UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench

Anonymous ACL submission

Abstract

The rise of Large Language Models (LLMs) has fueled the development of coding agents designed to solve real-world code generation tasks. SWE-Bench has become a widely used benchmark for evaluating the code generation capabilities of these agents, using real-world problems derived from GitHub issues and their corresponding pull requests. However, the manually written test cases included in these pull requests are often insufficient, allowing some generated patches to pass the tests while failing to resolve the underlying issue. To address this challenge, we introduce UTGenerator, an LLM-driven test case generator that automatically analyzes codebases and dependencies to generate test cases for real-world Python projects. Building on UTGenerator, we propose UTBoost, a comprehensive framework for test case augmentation. UTBoost ensures that the generated patch functions equivalently to the gold patch and passes the same test cases. In our evaluation, we identified 36 task instances with insufficient test cases and uncovered 345 erroneous patches incorrectly labeled as passed in the original SWE Bench. These corrections significantly impact leaderboard rankings, affecting 40.9% of entries in SWE-Bench Lite and 24.4% in SWE-Bench Verified.

1 Introduction

Advances in large language models (LLMs) have enabled the development of automated coding agents capable of generating code for software engineering tasks. To evaluate their effectiveness on real-world Python projects, prior work introduced SWE-Bench (Jimenez et al., 2024), a benchmark specifically designed for this purpose. Each instance in SWE-Bench consists of a repository, an issue description, and a set of manually written test cases to verify whether the issue is resolved. The task of coding agents is to generate a patch that resolves the issue, as demonstrated by successfully passing all relevant test cases.

However, the manually written test cases in SWE-Bench can be too narrow to comprehensively evaluate the correctness of the patches generated by coding agents (OpenAI, 2024; Chen and Jiang, 2024; Aleithan et al., 2024). Consequently, erroneous patches generated by agents may be incorrectly considered to resolve the issue, compromising the reliability of SWE-Bench.

To comprehensively evaluate the code generation ability of coding agents on real-world Python projects, we propose a novel LLM-based test case generator named UTGenerator, which automatically generates test cases. UTGenerator operates in two steps. First, it identifies where new test cases should be added by analyzing the codebase and issue description. Then, based on this location, UTGenerator analyzes package dependencies and generates code as unit test cases.

To verify whether the generated patch functions equivalently to the gold patch on the new test cases, we employ intramorphic testing (Rigger and Su, 2022) to construct a test oracle. Intramorphic testing is a white-box automated testing technique that establishes a test oracle by comparing the outputs of the original and modified systems using the same input. Since the gold patch and the generated patch are expected to resolve the issue equivalently, the test oracle ensures that both patches pass the same issue-related test cases.

As a motivating example, the issue description in the instance `mwaskom__seaborn-3010` requires `PolyFit`, a function that computes polynomial fits for data, to handle missing data in the inputs `x` and `y`. However, the original test case for this issue, as shown in Listing 1 (line 3), only considers scenarios where both `x` and `y` have missing data. A comprehensive set of tests should include cases where only one of the inputs, `x` or `y`, has missing data. Our solution adds a test case where only `x` has missing data to complement the original test cases, as shown in Listing 2 (lines 3-4). Unlike the gold patch that resolves the issue (Listing 3), the generated patch fails to handle these additional cases but throws an error message, as shown in Listing 4 (lines 4-5). Thus, while the generated patch passes the original test case, it does not resolve the issue.

However, adding new test cases is not sufficient if these test cases are not properly accounted for in the SWE-Bench evaluation pipeline. This is because SWE-Bench uses a parser based on regular expressions to extract test cases from the test log, but the original parser fails to parse many test cases due to certain defects. For example, it can not handle test cases that span multiple lines in the test log. To address these issues, we improved the original SWE-Bench parser by fixing its defects to enhance the rigor of the evaluation process. With the improved parser, we identified 64 erroneous patches generated by coding agents that were incorrectly labeled as passed in SWE-Bench Lite, and 79 erroneous patches that were similarly mislabeled in SWE-Bench Verified.

Building on UTGenerator and intramorphic testing, we propose UTBoost, a framework for augmenting test cases in real-world Python projects. Given a SWE-Bench instance and a generated patch as input, UTBoost generates new test cases (e.g., Listing 2) and flags the instance as suspicious if the gold patch and the generated patch behave differently in the new test cases. If the generated test cases complement the original ones, they are added to the original test suite.

We applied UTBoost to SWE-Bench Lite (Carlos E. Jimenez, 2024) and SWE-Bench Verified (OpenAI, 2024). Our analysis identified 176 erroneous patches in SWE-Bench Lite and 169 in SWE-Bench Verified that were incorrectly evaluated as passing in the original SWE-Bench. These corrections resulted in significant leaderboard updates, with ranking changes affecting 40.9% of entries in SWE-Bench Lite and 24.4% in SWE-Bench Verified. Notably, in the original SWE-Bench Verified leaderboard, `Amazon-Q-Developer-Agent` ranked 1st and `devlo` ranked 2nd; however, both now share the 1st rank in the updated leaderboard. With the augmented test cases generated by UTGenerator, we identified 7.7% (23/300) of instances in SWE-Bench Lite and 5.2% (26/500) of instances in SWE-Bench Verified as having insufficient test cases. Using our improved parser, we also identified annotation errors in 54.6% (164/300) of instances in SWE-Bench Lite and 54.2% (271/500) of instances in SWE-Bench Verified.

```python
def test_missing_data(self, df):
    groupby = GroupBy(["group"])
    df.iloc[5:10] = np.nan
    res1 = PolyFit()(df[["x", "y"]], groupby, "x", {})
    res2 = PolyFit()(df[["x", "y"]].dropna(), groupby, "x", {})
    assert_frame_equal(res1, res2)
```

Listing 1: The original test case in SWE-Bench that only considers the case when there is missing data both in x and y (`mwaskom__seaborn-3010`).

```python
def test_none_values(self):
    df = pd.DataFrame({
        "x": [1, 2, 3, None, 4, 5, 6],
        "y": [1, 4, 9, 16, 25, 36, 49],
        "group": [1, 1, 1, 1, 1, 1, 1]})
    groupby = GroupBy(["group"])
    res1 = PolyFit()(df, groupby, "x", {})
    res2 = PolyFit()(df.dropna(), groupby, "x", {})
    assert_frame_equal(res1, res2)
```

Listing 2: The augmented test case that considers the case when there is only missing data in x (`mwaskom__seaborn-3010`).

```python
def __call__(self, data, groupby, orient, scales):
    return (groupby.apply(
        data.dropna(subset=["x", "y"]),
        self._fit_predict))
```

Listing 3: The gold patch (`mwaskom__seaborn-3010`).

```python
1 def _fit_predict(self, data):
2     y = data["y"].dropna()
3     x = data["x"].dropna()
4     if x.shape[0] != y.shape[0]:
5         raise ValueError("x and y must have the same number of non-missing values")
6     if x.nunique() <= self.order:
7         # TODO warn?
8     xx = yy = []
```

Listing 4: The generated patch by `IBM SWE-1.0` (`mwaskom__seaborn-3010`).

2 SWE-Bench

This section introduces SWE-Bench (Jimenez et al., 2024) and its two splits: SWE-Bench Lite and SWE-Bench Verified (OpenAI, 2024).

SWE-Bench SWE-Bench is a benchmark for evaluating the code generation capabilities of coding agents on real-world GitHub projects. It features 12 popular Python repositories and focuses on generating pull requests to address specific issues by producing code edits represented as patch files. Each task instance includes a gold patch crafted by human developers, which serves as a reference for resolving the issue.

In the SWE-Bench evaluation, an agent generates a patch based on the issue description and the codebase. The patch is then applied, and its correctness is assessed using two types of unit tests: `PASS_TO_PASS` and `FAIL_TO_PASS`. The benchmark evaluates performance by measuring the percentage of patches that successfully pass both types of tests for each instance. To extract test results from logs generated in instance-specific virtual environments, the evaluation pipeline employs repository-specific parsers with manually crafted regular expressions, averaging 23 lines of code.

SWE-Bench Lite The full SWE-bench test split comprises 2,294 issue-commit pairs across 12 Python repositories. SWE-Bench Lite is a lite version of SWE-Bench, which is a subset of SWE-Bench with 300 task instances. These instances focus on evaluating functional bug fixes, ensuring they are more self-contained while maintaining the original diversity across 11 of the 12 repositories.

SWE-Bench Verified OpenAI introduced a new version of SWE-Bench (OpenAI, 2024), named SWE-Bench Verified, to improve the robustness and reliability of the evaluation. They identified two major problems with the data in SWE-Bench:

- **Unit tests**: The unit tests are sometimes too specific or unrelated to the issue, which potentially causes correct solutions to be rejected.
- **Issue description**: Many samples have an issue description that is underspecified, leading to ambiguity on the problem and how it should be fixed.

To address these two issues, OenAI launched a human annotation campaign with 93 professional software developers to screen each sample of the SWE-bench test set for appropriately scoped unit tests and well-specified issue descriptions. Finally, they released SWE-Bench Verified, a subset of 500 samples that the human annotators verified to be non-problematic.

3 Methodology

In this section, we introduce UTBoost, an architecture for comprehensively testing coding agents using intramorphic testing (Rigger and Su, 2022). We discuss the construction of the test oracle, detail the workflow of UTBoost, and present UTGenerator, our LLM-based test case generator.

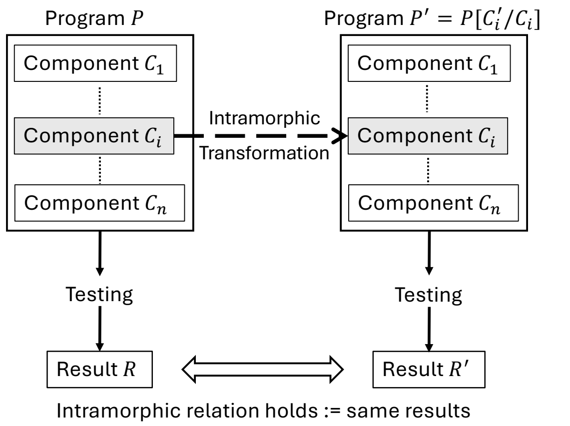


Figure 1: The architecture of intramorphic testing (we define P, C_i, R as the program, the i-th component of the program, and the program's output, respectively).

3.1 Test Oracle

A test oracle determines whether a system behaves correctly for a given input. Automated testing techniques rely on an automated test oracle to test the system without user interaction. In UTBoost, we employ intramorphic testing to establish a test oracle for evaluating the generated patches. Intramorphic testing creates a modified version of the system, enabling a single input to define a test oracle that establishes the relationship between the outputs of the original and modified systems.

In SWE-Bench, the gold patch serves as the ground truth for resolving the issue. A generated patch that resolves the issue provides an alternative implementation to achieve the same functionality as the gold patch and should pass the same test cases associated with the issue. We define P as the program to which the gold patch is applied and P' as the program to which the generated patch is applied. The difference between P and P' lies in the component C of P, which is transformed into C' in P' through an intramorphic transformation, as illustrated in Figure 1. We then construct a test oracle to evaluate the generated patches, defined by the intramorphic relation $P(T) = P'(T)$, where T represents the test cases. To the best of our knowledge, this is the first work to apply intramorphic testing for evaluating real-world software.

3.2 UTBoost Workflow

UTBoost is an automated testing approach that constructs a test oracle for evaluating generated patches through intramorphic testing, as illustrated in Figure 2. We define the original test cases in SWE-Bench as T_{orig} and the augmented test cases

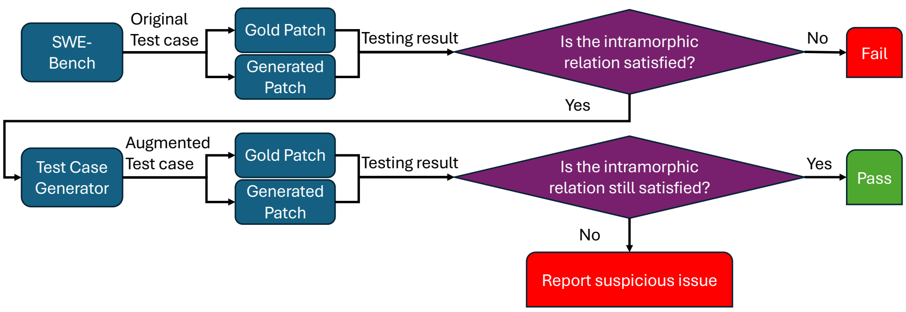


Figure 2: The architecture of UTBoost.

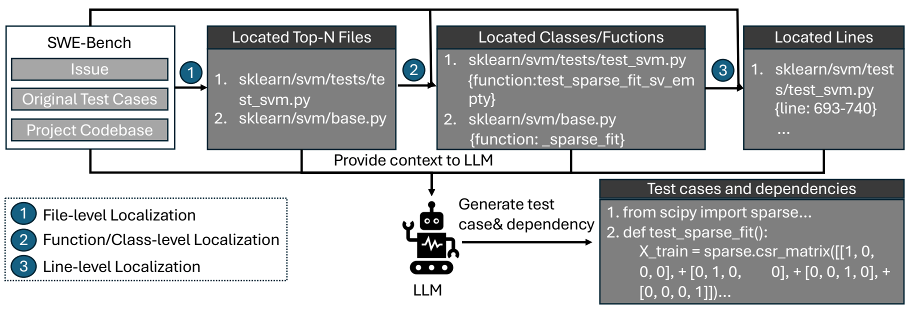


Figure 3: The architecture of UTGenerator.

as T_{aug}. The UTBoost process consists of two steps: (1) testing on the original test cases, and (2) testing on the augmented test cases.

In the first step, we select the generated patches that pass the original test cases as gold patches in SWE-Bench, satisfying the intramorphic relation $P(T_{orig}) = P'(T_{orig})$. We then invoke the test case generator, UTGenerator, to produce augmented test cases, T_{aug}.

In the second step, we apply the augmented test cases T_{aug} to both the program P and the program P' to check whether the intramorphic relation still holds for T_{aug}. If the intramorphic relation $P(T_{aug}) = P'(T_{aug})$ does not hold, we report it as a suspicious issue, as either the gold patch or the generated patch fails to pass the augmented test cases. This discrepancy indicates that the original test cases T_{orig} are insufficient for fully evaluating the patch's correctness. To achieve a more comprehensive evaluation, we add T_{aug} to the original test suite.

3.3 UTGenerator

In UTBoost, a test case generator is required to produce augmented test cases for more comprehensive testing. To enhance the diversity of these test cases, we introduce UTGenerator, an LLM-based test case generator. The architecture of UTGenerator, illustrated in Figure 3, consists of two steps: (1) localization and (2) test case generation. The localization step operates at three levels: file-level, function/class-level, and line-level.

3.3.1 File-level Localization

Since real-world project codebases are generally very large, we construct a tree-structured representation of the codebase to organize its files and their locations. Files and folders at the same directory level are aligned vertically. UTGenerator then takes the issue description, the original test patch from SWE-Bench, and the tree-structured codebase as input to an LLM, which identifies the Top-N files most likely to require edits for adding test cases. Figure 3 illustrates an example of the three levels of file localization in `scikit-learn__scikit-learn-14894`, an instance from SWE-Bench Verified. In the first step, UTGenerator identifies `sklearn/svm/tests/test_svm.py` and `sklearn/svm/base.py` as the most likely files to add the augmented test cases.

3.3.2 Function/class-level Localization

For function/class-level localization, we first compress the codebase files by retaining only the headers of classes and functions. After identifying the Top-N files for potential edits through file-level localization, we provide their compressed formats, along with the issue description and the original test patch, as input to an LLM. The LLM analyzes these inputs to identify the functions or classes most likely to require augmented test cases. As illustrated in the second step of Figure 3, this process identifies the functions `test_sparse_fit_sv_empty` and `_sparse_fit` as the most likely candidates for adding the augmented test cases.

3.3.3 Line-level Localization

After identifying the functions or classes where augmented test cases should be added, we extract their code snippets and provide them, along with the issue description and the original test patch, as input to an LLM. The LLM analyzes the inputs to determine the specific lines within the functions or classes that are most suitable for adding the augmented test cases. For instance, as shown in Figure 3, lines 693–740 of the file `sklearn/SVM/tests/test_svm.py` are identified as the most likely candidates for adding the augmented test cases.

3.3.4 Test Case Generation

The final step is to generate the augmented test cases and their dependencies. We use a context window of `x` lines of code to expand the located lines and control the range for adding the augmented test cases. For example, if the located lines are from line 693 to 740, the context window is defined as `[max(693-x, 0), min(740+x, end_line)]`, where `end_line` represents the last line of the file. We then provide the code snippets within this context window, along with the issue description and the original test patch, as input to an LLM, asking it to generate the augmented test cases and their dependencies. As shown in Figure 3, UTGenerator generates a test case named `test_sparse_fit` and the corresponding dependency.

3.4 Improved Parser

The parser plays a crucial role in SWE-Bench, as it extracts test cases from the test logs. However, we observed that many test cases are not successfully parsed by the original SWE-Bench parser. For instance, some test logs contain side messages that the original parser fails to handle correctly. As shown in Listing 5, the test log for the test case `test_immutable_content_type` spans two lines (lines 2-3). The original SWE-Bench parser incorrectly processes this log by splitting the line at the last occurrence of the suffix and assigning the portion before the suffix as the test case name (Listing 6, line 6). As a result, it extracts `"Regression for #9362"` as the test case name in `PASS_TO_PASS` of `django__django-13710`.[1]

```
1 # Test log of django__django-13710
2 test_immutable_content_type
     (admin_inlines.tests.TestInlineAdminForm)
3 Regression for #9362 ... ok
4 test_all_inline_media
     (admin_inlines.tests.TestInlineMedia) ... ok
```

Listing 5: Test Log of `django__django-13710` (before gold patch is applied).

```
1 # SWE-Bench parser for django
2 pass_suffixes = (" ... ok", " ... OK", " ...  OK")
3 for suffix in pass_suffixes:
4     if line.endswith(suffix):
5         ... # omits several lines of code
6         test = line.rsplit(suffix, 1)[0]
7         test_status_map[test] =
      TestStatus.PASSED.value
8         break
```

Listing 6: Original SWE-Bench parser for django.

To address the limitation that the original SWE-Bench parser can not parser test cases that span multiple lines in the test log, we improve the parser by using a queue (Listing 7, line 3) to record the neighbored log data and use regular expression to match the test case (line 2). While the log for one test case spans multiple lines, the improved parser will search until it gets the correct test case name (Listing 7, lines 12-18). By using the improved parser, we can get the correct test case name `test_immutable_content_type (admin_inlines.tests.TestInlineAdminForm)` in Listing 5 (lines 2-3). The improved parser addresses several limitations of the original SWE-Bench parser, not limited to this example, thereby enhancing the rigor of SWE-Bench.

4 Experiments

In this section, we evaluate the performance of UTBoost and our improved parser on SWE-Bench. Our investigation is guided by three research questions:

- RQ1: How effective is UTBoost in identifying insufficient test cases?
- RQ2: How does the parser affect the evaluation of SWE-Bench?

[1] https://huggingface.co/datasets/princeton-nlp/SWE-bench_Lite/

```
# Improved parser for django
pattern_test = r"[a-zA-Z_]\w*\s\([\w.]+\)"
previous_line = deque()
for line in lines:
    line = line.strip()
    pass_suffixes = (" ... ok", " ... OK", " ... OK")
    for suffix in pass_suffixes:
        if line.endswith(suffix):
            test = line.rsplit(suffix, 1)[0]
            # process when test log in separate lines
            if not re.fullmatch(pattern_test, test):
                pt = -1
                while previous_line[pt]:
                    if re.fullmatch(pattern_test, previous_line[pt]):
                        test = previous_line[pt]
                        break
                    pt -= 1
            test_status_map[test] = TestStatus.PASSED.value
            break
    previous_line.append(line)
```

Listing 7: Improved parser for Django.

- RQ3: How do insufficient test cases and erroneous annotations affect the SWE-Bench leaderboard?

4.1 Experiment Settings

In our evaluation, we use the generated patches of the coding agents from the official SWE-Bench experiment repository.[2] We extract the generated patches of the coding agents that pass the original SWE-Bench tests and evaluate them using our augmented test cases. Coding agents that do not provide generated patches are excluded from our analysis. Our code and data can be found at Anonymous (2024).

In UTGenerator, we use GPT-4o (gpt-4o-2024-08-06) as the LLM. We set a context window of 10 lines of code for test case generation and use Top-3 for file-level localization. During the localization phase, the temperature is set to 0.8. In the test case generation phase, we sample one patch with a temperature of 0, 20 patches with a temperature of 0.8, 20 patches with a temperature of 0.9, and 20 patches with a temperature of 0.99. When UTBoost identifies conflicts between the results of the gold patch and the generated patches under the augmented test cases, two of our authors manually review the test cases and patches, reaching a consensus on whether the issue arises from insufficient test cases. Generating test cases using UTGenerator costs an average of $1.6 per task instance of SWE-Bench for API usage. We use an AMD 5800X server with Ubuntu 22.04 LTS to evaluate the gold patches and generated patches on the test cases, which takes 300 hours to complete.

[2]https://github.com/swe-bench/experiments

4.2 Effectiveness of UTBoost

Overall, we identified 36 task instances with insufficient test cases in SWE-Bench using the augmented test cases generated by UTBoost. Of these, 23 instances are from SWE-Bench Lite, and 26 are from SWE-Bench Verified. We then applied the augmented test cases to evaluate the generated patches that passed the original SWE-Bench test cases.

There are 599 generated patches that pass the 23 instances in SWE-Bench Lite and 584 generated patches that pass the 26 instances in SWE-Bench Verified. However, our augmented test cases found that 28.4% (170/599) of the generated patches in SWE-Bench Lite and 15.7% (92/584) in SWE-Bench Verified are erroneous.

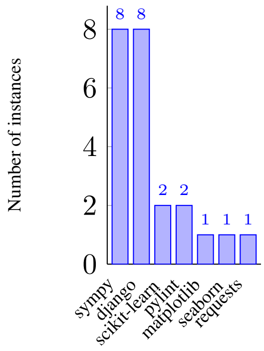
These findings demonstrate that a significant proportion of generated patches recorded as passing in SWE-Bench fail to address the issues effectively because the test cases in SWE-Bench are insufficient. This underscores the effectiveness of the augmented test cases generated by UTBoost.

Using UTBoost, we uncovered insufficient test cases across 9 of the 12 Python projects included in SWE-Bench. The distribution of the insufficient test cases and erroneous patches are shown in Figure 4. Notably, `django` and `sympy` are the most frequent projects with insufficient test cases and erroneous patches in both SWE-Bench Lite and SWE-Bench Verified. Together, `django` and `sympy` account for 84.1% (143/170) of the erroneous patches in SWE-Bench Lite and 82.6% (76/92) in SWE-Bench Verified.

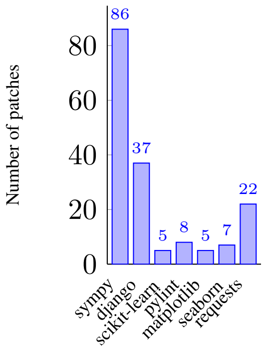
Answer to RQ1 *The augmented test cases generated by UTBoost identified 170 erroneous patches in SWE-Bench Lite and 92 in SWE-Bench Verified that were evaluated as passed by the original test cases, demonstrating the effectiveness of UTBoost in detecting erroneous patches.*

4.3 Impact of the Parser

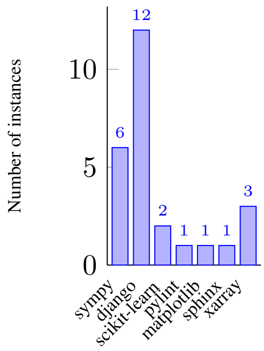
In our evaluation, we found many annotation errors in SWE-Bench stem from defects in the original SWE-Bench parser. For instance, 55 test cases in `django__django-15278`'s `PASS_TO_PASS` are unsuccessfully parsed in SWE-Bench Verified. We present three selected test cases in Listing 8 (lines 2-4). The message "`Tests altering of the`



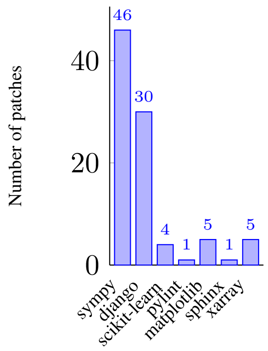
(a) Insufficient test cases in SWE-Bench Lite



(b) Erroneous patches in SWE-Bench Lite



(c) Insufficient Test Cases in SWE-Bench Verified



(d) Erroneous Patches in SWE-Bench Verified

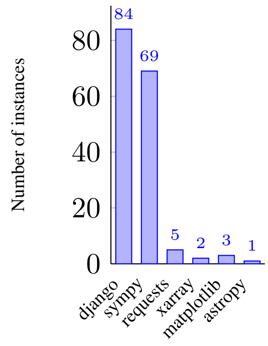
Figure 4: Distribution of Insufficient Test Cases And Erroneous Patches.

```
# Three selected unsuccessfully parsed test cases
Tests removing and adding unique_together
    constraints on a model.
Tries creating a model's table, and then deleting
    it.
Tests altering of the primary key.
```

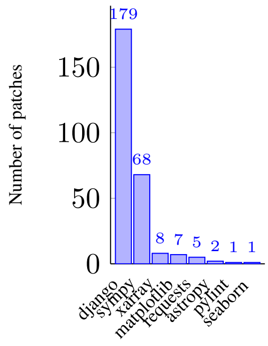
Listing 8: Three selected unsuccessfully parsed test cases in `django__django-15278` (SWE-Bench Verified).

`primary key."` is incorrectly identified as the name of a test case.

We applied the improved parser to correct the annotation data for `PASS_TO_PASS` and `FAIL_TO_PASS` in SWE-Bench Lite and SWE-Bench Verified. This update affected 54.7% (164/300) of the instances in SWE-Bench Lite and 54.2% (271/500) of the instances in SWE-Bench Verified. The distribution of instances with erroneous annotations is shown in Figure 5, with `django` and `sympy` accounting for the majority of annotation errors. We evaluated the generated patches using the improved parser with updated annotations, finding that some patches originally marked as passed were actually erroneous. 64 erroneous patches generated by coding agents were incorrectly evaluated as passed in the original SWE-Bench Lite. Similarly, 79 erroneous patches were incorrectly evaluated as passed in the original SWE-Bench Verified.



(a) Erroneous annotations in SWE-Bench Lite



(b) Erroneous annotations in SWE-Bench Verified

Figure 5: Distribution of erroneous annotations in SWE-Bench

Answer to RQ2 *The original parser in SWE-Bench failed to parse many test cases, resulting in incorrect evaluation outcomes. The improved parser corrected 54.7% of annotations in SWE-Bench Lite and 54.2% in SWE-Bench Verified, uncovering 64 erroneous patches in SWE-Bench Lite and 79 in SWE-Bench Verified that were incorrectly classified as passed in the original SWE-Bench evaluation pipeline.*

4.4 Update to the SWE-Bench Leaderboard

To enhance the accuracy of the SWE-Bench leaderboard, we added the augmented test cases generated by UTBoost to SWE-Bench and replaced the original SWE-Bench parser with the improved one. In total, we identified 176 erroneous patches in SWE-Bench Lite and 169 in SWE-Bench Verified that were incorrectly evaluated as passed in the original SWE-Bench. We recalculated the coding agents' scores on SWE-Bench Lite and SWE-Bench Verified and updated their respective leaderboards accordingly.

Significant ranking changes were observed in both SWE-Bench Lite and SWE-Bench Verified. For example, in the original SWE-Bench Verified leaderboard, `Amazon-Q-Developer-Agent (v20241202-dev)` ranked 1st with a pass@1 rate of 55%, while `devlo` ranked 2nd with 54.2%. After the update, both agents share the 1st rank with a pass@1 rate of 53.6%. This shift occurred because seven patches from `Amazon-Q-Developer-Agent` (`v20241202-dev`) were identified as erroneous, compared to only three patches from `devlo`. Overall, 40.9% (18/44) of rankings in SWE-Bench Lite and 24.4% (11/45) in SWE-Bench Verified changed. The updated leaderboards for SWE-Bench Lite and SWE-Bench Verified are provided in Appendix A.

Answer to RQ3 *The insufficient test cases and erroneous annotations in SWE-Bench greatly affect the accuracy of the SWE-Bench leaderboard. After updating the leaderboard with the augmented test cases and the improved parser, we observe significant ranking changes, affecting 40.9% of entries in SWE-Bench Lite and 24.4% in SWE-Bench Verified.*

5 Related Works

5.1 Code Generation Benchmark

Several benchmarks (Chen et al., 2021; Austin et al., 2021) evaluate the code generation ability of LLMs by letting them generate a function or class to solve a problem. The input for these benchmarks is always straightforward; for example, given an unordered list, the question is to write an algorithm to get the ordered one. EvalPlus (Liu et al., 2024) adds the augmented test cases via type-aware mutation, such as removing/repeating a random list item. However, SWE-Bench's test case is more complicated than MBPP and HumanEval because it may involve modification in multiple locations/files, and there are many dependencies to deal with, e.g., importing some function from another package. Therefore, we can not directly apply EvalPlus to add test cases for SWE-Bench since it does not know the locations to add the test cases. To address the challenges of generating augmented test cases for SWE-Bench, we propose UTGenerator to consider the codebase and dependencies while generating the test cases.

5.2 Robustness of SWE-Bench

The robustness of the coding benchmarks is significant to the rigorous evaluation of the coding agents' ability. To this end, several works are proposed to enhance or discuss the robustness of SWE-Bench. Aleithan et al. (Aleithan et al., 2024), Chen and Jiang (Chen and Jiang, 2024) manually check the passed generated patches of some coding agents and discovered that some of the passed patches are incorrect fixes. Manually checking these takes lots of time. Thus, Aleithan et al. (Aleithan et al., 2024) only check `SWE-Agent+GPT-4`, and Chen et al. (Chen and Jiang, 2024) select top-10 agents for evaluation. Comparatively, UTBoost adds the augmented test cases, which are reusable and easy to be applied for evaluating all the submitted coding agents.

OpenAI and SWE-Bench's team let 93 experienced engineers manually filter out a subset of 500 instances with high quality and named it SWE-Bench Verified (OpenAI, 2024). However, it is difficult even for experienced engineers to judge if the test case for an issue is comprehensive. UTBoost has identified 26 instances with insufficient test cases in SWE-Bench Verified and generated the augmented test cases for them, demonstrating the effectiveness of utilizing the LLM-based methods to achieve comprehensive testing. Based on our knowledge, UTBoost is the first method to address the challenge of insufficient test cases in SWE-Bench, while the existing methods only reveal this problem. Additionally, we are the first to discuss the impact of parsing errors by the original SWE-Bench parser, which also impedes rigorous evaluation of SWE-Bench.

6 Conclusion

In this paper, we introduce UTBoost, a framework for augmenting test cases in real-world Python projects using intramorphic testing. Built on UTGenerator, UTBoost generates dependency-aware test cases by analyzing codebases and issue descriptions. It is the first approach to address insufficient test cases in SWE-Bench, identifying 26 instances in SWE-Bench Verified that were overlooked despite a manual review by 93 engineers.

Furthermore, we improved the SWE-Bench parser, uncovering errors in over 54% of annotations in both SWE-Bench Lite and SWE-Bench Verified. Using the augmented test cases and improved parser, we identified 176 erroneous patches in SWE-Bench Lite and 169 in SWE-Bench Verified that were incorrectly evaluated as passed in the original SWE-Bench, leading to 40.9% ranking changes in SWE-Bench Lite and 24.4% in SWE-Bench Verified. UTBoost pioneers the application of intramorphic testing to evaluate open-sourced software systems and provides a versatile framework adaptable to real-world projects in other programming languages.

7 Limitations

UTBoost augments the test cases of SWE-Bench to achieve robust evaluation. The main limitation of UTBoost is that it can only generate test cases for those instances that at least one coding agent has resolved because UTBoost needs to cross-validate the gold patch and generated patches that pass the original SWE-Bench test cases. Currently, the submitted coding agents have resolved 74.6% (224/300) and 81.6% of the test cases in SWE-Bench Lite and SWE-Bench-Verified, and UTBoost can generate test cases for these instances.

The limitations of our experiments can be summarized in two aspects. First, we only utilize GPT-4o (gpt-4o-2024-08-06) in UTGenerator. However, integrating other LLM APIs into UTGenerator is straightforward and could potentially enhance the generation of diverse test cases. Second, the architecture of UTGenerator presents another limitation. Generating test cases and generating patches to address issues are inherently similar tasks, both requiring the identification of relevant locations followed by code generation. This similarity suggests that adapting a coding agent into a test case generator agent is feasible. UTGenerator adopts a simplified architecture inspired by Agentless (Xia et al., 2024), which eliminates the need for the LLM to plan future actions or interact with complex tools. Currently, there are 48 coding agents and 46 coding agents submitted to the SWE-Bench leaderboard. Incorporating test case generators with alternative architectures could further diversify the augmented test cases.

8 Ethics Statement

To mitigate the risk of LLMs generating harmful test cases that could compromise software systems, we conduct a thorough manual review of each test case produced by UTBoost. This ensures that no harmful code is introduced before integration into the SWE-Bench. In our paper, we use ChatGPT to check the grammar.

References

Aider. 2024. Aider is ai pair programming in your terminal).

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992*.

Amazon. 2024. Amazon q developer: The most capable generative ai–powered assistant for software development.

Anonymous. 2024. A toolkit for utboost.

Anthropic. 2024. Ai research and products that put safety at the frontier.

AppMap. 2024. Enterprise ready ai dev assistant: Let navie be your guide to ai-powered software development.

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

Bytedance. 2024. Code and innovate faster with ai.

Jiayi Geng Carlos E. Jimenez, John Yang. 2024. Swe-bench lite.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. Coder: Issue resolving with multi-agent and task graphs. *Preprint*, arXiv:2406.01304.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Zhi Chen and Lingxiao Jiang. 2024. Evaluating software development agents: Patch patterns, code quality, and issue complexity in real-world github scenarios. *arXiv preprint arXiv:2410.12468*.

Composio. 2024. Composio swebench-agent-v2: Oss sota software engineering assistant.

devlo. 2024. Say hello to your ai-developer teammate.

Globant. 2024. Introducing the power of globant ai agents.

Gru. 2024. We build ai developers.

All Hands. 2024. Openhands: Code less, make more.

Honeycomb. 2024. Honeycomb — bringing autonomy to software engineering.

IBM. 2024a. Agent-101: A software engineering agent for code assistance developed by ibm research).

IBM. 2024b. Ibm ai agent swe-1.0 (with open llms): A software engineering agent for code development.

Isoform. 2024. Isoform is your ai-driven integration engineer.

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

Emergent Labs. 2024a. A new era of code intelligence.

Engines Labs. 2024b. Ai software engineer to help your team ship faster).

Bin Lei, Yuchen Li, Yiming Zeng, Tao Ren, Yi Luo, Tianyu Shi, Zitian Gao, Zeyu Hu, Weitai Kang, and Qiuwu Chen. 2024. Infant agent: A tool-integrated, logic-driven agent with cost-effective api usage. *Preprint*, arXiv:2411.01114.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *arXiv preprint arXiv:2406.01422*.

MASAI. 2024. Masai: Modular architecture for software engineering ai agents).

NEBIUS. 2024. Leveraging training and search for better software engineering agents.

nFactorial AI. 2024. nfactorial ai: Empowering decision-making with ai-driven insights).

OpenAI. 2024. Introducing swe-bench verified.

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

Huy Nhat Phan, Tien N. Nguyen, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *Preprint*, arXiv:2409.16299.

Manuel Rigger and Zhendong Su. 2022. Intramorphic testing: A new approach to the test oracle problem. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 128–136.

SIMA. 2024. Sima (software intelligence: Multi-agents).

Solver. 2024. Solver: Self-driving software is here).

SuperAGI. 2024. Supercoder: Technology that builds technology.

EPAM Systems. 2024. Become an ai-first business to thrive in the next wave of disruption.

Moatless Tools. 2024. Moatless tools.

TURINTECH. 2024. Ai-driven automation for software excellence.

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

Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. 2024. Codeshell technical report. *Preprint*, arXiv:2403.15747.

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

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

A SWE-Bench Leaderboard

In Table 1, we present a comparison between the original and updated leaderboards for SWE-Bench Lite. Similarly, Table 2 shows the comparison for SWE-Bench Verified. Coding agents with ranking changes are highlighted in both tables. Notably, some coding agents share the same rank, such as `Amazon Q Developer Agent (v20241202-dev)` (Amazon, 2024) and `devlo` (devlo, 2024). Overall, there are 18 ranking changes in SWE-Bench Lite and 11 in SWE-Bench Verified. The original leaderboards for SWE-Bench Lite and SWE-Bench Verified correspond to the versions dated December 15, 2024. Since some coding agents do not provide their generated patches, we exclude them, resulting in 44 coding agents in SWE-Bench Lite and 45 in SWE-Bench Verified.

Table 1: Comparison between the original and updated SWE-Bench Lite leaderboard (We highlight the background for agents with ranking changes and the text for agents whose percentage of resolved cases has changed).

Leadearboard	Original SWE-Bench Lite leaderboard		Updated SWE-Bench Lite leaderboard	
Rank	Coding agent	% Resolved	Coding agent	% Resolved
1	Globant Code Fixer Agent (Globant, 2024)	48.33	Globant Code Fixer Agent (Globant, 2024)	46.33
2	devlo (devlo, 2024)	47.33	devlo (devlo, 2024)	46.00
3	OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022) (Hands, 2024)	41.67	OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022) (Hands, 2024)	40.67
4	Composio SWE-Kit (2024-10-30) (Composio, 2024)	41.00	Composio SWE-Kit (2024-10-30) (Composio, 2024)	39.00
5	Agentless-1.5 + Claude-3.5 Sonnet (20241022) (Xia et al., 2024)	40.67	Agentless-1.5 + Claude-3.5 Sonnet (20241022) (Xia et al., 2024)	38.33
6	Bytedance MarsCode Agent (Bytedance, 2024)	39.33	Bytedance MarsCode Agent (Bytedance, 2024)	38.00
7	Moatless Tools + Claude 3.5 Sonnet (20241022) (Tools, 2024)	38.33	Honeycomb (Honeycomb, 2024)	37.67
8	Honeycomb (Honeycomb, 2024)	38.33	Moatless Tools + Claude 3.5 Sonnet (20241022) (Tools, 2024)	36.67
9	AppMap Navie v2 (AppMap, 2024)	36.00	AppMap Navie v2 (AppMap, 2024)	35.00
10	Gru(2024-08-11) (Gru, 2024)	35.67	Isoform (Isoform, 2024)	33.33
11	Isoform (Isoform, 2024)	35.00	Gru(2024-08-11) (Gru, 2024)	33.00
12	SuperCoder2.0 (SuperAGI, 2024)	34.00	SuperCoder2.0 (SuperAGI, 2024)	32.00
13	Alibaba Lingma Agent (Ma et al., 2024)	33.00	Alibaba Lingma Agent (Ma et al., 2024)	31.33
14	Agentless-1.5 + GPT 4o (2024-05-13) (Xia et al., 2024)	32.00	Agentless-1.5 + GPT 4o (2024-05-13) (Xia et al., 2024)	30.33
15	CodeShellTester + GPT 4o (2024-05-13) (Xie et al., 2024)	31.33	AutoCodeRover (v20240620) + GPT 4o (2024-05-13) (Zhang et al., 2024)	30.00
16	AutoCodeRover (v20240620) + GPT 4o (2024-05-13) (Zhang et al., 2024)	30.67	CodeShellTester + GPT 4o (2024-05-13) (Xie et al., 2024)	29.67
17	AIGCode Infant-Coder(2024-08-30) (Lei et al., 2024)	30.00	AIGCode Infant-Coder(2024-08-30) (Lei et al., 2024)	28.67
18	Amazon Q Developer Agent (v20240719-dev) (Amazon, 2024)	29.67	Amazon Q Developer Agent (v20240719-dev) (Amazon, 2024)	28.00
19	Agentless + RepoGraph + GPT-4o (Xia et al., 2024; Ouyang et al., 2024)	29.67	Agentless + RepoGraph + GPT-4o (Xia et al., 2024; Ouyang et al., 2024)	27.67
20	CodeR + GPT 4 (1106) (Chen et al., 2024)	28.33	CodeR + GPT 4 (1106) (Chen et al., 2024)	26.67
21	SIMA + GPT 4o (2024-05-13) (SIMA, 2024)	27.67	MASAI + GPT 4o (2024-05-13) (MASAI, 2024)	26.67
22	MASAI + GPT 4o (2024-05-13) (MASAI, 2024)	27.33	SIMA + GPT 4o (2024-05-13) (SIMA, 2024)	26.33
23	Agentless + GPT 4o (2024-05-13) (Xia et al., 2024)	27.33	Agentless + GPT 4o (2024-05-13)	25.33
24	Moatless Tools + Claude 3.5 Sonnet (Tools, 2024)	26.67	Aider + GPT 4o & Claude 3 Opus (Aider, 2024)	25.33
25	OpenHands + CodeAct v1.8 (Hands, 2024)	26.67	Moatless Tools + Claude 3.5 Sonnet (Tools, 2024)	25.33
26	IBM Research Agent-101 (IBM, 2024a)	26.67	HyperAgent (Phan et al., 2024)	25.00
27	Aider + GPT 4o & Claude 3 Opus (Aider, 2024)	26.33	IBM Research Agent-101 (IBM, 2024a)	25.00
28	HyperAgent (Phan et al., 2024)	25.33	OpenHands + CodeAct v1.8 (Hands, 2024)	24.33
29	Moatless Tools + GPT 4o (2024-05-13) (Tools, 2024)	24.67	Moatless Tools + GPT 4o (2024-05-13) (Tools, 2024)	24.00
30	IBM AI Agent SWE-1.0 (with open LLMs) (IBM, 2024b)	23.67	SWE-agent + Claude 3.5 Sonnet (Yang et al., 2024)	23.00
31	SWE-agent + Claude 3.5 Sonnet (Yang et al., 2024)	23.00	IBM AI Agent SWE-1.0 (with open LLMs) (IBM, 2024b)	22.33
32	AppMap Navie + GPT 4o (2024-05-13) (AppMap, 2024)	21.67	AppMap Navie + GPT 4o (2024-05-13) (AppMap, 2024)	20.33
33	Bytedance AutoSE (20240828) (Bytedance, 2024)	21.67	Bytedance AutoSE (20240828) (Bytedance, 2024)	19.67
34	Amazon Q Developer Agent (v20240430-dev) (Amazon, 2024)	20.33	Amazon Q Developer Agent (v20240430-dev) (Amazon, 2024)	19.33
35	AutoCodeRover (v20240408) + GPT 4 (0125) (Zhang et al., 2024)	19.00	AutoCodeRover (v20240408) + GPT 4 (0125) (Zhang et al., 2024)	18.33
36	SWE-agent + GPT 4o (2024-05-13) (Yang et al., 2024)	18.33	SWE-agent + GPT 4 (1106) (Yang et al., 2024)	17.33
37	SWE-agent + GPT 4 (1106) (Yang et al., 2024)	18.00	SWE-agent + GPT 4o (2024-05-13) (Yang et al., 2024)	17.00
38	SWE-agent + Claude 3 Opus (Yang et al., 2024)	11.67	SWE-agent + Claude 3 Opus (Yang et al., 2024)	11.67
39	RAG + Claude 3 Opus (Jimenez et al., 2024)	4.33	RAG + Claude 3 Opus (Jimenez et al., 2024)	4.33
40	RAG + Claude 2 (Jimenez et al., 2024)	3.00	RAG + Claude 2 (Jimenez et al., 2024)	3.00
41	RAG + GPT 4 (1106) (Jimenez et al., 2024)	2.67	RAG + GPT 4 (1106) (Jimenez et al., 2024)	2.33
42	RAG + SWE-Llama 7B (Jimenez et al., 2024)	1.33	RAG + SWE-Llama 7B (Jimenez et al., 2024)	1.00
43	RAG + SWE-Llama 13B (Jimenez et al., 2024)	1.00	RAG + SWE-Llama 13B (Jimenez et al., 2024)	1.00
44	RAG + ChatGPT 3.5 (Jimenez et al., 2024)	0.33	RAG + ChatGPT 3.5 (Jimenez et al., 2024)	0.33

Table 2: Comparison between the original and updated SWE-Bench Verified leaderboard (We highlight the background for agents with ranking changes and the text for agents whose percentage of resolved cases has changed).

Leadearboard	Original SWE-Bench Verifiied leaderboard		Updated SWE-Bench Verified leaderboard	
Rank	Coding agent	% Resolved	Coding agent	% Resolved
1	Amazon Q Developer Agent (v20241202-dev) (Amazon, 2024)	55.00	Amazon Q Developer Agent (v20241202-dev) (Amazon, 2024)	53.60
2	devlo (devlo, 2024)	54.20	devlo (devlo, 2024)	53.60
3	OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022) (Hands, 2024)	53.00	OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022) (Hands, 2024)	51.80
4	Engine Labs (2024-11-25) (Labs, 2024b)	51.80	Engine Labs (2024-11-25) (Labs, 2024b)	50.80
5	Agentless-1.5 + Claude-3.5 Sonnet (20241022) (Xia et al., 2024)	50.80	Agentless-1.5 + Claude-3.5 Sonnet (20241022) (Xia et al., 2024)	49.60
6	Solver (2024-10-28) (Solver, 2024)	50.00	Bytedance MarsCode Agent (Bytedance, 2024)	49.40
7	Bytedance MarsCode Agent (Bytedance, 2024)	50.00	Solver (2024-10-28) (Solver, 2024)	49.20
8	nFactorial (2024-11-05) (nFactorial AI, 2024)	49.20	nFactorial (2024-11-05) (nFactorial AI, 2024)	48.40
9	Tools + Claude 3.5 Sonnet (2024-10-22) (Anthropic, 2024)	49.00	Tools + Claude 3.5 Sonnet (2024-10-22) (Anthropic, 2024)	48.20
10	Composio SWE-Kit (2024-10-25) (Composio, 2024)	48.60	Composio SWE-Kit (2024-10-25) (Composio, 2024)	47.40
11	AppMap Navie v2 (AppMap, 2024)	47.20	AppMap Navie v2 (AppMap, 2024)	46.40
12	Emergent E1 (v2024-10-12) (Labs, 2024a)	46.60	Emergent E1 (v2024-10-12) (Labs, 2024a)	45.60
13	AutoCodeRover-v2.0 (Claude-3.5-Sonnet-20241022) (Zhang et al., 2024)	46.20	AutoCodeRover-v2.0 (Claude-3.5-Sonnet-20241022) (Zhang et al., 2024)	45.40
14	Solver (2024-09-12) (Solver, 2024)	45.40	Solver (2024-09-12) (Solver, 2024)	44.80
15	Gru(2024-08-24) (Gru, 2024)	45.20	Gru(2024-08-24) (Gru, 2024)	43.8
16	Solver (2024-09-12) (Solver, 2024)	43.60	Solver (2024-09-12) (Solver, 2024)	42.80
17	nFactorial (2024-10-30) (nFactorial AI, 2024)	41.60	nFactorial (2024-10-30) (nFactorial AI, 2024)	40.60
18	Nebius AI Qwen 2.5 72B Generator + LLama 3.1 70B Critic (NEBIUS, 2024)	40.60	Honeycomb (Honeycomb, 2024)	40.20
19	Tools + Claude 3.5 Haiku (Anthropic, 2024)	40.60	Tools + Claude 3.5 Haiku (Anthropic, 2024)	40.00
20	Honeycomb (Honeycomb, 2024)	40.60	Nebius AI Qwen 2.5 72B Generator + LLama 3.1 70B Critic (NEBIUS, 2024)	39.60
21	Composio SWEkit + Claude 3.5 Sonnet (2024-10-16) (Composio, 2024)	40.60	Composio SWEkit + Claude 3.5 Sonnet (2024-10-16) (Anthropic, 2024)	39.00
22	EPAM AI/Run Developer Agent v20241029 + Anthopic Claude 3.5 Sonnet (Systems, 2024)	39.60	EPAM AI/Run Developer Agent v20241029 + Anthopic Claude 3.5 Sonnet (Systems, 2024)	39.00
23	Amazon Q Developer Agent (v20240719-dev) (Amazon, 2024)	38.80	Agentless-1.5 + GPT 4o (2024-05-13) (Xia et al., 2024)	38.40
24	Agentless-1.5 + GPT 4o (2024-05-13) (Xia et al., 2024)	38.80	Amazon Q Developer Agent (v20240719-dev) (Amazon, 2024)	38.00
25	AutoCodeRover (v20240620) + GPT 4o (2024-05-13) (Zhang et al., 2024)	38.40	AutoCodeRover (v20240620) + GPT 4o (2024-05-13) (Zhang et al., 2024)	37.80
26	Artemis Agent v1 (2024-11-20) (TURINTECH, 2024)	32.00	Artemis Agent v1 (2024-11-20) (TURINTECH, 2024)	30.80
27	nFactorial (2024-10-07) (nFactorial AI, 2024)	31.60	nFactorial (2024-10-07) (nFactorial AI, 2024)	30.80
28	Lingma Agent + Lingma SWE-GPT 72b (v0925) (Ma et al., 2024)	28.80	Lingma Agent + Lingma SWE-GPT 72b (v0925) (Ma et al., 2024)	27.20
29	EPAM AI/Run Developer Agent + GPT4o (Systems, 2024)	27.00	EPAM AI/Run Developer Agent + GPT4o (Systems, 2024)	26.80
30	AppMap Navie + GPT 4o (2024-05-13) (AppMap, 2024)	26.20	nFactorial (2024-10-01) (nFactorial AI, 2024)	25.60
31	nFactorial (2024-10-01) (nFactorial AI, 2024)	25.80	AppMap Navie + GPT 4o (2024-05-13) (AppMap, 2024)	25.20
32	Amazon Q Developer Agent (v20240430-dev) (Amazon, 2024)	25.60	Lingma Agent + Lingma SWE-GPT 72b (v0918) (Ma et al., 2024)	24.80
33	Lingma Agent + Lingma SWE-GPT 72b (v0918) (Ma et al., 2024)	25.00	Amazon Q Developer Agent (v20240430-dev) (Amazon, 2024)	24.80
34	EPAM AI/Run Developer Agent + GPT4o (Systems, 2024)	24.00	EPAM AI/Run Developer Agent + GPT4o (Systems, 2024)	23.80
35	SWE-agent + GPT 4o (2024-05-13) (Yang et al., 2024)	23.20	SWE-agent + GPT 4o (2024-05-13) (Yang et al., 2024)	22.40
36	SWE-agent + GPT 4 (1106) (Yang et al., 2024)	22.40	SWE-agent + GPT 4 (1106) (Yang et al., 2024)	21.80
37	SWE-agent + Claude 3 Opus (Yang et al., 2024)	18.20	SWE-agent + Claude 3 Opus (Yang et al., 2024)	17.80
38	Lingma Agent + Lingma SWE-GPT 7b (v0925) (Ma et al., 2024)	18.20	Lingma Agent + Lingma SWE-GPT 7b (v0925) (Ma et al., 2024)	17.80
39	Lingma Agent + Lingma SWE-GPT 7b (v0918) (Ma et al., 2024)	10.20	Lingma Agent + Lingma SWE-GPT 7b (v0918)	9.60
40	RAG + Claude 3 Opus (Jimenez et al., 2024)	7.00	RAG + Claude 3 Opus (Jimenez et al., 2024)	7.00
41	RAG + Claude 2 (Jimenez et al., 2024)	4.40	RAG + Claude 2 (Jimenez et al., 2024)	4.20
42	RAG + GPT 4 (1106) (Jimenez et al., 2024)	2.80	RAG + GPT 4 (1106) (Jimenez et al., 2024)	2.60
43	RAG + SWE-Llama 7B (Jimenez et al., 2024)	1.40	RAG + SWE-Llama 13B (Jimenez et al., 2024)	1.00
44	RAG + SWE-Llama 13B (Jimenez et al., 2024)	1.20	RAG + SWE-Llama 7B (Jimenez et al., 2024)	0.80
45	RAG + ChatGPT 3.5 (Jimenez et al., 2024)	0.40	RAG + ChatGPT 3.5 (Jimenez et al., 2024)	0.40