ProjectTest: A Project-level LLM Unit Test Generation Benchmark and Impact of Error Fixing Mechanisms

Anonymous ACL submission

Abstract

Unit test generation has become a promising and important use case of LLMs. However, existing evaluation benchmarks for assessing LLM unit test generation capabilities focus on function- or class-level code rather than more practical and challenging project-level codebases. To address such limitation, we propose ProjectTest, a project-level benchmark for unit test generation covering Python, Java, and JavaScript. ProjectTest features 20 moderate-sized and high-quality projects per language. We evaluate nine frontier LLMs on ProjectTest and the results show that all frontier LLMs tested exhibit moderate performance on ProjectTest on Python and Java, highlighting the difficulty of ProjectTest. We also conduct a thorough error analysis, which shows that even frontier LLMs, such as Claude-3.5-Sonnet, have significant basic yet critical errors, including compilation and cascade errors. Motivated by this observation, we further evaluate all frontier LLMs under manual error-fixing and self-error-fixing scenarios to assess their potential when equipped with error-fixing mechanisms.

1 Introduction

Unit testing plays an important role in software development, helping identify bugs and ensuring codes are solid and maintainable. Writing unit tests is time-consuming, usually accounting for approximately 15.8% of software development time for developers (Daka and Fraser, 2014). Therefore, automated test case generation, like search-based (Fraser and Arcuri, 2011; Harman and McMinn, 2009), constraint-based (Xiao et al., 2013), and random-based (Pacheco et al., 2007) methods, has been proposed to create unit tests. However, the generated unit tests are usually less readable than manually-written tests and limited to certain types of functions (Grano et al., 2018). Lately, large language models (LLMs) have become game-changers, significantly accelerating unit test generation and improving readability and generalizability with little to no human effort (Siddiq et al., 2024; Xie et al., 2023).

Given the rapid adoption of LLMs on unit testing, the evaluation of LLM unit test generation capabilities appears to be lagging behind. Previous unit test generation evaluation benchmarks primarily focus on function-level, class-level, or file-level (Chen et al., 2021; Du et al., 2023; Wang et al., 2024; Jain et al., 2024a) codes. However, project-level codes are more representative of real-world scenarios and practical needs. The complex dependency relationships between different files in project-level codebases make unit test generation more challenging. The only existing benchmark that has briefly explored project-level unit test generation is DevBench (Li et al., 2024). However, due to its broad focus, the number of projects included for unit test generation is low for each language (e.g., 5 for Java and 5 combined for C and C++), with varying quality. Half of its projects for unit test generation evaluation are difficult to track, and most of the identifiable projects have fewer than 250 Stars and fewer than 50 Forks. DevBench also does not provide a thorough analysis of error types, potentials, or self-fixing capabilities of frontier LLMs' project-level unit test generation.

Therefore, we propose a new project-level unit test generation evaluation benchmark, ProjectTest, to offer a larger, higher-quality project set for project-level unit test generation along with a more thorough error analysis of frontier LLMs on unit test generation. ProjectTest covers three programming languages: Python, Java, and JavaScript. For each programming language, we construct 20 projects filtered from GitHub[1]. ProjectTest applies clear filtering criteria to select projects. It includes moderate-sized projects with multiple files and de-

[1] https://github.com/

pendencies between them. Each project has less than 1,600 lines of code, which fits within the maximum input length of most code language models. Quality is ensured by the number of stars and forks.

We evaluate nine frontier LLMs, such as Claude-3.5-Sonnet (Anthropic, 2024), Gemini-1.5-Pro (Team et al., 2024b), and GPT-o1, on ProjectTest and conduct comprehensive error analyses. We find that all tested frontier LLMs perform moderately on ProjectTest on Python and Java, highlighting the difficulty of ProjectTest. We also observe that different LLMs have different language-level expertise. Claude-3.4-Sonnet ranks first in Java, while GPT-o1 ranks first in JavaScript. Among three programming languages, Java is the most difficult language, primarily due to stricter syntax. Among all the tested models, GPT-o1 performs the best in general, especially in JavaScript.

Error analyses from above also show that even frontier LLMs, like Claude-3.5-Sonnet, have significant compilation and cascade errors. Although these errors appear to be preliminary and may be relatively easy to fix, they prevent us from observing more advanced aspects of LLM performance on unit test generation, such as correctness and coverage. To address this, we first manually fix LLM's compilation and cascade errors and then re-evaluate the fixed unit tests. This allows us to measure not only the models' raw performance but also their potential for improvement when combined with error-fixing mechanisms. By incorporating error-fixing, we uncover critical insights into the effort required to refine generated tests and better understand the various types of errors that occur in unit tests generated by different LLMs. We observe that the model rankings change significantly after the manual fix, showing the significant differences in different LLMs' error distribution and their potentials after error-fixing. Inspired by such findings from manual fixes, we also explore using LLMs for self-fixing their errors in generating project-level unit tests. The results show that while LLMs can correct some errors in their generated unit tests, their self-fixing abilities still lag behind the quality and reliability of human fixes.

We summarize our contributions as follows: we introduce the first project-level evaluation benchmark for unit test generation and conduct an extensive evaluation of nine frontier LLMs. Additionally, we conduct thorough error analyses by manually fixing compilation and cascade errors and provide critical insights. Inspired by the error analysis, we are the first to assess LLMs' self-fixing capability on unit test generation.

2 Related Work

2.1 Traditional Unit Test Generation

Traditional unit test generation methods employ search-based (Harman and McMinn, 2009; Fraser and Arcuri, 2011; Lukasczyk and Fraser, 2022), constraint-based (Xiao et al., 2013), or random-based (Pacheco et al., 2007) strategies to construct test suites that maximize code coverage. Although these traditional approaches can generate unit tests with reasonable coverage, the resulting tests often have lower readability and less meaningfulness compared to developer-written tests. As a result, automatically generated tests are frequently not directly adopted by practitioners in real-world scenarios (Almasi et al., 2017; Grano et al., 2019).

2.2 LLM-enhanced Unit Test Generation

Large Language Models have demonstrated strong code generation capabilities, inspiring their use in automated unit test generation. Recent approaches in LLM-enhanced unit test generation leverage zero-shot strategies (Siddiq et al., 2024), iterative querying (Schäfer et al., 2023), fine-tuning on specialized datasets (Alagarsamy et al., 2024), adaptive context selection (Xie et al., 2023), and focusing on subtle code differences (Dakhel et al., 2024; Li et al., 2023). These methods are evaluated with various metrics—including compilation success, test correctness, coverage, and bug detection—and demonstrate that LLMs can effectively surpass traditional test generation techniques.

2.3 Unit Test Generation Benchmark

Current benchmarks for LLM-based unit test generation mainly focus on function-level (Wang et al., 2024), class-level (Du et al., 2023), or file-level code (Jain et al., 2024a). Project-level software testing benchmarks, on the other hand, often target tasks other than unit test generation. For instance, R2E-Eval1 (Jain et al., 2024b) is designed for the generation of equivalent test harnesses, SWT-Bench (Mündler et al., 2024) focuses on fixing specific bugs rather than entire projects, and DevBench (Li et al., 2024) centers on software development tasks. While DevBench touches on project-level unit testing, its dataset is limited in quantity and varies in quality, especially for C/C# and Java, with only five projects each. Moreover, due to its

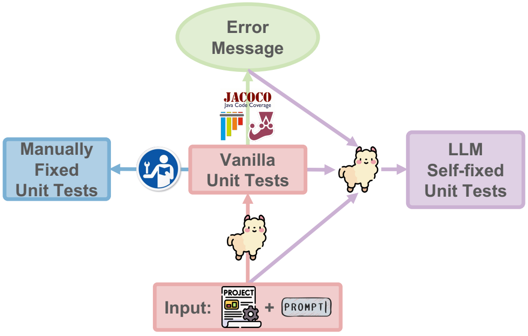

Figure 1: Overview of the unit test generation process.

broad focus, DevBench lacks a comprehensive evaluation and error analysis of LLM project-level unit test generation.

3 Methodology

We first introduce the dataset collection and preprocessing of creating ProjectTest (§3.1). Then, we introduce evaluation metrics (§3.2) and the Unit Test Generation pipeline (§3.3) that we use to evaluate LLMs on ProjectTest, including three unit test generation scenarios.

3.1 Benchmark Dataset

Dataset Collection. Our dataset is built from carefully selected project-level repositories on GitHub, which is a popular platform for hosting and collaborating on software development projects. We focus on three widely used programming languages: Python, Java, and JavaScript. We establish our selection criteria based on three key factors: 1) a reasonable size, 2) inter-file dependencies, and 3) a reliable source. Thus, we collect reliable and self-contained projects consisting of 2 to 15 files with fewer than 1600 lines of code (LOC). We limit our selection to repositories with publicly available licenses, such as the MIT license, ensuring the legality and openness of the code. To maintain the quality and reliability of the dataset, we choose projects with a high number of stars and forks, which signals community approval and widespread usage. For projects that fit all the requirements above but are too big for current frontier LLMs to handle, we also extracted smaller projects from these large codebases. These smaller projects were carefully adjusted to be self-contained without relying on the original larger projects. After applying these criteria, we constructed 20 representative projects per programming language. The summary of dataset statistics is presented in Table 1, and detailed information on the dataset sources and statistics for each project can be found in Appendix A.

Table 1: ProjectTest data statistics. LOC represents lines of code.

Language	Avg. #Files	Avg. LOC	Avg. #Stars	Avg. #Forks
Python	6.10	654.60	5810.30	996.90
Java	4.65	282.60	3306.05	1347.65
JavaScript	4.00	558.05	17242.30	5476.45

Pre-processing. Dataset pre-processing involves several key steps to ensure the projects are well-structured and suitable for testing. First, we double-check whether the selected projects have syntax errors, even though they are sourced from reliable codebases. Second, for projects extracted from a larger codebase, we modify them to be self-contained by reorganizing files, adjusting domain naming conventions, and/or modifying import paths to remove dependencies on external modules. Next, to enhance the accuracy of line coverage measurements, we consolidate statements that are split across multiple lines into a single line, ensuring that the metrics are more valid. Additionally, we maintain the integrity of the original code style as much as possible, preserving the diverse coding practices across different projects. This approach allows us to test how LLMs handle various code styles in a realistic environment.

3.2 Evaluation Metrics

We focus on three key aspects when evaluating the generated unit tests: compilation rate, correctness rate, and coverage rate. *Compilation rate* (ComR) measures the percentage of projects in which the generated test suites compile successfully, indicating how often LLMs produce unit test suites that can be executed without compilation errors. The compilation rate for all projects in X is defined as $ComR = \frac{|X^{com}|}{|X|}$, where X is the project set and $X^{com} \subset X$ denotes the subset of projects whose test suites compile successfully. *Correctness rate* (CR) calculates the percentage of unit tests that are correct out of all generated unit tests for each project, providing insight into the accuracy of the test generation process. On average, more than 95% of vanilla-generated unit tests compare expected and actual values, reinforcing the validity of CR as an evaluation metric. Detailed statistics see Appendix C. The correctness rate for the project x is defined as $CR_x = \frac{|T_x^{cor}|}{|T_x|}$, where T_x is the generated test suite and $T_x^{cor} \subset T_x$ denotes the correct unit test set for the project x. *Coverage rate* analyzes both line and branch coverage to understand how well the generated unit tests ex-

```python
# stock.py
from structure import Structure
from validate import String, ...
class Stock(Structure):
    ...
# structure.py
from validate import Validator, validated
class Structure:
    ...
# validate.py
class Validator:
    ...
```

Figure 2: An example of ProjectTest.

plore the code's functionality. The coverage rate for the project x is defined as $CR_x = \frac{covered(x)}{total(x)}$, where $covered(x)$ denotes the number of covered lines/branches in project x and $total(x)$ the total number of lines/branches in project $x \in X$.

These three evaluation metrics are not independent. If a project has the generated test suite containing compilation errors, none of its unit tests can be executed successfully, leading to both the correctness rate and the coverage rate for the project being zeros. Additionally, some errors resulting in failed tests, like missing Python dependencies, can also lead to a change in coverage rate. Therefore, considering the interdependencies between the three evaluation metrics, we extend our analysis beyond the evaluation of vanilla unit tests to include manually fixing these errors. This enables a more comprehensive assessment of LLMs' potential to generate high-quality unit tests once these errors are addressed. This assessment is conducted while maintaining the same quantity and diversity of unit tests originally generated by the LLMs. Furthermore, we extend our analysis to examine the self-fixing capabilities of LLMs.

3.3 Unit Test Generation

Figure 1 shows an overview of the unit test generation process by LLMs. Our unit test generation and evaluation aim to ensure fair and thorough assessments of unit tests generated by LLMs under different scenarios:

- **Scenario 1**: Vanilla unit tests extracted from LLMs' outputs.
- **Scenario 2**: Compilable unit tests after manually fixing all compilation and cascade errors.
- **Scenario 3**: Unit tests refined by LLMs self-fixing, provided with error messages and conversation history.

Scenario 1: Vanilla Unit Test Generation. We begin by inputting the entire project and the carefully crafted prompt into the LLM, ensuring the context and requirements are clearly communicated. The complete project codes are used as input to ensure the models have all the necessary context to generate unit tests for the entire project. An example is shown in Figure 2. We carefully design prompts for different LLMs to thoroughly test their actual capabilities. In addition, to address specific issues associated with different programming languages, we incorporate language-specific prompts tailored to solve particular challenges. We require the LLMs to generate unit tests for each file within the project. Furthermore, we provide detailed prompts instructing LLMs to focus on various evaluation aspects, including compilation rate, correctness rate, and coverage rate. This structured prompt engineering enhances the effectiveness and relevance of the outputs produced by the LLMs. An example of our designed prompt for Python is shown in Figure 3. All prompts used in our experiments are listed in Appendix B.1, and an ablation analysis of the prompts is shown in Appendix D.1. The vanilla unit tests are extracted from the LLM response based on the input project and prompt.

Scenario 2: Manual Fixing compilation and cascade errors. Manually fixing compilation and cascade errors is motivated by our empirical observation from scenario 1 that even the vanilla unit tests from state-of-the-art LLMs, such as Claude-3.5-Sonnet, contain significant compilation errors, making them non-compilable. Additionally, they exhibit cascade errors that are easy to fix but can affect multiple unit tests or the entire test suite (details in Section 5.5). Although these errors are preliminary and relatively simple to resolve, they hinder further analysis of other aspects of LLM performance on unit test generation, such as correctness and coverage.

Therefore, based on vanilla unit tests, we make the minimum necessary changes to resolve compi-

Vanilla Prompt for Python

System Prompt: You are a coding assistant. You generate only source code.
User Prompt: *{Original Codes}* Please generate enough unit test cases for each Python file in the project. Ensure that the import path is correct, depending on whether the project is structured as a package. Make sure the tests can successfully compile. Make sure the tests have correct results. Try to achieve the highest coverage rate.

Figure 3: The prompt used to generate unit tests for Python projects. Purple indicates language-specific instruction. Blue, orange, and red indicates instructions related to compilation rate, correctness rate, and coverage rate, respectively.

```
# stock_test.py
import unittest
from stock.stock import Stock
from validate import PositiveInteger, ...
class TestStock(unittest.TestCase):
    def setUp(self):
        self.stock = ...
        ...
    ...
if __name__ == '__main__':
    unittest.main()

Error Message:
    from stock.stock import Stock
E   ModuleNotFoundError: No module named 'stock.stock';
    'stock' is not a package
```

Figure 4: An example of compilation error generated by GPT-4-Turbo.

```
import unittest
from base import f_entropy, ...
from tree import Tree
class TestBaseFunctions(unittest.TestCase):
    def test_f_entropy(self):
        p = np.array([1, 2, 2, 3, 3, 3])
        ...
    def test_information_gain(self):
        y = np.array([1, 2, 2, 3, 3, 3])
        ...
    def test_mse_criterion(self):
        y = np.array([1, 2, 2, 3, 3, 3])
        ...
    ...
if __name__ == '__main__':
    unittest.main()

Error Message Obtained by Pytest:
FAILED tree_test.py::TestBaseFunctions::test_f_entropy -
    NameError: name 'np' is not defined
FAILED tree_test.py::TestBaseFunctions::test_get_split_mask -
    NameError: name 'np' is not defined
FAILED tree_test.py::TestBaseFunctions::test_information_gain -
    NameError: name 'np' is not defined
......
```

Figure 5: An example of cascade error generated by CodeQwen1.5-7B-Chat.

lation errors and cascade errors, focusing solely on eliminating these errors without altering the original test intent. Compilation errors are defined as errors that prevent testing frameworks from executing.[2] As shown in Figure 4, the ModuleNotFoundError causes pytest to fail before collecting any unit tests, making the entire test suite uncompilable. This results not only in compilation failure but also in unreachable correctness and coverage rates.[3] Cascade errors are defined as errors that cause cascading failures across multiple unit tests or even the entire test suite. As shown in Figure 5, although the tests are fundamentally correct, this NameError (missing NumPy) invalidates multiple or even the entire test suite. By resolving these errors, manual fixing ensures that all unit tests are compilable and no cascade errors invalidate tests that are fundamentally correct.

Manually fixing compilation and cascade errors plays a crucial role in evaluating the quality and reliability of generated unit tests. By addressing these errors, we gain deeper insights into the effectiveness of LLM-generated unit tests and identify areas for improvement. This process also helps assess the potential for LLMs to improve continuously once such simple errors are resolved. Additionally, we evaluate unit tests fixing only compilation errors in Appendix D.2.

[2]Technically, Python does not require compilation. We refer to errors that cause pytest to fail before it can collect and run any tests as compilation errors.

[3]We consider unreachable correctness and coverage rate as zero.

Self-fixing Prompt for Python

System Prompt: You are a coding assistant...
User Prompt: *{Original Codes}* Please generate enough unit test cases...
LLM Response: *{Generated Vanilla Unit Tests}*
User Prompt: Here are the error messages from the tests: *{Error Messages}*. Errors exist in the generated unit tests. Please fix the unit tests to address these errors and provide the entire unit tests.

Figure 6: The prompt used for the LLM self-fixing scenario for Python projects.

Scenario 3: LLM Self-fixing. Inspired by our observation from manual fixing that different LLMs exhibit significantly different potentials after manual fixing, we seek to investigate how LLMs perform in self-fixing on our benchmark. We explore LLMs' self-fixing abilities by incorporating conversation history and error messages as shown in Figure 6. We provide LLMs with the conversation history (including the system prompt, the user prompt for unit test generation requests, and LLM vanilla response), error messages obtained from the testing framework, and the user prompt for error fixing requests. When the open-source LLM's input length is limited, we prioritize the information in the following order: system prompt, LLM's initial response, error messages, error-fixing requests, and unit test generation requests. Less important information is truncated as needed. Additionally, we reserve at least 2,000 tokens for the open-source LLM's self-fixing outputs. LLM self-fixing scenario helps us understand LLMs' error-fixing ability and their potential to generate better unit tests when incorporating the self-fixing process. Note that during self-fixing, we do not constrain the target error types to just compilation or cascade errors.

4 Experimental Settings

4.1 Models

We evaluate five close-sourced models: GPT-o1, Gemini-1.5-Pro (Team et al., 2024b), Claude-3.5-Sonnet-20241022 (Claude-3.5-Sonnet) (Anthropic, 2024), GPT-4-Turbo (Achiam et al., 2023) and GPT-3.5-Turbo, and four open-sourced models: CodeQwen1.5-7B-Chat (CodeQwen1.5) (Bai et al., 2023), DeepSeek-Coder-6.7b-Instruct (DeepSeek-Coder) (Guo et al., 2024; Zhu

Table 2: Main Results. CR represents Correctness Rate; ComR represents Compilation Rate; LC represents Line Coverage; BC represents Branch Coverage.

Language	Model	CR	ComR	LC	BC	#Tests	#Correct
Python	GPT-4-Turbo	47%	65%	40%	36%	12.60	6.15
	GPT-3.5-Turbo	37%	60%	38%	34%	16.90	6.65
	GPT-o1	60%	65%	**56%**	**54%**	36.35	21.7
	Gemini-1.5-Pro	46%	65%	42%	39%	34.95	16.95
	Claude-3.5-Sonnet	**64%**	**70%**	51%	47%	18.05	10.40
	CodeQwen1.5	24%	65%	43%	40%	25.40	6.80
	DeepSeek-Coder	37%	**70%**	39%	35%	7.20	2.95
	CodeLlama	16%	60%	41%	37%	19.30	3.95
	CodeGemma	13%	50%	31%	28%	15.00	2.30
Java	GPT-4-Turbo	21%	35%	15%	12%	7.05	2.20
	GPT-3.5-Turbo	13%	25%	8%	7%	7.50	0.80
	GPT-o1	41%	60%	44%	**35%**	15.70	6.85
	Gemini-1.5-Pro	19%	30%	14%	12%	23.30	3.90
	Claude-3.5-Sonnet	**53%**	**75%**	**47%**	33%	12.35	7.30
	CodeQwen1.5	0%	0%	0%	0%	12.95	0.00
	DeepSeek-Coder	8%	20%	5%	5%	7.00	0.60
	CodeLlama	0%	0%	0%	0%	7.85	0.00
	CodeGemma	0%	0%	0%	0%	10.50	0.00
JavaScript	GPT-4-Turbo	67%	75%	56%	46%	16.30	11.10
	GPT-3.5-Turbo	51%	65%	37%	28%	13.25	8.05
	GPT-o1	**87%**	**95%**	**87%**	**75%**	39.40	33.30
	Gemini-1.5-Pro	59%	70%	64%	61%	45.85	22.55
	Claude-3.5-Sonnet	65%	80%	59%	53%	20.25	13.35
	CodeQwen1.5	23%	35%	25%	20%	8.45	4.80
	DeepSeek-Coder	62%	85%	50%	35%	11.85	7.90
	CodeLlama	26%	85%	20%	14%	48.75	18.00
	CodeGemma	29%	55%	28%	21%	9.00	3.00

et al., 2024), CodeLlama-7b-Instruct-hf (CodeLlama) (Roziere et al., 2023), and CodeGemma-7b-it (CodeGemma) (Team et al., 2024a). Detailed information is in Appendix B.2.

4.2 Implementation Details

We use zero-shot prompting for unit test generation. The temperature is set to 0 during inference. Experiments are conducted on 8 NVIDIA A100 GPUs. The maximum input length is configured to match the token limit of each LLM to evaluate model capabilities. We use Pytest[4] for Python, Jacoco[5] for Java, and JEST[6] for JavaScript regarding testing frameworks.

5 Experiments

We evaluate the generated unit tests from three scenarios, vanilla (§ 5.1), after manual fixing of compilation and cascade errors (§ 5.2), and LLM self-fixing (§ 5.3). For each scenario, we evaluate the Correctness Rate (CR), Compilation Rate (ComR), Line Coverage (LC), and Branch Coverage (BC). We also conduct unique contribution analyses (§5.4) and detailed error analyses (§ 5.5).

5.1 Main Results

The main results of the LLMs' unit test generation performance focus on the vanilla unit tests extracted directly from the LLMs' outputs without any changes. The goal of this scenario is to assess the LLMs' current raw capability to generate project-level unit tests.

Table 2 shows the evaluation results of the vanilla unit tests. First, we observe that different LLMs have varying language-level expertise. For example, Claude-3.5-Sonnet performs the best in Java but falls behind GPT-o1 in JavaScript. Second, we can see from the results that LLMs have different metric-level expertise as well, validating the effectiveness of different evaluation metrics. For example, in Python, Claude-3.5-Sonnet performs the best on CR and ComR while falling behind GPT-o1 on LC and BC.

Among three programming languages, Java is the most difficult language, primarily due to stricter syntax. Many models fail to generate valid Java code, leading to low compilation rates and execution coverage. Among all the evaluated models, GPT-o1 performs the best in general, especially in JavaScript. CodeLlama and CodeGemma have the worst general performance. We also observe that some models tend to generate more unit tests. However, generating more unit tests does not necessarily lead to better coverage rates. For example, Gemini-1.5-Pro tends to generate the most unit tests but does not obtain the best coverage rate. Additionally, we observe that sometimes the open-source model can even outperform some closed-source models. For example, DeepSeek-Coder works better than GPT-3.5-Turbo on Python and JavaScript. Finally, we confirmed from such results that dependencies exist in metrics. On Java, models like CodeQwen1.5, CodeLlama, and CodeGemma fail to generate compilable unit tests, resulting in the lowest correctness rates and coverage rates.

5.2 Manual Fixing Results

Table 3 shows the evaluation results with improvements compared to vanilla results after manual fixing. After fixing compilation and cascade errors, the results show significant improvements across all programming languages and LLMs compared to vanilla unit tests. This indicates that the unit tests generated by LLMs are highly sensitive to compilation and cascade errors.

Among all programming languages, Java benefits the most from manual fixing. In the case of vanilla unit tests, Java exhibits the lowest compilation rates, making it particularly challenging. However, after manual fixing, Java shows the most substantial improvement, highlighting the poten-

[4]https://docs.pytest.org/en/stable/
[5]https://www.eclemma.org/jacoco/
[6]https://jestjs.io/

Table 3: Manual Fixing Results with Improvements. CR represents Correctness Rate; ComR represents Compilation Rate; LC represents Line Coverage; BC represents Branch Coverage. The improvements are shown in parentheses.

Language	Model	CR	ComR	LC	BC	#Tests	#Correct
Python	GPT-4-Turbo	74% (+27%)	100%	65% (+25%)	59% (+23%)	12.60	9.30
	GPT-3.5-Turbo	64% (+27%)	100%	63% (+25%)	57% (+23%)	16.90	10.50
	GPT-o1	89% (+29%)	100%	**88%** (+32%)	**86%** (+32%)	36.35	32.25
	Gemini-1.5-Pro	61% (+15%)	100%	71% (+29%)	68% (+29%)	34.95	22.10
	Claude-3.5-Sonnet	**92%** (+28%)	100%	74% (+23%)	70% (+23%)	18.05	16.40
	CodeQwen1.5	46% (+22%)	100%	70% (+27%)	65% (+25%)	25.40	10.90
	DeepSeek-Coder	53% (+16%)	100%	60% (+21%)	54% (+19%)	7.20	4.10
	CodeLlama	31% (+15%)	100%	61% (+20%)	56% (+19%)	19.30	7.20
	CodeGemma	36% (+23%)	100%	54% (+23%)	49% (+21%)	15.00	7.85
Java	GPT-4-Turbo	59% (+38%)	100%	40% (+25%)	32% (+20%)	7.05	5.05
	GPT-3.5-Turbo	54% (+41%)	100%	36% (+28%)	27% (+20%)	7.50	4.55
	GPT-o1	64% (+23%)	100%	**65%** (+21%)	**56%** (+21%)	15.7	10.75
	Gemini-1.5-Pro	56% (+37%)	100%	54% (+40%)	53% (+41%)	23.30	15.25
	Claude-3.5-Sonnet	**74%** (+21%)	100%	60% (+13%)	53% (+20%)	12.35	9.65
	CodeQwen1.5	60% (+60%)	100%	42% (+42%)	31% (+31%)	12.95	8.40
	DeepSeek-Coder	52% (+44%)	100%	33% (+28%)	19% (+14%)	7.00	3.80
	CodeLlama	36% (+36%)	100%	25% (+25%)	20% (+20%)	7.85	4.95
	CodeGemma	57% (+57%)	100%	37% (+37%)	22% (+22%)	10.50	6.50
JavaScript	GPT-4-Turbo	89% (+22%)	100%	75% (+19%)	59% (+13%)	16.30	14.20
	GPT-3.5-Turbo	74% (+23%)	100%	58% (+21%)	45% (+17%)	13.25	11.20
	GPT-o1	**91%** (+4%)	100%	**92%** (+5%)	79% (+4%)	39.40	35.15
	Gemini-1.5-Pro	76% (+17%)	100%	88% (+24%)	**80%** (+19%)	45.85	33.45
	Claude-3.5-Sonnet	87% (+22%)	100%	77% (+18%)	68% (+15%)	20.25	17.55
	CodeQwen1.5	32% (+9%)	100%	35% (+10%)	27% (+7%)	8.45	6.15
	DeepSeek-Coder	67% (+5%)	100%	58% (+8%)	43% (+8%)	11.85	8.10
	CodeLlama	62% (+36%)	100%	44% (+24%)	28% (+14%)	48.75	31.50
	CodeGemma	58% (+29%)	100%	50% (+22%)	38% (+17%)	9.00	6.40

Table 4: Evaluation Results after Self-fixing. CR represents Correctness Rate; ComR represents Compilation Rate; LC represents Line Coverage; BC represents Branch Coverage. The comparisons with manual fixing are shown in parentheses.

Language	Model	CR	ComR	LC	BC	#Tests	#Correct
Python	GPT-4-Turbo	52% (-22%)	70% (-30%)	39% (-26%)	35% (-24%)	8.85	4.55
	GPT-3.5-Turbo	52% (-12%)	75% (-25%)	45% (-18%)	39% (-18%)	14.15	8.20
	GPT-o1	67% (-22%)	70% (-30%)	60% (-28%)	58% (-28%)	35.50	24.35
	Gemini-1.5-Pro	47% (-14%)	60% (-40%)	45% (-26%)	42% (-26%)	34.95	17.40
	Claude-3.5-Sonnet	**86%** (-6%)	**90%** (-10%)	**67%** (-7%)	**63%** (-7%)	18.00	15.55
	CodeQwen1.5	22% (-24%)	60% (-40%)	41% (-29%)	37% (-28%)	25.15	6.25
	DeepSeek-Coder	18% (-35%)	35% (-65%)	20% (-40%)	18% (-36%)	4.30	1.45
	CodeLlama	0% (-31%)	5% (-95%)	5% (-56%)	5% (-51%)	3.90	0.00
	CodeGemma	8% (-28%)	25% (-75%)	14% (-40%)	13% (-36%)	9.15	0.70
Java	GPT-4-Turbo	43% (-16%)	55% (-45%)	26% (-14%)	18% (-14%)	6.40	2.80
	GPT-3.5-Turbo	17% (-37%)	25% (-75%)	11% (-25%)	12% (-15%)	6.90	1.05
	GPT-o1	**68%** (+4%)	**85%** (-15%)	**58%** (-7%)	**54%** (-2%)	15.60	10.10
	Gemini-1.5-Pro	31% (-25%)	40% (-60%)	29% (-25%)	24% (-29%)	22.65	7.15
	Claude-3.5-Sonnet	55% (-19%)	70% (-30%)	39% (-21%)	31% (-22%)	10.95	6.70
	CodeQwen1.5	5% (-55%)	5% (-95%)	0% (-42%)	0% (-31%)	12.60	0.05
	DeepSeek-Coder	13% (-39%)	20% (-80%)	5% (-28%)	2% (-17%)	1.35	0.25
	CodeLlama	0% (-36%)	0% (-100%)	0% (-25%)	0% (-20%)	1.30	0.00
	CodeGemma	2% (-55%)	5% (-95%)	3% (-34%)	0% (-22%)	1.75	0.05
JavaScript	GPT-4-Turbo	70% (-19%)	85% (-15%)	48% (-27%)	35% (-24%)	8.35	6.35
	GPT-3.5-Turbo	64% (-10%)	75% (-25%)	40% (-18%)	30% (-15%)	9.70	5.00
	GPT-o1	54% (-37%)	65% (-35%)	47% (-45%)	38% (-41%)	20.30	12.25
	Gemini-1.5-Pro	**75%** (-1%)	85% (-15%)	**71%** (-17%)	**65%** (-15%)	40.95	28.65
	Claude-3.5-Sonnet	74% (-13%)	80% (-20%)	60% (-17%)	53% (-15%)	18.05	13.35
	CodeQwen1.5	55% (+23%)	**95%** (-5%)	66% (+31%)	52% (+25%)	26.10	15.50
	DeepSeek-Coder	14% (-53%)	35% (-65%)	15% (-43%)	10% (-33%)	2.90	1.00
	CodeLlama	9% (-53%)	35% (-65%)	7% (-37%)	5% (-23%)	7.15	0.55
	CodeGemma	31% (-27%)	60% (-40%)	29% (-21%)	21% (-17%)	10.85	3.05

tial of LLMs for Java after fixing compilation and cascade errors. Among all models, GPT-o1 still performs the best after manual fixing, and CodeLlama and CodeGemma still exhibit the worst general performance. Gemini-1.5-Pro shows the best coverage improvement overall, indicating its strong potential for better unit test generation once compilation and cascade errors are corrected. Finally, we observe that certain models that initially underperform can match or surpass stronger models after manual fixing. For example, in Java, CodeQwen1.5 outperforms DeepSeek-Coder and is now on par with GPT-4-Turbo. In Python, Gemini-1.5-Pro surpasses CodeQwen1.5-7B-Chat, showing better potential after manual fixing. On JavaScript, GPT-3.5-Turbo has reached parity with DeepSeek-Coder.

5.3 LLMs Self-fixing Results

During LLM self-fixing, conversation history and error messages are provided to help the model correct errors. This scenario assesses the LLM's ability to fix its own mistakes and its potential to generate better unit tests by incorporating self-fixing.

Table 4 shows the LLM self-fixing evaluation results in comparison with manual fixing. First, we observe that most closed-source models have the ability to self-fix errors and generate better unit tests compared to vanilla results, while the evaluated open-source models lack reliable self-fixing abilities. This limitation may stem from factors such as restricted input length, which leads to incomplete context, as well as weaker comprehension and instruction-following abilities. For instance, open-source models like CodeGemma and CodeLlama tend to generate textual instructions for fixing errors rather than directly producing the corrected unit tests specified in the prompt.

Second, we observe that LLM self-fixing follows similar but not identical trends to manual fixing, suggesting that while LLMs' potential for improvement generally aligns with self-fixing capabilities, some LLMs do not follow this pattern. For instance, in JavaScript, GPT-o1's self-fixing performance on the coverage rate is significantly worse compared to manual fixing due to a lower number of generated unit tests and a reduced compilation rate.

Although LLM self-fixing currently lags behind manual fixing in performance, LLM self-fixing still holds significant potential. Self-fixing has proven effective when LLMs have the necessary capabilities, and it even has the potential to surpass manual fixing due to its flexibility. For example, in JavaScript, CodeQwen1.5 shows greater improvement through self-fixing compared to manual fixing. This is because, in its vanilla output, CodeQwen1.5 sometimes fails to understand the prompt and does not generate any unit tests. Manual fixing based solely on these initial outputs cannot resolve this issue. However, LLM self-fixing can overcome this limitation by correctly interpreting unit test generation prompts when error messages indicate that unit tests are required.

5.4 Unique Contribution of Unit Tests

We also explore the unique contribution of the generated unit tests on Python. The unique contribution is defined as the total portion of coverage

Table 5: Unique Contribution on Vanilla Unit Tests.

Model	#Tests	LC	BC	Unique Contribution
GPT-4-Turbo	12.60	40%	36%	6.35%
GPT-3.5-Turbo	16.90	38%	34%	5.90%
GPT-o1	36.35	56%	54%	6.75%
Gemini-1.5-Pro	34.95	42%	39%	6.05%
Claude-3.5-Sonnet	18.05	51%	47%	**11.40%**
CodeQwen1.5	25.40	43%	40%	3.75%
DeepSeek-Coder	7.20	39%	35%	<u>8.90%</u>
CodeLlama	19.30	41%	37%	5.55%
CodeGemma	15.00	31%	28%	2.70%

contributed by each generated unit test that does not overlap with the coverage of other unit tests. This is important for several reasons. First, some LLMs generate more unit tests than others, making it insufficient to rely solely on coverage rate as a metric; the unique contribution of each test should also be considered. Second, it is crucial for LLMs to generate fewer unit tests while still achieving a high coverage rate, as running a large number of tests can sometimes be resource- or time-intensive.

Table 5 reveals that all the tested LLMs have low rates of unique contributions, indicating a tendency to produce redundant and repetitive unit tests. Although GPT-o1 has better coverage rates than Claude-3.5-Sonnet, GPT-o1 produces significantly more unit tests, and its unique contribution is lower than Claude-3.5-Sonnet's, indicating that it relies on quantity rather than quality to reach higher coverage. As a result, this approach may compromise the overall efficiency of the testing process.

5.5 Error Analyses

We conduct complex analyses of compilation, cascade, and post-fix errors per programming language, highlighting the common errors and potential reasons behind the errors. The full analyses are presented in Appendix E.

Compilation Error Analyses. In ***Python***, common compilation errors arise from incorrect import paths for project functions or classes, hallucinated import names or paths, and mismatched parentheses. ***Java***, a syntax-heavy programming language compared to Python and JavaScript, encounters various compilation errors, like hallucinated methods, constructors, or classes, missing essential elements like package declarations, illegal access to private or protected elements, invalid code generation, and improper use of mocking frameworks like Mockito, along with argument type mismatches, ambiguous references, and incompatible types. In ***JavaScript***, common errors include hallucinated imports with incorrect paths, empty test suites, and syntax errors from incomplete code generation or mismatched parentheses.

Cascade Error Analyses. For ***Python***, cascade errors include missing imports (e.g., numpy, unittest, project functions/classes) and FileNotFoundError due to unmocked external files. For ***Java***, the most common cascade error is improper or missing mocking of user interactions, leading to unusable coverage reports when tests terminate abruptly. For ***JavaScript***, the cascade errors include missing imports (e.g., chai, three, project functions/classes), confusion between named and default imports, and Jest framework compliance issues.

Post-Fix Error Analyses. For all programming languages, the mismatch between expected and actual values is the most common error. In ***Python***, AttributeError often occurs due to LLMs hallucinating non-existent attributes. In ***Java***, frequent errors include NullPointerException, zero interactions with mocks, and failures to release mocks due to improper usage. Another frequent error in ***JavaScript*** is TypeError, typically caused by LLMs hallucinating non-existent functions and constructors or LLMs invalidly mocking some variables.

Overall. Common errors across different programming languages include hallucinations of functions or classes, and missing required functions or classes. Missing required functions or classes often occurs because LLMs *prioritize logical structure over boilerplate code* and *fail to understand the codebase structure and the dependencies between functions, classes, or modules*. Failure to understand the codebase structure and dependencies can also cause other mistakes, such as confusing non-package and package-based projects (Python) or incorrectly using functions, classes, or packages (Java). The most common post-fix error is the mismatch between expected and received values, often caused by incorrect expected values due to the *weak reasoning abilities* of LLMs.

6 Conclusion

In conclusion, we build a reliable and high-quality project-level unit test generation benchmark – ProjectTest – with three programming languages. We comprehensively evaluate nine LLMs' unit test generation abilities with/without manual fixing and LLM self-fixing mechanism on ProjectTest. Besides, we conduct comprehensive error analyses per programming language.

Limitations

Our study has several limitations. First, due to our capacity, we mainly focus on three programming languages—Python, Java, and JavaScript—missing the chance to include other languages like C and C#. Additionally, given the fact that the input length restrictions of current LLMs make them unsuitable for handling larger projects in their entirety, we selected moderate-sized projects, allowing us to explore issues like the robustness of LLMs in unit test generation (e.g., hallucinations or incorrect assertions) rather than focusing solely on their ability to handle long-context inputs.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Saranya Alagarsamy, Chakkrit Tantithamthavorn, Chetan Arora, and Aldeida Aleti. 2024. Enhancing large language models for text-to-testcase generation. *arXiv preprint arXiv:2402.11910*.

M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE.

AI Anthropic. 2024. Claude 3.5 sonnet model card addendum. *Claude-3.5 Model Card*, 3:6.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

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

Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv e-prints*, pages arXiv–2308.

Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327.

Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*, pages 348–351.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv e-prints*, pages arXiv–2401.

Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2024a. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752*.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. 2024b. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. 2024. Devbench: A comprehensive benchmark for software development. *arXiv preprint arXiv:2403.08604*.

Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 14–26. IEEE.

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172.

Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE.

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

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–322.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. 2024a. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024b. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024. Testeval: Benchmarking large language models for test case generation. *arXiv preprint arXiv:2406.04531*.

Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 246–256. IEEE.

Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Table 6: Dataset Details (Python).

Project Name	License	Link	#Stars	#Forks
blackjack	MIT license	blackjack	2937	641
bridge	MIT license	bridge	2937	641
doudizhu	MIT license	doudizhu	2937	641
fuzzywuzzy	MIT license	fuzzywuzzy	9200	876
gin_rummy	GPL-2.0 license	gin_rummy	2937	641
keras_preprocessing	MIT license	keras_preprocessing	1024	443
leducholde	MIT license	leducholde	2937	641
limitholdem	MIT license	limitholdem	2937	641
mahjong	MIT license	mahjong	2937	641
nolimitholdem	MIT license	nolimitholdem	2937	641
slugify	MIT license	slugify	1500	109
stock	CC-BY-SA-4.0 license	stock	10700	1800
stock2	CC-BY-SA-4.0 license	stock2	10700	1800
stock3	CC-BY-SA-4.0 license	stock3	10700	1800
stock4	CC-BY-SA-4.0 license	stock4	10700	1800
structly	CC-BY-SA-4.0 license	structly	10700	1800
svm	MIT license	svm	10800	1800
the fuzz	CC-BY-SA-4.0 license	the fuzz	2949	141
tree	CC-BY-SA-4.0 license	tree	10800	1800
uno	MIT license	uno	2937	641

Table 7: Dataset Details (Java).

Project Name	License	Link	#Stars	#Forks
Actor_relationship_game	Apache-2.0 license	Actor_relationship_game	85	5
banking application	MIT license	banking application	341	366
CalculatorOOPS	MIT license	CalculatorOOPS	525	513
emailgenerator	MIT license	emailgenerator	525	513
heap	MIT license	heap	60500	19600
idcenter	Apache-2.0 license	idcenter	146	136
libraryApp	MIT license	libraryApp	341	366
libraryManagement	MIT license	libraryManagement	341	366
logrequestresponseundertow	Author Permission	logrequestresponseundertow	152	131
Password_Generator	MIT license	Password_Generator	341	366
Pong Game	MIT license	Pong Game	341	366
redis	Apache-2.0 license	redis	413	218
servlet	MIT license	servlet	341	366
simpleChat	MIT license	simpleChat	543	1500
springdatamongowithcluster	Author Permission	springdatamongowithcluster	152	131
springmicrometerundertow	Author Permission	springmicrometerundertow	152	131
springreactivenonreactive	Author Permission	springreactivenonreactive	152	131
springuploads3	Author Permission	springuploads3	152	131
Train	MIT license	Train	545	1600

A Dataset

We provide the detailed information of our datasets in Table 6, Table 7, and Table 8. We provide programming language, project name, license, link, number of stars, and number of forks for each individual project.

The license of "Author Permission" in Table 7 means that we obtain the usage permission from the author of the corresponding repository[7].

B More Implementation Details

B.1 Prompts

The prompts are displayed in Figure 7, 8, and 9.

B.2 Models

The detailed information of models, including license and link, is provided in Table 9.

C More Statistics

Table 10 presents the percentages of the vanilla-generated unit tests containing comparisons between expected and actual values per language and per model.

[7] https://github.com/frandorado/spring-projects/tree/master

Table 8: Dataset Details (JavaScript).

Project Name	License	Link	#Stars	#Forks
aggregate	MIT license	aggregate	1500	18
animation	MIT license	animation	103000	35400
check	MIT license	check	1500	18
circle	MIT license	circle	2700	330
ckmeans	ISC license	ckmeans	3400	226
controls	MIT license	controls	103000	35400
convex	MIT license	convex	2700	330
easing	MIT license	easing	418	9
magnetic	MIT license	magnetic	418	9
overlapkeeper	MIT license	overlapkeeper	2700	330
particle	MIT license	particle	2700	330
pixelrender	MIT license	pixelrender	2400	274
plane	MIT license	plane	2700	330
solver	MIT license	solver	2700	330
span	MIT license	span	2400	274
spherical	MIT license	spherical	103000	35400
synergy	MIT license	synergy	310	3
t_test	ISC license	t_test	3400	226
validate	MIT license	validate	1500	18
zone	MIT license	zone	2400	274

Vanilla Prompt for Java

System Prompt: You are a coding assistant. You generate only source code.
User Prompt: *{Original Codes}* Please generate enough unit test cases for each java file in the {method_signature} project. Ensure to use mock properly for unit tests. Make sure the tests can successfully compile. Make sure the tests have correct results. Try to achieve the highest coverage rate.

Figure 7: The prompt used to generate unit tests for Java projects.

D Ablation Study

D.1 Ablation Study on Prompts

We perform a detailed ablation study to analyze the impact of prompts on the performance of unit test generation by LLMs. As mentioned in § 3.3, the prompt is composed of programming language-specific requirements (PL), as well as requirements related to the correctness rate (CR), the compilation rate (ComR), and the coverage rate metrics (Coverage). We ablate each component and analyze the performance of unit test generation of GPT-4-Turbo using different prompts as shown in Table 11. Requirements related to CR and ComR can help improve performance in vanilla unit tests. Coverage-related requirements are not always beneficial, possibly because a high coverage rate is too abstract for LLMs to interpret effectively. Programming language-specific requirements improve performance in CR but have the opposite effect on ComR, LC, and BC.

Besides, we follow the prompt template from previous work like Siddiq et al. (2024) to move the prompts into comments (e.g., /*...*/). We compare the performance with and without comment signs in Table 11. Experimental results show that our prompt demonstrates a significant advantage in

Table 9: Model Details.

Model Type	Model Name	License	Link
Close-sourced	GPT-4-Turbo	-	https://platform.openai.com/docs/models/gpt-4#gpt-4-turbo-and-gpt-4
Close-sourced	GPT-3.5-Turbo	-	https://platform.openai.com/docs/models/gpt-4#gpt-3-5-turbo
Close-sourced	GPT-o1	-	https://platform.openai.com/docs/models#o1
Close-sourced	Gemini-1.5-Pro	-	https://ai.google.dev/gemini-api/docs/models/gemini#gemini-1.5-pro
Close-sourced	Claude-3.5-Sonnet	-	https://www.anthropic.com/claude/sonnet
Open-sourced	CodeQwen1.5-7B-Chat	Tongyi Qianwen LICENSE AGREEMENT	https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat
Open-sourced	DeepSeek-Coder-6.7b-Instruct	DEEPSEEK LICENSE AGREEMENT	https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct
Open-sourced	CodeLlama-7b-Instruct-hf	LLAMA 2 COMMUNITY LICENSE AGREEMENT	https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf
Open-sourced	CodeGemma-7b-it	Gemma Terms of Use	https://huggingface.co/google/codegemma-7b-it

Table 10: Percentages of the vanilla unit tests containing expected and actual value comparisons.

Model	GPT-4-Turbo	GPT-3.5-Turbo	GPT-o1	Gemini	Claude	CodeQwen	DeepSeek-Coder	CodeLlama	CodeGemma
Python	98%	99%	98%	89%	99%	97%	96%	99%	88%
Java	97%	90%	98%	98%	97%	89%	94%	85%	93%
JavaScript	100%	89%	96%	100%	100%	100%	96%	86%	100%

Vanilla Prompt for JavaScript

System Prompt: You are a coding assistant. You generate only source code.
User Prompt: *{Original Codes}* Please generate enough unit test cases for every javascript file in {method_signature} project. Make sure the tests can successfully compile. Make sure the tests have correct results. Try to achieve the highest coverage rate.

Figure 8: The prompt used to generate unit tests for JavaScript projects.

Prompt for Python with Comment Sign

System Prompt: You are a coding assistant. You generate only source code.
User Prompt: *{Original Codes}* # classname_test.py\n # Test class of {classname}.\n # Please generate enough unit test cases for each python file in the {method_signature} project. Ensure that the import path is correct, depending on whether the project is structured as a package. Make sure the tests can successfully compile. Make sure the tests have correct results. Try to achieve the highest coverage rate. \n # class {classname_test}\n

Figure 9: The prompt used to generate unit tests for Python projects.

CR, while the prompt with comment signs exhibits marginal advantages in ComR, LC, and BC.

D.2 Effect of Compilation Errors and Cascade Errors

We manually fix only compilation errors and evaluate the corrected unit tests in Table 12.

By fixing compilation errors, Table 12 shows significant improvements across all programming languages and LLMs compared to Table 2, indicating that all the programming languages and LLMs are highly sensitive to compilation errors. Comparing Table 12 with Table 3, we can observe that CodeQwen1.5, CodeGemma, and CodeLlama are more sensitive to cascade errors. For Java, the changes in Table 3 compared to Table 12 are primarily due to missing or invalid mocks of user interactions[8] which occur more frequently in unit tests generated

Table 11: Ablation Study. The Performance of Unit Test Generation by GPT-4-Turbo Using Different Prompts.

Phase	Settings	CR	ComR	LC	BC	#Tests	#Correct Tests
Vanilla	Full Prompt	47%	65%	40%	36%	12.60	6.15
	w/o CR	33% ↓	65%	42%	38%	12.75	4.75
	w/o ComR	35%	63% ↓	41%	38%	11.20	3.95
	w/o Coverage	43%	75%	46% ↑	42% ↑	9.80	4.20
	w/o PL	47%	75%	53%	49%	9.95	4.35
	w/ Comments	41%	65%	45%	41%	10.65	4.15
Manual Fixing	Full Prompt	74%	100%	65%	59%	12.60	9.30
	w/o CR	76% ↑	100%	69%	64%	12.75	9.90
	w/o ComR	75%	100%	70%	65%	11.20	8.35
	w/o Coverage	68%	100%	66% ↑	61% ↑	9.80	6.75
	w/o PL	70%	100%	70%	66%	9.95	6.90
	w/ Comments	66%	100%	68%	62%	10.65	7.00

by CodeQwen1.5 and CodeGemma.

E Detailed Error Analyses

We conduct complex analyses of compilation, cascade, and post-fix errors, highlighting the common errors and potential reasons behind the errors.

Compilation Error Analyses Figure 10 highlights the detailed compilation errors that occurred. One of the most common compilation errors in ***Python*** arises from the LLM's inability to determine whether the project being tested is a package. Specifically, LLMs struggle to recognize the presence or absence of *__init__.py* files, which define a package, leading to confusion between package-based and non-package projects. This inability leads LLM to fail to correctly import functions or classes from the tested project. Other compilation errors include hallucinating the paths or names of imported functions/classes and mismatched parentheses. ***Java***, a syntax-heavy programming language compared to Python and JavaScript, encounters various compilation errors, resulting in a significantly lower compilation rate than other languages. Java compilation errors often arise from issues like hallucinated methods, constructors, or classes, such as incorrect or non-existent imports and references. Missing essential information, such as required

[8] We consider coverage rates as not applicable when requiring user interactions.

Table 12: Evaluation Results When Only Manually Fixing Compilation Errors

Language	Model	CR	ComR	LC	BC	#Tests	#Correct Tests
Python	GPT-4-Turbo	73%	100%	65%	59%	12.60	9.10
	GPT-3.5-Turbo	63%	100%	62%	56%	16.90	10.40
	GPT-o1	89%	100%	**88%**	**85%**	36.35	32.25
	Gemini-1.5-Pro	61%	100%	71%	68%	34.95	22.10
	Claude-3.5-Sonnet	**92%**	100%	74%	70%	18.05	16.40
	CodeQwen1.5	40%	100%	65%	59%	25.40	9.60
	DeepSeek-Coder	53%	100%	60%	54%	7.20	4.10
	CodeLlama	26%	100%	56%	50%	19.30	6.15
	CodeGemma	30%	100%	52%	47%	15.00	6.15
Java	GPT-4-Turbo	59%	100%	42%	34%	7.05	5.05
	GPT-3.5-Turbo	48%	100%	37%	29%	7.50	4.20
	GPT-o1	62%	100%	**67%**	56%	15.70	10.50
	Gemini-1.5-Pro	55%	100%	54%	53%	23.30	15.00
	Claude-3.5-Sonnet	**73%**	100%	63%	**57%**	12.35	9.60
	CodeQwen1.5	49%	100%	49%	39%	12.95	7.50
	DeepSeek-Coder	40%	100%	36%	19%	7.00	2.85
	CodeLlama	30%	100%	26%	21%	7.85	4.25
	CodeGemma	46%	100%	44%	26%	10.50	5.55
JavaScript	GPT-4-Turbo	89%	100%	75%	59%	16.30	14.15
	GPT-3.5-Turbo	71%	100%	56%	44%	13.25	10.65
	GPT-o1	**91%**	100%	**92%**	79%	39.40	35.15
	Gemini-1.5-Pro	76%	100%	88%	**80%**	45.85	33.30
	Claude-3.5-Sonnet	83%	100%	75%	66%	20.25	16.75
	CodeQwen1.5	28%	100%	29%	22%	8.45	5.65
	DeepSeek-Coder	66%	100%	58%	43%	11.85	8.05
	CodeLlama	28%	100%	20%	15%	48.75	21.40
	CodeGemma	45%	100%	43%	30%	9.00	5.75

functions, classes, or packages, and package declarations, is also a common problem. Errors frequently occur due to illegal access to private or protected elements, invalid code generation (e.g., generating text instead of code), and improper use of mocking frameworks like Mockito, including incorrect objects, missing or misused MockMvc injections, and argument mismatches. Other errors include incorrect usage of other functions, classes, or packages—such as argument type errors, ambiguous references, or incompatible types. One of the most common compilation errors in ***JavaScript*** is the hallucination of imported functions or classes, where the issue often lies in incorrect paths for the imported functions or classes. CodeQwen1.5 has a particularly common compilation error involving invalid generation. This typically occurs due to difficulty understanding the prompt, the need for more specific or detailed code requirements, or the assumption that the code is part of a larger project, leading it to decline generating unit tests. Other compilation errors include test suites containing empty unit tests and syntax errors caused by incomplete code generation or mismatched parentheses.

Cascade Error Analyses Figure 11 highlights the detailed cascade errors that occurred. For ***Python***, the cascade errors include missing imports of commonly used packages such as numpy and unittest, missing imports of functions or classes from the tested project, and FileNotFoundError. For ***Java***, the most common cascade error is missing or invalid mocking of user interactions. A proper unit test should simulate user interactions through mocking rather than relying on real user inputs. This issue also results in unusable coverage reports for some tested projects, as the error forces an abrupt termination, preventing the generation of coverage data. For ***JavaScript***, the cascade errors include missing imports of commonly used packages such as chai and three, and missing imports of functions or classes from the tested project. Two other common errors specific to JavaScript are that LLMs may confuse named imports with default imports and fail to comply with the Jest framework.

Post-Fix Error Analyses Figure 12 highlights the incorrectness reasons after all manual fixes. For all programming languages, the mismatch between expected and actual values (AssertionError) is the most common error. Another frequent error in ***Python*** is AttributeError, typically caused by LLMs hallucinating non-existent attributes. Other frequent problems in ***Java*** include NullPointer Errors, zero interactions with mocks, and failures to release mocks, often due to improper mock usage. For projects tested with the Spring framework, errors specific to Spring are also common. Another frequent error in ***JavaScript*** is TypeError, mostly caused by LLMs hallucinating non-existent functions and constructors or LLMs invalidly mocking some variables.

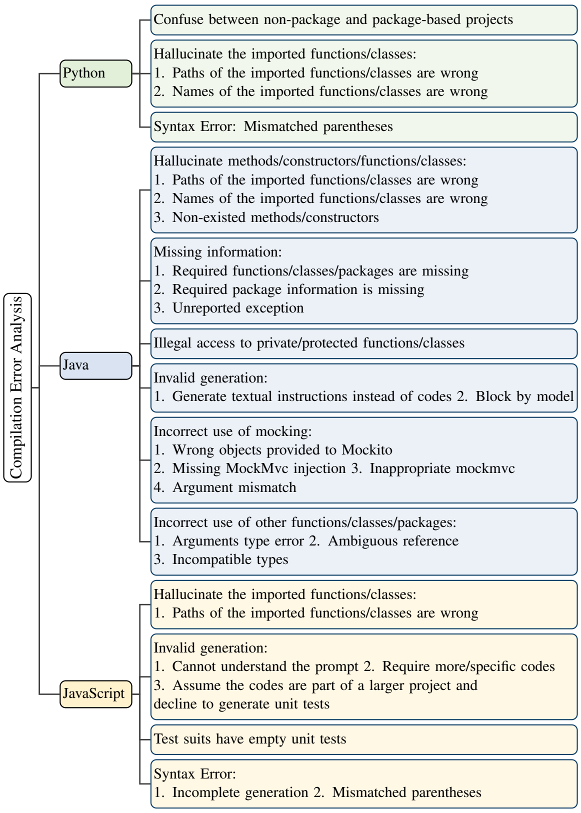

Figure 10: Frequent Compilation Errors in Main Results.

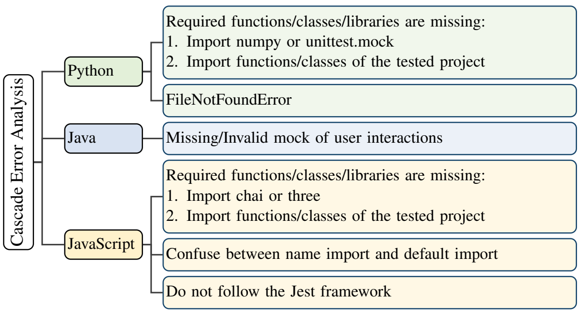

Figure 11: Frequent Cascade Errors.

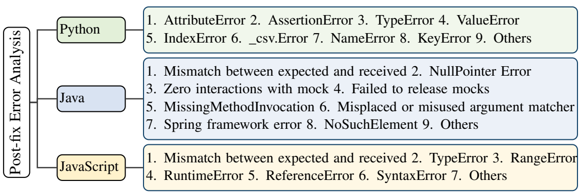

Figure 12: Frequent Post-Fix Errors.