A Comprehensive Analysis of LLM-Based Test Case Generation

Nikhil Yates
Dept. of Electrical Engineering and Computer Science
York University
Toronto, Canada
nikhilyy@my.yorku.ca

***Abstract* — This paper outlines findings from the study on the comprehensive performance of LLM-based test case generation techniques. I use GPT-4 to generate tests for the TensorFlow codebase and compare the quality of tests against a set of KPIs and a traditional test generation tool: Pynguin.**

***Keyword* —** Large Language Models, Automated Test Case Generation, Neural Networks, Generative AI For Software Development

I. Introduction

Strong software is characterized by robustness, functionality, usability, maintainability, and security [1]. These characteristics are cultivated through solid design, implementation, and rigorous testing. The latter, though laborious and time-consuming, remains the best method for proving, monitoring, and enhancing code performance. Consequently, there has been a growing demand to automate this task to reduce the time (and money) spent by developers and engineers on writing tests for their code.

Today, many extensively studied techniques and tools exist to assist in generating test cases. Traditional tools, such as Pynguin, can easily detect program flaws. However, they often lack the readability and understandability of manually created tests. In contrast, large language models (LLMs) and generative AI have the capability to produce high-performing and human-like output, signaling a clear quality difference over traditional counterparts. Current research supporting the strong performance of LLM-based test case generation is empirical, leaving space for further exploration into the compounded quantitative and qualitative differences between traditional and generative test case generation approaches.

II. Background

A. Automating Code Generation and Test Case Generation

Research exists surrounding generative AI and its practical applications in code generation, code comment generation, and its use in code test generation. However, few studies have been published regarding the efficacy of test case generation using the GPT-4 large language model. This paper project serves as a stepping stone to further work in the field. The current research that explores the strength of other LLM's in test case generation is still effective, and I will build upon the work done by Gregory Gay (et. al) [2] and Max Shäfer (et. Al) [3] to guide my investigation. Specifically, I am to emphasize that random test generation outperforms coverage-directed generation.

B. Generative Pre-Trained Transformer 4

GPT-4 can effectively contextualize input thanks to its transformer architecture (great for handling sequential data/text input), large-scale language modelling, and probabilistic generation techniques. Paired with its capability to take criticism and revise input, we have an NLP powerhouse. Critics of LLM-based test case generation argue that the quality of its edge-case-inclusive tests is poor [4]. However, this claim was made for older versions of generative AI. GPT-4 *independently* stresses the importance of random test generation that includes non-typical use contexts.

C. Pynguin - PYthoN general UnIt test geNerator

Pynguin is a tool developed to generate unit tests automatically for python programs and codebases [4]. It employs a dynamic test generation technique that generates a basic test, runs it against the program, and iteratively changes the test to improve coverage. It uses *mutants* (faulty implantations) to improve tests. It is largely different than a LLM and has performance limitations most notable when testing complex codebases and heavily inter-dependent programs. For this experiment, I will be using the random generation approach for a more direct LLM comparison.

III. Experiment Design

A. Baseline

The objective of the experiment is to assess the quality of GPT-4 and directly compare its tests with the traditional tool. Hence, I follow a simple yet consistent method for creating and comparing test cases from each tool.

Generating Tests with GPT-4

1. *Initial input and prompt:* copy and paste the entire module of which the method of interest belongs to into the GPT-4. Prompt it to analyze the code.
2. *Test generation prompt:* instruct GPT-4 to "generate 3 test cases that can be run using pytest for the method **func**. Place each test case in the same file".
3. *Add test to the repo:* copy the generated code into a text editor and save it to the tests folder of the appropriate directory in the TensorFlow repository.

Generating Tests with Pynguin

1. *Terminal command:* use the following input to generate test cases:

```
Pynguin --project-path /path/to/file
--output-path /path/to/test/destination
--maximum-test-executions 50 --module-name
module_of_interest -v
```

--module-name is followed by the name of the file that contains the method(s) I want to test.

For both sets of tests, I will run them using *pytest-cov:*

```
pytest /path/to/test_file.py
pytest –cov=/path/to/module /tests_folder/
```

B. Dataset

I will use the TensorFlow GitHub repository for experimentation, specifically focusing on the Python packages and modules from the /autograph/pyct directory. The 50 Python methods selected for testing vary in type (i.e., instance, class, static) and access level (i.e., public, private, protected). This ensures a diverse collection of data, which will aid in obtaining comprehensive results. Factors such as edge cases, training data, a deeper understanding of the language and frameworks over time, increased opportunities through feedback loops, benchmarking, and validation, all contribute to the richness of the data.

C. Evaluation Metrics

As outlined, current research about the competence of LTG (Large Language Model Test Generation) is empirical. I seek to understand the quantitative *and* qualitative differences between LTG and TTG (Traditional Tool Test Generation) and will employ the following metrics to gauge performance:

Quantitative KPIs

1. *Coverage:* how much of the code being tested is *covered* by the test cases? Use the *pytest-cov* plugin during testing for an accurate estimation of code coverage.
2. *Pass %:* This is a delicate indicator that will not independently reflect upon the efficacy of the tests. Alone, this metric is virtually useless. However, with the coverage metric, this attribute brings more clarity about how well the test cases are built.
3. *Execution time:* Strong test cases should be effective and efficient. This means that they should *not* be CPU-intensive nor time-insensitive. It should be noted that again, this attribute is intricate. Alone, it does not accurately reflect the quality of a test – it must be taken into consideration alongside the other indicators.
4. How consistent is the test? Does it maintain its assertions over time? Does it change? Robust and *predictable* tests are important. In other words, the test results should never change over time. Employ a measurement called *flakeIndex* that assesses changes in test behaviour after 4 runs (0.25 points per run). The close the score is to 0, the better, with 1 being the worst score.

Overall, I will assign each test a quantitative score called *quantScore* which is out of 8 to track empirical effectiveness of each test.

Qualitative KPIs

1. *Clarity:* how simple are the tests? Are they over complicated where they shouldn't be?
2. *Readability:* how easy is it to understand what the tests are doing? Are they annotated? How appropriate are variable names and assertions?
3. *Realistic:* test cases need to cover random and unpredictable user input to account for all users. Does this test do this?
4. *Identifiable:* Out of a test case suite, how different is this one than the rest? Is it redundant? Does it perform too similarly to other tests?

Like the *quantScore* for the qualitative analysis of tests, I will assign each a *qualScore* to track its qualitative performance.

Tests will be judged based on their *comprehensive* scores across *both* categories of KPI's – aligning with the focus of the study.

D. Research Questions

To guide my evaluation on my experiment further, I will refer to a set of questions that encapsulate the premise of the project.

- **RQ1 (Performance): How well does the LLM-based test generation tool perform with respect to the criteria for strong test cases?**
- **RQ2 (Competence): To what extent do traditional and LLM-based test generation tools differ quantitatively? Qualitatively?**

IV. Result Analysis

The first, and arguably most critical observation I made during testing was Pynguin's lack of precision. I was unable to generate test cases for specific methods as I did during GPT-4 test generation. Pynguin-generated tests are coverage-oriented and can be heavily configured to meet specific developer *needs but* lack the ability of precision-testing that GPT4 has. As a result, I generated a fraction of the tests in Pynguin as I did in GPT4 to maintain the direct test-to-test comparisons I outlined previously.

TABLE I. Test Files Generated Per Tool

Tool	*Test Files Generated*	*Tests Generated*
Pynguin	5	33
GPT-4	50	150

Note that each Pynguin-generated file contained more than 3 tests – GPT-4 test files only contain 3.

RQ1: Performance

GPT-4 performs well across all quantitative categories - on average, a *quantScore* of 7.3 (figure 1). Medium-high to high code coverage, high pass%, competitive execution time, and a very low flakiness index were consistently observed through testing. This is a testament to the LLM's

strong NLP capabilities and the efficacy of its 4 transformers when analysing code.

As outlined, this project seeks to provide a solid understanding of the quantitative *and qualitative* performance of LLM-based test generation. The qualitative KPIs I outlined help achieve this. GPT-4 attained a strong *qualScore* of 7.6 (figure 2). Each test case was annotated thoroughly , and supporting documentation was provided within the web browser GPT-4 client for further clarification about what it produced and why. This earned the tool high scores for clarity and readability.

Moving on, relevance and realism of the tests can be interpreted in a few ways:

- Code coverage and use of relevant data: the test is able to evaluate the code in its entirety: in a manner similar to an actual use case that the code was intended for. This metric requires an actual understanding of the TensorFlow library and what classes and modules are used for.
- Variableness and Randomness: to mimic real world usage as much as possible, the test cases should be random and be able to simulate unpredictable input.

TABLE II. QUANTITATIVE ANALYSIS USING QUANT KPIS

Tool	**Quantitative Analysis**				
	Coverage	*Pass %*	*Time(s)*	*Flakiness*	***quantScore***
GPT4	76.4%	70%	6.2	0.0	7.3
Pynguin	64.2%	21%	5.7	0.1	5.2

Fig. 1. Average *quantScore*s corresponding to average component scores after runs on both GPT-4 and Pynguin generated tests

The LLM-based tests do both of these things well and GPT-4 emphasized the significance of these factors in each test it generated.

RQ2: Competence

Throughout experimentation, I observed a substantial difference in average *quantScores* between the LLM and traditional tool. Namely, pass percentage (figure 3).

TABLE III. QUALITATIVE ANALYSIS USING QUAL KPIS

Tool	**Qualitative Analysis**				
	Clarity	*Readability*	*Realism*	*ID'able*	***qualScore***
GPT4	2	1.7	1.9	2	7.6
Pynguin	0.9	1.1	1.5	1.2	4.7

Fig. 2. Average *qualScore*s corresponding to average component scores after runs on both GPT4 and Pynguin generated tests

As forementioned, this quantitative indicator should not be used to judge the effectiveness of the created test cases singlehandedly. However, it is interesting to note the substantial difference between the two methods. Coverage differences between the two tools are notable (figure 1) – especially with the large difference in pass %. This can be attributed to Pynguin's limited performance on TensorFlow's complex codebase and the strong dependencies between the modules in the *pyct* directory. This is a clear example of how GPT4's NLP processing capability sets it apart from the traditional tools that cannot contextualize code as coherently. GPT4 was able to assess the relevance new code with respect to previous inputs and create test cases that not only suited the current input but could assimilate within the context of the entire test generation session.

Both sets of tests executed in similar time – all within a range of $\delta < 0.51$s. This was expected. The quality of each tool is such that test cases should not consume an inappropriate number of resources for computation. Pynguin's slightly superior performance in this category may be indicative of its lower code coverage: quicker test execution -> lower coverage -> less intensive testing. Within the context of the average coverage and pass scores, I identify GPT4 to be the better *timely* performer.

Both sets of tests received low flakiness scores. However, it is critical to note that *any* flakiness is indicative of a flawed test. Pynguin-generated test cases that did perform differently over the 3 test iterations contributed to this number and reflect on the very strong yet noticeably imperfect precision of the tool. **GPT4 again outperforms the traditional tool and receives an overall *quantScore* of 7.4; Pynguin trailing with 5.2.**

Tool	*Pass Percentage %*	*Fail Percentage %*
GPT4	70%	30%
Pynguin	23%	77%

Fig. 3. Average test pass and fail percentages between GPT-4 and Pynguin

The qualitative attributes of TTG tests were considerably lower than the LTG tests. Specifically, readability and clarity. Out of the 33 test cases that were by Pynguin, none contained comments that could have described the tests – compared to every GPT-4 test case. As well, variable names made the tests unreadable and lacked any real notion of context that could help understand their usage. Pynguin averaged a low *qualScore* of 4.7 (figure 2) and is not near the level of comprehensiveness GPT-4 displays.

Overall, the LLM-based test cases perform incredibly well with respect to both sets of KPIs and can be considered a stronger alternative to the traditional tool, Pynguin.

V. DISCUSSION

This project focused on developing test cases for the python codebase in the TensorFlow repository. However, I also generated Java unit tests and evaluated them using the same quantitative and qualitative sets of KPIs in the python-based tests. This set of test cases performed comparably well with average *quantScores* and *qualScores* of 7.1 and 7.5 respectively. An additional metric that sets LLMs apart from traditional tools is its scalability and universal application. Not only was GPT-4 able to generate tests for specific methods in TensorFlow Java classes, but it also generated *high-quality* tests. Looking forward, I would like to perform further research on the universal application of LLM-based test generation: specifically, its effectiveness across popular languages compared to existing, non-LLM-based tools.

VI. Example Tests

GPT-4

```
import pytest
from tensorflow.python.autograph.pyct.cache import
_TransformedFnCache

class MockTransformedFnCache(_TransformedFnCache):
        # Test Case 1: Retrieve an item that exists
        # in the cache
        def  get key(self, entity):
            return entity
        # Simplified key generation for testing

        #Test Case 2: Retrieve an item that does
        #not exist in the cache
        def test_retrieve_nonexistent_item(self):
            cache = MockTransformedFnCache()
            test_entity = "test_entity"
            assert cache[test_entity] == {}, "Should
        initialize an empty entry for a new entity"
        # Test Case 3: Retrieve an item and ensure
        # it is modifiable
        def test_retrieve_and_modify_item(self):
            cache = MockTransformedFnCache()
            test_entity = "test_entity"
            cache[test_entity]["new_subkey"] =
                "new_value"
            assert cache[test_entity]["new_subkey"]
                == "new_value", "Should be able to
                modify the retrieved item"
    # If you wish to run the tests directly using
    # this script
    if __name__ == "__main__":
        pytest.main()
```

Fig. 4. GPT-4 gnerated test case example for the '_TransformedFnCache method in cache.py

Pynguin

```
    # Test cases automatically generated by Pynguin
(https://www.pynguin.eu).
    # Please check them before you use them.
    import pytest
    import
tensorflow.python.autograph.pyct.inspect_utils as
module_0
    import inspect as module_1
    import tokenize as module_2
    import ast as module_3
    def test_case_0():
        none_type_0 = None
        var_0 = module_0.isconstructor(none_type_0)
        assert var_0 is False

    @pytest.mark.xfail(strict=True)
    def test_case_1():
        none_type_0 = None
        var_0 = module_0.islambda(none_type_0)
        module_0.getqualifiedname(none_type_0,
none_type_0, none_type_0)
    def test_case_2():
        none_type_0 = None
        var_0 = module_1.istraceback(none_type_0)
        var_1 = module_0.isnamedtuple(var_0)
        assert var_1 is False
    @pytest.mark.xfail(strict=True)
    def test_case_3():
        bool_0 = True
        var_0 = module_0.isbuiltin(bool_0)
        assert var_0 is True
        var_1 = module_2.group()
        module_0.getnamespace(var_1)
    @pytest.mark.xfail(strict=True)
    def test_case_4():
        none_type_0 = None
        var_0 =
module_0.getfutureimports(none_type_0)
        var_1 = module_0.isbuiltin(var_0)
        assert var_1 is False
        var_2 = module_0.isnamedtuple(none_type_0)
        assert var_2 is False
        module_0.getqualifiedname(none_type_0,
none_type_0, none_type_0, none_type_0)
    @pytest.mark.xfail(strict=True)
    def test_case_5():
        bool_0 = False
        var_0 = module_0.isconstructor(bool_0)
        assert var_0 is False
        var_1 = module_0.islambda(bool_0)
        module_0.getimmediatesource(bool_0)
    @pytest.mark.xfail(strict=True)
    def test_case_6():
        str_0 = "Resolves the type/value an external
(e.g. closure, global) variable.\n\n    Args:\n
ns: namespace\n      types_ns: types namespace\n
name: symbol name\n    Returns:\n      Tuple (type,
static_value). The first element is the type to use
for\n      inferrence. The second is the static
value to use. Return None to treat it\n      as
unknown.\n    "
        module_0.getqualifiedname(str_0, str_0,
visited=str_0)
    @pytest.mark.xfail(strict=True)
    def test_case_7():
        none_type_0 = None
        var_0 =
module_3.get_source_segment(none_type_0,
none_type_0)
```

```
        module_0.getmethodclass(var_0)
    @pytest.mark.xfail(strict=True)
    def test_case_8():
        int_0 = 3
        var_0 = module_0.islambda(int_0)
        var_1 = var_0.__gt__(var_0)
        var_2 = module_0.getfutureimports(int_0)
     var_0.visit_Dict(var_2)
```

Fig. 5. Pynguin generated test case for the module inspect_utils

VII. References

[1] Choudhary, B. (2023, July 12). *Top software quality characteristics: What makes a good software application?*. Finoit Technologies. https://www.finoit.com/articles/software-quality-characteristics/

[2] Gay, G., Staats, M., Whalen, M., & Heimdahl, M. P. (2015). The risks of coverage-directed Test case generation. *IEEE Transactions on Software Engineering, 41(8),* 803-819. https://doi.org/10.1109/tse.2015.2421011

[3] Shäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023) An empirical evaluation of using large language models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering,* 1-21. https://doi.org/10.1109/tse.2023.3334955

[4] *Pynguin.* PyPI. (n.d.). https://pypi.org/project/pynguin/.