SWE-BENCH+: ENHANCED CODING BENCHMARK FOR LLMS

REEM ALEITHAN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

MARCH 2025

Abstract

Large Language Models (LLMs) in Software Engineering (SE) can offer valuable assistance for coding tasks. To facilitate a rigorous evaluation of LLMs in practical coding contexts, Carlos et al. introduced the *SWE-bench* dataset, which comprises 2,294 real-world GitHub issues. Several impressive LLM-based toolkits have recently been developed and evaluated on this dataset. However, a systematic evaluation of the quality of *SWE-bench* remains missing.

In this thesis, we address this gap by presenting an empirical analysis of the *SWE-bench* dataset. We manually screen instances where *SWE-Agent + GPT-4* successfully resolved the issues by comparing model-generated patches with developer-written pull requests.

Our analysis reveals two critical issues: (1) 33.47% of patches have solution leakage, where the fix is directly or indirectly revealed in the issue report or comments; and (2) 24.70% of successful patches are suspicious due to weak test cases that fail to detect incorrect, incomplete, or irrelevant fixes. Filtering out these problematic instances drops *SWE-Agent + GPT-4*'s resolution rate from 12.47% to 4.58%.

Motivated by these findings, we propose *SWE-Bench+*, a refined version of the benchmark using two LLM-based tools: *SoluLeakDetector* to identify solution-leak issues and *TestEnhancer* to reduce weak test cases. *SWE-Bench+* identifies solution-leak issues with 86% accuracy and reduces suspicious patches by 19%.

To reduce the risk of potential data leakage, we collect a new set of post-cutoff GitHub issues. We then evaluate models on this dataset, observing a consistent performance drop across all models. This highlights the impact of solution leakage and weak tests in inflating resolution rates in current benchmarks.

Acknowledgements

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Song Wang, for his invaluable support in conceptualizing the core ideas of this work, guiding me through various research challenges, and providing continuous support throughout the research process.

I am also deeply thankful to Prof. Gias Uddin, a member of my thesis supervision committee, for his time, thoughtful feedback, and constructive suggestions, which greatly improved the quality of this thesis.

Table of Contents

List of Tables

List of Figures

Abbreviations

Table 1: Abbreviations List

Abbreviation	**Definition**
AI/ML	Artificial Intelligence / Machine Learning
SE	Software Engineering
APR	Automated Program Repair
DEI	Diversity Empowers Intelligence
FTP	Fail-To-Pass (test case)
LLM	Large Language Model
PR	Pull Request
PTP	Pass-To-Pass (test case)
RAG	Retrieval-Augmented Generation
SOTA	State-of-the-Art

Chapter 1

Introduction

1.1 Background and Context

SWE-bench is an evaluation framework designed to assess large language models (LLMs) in code generation and issue resolution. Unlike traditional benchmarks, it incorporates complex, real-world GitHub issues, providing an authentic dataset for evaluating LLM capabilities in software development. This real-world focus has generated significant interest in the software engineering community [1, 2, 3, 4, 5, 6, 7, 8]. *SWE-bench* is sourced from the 12 widely used Python repositories. The dataset includes issues and pull requests (PRs) selected based on their test cases and successful PR merges [1, 6, 4]. These issues cover various tasks such as bug fixing, feature enhancements, and code refactoring, requiring an in-depth understanding of project contexts and dependencies [2, 3, 8, 4, 5]. The *SWE-bench* dataset comprises 2,294 complex GitHub issues [1]. Given an issue description and PR reference, an LLM is tasked

to modify the corresponding codebase to resolve the issues. These issues include bug reports and feature requests, with PRs containing developer-written code changes and test cases. Two *SWE-bench* variants have been developed: *SWE-bench Lite*[1] and *SWE-bench Verified*[2]. *SWE-bench Lite* focuses on 300 bug-fixing issues with lower evaluation costs and increased accessibility, while *SWE-bench Verified* includes 500 carefully validated issues with clear descriptions and robust test cases.

A key contribution of *SWE-bench* is its evaluation of LLMs to resolve issues within a live codebase [1, 4, 7, 8]. Unlike benchmarks that assess isolated code snippets, *SWE-bench* evaluates an LLMs' ability to navigate and modify an existing codebase while ensuring correctness, handling dependencies, and avoiding new errors [1, 3, 6, 7]. Academic and industry researchers have widely used *SWE-bench* and its variants to evaluate LLMs' coding abilities [2, 3, 4, 5, 6, 7, 8]. These approaches involve reasoning about bugs, analyzing root causes, formulating fixes, and implementing patches. Within a year, resolution rates on *SWE-bench Full* have improved from 0.17% (*RAG+GPT3.5*) to 33.83% (*SWE-Agent 1.0*). However, LLMs still struggle to autonomously generate reliable patches for complex software issues, emphasizing the need for further advancements in AI-driven code generation.

1.2 Problem Statement

Despite the promising emergence of *SWE-bench* as an evaluation benchmark and the increasing number of LLMs with higher resolution rates on the *SWE-bench Leaderboard*, there remains

[1]https://www.swebench.com/lite.html
[2]https://openai.com/index/introducing-swe-bench-verified/

a critical question: ***Are the LLMs really resolving the issues in SWE-bench?*** *SWE-bench* is becoming increasingly popular among developers as a benchmark for evaluating LLMs in the context of code generation and issue resolution. However, no comprehensive empirical study has been conducted on the *SWE-bench* dataset to validate its efficiency, precision and general reliability. Due to the lack of validation, the effectiveness of LLMs in solving complex real-world software problems becomes questionable.

Since there is no formal evaluation to verify the credibility of *SWE-bench*, it cannot be considered a fully reliable benchmark for assessing LLM capabilities in resolving GitHub issues. This underscores the need for a rigorous study to determine whether the reported improvements on the leaderboard truly reflect advancements in LLM performance or whether they are influenced by inherent limitations within the benchmark itself.

1.3 Objective

This study has three main objectives. The first objective is to carry out an empirical study of state-of-the-art (SOTA) LLMs using *SWE-Bench Full* that explores: (1) the quality of *SWE-bench* issues with a focus on the adequacy of the test cases used to validate patches, (2) the quality of patches generated by LLMs to fix these issues, and (3) the impact of potential data leakage on the performance of LLMs. This includes evaluating whether prior exposure to benchmark instances influences the reported results.

The second objective is to present *SWE-Bench+*, a framework that improves the quality of *SWE-Bench* by filtering solution-leak instances and strengthening test cases to eliminate

suspicious patches. This is achieved through two novel techniques: *SoluLeakDetector* and *TestEnhancer*.

The third objective is to collect new GitHub issues created beyond the training cut-off dates of the evaluated LLMs and assess their performance on this leakage-free dataset, providing a more accurate picture of real-world generalization.

1.4 Study Overview

Our thesis follows a structured approach, beginning with an empirical study on the *SWE-bench* dataset to assess its reliability by comparing developer-generated patches with LLM-generated patches. We then analyzed key patterns in the generated patches and identified concerns such as dataset reliability and test case adequacy. Then, we proposed novel approaches to improve the quality of data and the effectiveness of test cases. Finally, we gathered new instances from recent GitHub issues created after the training cut-off dates and evaluated the performance of various LLMs on this new dataset.

Firstly, we selected instances that passed all tests and were marked as resolved by *SWE-Agent+GPT-4*, as it was the top-performing open-source model on the *SWE-bench* leaderboard at the time of our study. To confirm the resolution of selected instances by the model, we reviewed the execution and test logs published by the *SWE-bench* to shortlist the instances that passed all the tests after the generated patches were applied.

Secondly, we conducted a patch validation study, comparing patches generated by developers (through pull requests) with model-generated patches. This analysis reviewed patch

intent, changed lines and files, and code correctness. We manually reviewed the differences between these patches to determine whether the model-generated solutions were correct, incomplete, or unrelated to the original issue.

Thirdly, we summarized five problematic patterns found in the model-generated patches, revealing two fundamental issues in the benchmark: (1) solution leaks leading to the models' "cheating" instead of resolving the issues, and (2) weak test cases, where the generated patches are either irrelevant, incomplete, or incorrect and yet still pass all the tests. Hence, the test cases are too permissive, leading to higher resolution rates for the LLMs than what is truly deserved.

Finally, to address these issues, we proposed two new approaches. *SoluLeakDetector*, an LLM-based technique for detecting and filtering out instances with solution leakage. This technique ensures a cleaner dataset to evaluate whether the models are truly resolving the issues rather than simply copying the provided solutions. *TestEnhancer*, an LLM-based technique to generate stronger test cases, preventing weak tests from allowing incorrect, incomplete, or irrelevant fixes to pass. This approach ensures that model-generated patches undergo more rigorous validation. *TestEnhancer* is designed as a diagnostic tool with a distinct purpose from general-purpose test generators such as EvalPlus [9]. Rather than producing more tests for generated code, it focuses on exposing inaccuracies and incompleteness in the existing SWE-Bench test suites. It leverages key metadata, including the gold patch, modified functions, and contextual diffs, to generate targeted FAIL-TO-PASS and PASS-TO-PASS tests. These tests are crafted to reveal silent failure and invalid patches that pass due to

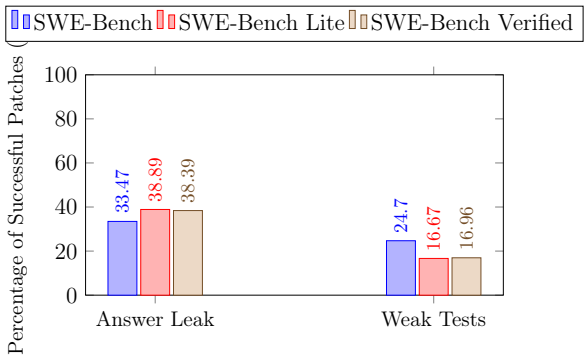
weak or incomplete test coverage. Unlike EvalPlus, which focuses on improving test suite completeness, *TestEnhancer* identifies structural flaws in the benchmark itself that inflate resolution rates.

By implementing the above enhancements, we refined the *SWE-bench* dataset into a more reliable benchmark for evaluating LLMs in software engineering tasks.

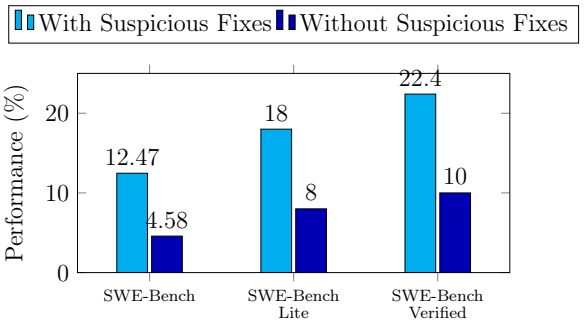
To address issues related to the impact of data leakage, we collected a new dataset using the same collection, filtering, and preprocessing techniques as *SWE-bench*. The newly collected dataset consists of GitHub issues created after October 2023, which is the training cut-off date for all the models we tested. Then, we applied *SoluLeakDetector* to identify and remove instances with solution leakage. Finally, we evaluated the performance of different LLMs on this dataset, ensuring that the reported results are not influenced by the information learned previously.

1.5 Contributions

Our study identifies and addresses key limitations in the existing benchmark evaluations for automated program repair, particularly in *SWE-bench*. Current resolution rates for model-generated patches are determined solely based on test case results. However, test cases often lack comprehensive coverage, failing to capture edge cases and unintended side effects. This reliance on test case outcomes alone can result in misleading success rates, where patches appear correct but do not truly resolve the underlying issue.



(a) Answer Leak vs Weak Tests across Datasets



(b) Performance of SWE-Agent + GPT-4 across datasets.

Figure 1.1: Comparison of performance metrics and patterns across *SWE-bench* datasets

Our analysis highlights the fundamental weaknesses in test-based evaluation and proposes improvements to enhance benchmarking reliability. In contrast, developers have a more comprehensive understanding of the problem context and can produce semantically correct patches that align with the intended solution. This level of reasoning is challenging to evaluate, making it difficult to determine whether LLM-generated fixes genuinely reflect the correct approach.

Additionally, *SWE-bench* defines an issue as resolved if the associated test cases pass. However, our study revealed that such an approach cannot guarantee the semantic correctness or completeness of the fix. Without incorporating human code reviews or qualitative assessments, the benchmark risks overestimating the real-world effectiveness of LLM-generated patches.

As part of our contribution, we analyzed 251 patches generated by *SWE-Agent+GPT-4* and identified six distinct patterns. These patterns broadly fall into two categories: *suspicious*

fixes and *correct fixes*. Our findings reveal that suspicious fixes account for 58.17% (146) of the patches, with two dominant failure modes:

- **Answer Leak.** In 33.47% of resolved instances, the solutions were either explicitly stated or subtly implied within the issue reports or comments. Some issue descriptions directly include the correct code for fixing the reported bug, while others provide strong hints guiding the solution.

- **Weak Tests.** In 24.70% of the resolved instances, the model-generated changes were incorrect, incomplete, or applied to unrelated files or functions compared to the gold patch. Despite these discrepancies, the modifications still passed the test cases, suggesting that the test suites lacked the necessary rigor to catch such errors.

Given these findings, we classify fixes associated with answer leakage and weak tests as *suspicious fixes*. Figure 1.1a illustrates the distribution of these failure cases across *SWE-Bench Full*, *SWE-Bench Lite*, and *SWE-Bench Verified*. Figure 1.1b further demonstrates that once these suspicious fixes are filtered out, the correct resolution rate of *SWE-Agent+GPT-4* is significantly lower than initially reported, underscoring the need for more robust evaluation methods.

Furthermore, we evaluated LLMs' performance on a newly collected dataset of GitHub issues created after the models' training cut-off dates to eliminate the risk of data leakage. Our results show a significant inflation in prior resolution rates. For example, *SWE-Agent + GPT-4* declined from 3.97% to 1.99%. Similarly, *SWE-RAG + GPT-4*, *SWE-RAG +*

GPT-3.5, and *AutoCodeRover + GPT-4o* achieved resolution rates of only 0.73%, 1.99%, and 3.83%, respectively.

These findings underscore the substantial impact of data leakage on LLM performance and raise concerns about the extent to which models rely on prior knowledge to generate seemingly correct solutions.

1.6 Significance of the Study

This study aims to validate the reliability of *SWE-bench* as an evaluation benchmark for assessing the ability of LLMs to generate code patches for real-world software issues autonomously. Without proper validation, *SWE-bench* cannot serve as an accurate measure of LLM effectiveness in this domain. In particular, data leakage and weak tests raise concerns about the validity of the benchmark's reported performance.

The discrepancies identified in this study highlight the critical need to verify evaluation benchmarks before relying on them to assess LLMs and draw conclusions about their capabilities. By addressing these concerns, this study contributes to improving the trustworthiness of LLM-based evaluations and strengthening their application in software engineering tasks.

1.7 Thesis Organization

The remainder of this thesis is divided into different chapters. Chapter 2 provides a comprehensive literature review, highlighting state-of-the-art prior work on large language models in

software engineering, benchmark datasets, and evaluation methods. Chapter 3 describes the research design, including the methodology, analysis, and experimental results. Chapter 4 discusses threats to validity and examines the limitations of our study. Finally, Chapter 5 concludes the thesis and outlines directions for future work.

1.8 Data Availability

We release our experiments' dataset and source code to enable other researchers to replicate and extend our study in [10, 11].

Chapter 2

Literature Review

2.1 LLM for Software Engineering

Large Language Models (LLMs) have emerged as powerful tools and demonstrated impressive capabilities in various software engineering tasks, including code generation [12, 13, 14, 15, 16], program repair [17, 18, 19], and bug detection [20, 21]. The development of code generation benchmarks has been crucial for evaluating LLM performance. There are several comprehensive studies that have explored LLM applications across various software engineering domains [22, 23], delved into natural language to code generation [24], and analyzed the evolution and performance of code LLMs across different tasks [25].

2.1.1 Code Generation

The rise of LLMs in recent years has led to impressive progress in code generation. In particular, a number of sophisticated models have been developed to generate code efficiently based on user-provided specifications. One notable example is *CodeT5+* [26], a powerful encoder-decoder LLM family introduced by Wang et al., designed to tackle a wide range of software engineering tasks, including code generation, summarization, and translation. *CodeT5+* is especially effective at translating natural language descriptions into functional code and completing partial code snippets.

Building on this direction, Shen et al. introduced *PanGu-Coder2* [27], which incorporates ranking-based feedback to enhance LLM performance in code generation. It leverages feedback from test cases to rank multiple generated solutions and prioritize higher-quality outputs.

In parallel, Xu et al. proposed an open-source alternative, *CodeGen* [28], a family of decoder-only models designed to generate high-level code from natural language in multiple programming languages, including Python, Go, and Java. CodeGen employs a transformer-based architecture and is sequentially pre-trained on diverse datasets such as The Pile, BigQuery, and BigPython to improve its generalization and multilingual capabilities.

Around the same time, OpenAI introduced *Codex* [14, 29], a decoder-only model fine-tuned from *GPT-3* on a large corpus of public GitHub code. Unlike CodeGen, which prioritized open-source accessibility, *Codex* focused on practical usability and became the foundation of GitHub Copilot. It processes natural language prompts through autoregressive

token generation and can produce multiple candidate outputs, ranked using top-k sampling techniques to select the highest-quality code.

Building on the foundation laid by *Codex*, *ChatGPT* [30] was developed with a focus on instruction-based learning, enhanced through continual human feedback and reinforcement learning. This approach made the model more interactive and user-focused. *ChatGPT* is based on the *GPT-3.5* and *GPT-4* architectures, and it allows users to iteratively refine the output, leading to a more user requirements focus. Similarly, Meta AI developed *Code Llama* [31], an open-source LLM specialized in code generation and understanding. Built by fine-tuning *LLaMA 2*, it supports multiple programming languages and is released in various sizes (7B, 13B, and 34B) to accommodate different use cases, resource constraints, and performance needs. Given this flexibility and accessibility, it provides a strong and practical solution for a wide range of software engineering tasks.

2.1.2 Program Repair

Traditionally, automated program repair (APR) techniques relied on four prominent approaches: heuristic-based methods, which use genetic programming to explore the patch space [32, 33]; constraint-based systems, which formulate repair as a constraint-solving task [34]; pattern-based approaches, which apply predefined templates to fix known bug types [35]; and learning-based models, which treat patch generation as a translation task using deep learning [36, 37, 38].

With the recent advancements in LLMs, research has increasingly shifted toward exploring their effectiveness in program repair, often surpassing traditional techniques, especially when fine-tuned on domain-specific data [39].

For example, *AlphaRepair* [40] employed a zero-shot learning approach that outperformed conventional APR tools. *AlphaRepair* utilizes a pre-trained LLM to generate code fixes by interpreting the surrounding code context and analyzing error messages. This allows the solution to scale across different programming languages without requiring additional domain-specific training. Compared to search-based techniques, it can address a broader range of software defects with greater flexibility and accuracy.

Building on this, *RepairAgent* [41] expanded the scope of LLM-based APR by introducing an autonomous, multi-turn repair agent. This agent interacts closely with the codebase, looking for patterns, context, and relevant information to aid in the repair process. It also applies test cases to validate patch correctness. Furthermore, *RepairAgent* uses a dynamic prompt updating mechanism to support adaptive decision-making and employs a finite-state machine to ensure the generated repairs are contextually appropriate. This multi-stage design reflects how human developers reason through complex bug fixes.

GIANTRepair [42] further advanced this direction by combining LLMs with traditional program analysis. It introduces a hybrid pipeline that generates patch skeletons and instantiates them using contextual information from the codebase. This results in patches that are not only correct but also semantically consistent with the surrounding code, offering a more precise and robust approach to repair.

Recent work has focused on developing more autonomous tools to reduce costly human involvement and push the boundaries of automated program repair.

A prominent example is *SWE-agent*, developed by Wang et al. [43], an autonomous framework that uses a user-selected LLM as its backbone to fix software defects. Designed to operate on real-world GitHub repositories, *SWE-agent* behaves like a human developer in its repair process. The framework follows four main steps to generate and validate a fix: (1) planning a strategy based on the issue and test failures, (2) editing the identified buggy code, (3) running tests to evaluate correctness, and (4) iterating through these steps until the bug is successfully resolved. This structured and self-directed approach makes *SWE-agent* one of the earliest and most effective LLM-based agents for realistic program repair tasks.

In a similar direction, Yu et al. [44] introduced *AutoCodeRover*, a context-aware, retrieval-augmented framework designed for program repair in multi-file, complex issues. The framework performs task decomposition to break down complex bugs into file-specific problems and solutions, leveraging a user-selected LLM backbone such as GPT models. It retrieves relevant information and code segments from the software project to address issues at the file level. Additionally, *AutoCodeRover* employs iterative patch validation by applying test cases to ensure that the generated fixes are correct.

2.1.3 Bug Detection

Traditionally, static analysis tools were the dominant method for detecting software bugs, with numerous studies [45, 46, 47, 48, 49] focusing on their effectiveness in real-world settings.

Developed by Meta, *Infer* [50] is a prominent static analysis tool designed to detect bugs in the backend and mobile applications [51]. It supports *Java*, *C*, *C++*, *Objective-C*, and *C#*, and performs interprocedural analysis along with integration into CI pipelines to catch issues early in development. Similarly, *SpotBugs* [52] inspects bytecode to detect specific types of bugs. It relies on predefined bug patterns with a set of rules and definitions to identify issues. However, it only supports *Java*, limiting its scalability. Another widely-used static analysis tool is *ErrorProne* [53], developed by Google. It integrates directly into the *Java* compiler, enabling proactive detection of bugs during compilation. This allows developers to catch and address issues early in the development cycle before code is deployed.

While static analysis tools played a significant role in bug detection, their rule-based nature and limited contextual understanding restricted their effectiveness in complex software environments. The emergence of LLMs in software engineering has introduced advanced bug detection approaches that go beyond traditional static methods. These models enable deeper contextual reasoning, natural language understanding, and improved flexibility and scalability. This shift has inspired novel techniques such as *FUzzGPT* [54] and *TitanFuzz* [55], which leverage these models to generate edge-case test inputs and perform mutation-based fuzzing for deep learning libraries.

During the fuzzing process, *FUzzGPT* utilizes natural language descriptions to generate complex test cases with LLMs, enabling a broader exploration of edge cases. This increased coverage enhances accuracy and improves the detection of vulnerabilities that are typically

difficult to uncover. In contrast, *TitanFuzz* refines the fuzzing process by integrating LLM-generated test cases with adaptive mutation strategies. This approach contextually identifies deeply embedded defects within project-specific details, making it more effective at uncovering hard-to-detect issues. By embedding LLMs into the fuzzing process, both *FUzzGPT* and *TitanFuzz* outperform traditional fuzzing techniques, achieving higher efficiency and accuracy in bug detection due to their improved contextual awareness.

Beyond fuzzing-based techniques, several recent efforts have explored using pre-trained LLMs for direct bug detection and localization. For example, *BugLab* [56], this technique works using a self-supervised approach. It leverages two LLMs, a detector model to detect bugs and repair them, and a selector model that automatically generates artificial bugs in the code. This co-training strategy allows the model to learn from a wide variety of bugs, enhancing its ability to resolve real-world software defects. Recent general-purpose approaches have also demonstrated strong capabilities in bug detection. Notably, *CodeBERT* [57], a general-purpose pre-trained model, has been fine-tuned for tasks such as bug detection, localization, and code understanding. It remains one of the top-performing and most widely cited models in this space.

In summary, while recent LLMs have shown remarkable progress in code generation, repair, and bug detection, their evaluation still relies heavily on benchmark datasets. This thesis shifts the focus from proposing new models to ensuring the reliability of existing ones by analyzing their behavior in real-world scenarios, using *SWE-bench*. We argue that data

quality can significantly impact perceived model performance and must be carefully addressed to enable fair and meaningful comparisons.

2.2 Benchmark Dataset Quality in Software Engineering

Recent studies have shown the importance of high-quality benchmarks in software engineering. Moreover, many works have demonstrated that existing benchmarks often contain a significant amount of noise and low-quality labels [58]. Sun et al. [59] focused on improving the quality of code search datasets. They removed noise, including invalid or incomplete queries, redundant tokens, and non-natural-language queries, using a combination of rule-based syntactic and model-based semantic filters. After cleaning the dataset, they observed a significant improvement in the performance of a popular code search model, *DeepCS*. On the other hand, Gro et al. [60], focusing on code-comment datasets, demonstrated that high repetition (e.g., frequent trigrams) in these datasets can inflate model performance, highlighting the need to address such noise for more reliable evaluation. In another study, Shi et al. [61] showed that both the size of benchmark datasets and the choice of preprocessing techniques have a noticeable direct impact on model performance. More interestingly, Huang et al. [62] highlighted that the data selected in many benchmarks can suffer from noise and bias, particularly due to the missing documentation in many real-world codebases. They observed that only a small fraction of code is well documented, which can limit the effectiveness of models trained on such data. To address this, they proposed a machine

learning technique that automatically adds comments to under-documented code, helping to enhance both data quality and model performance.

Beyond data quality, multiple studies have questioned the reliability of evaluation metrics used for assessing model outputs. Several studies examined the reliability of BLEU, one of the most widely used metrics for evaluating code generation and summarization. Roy et al. [63] show that minor increases (less than 2 points) in metric scores are not actually reflective of real code improvements, as judged by human annotators. Similarly, Mahmud et al. [64], it was observed that semantically correct auto-generated comments may be scored poorly simply because they don't match the exact wording of the reference. Furthermore, some minor variations in BLEU implementations have been shown to produce significantly different performance scores [60, 61]. These issues raise concerns about the reliability of BLEU-based evaluations and the validity of model performance claims based on them.

Reliable assessment of LLMs in software engineering tasks is heavily dependent on the quality of evaluation benchmarks, not only during training but also during the evaluation stage [1]. Xu et al. [65] highlight the critical role of benchmark dataset quality in evaluating models for software engineering tasks, with a particular focus on code summarization. They developed an automated code-comment cleaning tool to reduce noise in four widely used benchmarks. Their results demonstrate that with reduced noise, models perform significantly better. This highlights the importance of high-quality, noise-free evaluation datasets for reliably assessing model capabilities.

With the growing interest in applying LLMs across various software engineering tasks, including code generation, program repair, and bug detection, there is a critical need for trustworthy and reliable evaluation benchmark datasets [1, 8]. To tackle these challenges, *SWE-bench* was developed to assist in evaluating LLMs using real-world GitHub issues [1]. Unlike other state-of-the-art evaluation datasets that rely on synthetic issues, *SWE-bench* is built entirely from real-world GitHub issues, ensuring that the benchmark reflects actual software development challenges. Additionally, it incorporates both developer-generated code changes and requires models to submit their generated code fix patches. This creates a more precise and transparent assessment of LLM capabilities in automated code fixing, providing a more reliable measure of their effectiveness in real-world settings.

A critical component of a high-quality dataset is ensuring it is representative of real-world software challenges [2, 66]. Multiple recent studies also reinforce the need for realistic and challenging benchmarks. Li et al. [2] showed that even sophisticated multi-agent systems, like *CodeR*, struggled with *SWE-bench Lite*, resolving fewer than 28.33% of issues—mainly due to the dataset's strict evaluation criteria and complexity. These findings highlight the natural complexity of real-world software engineering tasks as captured by *SWE-bench*, in contrast to synthetic benchmarks such as *HumanEval* [14] and *EvalPlus* [67, 68], where models tend to achieve much higher performance.

The "Diversity Empowers Intelligence" (DEI) framework [3] further shows how diversity and comprehensiveness of data can enhance the performance of LLMs in code generation tasks. By leveraging diverse SWE agents, DEI capitalizes on the complementary strengths

of these agents in software engineering tasks such as bug fixing and patch generation. This broad spectrum of expertise significantly improves the collective performance [3], highlighting the importance of including a wide range of issues that represent real-world variability in software engineering.

Given the critical need for high-quality benchmark datasets in evaluating code generation models, Liu et al. [68] introduce *EvalPlus*, a framework specifically designed to improve the evaluation of LLM-generated code. This framework addresses challenges like insufficient tests and noisy benchmarks by augmenting existing datasets with automated test input generation using LLMs and mutation-based strategies. Their findings show the need for more rigorous testing in LLM code generation to improve benchmark quality in the field.

Chapter 3

Research Design

3.1 Robustness Analysis of *SWE-bench*

We conducted an empirical study on patches generated by *SWE-Agent + GPT-4* for issues in the *SWE-Bench Full* dataset. We selected *SWE-Agent + GPT-4*, a state-of-the-art agent-based model that, at the time of our study, held the highest resolution rate (12.47%) among all open-source models on the *SWE-bench Leaderboard*. The benchmark is used in software engineering to provide a standardized metric for evaluating the performance of automated patch generation models. The leaderboard tracks resolution rates across various datasets, offering insights into how different models approach complex software issues.

We selected this model because of its leading performance, which offers a valuable opportunity to analyze how top-tier models approach patch generation and issue resolution.

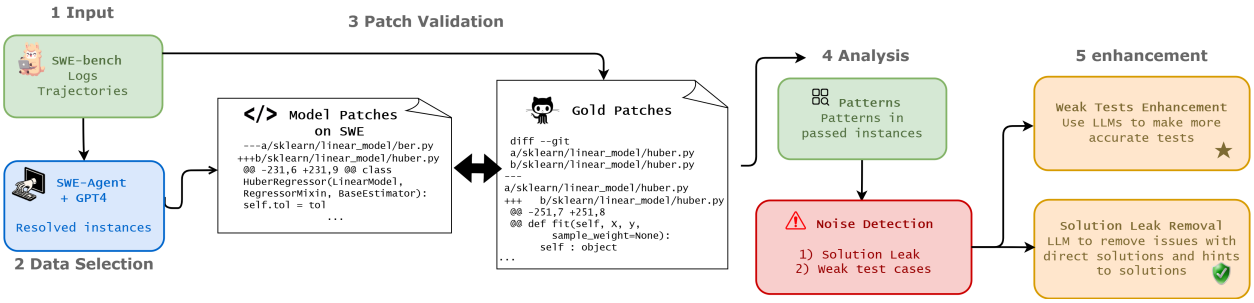


Figure 3.1: Overview of robustness analysis for *SWE-bench* datasets

The primary goal of our thesis was to assess whether the generated patches contained potential issues and, if so, to categorize these issues into distinct *patterns*.

Figure 3.1 outlines the major steps we followed in our thesis. The input is the set of all issues in *SWE-bench*. Each issue contains a description and a patch to address the issue. Each patch is a diff of code changes, which we call a "gold patch." We used *SWE-Agent + GPT-4* to generate fixes for each issue. We refer to the model output for an issue as a "generated patch." We then compared the gold and generated patches by analyzing the corresponding files changed in the pull requests on GitHub with the same *instance_id*. As we studied the model-generated patches, we also examined the logs and trajectories generated by the model. Logs provide the step-by-step execution of the model. The trajectory data provide a detailed record of the model's decision-making process while producing a patch, showing how it iterated through possible solutions before arriving at a final version.

The comparison between the gold and generated patches was essential to evaluate how closely the model's output aligned with the developer-generated patches and to assess the quality of the solutions provided.

To reduce potential bias, three developers with background in Python programming and code analysis independently conducted the patch validation study. Each annotator carefully examined the modified files and lines, reviewed issue descriptions, and analyzed the implementation styles and intentions behind both model-generated and developer-generated patches. The goal was to identify patterns in the model's performance, whether it handled certain types of issues well or struggled with others. For example, the model might successfully resolve well-defined, simpler issues but struggle with more ambiguous ones. Additionally, we assessed whether the model's patches were of higher or lower quality compared to those created by developers.

Any disagreements among the annotators were resolved through group discussions. This analysis aimed to determine whether the model's performance varied based on issue complexity, description clarity, or the overall quality of its generated solutions.

For the model-generated patches, we focused on instances where the generated patches resolved the issue and passed all associated tests. As a result, we identified 251 instances from the *SWE-Bench Full* dataset. Note that, to ensure the patches passed all tests, we reviewed the evaluation logs of 286 instances initially marked as resolved by *SWE-Agent + GPT-4* in the *results.json* file from the *SWE-Bench Full* evaluation repository and selected only those with logs confirming that all tests passed following the application of the generated patches.

Table 3.1: Patterns found among the 251 successful patches generated by *SWE-Agent + GPT-4*

Type	Pattern	Numbers (percentage)	Root cause
	Solution Leak	57 (22.71%)	solution leakage
	Solution hint leak	27 (10.76%)	solution leakage
Suspicious fixes	Incorrect fixes	19 (7.75%)	weak tests
	Different files/functions changed	10 (3.98%)	weak tests
	Incomplete fixes	33 (13.15%)	weak tests

3.1.1 Solution Leak and Weak Tests

Among the 251 generated patches that passed all test cases in the *SWE-Bench Full* dataset, we found around 58.17% of the successful patches to be problematic/suspicious, which can be summarized into five different patterns. Table 3.1 outlines five patterns of suspicious patches in the 251 generated patches. To explain each pattern, we provide definitions, including the number of instances associated with each pattern and the likely root causes. We discuss each pattern below.

1. Solution Leak: This represents instances where the solution to the issue is clearly outlined in the issue description or comments on GitHub. Since both the issue descriptions and comments (referred to as *hints_text* in the *SWE-bench* study) are provided as input to the models, these LLM models can extract the solutions directly from this information instead of generating it independently. 22.71% of the successfully resolved issues followed this pattern, making it the most common among resolved patches. This raises significant concerns about the model's actual performance and the validity of the *SWE-bench* instances

Figure 3.2: Solution leakage in issue report for sympy-16669

as a benchmark. If a model is simply copying the solution it already has access to, it isn't demonstrating true problem-solving capabilities but rather replicating what is provided, thus limiting the assessment of its ability to generate new solutions.

The example shown in Figure 3.2 illustrates issue report 16669[3] from the *sympy* project, where the issue description provided the exact solution code patch required to resolve the issue, making it possible for the model to directly copy the solution from the issue report and generate the same solution as provided.

2. Solution Hint Leak: This pattern emerges when issue descriptions or comments contain partial information or indirect suggestions that guide models toward the solution without explicitly providing the complete fix. This pattern was present in 10.76% of instances. For instance, as shown in Figure 3.4, a solution hint is provided in the problem description. While it does not provide the complete implementation, it gives a clear direction about which

[3] `https://github.com/sympy/sympy/issues/16669`

Problem Statement:

```
The SQLCompiler is incorrectly removing "order by" clauses because it
    determines the clause was already "seen" in SQLCompiler.get\_order\
    _by(). The issue occurs with expressions written as multiline
    RawSQL.The bug is located in SQLCompiler.get_order_by(),
    specifically at: without_ordering = self.ordering_parts.search(sql).
    group(1)
As a quick/temporal fix I can suggest making sql variable clean of
    newline characters, like this:
sql_oneline = ' '.join(sql.split('\n'))
without_ordering = self.ordering_parts.search(sql_oneline).group(1)
```

Problem Statement After Solution Removal:

```
The SQLCompiler is incorrectly removing "order by" clauses because it
    determines the clause was already "seen" in SQLCompiler.get\_order\
    _by(). The issue occurs with expressions written as multiline
    RawSQL.The bug is located in SQLCompiler.get_order_by(),
    specifically at: without_ordering = self.ordering_parts.search(sql).
    group(1)
```

Figure 3.3: Analysis of Direct Solution Leak in Problem Statement

function should be used and where it should be applied. Such hints can influence how a model approaches the solution, unintentionally simplify the task.

3. Incorrect Fixes: This refers to cases where the model-generated patches provide incorrect solutions, yet pass the test cases when they should have failed. This pattern was present in 7.75% of the passed instances, suggesting a weakness in test cases where the functionality of the issue resolution is not correctly captured. The fact that incorrect patches can pass the test cases raises suspicion about the relevance and accuracy of the test cases in assessing whether the issue has been fully resolved.

Figure 3.5 shows a comparison between the model-generated patch and the gold patch

Problem Statement:

```
1 A security vulnerability exists in the password reset functionality
      where tokens remain valid even after a user changes their email
      address. The current implementation of PasswordResetTokenGenerator
      does not consider email address changes when validating tokens.
2 The fix is to add the user email address into
      PasswordResetTokenGenerator._make_hash_value()
3 Nothing forces a user to even have an email as per AbstractBaseUser.
      Perhaps the token generation method could be factored out onto the
      model, ala get_session_auth_hash().
```

Problem Statement After Solution Removal:

```
1 A security vulnerability exists in the password reset functionality
      where tokens remain valid even after a user changes their email
      address. The current implementation of PasswordResetTokenGenerator
      does not consider email address changes when validating tokens.
```

Figure 3.4: Analysis of hint leak in problem statement

Gold Patch

```
django/utils/datastructures.py
 def __iter__(self):
     return iter(self.dict)
+def __reversed__(self):
+    return reversed(self.dict)
```

Generated Patch

```
django/utils/datastructures.py
def __len__(self):
     return len(self.dict)
+def __reversed__(self):
+    return reversed(self.dict.keys())
```

Figure 3.5: Incorrect fix generated by the model for django-32517

for django-32517[4]. According to the issue description, a new functionality is needed to reverse a Python *OrderedSet* by implementing the *__reversed__* function. The gold patch demonstrates the correct behavior, where the entire dictionary is reversed, while the generated patch only reverses the dictionary's keys. As a result, the two patches produce entirely different outputs, as they apply different methods to the dictionary.

[4] https://code.djangoproject.com/ticket/32517

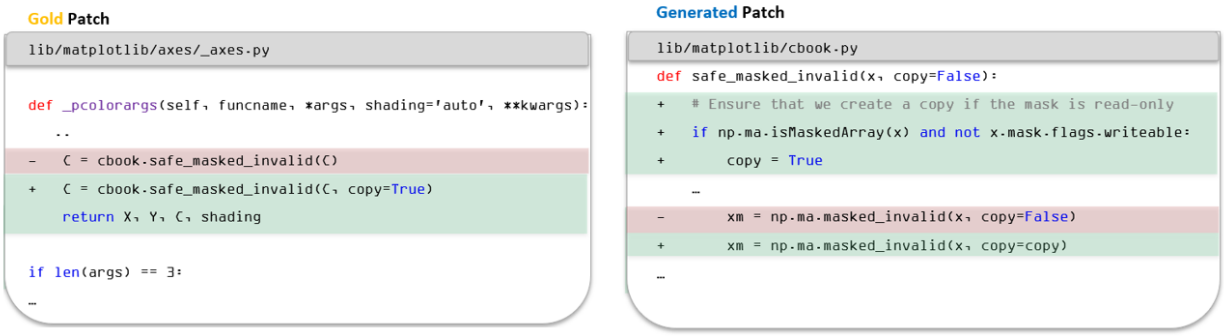


Figure 3.6: Different files changed by model for issue-26093 of Matplotlib

4. Different Files/Functions Changed: This pattern refers to cases where the model-generated patches modify files or functions unrelated to the issue at hand. These files differ from those altered in the gold patch, yet the model's patches still pass the test cases despite this discrepancy. This pattern appears in 3.98% of the passed instances. It highlights a weakness in the model's ability to accurately locate and address the source of the issue. The fact that the test cases pass, even though changes were made in irrelevant files, suggests that the test cases are either weak or irrelevant and should have failed in detecting the incorrect modifications.

Figure 3.6 presents an example from issue-26093 of the Matplotlib project[5], where the model-generated patch modifies the *cbook.py* file, while the gold patch makes changes to the *_axes.py* file. This shows that the model's patch affects a completely different file from the gold patch, highlighting the model's inability to accurately identify the correct file containing the bug.

[5]https://github.com/matplotlib/matplotlib/issues/26093

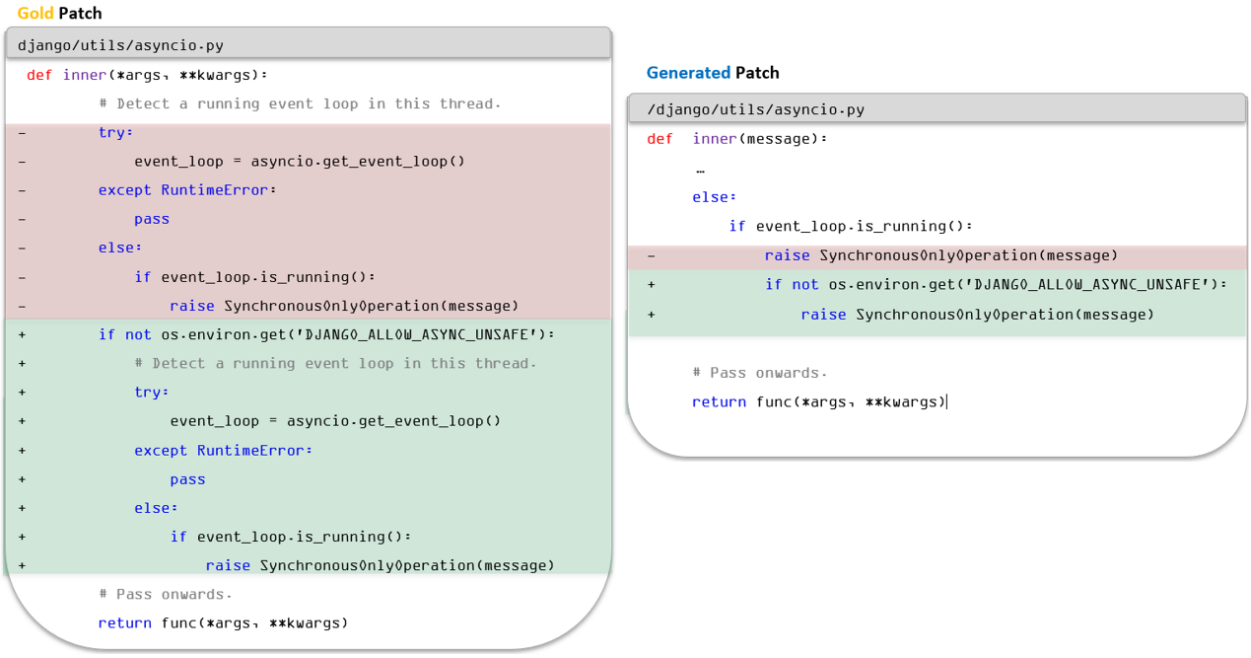


Figure 3.7: Incomplete fix generated by the model for django-31056

5. Incomplete Fixes: This pattern refers to model-generated patches that offer incomplete implementations compared to the gold patches, often omitting critical details. This pattern appears in 13.15% of passed instances. For instance, some patches include only partial if-else statements, neglecting edge cases that the gold patch addresses. Although the model-generated patches follow the correct implementation approach, they overlook important aspects that could lead to failures in production or when handling edge cases. This underscores a weakness in the test cases, as they fail to capture the finer details necessary for a comprehensive issue resolution.

The example provided in Figure 3.7 shows the same change being made by the model and the one made by the developers in the gold patch[6]. The gold patch provides a complete

[6]`https://code.djangoproject.com/ticket/31056`

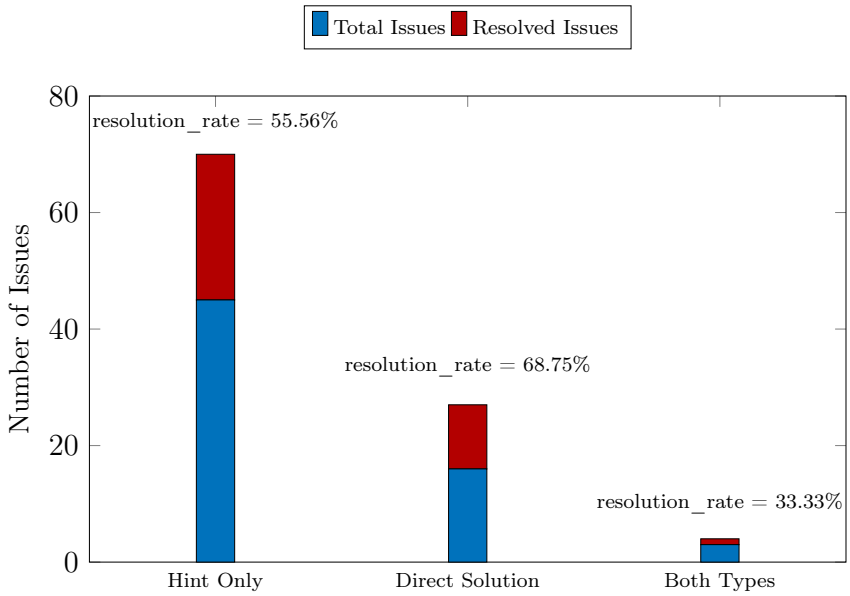


Figure 3.8: Analysis of solution leak impact on model performance

fix while the model patch provides a partial fix. Specifically, the gold patch properly handles the detection of an event loop in the current thread by including a *try-except* block to catch *RuntimeError* when an event loop is unavailable and checks if the event loop is running before raising an exception. Additionally, it wraps the entire logic in a condition that checks the environment variable *DJANGO_ALLOW_ASYNC_UNSAFE*. In contrast, the generated patch is missing critical parts of this logic, such as the *try-except* block and the check for a running event loop. As a result, the model-generated patch is incomplete, missing key error handling and flow control that are necessary for ensuring safe operation.

Impact of Solution (Hint) Leak

To quantitatively assess the impact of solution leakage on model performance, we conducted an empirical study using *AutoCodeRover + GPT-4o*. From the 94 issues initially successfully

resolved by *AutoCodeRover + GPT-4o*, we identified 64 issues containing some form of solution leakage (either direct solution leak or solution hint leak). We then manually removed all solution-related information, including direct code solutions, suggestions, and hints (e.g., expected behavior) from the issue description and re-evaluated the model's performance to see if the model could still fix the issues. As shown in Figure 3.8, after removing the leaked information, the model could still successfully resolve 39 out of 64 issues (60.94% resolution rate), with 23 failed cases and 2 invalid patches. This indicates that a solution leak showed a positive impact on the model in producing the correct fix. To further understand the impact of different types of leaked information, we categorized the issues based on their leakage type. Among 45 issues containing only hints (e.g., suggested approaches or expected behaviors), the model maintained a 55.56% resolution rate (25 successful fixes) after hint removal. For the 16 issues containing direct solution leaks, the model achieved a resolution rate of 68.75% (11 successful fixes) after solution removal. For the 3 issues containing both hints and solutions, the resolution rate was 33.33% (1 successful fix) after removing the information. These results show that the highest resolution rate for issues with direct solutions suggests that direct solutions were actually more useful for the model in fixing issues.

3.1.2 Updated Resolution Rate of *SWE-Agent + GPT-4* on *SWE-bench Full*, *Lite*, and *Verified*

After identifying the five suspicious patterns, we recalculated the resolution rate of *SWE-Agent + GPT-4* by excluding these suspicious patches. This new resolution criterion resulted in

a significant drop in the resolution percentage, as shown in Figure 1.1b. The performance of *SWE-Agent + GPT-4* decreased from 12.47% to 4.58% when considering only correct fixes, those in which the model generated correct but different implementations from the gold patch or more comprehensive fixes than the gold patch. We excluded all suspicious fixes, including instances where the model generated incorrect solutions (7.75%), the issue report contained a direct solution (22.71%), or provided a hint (10.76%). Additionally, we removed cases where changes were made in files unrelated to the gold patch (3.98%) and instances where the model produced incomplete fixes, omitting critical details of the solution (13.15%).

Two new variants of *SWE-bench* are recently developed, *SWE-Bench Lite* and *SWE-Bench Verified*, each designed with different goals. *SWE-Bench Lite* focuses on instances with lower evaluation costs and increased accessibility. On the other hand, *SWE-Bench Verified* aims to provide a curated subset of *SWE-bench*, where human annotators filter out issues with underspecified descriptions or weak unit tests that might reject valid solutions. However, neither of these new datasets addresses the solution leakage problem, which was the primary motivation for our thesis. To investigate this, three developers with background in Python programming and code analysis reviewed all issue reports for the instances in *SWE-Bench Lite* and *SWE-Bench Verified* for cases of solution leakage.

In SWE-Bench Lite, we identified 18 instances where the solution was directly provided in the issue description or discussion on GitHub. Similarly, in *SWE-Bench Verified*, 30 instances contained direct solutions in either the issue description or the discussion on GitHub. Table 3.2 shows more details.

Table 3.2: Patterns found among *SWE-Bench Lite* (total passed: 54, 18.0%) and *SWE-Bench Verified* (total passed: 112, 22.4%) datasets with successful patches generated by *SWE-Agent + GPT-4*.

Pattern	**SWE-Bench Lite**	**Percentage (Lite)**	**SWE-Bench Verified**	**Percentage (Verified)**
Incorrect fixes	2	3.70%	8	7.14%
Incomplete fixes	5	9.26%	9	8.04%
Different Files/Functions Changed	2	3.70%	2	1.79%
Solution Leak	18	33.33%	30	26.79%
Solution Hint Leak	3	5.56%	13	11.61%

We also observed additional suspicious fixes in both *SWE-Bench Lite* and *SWE-Bench Verified*, as summarized in Table 3.2. Specifically, *SWE-Agent + GPT-4* produced incorrect fixes in 3.70% of cases in *SWE-Bench Lite* and 7.14% in *SWE-Bench Verified*. We also observe that 3.70% of changes in *SWE-Bench Lite* were made in different files or functions than the gold patches, and 1.79% in *SWE-Bench Verified*. Additionally, 33.33% of fixes in *SWE-Bench Lite* and 26.79% in *SWE-Bench Verified* involved direct solution leaks. Hint solution leaks were observed in 5.56% of fixes in *SWE-Bench Lite* and 11.61% in *SWE-Bench Verified*.

The overall suspicious fix patterns led to 55.56% suspicious fixes in *SWE-Bench Lite* and 55.36% in *SWE-Bench Verified*, significantly reducing the resolution rates from 18% to 8.00% in *SWE-Bench Lite* and from 22.4% to 10.00% in *SWE-Bench Verified*.

3.2 Building *SWE-Bench+*

To address the issues of the current *SWE-bench* datasets and ensure a more accurate evaluation of the models' effectiveness in resolving issues, we created a more rigorous evaluation dataset, *SWE-Bench+*, by filtering out suspicious patches in *SWE-bench*. Specifically, we introduce an LLM-based solution leak detection tool, i.e., `SoluLeakDetector`, to filter out issues with the solution leak problem, and an LLM-based test generation approach, i.e., `TestEnhancer`, to enhance the test suite of issues and avoid the weak test problem.

The primary objective of *SWE-Bench+* is to assess the models' ability to generate accurate patches for real-world GitHub issues without the risk of bias or prior exposure to the solution.

3.3 `SoluLeakDetector`: LLM-based Solution Leak Issue Detection

Our study revealed the persistent presence of solution leakage in *SWE-bench* instances, where issue reports often contain direct solutions or indirect hints regarding the resolution of reported bugs. To mitigate this issue, we developed `SoluLeakDetector`, an LLM-based technique designed to systematically detect and filter instances in *SWE-bench* that exhibit either direct solution leaks or indirect solution hint leaks.

`SoluLeakDetector` leverages *GPT-4* to classify instances into three distinct categories: (1) instances containing direct solution leaks, (2) instances with hint-based solution leaks, and (3) instances free from any form of solution leakage.

As seen in Figure 3.9, the prompt defines direct solution leaks as being explicitly mentioned in the issue description with code snippets and instructions on how and where these code snippets should be applied in a step-by-step manner, and defines indirect solution leaks as being subtly implied in the issue description by explaining a direction that the model should take to resolve the issue. Any instance that does not include either of the solutions is identified as a clean instance without any solution leak.

As illustrated in Figure 3.9, the prompt explicitly defines direct solution leaks as those where the issue description contains code snippets along with step-by-step instructions detailing how and where these snippets should be applied. Indirect solution leaks, on the other hand, are subtly embedded within the issue description, guiding the model toward a specific resolution strategy without explicitly stating the fix. Any instance that lacks both types of leaks is classified as a clean instance without any solution leakage.

The classification process employs a three-shot prompting technique, wherein three well-defined examples from each category are presented to guide *GPT-4* in accurately categorizing *SWE-bench* instances. The issue descriptions used for classification are extracted from the root of the instance's pull request. Each example is paired with an expected output label corresponding to one of the three categories.

Furthermore, `SoluLeakDetector`identifies and extracts the solutions and hints from the issue descriptions. It then generates a detailed explanation justifying why the extracted fragment is categorized as either a hint-based or direct solution leak.

You are a solution leakage detection expert.

Task

Analyze GitHub issue descriptions (`problem_statement`) and related comments (`hints_text`) for solution leakage.

Definition

solution leakage occurs when:

1. The solution is **explicitly mentioned**, such as:
   - Code snippets providing a direct fix.
   - Step-by-step instructions leading directly to the solution.
2. The solution is **subtly implied**, for example:
   - Explanatory text that indirectly leads to the correct fix.
   - Comments hinting at how to resolve the issue.

Example 1

Description: I propose to add the following settings, with the following default values:

```
LANGUAGE_COOKIE_SECURE = False
LANGUAGE_COOKIE_HTTPONLY = False
LANGUAGE_COOKIE_SAMESITE = None
```

The default values maintain the current behavior. These settings do not provide much security value, since the language is not secret or sensitive.

Expected Output:

```
{
  "solution_leakage_detected": true - direct,
  "reason": "The solution is explicitly provided in the description.",
  "extracted_solution": "LANGUAGE_COOKIE_SECURE = False,
     LANGUAGE_COOKIE_HTTPONLY = False, LANGUAGE_COOKIE_SAMESITE = None"
}
```

Example 2

Description: Shape of `coef_` is incorrect for `linear_model.Lasso` when using `fit_intercept=False`.
Expected Output:

```
{
  "solution_leakage_detected": false,
  "reason": "The description identifies a bug but does not explicitly provide
      a solution.",
  "extracted_solution": null
}
```

Example 3

Description: A typo in `Poly3DCollection.__init__()` causes a `TypeError` exception.
Expected Output:

```
{
  "solution_leakage_detected": true - hint,
  "reason": "The solution is explicitly provided as a corrected code snippet.
      ",
  "extracted_solution": "edgecolors in None should be edgecolors is None"
}
```

Figure 3.9: Prompt for *SoluLeakDetector*

3.4 TestEnhancer: LLM-based Tests Enhancement

Our thesis highlights that 24.70% of the suspicious fixes described in 3.1 are due to weak tests. In other words, the generated patches pass the tests even when they are incorrect, incomplete, or entirely irrelevant, indicating that the tests themselves may be overly permissive. To address this issue, we developed `TestEnhancer`, an automated test generation and validation process aimed at enhancing the original *SWE-bench* test patches. Once the relevant instances

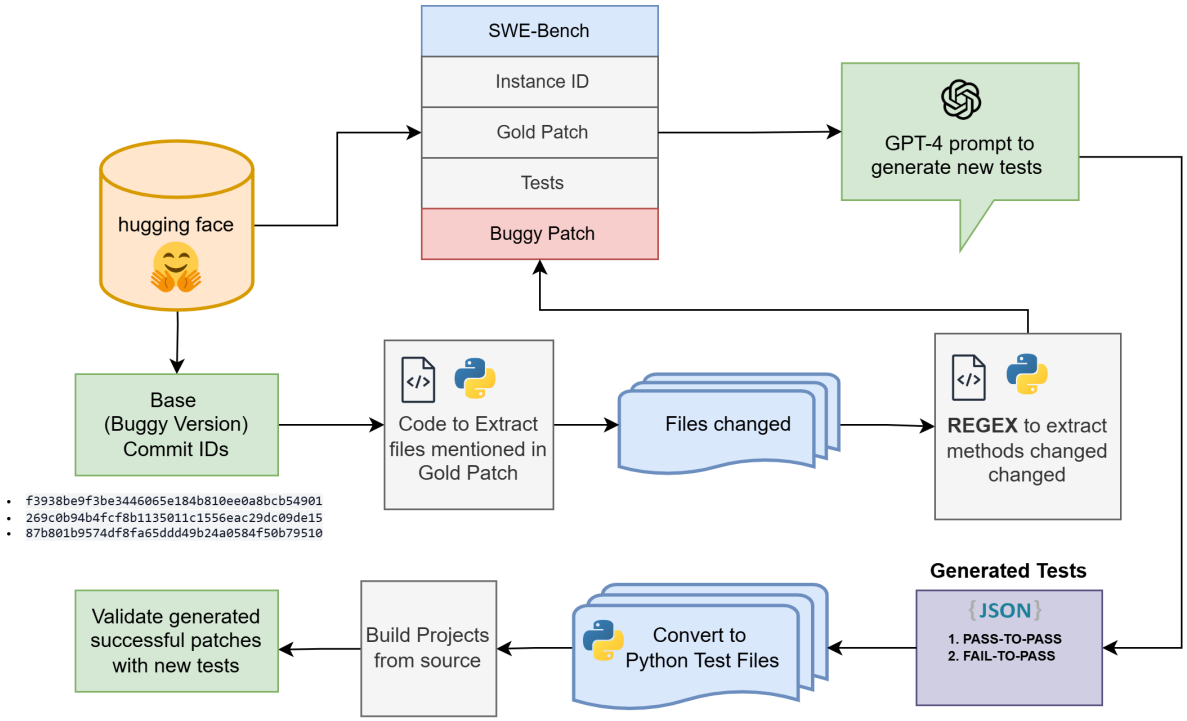


Figure 3.10: Methodology for `TestEnhancer`

are filtered, we prompt *GPT-4* to generate enhanced test cases using the extracted data as input. For each instance, we generate 20 *PASS-TO-PASS* tests and 20 *FAIL-TO-PASS* tests. The generated test cases are initially structured in *JSON format* and later converted into executable *Python test files* following the *pytest* format. After successful compilation, we run the tests on the patches generated by *SWE-Agent + GPT-4* to verify whether the generated fixes successfully resolve the reported issues.

Figure 3.10 outlines the full methodology we used to create `TestEnhancer`. First, we begin by selecting specific attributes from the *SWE-bench* dataset. Specifically, we extract the following attributes: *instance_id*, *gold_patches*, *test_patches*, and *base_commits*. The *base_commits* represent the commit IDs leading to the buggy version of each instance before

the corresponding PR. To retrieve the buggy code patches, we trace these *base_ commits* and extract the affected source code files from the repository. Next, we identify the specific files modified in the *gold patch* and apply *REGEX-based pattern matching* to extract the functions or methods that were changed.

Now, we have the full list of attributes needed to provide the LLM with more context in order to enhance the tests. To do so, we explain in the prompt the task required, which is generating one PASS-TO-PASS and one FAIL-TO-PASS test per instance using *pytest*. We provide the expected output and format to make it easier to summarize passing and failing tests and ensure non-buggy code. We then give the model all the input attributes extracted as described earlier and run the prompt twenty different times to generate twenty PASS-TO-PASS and twenty FAIL-TO-PASS tests.

The generated tests are stored in *JSON* format and later converted into executable *Python* test files following the *pytest* format.

In order to run the tests, we built each project from source with the test requirements. After building the projects, we added the generated tests into the test suite of each project. We wrote a script to summarize the output when running the tests with *pytest* in an Excel file, showing the number of passing and failing tests per project.

Finally, we validate the generated patches by *SWE-Agent + GPT-4* that were successful over the newly generated tests. If at least one test fails, then the original tests are weak, and that's why all of them passed.

You are an expert in generating runnable Python test cases using `pytest`.
TASK: Your task is to generate exactly **two** test cases per run:

- **One pass-to-pass (ptp) test** for valid inputs.
- **One fail-to-pass (ftp) test** for invalid inputs and edge cases.

EXPECTED OUTPUT:

1. Each test case must:
   - `test_ptp_` for pass-to-pass tests.
   - `test_ftp_` for fail-to-pass tests.
   - Contain **one or two assertions** at most.
   - Provide setup code and context before assertions.
2. Use proper `pytest` syntax, such as `pytest.raises` for exceptions.
3. Ensure the test is self-contained and runnable with `pytest`.

Buggy Code: `buggy_code`
Gold Patch: `gold_patch`
Original Tests: `tests`
OUTPUT FORMAT: Provide exactly **two** test cases in a Python code block. The test functions should:

1. Begin with `test_ptp_` or `test_ftp_` depending on the category.
2. Include setup code and one or two assertions.
3. Only Python code (no comments or descriptions).

Figure 3.11: Prompt for `TestEnhancer`.

3.5 Evaluation of *SWE-Bench+*

3.5.1 Performance Analysis of `SoluLeakDetector`

To assess the accuracy of `SoluLeakDetector`, we first applied it to *SWE-Bench Lite*. To ensure the reliability of the classification, three developers with background in Python programming and code analysis independently reviewed the categorized instances. This human evaluation process helped mitigate potential biases and verify the correctness of `SoluLeakDetector`'s predictions.

The manual review confirmed that `SoluLeakDetector` achieved an 86% accuracy in correctly classifying solution leaks within *SWE-Bench Lite*. Following this validation, we applied

SoluLeakDetector to the full *SWE-bench* and *SWE-Bench Verified* datasets to categorize all instances accordingly.

After categorizing all instances, we constructed *SWE-Bench+*, a refined benchmark that excludes any instance exhibiting solution leakage. *SWE-Bench+* consists of 707 instances that were verified to contain no direct or indirect solution hints, ensuring a more reliable evaluation dataset. By eliminating biased instances, *SWE-Bench+* provides a fairer and more robust framework for evaluating LLM-based models in real-world GitHub issue resolution.

3.5.2 Performance Analysis of TestEnhancer

To evaluate the performance of TestEnhancer, we filter out instances where at least one *PASS-TO-PASS* or one *FAIL-TO-PASS* test fails. In other words, if a generated patch fails one of the newly generated tests but passes all the original test patches in *SWE-bench*, this indicates that the test patches in *SWE-bench* are too permissive.

To apply TestEnhancer on *SWE-bench*, we selected the suspicious instances whose root cause is "weak tests" as shown in Table 3.1, while excluding instances from *Django* (since the project migrated its issue tracking outside of GitHub), leaving us with 49 instances. We applied TestEnhancer to these instances, but only 31 were successfully processed. The remaining instances either exceeded the token limit for *GPT-4* due to long gold and buggy patches or produced non-runnable tests.

Our evaluation revealed that 6 out of 31 instances (19%) had at least one *PASS-TO-PASS* test failing. This means that although the generated patches initially passed the original

SWE-bench tests, they were later identified as incorrect when validated with the newly generated tests. These results highlight that `TestEnhancer` enhances test robustness by uncovering cases where the original test suite failed to detect faulty fixes.

3.6 Data Leak

During patch validation, we focused on the *SWE-Agent + GPT-4* model, which is based on GPT-4 (1106) with a training cut-off date of April 2023. In this process, we observed that many issues arose after the model's cut-off date. To investigate potential data leakage, we developed a script to extract the creation dates of issues and pull requests for all *SWE-bench* instances. We then categorized the instances into three groups: (1) both issues and pull requests created before the cut-off date, (2) issues created before but pull requests after the cut-off date, and (3) both issues and pull requests created after the cut-off date. The distribution of these categories is shown in Figure 3.12.

As shown in Figure 3.12, approximately 94% of *SWE-bench* instances have both issues and pull requests created before the model's training cut-off date. This raises concerns about potential data leakage, as the model may have been exposed to these solutions during training, leading to inflated resolution rates.

To evaluate the performance of LLMs in generating code fixes without the risk of data leakage, we curated a new dataset composed exclusively of issues created after the training cut-off dates of a selected subset of models used in this study.

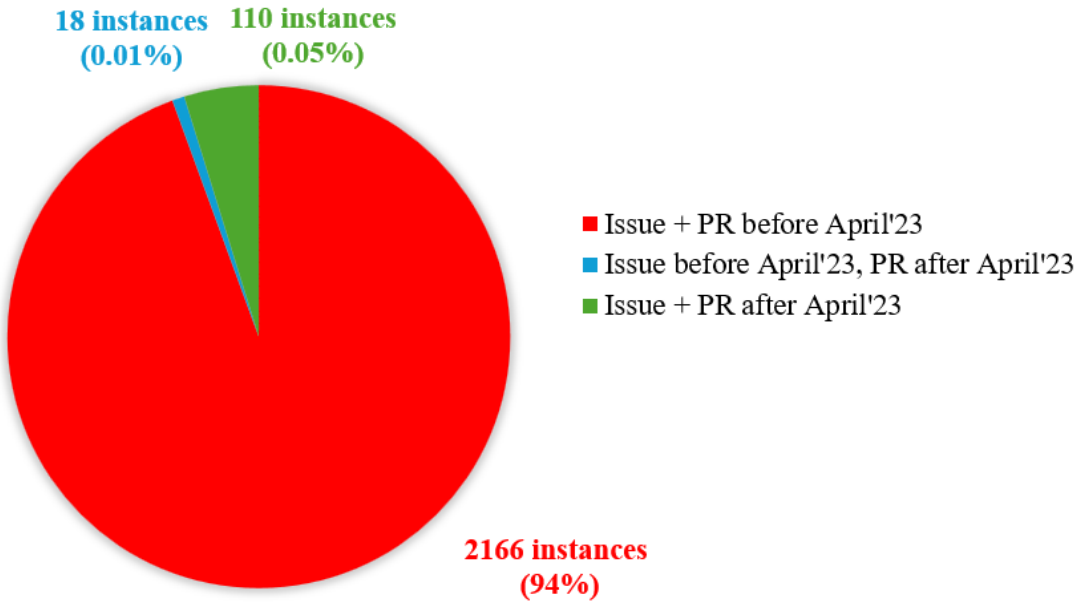


Figure 3.12: SWE-Bench data distribution relative to GPT-4 cut-off date

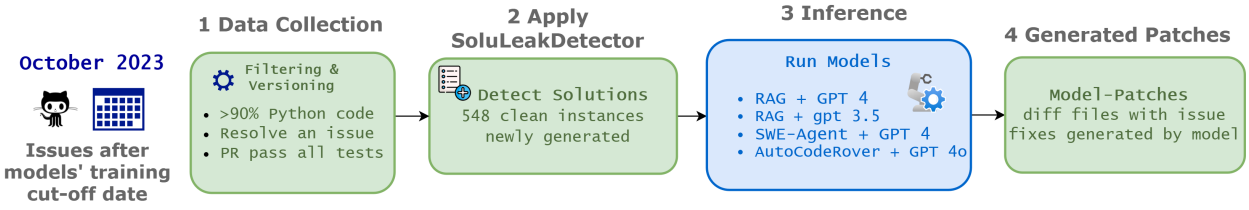


Figure 3.13: Collection and evaluation methodology for data created post training cut-off dates.

As shown in Figure 3.13, we followed a systematic process to collect new data and evaluate model performance while minimizing the risk of data leakage. To do this, we selected a cut-off date that falls after the training cut-off dates of several key models of interest. Specifically, we focused on *SWE-Agent* with *GPT-4 (1106)*, the primary model used in our study; *AutoCodeRover (v20240620)* with *GPT-4o (2024-05-13)*, the top-performing open-source model on the *SWE-bench* leaderboard at the time; and two retrieval-augmented generation baselines, *SWE-RAG* with *GPT-4 (1106)* and *SWE-RAG* with *GPT-3.5 (turbo-16k-0613)*, to

examine how retrieval methods perform on this new dataset. The respective training cut-off dates for these models were September 2021 for GPT-3.5, April 2023 for *GPT-4*, and October 2023 for *GPT-4o*. Consequently, we set our data collection start date to after October 2023 to ensure that all extracted issues were created beyond these cut-off dates.

3.6.1 Data Collection

First, we selected the same 12 projects from *SWE-bench*, excluding *Django*, as its issues are now tracked outside of GitHub.

As shown in Figure 3.13, we began by using the official *SWE-bench* collection pipeline provided by the authors. This pipeline utilizes the GitHub API to retrieve closed issues labeled as bugs along with their corresponding pull requests that resolve the issue. For each instance, the script collects key metadata, including issue and pull request numbers, titles, descriptions, comments, and creation and merge dates.

To prevent potential data leakage, we applied a strict cut-off, collecting only issues created after October 2023, as described in the previous section.

After collecting, we applied the same filtering process used in the original *SWE-bench* study. This involved attribute filtering to retain only issues that resolve a problem, contribute tests, and consist of at least 90% Python code, followed by execution filtering to keep only instances that install successfully and whose pull requests pass all tests.

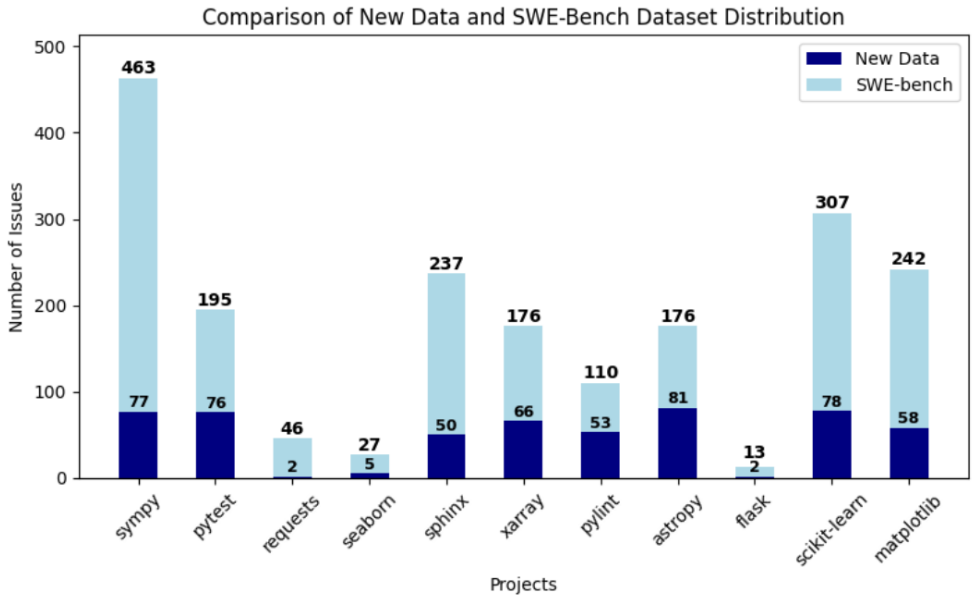


Figure 3.14: Distribution of collected instances compared to *SWE-bench*.

3.6.2 Apply *SoluLeakDetector*

After collecting the new dataset post-cut-off dates, we apply *SoluLeakDetector* as the last step to optimize data quality and remove solution leakage. As described in Section 3.3, this ensures that instances containing direct solutions or hints within issue descriptions or comments are filtered out, leaving a clean and leakage-free dataset. Following this pipeline, we curated a dataset of 548 instances from the selected projects, consisting of real-world issues created after the models' training cut-off dates that passed all filtering criteria, ensuring a robust and leakage-free evaluation.

The distribution of instances across the selected projects is shown in Figure 3.14. The figure shows the ratio of the new instances collected per project relative to the number of instances in the original *SWE-bench* dataset for each project.

3.6.3 Running the Models on New Data

After collecting data and identifying the correct repository versions, the next step was to run the selected models to generate patches that address the issues. As mentioned in the previous section, the models selected for this task were *SWE-RAG* with *GPT-3.5 (turbo-16k-0613)*, *SWE-RAG* with *GPT-4 (1106)*, *SWE-Agent* with *GPT-4 (1106)*, and *AutoCodeRover (v20240620)* paired with *GPT-4o (2024-05-13)*.

To run the models, we followed the guidelines provided by the *SWE-bench* repository, which includes versioning and performing inference on the instances.

As outlined in *SWE-bench*, each task must be linked to the correct version of the project repository at the time of issuance, ensuring that the appropriate installation instructions are followed. This versioning process is essential for enabling an accurate execution-based evaluation. To obtain these versions, we used the script provided by the *SWE-bench* open-source code, specifically `get_versions.py`, which ensures that the existing issue is selected for each instance.

To execute the patch generation process, we used two primary scripts: the `run_live.py` script for the RAG models on the *SWE-Bench+* issue URLs, and the `run.py` script from the *SWE-bench* open-source project for *SWE-Agent + GPT-4*. Both scripts were run with the same configurations as used in the original *SWE-bench* setup. These scripts were responsible for generating the model-generated patches (`model_patches`) intended to resolve the issues collected from the dataset.

3.6.4 Evaluating Model Performance on Post-Cutoff Issues

Our evaluation followed a four-step process, described in Figure 3.1, to ensure that the resolution rates of the models on the new dataset accurately reflect their correctness.

Step 1: We stored all model-generated diff files as patches in separate *json* files and utilized scripts from *SWE-bench* to evaluate them.

Step 2: We manually reviewed the resolved instances and filtered out cases where all tests in *PASS_TO_PASS* and *FAIL_TO_PASS* were not passed.

Step 3: We analyzed the resolution rates of the models. Since the new dataset contains issues that were unseen by the models earlier. This provided a more accurate assessment of their real-world performance.

Step 4: We performed a patch validation study similar to Section 3.1, comparing model-generated patches to developer-written patches.

On average, around 67.72% of the resolved instances did not truly resolve the issue, despite passing all the tests. The most prominent pattern observed was the models' inability to accurately locate the buggy files or lines. This raises concerns about the ability of models to identify the source of errors without prior knowledge. In other cases, the models generated incomplete or incorrect fixes that did not resolve the issue. Findings suggest that the data leakage issue has been addressed, whereas weak test cases remain a significant issue.

As a result of the patch validation analysis, the final resolution rates are described in Table 3.3

Table 3.3: Resolution Rates of LLMs on Post-Cutoff Dataset

Model	**Resolution Rate (%)**
SWE-RAG + GPT-4	0.73
SWE-RAG + GPT-3.5	0.55
SWE-Agent + GPT-4	1.99
AutoCodeRover + GPT-4o	3.83

As compared to the reported resolution rates on the *SWE-bench* leaderboard, we observed a significant decline. The reported rates were 1.31% for *SWE-RAG + GPT-4*, 0.17% for *SWE-RAG + GPT-3.5*, and 18.83% for *AutoCodeRover + GPT-4o*. This decline in performance suggests that exposure to prior knowledge during LLM training influenced code generation, which led to artificially higher resolution rates. By eliminating this factor, the models' true problem-solving capabilities are more accurately reflected.

3.6.5 Impact of Data Leak on Resolution Rates

Findings suggest that data leakage can significantly impact the evaluation of LLM performance in code generation. As discussed earlier, approximately 94% of the *SWE-bench* instances include issues and corresponding pull requests created before the *GPT-4 (1106)* training cut-off date. This raises the possibility that models were exposed to these examples during pretraining, which could artificially inflate resolution rates and give a misleading impression of real-world performance.

To assess this impact, we evaluated selected models using our newly curated dataset composed exclusively of issues created after the training cut-off dates of all models under study. The results demonstrated a consistent and substantial decline in performance across the majority

of the models. For example, *AutoCodeRover + GPT-4o* dropped from 18.83% on the original benchmark to 3.83% on the new dataset. Similarly, *SWE-Agent + GPT-4* declined from 4.78% to 1.99%, and *SWE-RAG + GPT-4* decreased from 1.13% to 0.73%. The only model that showed a marginal increase was *SWE-RAG + GPT-3.5*, rising from 0.17% to 0.55%; however, the change is minor and remains under 1%.

These results strongly suggest that previously reported resolution rates were at least partially influenced by the prior exposure to solutions during training. When evaluated on truly unseen data, the models' ability to reason about novel issues drops noticeably. This underscores the importance of eliminating data leakage to obtain a more accurate measure of a model's generalization capabilities and practical utility in code generation.

Beyond the overall decline in resolution rates, patch validation on the new post-cut-off dataset revealed that 67.73% of the model-generated fixes failed to solve the issues genuinely. In many instances, the models could not identify the root cause correctly, locate the buggy files or lines, or generate complete and relevant patches. It reinforces the notion that LLMs significantly struggle when deprived of prior exposure to the issue during training. In other words, when encountering unseen bugs, the models demonstrate limited problem-solving ability.

This challenge becomes increasingly important as LLMs are continuously retrained with access to more recent data. Without careful dataset curation, future benchmarks may inadvertently evaluate models on examples already seen during training, thus undermining the validity of reported performance. Ensuring benchmark instances are free from such leaks enables a

Table 3.4: Average cost of different models on *SWE-bench*

Model	**Avg cost per instance**	**Cost per issue fixing**	**Avg time per instance**
RAG+GPT-4	\$0.24	\$10.00	30 sec
RAG+GPT-3.5	\$0.05	\$32.50	30 sec
SWE-Agent+GPT-4	\$3.59	\$655.00	4 min
AutoCodeRover+GPT-4o	\$0.46	\$12.61	4.5 min

more faithful assessment of a model's capabilities and drives more meaningful progress in LLM-based software engineering research.

Findings suggest that there is a need for continuous dataset maintenance and stricter filtering criteria when selecting benchmark instances. Filtering strategies must adapt dynamically as the models evolve, this will mitigate leakage risks. Without such safeguards, benchmarks like *SWE-bench* may reflect memorization rather than true generalization, ultimately misrepresenting a model's real-world problem-solving potential.

3.6.6 Effectiveness-aware Evaluation

While models demonstrated varying degrees of resolution capability, we observed that these differences were also accompanied by significant variations in computational cost. Some models achieved higher accuracy and resolution rates, but they did so at the expense of greater computational resources, longer inference times, and higher monetary cost. Specifically, *SWE-Agent + GPT-4* and *AutoCodeRover + GPT-4* had the longest code generation times, with *SWE-Agent + GPT-4* averaging approximately 4 minutes per instance, resulting in a total of around 37 hours to generate patches for all *SWE-Bench+* issues.

This extensive runtime makes *SWE-Agent + GPT-4* not feasible for real-world issue resolution, where many issues are reported and require faster resolution. Developer-written fixes could be faster, defeating the purpose of automation. Additionally, this compute-heavy inference significantly limits its feasibility for large-scale or continuous deployment.

Similarly, *AutoCodeRover + GPT-4* required an average of 4.5 minutes per instance, totaling 41 hours to generate patches for all *SWE-Bench+* issues. Although it was one of the top performers in issue resolution, its high computational cost and long processing time present challenges for real-world adoption.

This disparity highlights the trade-offs between performance and cost-effectiveness, particularly for models like *SWE-Agent + GPT-4*, where balancing time, cost, and resource allocation is critical for practical applications.

To analyze cost efficiency, we considered two metrics: the average cost per instance and the effectiveness-aware cost per instance (calculated by dividing the total cost by the number of successfully resolved issues). The detailed cost breakdown of each model using these metrics is presented in Table 3.4.

Among all models, *SWE-Agent + GPT-4* was the most expensive, with an average cost of $0.24 per instance and an effectiveness-aware cost of $655.00 per resolved issue, as shown in Table 3.4. Although it outperforms other methods in resolution rate, it is very costly and would not be feasible to adopt in real-world scenarios, especially for organizations operating under budget constraints. In other words, developer-written fixes may cost less and take less time to produce.

Despite its high cost, *SWE-Agent + GPT-4* performance is comparable to *RAG + GPT-4*, which was significantly more cost-efficient, with an average cost of $0.24 per instance and an effectiveness-aware cost of $10.00. On the other hand, *AutoCodeRover + GPT-4* delivered the highest resolution rate of 3.83% among all models. While its average cost was relatively high ($0.46 per instance), its effectiveness-aware cost was relatively low at $12.61 per resolved issue, making it more cost-effective than *SWE-Agent + GPT-4*.

Meanwhile, *RAG + GPT-3.5* had the lowest average cost at $0.05 per instance, but due to its lower resolution rate, its effectiveness-aware cost was high at $32.50. Thus, despite its cost advantage per instance, it remains an inefficient choice when considering effectiveness-aware metrics.

Future research efforts should emphasize not only enhancing model accuracy but also balancing performance with financial cost associated with their deployment. As new models continue to be added to the *SWE-bench Leaderboard*, it is crucial to evaluate not only their resolution performance but also their cost-effectiveness. The focus should shift beyond just resolution rates to identifying models that maintain strong performance while minimizing costs, ensuring practical deployment at scale.

Chapter 4

Threats to Validity

The threats to the validity of this thesis can be summarized in two major points:

(1) **Test Case Reliability and Benchmark Limitations** The resolution rates of the models are determined solely based on test case results. However, test cases often lack comprehensive coverage, failing to capture edge cases and unintended side effects. As a result, relying exclusively on test case outcomes can lead to misleading success rates that appear correct but overlook critical scenarios.

While our work improves test cases and demonstrates their validity, test outcomes alone do not fully assess deeper semantic correctness. Developers possess a broader understanding of the problem, enabling them to produce patches that align with the intended solution. In contrast, LLM-generated patches may pass test cases but still be incorrect due to unintended regressions, dependencies, poor code quality, security risks, or lack of developer acceptance.

This highlights the need for evaluation methods beyond test case results to ensure correctness and reliability.

Since issues are classified as resolved merely based on passing test cases, human code reviews remain essential to validate whether LLM-generated patches are truly correct and acceptable. However, manual reviews are costly and time-consuming, underscoring the need for automated methods to assess code quality, maintainability, and correctness in a more efficient manner.

(2) **Limited Scope of Patch Validation and Potential Model Bias** Our patch validation study is based solely on the 251 instances successfully resolved by *SWE-Agent + GPT-4*. While this model was the top-performing open-source model on the *SWE-bench* leaderboard at the time of our study and provided valuable insights, our analysis remains inherently limited. Due to the high cost, time, and resource demands of manual patch validation, we focused exclusively on this model. However, to generalize the findings, patterns, and conclusions across the broader *SWE-bench* dataset, it is important to expand this validation to include multiple models

The identified benchmark-level issues such as solution leakage and weak test cases are expected to affect all models, as they reflect limitations in the dataset itself. However, other patterns we observed, such as incorrect or incomplete patches, or patches modifying different files or functions compared to the gold patch, may be specific to *SWE-Agent + GPT-4*. Different LLMs may be creating more correct patches or may introduce new problematic patterns not captured in our analysis. Since different models follow different reasoning paths and

strategies, analyzing their outputs through patch validation may uncover additional insights or limitations in the benchmark

This limitation also affects the scope of our benchmark enhancement proposals. The techniques we introduce, *SoluLeakDetector* and *TestEnhancer*, are designed to address the specific problems uncovered through GPT-4's outputs. While these methods target general benchmark flaws, their implementations may need to be expanded or adapted to remain effective when applied to outputs from other models. A more dynamic and flexible framework may be required to continuously capture and address emerging patterns as new LLMs are evaluated

In summary, while our patch validation study provides valuable insights into both dataset and model behaviors, it represents only a partial view. To fully generalize the findings and evaluate the robustness of *SWE-bench* across diverse LLMs, a broader validation effort involving multiple models is necessary

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we present the first empirical study on the robustness and reliability of the evaluation framework in the *SWE-bench* dataset. Our analysis reveals significant limitations in the original *SWE-bench*, particularly in how model-generated patches are evaluated against ground-truth developer fixes.

To investigate these limitations, we conducted an in-depth patch validation analysis of instances resolved by *SWE-Agent + GPT-4*. Specifically, we compared model-generated patches with gold patches from pull requests to identify systematic patterns. Through this analysis, we observed five distinct patterns in the generated patches, which we categorized into two groups: correct fixes and inaccurate, irrelevant or incomplete fixes that are labeled as "suspicious fixes."

The suspicious fixes revealed two critical issues with the dataset:

1. **Solution Leakage** – In several instances, the issue description provides explicit solutions or strong hints that guide the model toward the correct fix. This solution leakage allows models to "cheat," inflating their performance on *SWE-bench* significantly.

2. **Weak Test Cases** – In some cases, model-generated solutions are incorrect, incomplete, or irrelevant, yet they still pass the provided test cases. This suggests that the test suites lack the robustness needed to distinguish between correct and incorrect patches, leading to misleading evaluation results on the *SWE-bench Leaderboard*.

We also investigated the potential presence of data leakage and found that it significantly impacts model performance, highlighting that even models evaluated on this dataset may have been exposed to the solution during the training.

These issues compromise the reliability of *SWE-bench* as a benchmarking dataset and highlight the need for a more robust evaluation framework.

To address these challenges, we introduce *SWE-Bench+*, a refined evaluation dataset that filters out suspicious patches from *SWE-bench*. Specifically, we propose two key enhancements:

- `SoluLeakDetector` – A solution leak detection tool that leverages a three-shot prompting technique with *GPT-4* to systematically identify and remove issues affected by solution leakage.

- `TestEnhancer`– A test augmentation approach that strengthens test suites to mitigate weak test cases. *`TestEnhancer`* prompts *GPT-4* with additional context extracted

from *SWE-bench* to generate *PASS-TO-PASS* and *FAIL-TO-PASS* tests, ensuring a more rigorous assessment of model-generated patches.

Our evaluation demonstrates that *SWE-Bench+* achieves 86% accuracy in detecting solution-leak issues and reduces the number of suspicious patches caused by weak tests by 19%. Moreover, *SWE-Bench+* establishes a framework for the continuous, automated refinement of *SWE-bench*, paving the way for more reliable and robust benchmarking in the future.

5.2 Future Work

This thesis can be further strengthened with the following future explorations.

1. **Enhancing Solution Leak Detection:**

   While *SoluLeakDetector* has been effective in identifying solutions described within issue reports, its accuracy is influenced by the quality and design of the detection prompts. This reliance on *GPT-4*'s contextual understanding and natural language processing introduces limitations, potentially resulting in false positives or false negatives if subtle hints or indirect descriptions of solutions are overlooked. Additionally, the approach may struggle with issues that reference external documentation, previous commits, or related discussions, where solutions are implied rather than explicitly stated.

   Moreover, applying this technique to large datasets can become computationally expensive due to the high cost of *GPT-4*.

For future work, a more automated approach could be explored, such as fine-tuning dedicated models that are context-aware and capable of detecting subtle hints with higher accuracy. This would enhance the generalization of solution leak detection while maintaining a cost-efficient and scalable method.

2. **Improving Test Case Robustness:**

   TestEnhancer aims to address the weaknesses of existing test cases by generating complementary tests using *GPT-4*. However, its effectiveness is constrained by the model's ability to comprehend issue reports, code behavior, and expected outputs. In cases where *GPT-4* lacks sufficient domain knowledge or a deep understanding of the issue, it may generate test cases that are incomplete or irrelevant. Additionally, while prompts explicitly instruct the model to cover edge cases, there is no verification mechanism to ensure comprehensive coverage, leaving potential gaps in test robustness that could further reduce the accuracy of the original test cases.

   Moreover, the technique does not fully capture the broader project dependencies and interactions between different files, as the model is limited to the information explicitly provided in the prompt. This restriction may lead to generated test cases that fail to account for critical details necessary for complete and correct validation, ultimately weakening their effectiveness in evaluating code correctness.

   Another key limitation is the computational cost associated with test generation. Since *TestEnhancer* relies on *GPT-4*, scaling it to large datasets can be costly, making it impractical to scale.

For future directions, one could explore fine-tuning a model specifically for test case generation, edge-case detection, debugging, and coverage enhancement. A dedicated fine-tuned model could improve test coverage, generate more relevant test cases, and significantly reduce the cost associated with API usage, making automated test generation more efficient and scalable.

3. **Incorporating Cost-Effectiveness into Evaluation Metrics:**

   Currently, the *SWE-bench Leaderboard* primarily reports model performance in terms of accuracy, without accounting for the associated computational costs. This omission is significant, as cost plays a crucial role in real-world applications where resource efficiency is a key consideration. As observed in our analysis, models like *SWE-Agent + GPT-4* achieve high resolution rates but come with substantial computational expenses, whereas smaller models like *RAG + GPT-3.5* offer a more cost-efficient alternative at the expense of lower accuracy.

   Future research should integrate cost-aware evaluation metrics to provide a more comprehensive assessment of model performance. By explicitly considering the trade-offs between accuracy and computational cost, researchers and practitioners can make more informed decisions about model deployment, ensuring a balanced approach that reflects real-world feasibility.

4. **Adapting Data Leak Filtering for Evolving Model Cut-off Dates:**

As new models continue to be released with increasingly recent training data, the risk of data leakage grows, particularly as training cut-off dates approach or overlap with the timeframe of evaluation datasets. Future work should establish dynamic filtering methodologies that automatically adjust based on the cut-off dates of new models. This would ensure that future evaluations remain leakage-free, regardless of the pace of model updates, and maintain the validity of benchmarks as models become more frequently retrained on up-to-date codebases.

Bibliography

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[2] D. Chen, S. Lin, M. Zeng, D. Zan, J.-G. Wang, A. Cheshkov, J. Sun, H. Yu, G. Dong, A. Aliev, J. Wang, X. Cheng, G. Liang, Y. Ma, P. Bian, T. Xie, and Q. Wang, "Coder: Issue resolving with multi-agent and task graphs," 2024. [Online]. Available: https://arxiv.org/abs/2406.01304

[3] K. Zhang, W. Yao, Z. Liu, Y. Feng, Z. Liu, R. Murthy, T. Lan, L. Li, R. Lou, J. Xu, B. Pang, Y. Zhou, S. Heinecke, S. Savarese, H. Wang, and C. Xiong, "Diversity empowers intelligence: Integrating expertise of software engineering agents," 2024. [Online]. Available: https://arxiv.org/abs/2408.07060

[4] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.01489

[5] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[6] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," 2024. [Online]. Available: https://arxiv.org/abs/2404.05427

[7] R. L. Rosa, C. Hulse, and B. Liu, "Can github issues be solved with tree of thoughts?" 2024. [Online]. Available: https://arxiv.org/abs/2405.13057

[8] D. Zan, Z. Huang, A. Yu, S. Lin, Y. Shi, W. Liu, D. Chen, Z. Qi, H. Yu, L. Yu, D. Ran, M. Zeng, B. Shen, P. Bian, G. Liang, B. Guan, P. Huang, T. Xie, Y. Wang, and Q. Wang, "Swe-bench-java: A github issue resolving benchmark for java," 2024. [Online]. Available: https://arxiv.org/abs/2408.14354

[9] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[10] R. Aleithan, "SWE-bench+ Patterns: Patch Classification of SWE-agent + GPT-4 Resolved Issues," 2024. [Online]. Available: https://zenodo.org/records/13879453

[11] R. A. eithan, "swe-bench-patterns: Analysis of llm-generated patches on swe-bench," https://github.com/reemoo1100/swe-bench-patterns, 2024, gitHub repository.

[12] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[13] X. Li and T. Döhmen, "Towards efficient data wrangling with llms using code generation," in *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning*, ser. DEEM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 62–66. [Online]. Available: https://doi.org/10.1145/3650203.3663334

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[15] H. Luo, J. Wu, J. Liu, and M. F. Antwi-Afari, "Large language model-based code generation for the control of construction assembly robots: A hierarchical generation approach," *Developments in the Built Environment*, vol. 19, p. 100488, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666165924001698

[16] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639219

[17] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," 2024. [Online]. Available: https://arxiv.org/abs/2310.08879

[18] B. Yang, H. Tian, W. Pian, H. Yu, H. Wang, J. Klein, T. F. Bissyandé, and S. Jin, "Cref: An llm-based conversational software repair framework for programming tutors," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 882–894. [Online]. Available: https://doi.org/10.1145/3650212.3680328

[19] D. de Fitero-Dominguez, E. Garcia-Lopez, A. Garcia-Cabot, and J.-J. Martinez-Herraiz, "Enhanced automated code vulnerability repair using large language models," *Engineering Applications of Artificial Intelligence*, vol. 138, p. 109291, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197624014490

[20] K. Alrashedy and A. Binjahlan, "Language models are better bug detector through code-pair classification," 2024. [Online]. Available: https://arxiv.org/abs/2311.07957

[21] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660773

[22] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," 2023. [Online]. Available: https://arxiv.org/abs/2310.03533

[23] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2024. [Online]. Available: https://arxiv.org/abs/2308.10620

[24] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J.-G. Lou, "Large language models meet nl2code: A survey," 2023. [Online]. Available: https://arxiv.org/abs/2212.09420

[25] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," 2024. [Online]. Available: https://arxiv.org/abs/2311.10372

[26] Y. Wang, H. Le, S. Joty, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[27] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao, Y. Guo, and Q. Wang, "Pangu-coder2: Boosting large language models for code with ranking feedback," 2023. [Online]. Available: https://arxiv.org/abs/2307.14936

[28] F. F. Xu, V. J. Hellendoorn, R. Singh, P. Maniatis, and A. Terzis, "A systematic evaluation of large language models of code," *arXiv preprint arXiv:2202.13169*, 2022.

[29] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[30] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[31] B. Roziere, G. Izacard, S. Withnall, V. Dinh, C. Lovett, K. Simonyan, Y. Belinkov, S. Edunov, V. Karpukhin, A. Parikh *et al.*, "Code llama: Open foundation models for code," 2023. [Online]. Available: https://arxiv.org/abs/2308.12950

[32] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.

[33] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 298–309.

[34] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[35] H. Liu, J. Chen, Y. Xiong, D. Wang, and L. Zhang, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[36] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," in *IEEE Transactions on Software Engineering*. IEEE, 2019.

[37] T. Lutellier, W. Yu, L. Tan, and V. Le, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.

[38] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[39] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[40] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," 2022. [Online]. Available: https://arxiv.org/abs/2207.08281

[41] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *arXiv preprint arXiv:2403.17134*, 2024.

[42] F. Li, J. Jiang, J. Sun, and H. Zhang, "Hybrid automated program repair by combining large language models and program analysis," *arXiv preprint arXiv:2406.00992*, 2024.

[43] P. Wang, S. Jain, E. Zelikman, E. Shi, J. Wu, P. Liang, and T. B. Hashimoto, "Swe-agent: Autonomously fixing software bugs using large language models," *arXiv preprint arXiv:2305.13214*, 2023. [Online]. Available: https://arxiv.org/abs/2305.13214

[44] Z. Yu, X. Lu, F. Wang, and Y. Xue, "Autocoderover: Revisiting program repair from a retriever-generator perspective," *arXiv preprint arXiv:2403.19827*, 2024. [Online]. Available: https://arxiv.org/abs/2403.19827

[45] R. Bavishi, H. Yoshida, M. Sridharan, S. Chandra, M. Kulkarni, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 613–624.

[46] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 151–162.

[47] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[48] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 209–219.

[49] A. Carvalho, W. Luz, D. Marcílio, R. Bonifácio, G. Pinto, and E. D. Canedo, "C-3pr: A bot for fixing static analysis violations via pull requests," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 161–171.

[50] C. Calcagno and D. Distefano, "Moving fast with software verification," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 3–11.

[51] M. M. Mohajer, R. Aleithan, N. S. Harzevili, M. Wei, A. B. Belle, H. V. Pham, and S. Wang, "Skipanalyzer: An embodied agent for code analysis with large language models," *CoRR*, 2023.

[52] SpotBugs Contributors, "SpotBugs: Find Bugs in Java Programs," https://spotbugs.github.io/, 2020, accessed: 2025-04-06.

[53] Google Developers, "Error Prone: Catch common Java mistakes at compile-time," https://errorprone.info, 2020, accessed: 2025-04-06.

[54] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," 2023. [Online]. Available: https://arxiv.org/abs/2304.02014

[55] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," 2023. [Online]. Available: https://arxiv.org/abs/2212.14834

[56] M. Allamanis, P. Chanthirasegaran, M. Brockschmidt, O. Polozov, and D. Tarlow, "Self-supervised bug detection and repair," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, 2021, pp. 25 638–25 650.

[57] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 1536–1547.

[58] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization,"

in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 107–119.

[59] Z. Sun, L. Li, Y. Liu, X. Du, and L. Li, "On the importance of building high-quality training datasets for neural code search," *CoRR*, vol. abs/2202.06649, 2022. [Online]. Available: https://arxiv.org/abs/2202.06649

[60] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment "translation": Data, metrics, baselining & evaluation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 746–757.

[61] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "Neural code summarization: How far are we?" *arXiv preprint arXiv:2107.07112*, 2021.

[62] Y. Huang, N. Jia, J. Shu, X. Hu, X. Chen, and Q. Zhou, "Does your code need comment?" *Software: Practice and Experience*, vol. 50, no. 3, pp. 227–245, 2020.

[63] S. Roy, M. Allamanis, and C. Sutton, "On the reliability of automatic evaluation in code summarization," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1147–1158.

[64] A. Mahmud, S. Roy, and M. M. Rahman, "Code summarization: How far have we come?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1077–1089.

[65] W. Xu, S. Wang, X. Zhang, L. Ou, X. Wang, and B. Sun, "How good is the software produced by github copilot? an exploratory study," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022, pp. 1213–1225.

[66] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," 2024. [Online]. Available: https://arxiv.org/abs/2403.17927

[67] Y. Zheng, X. Mu, Y. Wang, H. Zhang, H. Zhao, Y. Gao, X. Zou, X. Jin, V. Srikumar, K.-W. Chang *et al.*, "Code evaluation with enhanced test suites," *arXiv preprint arXiv:2305.01210*, 2023.

[68] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.