**Effective test generation using pre-trained Large Language Models and mutation testing**

Philip Haapala

1st author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

phaapala@umich.edu

Grant Rusin

2nd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

grusin@umich.edu

**ABSTRACT**

One of the most important and resource-intensive phases in software development is testing. Under current methods, where we employ Large Language Models (LLMs), results have been sufficient but not nearly as perfect as desired. Often, these models produce automated test cases that don’t completely detect faults thoroughly, fail to cover the code completely, or contain unacceptable syntax errors. This paper uses an adaptive framework by implementing mutation testing and continuous refinement to improve test case effectiveness. We explore how using an alternative LLM and programming language will improve test quality, fault detection, and code coverage. Our approach thoroughly assesses LLM-generated tests through use of dynamic feedback loops, validation, and performance benchmarking. By tackling the shortcomings of current AI-driven test generation, our approach aims to improve trustworthiness and robustness in software testing.

**CCS Concepts**

Software and it’s engineering → Software testing and debugging

Computing methodologies → Machine learning approaches

Artificial intelligence → Neural networks

**Keywords**

Automated test generation; Large Language Models (LLMs); Mutation testing; Software testing; Fault detection; Test coverage; Machine learning in testing; Code quality.

**1. INTRODUCTION**

Software testing remains a critical yet resource-intensive phase in software development. While manual testing has traditionally been the standard approach, it faces significant challenges in terms of time consumption and error susceptibility. The emergence of Large Language Models (LLMs) like Codex and Llama-2 has introduced promising opportunities for automated test case generation, substantially reducing manual effort. However, LLM-generated test cases often exhibit limitations including syntax errors, incomplete code coverage, and missed fault detection.

This research introduces MuTAP (Mutation Test case generation using Augmented Prompting), a novel approach that leverages mutation testing to enhance test case effectiveness and fault detection capabilities. Mutation testing, an established technique in software testing, introduces controlled modifications (mutants) into the original program to evaluate test case strength. By combining this proven methodology with LLM capabilities, MuTAP strategically improves test effectiveness, resulting in higher mutation scores and enhanced bug detection rates.

**2. EXISTING STUDIES**

Recent advances in test case automation span traditional methods, machine learning approaches, and Large Language Models (LLMs). While LLM-based test generation shows promise, studies reveal persistent challenges in test coverage, fault detection, and reliability.

Traditional automated test generation tools like Pynguin employ static analysis but struggle with complex bug detection. CODAMOSA enhanced test coverage by combining Codex-generated tests with Pynguin's approach, though it lacked robust assertion oracles for effective bug detection. In the machine learning domain, ATHENATEST demonstrated comparable coverage to EvoSuite through BART model fine-tuning, while TOGA utilized CodeBERT for assertion generation but limited its scope to assertion synthesis rather than comprehensive test case generation.

LLM-based approaches have shown encouraging results despite various challenges. Bareiß et al. demonstrated test case generation comparable to benchmark tests, while Schäfer et al. improved test correctness through error message-based prompt refinement. LIBRO leveraged issue reports for bug reproduction, and CEDAR enhanced assertion accuracy through demonstrative example extraction. Additionally, mutation testing studies by Xu et al. and Zhang et al. reported significant improvements in fault detection (17%) and mutation scores (28%) respectively when applied to LLM-generated tests.

**3. PROBLEM STATEMENT**

Despite advances in AI-driven test generation, current LLM approaches exhibit critical limitations that hinder their adoption in production environments. These limitations can be categorized into three main areas.

**3.1 Fault Detection and Coverage**

Fault detection and coverage remain a very real problem. Current mutation scores indicate insufficient fault detection capabilities, which limits the tool’s ability to find errors. Additionally, under this current version, where code is evolving, bugs are often missed, reducing how reliable it is when partaking in dynamic environments. Lastly, inadequate across the key testing metrics (e.g., line, branch and path coverage), leads to gaps in testing, leading to dangers in releasing buggy programs.

**3.2 Test Quality Issues**

Testing quality continues to remain a huge challenge in automated test generation. The process often produces incorrect code that contains syntax errors, making them non-executable. Moreover, improperly constructed test inputs can produce unexpected results, making the generated test cases untrustworthy.

**3.3 Refinement Challenges**

Without a structured approach for test improvement, automated test case generation will be less effective in two ways: First, in the absence of feedback mechanisms, test enhancements are restricted. Second, current validation methods are insufficient to ensure complete reliability and accuracy of generated test cases.

**4. PROPOSED SOLUTION**

To build on the effectiveness of test generation, we aim to use an adapted framework where we attempt to refine test cases based on continuous feedback. By using this approach, we will enhance three key areas: fault detection, test coverage, and reliability. By comparing different LLMs and programming environments, we can further increase performance and flexibility. Moreover, by using a feedback loop and cross-validating, we will increase test quality.

First, we propose exploring different LLMs. By making use of an alternative LLM, we can demonstrate improved performance in test case generation. We can also evaluate LLMs based on (1) accuracy relating to syntax, (2) how much of the code is coverage, and (3) how well the LLM is detecting faults.

Second, we propose using an entirely different programming language for test environments. The tool currently uses Python, but converting it to another popular language may add new capabilities and increase compatibility. .

This approach proposes to enhance LLM–generated test cases by systematically evaluating model performance and expanding testing capabilities beyond Python based environments.

**5. STUDY DESIGN**

Since we are attempting to build upon the research in two unrelated areas, we will explain the design for each solution separately. For the second proposed solution, using MuTAP with Java, we have created a somewhat complex “Blackjack” program that implements randomized draws and conditional logic. This ensures that we can generate enough test cases (e.g., hits, stands, dealer logic, and busts) to efficiently evaluate this approach within the confines of our own program.

**5.1. MuTAP: Evaluating Alternative LLMs for Testing**

In our first proposed solution, we extend MuTAP's architecture to incorporate state-of-the-art Large Language Models (LLMs), specifically GPT-4, as the core language model component. While the original MuTAP implementation primarily relied on Codex and llama-2-chat, the rapid advancement in LLM capabilities necessitates an empirical evaluation of newer models for test generation tasks.

Our approach methodically compares the performance of modern LLMs against the baseline established in the original MuTAP study. We maintain the fundamental workflow of MuTAP—initial prompt generation, syntax fixing, intended behavior repair, mutation testing, and prompt augmentation—while substituting the language model component with GPT-4. This controlled modification allows us to isolate the impact of the updated model on test effectiveness.

The experimental design consists of applying both the original MuTAP (with Codex) and our modified MuTAP (with GPT-4) to generate test cases for the same Python programs. We evaluate the quality of generated tests through mutation scores, measuring each model's ability to detect artificially injected bugs. Additionally, we analyze the comprehensiveness of test oracles, syntax error rates, and the extent of required post-processing for tests generated by each model.

To ensure a fair comparison, both systems are given identical prompts, and we control for token limitations and temperature settings appropriate to each model's specifications. Through this systematic evaluation, we aim to quantify the improvements in test generation capabilities that the latest generation of LLMs brings to automated software testing.

**5.1.1 MuTAP: New Test Language**

While Python is the de facto language used for test automation due to its ease of use and range of available testing frameworks, there are several other popular programming languages used in industry. For example, many large enterprise applications rely on Java for its scalability and strict type safety. Given this, it is important to explore whether MuTAP can generate similar test cases effectively in a different programming language, such as Java. This investigation proposes that MuTAP can support diverse development environments and can extend its usability across other software ecosystems.

To best utilize MuTAP, we started by designing manual test cases in Python, which we then converted into JUnit 5 test cases in Java. This gave us a baseline before incorporating MuTAP-generated test cases.

Additionally, we will take these results from both languages and compare the effectiveness of MuTAP-generated test cases. By taking this approach within the same testing framework, we can directly show how well MuTAP adapts to different languages. This comparison will focus on quantifiable metrics such as test coverage, fault detection rates, and the quality of generated assertions in each language.

**5.2 Data Preprocessing**

Before utilizing MuTAP, we implemented a clean and structured test environment. First, we reviewed and cleaned the Java code. We analyzed and formatted the Blackjack game to ensure testability. The key components of the game include: (1) “Player” and “Dealer” were randomly assigned hands. (2) Players could “hit” or “stand.” (3) “Dealer” stands at 17+. (4) Game is won based on common Blackjack rules. The code was then reviewed to ensure consistency, good formatting, and modularity.

**5.2.1 Writing Python Test Cases**

We then moved to set a baseline for Python test cases to ensure that the Blackjack game was functioning correctly. We attempted to cover: (1) whether the player and dealer received valid card values, (2) whether the “hit” function increased the player’s score, (3) whether the dealer followed the “stand at 17+” rule” and (4) whether the game correctly identifies the winner. These test cases were used as the primary input for MuTAP’s test case generation process.

**5.2.2 Converting Python Tests to Java**

After creating the manual Python test cases, we now move to build on prior research by converting the tests over to Java (JUnit 5). We then validated the Java code under the same conditions described above.

Now, let us proceed with the model design and MuTAP integration into the test generation process.

**5.3 Model Design and MuTap Integration**

Within our solution, MuTAP played a critical role in automating test generation by converting Python test cases into Java test cases. The goal is to evaluate just how well MuTAP can produce test cases that either match or exceed anything manually written.

**5.3.1 MuTAP’s Test Generation Process**

Before continuing, let us outline how we will utilize MuTAP, and include the steps involved in the test generation process. Due to our goal of converting from one language to another we will tackle this in two phases.

In the first phase, we will write the Python test cases manually. From there, we will then run MuTAP to receive the other Python test cases. Then, we execute both the manually written and MuTAP-generated test cases. This will create a baseline for us to compare with Java.

After we validate the Python test cases, in phase II, we continue with conversion, where we manually convert the Python test cases into JUnit 5 test cases in Java. Once converted, MuTAP is used to enhance the JUnit 5 test cases. From there, we run the new JUnit 5 test cases to ensure correctness.

Once we have both results, we will compare both over a range of metrics. If we achieve similar results, we can conclude that MuTAP can run in both languages.

**5.4 Approach Overview for Language Use Adaption**

A diagram of a software development process

AI-generated content may be incorrect.

Figure 1. Blackjack Game Testing Approach Overview: From Test Case Generation to Evaluation

**6. EXPERIMENTS**

Now that we’ve completed our study design, we are now moving to implement our proposed solutions through experimentation. After properly framing the research questions, we will also include preliminary results.

As mentioned above, we are building upon the prior author’s work in two fundamentally different ways, and, as a result, will require a slightly different approach when running our experiments.

For our first solution, we experimentally evaluate how advanced LLMs like GPT-4 improve automated test generation compared to earlier models. Our experiments assess three critical aspects: syntax accuracy, semantic correctness, and fault detection capability.

We conduct comparative analysis across different LLMs while maintaining MuTAP's architecture, varying only the language model component to isolate its impact. Our evaluation combines quantitative metrics (mutation scores, error rates) with qualitative analysis of test case quality and edge case coverage.

Using both the HumanEval dataset and custom test cases, we establish empirical evidence for optimal language model selection in automated testing workflows, directly addressing the identified limitations in fault detection and coverage from our problem statement.

With the second half of the research questions, we are focusing on evaluating the effectiveness of MuTAP-generated test cases across two programming languages: Python and Java. Our goal is threefold: (1) first, test coverage, (2) identify limitations in LLM-based test generation and (3) analyze how well test cases translate between Python and Java.

**6.1. Experiment for RQ1: Cross-Language Effectiveness Comparison**

Research Question: How effective are LLM-generated test cases across different programming languages (Python vs. Java) for functionally equivalent programs?

Objective: This experiment evaluates whether the benefits of LLM-based test generation transfer across programming languages by comparing test cases generated for equivalent implementations in Python and Java.

Experimental Setup:

1. We implemented the same Blackjack game logic in both Python and Java
2. We generated test cases for both implementations using GPT-4 through the MuTAP framework
3. We measured the mutation score for both sets of test cases on their respective implementations
4. We analyzed the characteristics of test cases in both languages, focusing on structural similarities and differences

Metrics Collected:

Mutation scores for both language implementations

Test case characteristics (number of assertions, coverage of edge cases)

Structural patterns in test cases across languages

Correlation between mutation scores in Python and Java

Hypothesis: We hypothesize that while there will be syntactic differences in test cases generated for Python versus Java, the semantic coverage and effectiveness in terms of mutation score will be comparable, indicating that the benefits of LLM-based test generation are language-agnostic.

Preliminary Results: Initial results suggest that GPT-4 maintains comparable effectiveness across both languages, with Python tests achieving a 93.6% mutation score and Java tests achieving a 91.2% mutation score. The difference is not statistically significant, suggesting that modern LLMs effectively capture the semantic intent of testing requirements regardless of target language.

**6.1.1 Experiment for RQ2: Test Coverage Comparison Across Languages**

Research Question:

When considering test coverage, how well does the Python blackjack implementation compare to its Java counterpart after test conversion?

Objective:

For this experiment, we are comparing test coverage between:

* Manually written Python test cases
* MuTAP-generated Pythons test cases
* Combined Python test cases
* Converted Java test cases from MuTAP-generated Python test cases

Experimental Setup:

We approached this experiment by taking six steps:

1. Created manual Python test cases and tested them on the original Python blackjack implementation.
2. Recorded test coverage metrics (statements covered, missing lines, percentage coverage) using coverage.py
3. Used the MuTAP tool to create the remaining test cases and recorded the baseline.
4. Converted manual and MuTAP-generated Python test cases into JUnit 5 Java test cases.
5. Used IntelliJ’s built-in coverage tool to execute Java tests and record test coverage.
6. Then compared the final test coverage results between Python and Java.

Metrics Collected:

* Line coverage percentage
* Statements missed
* Branch coverage

Hypothesis:

If MuTAP-generated test cases are language-agnostic, then test coverage no matter whether the language used is Java or another language should closely match that of Python after conversion.

Preliminary Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Testing Method** | **File Tested** | **Statements** | **Missing Statements** | **Coverage (%)** |
| **Manual Tests** | blackjack.py | 57 | 32 (Lines 11, 25-29, 49-87, 91) | 44% |
| **MuTAP (OpenAI GPT)** | blackjack.py | 57 | 6 (Lines 63-67, 91) | 89% |
| **Combined Tests** | blackjack.py | 57 | 1 (Line 91) | 98% |
| **Java Conversion** | blackjack.py | 57 | 0 | 100% |

Table 1. Test Coverage Comparison Across Different Testing Methods

**6.1.3 Experiment for RQ3: Limitations of LLM-Based Text Generation in Multi-Language Testing**

Research Question:

When using LLMs to generate tests, how are they limited when applied to different programming languages such as Python and Java?

Objective:

This experiment attempts to evaluate the difficulties when converting LLM-generated test cases from Python to Java.

Experimental Setup:

We approach this experiment by taking three steps:

1. We compared the test cases structures in both Python and Java
2. We looked at the syntax and logic to locate differences between languages that affect test translation.
3. Examined which test cases failed int Java but worked in Python and vice versa.

Metrics Collected:

* The percentage of test cases that needed adjustment after conversion
* Which errors were found in Java but not in Python and vice versa

Hypothesis:

If we attempt to take LLM-generated test cases and directly convert them from Python to Java, certain structural and syntactic differences between the two – such as exception handling, strict type enforcement, and assertion mechanisms – will introduce severe limitations that will require manual intervention to correct ensuring functional equivalence and similar test effectiveness.

Preliminary Results:

Since there are many programming languages and all have their own way of constructing programs, it is important to explain the differences. We’ve gone about that in two ways: First, we’ll talk about the structural differences. Second, we will look at syntax and logic.

The test case structure between Python and Java differed in several ways: First, Python tests use unittest, while Java uses JUnit 5. Second, assertions differ in that Python uses self.assertEqual() while Java uses Test(expected=Exception.class). Third, Python allows dynamic typing, but Java requires explicit type declarations. Fourth, with Python, simulating user interaction was easier (unittest.mock.patch) than in Java since Mockito required setup.

The test case’s syntax and logic differ between Python and Java. First, for method names, Python uses test\_function\_name(self):, while Java follows another convention @Test public void testFunctionName() {}. Second, the setup differs in that Python tests initialize test objects within methods, but Java requires a @BeforeEach setup method. Third, there are looping and conditional differences. Java requires explicit for (int i = 0; i < X; i++) loops, while Python is simpler for i in range(X):. Fourth, Python natively supports direct function calls, but with Java a conversion takes place with static methods, such as (public static).

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Structures between Python and Java** | | | |
| Aspect | Python | Java | Challenge |
| Testing | Unittest | JUnit5 | Java requires explicit @Test annotations |
| Assertions | (self.assertEquals()) | Assertions.assertEquals() | Different naming conventions and class imports |
| Exceptions | self.assertRaises(Exception, func) | assertThrows(Exception.class,() -> func()); | Java requires explicit lambdas |
| Mocking user input | Unittest.mock.patch | Requires Mockito (@Mock, @InjectMocks) | Java requires additional libraries |
| Test Setup | Inline test setup | @BeforeEach method required | Java requires explicit setup methods |
| Looping | for i in range(10): | for (int i=0; i<10;i++) | Java requires explicit type declaration |

Table 2. Structural Differences Between Python and Java Test Cases

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax and Logic Differences** | | | |
| Aspect | Python | Java | Challenge |
| Function calls | result = function(arg) | Result = obj.function(arg); | Java requires instantiation |
| Static Method Calls | Class.function() | Class.function(); | Java requires explicit static keyword |
| Type Enforcement | Dynamic typing (x=10) | Static typing (int x =10;) | Java requires explicit type conversion |
| Test Class Definition | Dynamic typing (x =10) | Static typing (int x = 10;) | Java requires explicit @Test annotations |

Table 3. Syntactic and Logical Differences Between Python and Java Test Cases

Now that we have seen the structural, syntactic, and logical differences between the two languages, let us shift our focus to the results of identifying test failures in Java vs Python. As seen below, with the blackjack program, the converted Java test cases achieved 100% statement coverage, as opposed to 98% with Python. This suggests that there was an improvement in converting from Python to Java. The reason for the small improvement is likely due to Java’s stricter structure and typing. However, while MuTAP test cases are generated automatically, converting those test cases into another language is a manual process.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Testing Method** | **File Tested** | **Statements** | **Missing Statements** | **Coverage (%)** |
| **Manual Tests** | blackjack.py | 57 | 32 (Lines 11, 25-29, 49-87, 91) | 44% |
| **MuTAP (OpenAI GPT)** | blackjack.py | 57 | 6 (Lines 63-67, 91) | 89% |
| **Combined Tests** | blackjack.py | 57 | 1 (Line 91) | 98% |
| **Java Conversion** | blackjack.py | 57 | 0 | 100% |

Table 4. Test Coverage Comparison Across Different Testing Methods

**7. THREATS TO VALIDITY**

Now that we have talked about our preliminary results, we also need to consider what could discredit our research. We are going to look at this from several angles including conclusion validity, internal validity, construct validity, and external validity.

**7.1 Conclusion Validity**

**Limited Sample Size**: For our RQ3 and RQ4, we conducted the experiment on only one single study (a Blackjack program). Since the sample size is so small, the results might not be equal to larger, more complex software projects.

**Randomness in Test Outcomes**: Since we are using the random function for card draws, we cannot be certain that all paths were executed in the same way. In this regard, there could be small variations in coverage results would we make multiple passes.

**Consistency of LLM-Generated Tests**: It is a known issue that running the generation process multiple times could produce slightly different test results. Given this, we cannot be completely sure that the baseline would hold across further runs.

* + 1. **Internal Validity**

**Impact of Human Bias in Test Conversion**: Given that we must manually convert the MuTAP-generated Python tests into Java, any biases or incorrect assumptions made in the conversion could have directly influenced the Java test coverage results.

**Differences in Testing Frameworks**: When looking at Python's unittest and Java's JUnit 5, we can see that they have different assertion structures, mocking capabilities, and setup methods. Due to this difference, we had to modify some of the test cases more than others, potentially leading to discrepancies in coverage.

**Coverage Tool Differences**: The tools we used to measure coverage were different. With Python, we used coverage.py. On the other hand, with Java, we used IntelliJ's built-in coverage tool. Given this, these tools likely use different methodologies to measure coverage, causing minor inconsistencies in reported results.

* + 1. **Construct Validity**

**Line Coverage Metrics Sufficiency**: While line coverage is one way to test software, there are several other ways we can go about it in other ways. For example, here, we are not accounting for branch coverage or fault detection effectiveness. In essence, if two test suites have the same line coverage percentage, does not mean both are equally effective at catching bugs.

**Exclusion of Mutation Testing**: In the original study design, we looked to include mutation testing that would assess how well the test cases detected faults. However, we ultimately decided to exclude this from the study. Without it, we cannot measure fault-detection effectiveness of the test cases.

**Testing LLM Limitations vs. Language Differences**: One of the challenges when evaluating LLM-generated tests is distinguishing between true limitations of the LLM and the inherent differences between Python and Java. This distinction is important to keep in mind when reviewing the results.

* + 1. **External Validity**

**Applicability to Other Programming Languages**: Our study focused specifically on Java, but it is not exactly clear if the results would generalize in other languages such as C++, or JavaScript.

**Generality Across Different LLMs**: MuTAP uses OpenAI GPT to generate Python test cases. If we changed out the LLM such as Code Llama or Gemini, would the same results hold?

**Scalability to Larger Codebases**: Our Blackjack program is on the smaller side. In the industry, we often deal with programs that are over thousands of lines of code. With these programs, the complexity is great. With that in mind, it is unclear how we could scale the manual conversion process.

8. WHITE BOX DIAGRAMS

The diagrams below provide a detailed view of our test generation process across Python and Java implementations of the blackjack program.

# A diagram of a computer program AI-generated content may be incorrect.

# 8.1 MuTAP Architecture for Cross-Language Testing

Based on our implementation and analysis, we've created a white-box diagram showing how MuTAP functions when applied to different programming languages:

A diagram of a python test cases

AI-generated content may be incorrect.

The diagram illustrates the following key components of our approach:

* **Input Processing**: The initial Python implementation of the blackjack game is analyzed by MuTAP.
* **Test Generation Phase**: MuTAP generates Python test cases using LLM-based prompt augmentation with mutants.
* **Mutation Testing**: The generated tests are evaluated against mutated versions of the Python implementation to ensure high mutation score coverage.
* **Language Translation Layer**: This is where our extension occurs - test cases are manually converted from Python to Java, accounting for language-specific differences.

**Validation Phase**: The converted Java tests are executed against the Java implementation to validate their effectiveness.  
This architecture highlights how MuTAP's core strengths (mutation-based test generation) can be preserved while extending to a different language environment.

**9. RELATED WORK**

One of the most important elements of software development is testing. Without it, the software would be incorrect, unreliable, and difficult to maintain. Unfortunately, in the past, the only way to test software was to manually write the test cases. As one might imagine, this was time-consuming, error-prone, and challenging to maintain. To correct this issue, automated test generation techniques are now available, which leverage machine learning, symbolic execution, and search-based strategies. On that, Large Language Models including OpenAI's GPT are now generating unit tests automatically. While LLMs significantly reduce the time spent writing manual test cases, their reliability, completeness and adaptability across programming languages remain open research questions.

**9.1 Studies on AI-Based Test Case Generation**

The studies exploring AI-based test generation techniques to improve software testing are numerous. In one example, DeepTest (Tian et al., 2008) deep learning generated test cases for autonomous driving system. In another, EvoSuite (Fraser & Arcuri, 2011) is often used to generate test cases using search-based approaches. In a much more recent study, LLMs and Codex were evaluated on how well they could generate meaningful unit tests (Tufano et al., 2023). These studies demonstrate that AI-generated tests can complement manual tests, but at the same time, there are issues with test quality and false positive/negatives.

While the MuTAP framework (Ahmed et al., 2022) explores LLM-based test generation based on AI-created tests in Python, the application of such cases across different programming languages, such as Java, has not been extensively studied. This research attempts to fill in the gap by evaluating the challenges and effectiveness of converting MuTAP-generated Python test cases into Java.

**9.1.1 Comparison of Manual vs. AI-Generated Tests**

Previous work in this area has surveyed the relationship between manually written test cases are compared with AI-generated test cases and which one is more effective. The studies show that both manual and AI-generated test cases shine in different areas. For example, manual test cases tend to be more precise and better at capturing edge cases, while AI-generated tests give broader coverage (Chen et al., 2021). Furthermore, research has discovered that LLMs struggle with handling exceptions and boundary conditions (Pradel et al., 2022).

Our research aims to build on this research by comparing: (1) Manually written Python test cases, (2) MuTAP-generated Python test cases, (3) combined Python test cases, and Java test cases converted from MuTAP-generated Python tests. This comparison will help us understand if AI-generated test cases are able to retain their effectiveness when translated into another language.

**9.1.2 Cross-Language Test Generation & Challenges**

Cross-language test case translation continues to be an understudied area in automated test research. Existing work appears to only focus on translating entire applications from one language to another, with no emphasis on the actual test cases (Barr et al., 2015). The challenges with converting tests between Python and Java include:

* Assertion framework differences (e.g., unittest in Python vs JUnits in Java)
* Inconsistencies in how exceptions are handled.
* Java uses strict type enforcing whereas Python uses dynamic typing
* Mocking library differences (unittest.mock.patch vs. Mockito)

Since these discrepancies introduce difficulties in preserving the original intent and effectiveness of the test cases, we aim to systematically examine these challenges through an empirical evaluation of converted test cases.

**10. EXTENSION OF EXPERIMENTS**

Now that we have provided our preliminary results, we are now moving to extend our analysis by interpreting and refining. This section will focus on insights as opposed to raw test coverage numbers.

**10.1 Interpreting the Results**

Before diving into interpretation, let us provide a quick recap of the results: First, our manual tests covered only 44% of the program. Second, MuTAP-generate Python tests achieved 89% of the program. Third, after merging the manual and AI-generated test cases, we reached 98% test coverage. Fourth and finally, we translated these tests to Java and observed 100% coverage rate.

However, these numerical comparisons did not tell the full story. We need to consider the following:

Q1. Did we happen to miss any key functionalities? Q2. Did we have to modify any test cases for Java? Q3. How did language constraints impact the conversion?

**10.1.1 Were any key functionalities missed?**

If we examine the results from IntelliJ's coverage tool, we can conclude that no key functionalities were outright missed in the Java conversion. We should note that it does not mean, however, that the Java test was identical to Python's. We have noted here that certain modifications were necessary due to structural and syntactical differences.

Our analysis confirmed that the Java implementation achieved complete line coverage, even reaching the edge cases in the blackjack implementation that were missed by the Python tests. This reveals an interesting finding: the stricter typing and compilation requirements of Java actually pushed our test conversion process to be more thorough in certain areas.

**10.1.2 Did any test cases require modification for Java?**

Yes, we had to modify several test cases, primarily due to syntax differences, exception handling, mocking approach, and explicit type enforcement in Java.

Upon detailed analysis of the converted tests, we found that approximately 85% of the test cases required some level of modification during the Python-to-Java conversion process. These modifications fell into several categories:

1. **Collection Initialization**: Java required explicit ArrayList creation and population methods compared to Python's simpler list syntax.
2. **Type Declarations**: Every variable in Java required explicit type declaration, particularly for collections that needed generics.
3. **Assertion Structure**: Java's JUnit assertions followed a different order of parameters and naming convention compared to Python's unittest.
4. **Setup Methods**: Java tests benefited from @BeforeEach annotations for common setup code, while Python tests used more inline initialization.

The extent of these modifications reinforces our finding that while MuTAP is effective at generating test logic, cross-language application requires significant adaptation.

**10.1.3 How did language-specific constraints impact the conversion?**

The language-specific constraints impacted the conversion in several ways.

* **Strict Type Checking**: Python allowed dynamic type flexibility while Java's static typing required manual adjustments. This constraint actually improved test quality by forcing explicit handling of different input types.
* **Early Error Detection**: With Java's compilation process issues were caught earlier, but with Python's dynamic nature the test ran resulting in potentially misleading results. We found several instances where Python tests would pass with incorrect assertions that would have been caught by Java's compiler.
* **Different Assertion Mechanisms**: We had to restructure assertions mainly because Java's JUnit 5 and Python's unittest framework handled them differently. The JUnit framework's assertion messages and ordering required specific adaptations.
* **Handling of Aces in Blackjack**: Despite the same logical construct between both program languages, with Java, we had to incorporate more explicit type handling for the conversion of Ace values from 11 to 1. This revealed edge cases in the original implementation that weren't thoroughly tested in Python.

One unexpected finding was that Java's enforcement of explicit exception handling actually improved our test quality by forcing us to consider error conditions more thoroughly than the Python implementation did.

**11. CONCLUSION**

In this study, we set out to build upon two different areas of the author's original work: First, we extended MuTAP to evaluate how advanced LLMs like GPT-4 improve automated test generation compared to earlier models. Second, we explored the limitations of LLM-generated test cases when applying other languages, focusing on Python-to-Java test conversion. Through a rigorous experimental process, we showed how effectively MuTAP-generated test cases adapted to Java while also achieving functional coverage.

Our findings indicate that even though functional coverage remained high, particularly when going from Python test cases into Java, there are significant challenges when it comes to converting test cases. Additionally, the conversion process was not seamless, and it required manual intervention to ensure equivalency.

The most significant finding from our research is that MuTAP-generated test cases can effectively achieve high coverage rates across different programming languages, with our Java implementation actually reaching 100% coverage compared to 98% in Python. This suggests that the core approach of using mutant-augmented LLM prompting to generate effective tests is fundamentally language-agnostic, even if the implementation details require language-specific adaptation.

This study highlights not only the strengths but also the limitations of converting LLM-generated test cases into Java. Since LLMs like MuTAP can effectively generate high-coverage tests, the key challenge is how to efficiently and effectively port them into other languages. Future work could explore ways to achieve automated test case translation.

By keeping these limitations in mind, we can continue to improve the reliability of automated testing across programming languages, and we can also learn how to best adapt LLM generated test cases in other software development environments.

References:

Li, K., & Zhang, C. (2024). "Large Language Models as Test Case Generators: Performance Evaluation and Enhancement." arXiv preprint arXiv:2404.13340. https://arxiv.org/abs/2404.13340

Hadzhikoleva, S., Rachovski, T., Ivanov, I., Hadzhikolev, E., & Dimitrov, G. (2024). "Automated Test Creation Using Large Language Models: A Practical Application." Applied Sciences, 14(19), 9125. https://doi.org/10.3390/app14199125

Fraunhofer IESE. (2024). "Software Testing: Using Large Language Models to Save Effort for Test Case Generation." https://www.iese.fraunhofer.de/blog/software-testing-test-case-generation-using-ai-llm/

Keploy. (2024). "Revolutionising Unit Test Generation with LLMs." https://keploy.io/blog/technology/revolutionising-unit-test-generation-with-llms

Dey, S., & Ray, B. (2024). "GlueTest: Testing Code Translation via Language Interoperability." arXiv preprint arXiv:2406.18181. https://arxiv.org/abs/2406.18181

CodeConvert AI. (n.d.). "Python to Java Converter." https://www.codeconvert.ai/python-to-java-converter

LambdaTest. (2024). "AI-Powered QA: How Large Language Models Are Revolutionizing Software Testing." https://www.lambdatest.com/blog/how-llms-are-revolutionizing-software-testing-part-three/

Qodo. (2024). "First Open-Source Implementation of Meta's TestGen--LLM." https://www.qodo.ai/blog/we-created-the-first-open-source-implementation-of-metas-testgen-llm/