

## 1 A Related work

2 **Benchmarks for software engineering** With the progress of LLM capabilities, there has been  
3 a growing need for benchmarks to evaluating their coding performance. While HumanEval [Chen  
4 et al., 2021], MBPP [Austin et al., 2021], and APPS [Hendrycks et al., 2021] have proven valuable  
5 for standalone function-level evaluation, their utility has largely saturated for state-of-the-art models.  
6 Consequently, LiveCodeBench [Jain et al., 2024] has emerged, which is continuously updated to  
7 reflect the evolving capabilities of LLMs. However, these benchmarks often differ significantly from  
8 how LLMs are applied in real-world development workflows, which typically involve repository-level  
9 context and interaction. SWE-bench [Jimenez et al., 2024] introduced execution-based, repository-  
10 level tasks sourced from real-world repositories, offering a more realistic evaluation paradigm. A  
11 challenge with SWE-bench is that many new LLMs were released after its construction, raising  
12 concerns about its continued relevance and potential contamination. Several follow-up works, such  
13 as SWE-bench+ [Aleithan et al., 2024] and Multi-SWE-bench [Zan et al., 2025], address specific  
14 shortcomings like post-cutoff filtering and multilingual annotation. Nevertheless, all these extensions  
15 fundamentally rely on a manually curated construction process. This manual element complicates  
16 timely updates and makes it difficult to keep the benchmark consistently aligned with the rapidly  
17 advancing capabilities of LLMs. In contrast, SWE-rebench automates the entire process of task  
18 creation from real-world repositories at scale. This automation enables a new class of dataset and  
19 benchmark: one that is task-rich, diverse, regularly updated, and suitable for both training and  
20 evaluation without the bottlenecks of manual curation.

21 **Training datasets for code generation** Most prior work on collecting data to improve LLMs  
22 for code has focused on supervised fine-tuning [Muennighoff et al., 2024, Luo et al., 2023, Liu  
23 et al., 2024], often by leveraging samples from more powerful proprietary models. On the other  
24 hand, the remarkable success of large-scale reinforcement learning (RL) in math reasoning tasks  
25 [DeepSeek-AI, 2025, Seed et al., 2025, OpenThoughts, 2025] implies that it should also excel in  
26 software engineering, as both domains involve rich, multi-step workflows with potentially verifiable  
27 outcomes. However, leveraging RL effectively in this setting demands gathering a large collection  
28 of real-world, interactive SWE tasks which is challenging, as each task requires the construction  
29 of a suitable and reliable execution environment. Some works attempt to gather such tasks by  
30 artificially injecting bugs into code [Yang et al., 2025] or generating synthetic tests [Xie et al., 2025].  
31 While these approaches can generate data at scale, they may not fully capture the nuances and  
32 complexities of real-world software issues. The original SWE-bench dataset and its extensions for  
33 training [Pan et al., 2024b] provide valuable real-world tasks, but still suffer from limited diversity  
34 and scale due to their reliance on manual construction. In contrast, SWE-rebench reflects real-world  
35 development processes and preserves high task quality through its grounding in real repositories,  
36 while achieving significantly greater scale and diversity via its automated construction pipeline. This  
37 makes it particularly well-suited for training agents through an automated construction pipeline.

## 38 B Automated dependency installation

39 As a reminder, the goal of automated dependency installation is to generate correct environment setup  
40 instructions for each task instance without using manual curation. The process is entirely LLM-driven  
41 and consists of the following sub-stages:

- 42 1. **Identifying relevant files:** An LLM is asked to analyze the list of repository files and select  
43 those most likely to contain installation instructions, such as `README.md`, `setup.py`, or  
44 `requirements.txt`.
- 45 2. **Extracting installation recipe:** Based on the content of files found in the previous stage,  
46 the model generates a structured JSON recipe specifying how to install dependencies and  
47 run tests for the repository.
- 48 3. **Updating the recipe (if needed):** If the recipe fails, another LLM call is made to analyze  
49 the failure and refine the recipe.

50 In the following subsections, we show the exact prompts used at each of these stages, along with  
51 examples of LLM-generated outputs. These materials are intended to provide transparency and  
52 facilitate reproducibility of our environment setup pipeline.

## 53 B.1 Identifying installation-related files

54 The automated generation of installation recipes, a key part of our agentless approach described in  
55 Section 2.2, begins with identifying repository files that are likely to contain setup instructions. To  
56 achieve this, an LLM is provided with a list of all files present in the repository. The LLM's task  
57 is to analyze this list and return a curated subset of file paths deemed most relevant for extracting  
58 installation, dependency, and testing information. The prompt below details the instructions given  
59 to the LLM for this file identification step. The output of this step (a JSON array of file paths)  
60 subsequently serves as input for the next LLM call, which extracts the actual installation commands.

Listing 1: LLM prompt for identifying installation-related files

```
61
62
63 You are tasked with identifying files that likely contain installation
64   instructions for a GitHub repository.
65
66 Repository: {{ repo_name }}
67
68 Below is a list of files in the repository that may be helpful for
69   understanding the installation and setup process:
70
71 {{ repo_name }}
72
73 Please analyze this list and identify the files that are most likely to
74   contain information about:
75   * Installation instructions
76   * Dependencies or requirements
77   * Setup procedures
78   * Development environment configuration
79   * Testing setup
80
81 Think step by step:
82   * Identify README files, as they often contain installation instructions.
83   * Look for setup.py, pyproject.toml, requirements.txt, environment.yml.
84   * Consider files in directories like \texttt{docs} that might contain
85     installation guides.
86   * Look for test configuration files that might help understand how to run
87     tests.
88   * Consider \textbf{only} files from the list provided above.
89   * Prioritize files in the repository root (top-level directory).
90   * Only include files from subdirectories if they are clearly relevant to
91     installation or setup.
92
93 Return only a JSON array containing the paths to the most relevant
94   files for installation and setup. Include only files that are directly
95   relevant to the tasks above. Sort the files from most to least relevant
96   and limit your response to no more than 10 files, preferring fewer
97   files that are truly essential.
98
99 For example:
100 [
101   "README.md",
102   "setup.py",
103   "requirements.txt"
104 ]
```

## 106 B.2 Extracting installation recipe

107 Following the identification of relevant files, the next step in our automated pipeline (Section 2.2)  
108 is to extract a concrete installation recipe. For this, the LLM receives the concatenated content of

109 the previously selected relevant files. The primary task of the LLM is to analyze this content and  
110 synthesize a structured JSON object, *installation recipe*, which encapsulates all necessary commands  
111 and configurations to set up the project environment, install dependencies, and execute tests. The  
112 prompt presented below guides the LLM through this extraction process, emphasizing the expected  
113 JSON structure and providing context on how the generated recipe will be utilized by downstream  
114 automation scripts.

Listing 2: LLM prompt for extracting installation recipe

```
115 You are tasked with extracting detailed installation instructions from the
116 following repository files. Repository: {{ repo_name }}
117
118 Please analyze the content of these files and extract comprehensive
119 installation instructions: {{ rendered[:100000] }}
120
121 First, think step by step. After your analysis, return your findings in the
122 following JSON format:
123
124 {
125   "python": "3.9",
126   "packages": "requirements.txt",
127   "install": "pip install -e .[dev]",
128   "test_cmd": "pytest --no-header -rA --tb=line --color=no -p no:
129     cacheprovider -W ignore::DeprecationWarning",
130   "pre_install": ["apt-get update", "apt-get install -y gcc"],
131   "reqs_path": ["requirements/base.txt"],
132   "env_yaml_path": ["environment.yml"],
133   "pip_packages": ["numpy>=1.16.0", "pandas>=1.0.0"]
134 }
135
136
137 Here is how this JSON will be used:
138
139 ```bash
140 git clone <repo_url> repo
141 cd repo
142 git checkout <base_sha>
143 bash <pre_install>
144 conda create -n <repo> python=<python> -y
145 conda activate <repo>
146 if <packages> == requirements.txt; then
147     for path in <reqs_path>:
148         pip install -r $path
149 elif <packages> == environment.yml; then
150     for path in <env_yaml_path>:
151         conda env update -f $path
152 else:
153     pip install <packages>
154     pip install <pip_packages>
155     bash <install>
156     bash <test_cmd>
157 ```
158
159
160 **IMPORTANT:**
161 * For the "install" field, always use local install commands like pip
162   install -e .[dev]
163 * Do NOT include packages in pip_packages that will be installed by pip
164   install -e .
165 * Include only explicitly needed packages in pip_packages.
```

```

166 * reqs_path and env_yaml_path must match filenames from the provided files
167   (e.g., [File: filename]).
168 * If "packages" is requirements.txt, you must provide at least one
169   reqs_path.
170 * Add relevant test frameworks to pip_packages (e.g., pytest, nose).
171 * Use -y in all conda commands.
172 * Prefer direct and specific pytest commands over general wrappers.
173 * Avoid test commands with placeholders like {test_name}.
174 * If a Makefile runs tests, extract the actual test command (e.g., pytest)
175   .
176
177 You must ensure the final JSON includes required fields (, install,
178   test_cmd), and optionally packages, pre_install, reqs_path,
179   env_yaml_path, pip_packages if relevant.
180
181 Base your reasoning on all provided files and return it followed by the
182   JSON output in a Markdown code block (without comments).
183

```

### 184 B.3 Example of LLM-generated installation recipe

185 Below is an illustrative example of an output from the installation recipe extraction stage that includes  
 186 both the reasoning and the final JSON.

Listing 3: LLM-generated installation recipe

```

187
188 1. Python version. The setup.py file specifies that the project requires
189   Python 3.6 or greater. The README.rst and setup.py files also indicate
190   that the project supports Python 3.6, 3.7, 3.8, 3.9, and 3.10. For a
191   safe middle-ground, Python 3.9 is chosen.
192
193 2. Installation Method. The README.rst and DEVELOPMENT.md files both
194   mention using pip to install the project. Specifically, the development
195   installation command is pip install -e .[dev]
196
197 3. Dependencies. The setup.py file lists flake8 >= 3.0.0 and attrs >=
198   19.2.0 as dependencies. The DEVELOPMENT.md file also mentions
199   additional development dependencies: coverage, hypothesis, hypothesis
200   >= 0.2, and pre-commit. These are included in the pip_packages field.
201
202 4. Test command. The README.rst and DEVELOPMENT.md files both mention
203   running tests with coverage run tests/test_bugbear.py. However, for
204   simplicity and to follow the guidelines, the test command is simplified
205   to pytest tests/test_bugbear.py.
206
207 5. Pre-install steps. No system-level pre-install steps are mentioned in
208   the provided files.
209
210 6. Additional packages. The pytest package is needed for running tests, and
211   it is not explicitly listed in the project's dependencies. Therefore,
212   it is added to the pip_packages field.
213
214 {
215   "python": "3.9",
216   "install": "pip install -e .[dev]",
217   "test_cmd": "pytest tests/test_bugbear.py",
218   "pip_packages": [
219     "pytest",
220     "coverage",
221     "hypothesis",

```

```

222     "hypothesmith">=0.2",
223     "pre-commit"
224 ]
225 }
226

```

## 227 B.4 Updating installation recipe

228 As described in Section 2.2, our agentless installation recipe generation includes an iterative refine-  
 229 ment step. If the initial attempt to set up an environment using an LLM-generated recipe fails, we  
 230 trigger a correction mechanism. This mechanism involves a subsequent call to the LLM, providing it  
 231 with the original (failed) installation recipe, the error logs produced during the failed setup or test  
 232 execution, as well as the contents of relevant files. The LLM's objective is then to diagnose the errors  
 233 based on the logs and propose modifications to the recipe to resolve these issues. The prompt detailed  
 234 below instructs the LLM on how to perform this error analysis and generate an updated, corrected  
 235 installation recipe in the same structured JSON format.

Listing 4: LLM prompt for updating installation recipe

```

236 You are an expert in fixing software installation and testing issues.
237 Analyze the installation logs provided and update the installation
238 configuration to fix any errors.
239
240 Install config fields:
241
242 {
243   "python": "3.9",
244   "packages": "requirements.txt",
245   "install": "pip install -e .[dev]",
246   "test_cmd": "pytest --no-header -rA --tb=line --color=no -p no:
247     cacheprovider -W ignore::DeprecationWarning",
248   "pre_install": ["apt-get update", "apt-get install -y gcc"],
249   "reqs_path": ["requirements/base.txt"],
250   "env_yaml_path": ["environment.yml"],
251   "pip_packages": ["numpy>=1.16.0", "pandas>=1.0.0"]
252 }
253
254 Current installation configuration:
255 {{ install_config }}
256
257 Error logs from installation/testing (last relevant lines): {{ cut_logs }}
258
259 Your task:
260 * Identify the root causes of the errors in the logs.
261 * Modify the installation configuration to address these issues.
262 * You might need to:
263   - Add missing dependencies
264   - Fix command syntax errors
265   - Change installation order
266   - Add environment variables
267   - Modify test commands
268
269 First, provide brief reasoning (<100 words) about what's causing the errors
270 and your planned fixes.
271
272 Then return the complete updated installation configuration as a valid JSON
273 object.
274
275

```

## C Evaluation of automated installation recipe generation

To assess the efficacy of our automated installation recipe generation and to select effective prompts, we conducted a validation study. We utilized a curated set of 18 task instances, each from a distinct repository within the original SWE-bench dataset. These instances were chosen because their environment installation instructions were originally manually crafted, providing a reliable baseline. For each instance, we attempted to automatically generate an installation recipe, set up the corresponding environment, and execute its tests.

The performance of the agentless LLM-based recipe generation (with varying numbers of candidate generations) was compared against an interactive agent designed for the same purpose. The results are summarized in Table 3. Some unsuccessful configuration attempts are due to limitations in our automated log parsing during verification, especially for repositories with custom test frameworks such as Django. Our generic parsers (e.g., for pytest) may fail to extract test results from highly customised test outputs. Nevertheless, this small-scale validation served as a sanity check and informed our selection of prompts and the number of candidate generations for the agentless approach in the main pipeline. The list of specific SWE-bench instances used for this validation can be found in Appendix C.1.

Table 3: Comparison of installation method success rates on 18 SWE-bench tasks.

Approach	Candidates/Trials	Successfully Configured
Agentless	1	6 out of 18
Agentless	3	8 out of 18
Agentless	10	9 out of 18
Agent-based	1	8 out of 18

### C.1 List of instances to validate automatic installation instruction extraction

- astropy\_\_astropy-12907
- django\_\_django-15315
- marshmallow-code\_\_marshmallow-1164
- matplotlib\_\_matplotlib-20826
- mwaskom\_\_seaborn-3069
- pallets\_\_flask-5014
- psf\_\_requests-2317
- pvlib\_\_pvlib-python-1160
- pydata\_\_xarray-6744
- pydicom\_\_pydicom-996
- pylint-dev\_\_astroid-1741
- pylint-dev\_\_pylint-4604
- pytest-dev\_\_pytest-7982
- pyvista\_\_pyvista-4853
- scikit-learn\_\_scikit-learn-13142
- sphinx-doc\_\_sphinx-8721
- sqlfluff\_\_sqlfluff-4764
- sympy\_\_sympy-22080

## D Permissive licenses included in data collection

Our data collection process exclusively targeted repositories under permissive open-source licenses. We identified repositories matching common permissive licenses using their SPDX identifiers where available. For repositories with licenses not matching this predefined list, or those with custom license text (categorized as “Other”), a manual review was conducted to confirm that their terms permit the use cases relevant to our work. The primary permissive license types included were:

- MIT
- Apache-2.0
- BSD-4-Clause
- BSD-3-Clause
- BSD-2-Clause
- ISC
- CC0-1.0
- ZPL-2.1
- Other (manually verified permissive licenses)

## E Example of a task instance with annotations

Below is an example of a single task instance from the SWE-rebench dataset, illustrating the structure and some of the key annotations collected or generated by our pipeline. For brevity, some lengthy fields like full file contents or complete dependency lists were truncated or summarized. The instance is presented in the form of a Python dictionary.

Listing 5: SWE-rebench task instance contents example

```
{
  'instance_id': 'tarohi24__typedflow-68',
  'repo': 'tarohi24/typedflow',
  'base_commit': '2127e74314d2b97d596cfc12ed8fb257bb688d6f',
  'version': '1.0',
  'created_at': '2019-12-10 15:26:34',
  'problem_statement': "The new syntax doesn't work\nIt doesn't accept args
    in the correct way. For instance, life of cache tables are never
    incremented.",
  'patch': """diff --git a/typedflow/nodes/base.py b/typedflow/nodes/base.
    py
index ece0895..b9853f9 100644
--- a/typedflow/nodes/base.py
+++ b/typedflow/nodes/base.py
@@ -113,7 +113,8 @@ class ConsumerNode:
     None
    """
    assert len(self.precs) == 0, 'Some arguments have been already set'
- self.precs: Dict[str, ProviderNode] = args
+ for name, prec in args.items():
+ self.set_upstream_node(name, prec)
    return self
    """
  'test_patch': """diff --git a/typedflow/tests/flow/test_flow.py b/
    typedflow/tests/flow/test_flow.py
index aa31917..7682475 100644
--- a/typedflow/tests/flow/test_flow.py
```

```

361 +++ b/typedflow/tests/flow/test_flow.py
362 @@ -209,3 +209,4 @@ def
363     test_declare_inputs_when_definition_with_multiple_args():
364         node_dump = DumpNode(dump)({'a': node_task})
365         flow = Flow([node_dump, ])
366         flow.typecheck()
367 + assert node_task.cache_table.life == 1
368 """
369     'meta': {
370         'commit_name': 'head_commit',
371         'num_modified_files': 1,
372         'llm_score': {'issue_text_score': 3, 'test_score': 1, 'difficulty_score': 1}
373     },
374     # Other meta fields like 'failed_lite_validators', 'has_test_patch', 'is_lite'
375 },
376     'install_config': {
377         'python': '3.8',
378         'packages': 'requirements.txt',
379         'reqs_path': ['requirements.txt'],
380         'install': 'pip install -e .',
381         'pip_packages': ['pytest', 'pytest-runner', ...], // Truncated for
382             brevity
383         'pre_install': ['apt-get update', 'apt-get install -y gcc build-essential ...'], // Truncated
384         'test_cmd': 'pytest --no-header -rA --tb=line --color=no -p no:cacheprovider ...' # Truncated
385         # Other install_config fields like 'env_vars', 'env_yaml_path', 'log_parser'
386     },
387     'FAIL_TO_PASS': ['typedflow/tests/flow/test_flow.py::test_declare_inputs_when_definition_with_multiple_args'],
388     # Other test outcome fields like 'FAIL_TO_FAIL', 'PASS_TO_PASS', 'PASS_TO_FAIL'
389     'license_name': 'MIT License'
390     # Fields like 'requirements', 'environment', 'hints_text', 'total_len' also exist.
391 }
392

```

400 This example showcases the core components of a task: the problem description, the code changes  
 401 (patch and `test_patch`), and associated metadata including LLM-generated quality scores and  
 402 installation configurations. The full dataset contains more extensive information about each instance.

## 403 **F Comparison of models for automatic task quality assessment**

404 As mentioned in Section 2.4, we fine-tuned a Qwen 2.5-72B-Instruct model using human annotations  
 405 from SWE-bench Verified to predict three quality assessment labels: **Test Patch Correctness**, **Task Complexity**, and **Issue Clarity**. Table 4 provides a detailed classification quality report, comparing  
 406 the performance of our fine-tuned model against the vanilla Qwen 2.5-72B-Instruct model on the  
 407 validation set.  
 408

## 409 **G Refinements to the original SWE-bench methodology**

410 Our goal is to create a dataset suitable not just for evaluation, but also for reinforcement learning.  
 411 Ideally, its tasks should remain unsolved only due to the agent’s inherent limitations, not due to faulty  
 412 tests or incorrect problem specifications. This is why we enhanced mechanisms ensuring task validity  
 413 compared to those used to build SWE-bench.



Table 4: Classification report for task label prediction, comparing the vanilla Qwen-2.5-72B-Instruct model with its fine-tuned counterpart.

Label		Instruct			Fine-tuned			Support
		Prec	Rec	F1	Prec	Rec	F1	
Test Patch Correctness	low-score	0.59	0.97	0.73	0.66	0.85	0.74	233
	high-score	0.76	0.12	0.21	0.69	0.42	0.52	180
	accuracy			0.60			0.67	413
	macro avg	0.67	0.55	0.47	0.67	0.64	0.63	413
	weighted avg	0.66	0.60	0.51	0.67	0.67	0.65	413
Task Complexity	low-score	0.90	0.64	0.75	0.90	0.85	0.87	313
	high-score	0.41	0.78	0.54	0.60	0.71	0.65	100
	accuracy			0.68			0.81	413
	macro avg	0.66	0.71	0.64	0.75	0.78	0.76	413
	weighted avg	0.78	0.68	0.70	0.83	0.81	0.82	413
Issue Clarity	low-score	0.83	0.94	0.88	0.82	0.94	0.88	329
	high-score	0.51	0.26	0.35	0.47	0.20	0.28	84
	accuracy			0.80			0.79	413
	macro avg	0.67	0.60	0.61	0.65	0.57	0.58	413
	weighted avg	0.77	0.80	0.77	0.75	0.79	0.76	413

- Patch generation from git history:** In the original SWE-bench, the diff between `base_commit` (where the PR branch starts) and `merge_commit` (the PR is merged into the main branch) forms the solution and test patches. However, intervening merges into the main branch can introduce unrelated changes into this diff, potentially invalidating tasks by including tests for functionality that is external with respect to the PR. As an example, `sympy__sympy-14821` was not included in SWE-bench Verified for this reason. To mitigate this problem, we prioritize diffing `head_commit` (the last commit in the PR branch) against `base_commit` to generate patches, as this isolates changes made directly within the branch. If `head_commit` is unavailable (e.g., due to branch deletion after merge), we fall back to `merge_commit` and record this choice in the task metadata.
- Test directive generation:** SWE-bench generates test directives (commands to run specific tests for verification) from the test patch using regular expressions. This can erroneously include deleted test files, leading to invalid commands. Our refinement filters out deleted files from test directives.
- AttributeError/ImportError checks:** Tasks where tests check for new, not-yet-existing attributes or imports can be problematic, as the agent must guess exact signatures when implementing a solution. While often detectable, such errors can be masked if tests are being run with reduced traceback verbosity (e.g., Pytest’s `tb=no` flag, used in SWE-bench for some repositories). We run tests with full error output (`tb=line`) to accurately identify these cases. This information is recorded in task metadata, allowing for optional filtering, as some instances (e.g., fixing an import error) are valid.
- Dependency pinning:** Python projects often lack explicitly pinned dependency versions. Defaulting to the latest versions during installation can lead to test failures due to package compatibility changes over time. To ensure reproducibility, we freeze and record all dependency versions (using `pip freeze` and `conda env export`) after the first successful environment setup for a task (or a task group sharing the same environment). These pinned versions are then reused for subsequent rebuilds.

## H Filtering criteria for the SWE-rebench benchmark subset

The SWE-rebench benchmark subset, used for evaluating LLM-based agents as described in Section 3, is curated from the larger SWE-rebench dataset by applying a series of specific filtering criteria.

444 These filters are designed to select tasks that are well-defined, of manageable complexity for current  
445 models, and ensure a consistent evaluation environment. The following conditions must be met for a  
446 task instance to be included in the benchmark subset:

- 447 • **Clean test execution:** Test execution logs prior to applying any solution patch must be free  
448 of critical errors such as `AttributeError` or `ImportError`.
- 449 • **Code modification scope:** The number of files modified by the solution patch must be no  
450 more than 3.
- 451 • **Patch size:** The total number of words in the solution patch must not exceed 500.
- 452 • **Problem statement length:** The problem statement (github issue description) must contain  
453 between 16 and 1000 words (inclusive).
- 454 • **Problem statement language:** The problem statement must be in English.
- 455 • **Recency:** The github issue the task is based on must have been created in the year 2025.
- 456 • **Assessed difficulty:** The LLM-assessed difficulty label for the task must be less than 3  
457 (indicating low to moderate complexity).
- 458 • **Test case count:** The number of tests that transition from a failing to a passing state  
459 (fail-to-pass tests) must be 50 or fewer.

## 460 I System prompt for agent runs

Listing 6: System prompt for LLM evaluation runs on SWE-rebench leaderboard tasks

```
461 # SETTING
462
463 You are an autonomous programming agent. Your goal is to resolve the issue
464 given to you.
465 You are given access to a terminal environment with some special tools to
466 make your job easier.
467 You must use the terminal to gain information about the codebase, find or
468 modify the relevant files in order to resolve the issue.
469 In this environment, all standard unix commands (e.g. grep, sed, echo etc.)
470 will be available to you.
471 However, the environment does NOT support interactive session commands
472 that expect user input (e.g. vim), so please do not invoke them, it
473 will result in an error
474 You can however create python scripts and run them, this is very useful to
475 reproduce errors or test something.
476 If some packages are missing, you can install them using an appropriate
477 package manager (e.g. pip, apt, etc.).
478 Do not ask any questions to the environment, it's an automated system that
479 can only execute your commands.
480 When you are satisfied with the changes you made, you should explicitly
481 submit them using a special command. This will terminate your session.
482
483 # SPECIAL TOOLS
484
485 In addition to standard unix commands you can use special tools described
486 below.
487 Please note that some of these commands work with the currently open file,
488 so pay attention to what file is open.
489
490 Usage: create [OPTIONS] FILENAME
491 Creates and opens a new filename with the given name.
492
493 Usage: edit [OPTIONS] LINE_RANGE [REPLACEMENT_TEXT]
494 Replaces lines in LINE_RANGE=<start_line>:<end_line> (inclusive) with the
```

```

496 given text in the currently open or specified file. The REPLACEMENT_TEXT
497 will be used as provided including all whitespaces, so make sure your
498 indentation is correct.
499 To input multiple lines into REPLACEMENT_TEXT, you may use the following
500 syntax:
501 ```
502 edit 1:1 << 'EOF'
503 Line1
504 Line2
505 EOF
506 ```
507 You can also provide the file to edit via `--file` option.
508 ```
509 edit --file path/to/file 1:1 "Your Replacement Text Here"
510 ```
511 Please note that THIS COMMAND REQUIRES PROPER INDENTATION. If you'd like
512 to
513 add the line ' print(x)' you must fully write that out, with all
514 those spaces before the print statement!
515 Options:
516 --file PATH The file to edit. (If not provided, edits the currently open
517 file)
518
519 Usage: goto [OPTIONS] LINE_NUMBER
520 Navigates the current window to a given line in the currently open file.
521
522 Usage: open [OPTIONS] [FILE] [LINE_NUMBER]
523 Opens the file at the given path in the editor. If file is not specified,
524 the last open file will be reopened. If line_number is provided, the
525 current
526 window will move to show that line.
527
528 Usage: replace [OPTIONS] SEARCH REPLACE
529 Replaces a given string with another string in the currently open file.
530 Options:
531 --replace-all Replace all occurrences of the SEARCH text.
532
533 Usage: scroll_down [OPTIONS]
534 Scroll down the window in the currently open file and output its contents
535 .
536
537 Usage: scroll_up [OPTIONS]
538 Scroll up the window in the currently open file and output its contents.
539
540 Usage: search_file [OPTIONS] SEARCH_TERM [FILE]
541 Searches for SEARCH_TERM in file. If FILE is not provided, searches in
542 the currently open file.
543
544 Usage: submit [OPTIONS]
545 Submits your current code and terminates the session.
546
547
548 # ENVIRONMENT RESPONSE
549
550 At the very beginning the environment will provide you with an issue
551 description. In response to every command that you invoke,
552 the environment will give you the output of the command or an error
553 message followed by a shell prompt.
554 The shell prompt will be formatted as follows:

```

```

555  ```
556  (Current directory: <current_dir>, current file: <current_file>) bash-$
557  ```
558  so that you always know what the current directory is and what file is
559  currently open.
560
561  # YOUR RESPONSE
562
563  Your response should consist of two parts: reasoning (arbitrary text) and
564  command (surrounded by triple ticks and a special 'command' keyword).
565  Your response should always include A SINGLE reasoning and A SINGLE
566  command as in the following examples:
567
568  <response example>
569  First I'll start by using ls to see what files are in the current
570  directory. I'll look at all files including hidden ones.
571  ```command
572  ls -a
573  ```
574  </response example>
575
576  <response example>
577  Let's search the file `models.py` for the UserEntity class definition.
578  ```command
579  search_file "class UserEntity" models.py
580  ```
581  </response example>
582
583  Everything you include in the reasoning will be made available to you when
584  generating further commands.
585  If you'd like to issue two command blocks in a single response, PLEASE DO
586  NOT DO THAT! THIS WILL RESULT IN AN ERROR.
587
588  # HANDLING TESTS
589
590  * You can run existing tests to validate the changes you made or make sure
591  you didn't break anything.
592  * If missing packages or some environment misconfiguration is preventing
593  you from running the tests, you can install missing packages or fix the
594  environment.
595  * However UNDER NO CIRCUMSTANCES should you modify existing tests or add
596  new tests to the repository.
597  This will lead to an error in the system that evaluates your performance.
598  Instead, you can just create a temporary script, use it to test
599  changes and remove it before submitting.
600  * If existing tests break because they need to be updated to reflect the
601  changes you made, just ignore it. Evaluation system will not take it
602  into account.
603  * However if existing tests are broken because your fix is incorrect, you
604  should fix your code and make sure all tests pass before submitting the
605  change.
606
607  # USEFUL ADVICE
608
609  * As a first step, it might be a good idea to explore the repository to
610  familiarize yourself with its structure.
611  * You should also come up with a rough plan of how to resolve the issue
612  and put it into your reasoning.

```

```

613 * If the issue description reports some error, create a script to
614     reproduce the error and run it to confirm the error. THIS IS USUALLY A
615     VERY GOOD FIRST STEP!
616 * Edit the source code of the repo to resolve the issue
617 * Rerun your reproduce script and confirm that the error is fixed! THIS IS
618     QUITE IMPORTANT!
619 * Think about edge cases and make sure your fix handles them as well.
620 * Make sure your solution is general enough and not hardcoded to the
621     specific cases reported in the issue description.
622 * It might be a good idea to ensure that existing tests in the repository
623     pass before submitting the change. Otherwise it's easy to break
624     existing functionality.
625
626 # DEMONSTRATION
627
628 Here is a very simple demonstration of how agent can interact with the
629     environment using the provided interface.
630
631 <demonstration><environment>
632 # ISSUE DESCRIPTION
633
634 Here is a script that is supposed to print out first 10 prime numbers, but
635     it seems to have a bug. Can you fix it?
636
637 ```
638 def is_prime(n):
639     if n <= 1:
640         return False
641     for i in range(2, int(n**0.5)):
642         if n % i == 0:
643             return False
644     return True
645
646 primes = []
647 num = 2
648 while len(primes) < 10:
649     if is_prime(num):
650         primes.append(num)
651     num += 1
652
653 print("First 10 prime numbers:", primes)
654 ```
655
656 # ADDITIONAL ADVICE
657
658 Since you are given a git repository, you can use git commands to simplify
659     your work.
660 For example, if you made a mistake and want to revert your changes, you
661     can use `git checkout <file>` to restore the file to its original state
662     .
663 You can also reset all changes in the repository using `git reset --hard`
664     command.
665 Additionally, you can use `git stash` and `git stash pop` to temporarily
666     save your changes and restore them later.
667 Finally, you can see the changes that you've made (which will be submitted
668     when you call `submit`) using `git status` or `git diff` commands.
669 However you don't need to use `git add` or `git commit` before submitting
670     your changes. In fact,

```

```

671 `submit` only submits currently unstaged changes because it uses `git diff
672 ` to get the changes that need to be submitted.
673
674 # CHECKLIST
675
676 Before submitting your solution, please go over the following checklist
677 and make sure you've done everything:
678 - [ ] If an error was reported in the issue description, I have
679 successfully reproduced it.
680 - [ ] If an error was reported in the issue description, I have confirmed
681 that my fix resolves the error.
682 - [ ] I have thought about edge cases and made sure my fix handles them as
683 well.
684 - [ ] I have run existing tests in the repository that might have been
685 affected by the change I made and confirmed that they pass.
686 I want you to list every bullet from this checklist and write your
687 reasoning for why you think you did it or didn't need to.
688
689 Repository has been uploaded and your shell is currently at the repository
690 root. Time to solve the issue!
691
692 (Current directory: /project, current file: none) bash-$
693 </environment>
694 <agent>
695 Let me analyze the issue and fix the prime number generator script step by
696 step.
697
698 First, let's create a Python script to reproduce the reported issue and
699 verify the current behavior.
700
701 ```command
702 create prime_numbers.py
703 ```
704 </agent>
705 <environment>
706 [File: prime_numbers.py (1 lines total)]
707 1:
708
709 (Current directory: /project, current file: none) bash-$
710 </environment>
711 <agent>
712 Now I'll add the provided code to the file to test it and see what output
713 it produces. The issue description mentions there's a bug, but doesn't
714 specify what exactly is wrong. First we need to identify the issue.
715
716 ```command
717 edit 1:1 << 'EOF'
718 def is_prime(n):
719     if n <= 1:
720         return False
721     for i in range(2, int(n**0.5)):
722         if n % i == 0:
723             return False
724     return True
725
726 primes = []
727 num = 2
728 while len(primes) < 10:
729     if is_prime(num):

```

```

730     primes.append(num)
731     num += 1
732
733 print("First 10 prime numbers:", primes)
734 EOF
735 ```
736 </agent>
737 <environment>
738 File updated. Please review the changes and make sure they are correct (
739     correct indentation, no duplicate lines, etc). Edit the file again if
740     necessary.
741 [File: /project/prime_numbers.py (16 lines total)]
742 1:def is_prime(n):
743 2: if n <= 1:
744 3: return False
745 4: for i in range(2, int(n**0.5)):
746 5: if n % i == 0:
747 6: return False
748 7: return True
749 8:
750 9:primes = []
751 10:num = 2
752 11:while len(primes) < 10:
753 12: if is_prime(num):
754 13: primes.append(num)
755 14: num += 1
756 15:
757 16:print("First 10 prime numbers:", primes)
758
759 (Current directory: /project, current file: none) bash-$
760 </environment>
761 <agent>
762 Now let's run the script to see what output it produces and identify the
763     bug.
764
765 ```command
766 python prime_numbers.py
767 ```
768 </agent>
769 <environment>
770 First 10 prime numbers: [2, 3, 4, 5, 6, 7, 8, 9, 11, 13]
771
772 (Current directory: /project, current file: none) bash-$
773 </environment>
774 <agent>
775 Ah, I see the issue! The script incorrectly includes numbers 4, 6, 8, and
776     9 in the output as "prime" numbers when they're actually not prime. The
777     bug is in the `is_prime()` function - the range should go up to `int(n
778     **0.5) + 1` because the upper bound in Python's range is exclusive. Let
779     's fix this by modifying the range in the `is_prime()` function.
780
781 ```command
782 edit 4:4 << 'EOF'
783     for i in range(2, int(n**0.5) + 1):
784 EOF
785 ```
786 </agent>
787 <environment>

```

```

788 File updated. Please review the changes and make sure they are correct (
789     correct indentation, no duplicate lines, etc). Edit the file again if
790     necessary.
791 [File: /project/prime_numbers.py (16 lines total)]
792 1:def is_prime(n):
793 2: if n <= 1:
794 3: return False
795 4: for i in range(2, int(n**0.5) + 1):
796 5: if n % i == 0:
797 6: return False
798 7: return True
799 8:
800 9:primes = []
801 10:num = 2
802 11:while len(primes) < 10:
803 12: if is_prime(num):
804 13: primes.append(num)
805 14: num += 1
806 15:
807 16:print("First 10 prime numbers:", primes)
808
809 (Current directory: /project, current file: none) bash-$
810 </environment>
811 <agent>
812 Now let's run the script again to verify that the fix works and produces
813     the correct list of prime numbers.
814
815 ```command
816 python prime_numbers.py
817 ```
818 </agent>
819 <environment>
820 First 10 prime numbers: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
821
822 (Current directory: /project, current file: none) bash-$
823 </environment>
824 <agent>
825 The output now correctly shows the first 10 prime numbers. Let's go
826     through the checklist:
827
828 1. Reproduced the error: Yes, we saw the script incorrectly included non-
829     prime numbers in its output.
830 2. Confirmed the fix resolves the error: After modifying the range, the
831     script now correctly lists only prime numbers.
832 3. Edge cases considered: The fix handles all cases correctly - numbers
833     <=1 are non-prime, and the +1 ensures we check all potential divisors
834     up to the square root.
835 4. Existing tests: Since this is a standalone script and there are no
836     existing tests mentioned in the repository, this point isn't applicable
837     .
838
839 The fix is complete and correct. We can now submit the changes.
840
841 ```command
842 submit
843 ```
844 </agent>
845 </demonstration>

```



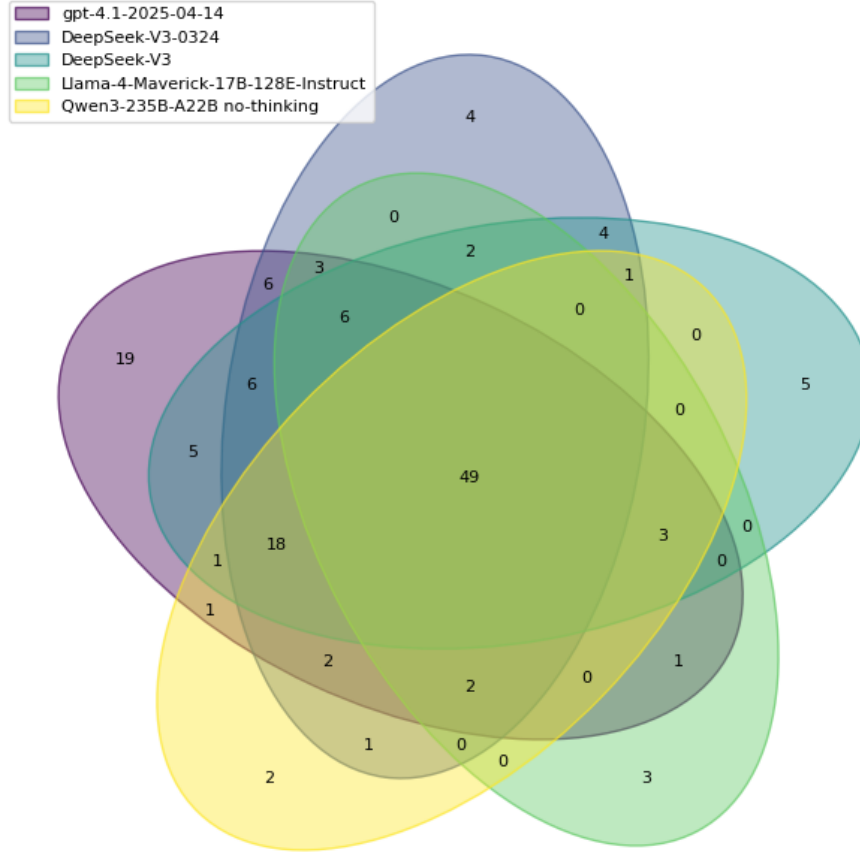


Figure 2: Overlap of solved tasks across selected models.

## J Evaluation details and experimental setup

To evaluate open-source models, we deployed them internally using the vLLM [Kwon et al., 2023] inference engine. Each model was served from 2 nodes equipped with 8×H200 GPUs each, and we used generation hyperparameters recommended by the respective model developers (e.g., temperature, p in top-p, max context length etc.). All evaluations were conducted using the same ReAct-style scaffolding and the system prompt described in Appendix I.

All evaluations were conducted using the same set of 294 SWE-rebench tasks. To reduce variance and enable pass@k computation, each model was run 5 times on the full benchmark using different random seeds. Among the evaluated models, DeepSeek-V3 was the largest and most resource-intensive; a single run over the full task set required approximately 7 hours.

As part of our analysis, we examined the overlap in successfully solved tasks across models to understand their relative capabilities and potential complementarity. Figure 2 visualizes the overlap of tasks solved at least once across selected models. While overall performance varies, the figure highlights that certain tasks are uniquely solved by specific models, suggesting complementary strengths.

### J.1 GPT-4.1 Trajectory Example

To better understand agent behavior on SWE-rebench tasks, we present a trajectory example that illustrates how a model attempts to resolve an issue—searching for relevant files, making edits,

865 running tests, and reacting to feedback. For brevity, some of the observations, e.g., ones containing  
866 file contents, are truncated.

### System Prompt

See Appendix I

867

868

### Environment Observation

I've uploaded a python code repository, your shell is currently at the repository root. Consider the following PR description:

DESCRIPTION:

Network.plot\_attribute

I find something wrong with:

```
```python
if attribute[0].lower() == "z":
    y_label = "Z (Ohm)"
    y[x == 1.] = 1. + 1e-12 # solve numerical
    singularity
    y[x == -1.] = -1. + 1e-12 # solve numerical
    singularity
    y = self.z0[0].real * (1+y) / (1-y)
...
```
```

I think it should be:

```
```python
if attribute[0].lower() == "z":
    y_label = "Z (Ohm)"
    y[y == 1.] = 1. + 1e-12 # solve numerical
    singularity
    y[y == -1.] = -1. + 1e-12 # solve numerical
    singularity
    y = self.z0[0].real * (1+y) / (1-y)
...
```
```

Am I right?

Can you help me implement the necessary changes to the repository so that the requirements specified in the <pr\_description> are met? I've already taken care of all changes to any of the test files described in the <pr\_description>. This means you DON'T have to modify the testing logic or any of the tests in any way!

Your task is to make the minimal changes to non-tests files in the {location} directory to ensure the <pr\_description> is satisfied.

Follow these steps to resolve the issue:

1. As a first step, it might be a good idea to explore the repo to familiarize yourself with its structure.

869

2. If <pr\_description> reports some error, create a script to reproduce the error and execute it with `python <filename.py>`, to confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well

Your thinking should be thorough and so it's fine if it's very long.  
(Current directory: /scikit-rf)  
bash-\$

870

#### Agent Output

Let's start by searching for the `plot_attribute` method in the repository to locate where the code exists and verify the current implementation that is in question.

```
grep -rn "def plot_attribute"
```

871

872

#### Environment Observation

```
skrf/network.py:4830: def plot_attribute( self,  
  
(Current directory: /scikit-rf)  
bash-$
```

873

#### Agent Output

The `plot_attribute` function is located in `skrf/network.py` at line 4830. Let's open the surrounding lines to inspect the relevant section and verify the code snippet described in the issue.

```
open skrf/network.py 4830
```

874

875

#### Environment Observation

```
[File: skrf/network.py (8517 lines total)]  
...  
(Current directory: /scikit-rf, Current file: /scikit-rf/skrf/network.py)  
bash-$
```

876

### Agent Output

We can see in the `plot_attribute` function, around line 4872, the code matches the initial suggestion:

```
if attribute[0].lower() == "z":
    y_label = "Z (Ohm)"
    y[x == 1.] = 1. + 1e-12 # solve numerical
    singularity
    y[x == -1.] = -1. + 1e-12 # solve numerical
    singularity}
y = self.z0[0].real * (1+y) / (1-y)
```

The issue points out that `y[x == 1.]` should really be `y[y == 1.]`, as the intent is to fix elements of `y` with possible singularity issues, not `x`.  
The correction is:

```
y[y == 1.] = 1. + 1e-12
y[y == -1.] = -1. + 1e-12
```

Let's replace these two lines with the correct ones.

```
edit 4872:4873 << 'EOF
    y[y == 1.] = 1. + 1e-12 # solve numerical
    singularity
    y[y == -1.] = -1. + 1e-12 # solve numerical
    singularity
EOF
```

877  
878

### Environment Observation

File updated. Please review the changes and make sure they are correct (correct indentation, no duplicate lines, etc). Edit the file again if necessary.

```
[File: /scikit-rf/skrf/network.py (8517 lines total)]
...
(Current directory: /scikit-rf, Current file: /scikit-
rf/skrf/network.py)
bash-$
```

879

### Agent Output

The code has been updated as intended. Now, I need to ensure that the fix behaves as expected and doesn't cause any regressions. I'll check if there are tests that might cover this situation and run them. Let's look for a tests directory or test files in the repository.

```
ls -l
```

880

881

#### Environment Observation

```
total 52
...
(Current directory: /scikit-rf, Current file: /scikit-
rf/skrf/network.py)
bash-$
```

882

#### Agent Output

There is a skrf directory which is likely where the main package code lives, but there is no obvious tests directory here. Let's look inside the skrf directory to see if the tests are located there.

```
ls -l skrf
```

883

884

#### Environment Observation

```
total 812
...
(Current directory: /scikit-rf, Current file: /scikit-
rf/skrf/network.py)
bash-$
```

885

#### Agent Output

There is a tests directory inside skrf. Let's list its contents to see what test files are available and determine if there is test coverage for plotting or the plot\_attribute method.

```
ls -l skrf/tests
```

886

887

#### Environment Observation

```
total 7272
...
(Current directory: /scikit-rf, Current file: /scikit-
rf/skrf/network.py)
bash-$
```

888

### Agent Output

The skrf/tests directory contains several test files, including test\_network.py and test\_plotting.py, both of which are likely candidates for testing the (now-fixed) code. To ensure nothing is broken, let's run the tests in this directory.

I'll use pytest to run all tests in skrf/tests.

```
pytest skrf/tests
```

889

890

### Environment Observation

```
===== test session starts =====
```

```
...
```

```
===== 612 passed, 3 skipped, 4 xfailed in 91.69s  
(0:01:31) =====
```

```
(Current directory: /scikit-rf, Current file: /scikit-  
rf/skrf/network.py)
```

```
bash-$
```

891

### Agent Output

1. If an error was reported in the issue description, I have successfully reproduced it.

The issue was a logic mistake in the code, not a runtime error, so reproduction was via code inspection.

2. If an error was reported in the issue description, I have confirmed that my fix resolves the error.

I updated the lines so the fix is implemented exactly as intended, addressing the correct variable (y).

3. I have thought about edge cases and made sure my fix handles them as well.

The fix only adjusts y values exactly at 1.0 or -1.0, so it specifically targets the singularity and doesn't affect other cases. Edge cases are thus preserved as intended.

4. I have run existing tests in the repository that might have been affected by the change I made and confirmed that they pass.

All tests in the skrf/tests directory were run. There were no new errors, and the status quo was maintained (some expected skips/xfails), with all others passing.

892

This solution is correct and safe to submit.

submit

## J.2 Qwen2.5-32B Coder hallucination example

Below is a representative failure case for Qwen2.5-Coder-32B-Instruct. Despite its strong performance on various code generation benchmarks, this model repeatedly hallucinates environment responses and gets stuck in a loop of formatting and syntactic errors when evaluated on SWE-rebench. The behavior reflects poor instruction following abilities of the model with limited capacity to correct its course based on feedback. This failure mode is common among low-performing models in our benchmark, but it is particularly pronounced in the Qwen2.5-Coder, where such behavior frequently prevents meaningful progress toward a solution.

### Environment Observation

Error: Multiple commands were extracted, but only one is allowed: ...

(Current directory: /sqlacodegen)  
bash-\$

### Agent Output

It seems there is still an issue with multiple commands being extracted. Let's ensure that we only include a single command block to modify the `get_adapted_type` method in `generators.py`.

Here is the single command block again:

```
edit 183:183 << 'EOF'
def get_adapted_type(self, coltype: TypeEngine) ->
    TypeEngine:
    if isinstance(coltype, TypeDecorator) and not
        hasattr(coltype.__class__, 'impl'):
        log.warning(f'TypeDecorator implementation {
            coltype.__class__.__name__} lacks a
            class-level variable \'impl\'. Setting
            to default type TEXT.')
        coltype.__class__.impl = TEXT # Set to
            default type TEXT at the class level
    try:
        supercls = self.get_super_class(coltype)
        if supercls is not None:
            new_coltype = coltype.adapt(supercls)
            if new_coltype.__class__ is not coltype.
                __class__:
                return new\_coltype
    except AssertionError as e:
        log.error(f'Error adapting type {coltype.
            __class__.__name__}: {e}. Using original
            type.')
```

```

        return coltype
    return coltype
EOF

```

```

</agent> <environment>
File updated. Please review the changes and make sure
they are correct (correct indentation, no duplicate
lines, etc). Edit the file again if necessary.

[File: /sqlacodegen/sqlacodegen/generators.py (1016
lines total)]
...
(Current directory: /sqlacodegen, current file: /
sqlacodegen/sqlacodegen/generators.py)
bash-$

```

## K Dataset schema and fields

We describe task instances using SWE-bench schema with a few extra annotations related to license type, dependency versioning and task quality. Key distinctions and additions compared to the standard SWE-bench schema include

- **License information** (`license_name`): A new field specifying the open-source license of the source repository.
- **Enhanced metadata** (`meta`): This JSON dictionary now includes richer information beyond “lite” criteria compliance, such as the specific commit type used for patch generation (`commit_name`), the number of modified files (`num_modified_files`), and LLM-predicted scores (`llm_score`) for task difficulty, issue clarity, and test quality.
- **Pinned dependencies** (`requirements`, `environment`): Two new fields storing exact dependency versions (e.g., from `pip freeze` or `conda env export`) captured after successful environment setup, ensuring high reproducibility.

Table 5 lists all metadata fields included with each SWE-rebench task instance.

## L Data collection funnel and potential enhancements

Our automated pipeline processes a large volume of raw data through several stages to curate the final SWE-rebench dataset. Table 6 summarizes the data flow, showing the approximate input and output sizes at each key stage, along with acceptance rates. Understanding this funnel helps identify bottlenecks and areas for future improvement to increase the yield of task instances.

Several strategies could be employed to potentially increase the yield at various stages of this pipeline:

- **Expanding language scope:** Currently, the pipeline focuses exclusively on Python projects. Extending support to other popular programming languages (e.g., JavaScript, Java, C++) would significantly broaden the pool of potential repositories and tasks.
- **Generalizing task sources**
  - **Beyond issue-linked PRs:** Instead of strictly requiring a PR to be linked to a formal issue, we could consider general pull requests that describe a problem and solution directly within the PR description. This would capture a wider range of code changes.
  - **Using tasks without pre-existing tests:** For PRs that solve a problem but do not include new or modified tests, an LLM could potentially be employed to generate test cases based on the issue description and the provided solution patch. This would require validation of LLM-generated tests.



Table 5: SWE-rebench dataset schema.

| Field name               | Type        | Description   |
|--------------------------|-------------|---|
| instance_id              | str         | A formatted instance identifier, typically <code>repo_owner__repo_name-PR_number</code> .   |
| patch                    | str         | The gold solution patch (code changes from the PR, excluding test files) that resolved the issue.   |
| repo                     | str         | The repository owner/name identifier from GitHub.   |
| base_commit              | str         | The commit hash representing the repository’s HEAD before the solution PR was applied.  |
| hints_text               | str         | Comments made on the issue before the creation of the solution PR’s first commit.   |
| created_at               | str         | The creation timestamp of the pull request (ISO format).  |
| test_patch               | str         | A patch containing only changes to test files contributed by the solution PR.   |
| problem_statement        | str         | The concatenated title and body of the GitHub issue.  |
| version                  | str         | The normalized project version (e.g., "1.2") used for grouping and environment setup.   |
| environment_setup_commit | str         | The specific commit hash used as a basis for environment setup and dependency installation for this task’s version group.   |
| FAIL_TO_PASS             | list[str]   | JSON list of test identifiers that failed before and passed after applying the solution patch.  |
| PASS_TO_PASS             | list[str]   | JSON list of test identifiers that passed both before and after applying the solution patch.  |
| meta                     | dict (JSON) | A dictionary containing extended metadata. Includes:<br><code>commit_name</code> : (str) 'head_commit' or 'merge_commit' used for patch generation.<br><code>failed_lite_validators</code> : (list[str]) List of reasons an instance is not "lite".<br><code>has_test_patch</code> : (bool) Whether a test patch exists.<br><code>is_lite</code> : (bool) Whether the instance meets "lite" criteria.<br><code>num_modified_files</code> : (int) Number of files changed by the solution patch.<br><code>llm_score</code> : (dict) LLM-predicted scores:<br><code>difficulty_score</code> : (int) Predicted task difficulty.<br><code>issue_text_score</code> : (int) Predicted issue clarity.<br><code>test_score</code> : (int) Predicted test patch correctness. |
| license_name             | str         | The SPDX identifier or common name of the repository’s license (e.g., "MIT", "Apache-2.0").   |
| install_config           | dict (JSON) | A dictionary with the configuration used for automated environment setup. Includes fields like <code>python version</code> , <code>install command</code> , <code>test_cmd</code> , <code>dependency file paths</code> , etc.   |
| requirements             | str         | A string containing the frozen Python dependencies (e.g., output of <code>pip freeze</code> ) for the specific environment.   |
| environment              | str         | A string containing the full environment specification (e.g., output of <code>conda env export</code> ) for the specific environment.   |

Table 6: Summary of the data collection funnel for SWE-rebench.

| Stage              | Input Size                     | Output Size           | Acceptance Rate |
|--------------------|--------------------------------|-----------------------|-----------------|
| Match PRs & Issues | ~10M PRs, 6M issues, 32K repos | ~450K task candidates | ~5% (of PRs)    |
| Filter Tasks       | ~450K candidates               | ~150K filtered tasks  | ~33%            |
| Install & Validate | ~150K tasks                    | ~21K valid tasks      | ~14%            |
| Annotate           | ~21K tasks                     | ~21K annotated tasks  | ~100%           |

- **Improving dependency installation:** The automated dependency installation stage, with a current success rate of approximately 14%, represents a bottleneck. Substantial gains could be achieved by further improving the LLM for generating installation recipes (e.g., through more extensive fine-tuning, enhanced prompting, or employing more capable base models). Additionally, for repositories that yield a high number of task instances or are particularly popular, a more resource-intensive interactive agent could be selectively deployed to handle complex installation scenarios.

## M SWE-rebench dataset statistics

Table 7 presents key statistics of the SWE-rebench dataset, offering insights into characteristics of its tasks. These statistics, computed across all 21,336 instances, include measures of task description length, code modification size, test suite changes, and LLM-assessed quality scores.

Table 7: SWE-rebench dataset statistics.

| Metric               | Mean   | p50 | p75   | p95    |
|----------------------|--------|-----|-------|--------|
| Issue Length (words) | 141.67 | 91  | 173   | 412.25 |
| Files Edited         | 3.46   | 2   | 4     | 10     |
| Lines Edited         | 142.17 | 37  | 112   | 500    |
| Fail-to-Pass Tests   | 14.56  | 2   | 5     | 37     |
| Pass-to-Pass Tests   | 85.81  | 22  | 64    | 351    |
| Total Tests          | 105.43 | 31  | 82.25 | 428    |
| Difficulty Score     | 1.13   | 1   | 2     | 2      |
| Issue Text Score     | 1.04   | 1   | 2     | 3      |
| Test Score           | 1.38   | 2   | 2     | 3      |

On average, task instances in SWE-rebench feature issue descriptions of approximately 142 words, with 75% of tasks having descriptions of 173 words or fewer. This suggests that most tasks have concise problem statements. In terms of code changes, tasks typically involve modifications to around 3-4 files (mean Files Edited is 3.46), with a 75th percentile of 4 files. The number of added lines averages around 97, while the 75th percentile is 76 lines, indicating that the majority of solutions are not excessively large.

Regarding test dynamics, the average number of tests transitioning from fail to pass (F2P Tests) is approximately 15. The 75th percentile for F2P tests is 5, suggesting that many tasks have a focused set of critical tests verifying the fix. The average number of tests that consistently pass before and after the patch (P2P Tests) is about 86, contributing to an average total of around 105 tests involved per task.

The LLM-assessed quality scores (where lower scores generally indicate higher quality or lower difficulty) show that, on average, tasks are perceived as having relatively low difficulty (mean Difficulty Score is 1.13) and clear issue descriptions (mean Issue Score is 1.04). The 75th percentile for these scores is 2, reinforcing that a majority of tasks are considered well-defined and not overly complex by the automated assessment model. These statistics illustrate the typical characteristics of tasks in SWE-rebench, highlighting their suitability for training and evaluating software engineering agents.