MultiFileTest: A Multi-File-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 Large Language Model (LLM) use case. However, existing evaluation benchmarks for LLM unit test generation focus on function- or class-level code (single-file) rather than more practical and challenging multi-file-level codebases. To address such a limitation, we propose MultiFileTest, a multi-file-level benchmark for unit test generation covering Python, Java, and JavaScript. MultiFileTest features 20 moderate-sized and high-quality projects per language. We evaluate nine frontier LLMs on MultiFileTest, and the results show that all frontier LLMs tested exhibit moderate performance on MultiFileTest on Python and Java, highlighting the difficulty of MultiFileTest. We also conduct a thorough error analysis, which shows that even advanced LLMs, such as Claude-3.5-Sonnet, exhibit basic yet critical errors, including executability 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 code quality. Unit tests verify whether individual components of a software program work as expected—for example, checking if add(2, 3) returns 5. Writing unit tests is time-consuming, usually accounting for approximately 15.8% of software development time (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, these methods often produce less readable tests and are limited to certain types of functions (Grano et al., 2018). Recently, Large Language Models (LLMs) have significantly accelerated unit test generation and improved readability and generalizability with little to no human effort (Siddiq et al., 2024; Xie et al., 2023).

Despite the rapid adoption of LLMs for unit testing, evaluation of LLM unit test generation capabilities appears to be lagging behind. Existing benchmarks primarily focus on function, class, or single-file level code (Chen et al., 2021; Du et al., 2023; Wang et al., 2025; Jain et al., 2024a), while real-world scenarios typically involve multi-file codebases where functions interact across files with complex dependencies. For instance, a function in file A might import and use classes from files B and C, which themselves depend on other modules. To properly test such codebases, LLMs must understand these cross-file dependencies and correctly set up the test environment, making it significantly more complex than testing function, class, or single-file level code. The only benchmark that briefly explores multi-file testing, DevBench (Li et al., 2024), includes too few projects per language (e.g., 5 for Java) with varying quality and lacks thorough analysis of error types, potentials, or self-fixing capabilities of frontier LLMs' multi-file level unit test generation.

Therefore, we propose MultiFileTest, a new multi-file-level unit test generation benchmark that offers a larger, higher-quality project set along with comprehensive error analysis of state-of-the-art LLMs. MultiFileTest covers three programming languages: Python, Java, and JavaScript. For each language, we construct 20 self-contained multi-file projects from GitHub[1] using clear filtering criteria: moderate-sized projects with multiple files and dependencies between them, each under 1,600 lines of code (fitting within input constraints of most code language models), with quality ensured

[1] https://github.com/

through substantial stars and forks. This carefully constructed benchmark enables comprehensive evaluation of LLMs' capabilities in handling realistic multi-file testing scenarios.

Our evaluation of nine frontier LLMs (including Claude-3.5-Sonnet (Anthropic, 2024), Gemini-2.0-Flash (Team et al., 2024b), and GPT-o1) reveals moderate performance across models, highlighting the difficulty of MultiFileTest. We observe that different LLMs exhibit different language-level expertise: Claude-3.5-Sonnet ranks first in Java, while GPT-o1 ranks first in JavaScript. Among three programming languages, Java is the most difficult, primarily due to its stricter syntax. Among all tested models, GPT-o1 performs best overall, especially in JavaScript.

Error analysis reveals that even advanced LLMs, such as Claude-3.5-Sonnet, produce significant executability and cascade errors (defined in §3.3). These errors often stem from misunderstandings of contextual dependencies and program structure—areas where reasoning capabilities of LLMs are critical. 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 manually fix LLM's executability and cascade errors and then re-evaluate the fixed unit tests. This allows us to measure both raw performance and potential improvement when combined with error-fixing mechanisms. By incorporating error-fixing, we uncover critical insights into both the effort required to refine generated tests and the various types of errors that occur in unit tests generated by different LLMs. We observe that model rankings change significantly after manual fixes, revealing substantial differences in error distributions and improvement potential among LLMs. Inspired by these findings from manual fixes, we also explore using LLMs for self-fixing their errors in generating multi-file-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.

Our contributions include: (1) the first multi-file level benchmark for unit test generation with evaluation of nine frontier LLMs, (2) thorough error analysis through manual fixing of executability and cascade errors to reveal model potential, and (3) the first assessment of LLMs' self-fixing capabilities for unit test generation.

2 Related Work

2.1 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. While achieving reasonable coverage, these tests often have lower readability and less meaningfulness compared to developer-written tests, limiting their adoption in real-world scenarios (Almasi et al., 2017; Grano et al., 2019).

Large Language Models (LLMs) have demonstrated strong code generation capabilities (Feng et al., 2020; Wang et al., 2023), 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., 2025), adaptive context selection (Xie et al., 2023), dynamic scaling (Ma et al., 2025), 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.

2.2 LLM Unit Test Generation Benchmark

Current benchmarks for LLM-based unit test generation mainly focus on function-level (Wang et al., 2025; Villmow et al., 2021), class-level (Du et al., 2023), or single-file-level code (Jain et al., 2024a). Multi-file-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 equivalent test harnesses generation, 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 multi-file-level unit testing, its dataset is limited in quantity and varies in quality, especially for C/C# and Java, with only five projects each. Half of its projects for unit test generation are difficult to track, and most identifiable projects have fewer than 250 Stars and 50 Forks. Moreover, its broad focus prevents comprehensive evaluation and error analysis of LLM-based multi-file-level unit

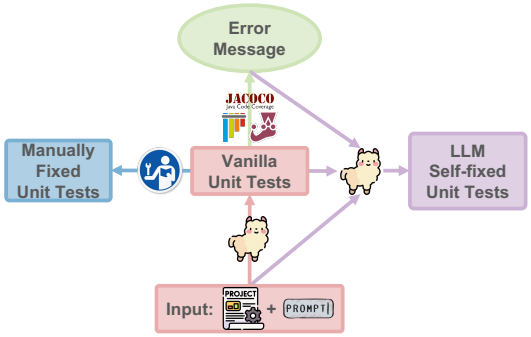

Figure 1: Overview of the unit test generation process.

test generation. We include a detailed comparison with other benchmarks in Appendix F.

3 Methodology

We introduce MultiFileTest dataset collection and preprocessing (§3.1), evaluation metrics (§3.2), and the unit test generation pipeline (§3.3) for evaluating LLMs on MultiFileTest across three unit test generation scenarios.

3.1 Benchmark Dataset

Dataset Collection. Our dataset comprises carefully selected multi-file-level GitHub repositories in Python, Java, and JavaScript. We establish selection criteria based on three key factors: 1) appropriate size (2-15 files, <1600 lines of code), 2) inter-file dependencies, and 3) reliable sources. The size threshold ensures code fits within standard LLM input windows without truncation, enabling fair comparison across models with different context lengths. This approach isolates our core evaluation target—the model's ability to generate unit tests—rather than testing long-context management or external tooling. We limit our selection to repositories with publicly available licenses, ensuring legality and openness. To maintain quality and reliability, we prioritize projects with high numbers of stars and forks, signaling community approval and widespread usage. We also extract smaller, self-contained projects from oversized codebases, carefully adjusting them to function independently without relying on the original larger projects. After applying these criteria, we construct 20 representative projects per programming language. Dataset statistics are summarized in Table 1, with detailed information on dataset sources and project-specific information in Appendix A.

Pre-processing. Dataset pre-processing involves several key steps to ensure the projects are well-

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

Table 1: MultiFileTest Data Statistics (LOC = Lines of Code).

structured and suitable for testing. First, we verify all selected projects for syntax errors despite their reliable sources. Second, for projects extracted from larger codebases, 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 span multiple lines into a single line, ensuring more valid metrics. Additionally, we maintain original coding styles as much as possible to preserve diversity across projects, allowing us to assess how LLMs perform when faced with various programming styles.

3.2 Evaluation Metrics

We focus on three key aspects when evaluating the generated unit tests: executability rate, correctness rate, and coverage rate. *Executability rate* (ER) measures the percentage of projects in which the generated test suites compile/execute successfully, indicating how often LLMs produce executable unit test suites. The executability rate for all projects in X is defined as $ER = \frac{|X^{er}|}{|X|}$, where X is the project set and $X^{er} \subset X$ denotes the subset of projects whose test suites compile/execute 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.1. 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 explore 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

```
# 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 MultiFileTest.

project $x \in X$.

These three evaluation metrics are interdependent. If a project's generated test suite contains executability errors, none of its unit tests can execute successfully, resulting in zero correctness and coverage rates for the project. Additionally, errors causing test failures, such as missing Python dependencies, can also impact coverage rates. Therefore, considering these interdependencies, we extend our analysis beyond vanilla unit test evaluation to include manually fixing these errors. This enables a more comprehensive assessment of LLMs' potential to generate high-quality unit tests once such 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 LLM unit test generation process. Our unit test generation and evaluation aim to ensure fair and thorough assessments under different scenarios:

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

Scenario 1: Vanilla Unit Test Generation. We input the entire project and a carefully crafted prompt into LLMs, ensuring the context and requirements are clearly communicated. Complete project codes are provided to ensure LLMs have all the necessary context to generate unit tests for the entire project, as shown in Figure 2. To rigorously evaluate LLM capabilities, we craft language-specific prompts addressing the unique challenges of each programming language. A comprehensive assessment is ensured by requiring LLMs to generate unit tests for all project files and providing targeted instructions on executability rate, correctness rate, and coverage metrics. This methodical prompt engineering significantly enhances the quality and relevance of the LLM-generated outputs. Appendix B.1 lists all experiment prompts, while Appendix D.1 contains the prompt ablation analysis. Vanilla unit tests are extracted directly from the LLM response based on the input project and prompt.

```
# 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 3: Executability error (tests don't run).

```
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 4: Cascade error (tests run, multiple fail).

Scenario 2: Manual Fixing. Manually fixing executability and cascade errors is motivated by empirical observations from scenario 1, where even unit tests generated by state-of-the-art LLMs like Claude-3.5-Sonnet contain significant executability errors, making them non-executable. These tests also exhibit cascade errors that, while easily fixable, can impact multiple unit tests or the entire test suite (details in § 5.5). Although these errors are preliminary and straightforward to resolve, they obstruct a deeper analysis of other critical aspects of LLM performance in unit test generation, particularly correctness and coverage.

Therefore, we apply minimal necessary changes to vanilla unit tests, resolving executability and cascade errors while preserving the original test intent. Executability errors prevent test execution

entirely. For example, Figure 3 shows a ModuleNotFoundError causing pytest to fail before collecting tests, resulting in zero correctness and coverage. [2] Cascade errors occur during test execution and cause cascading failures across multiple unit tests due to a single root cause. For instance, Figure 4 shows a missing NumPy import invalidating multiple fundamentally correct tests.

Manual Fixing Protocol. To ensure consistency and minimize annotator bias, we target only mechanical errors with objectively verifiable solutions, detailed in Appendix I. These fixes require minimal reasoning—the correct solution is determinable from codebase structure (e.g., import paths from file organization). Fixes average 2-4 lines per project (Table 13), are validated by successful test execution, and preserve all test logic and assertions. Since the protocol applies uniformly across all models, this scenario enables fair comparative assessment of model potential. Manual fixes are performed by CS PhD candidates specializing in software engineering and program analysis. Additionally, we evaluate unit tests with only executability errors fixed in Appendix D.2.

Scenario 3: LLM Self-fixing. Inspired by our observation from manual fixing that different LLMs exhibit significantly different potentials after manual fixing, we investigate how LLMs perform in self-fixing on our benchmark. We explore LLMs' self-fixing abilities by incorporating human-LLM conversation history and error messages as shown in Figure 5. 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 an open-source LLM's input length is limited, we prioritize the information hierarchically: system prompt, LLM's initial response, error messages, error-fixing requests, and unit test generation requests. We truncate less critical information as necessary while reserving at least 2,000 tokens for the 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 executability or cascade errors.

[2]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 5: The prompt used for the LLM self-fixing scenario for Python projects.

4 Experimental Settings

4.1 Models

We evaluate five close-sourced models: GPT-o1, Gemini-2.0-Flash-Exp (Gemini-2.0-Flash) (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 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 with temperature 0 for unit test generation, running experiments on 8 NVIDIA A100 GPUs with input length maximized to each LLM's token limit. We use Pytest[3] for Python, JUnit[4] for Java, and Jest[5] for JavaScript regarding testing frameworks. For Java code coverage, we use JaCoCo[6].

5 Experiments

We evaluate the generated unit tests from three scenarios, vanilla (§ 5.1), after manual fixing of executability and cascade errors (§ 5.2), and LLM self-fixing (§ 5.3). For each scenario, we evaluate the Correctness Rate (CR), Executability Rate (ER), 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

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

Model	CR	ER	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-2.0-Flash	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-2.0-Flash	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-2.0-Flash	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

Table 2: Main Results. CR: Correctness Rate (%), ER: Executability Rate (%), LC: Line Coverage (%), BC: Branch Coverage (%).

any changes. This scenario assesses the LLMs' raw capability to generate multi-file-level unit tests.

Table 2 shows the evaluation results for vanilla unit tests. First, LLMs demonstrate varying language-level expertise. For example, Claude-3.5-Sonnet performs the best in Java but falls behind GPT-o1 in JavaScript. Second, 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 in CR and ER while falling behind GPT-o1 in LC and BC.

Among three programming languages, Java poses the greatest challenge due to its stricter syntax requirements. Many models fail to generate valid Java code, leading to low executability 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-2.0-Flash tends to generate the most unit tests but does not obtain the best coverage rate. Additionally, sometimes the open-source model can even outperform some closed-source models. For example, DeepSeek-Coder surpasses 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 executable unit tests, resulting in the lowest correctness rates and coverage rates. We verify the robustness of these experimental results through multiple runs in Appendix C.2.

5.2 Manual Fixing Results

Table 3 presents evaluation results after manual fixing, highlighting substantial improvements compared to vanilla outputs across all programming languages and LLMs. These significant gains demonstrate that LLM-generated unit tests are highly sensitive to executability and cascade errors.

Among programming languages, Java benefits most from manual fixing. In the vanilla scenario, Java exhibits the lowest executability rates, making it particularly challenging. However, after manual fixing, Java shows the most substantial improvement, highlighting the potential of LLMs for Java after fixing executability and cascade errors. Among all models, GPT-o1 maintains its superior performance after manual fixing, while CodeLlama and CodeGemma continue to demonstrate the weakest overall results. Gemini-2.0-Flash shows the best coverage improvement overall, indicating exceptional potential for better unit test generation once executability and cascade errors are fixed. Our analysis reveals that manual fixing can reorder model performance rankings. For example, in Java, CodeQwen1.5 outperforms DeepSeek-Coder and is now on par with GPT-4-Turbo after fixing. In Python, Gemini-2.0-Flash surpasses CodeQwen1.5, showing better potential post-fix. In JavaScript, GPT-3.5-Turbo reaches parity with DeepSeek-Coder.

5.3 LLMs Self-fixing Results

LLM self-fixing utilizes human-LLM conversation history and error messages to assist LLMs in fixing errors. This scenario assesses LLMs' self-fixing

Model	CR	ER	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-2.0-Flash	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-2.0-Flash	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-2.0-Flash	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 3: Manual Fixing Results with Improvements Shown in Parentheses.

Model	CR	ER	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-2.0-Flash	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-2.0-Flash	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-2.0-Flash	**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

Table 4: Evaluation Results after Self-fixing. The Comparisons with Manual Fixing are Shown in Parentheses.

capabilities and their potential to generate better unit tests by incorporating self-fixing.

Table 4 shows the LLM self-fixing evaluation results compared with manual fixing results. First, we observe that most closed-source models have effective self-fixing abilities, generating better unit tests than vanilla results. In contrast, the evaluated open-source models lack reliable self-fixing abilities. This limitation likely stems from restricted input length, which leads to incomplete context, alongside weaker comprehension and instruction-following abilities. For instance, CodeGemma and CodeLlama tend to generate textual instructions for fixing errors rather than directly producing the corrected unit tests specified in the prompt.

Second, LLM self-fixing follows similar but not identical trends to manual fixing, suggesting that although LLMs' improvement potential generally aligns with self-fixing capabilities, some LLMs deviate from this pattern. In JavaScript, GPT-o1's self-fixing yields substantially lower coverage rates compared to manual fixing due to generating fewer unit tests and achieving lower executability rates.

Despite currently underperforming compared to manual fixing, LLM self-fixing demonstrates 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 than manual fixing. This occurs because CodeQwen1.5 occasionally misinterprets prompts in vanilla outputs, generating no unit tests. While manual fixing cannot remedy this fundamental understanding issue, self-fixing enables the model to correctly interpret test generation requirements when error messages indicate missing tests.

5.4 Unique Contribution of Unit Tests

Beyond standard coverage metrics, we introduce unique contribution to assess the efficiency and non-redundancy of generated unit tests. While similar uniqueness metrics have been explored in prior work (e.g., TestPilot (Schäfer et al., 2023)), we apply this concept to our distinct context of multi-file test generation scenarios. The unique contribution is defined as the total portion of coverage

Model	#Tests	LC	BC	Unique
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-2.0-Flash	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	8.90
CodeLlama	19.30	41	37	5.55
CodeGemma	15.00	31	28	2.70

Table 5: Unique Contribution on Vanilla Unit Tests.

contributed by each generated unit test that does not overlap with the coverage of other unit tests. This measure addresses two critical limitations of conventional metrics. First, it accounts for variations in test quantity across different LLMs, as relying solely on coverage rate becomes insufficient when models produce widely differing numbers of tests. Second, it recognizes the importance of achieving high coverage with minimal tests, as executing numerous tests can be resource-intensive and time-consuming. Further details in Appendix G.

Table 5 reveals that all tested LLMs exhibit low unique contribution rates, indicating a tendency toward redundant and repetitive unit tests. Although GPT-o1 has better coverage rates than Claude-3.5-Sonnet, it produces significantly more unit tests, and its unique contribution is lower than Claude-3.5-Sonnet's, indicating it prioritizes quantity over quality to attain higher coverage. This approach potentially compromises the overall efficiency of the testing process.

5.5 Error Analyses

We analyze executability, cascade, and post-fix errors per programming language, identifying common errors and their underlying causes. Full analyses in Appendix E.

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

Cascade Error Analyses. ***Python*** cascade errors include missing imports (e.g., numpy, unittest, project functions/classes) and FileNotFoundError from unmocked external files. ***Java***'s primary cascade error is improper/missing mocking of user interactions, causing unusable coverage reports when tests terminate abruptly. ***JavaScript*** struggles with missing imports (e.g., chai, three, project functions/classes), confusion between named and default imports, and Jest framework compliance issues.

Post-Fix Error Analyses. Across all languages, mismatches between expected and actual values are the most common error. ***Python*** frequently encounters AttributeError from hallucinated attributes. ***Java*** suffers from NullPointerException, zero interactions with mocks, and failures to release mocks due to improper usage. ***JavaScript*** commonly faces TypeError, typically caused by LLMs hallucinating non-existent functions and constructors or LLMs invalidly mocking some variables.

Overall. Persistent errors across 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*, which highlights the significant gap between LLM unit test generation at function/class/single-file levels and at multi-file level. Failure to understand the codebase structure and dependencies can further cause issues like 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 multi-file-level unit test generation benchmark – MultiFileTest – with three programming languages. We comprehensively evaluate nine LLMs' unit test generation abilities with/without manual fixing and LLM self-fixing mechanism on MultiFileTest. Besides, we conduct comprehensive error analyses per programming language.

Limitations

Our study has several limitations. First, our focus is primarily on three programming languages—Python, Java, and JavaScript—excluding other relevant languages such as C and C#.

Second, the scale of projects in our benchmark is limited to approximately 1600 lines of code, which is smaller than many production-scale codebases. This constraint stems from the inherent input length restrictions and context window limitations of current LLMs, which make processing very large codebases impractical for tasks like unit test generation without introducing confounding variables. Despite this size constraint, these projects are designed to retain key structural characteristics of larger codebases, including multiple files with meaningful inter-file dependencies, cross-file function calls, class inheritance, and shared utility components. This ensures the benchmark still evaluates reasoning across files, which is central to multi-file-level unit test generation. Our experimental results demonstrate that even at this reduced scale, multi-file-level unit test generation remains challenging for state-of-the-art models like Claude-3.5-Sonnet. Expanding to significantly larger codebases would likely shift the evaluation focus toward context handling techniques (e.g., truncation, retrieval, or hierarchical methods) rather than core LLM test generation ability. While our benchmark does not represent the full complexity of production systems, it serves as a meaningful and challenging step toward that goal, providing valuable evaluation grounded in the practical capabilities of current LLMs.

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, Wannita Takerngsaksiri, Chetan Arora, and Aldeida Aleti. 2025. Enhancing large language models for text-to-testcase generation. *Journal of Systems and Software*, page 112531.

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.

Amr Almorsi, Mohanned Ahmed, and Walid Gomaa. 2024. Guided code generation with llms: A multi-agent framework for complex code tasks. In *2024 12th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*, pages 215–218. IEEE.

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

Mihir Athale and Vishal Vaddina. 2025. Knowledge graph based repository-level code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 169–176. IEEE.

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*.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576.

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.

Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. In *Forty-first International Conference on Machine Learning*.

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.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and

natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

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*.

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

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

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.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6917–6935, Vienna, Austria. Association for Computational Linguistics.

AI Meta. 2025. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. *https://ai. meta. com/blog/llama-4-multimodal-intelligence/, checked on*, 4(7):2025.

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*.

Johannes Villmow, Jonas Depoix, and Adrian Ulges. 2021. ConTest: A unit test completion benchmark featuring context. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 17–25, Online. Association for Computational Linguistics.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1258–1268.

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*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484.

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*.

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].

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 6: Dataset Details (Python).

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

Table 7: Dataset Details (Java).

B More Implementation Details

B.1 Prompts

The prompts are displayed in Figure 6, 7, 8, and 9.

B.2 Models

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

C More Experiments and Statistics

C.1 Assert 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

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

Table 8: Dataset Details (JavaScript).

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

C.2 Robustness Analysis

To address concerns about statistical robustness, we conduct three independent runs of unit test generation using GPT-3.5-Turbo as shown in Table 11. The variance across these runs is minimal, indicating that model performance on MultiFileTest is stable and reproducible, further supporting the benchmark's reliability.

We also conduct additional statistical validation using the correctness rate across all nine models and three scenarios as paired observations, as shown in Table 12. We find statistically significant differences (all p-values ≤ 0.009) with large effect sizes (Cohen's d > 1.1) across vanilla, manual, and self-fixing scenarios.

C.3 Changed LOC Statistics of Manual Fixing

We calculated the average number of lines of code (LOC) changed during manual fixing for Python projects across all models in Table 13. We observe that the amount of manual edits is modest and con-

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-2.0-Flash	-	https://ai.google.dev/gemini-api/docs/models/gemini#gemini-2.0-Flash
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 9: Model Details.

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%

Table 10: Percentages of the Vanilla Unit Tests Containing Expected and Actual Value Comparisons.

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.

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.

Metric	**Run 1**	**Run 2**	**Run 3**	**Mean**	**Variance**	**Std Dev**
CR	0.37	0.34	0.37	0.36	0.0003	0.0141
ER	0.60	0.65	0.65	0.633	0.0003	0.0236
LC	38%	40%	39%	39%	0.0001	0.01
BC	34%	37%	35%	35.3%	0.00015	0.0122

Table 11: Performance Metrics across Multiple Runs Using GPT-3.5-Turbo on Python.

sistent across models. These findings suggest that while models frequently produce errors, many are shallow and fixable with minimal human effort, which reinforces the value of human-in-the-loop and LLM-self-fix workflows.

C.4 Comparison with Other Methods

We use the ChatUnitest (Chen et al., 2024) Maven Plugin and follow Wang et al. (2024) to evaluate GPT-3.5-Turbo on Java projects. HITS not only uses iterative debugging, but also uses sophisticated techniques like method slicing to improve unit test performance. These results, as shown in Table 14, further emphasize the difficulty of MultiFileTest, as even the iterative debugging and complex method still achieve low coverage rates. This comparison helps validate our benchmark and encourages further innovation in LLM-driven test generation.

We conduct additional experiments with EvoSuite (Fraser and Arcuri, 2011), a leading search-based test generation tool for Java. Table 14 presents the Line Coverage (LC) and Branch Coverage (BC) of EvoSuite compared to GPT-o1 on Java projects. The results show that vanilla LLMs fall behind EvoSuite, while LLM self-fixing has comparable performance with EvoSuite under this multi-file unit test generation setting.

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 executability rate (ER), and the coverage rate metrics (Coverage). We ablate each component and analyze the performance of unit test generation of GPT-4-Turbo

Comparison	Language	Vanilla/Manual Mean	Manual/Self Mean	t-statistic	p-value	Cohen's d
Vanilla vs Manual	Python	38.2%	61.8%	-4.85	0.001	1.62
Vanilla vs Manual	Java	17.2%	57.9%	-8.71	<0.001	2.91
Vanilla vs Manual	JavaScript	52.1%	74.0%	-5.23	0.001	1.74
Manual vs Self-fix	Python	61.8%	39.1%	3.42	0.009	1.14
Manual vs Self-fix	Java	57.9%	25.9%	4.67	0.002	1.56
Manual vs Self-fix	JavaScript	74.0%	49.6%	3.89	0.005	1.30

Table 12: Statistical comparison of the Correctness Rate of different scenarios across languages.

Model	GPT-4	GPT-3.5	GPT-o1	Gemini-2.0	Claude-3.5	CodeQwen1.5	DeepSeek	CodeLlama	CodeGemma
LOC Changed	2.45	3.35	3.15	3.15	4.05	3.4	2.35	3.0	3.4

Table 13: Lines of code changed during manual fixing for Python projects.

Method	Model	CR	ER	LC	BC
Vanilla	GPT-3.5-Turbo	13	25	8	7
Manual fix	GPT-3.5-Turbo	54	100	36	27
Self-fix	GPT-3.5-Turbo	17	25	11	12
HITS	GPT-3.5-Turbo	75	80	41	29
Vanilla	GPT-o1	41	60	44	35
Manual fix	GPT-o1	64	100	65	56
Self-fix	GPT-o1	68	85	58	54
EvoSuite	-	-	-	55	57

Table 14: Comparison with Traditional Method EvoSuite, and LLM-based Methods HITS on Java Projects.

Phase	Settings	CR	ER	LC	BC	#Tests	#Correct
Vanilla	Full Prompt	47	65	40	36	12.60	6.15
	w/o CR	33 ↓	65	42	38	12.75	4.75
	w/o ER	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	Full Prompt	74	100	65	59	12.60	9.30
	w/o CR	76 ↑	100	69	64	12.75	9.90
	w/o ER	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

Table 15: Ablation Study. The Performance of Unit Test Generation by GPT-4-Turbo Using Different Prompts.

using different prompts as shown in Table 15. Requirements related to CR and ER 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 ER, LC, and BC.

Besides, we follow the prompt template from previous work, such as Siddiq et al. (2024), to move the prompts into comments (e.g., /*...*/). We compare the performance with and without comment signs in Table 15. Experimental results show that our prompt demonstrates a significant advantage in CR, while the prompt with comment signs exhibits marginal advantages in ER, LC, and BC.

D.2 Effect of Executability Errors and Cascade Errors

We manually fix only executability errors and evaluate the corrected unit tests in Table 16.

By fixing executability errors, Table 16 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 executability errors. Comparing Table 16 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 16 are primarily due to missing or invalid mocks of user interactions[8] which occur more frequently in unit tests generated by CodeQwen1.5 and CodeGemma.

E Detailed Error Analyses

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

Executability Error Analyses Figure 10 highlights the detailed executability errors that occurred. One of the most common executability 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

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

Model	CR	ER	LC	BC	#Tests	#Correct
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-2.0-Flash	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-2.0-Flash	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-2.0-Flash	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

Table 16: Evaluation Results When Only Manually Fixing Executability Errors.

leads LLM to fail to correctly import functions or classes from the tested project. Other executability 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 executability errors, resulting in a significantly lower executability rate than other languages. Java executability 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 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 executability 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 executability 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 refuse to generate unit tests. Other executability 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.

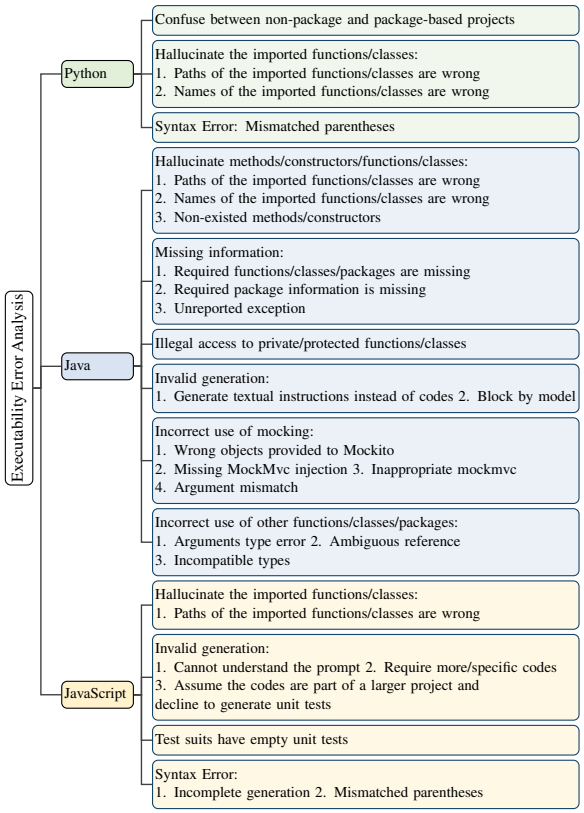

Figure 10: Frequent Executability Errors in Main Results.

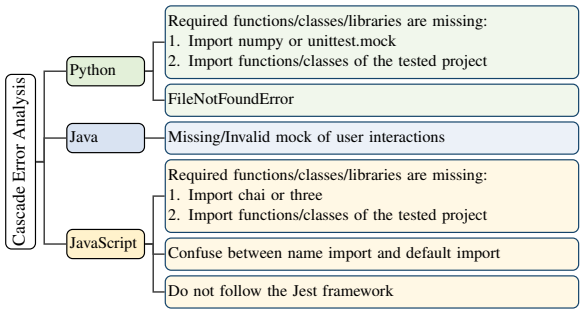

Figure 11: Frequent Cascade Errors.

F Comparison with Other Benchmarks

Table 17 presents a comprehensive comparison of major code evaluation datasets across multiple dimensions. Among these, MultiFileTest stands out as the first benchmark specifically designed for multi-language, multi-file unit test generation with robust error analysis capabilities. We particularly highlight the distinction between DevBench and MultiFileTest: while DevBench addresses broader software engineering tasks across the entire development lifecycle, MultiFileTest is specifically designed for unit test generation, providing 60 projects (20 per language) compared to DevBench's smaller subset for unit testing. Furthermore, MultiFileTest uniquely offers fine-grained error analysis and both manual fixing and LLM self-fixing mechanisms, which are not present in DevBench. This makes MultiFileTest particularly valuable for evaluating and improving LLMs' capabilities in generating functional test suites for multi-file software projects.

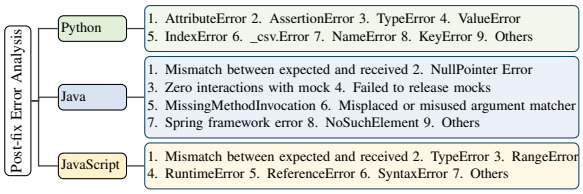

Figure 12: Frequent Post-Fix Errors.

G Comparison between Unique Contribution and Other Metrics

While alternative metrics such as test execution time or lines of code provide valuable insights in single-project contexts, they present significant challenges in multi-project benchmarks. The heterogeneous nature of our benchmark—spanning diverse programming languages, project scales, and architectural paradigms—makes these conventional metrics difficult to normalize meaningfully across projects. Test execution times fluctuate based on external dependencies and environmental factors, while code size metrics vary substantially due to languages and coding styles. In contrast, our unique contribution metric offers a project-agnostic measurement framework that maintains consistent interpretability across the entire benchmark suite. It provides a standardized proxy for test utility that transcends project boundaries. This normalized approach enables meaningful cross-project comparisons that would be impractical with traditional metrics, addressing the specific evaluation requirements of diverse multi-project benchmarks.

H Discussion on Context Window Limitations

To address the context window limitation, we identify three primary lines of methods that have emerged in recent research.

The first line focuses on extending context windows to accommodate larger codebases directly. Recent models demonstrate dramatic improvements, expanding from early limits of thousands of tokens to millions by 2024. LongRoPE (Ding et al.) extends pre-trained LLMs to 2048k tokens

Dataset	Language	Code Level	Multi-file	TestGen	Size	Avg. #Files	Self-contained	Error Analyses	Error Fixing
HumanEval (Chen et al., 2021)	Python	Function	✗	✗	164	1	✓	✗	✗
ClassEval (Du et al., 2023)	Python	Class	✗	✗	100	1	✓	✗	✗
SWE-bench (Jimenez et al.)	Python	Multi-file	✓	✗	12	-	✓	✗	✗
TestEval (Wang et al., 2025)	Python	Function	✗	✓	210	1	✓	✗	✗
TestGenEval (Jain et al., 2024a)	Python	Single-file	✗	✓	1,210	1	✗	✓	✗
DevBench (Li et al., 2024)	Python, Java, C/C#	Multi-file	✓	✓✗	20	4.20	✓	✗	✗
MultiFileTest (ours)	Python, Java, JavaScript	Multi-file	✓	✓	60	4.92	✓	✓	✓

Table 17: Benchmarks comparison. "TestGen" refers to whether the benchmark is designed for unit test generation. "Self-contained" refers to whether the data sample is independent rather than being part of a larger project. ✓✗ indicates partial satisfaction of the condition. "Error Analyses" refers to specific error analyses for unit test generation by LLMs.

with minimal fine-tuning while maintaining performance at shorter context windows. Llama 4 Scout (Meta, 2025) achieves a 10 million token context window.

The second line of methods employs Retrieval-Augmented Generation (RAG) to provide only important context instead of full context. This approach involves indexing codebase components and dependencies, then dynamically retrieving only the code segments most relevant to the target function for test generation (Lewis et al., 2020; Athale and Vaddina, 2025; Zhang et al., 2023). This methodology enables scalability while maintaining dependency awareness without overwhelming the context window.

The third approach utilizes hierarchical decomposition to break down large codebases into manageable components (Almorsi et al., 2024; Mündler et al., 2024). Often, this is achieved through agent-like methods that employ multi-pass strategies. These agents first analyze the high-level structure, then progressively focus on specific components while maintaining broader context awareness. This allows for the effective handling of larger systems by managing contextual information at different abstraction levels and enabling specialized agents to tackle sub-problems.

While these approaches show promise for scaling to production-size codebases, they introduce confounding variables that would complicate fair evaluation of core LLM test generation capabilities. Our benchmark's constraint to 1600 lines of code enables evaluation without truncation, retrieval strategies, or hierarchical preprocessing, allowing fair comparison across models with different context lengths. This approach isolates our core evaluation target—the model's ability to generate unit tests—rather than testing long-context management or external tooling. The three methods discussed above represent important future directions once foundational test generation capabilities are well-established and benchmarked at the scale our current LLM capabilities can reliably handle.

I Manual Fixing Protocol

I.1 Overview

Scope: Fix only executability errors (prevent test execution) and cascade errors (single root cause affecting multiple tests). Do NOT fix correctness errors, coverage issues, or test logic flaws.
Principle: Apply minimal necessary changes determinable objectively from error messages and codebase structure.

I.2 General Decision Framework

Algorithm 1 Error Fixing Decision Process

```
for each error in test execution do
    identify error type from error message
    locate matching rule in protocol
    if rule conditions satisfied then
        apply specified fix
        re-run tests to verify resolution
        document: file, line, error type, fix
    else
        do not modify
    end if
end for
```

I.3 Python Fixing Rules

We categorize fixes into executability errors (preventing test execution) and cascade errors (a single root cause affecting multiple tests).

I.3.1 Executability Errors

Table 18 summarizes fixable executability errors with their conditions and required actions.

Error Pattern	Condition	Fix Action	LOC
`ModuleNotFoundError: 'X'`	`__init__.py` exists	Change to `from pkg.module import X`	1
	No `__init__.py`	Change to `from module import X`	1
	X is external lib	Add `import X` at top	1
`ModuleNotFoundError: 'X.Y.Z'`	Extra nesting in path	Remove nesting to match file structure	1
`ImportError: name 'X'`	Similar name	Fix typo to existing name	1
	No similar name	Remove related lines	–
`SyntaxError: EOF/unmatched`	Unbalanced `()` or `[]`	Count and add/remove to balance	1

Table 18: Python Executability Error Fixes

Error Pattern	Undefined Name	Fix Action	LOC
`NameError: name 'X' not defined`	`np, pd, plt`	Add `import numpy as np` (etc.)	1
	`unittest, mock`	Add `import unittest` (etc.)	1
	Project class/function	Search project, add import per E1.1	1
`FileNotFoundError`	File access error	Fix file access	1-2

Table 19: Python Cascade Error Fixes

I.3.2 Cascade Errors

Cascade errors occur when a single error causes multiple test failures. These are mechanically fixable by identifying the undefined name and adding the appropriate import. Table 19 lists common patterns.

I.4 Java Fixing Rules

Java presents unique challenges due to its stricter syntax requirements and package system. Many executability errors stem from missing package declarations and incorrect import paths.

I.4.1 Executability Errors

Table 20 summarizes Java executability errors. The most distinctive Java error is the missing package declaration, which is deterministically derivable from the file's directory path.

I.4.2 Cascade Errors

Java cascade errors primarily involve missing test framework imports and user interaction mocks. Table 21 shows fixable patterns. User interaction errors are particularly common in Java due to reliance on `Scanner` for console input.

I.5 JavaScript Fixing Rules

The primary challenge for JavaScript is calculating correct relative import paths and distinguishing between named and default exports.

I.5.1 Executability Errors

Table 22 presents JavaScript executability errors. Unlike Python's package detection or Java's package declarations, JavaScript requires calculating relative paths between test and source files.

I.5.2 Cascade Errors

JavaScript cascade errors involve missing library imports and framework syntax mismatches. Table 23 shows common patterns.

I.6 Validation Protocol

Each fix must pass a five-step validation process to ensure objectivity and reproducibility, as shown in Table 24. This protocol is applied identically by all annotators to minimize subjectivity.

Error Pattern	Condition	Fix Action	LOC
`public class should be in file X.java`	Always	Extract package from path, add `package X;` at line 1	1
`cannot find symbol: class X`	Class exists in project	Find file, extract package, add `import pkg.X;`	1
`cannot find symbol: method X()`	Missing method import	Add `import static Y.X;`	1
`X has private access`	Always	Remove related lines	–
`incompatible types`	Always	Modify to correct types	–
Text/comments instead of code	Always	Invalid generation	–
`Mockito exceptions`	Missing import only	Add Mockito imports	1
	Wrong mock usage	Remove related lines	–

Table 20: Java Executability Error Fixes

Error Pattern	Condition	Fix Action	LOC
Test hangs / `NoSuchElementException`	Scanner input needed	Continue	1
`cannot find symbol: @Test, Assert, etc.`	`@Test`	Add `import org.junit.Test;`	1
	`assertEquals`	Add `import static org.junit.Assert.*;`	1
	`@Mock`	Add `import org.mockito.Mock;`	1

Table 21: Java Cascade Error Fixes

Error Pattern	Condition	Fix Action	LOC
`Cannot find module 'X'`	X is project file	Calculate relative path, fix import	1
	X is external	Add `require('X')` or `import X`	1
Import/require not working	Check export type	`export default` → `require()`, `export class` → `{X} = require()`	1
`SyntaxError`	Unbalanced brackets	Count and balance (same as Python)	1
	Incomplete generation	Remove incompleted parts	–
Empty test suite	Model refused to generate	Invalid generation	–

Table 22: JavaScript Executability Error Fixes

Error Pattern	Undefined Name	Fix Action	LOC
`ReferenceError: X not defined`	`expect, assert`	Add `const {expect} = require('chai');`	1
	`THREE, d3`	Add `const THREE = require('three');`	1
	Project component	Calculate path, add `require('../X')`	1
Wrong framework syntax	Jest in Mocha, etc.	Convert to correct framework	1-2

Table 23: JavaScript Cascade Error Fixes

Step	Check	Action
1	Error resolved?	Re-run test, verify specific error gone
2	New errors?	Treat as separate issues, apply protocol
3	Changes minimal?	Must directly address error, nothing extra
4	Deterministic?	Another annotator would make same fix
5	Documented?	Log: file, line, error type, rule, fix, LOC

Table 24: Per-Fix Validation Checklist