Data and Education*, pages 29–35, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

DeepSeek AI. Deepseek coder: Let the code write itself. https://deepseekcoder.github.io/, 2024. Accessed: 2025-05-13.

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

Google DeepMind. Gemini 2.0 flash. https://deepmind.google/technologies/gemini, 2024. Accessed: 2025-06-04.

Google DeepMind. Gemini 2.5 pro. https://deepmind.google/models/gemini/pro/, 2025. Accessed: 2025-06-04.

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

Shousheng Jia Haosheng Zou, Xiaowei Lv and Xiangzheng Zhang. 360-llama-factory, 2024. URL https://github.com/Qihoo360/360-LLaMA-Factory.

Jordan Hoffmann, Sebastian Biette, Jacopo de Aiken, James Lei, Zi Lin Chen, Jeff Clune, Li Zhang, Jose Suarez, Frank M. Hutter, Greg Brockman, Jack W. Rae, Lei Chinchilla, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022a. URL https://arxiv.org/abs/2203.15556.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, and et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022b.

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

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 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.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

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

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020a.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Scott Gray, Alec Radford, Dario Amodei, and Ilya Sutskever. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020b. URL https://arxiv.org/abs/2001.08361.

Rémi Leblond, Felix Gimeno, Florent Altché, Alaa Saade, Anton Ruddock, Corentin Tallec, George Powell, Jean-Bastien Grill, Maciej Mikuła, Matthias Lochbrunner, et al. Alphacode 2 technical report. Technical report, DeepMind, December 2023. Accessed: 2025-01-14.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

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

Mistral AI. Devstral: Introducing the best open-source model for coding agents. `https://mistral.ai/news/devstral`, 2025. Accessed: 2025-06-03.

OpenAI. Introducing gpt-4.1 in the api. `https://openai.com/index/gpt-4-1/`, 2024. Accessed: 2025-06-04.

OpenAI. Hello gpt-4o, 2024a. URL `https://openai.com/index/hello-gpt-4o/`. Accessed: 2025-05-30.

OpenAI. Introducing openai o1, 2024b. URL `https://openai.com/o1/`. Accessed: 2025-05-30.

OpenAI. Introducing openai o1 preview, 2024c. URL `https://openai.com/index/introducing-openai-o1-preview/`. Accessed: 2025-05-30.

OpenAI. Learning to reason with llms. `https://openai.com/index/learning-to-reason-with-llms/`, September 2024d. Accessed: 2025-01-14.

OpenAI. Swe-bench verified: A human-validated subset for ai model evaluation. `https://openai.com/index/introducing-swe-bench-verified`, 2024e. Accessed: 2025-05-13.

OpenAI. Openai o3-mini. `https://openai.com/index/openai-o3-mini/`, 2025. Accessed: 2025-06-04.

OpenAI. Openai o3 and o4-mini system card. `https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf`, April 2025. Accessed: 2025-05-13.

Arvid Orwall. Moatless tools: Practical large-codebase editing with language models. `https://github.com/aorwall/moatless-tools`, 2024. Commit <hash> accessed May 2025.

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

Python Software Foundation. The python package index (pypi), 2023. URL `https://pypi.org/`.

Qwen. Qwen2.5-coder series: Powerful, diverse, practical. `https://qwenlm.github.io/blog/qwen2.5-max/`, 2024a. Accessed: 2025-06-04.

Qwen. Qwen2.5-max: Maximizing reasoning and tool use abilities. `https://qwenlm.github.io/blog/qwen2.5-max/`, 2024b. Accessed: 2025-06-04.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

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

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

Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

torchtune maintainers and contributors. torchtune: Pytorch's finetuning library, April 2024. URL https://github.com/pytorch/torchtune.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. Swe-dev: Building software engineering agents with training and inference scaling. https://ubecwang.notion.site/1bc32cf963e080b2a01df2895f66021f?v=1bc32cf963e0810ca07e000c86c4c1e1, 2025. Accessed: 2025-05-13.

Xingyao Wang. Sota on swe-bench verified with inference-time scaling and critic model, 2025. URL https://www.all-hands.dev/blog/sota-on-swe-bench-verified-with-inference-time-scaling-and-critic-model.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2024.

Lilian Weng. Llm-powered autonomous agents. *lilianweng.github.io*, Jun 2023. URL https://lilianweng.github.io/posts/2023-06-23-agent/.

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

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

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. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Leret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

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*, ISSTA 2024, page 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

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