Developer vs. DSpot vs. ChatGPT: A Comparative Study of JUnit Test Amplification

David Onyango Owuor
North Dakota State University
Fargo, USA
david.owuor@ndsu.edu

Ajay Kumar Jha
North Dakota State University
Fargo, USA
ajay.jha.1@ndsu.edu

***Abstract*—Test amplification enhances the effectiveness of existing tests. As software evolves, developers often manually amplify tests, which is tedious and time-consuming. Automated techniques have shown promise in addressing this challenge, while Large Language Models (LLMs) have demonstrated potential for test generation. However, empirical evidence of how developers amplify tests remains limited, and the effectiveness of these approaches has not yet been evaluated on a common dataset. To address this gap, we construct a dataset of 117 developer-amplified tests and characterize manual test amplifications. We amplify the corresponding original tests using DSpot (a tool) and ChatGPT (an LLM) and compare their effectiveness in terms of code and mutation coverage. We find that manual amplifications achieve modest improvements (mean line and branch: 0.74%, mutation: 0.90%). Method calls, inputs, expected outputs, and assertions are modified in 102, 104, 85, and 65 tests, respectively. DSpot substantially outperforms developers in all metrics (mean line: 5.66%, branch: 4.13%, mutation: 4.44%). While ChatGPT outperforms both developers and DSpot in line (mean 8.91%) and branch (mean 9.74%) coverage, it underperforms DSpot in mutation score (mean 3.20%). These findings can guide the design of more effective and intelligent test amplification tools.**

***Index Terms*—Software testing, test amplification, unit test**

I. Introduction

Testing is essential for ensuring software reliability [1]. Nonetheless, maintaining high code and mutation coverage remains challenging, as software evolves [2]. Test amplification mitigates this challenge by enhancing existing tests [3].

Developers perform test amplification during software maintenance by adding inputs, refining assertions, or expanding method calls [4], [5]. However, manual amplification is tedious, time-consuming, and difficult to scale, as developers must identify uncovered code behavior and determine effective test inputs and oracles to exercise the behavior. Test amplification tools, such as DSpot [3], systematically transform tests to increase coverage and fault detection capabilities, thereby reducing manual effort [6]. Recently, large language models (LLMs) have shown promise in test generation [7]–[10].

Despite these advances, gaps remain in test amplification. First, developer-amplified tests have not been systematically characterized, leaving the strategies used by developers unexplored. Second, although tools such as DSpot have been evaluated on real-world test suites, their performance has not been assessed on manually amplified tests, limiting understanding of how tool amplifications compare to developer amplifications. Finally, no prior study has systematically compared manual, tool-based, and LLM-based amplification using a common dataset, leaving their relative effectiveness unclear.

To address these gaps, we conduct an empirical study on test amplification, focusing specifically on enhancing existing tests [11]. We first construct AmpTest, a dataset of 117 developer-amplified tests from five open-source Java projects. We then manually analyze these tests to identify the amplification strategies used by developers. Finally, we amplify the corresponding original tests using DSpot [3] and ChatGPT (GPT-5), and compare the effectiveness of developer (manual), DSpot (tool-based), and ChatGPT (LLM-based) amplification in terms of code and mutation coverage improvements.

We find that developers amplify tests by modifying method calls, inputs, expected outputs, and assertions. Focal method calls are added in 93 tests, with 81 reinvoking existing methods using different inputs and 78 introducing new focal methods. Test inputs are modified in 104 tests, including 21 null cases. DSpot outperforms developers across all metrics, improving mean line, branch, and mutation coverage by 5.66%, 4.13%, and 4.44% (vs. 0.74%, 0.74%, 0.90). ChatGPT achieves even higher line and branch coverage (mean 8.91% and 9.74%), but underperforms DSpot in mutation coverage (mean 3.20%), likely due to lacking DSpot's systematic mutation exploration.

The results suggest that DSpot and ChatGPT amplifications are largely complementary. Test amplification tools could leverage their complementary strengths, with insights from manual strategies, to achieve greater effectiveness. To enable replication and future research, we share all artifacts[1].

To summarize, this study makes the following contributions:

- Constructs AmpTest, a dataset of 117 developer-amplified tests from five open-source Java projects.
- Characterizes manual test amplification strategies.
- Evaluates manual, tool-based, and LLM-based approaches, highlighting their strengths and limitations.

II. Empirical Study

The following two research questions guide this study:

- **RQ1:** *What strategies do developers use to amplify existing tests?* We construct AmpTest, a dataset of 117 developer-amplified tests from five Java projects, and analyze the tests to identify amplification strategies.
- **RQ2:** *How effective are manual, tool-based, and LLM-based test amplification approaches in improving test*

[1]https://figshare.com/s/af74252a9171e79676d6

quality? We use DSpot [3] and ChatGPT as representative techniques, amplify the original versions of 117 developer-amplified tests with both tools, and evaluate all resulting tests using code and mutation coverage.

The results of this study can inform researchers and practitioners in the development of test amplification tools by elucidating how developers amplify tests and highlighting the opportunities and limitations of different approaches.

A. AMPTEST*: A Dataset of Developer-amplified Tests*

We begin constructing AMPTEST by identifying developer-amplified tests from six open-source Java projects used in prior studies [3], [4], shown in Table I. The projects are popular, mature, and actively maintained.

TABLE I: Studied Projects

Project	#Commits	#Tests	GitHub Stars
Commons Lang	8,255	3,847	2.8k
Commons Math	7,192	4,484	605
Gson	2,064	1,443	23.7k
PMD	28,863	1,922	5k
JFreeChart	4,226	2,281	1.3k
Joda-Time	2,289	1,157	5k
Total	52,889	15,134	—

1) Identify Candidate Commits: We identify commits that may contain amplified tests by focusing on those with modified tests that improve coverage. However, calculating coverage improved by each modified test in each commit is challenging. Therefore, we target self-admitted amplified tests where developers state they improved coverage by modifying tests.

We search commits in each project using *coverage* keyword, and manually analyze the commit message and the modified code in each resulting commit. We consider a commit as a candidate commit if it satisfies all the following five criteria. (1) The message mentions increased coverage. (2) It contains at least one modified non-empty test. (3) It does not modify production code, as we are only interested in test amplification through test code modification. (4) It is not a merge commit. (5) It is available on GitHub.

The search results in 1,523 commits; among these, only 523 commits have a message indicating an increase in coverage (the first criteria). Applying the remaining criteria to 523 commits results in 64 commits containing 121 modified tests.

2) Identify Amplified Tests: We confirm that 121 modified tests are amplified by computing their class-level line and branch coverage improvements. We first calculate the original coverage by running JaCoCo[2] on the direct parent commit of each candidate commit. Then, for each modified test in the candidate commits, we replace the corresponding original test in the parent commit with the modified test and compute the coverage. If the modified test improves the original line or branch coverage, we consider the test as amplified. Out of 121 tests, 77 tests from three projects improve coverage. The remaining 44 tests are refactored and do not improve coverage.

To enhance the generalizability of our findings, we expand AMPTEST by including two more projects containing amplified tests. We search GitHub repositories with at least 1,000 stargazers and apply the steps described in Sections II-A1 and II-A2, yielding 40 amplified tests. Overall, AMPTEST contains 117 developer-amplified tests across 45 commits from five projects, as shown in Table II.

TABLE II: AMPTEST: Developer-amplified Tests

Project	#Commits	#Amplified Tests
Commons Lang	25	57
Commons Math	10	18
PMD	1	2
Commons IO	4	12
Jackson Databind	5	28
Total	45	117

B. ***RQ1:*** *What strategies do developers use to amplify existing tests?*

1) Method: We analyze developer-amplified tests using a closed coding approach [12]. A JUnit test typically consists of four key elements. (1) *Method call* – one or more methods or constructors under test, or supporting methods. (2) *Input* – values provided to exercise the methods, and the receiver object on which a method is invoked. (3) *Expected output* – values the methods or constructors under test are expected to produce. (4) *Assertion* – a mechanism for comparing expected and actual outputs. Each code change in the amplified tests is labeled as a method call, input, expected output, or assertion.

To gain further insights into amplification strategies, we classify each change as (1) *new* – if the element does not exist in the original version, or (2) *existing* – if it is reused from the original version without modification. If an input or expected output has multiple values (e.g., multiple arguments, a list), modifying any value classifies it as new. We also recorded their data types, distinguished between focal methods (the methods under test) and supporting methods, and noted whether methods were overloaded. Two authors jointly review the amplified test code changes in online sessions, discussing and assigning labels to each change.

2) Results: Table III shows the types of test code elements added to amplify existing tests. Method calls, inputs, expected outputs, and assertions are added in 102 (87.2%), 104 (88.9%), 85 (72.6%), and 65 (55.6%) tests, respectively. The total percentage may not equal 100%, since a single test amplification can involve adding multiple types of code elements.

Focal method calls are added in 93 tests, with new focal methods introduced in 78 tests and existing methods reinvoked in 81 tests. Among the existing ones, 60 use the same signature and 21 are overloaded variants. Supporting method calls are added in 54 tests. New inputs are introduced in 99 of 104 input-modified tests (95.2%), including 21 null cases. New

[2]https://www.eclemma.org/jacoco/

TABLE III: Code elements types added to amplify tests. Percentages are relative to the immediate parent type.

Modified Code Elements				Num. of Tests
Method Call				**102 (87.2%)**
	Focal			**93 (91.2%)**
		New		78 (83.9%)
		Existing		**81 (87.1%)**
			Same Signature	60 (74.1%)
			Overloaded	21 (25.9%)
	Supporting			**54 (52.9%)**
		New		54 (100%)
		Existing (Same Signature)		11 (20.4%)
Input				**104 (88.9%)**
	New			**99 (95.2%)**
		Primitive		27 (27.3%)
		String		33 (33.3%)
		Class/Object		58 (58.6%)
		Null		21 (21.2%)
	Existing			**5 (4.8%)**
Expected Output				**85 (72.6%)**
	New			**56 (65.9%)**
		Primitive		21 (37.5%)
		String		10 (17.9%)
		Class/Object		32 (57.1%)
		Null		3 (5.4%)
	Existing			**29 (34.1%)**
Assertion				**65 (55.6%)**
	New			61 (93.8%)
	Modified			4 (6.2%)

TABLE IV: Combinations of modified code elements

Modified Code Elements	Number of Tests
Input + Method + Assertion + Output	49 (41.9%)
Input + Method + Output	24 (20.5%)
Input + Method	17 (14.5%)
Method + Assertion + Output	12 (10.3%)
Other	15 (12.8%)

expected outputs appear in 56 of 85 tests (65.9%), while 29 reuse existing outputs (34.1%). Class/object instances are most frequent among new outputs (57.1%). Among the 65 tests involving assertions, 61 have new assertions (93.8%).

Table IV presents the observed combination patterns. The most common pattern, involving all four types (*input–output–method–assertion*), appeared in 49 tests (41.9%).

3) Discussion: Test amplifications mainly reinvoke existing focal methods (81 tests), often with new inputs. This suggests that developers frequently aim to explore known behaviors under varied conditions, thereby increasing coverage of the original functionality. 78 tests also introduce new focal method calls, reflecting efforts to expand the behavioral scope of tests. While such extensions can improve coverage, introducing new focal methods in separate tests is preferable to maintain focus and diagnostic clarity and to avoid eager test smells [13].

RQ1: Developers primarily amplify tests by reinvoking existing focal methods (81 tests) with new inputs (99 tests), increasing coverage of known behaviors. Introducing new focal method calls is also common (78 tests), reflecting efforts to expand behavioral scope.

C. ***RQ2:*** *How effective are manual, tool-based, and LLM-based test amplification approaches in improving test quality?*

1) Method: We select DSpot [3] version 3.2.0 [14] and ChatGPT (GPT-5) as tool-based and LLM-based approaches, respectively. We configure DSpot with its mutation-based selector (DSpotMut) and a time budget of 120 seconds per test. Using these tools, we amplify the original tests corresponding to the 117 developer-amplified tests and measure improvements in line, branch, and mutation coverage to compare developer, DSpot, and ChatGPT amplifications.

We apply DSpot [3] at the class level, amplifying all tests within classes that contain the original tests. This setup allows DSpot to operate with full context, including shared fixtures, helper methods, and dependencies. DSpot generates candidate amplifications through input, assertion, and method-level transformations, guided by coverage and mutation-based fitness objectives. From DSpot's outputs, we retain only those corresponding to developer-amplified tests in AMPTEST.

For ChatGPT-based amplification, we adopt an iterative refinement prompting strategy to construct the final prompt. The initial prompt is refined over multiple iterations to improve output quality while maintaining a zero-shot setting (no examples). For each amplification, ChatGPT is supplied with the source code of the target production class containing the tested method, along with the original test class. The final prompt used is: *"Amplify the JUnit test named <<testName>> by modifying inputs, expected outputs, assertions, and method calls. Generate one meaningful amplified version focused on improving code coverage and mutation score."*

We calculate line, branch, and mutation coverage improved by the amplified tests at the class-level using the method described in Section II-A2. Mutation scores are calculated using PIT[3].

2) Results: ChatGPT and DSpot successfully amplify all 117 tests, each improving at least one of line, branch, or mutation coverage. Figure 1 presents the improvements achieved by all three approaches across line, branch, and mutation metrics. Before amplification, the average coverage of the target classes was 91.1% for line, 89.8% for branch, and 88.9% for mutation, indicating that the code was already well tested.

All 117 tests amplified by developers and ChatGPT improve line coverage, whereas DSpot improves line coverage in 114 tests. ChatGPT substantially outperforms both developers and DSpot, increasing mean line coverage by 8.91% compared to 0.74% for developers and 5.66% for DSpot. All three approaches improve branch coverage for all 117 tests. ChatGPT substantially outperforms developers (9.74% vs. 0.74% mean gain), while DSpot also surpasses developers with a 4.13%

[3]https://pitest.org/

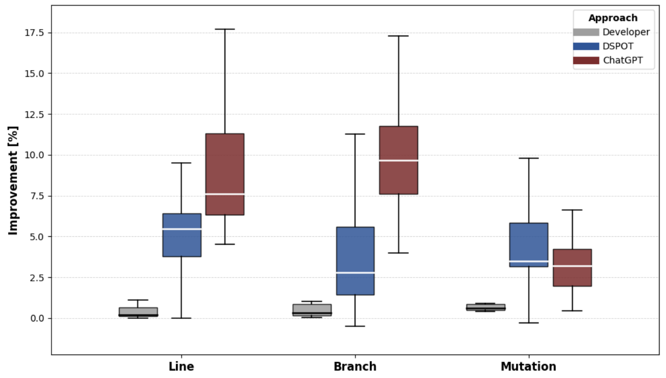

Fig. 1: Coverage improved by ChatGPT, DSPOT, and developer-amplified tests

mean gain. Developer and ChatGPT improve mutation scores for all 117 tests, with ChatGPT outperforming developers (3.20% vs. 0.90% mean gain). DSpot achieves the highest mean gain (4.44%) and outperforms ChatGPT in 99 tests.

3) Discussion: All 117 developer-amplified tests improve line (mean 0.74%), branch (mean 0.74%), and mutation (mean 0.90%) coverage. These amplifications yield modest gains across all metrics, underscoring the challenges of manually enhancing tests for already well-tested code. In contrast, DSpot and ChatGPT amplifications achieve substantial gains across all metrics, with a mean increase exceeding 3.20%, indicating that these approaches are effective for amplifying real-world tests and may reduce manual effort.

ChatGPT outperforms DSpot in improving line and branch coverage across all 117 tests, indicating its effectiveness in broadening test coverage by generating diverse inputs that explore program behaviors. However, DSpot surpasses ChatGPT in improving mutation scores in 99 tests, suggesting that its systematic test code transformations, specifically targeting mutation scores, are effective. These results highlight that combining the strengths of both approaches could lead to more comprehensive and effective test amplification.

***RQ2**:* ChatGPT outperforms both developers and DSpot in improving code coverage across all 117 tests, yielding mean line and branch coverage gains of 8.91% and 9.74%, respectively. However, DSpot surpasses both ChatGPT and developers in mutation score, with a mean gain of 4.44%.

D. Threats to Validity

1) Internal validity: We manually analyzed code changes in the developer-amplified tests to identify test amplification strategies. Manual analysis is subject to errors and biases; to mitigate this, two authors peer-reviewed the changes following a closed coding approach. While ChatGPT may have recited code from the developer-amplified tests, the significant improvements achieved by ChatGPT-amplified tests across all metrics compared to developer-amplified tests indicate that ChatGPT is not merely reciting existing code.

2) External validity: This study was conducted using 117 amplified tests from five open-source Java projects, which may limit the generalizability of the findings to projects using other languages or closed-source contexts. Replication on larger and more diverse datasets would increase confidence in these results. Although we used the advanced tools, DSpot and ChatGPT, outcomes may vary with different tools or LLMs.

3) Construct validity: We consider amplification as modifying existing tests to isolate its impact on pre-existing test behaviors. However, amplification could also involve adding new tests, which may affect amplification strategies. While code coverage and mutation metrics are widely used to evaluate test effectiveness [3], [5], they do not capture other important quality aspects of tests, such as readability or maintainability.

III. Related Work

Test amplification can be performed by adding new tests or modifying existing ones [11]. In this study, we focus on amplification through modifications to existing tests to isolate the impact on pre-existing test behaviors and better understand how developers enhance tests when the focal method under test and test structure are already established. This enables us to examine the strategies developers use when incrementally improving coverage without introducing entirely new test cases. To the best of our knowledge, no prior work has analyzed real-world amplified tests to understand developers' amplification strategies.

To assess how test amplification tools compare to manual amplification, we selected DSpot [3], a state-of-the-art technique that uses input space exploration [15] and assertion amplification [16] to amplify existing JUnit tests. Similar techniques have also been introduced for other programming languages [17], [18]. However, although these tools have been evaluated on real-world tests, their outputs have not been directly compared against manually amplified tests. While code coverage and mutation scores are key effectiveness metrics, they may not fully capture the qualities that influence developer adoption. In this context, DSpot has been used in several studies to investigate developer perceptions of automatically amplified tests [5], [6], [19]. Our dataset enables a direct, objective comparison between tool-generated and manually amplified tests.

While LLMs have shown promise in improving unit tests at the class level [7], [20], they have neither been applied to amplifying individual existing tests nor compared with developer- or DSpot-amplified tests.

IV. Conclusion

We created a dataset of 117 developer-amplified tests from five open-source Java projects. Our analysis showed that developers often amplify tests by reinvoking existing methods with different inputs, including overloaded methods in 21 cases. Amplifications also introduced new focal method calls in 78 tests. We further assessed the effectiveness of three approaches, developers, DSpot, and ChatGPT, in amplifying tests. ChatGPT outperformed both developers and DSpot in improving line and branch coverage, while DSpot surpassed developers and ChatGPT in improving mutation scores. These findings indicate that automated approaches substantially outperform manual amplification, with ChatGPT excelling at coverage and DSpot at fault detection. The complementary strengths observed suggest that combining these approaches could lead to more effective test amplification.

References

[1] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, 2017.

[2] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 435–445.

[3] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with dspot: a study with ten mature open-source projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2603–2635, 2019.

[4] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–11.

[5] C. Brandt and A. Zaidman, "Developer-centric test amplification: The interplay between automatic generation human exploration," *Empirical Software Engineering*, vol. 27, no. 4, p. 96, 2022.

[6] C. Brandt, A. Khatami, M. Wessel, and A. Zaidman, "Shaken, not stirred: How developers like their amplified tests," *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1264–1280, 2024.

[7] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2023.

[8] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[9] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[10] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[11] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[12] M. Williams and T. Moser, "The art of coding and thematic exploration in qualitative research," *International Management Review*, vol. 15, no. 1, pp. 45–55, 2019.

[13] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, p. 170, 2022.

[14] M. F. Roslan, J. M. Rojas, and P. McMinn, "An empirical comparison of evosuite and dspot for improving developer-written test suites with respect to mutation score," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 19–34.

[15] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[16] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *European conference on object-oriented programming*. Springer, 2006, pp. 380–403.

[17] M. Abdi, H. Rocha, S. Demeyer, and A. Bergel, "Small-amp: Test amplification in a dynamically typed language," *Empirical Software Engineering*, vol. 27, no. 6, p. 128, 2022.

[18] E. Schoofs, M. Abdi, and S. Demeyer, "Ampyfier: Test amplification in python," *Journal of Software: Evolution and Process*, vol. 34, no. 11, p. e2490, 2022.

[19] C. Brandt, D. Wang, and A. Zaidman, "When to let the developer guide: Trade-offs between open and guided test amplification," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 231–241.

[20] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 185–196.