Experiments with Large Language Models in Software Engineering Tasks

Ranjana Raghavan
ranjanaraghavan911@gmail.com

Keywords: Large Language Models, Software Engineering, SWE-Bench, Python, Github, Software benchmarking.

Abstract. In this document, we analyze the performance of Large Language Models in real-world Software Engineering tasks such as program repair and raising pull-requests. The outline of the paper is as follows - first, we review the role of Large Language Models in Software Development lifecycle, then give an overview of the related work in this area before outlining the experiments conducted. The aim of the study is to assess the success/failure modes of these LLMs in benchmarks such as SWE-Bench. Finally, we set out to answer the question ourselves by conducting some experiments where we test these LLMs locally on a custom set of PRs.

1 Introduction

Given the practical value and advances in reasoning, Large Language Models (LLMs) have emerged as powerful tools in Software Development lifecycle, leading to popular products such as Github Copilot [1], Cursor and Windsurf. This paper presents a performance analysis of LLM's on real-world Software Engineering tasks, with a focus on solving pull requests (PRs). We begin by providing an overview of LLMs in the context of Software Engineering, highlighting their potential to revolutionize code generation, analysis, and maintenance[2].

Our study then delves into the current state-of-the-art performance of LLMs on SWE-Bench, a benchmark designed to evaluate AI models on software engineering tasks. We critically examine success and failure modes of some of these models, shedding light on their strengths and limitations in addressing complex coding challenges[3].

To further investigate LLMs' capabilities, we conduct a series of experiments using a custom set of pull requests. These experiments aim to provide a more nuanced understanding of how LLMs perform in real-world scenarios, beyond standardized benchmarks. Our findings offer valuable insights into the practical applicability of LLMs in software development workflows and their potential impact on developer productivity[4].

By combining an analysis of existing benchmarks with our custom experiments, this paper contributes to the ongoing discourse on the role of LLMs in Software Engineering. Our results not only highlight the current capabilities of these models but also identify areas for improvement and future research directions in the field of AI-assisted software development[5].

2 Related Work

The application of Large Language Models (LLMs) to the software development lifecycle has been an area of intense research and development in recent years. This section presents an overview of the existing literature, focusing on the most recent advancements.

2.1 LLMs in Software Development Processes

Li et al. (2025) conducted a groundbreaking case study, DevEval exploring the performance of LLMs across the entire software development lifecycle[6]. DevEval is a benchmark encompassing software design, environment setup, implementation, acceptance testing, and unit testing. The study revealed that even advanced models like GPT-4 struggled with the comprehensive challenges presented in DevEval, highlighting the complexity of applying LLMs to real-world programming tasks. In 2024, Ren et al. surveyed the landscape of LLMs in software engineering, identifying open problems and future research directions[4]. Their work provided a comprehensive overview of the state of the art and set the stage for subsequent research in 2025.

Along with development of Large Language Models, there has been work in parallel to develop benchmarks. Jimenez et al. (2023) introduced SWE-bench, a benchmark designed to evaluate LLMs' ability to resolve real-world GitHub issues[3]. This work laid the foundation for more comprehensive evaluations in 2024 and 2025. Building on these efforts, the LLM4SE 2025 workshop focused specifically on applying LLMs to software engineering tasks[7]. This workshop brought together researchers and practitioners to discuss the potential of LLMs in automating and augmenting various aspects of the software development lifecycle.

Industry reports and analyses have also contributed significantly to our understanding of LLMs in software development. A comprehensive guide published by Turing in 2025 explored how LLMs work, their benefits, challenges, and emerging trends in the context of software engineering[8]. Furthermore, Turintech's analysis of AI's evolution in software development predicted that by 2025, static code generation would give way to evolutionary code optimization, leveraging genetic algorithms and other advanced techniques[9].

In conclusion, the rapid evolution of LLMs in software engineering in recent years has shown both the immense potential and the significant challenges in applying these models to the full software development lifecycle. As the field continues to advance, researchers and practitioners are focusing on creating more efficient, interpretable, and ethically aligned models that can truly transform software development practices.

3 SWE Bench

SWE-bench is an AI evaluation benchmark that assesses a model's ability to complete real-world software engineering tasks. Specifically, it tests how the model can resolve GitHub issues from popular open-source Python repositories. For each task in the benchmark, the AI model is given a Python environment and the checkout (a local working copy) of the repository from just before the issue was resolved. The model then needs to understand, modify, and test the code before submitting its proposed solution.

Each solution is graded against the real unit tests from the pull request that closed the original GitHub issue. This process tests whether the AI model was able to achieve the same functionality as the original human author of the PR.

SWE-bench doesn't just evaluate the AI model in isolation, but rather an entire ”agent” system. In this context, an ”agent” refers to the combination of an AI model and the software scaffolding around it. This scaffolding is responsible for generating the prompts that go into the model, parsing the model's output to take action, and managing the interaction loop where the result of the model's previous action is incorporated into its next prompt. The performance of an agent on SWE-bench can vary significantly based on this scaffolding, even when using the same underlying AI model.

A 3-stage pipeline was used to select high-quality task instances from GitHub repositories. We briefly outline the process here:

- **Repository selection and data scraping**: Authors began by collecting pull requests (PRs) from 12 popular open-source Python repositories on GitHub, resulting in approximately 90,000 PRs. The focus was on popular repositories because they tend to be better maintained, have clear contributor guidelines, and have better test coverage. Each PR was associated with a codebase specified by its base commit.

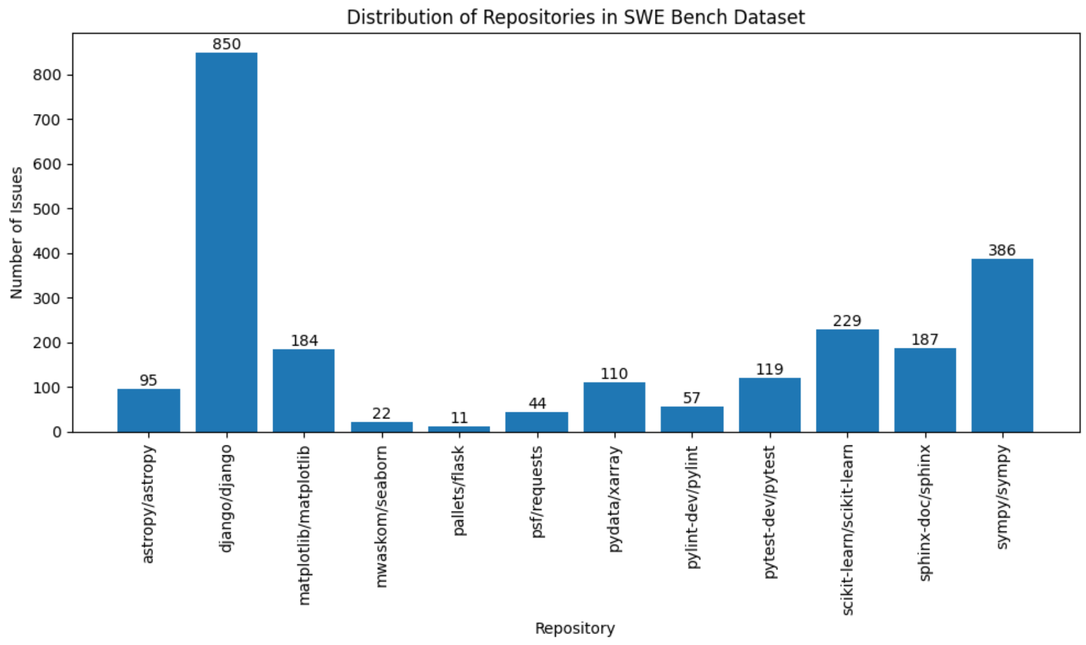


Figure 1: **Repository distribution** Distribution of SWE-bench tasks across 12 open source GitHub repositories that each contains the source code for a popular, widely downloaded PyPI package.

- **Attribute-based filtering**: Candidate tasks were created by selecting merged PRs that met two criteria: (1) they resolved a GitHub issue, and (2) they made changes to the test files of the repository, indicating that the user likely contributed tests to check whether the issue had been resolved.
- **Execution-based filtering**: For each candidate task, the PR's test content was applied, and the associated test results were logged. Task instances were filtered out if they did not have at least one test where the status changed from ”fail” to ”pass” (referred to as a ”fail-to-pass” test). Additionally, instances that resulted in installation or runtime errors were removed.

Through these filtering stages, the initial 90,000 PRs were reduced to the 2,294 task instances that constitute SWE-bench.

4 Experiments

Our initial research objective was to assess the effectiveness of Large Language Models (LLMs) using the SWE-Bench framework. However, due to resource constraints and technical complexities, we had to modify our approach. Specifically this would cost significant computational expenses, besides the requirement to develop scaffolding to handle prompt generation, output parsing, and managing the interaction loop. In light of these constraints, we adapted our methodology, instead of utilizing SWE-Bench, we developed a local evaluation approach:

- **Custom test suite**: We created our own set of test cases tailored to assess LLM performance.

- **Local evaluation**: We conducted the assessment of Large Language Models using our custom test suite on local infrastructure.

This adjustment allowed us to maintain the core objective of evaluating LLMs while working within our available resources and technical capabilities.We evaluated the performance of locally hosted Large Language Models (LLMs), specifically Lllama3 via **Ollama** [10], on software engineering tasks. Meta's Llama 3, released in April 2024, represents a significant advancement in open-source large language models. Available in 8B and 70B parameter versions, Llama 3 demonstrates improved reasoning abilities, code generation, and instruction following compared to its predecessors, while also outperforming other open models on various language understanding and response benchmarks. We have used **OllamaModel** APIs to manage API interactions with the models running in locally hosted Ollama server, including prompt construction and response processing.

We created a synthetic dataset to evaluate the ability of language models to automatically resolve real-world software engineering issues. Each entry provides a context-rich scenario, including a problem description, the buggy code, and the corrected code. This format allows for a clear evaluation of a language model's ability to understand, diagnose, and fix bugs in a variety of popular Python libraries. A custom JSON-based test case format was implemented for initial testing, alongside basic similarity scoring and exact match metrics for evaluation - the dataset is aimed at automating code repair and improving software development workflows.

5 Results & Evaluation

We used the following criteria to evaluate output from the language model with the ground truth. These metrics provide a nuanced view of structural and semantic similarity.

- Line-based similarity (30% weight): Simple metric comparing lines between output and ground truth.
- Token-based similarity (40% weight): Compares code tokens after normalization. The normalization step involves removing comments and empty lines, standardizes whitespace and converts to lowercase for token comparison.
- Structure similarity (20% weight): Compares code structure (functions, conditionals, etc.).The metric aims to identify key structural elements (def, class, if, for, etc.) and compares the overall code organization.
- Length similarity (10% weight): Considers overall code length differences

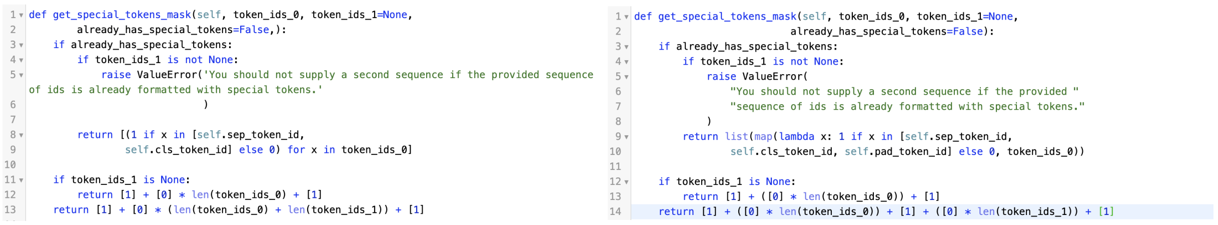
We analyze two representative cases from our evaluation: a successful case of program repair with high similarity (87.85%) and a failure case with low similarity (39.67%).

Success Case: From Figure 2, we can see that the AI generated code has a perfect structural similarity (100%) - maintains the same control flow. We also observe the correct API usage with high token similarity (92.59%). The generated code only missed including `pad_token_id` in the token check. Although the generated code performs a list comprehension, it doesn't affect functionality as compared to the `map` API.

Failure Case: From Figure 3, we can see there is a low line similarity (16.67%) - completely different approach. It uses the iteration instead of vectorized operations, however, misses the core optimization of using `asi8` for performance. It also handles NaT values differently and less efficiently. The generated code also unnecessarily complicates timezone handling.

Problem	Line	Token	Structure	Length	Overall
PyTorch: Empty tensor handling in `_make_tensor`	66.67	92.86	71.43	98.49	**81.28**
Transformers: Padded sequence handling in `get_special_tokens_mask`	71.43	92.59	100.00	93.83	**87.85**
Pandas: Mixed timezone handling in `to_numpy`	16.67	50.00	40.00	66.67	**39.67**
Requests: SSL certificate verification bypass	46.15	90.32	33.33	90.81	**65.72**
FastAPI: Default value handling in dependency injection	18.92	81.67	28.57	84.04	**52.46**

Table 1: **Similarity Metrics for Generated Solutions**. The table shows different similarity metrics for each test case. Higher percentages indicate better matches with the expected solutions. Token similarity consistently shows higher scores, suggesting that the model maintains good API usage even when the exact implementation differs.



```python
def get_special_tokens_mask(self, token_ids_0, token_ids_1=None,
        already_has_special_tokens=False,):
    if already_has_special_tokens:
        if token_ids_1 is not None:
            raise ValueError('You should not supply a second sequence if the provided sequence of ids is already formatted with special tokens.'
                            )

        return [(1 if x in [self.sep_token_id,
                self.cls_token_id] else 0) for x in token_ids_0]

    if token_ids_1 is None:
        return [1] + [0] * len(token_ids_0) + [1]
    return [1] + [0] * (len(token_ids_0) + len(token_ids_1)) + [1]
```

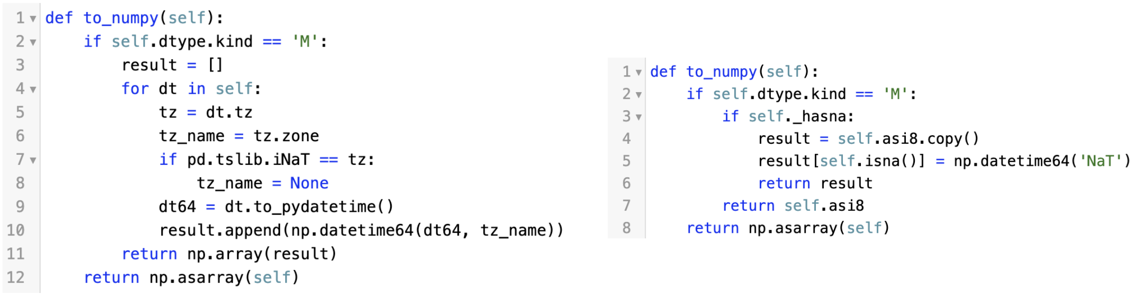
```python
def get_special_tokens_mask(self, token_ids_0, token_ids_1=None,
                            already_has_special_tokens=False):
    if already_has_special_tokens:
        if token_ids_1 is not None:
            raise ValueError(
                "You should not supply a second sequence if the provided "
                "sequence of ids is already formatted with special tokens."
            )
        return list(map(lambda x: 1 if x in [self.sep_token_id,
                self.cls_token_id, self.pad_token_id] else 0, token_ids_0))

    if token_ids_1 is None:
        return [1] + ([0] * len(token_ids_0)) + [1]
    return [1] + ([0] * len(token_ids_0)) + [1] + ([0] * len(token_ids_1)) + [1]
```

Figure 2: Figure showing a success case with AI generated code at **87.85%** similarity. **Prompt** : Fix the bug in the Transformers library where the `get_special_tokens_mask` method doesn't properly handle already padded sequences. The generated code is in the left whereas the correct code in on the right.

From our success and failure cases, the success case maintains the core structure and API usage while potentially missing minor details. We also infer that token similarity alone isn't sufficient - the failure case had 50% token similarity but poor overall performance. In addition, simpler, more performant solutions are sometimes overlooked in favor of more complex approaches. Structural and semantic similarity is a strong indicator of solution quality.

6 Socio-economic impact of Large Language Models on SDE lifecycle

In this section we briefly touch upon the socio-economic impact of AI in software engineering.LLMs demonstrate measurable efficiency improvements in code generation, debugging, and documentation tasks. From SWE-Bench benchmarks, researchers estimate that 65% of software



```python
def to_numpy(self):
    if self.dtype.kind == 'M':
        result = []
        for dt in self:
            tz = dt.tz
            tz_name = tz.zone
            if pd.tslib.iNaT == tz:
                tz_name = None
            dt64 = dt.to_pydatetime()
            result.append(np.datetime64(dt64, tz_name))
        return np.array(result)
    return np.asarray(self)
```

```python
def to_numpy(self):
    if self.dtype.kind == 'M':
        if self._hasna:
            result = self.asi8.copy()
            result[self.isna()] = np.datetime64('NaT')
            return result
        return self.asi8
    return np.asarray(self)
```

Figure 3: Figure showing a success case with AI generated code at **39.67%** similarity. Fix the bug in pandas where the `to_numpy` method doesn't properly handle datetime64 arrays with mixed timezones. The generated code is in the left whereas the correct code in on the right

engineering tasks exhibit high exposure to automation via LLMs. These productivity gains translate to economic value through reduced development cycles and lower operational costs. For example, enterprises adopting LLM-powered IDEs report 20–40% reductions in time-to-market for new features [11]. However, these benefits are unevenly distributed: organizations with existing AI infrastructure capture disproportionate value, potentially widening the gap between tech giants and smaller firms.

The economic impact extends beyond direct cost savings. By lowering barriers to entry, LLMs enable non-specialists to participate in software development through natural language interfaces. This ”democratization effect” could stimulate innovation in underserved markets but risks flooding the sector with low-quality code outputs.

Contrary to simplistic displacement narratives, research reveals complex labor dynamics. While LLMs automate specific tasks (e.g., boilerplate code generation, test case writing), they increase demand for hybrid roles combining technical oversight and prompt engineering skills. The OpenAI/University study finds that 80% of software engineering occupations face partial LLM exposure, but only 4% of roles risk full automation [12].

Conclusion

This report studies the role of Large Language Models in the Software Development Lifecycle. We surveyed existing benchmarks such as SWE-Bench and conducted our own experiment to understand the efficacy of models in real-word software engineering scenarios. Finally, we interpret the results of the model using our own evaluation metric and briefly discuss the socio-economic impact.

Availability of software code

Our code is available at the following URL: `https://github.com/RanjanaRaghavan/swe-bench-evaluation`.

References

[1] GitHub and Microsoft Research. The economic impact of ai pair programmers. Technical report, GitHub Technical Report, 2023.

[2] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2024.

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

[4] Hao Ren, Xin Peng, Haonan Li, Zihan Li, Yilin Zhang, Wenqing Chu, Mingwei Liu, Xuanyi Li, Zhuobing Han, Xiaoxue Wu, Zhaoyi Liu, and Tao Xie. Large language models for software engineering: Survey and open problems, 2023.

[5] Xin-Cheng Wen, Jia-Hui Ding, Shi-Min Li, Xin-Yi Wang, Xin-Yi Huang, Xin-Yu Wang, Qing-Hua Zheng, and Zhi-Hua Zhou. Towards effective and reliable machine learning for software engineering: A survey, 2023.

[6] Jingfeng Li, Xiao Zhang, Yixin Chen, and Bolin Wang. Deveval: Evaluating large language models for full software development lifecycle. In *Proceedings of the 30th International Conference on Computational Linguistics*. Association for Computational Linguistics, 2025.

[7] Proceedings of the 2nd workshop on large language models for software engineering. In *2025 IEEE/ACM 47th International Conference on Software Engineering Companion (ICSE-Companion)*. IEEE, 2025.

[8] Turing. Large language models in software development: A comprehensive guide, 2025.

[9] Turintech Research Team. The evolution of ai in software development: From static code generation to evolutionary optimization. Technical report, Turintech, December 2024.

[10] Ollama. Ollama: Open source large language models. `https://ollama.com`, 2024. Accessed: 2025-02-16.

[11] P. A. Naeini et al. Cognitive augmentation through llms in software development. In *Proceedings of the ACM SIGSOFT Symposium*, pages 1123–1135, 2021.

[12] OpenAI and University of Pennsylvania. The impact of ai on the future of work. *arXiv preprint arXiv:2303.10130*, 2023.