# PyTestPilot: Large Language Models for Automated Unit Test Generation for PyPi Packages

Khandaker Rifah Tasnia
Concordia University
Montreal, QC, Canada
khandakerrifah.tasnia@mail.concordia.ca

Soumit Kanti Saha
Concordia University
Montreal, QC, Canada
s_soumit@live.concordia.ca

## Abstract

Unit testing is essential for building reliable software, but it's often overlooked due to the time and effort involved. Traditional test generation techniques, including static and dynamic analysis, typically focus on coverage but fall short when it comes to producing readable and meaningful test assertions. In this study, we explore how Large Language Models (LLMs), specifically GPT-4o, can help automate unit test generation for Python packages. Building on the TestPilot framework originally designed for JavaScript, we adapted and extended it into a Python-based framework, which we call PyTestPilot. This system includes components for exploring APIs, mining documentation, generating effective prompts, and validating generated tests. We applied our framework to five widely-used PyPI packages—emoji, pyfiglet, pytz, shortuuid, and yarl and generated unit tests under different prompt strategies. We then evaluated the tests based on coverage, assertion quality, and their similarity to manually written tests. Our findings show that LLM-generated tests can achieve substantial statement and branch coverage for PyPI packages, offering a promising approach to automating unit test generation. This work provides practical guidance for developers and researchers interested in leveraging LLMs to reduce the burden of writing unit tests.

## Keywords

Automated Software Testing, Large Language Models (LLMs), Prompt Engineering, Unit Test Generation

**ACM Reference Format:**
Khandaker Rifah Tasnia and Soumit Kanti Saha. 2025. PyTestPilot: Large Language Models for Automated Unit Test Generation for PyPi Packages. In . ACM, New York, NY, USA, 7 pages.

## 1 Introduction

Unit testing is a critical practice in the software development lifecycle, used to verify the correctness of individual components in isolation. Effective unit tests not only support continuous integration and safe refactoring but also contribute to long-term software maintainability and reliability. Despite its recognized importance, writing unit tests is often viewed as tedious and time-consuming, especially in fast-paced development environments. As a result, many developers either write minimal tests or skip testing altogether, leading to low coverage and increased technical debt[10][7].

To address this challenge, researchers have proposed automated unit test generation techniques based on static and dynamic analysis. Tools such as Randoop[13] and EvoSuite[5] automatically generate test inputs that achieve high code coverage. However, these methods often struggle to produce meaningful and readable assertions, resulting in tests that are difficult to interpret or maintain. Moreover, such techniques are limited by their dependence on structural heuristics and lack of contextual understanding.

Recent advances in Large Language Models (LLMs)—notably OpenAI's GPT-3 and GPT-4—have opened new avenues in automated code generation. Trained on large-scale corpora of source code and natural language, LLMs exhibit strong capabilities in code generation, documentation, and synthesis tasks[3], [1]. When applied to unit test generation, these models offer the potential to generate natural-looking, context-aware test cases with more realistic assertions. Early work by Schäfer et al. [14] introduced TestPilot, a framework for LLM-based test generation in JavaScript. Their findings demonstrated that with carefully crafted prompts, LLMs can generate unit tests that resemble human-written ones in both structure and quality, while achieving meaningful coverage.

However, it remains unclear how well this approach generalizes to other programming languages. Python, in particular, is a dominant language in scripting, data science, and backend development. Its dynamic typing, flexible syntax, and extensive ecosystem introduce unique challenges for automated test generation. Moreover, the influence of prompt engineering strategies on LLM-generated test quality in Python remains underexplored. Most prior work has focused on code generation, with limited attention to evaluating test quality in depth.

This study extends the work of Schäfer et al. [14] by adapting the *TestPilot* framework for Python and developing **PyTestPilot**, a Python-based test generation framework built on top of GPT-4o. We implemented the key components of the TestPilot pipeline in Python for our PyTestPilot—such as the API explorer, documentation miner, prompt generator and refiner, and test validator. Our study is designed to assess the applicability, quality, and limitations of LLM-generated tests in Python projects and to understand how different prompt strategies impact the results.

To validate our approach, we generated over 11,000 unit tests across five popular Python packages—emoji, pyfiglet, pytz, shortuuid, and yarl. After applying automated validation to filter out incorrect tests, 7,751 valid tests were retained. These achieved a median statement coverage of 83.16% and a median branch coverage of 74.19%. These results highlight the capability of LLMs to produce high-coverage with realistic assertions, especially when guided by prompt strategies that include full function bodies.

The following research questions guide our work:

**RQ1: How effective are LLMs in generating unit tests for Python packages?**

We evaluate the generated tests in terms of statement and branch coverage and their usefulness in real-world contexts.

**RQ2: How does the quality of LLM-generated Python tests compare to manually written tests?**

This includes an analysis of assertion accuracy, completeness, and readability.

**RQ3: How do different prompt strategies impact Python test generation?**

We investigate the effect of variations in prompt content (e.g., function body, examples, docstrings) on test quality.

To answer these questions, we selected five popular Python packages—emoji, pyfiglet, pytz, shortuuid, and yarl and generated unit tests using GPT-4o under multiple prompt conditions. The preliminary evaluations focus on branch and statement coverage. This study offers early insights into how LLMs can be leveraged to assist developers in generating high-quality, automated unit tests for Python projects and beyond.

## 2 Related Works

Automated unit test generation is an active research area due to time consuming and tedious characteristic of manual unit test generation. Different techniques such as fuzzing [17], feedback-directed random test generation, dynamic symbolic execution , and search-based and evolutionary techniques. Most of these techniques explore the flow of the program statically or dynamically to increase the coverage. But these techniques lacks readability and understandability of the written tests. These approaches makes the test generation task a mathematical problem.

There are some recent efforts to automate generation of unit tests [4], [12], [16], [6], [15], [14]. But there are very limited number of research on automated unit test generation for Python. And most of the other study are on Java, JavaScript and other languages. For instance, the study [6] enhances test generation by exploiting polyglot and temperature controlled diversity. Authors used a curated benchmark EvalPlus [9] to generate 1 test case for 5 languages and 5 test cases for 1 language. Then they merged the test cases to get the test suite.

TICODER [8] and CODET [2] both use Codex to generate code and test cases from natural language problem descriptions. TICODER follows a test-driven user-intent formalization (TDUIF) loop, where the user collaborates with the model to iteratively produce an implementation that aligns with their intent, along with corresponding test cases to ensure correctness. In contrast, CODET automatically generates multiple candidate implementations and test cases from the same prompt, executes the tests on each candidate, and selects the best-performing solution based on the results. But none of them actually solves the problem of generating unit tests for existing codes.

The study CEDAR [11] provides a retrieval based few shot example prompt construction strategy. This study does not evaluate multiple prompt techniques.

The study [15], TestPilot, provides an approach to generate unit tests for npm packages and provides an evaluation result of the approach on 25 npm packages.

As best of our knowledge there is no existing study on generating unit test automatically with high statement and branch coverage for existing pypi packages. Our study incorporates multiple prompt techniques to generate unit tests for all the public methods of pypi packages, validating them and correcting the assertion errors. We have incorporated the approach of TestPilot [15] for this study.

## 3 Methodology

This section outlines the design of our study, detailing the dataset, system architecture, and prompt engineering strategies used in developing *PyTestPilot*, a framework for automated unit test generation in Python using LLMs. We explain how each component of the system works together to generate and refine test cases based on different types of contextual input.

### 3.1 Dataset Selection

To evaluate the generalizability and effectiveness of LLM-generated unit tests, we selected five widely-used Python packages from the PyPI repository that represent diverse functional domains and moderate code complexity. The selected packages are:

- Emoji (v2.14.1) – Unicode emoji processing
- PyFiglet (v1.0.2) – ASCII art generator
- Pytz (v2025.1) – Timezone definitions
- Shortuuid (v1.0.13) – Short UUID generator
- Yarl (v1.20.0) – URL parsing and manipulation

These packages were chosen because they are self-contained, have clear documentation, and expose public APIs suitable for unit-level testing.

### 3.2 System Architecture

We implemented the core components of the original TestPilot framework [14] and extended it into our *PyTestPilot*, a Python-based unit test generation framework powered by GPT-4o. Our system architecture (shown in Figure 1) is designed to efficiently extract and process relevant information from Python packages. The API Explorer and Documentation Miner work in parallel to gather both the structural API data and the associated documentation, which are then fed into the Prompt Generator to create detailed prompts for test generation. The generated tests are validated and, if necessary, refined to ensure their correctness, completing the test generation pipeline.

#### 3.2.1 *API Explorer.*

This component is responsible for identifying and extracting the public interfaces of the selected package. It inspects the package to gather information about public functions, classes, and methods. The API Explorer recursively explores submodules and retrieves detailed information, including function signatures, docstrings, and source code. This information is saved in a structured JSON file, capturing details such as - function/method signatures, parameter, version, return description, docstrings, source code, examples and related metadeta.

#### 3.2.2 *Documentation Miner.*

The Documentation Miner utilizes the JSON file generated by the API Explorer. This JSON file contains

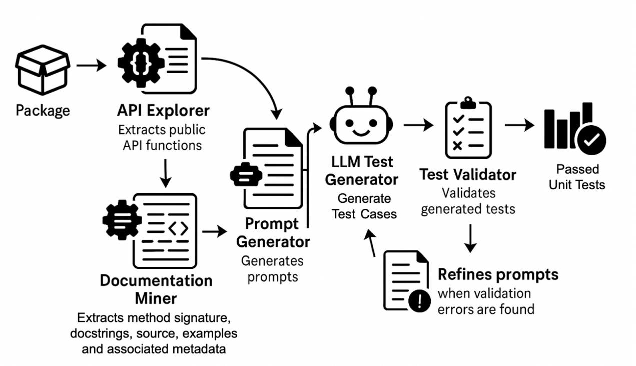

Package
API Explorer
Extracts public API functions
Documentation Miner
Extracts method signature, docstrings, source, examples and associated metadata
Prompt Generator
Generates prompts
LLM Test Generator
Generate Test Cases
Test Validator
Validates generated tests
Passed Unit Tests
Refines prompts
when validation errors are found

**Figure 1: System architecture showing the flow of the test generation process.**

structured data about the public APIs, including function signatures, docstrings, and associated metadata. The Documentation Miner parses this JSON file to extract relevant information for test generation. Key elements extracted include:

- Function/Method Signatures: Parameters, return types, and default values.
- Docstrings: Descriptions of parameters, return values, exceptions, and examples.
- Source Code: Full source code for the function or method.
- Usage Example: Any usage example of that particular function.

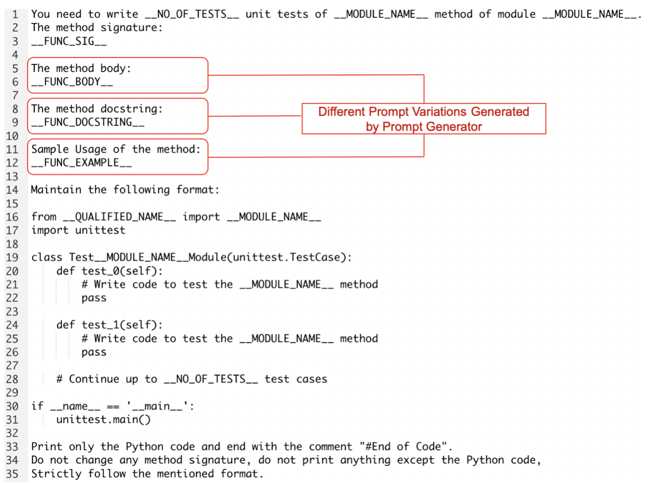
By processing the API Explorer and Documentation Miner concurrently, we ensure that both the structure and the detailed documentation of the package are efficiently processed and ready for the next steps in the test generation pipeline.

*3.2.3 Prompt Generator.* The Prompt Generator is responsible for creating prompts that guide the LLM in generating accurate unit tests for identified functions and methods. It uses data from the Documentation Miner to create several prompt variations, each with different levels of detail. The goal is to provide the LLM with enough context to generate high-quality tests while avoiding unnecessary complexity.

We start with a Base Prompt, which includes only the method's signature, as we found that adding too much information at once can cause the LLM to hallucinate, leading to worse results rather than improvements. By starting with minimal context, we can better control the quality of the generated tests. The Prompt Generator creates four types of prompts:

- **Base Prompt**: Contains only the method's signature, asking the LLM to generate tests based on minimal context.
- **Prompt with Function Body**: Provides the LLM with the full source code of the method for more detailed test generation.
- **Prompt with Function Example**: Includes usage examples (if available) from the method's docstring, providing LLM concrete examples of how the function is used in practice.
- **Prompt with Function Docstring**: Includes the method's docstring, which provides a description of the function's purpose, parameters, return values, and usage.

Each prompt variation is designed to help the LLM generate high-quality unit tests with different amounts of context. As shown in Figure 2, these variations are visualized, demonstrating how the placeholders in the prompts are replaced with actual data.


```
You need to write __NO_OF_TESTS__ unit tests of __MODULE_NAME__ method of module __MODULE_NAME__.
The method signature:
__FUNC_SIG__

The method body:
__FUNC_BODY__

The method docstring:
__FUNC_DOCSTRING__

Sample Usage of the method:
__FUNC_EXAMPLE__

Maintain the following format:

from __QUALIFIED_NAME__ import __MODULE_NAME__
import unittest

class Test__MODULE_NAME__Module(unittest.TestCase):
    def test_0(self):
        # Write code to test the __MODULE_NAME__ method
        pass

    def test_1(self):
        # Write code to test the __MODULE_NAME__ method
        pass

    # Continue up to __NO_OF_TESTS__ test cases

if __name__ == '__main__':
    unittest.main()

Print only the Python code and end with the comment "#End of Code".
Do not change any method signature, do not print anything except the Python code,
Strictly follow the mentioned format.
```

**Figure 2: Different prompt variations(Base prompt, Prompt with function body/ example(s)/ docstring) generated by the Prompt Generator.**

*3.2.4 LLM Test Generator.* Once the prompts are generated, they are passed to the LLM Test Generator, which is responsible for generating the actual unit tests. The LLM Test Generator sends the prompts to the LLM (in our case, GPT-4), instructing it to produce a specified number of unit tests for the target method. The generated tests are stored for subsequent validation.

*3.2.5 Test Validator.* The Test Validator ensures that the unit tests generated by the LLM Test Generator are correct and executable. The Test Validator receives the generated test file from the LLM Test Generator and then it runs the code using Python's unittest framework. It checks for syntax errors, assertion failures, and other runtime issues.

If no errors are detected and the test passes, the validation is considered successful and moves on to the next step. However, if errors are found, the Test Validator logs the issues and triggers the Prompt Refiner to generate a new refined prompt that addresses the detected errors. The refinement process iterates until the generated tests pass all checks.

*3.2.6 Prompt Refiner.* The Prompt Refiner is responsible for iteratively improving the unit tests generated by the LLM Test Generator when validation errors are detected. If the Test Validator identifies issues in the generated tests, such as assertion failures or other errors, the Prompt Refiner generates a refined prompt that includes the original test code along with the error details. A sample of the refined_prompt format used by the Prompt Refiner is shown in Figure 3.

The refined prompt instructs the LLM to fix the identified errors without altering the method signatures or introducing new functions. The prompt is then sent back to the LLM, and a new version of the test code is generated. This process continues iteratively, with the Prompt Refiner refining the prompt and generating new test code until the test passes or the maximum number of iterations is reached.

```
__TEST_CODE__

The unittest code to test __QUALIFIED_NAME__ has following error(s):
__TEST_ERROR__

Fix the error. (Note that the __QUALIFIED_NAME__ function exits in __MODULE_NAME__.
So, do not add any new function to the code on your own. Try to fix assertion errors.
Our main target is to write correct unittest code for __QUALIFIED_NAME__)
Print only the Python code and end with the comment "#End of Code".
Do not change any method signature, do not print anything except the Python code,
Strictly follow the mentioned format.
```

**Figure 3: Sample refined prompt format used in the Prompt Refiner.**

To summarize, the *PyTestPilot* framework brings together the key components—API Explorer, Documentation Miner, Prompt Generator, Test Generator, Test Validator, and Prompt Refiner—to automate unit test generation for Python using LLMs. By applying different prompt strategies and refining tests when errors occur, our method ensures that the generated tests are not only syntactically correct but also meaningful and executable. This pipeline allows us to carefully study how different types and levels of context provided in prompts affect the quality of the tests, and to better understand the strengths and limitations of LLMs in real-world Python testing scenarios.

## 4 Experiment Setup

We evaluated our automated test generation pipeline across five Python packages: *emoji, pyfiglet, pytz, shortuuid, and yarl.* For each package, we extracted all public functions and methods using our API Explorer and generated unit tests using GPT-4o via the OpenAI API.

All prompts were executed with the model's temperature set to 0.7 to introduce moderate variation while maintaining consistent outputs. A fixed system message was used to frame the LLM's role: "You are a professional Python developer and Quality Assurance Engineer." For each function, we generated five unit tests per four prompt types: base prompt, prompt with function body, prompt with docstring, and prompt with usage examples. This resulted in up to 20 test cases per function. The prompt-response pipeline managed the token budget dynamically, ensuring the generated responses did not exceed model limits. Each prompt was submitted via OpenAI's ChatCompletion API, with up to five retry attempts in case of transient errors or rate limits.

After that the generated test cases were validated using Python's unittest framework. If a test failed, the error message was passed to the Prompt Refiner, which reformulated the prompt to include the error context. The refinement process allowed for up to three iterations per test, resulting in a maximum of four attempts per prompt variant. Each refinement prompt and corresponding LLM response were saved for traceability.

To optimize resource usage, existing LLM responses were reused if already saved locally. All subprocess calls, including validation, prompt execution, and coverage analysis, were executed with a timeout of 100 seconds to prevent indefinite stalls.

After validation, all passing test methods were merged into a single consolidated file per package using Python's ast module. Only test methods that executed successfully (or included standard setup/teardown routines) were retained.

We evaluated the final merged test suites using pytest with coverage.py, generating reports for both statement and branch coverage in XML format.

All experiments were conducted on a local Python environment with API access to GPT-4o. This setup enabled systematic generation, refinement, and validation of LLM-generated unit tests at scale.

## 5 Results

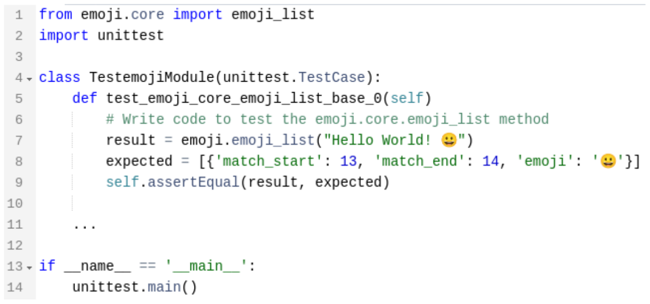
### 5.1 RQ1: How effective are LLMs in generating unit tests for Python packages?

Table 1 shows the number of correctly generated unit tests for each modules, statement and branch coverage achieved by those correctly generated tests. From table 1 we can see that we have generated total 7751 correct tests for the 5 pypi packages. We have achieved coverage ranging from 71.62% to 90.85% with a median 83.16% and branch coverage ranging from 60.31% to 90.48% with a median 74.19%.

We have run our test validator and prompt refiner multiple times (maximum 3 times) to fix the assertions errors. If any test that was not fixed by refined prompt for 3 times, we discarded that test. So, we actually generated more than 7751 tests, but we have not considered those tests due to assertion errors.

### 5.2 RQ2: How does the quality of LLM-generated Python tests compare to manually written tests?

To answer this question, we have manually analyzed the generated unit tests by **PyTestPilot**. We have seen that the quality of generated tests is almost similar to manually written tests. To compare we have considered the examples in documentation as manually written tests. But there are also some issues with generated tests.



```python
from emoji.core import emoji_list
import unittest

class TestemojiModule(unittest.TestCase):
    def test_emoji_core_emoji_list_base_0(self)
        # Write code to test the emoji.core.emoji_list method
        result = emoji.emoji_list("Hello World! 😀")
        expected = [{'match_start': 13, 'match_end': 14, 'emoji': '😀'}]
        self.assertEqual(result, expected)

    ...

if __name__ == '__main__':
    unittest.main()
```

**Figure 4: Incorrect module path during calling a method under test**

| Package Name | # Modules | # Tests | LOC | Statement Coverage | Branch Coverage |
|---|---|---|---|---|---|
| emoji | 2 | 630 | 354 | 87.40% | 73.38% |
| pyfiglet | 2 | 886 | 689 | 90.42% | 85.39% |
| pytz | 1 | 1159 | 629 | 75.52% | 61.39% |
| yarl | 1 | 1598 | 1177 | 71.62% | 60.31% |
| shortuuid | 1 | 3478 | 295 | 90.85% | 90.48% |
| | | 7751(total) | | 83.16%(avg) | 74.19%(avg) |

**Table 1: Test Results and Coverage for Python Packages**

In figure 4, we can see that the method that we want to test is $emoji\_list$ from module $emoji.core$. But in line 7,

$$result = emoji.emoji\_list(...)$$

, $emoji\_list$ method of called from $emoji$ module instead of $emoji.core$ module. This type of problem causes the unit test to fail.

By manual analysis we have found some causes for test failures:

- method calling from wrong module
- spelling mistake during module or method import or method calling
- incorrect assumption on expected output due to lack of context about method return object.

From table 2, we can see that with our approach we have generated 11k+ unit tests and 70% (on average) of those test cases were correct. We have considered only these 70% unit tests during calculation of statement and branch coverage. We have not considered the failing test cases because they are mostly assertion errors. So, manually written test cases are still better in quality that LLM generated test cases. But LLM generated test cases are not far behind.

## 5.3 RQ3: How do different prompt strategies impact Python test generation?

The table 3 shows the impact of Different Prompt Strategies on Statement and Branch Coverage for five Python packages: Emoji, PyFiglet, Pytz, ShortUUID, and Yarl. Based on the data in the table and the accompanying figures, we can observe how varying levels of context in the prompts affect the statement coverage and branch coverage in the generated unit tests.

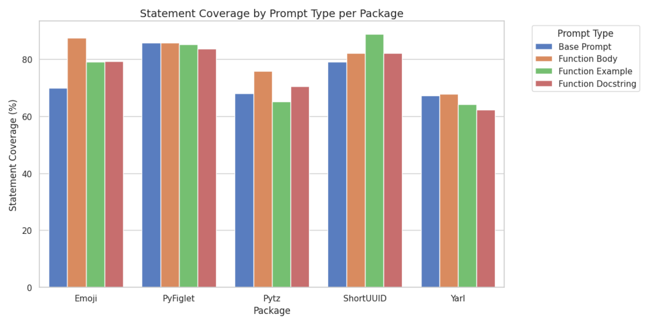


**Figure 5: Statement Coverage with different prompt technique**

*5.3.1 Impact on Statement Coverage.* As illustrated in Figure 5, the Prompt with Function Body strategy consistently results in the highest statement coverage across all packages. We can see that evident for Emoji, PyFiglet, Pytz and Yarl where Function Body leads to the highest coverage. For instance, Emoji shows about 87.40% statement coverage with the Function Body prompt (as shown in the table), which is the highest among all the prompt strategies. This suggests that providing more details about the function allows the model to generate better tests cover a broader range of statements within the code.

In comparison, the Base Prompt, which only includes the function signature, results in lower coverage. This is especially noticeable in packages like Pytz and Yarl, where statement coverage drops to around 67%. This shows that without enough context, the model struggles to generate comprehensive tests.

The Function Example and Function Docstring strategies perform better than the Base Prompt but not as well as the Function Body in most cases. Function Example generally performs better than Function Docstring, which suggests that giving the model examples of how the function works helps it generate more effective tests.

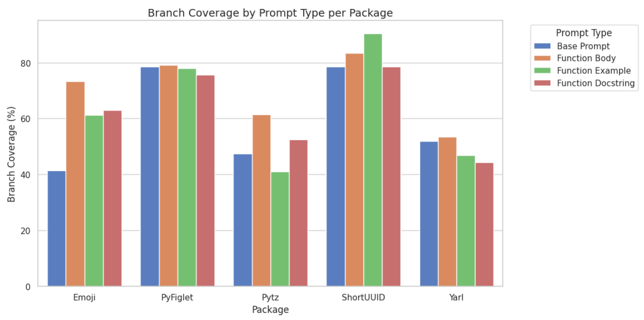


**Figure 6: Branch Coverage with different prompt technique**

*5.3.2 Impact on Branch Coverage.* The trends for branch coverage, shown in Figure 6, follow a similar pattern to statement coverage. In most cases, the Function Body strategy results in the highest branch coverage. For example, with ShortUUID, it reaches up to 90%, although the Function Example strategy slightly leads in this case. This indicates that providing the full function context helps the model generate tests that cover various branches in the code.

On the other hand, the Base Prompt results in the lowest branch coverage, especially for Emoji and Pytz, as shown in Figure 6. This

| Package Name | # Genereted Tests | # Correct Tests | # Incorrect Tests |
|---|---|---|---|
| emoji | 1138 | 630 (55.36%) | 508 |
| pyfiglet | 1890 | 886 (46.88%) | 1004 |
| pytz | 1830 | 1159 (63.33%) | 671 |
| yarl | 2072 | 1598 (77.12%) | 474 |
| shortuuid | 4114 | 3478 (84.54%) | 636 |
| total | 11044 | 7751 (70.18%) | |

**Table 2: Correct and Incorrect Generated Unit Tests**

| Package | # Tests | Prompt Type | Statement Coverage | Branch Coverage | # Tests/Prompts |
|---|---|---|---|---|---|
| Emoji | 630 | Base Prompt | 69.82% | 41.48% | 135 |
| | | Function Body | **87.40**% | **73.38**% | **178** |
| | | Function Example | 79.07% | 61.21% | 156 |
| | | Function Docstring | 79.23% | 62.91% | 161 |
| PyFiglet | 886 | Base Prompt | 85.70% | 78.52% | 211 |
| | | Function Body | **85.70**% | **79.05**% | **258** |
| | | Function Example | 85.19% | 77.99% | 210 |
| | | Function Docstring | 83.60% | 75.70% | 207 |
| Pytz | 1159 | Base Prompt | 67.89% | 47.52% | 272 |
| | | Function Body | **75.68**% | **61.39**% | **334** |
| | | Function Example | 65.01% | 41.09% | 265 |
| | | Function Docstring | 70.43% | 52.48% | 288 |
| ShortUUID | 3478 | Base Prompt | 78.98% | 78.57% | 854 |
| | | Function Body | 82.03% | 83.33% | 879 |
| | | Function Example | **88.81**% | **90.48**% | 863 |
| | | Function Docstring | 82.03% | 78.57% | **882** |
| Yarl | 1598 | Base Prompt | 67.12% | 51.95% | 395 |
| | | Function Body | **67.80**% | **53.50**% | **409** |
| | | Function Example | 64.06% | 46.89% | 392 |
| | | Function Docstring | 62.28% | 44.36% | 402 |

**Table 3: Evaluation of Test Generation Performance by Package and Prompt Strategy**

confirms that without enough information, the model can't effectively cover all branches in the code.

The Function Example strategy typically outperforms the Function Docstring strategy in terms of branch coverage, similar to the statement coverage results. The Function Example strategy helps the model better understand the branching logic by providing concrete examples, as seen in PyFiglet and Shortuuid where it achieves better coverage than Function Docstring.

*5.3.3 Number of Tests Generated.* The Function Body strategy leads to the highest number of successful tests generated for most of the packages. This is likely due to the richness of the function context, which enables the model to generate more comprehensive test cases. In contrast, the Base Prompt strategy results in fewer tests, aligning with its lower coverage metrics. Function Example and Function Docstring generate a moderate number of tests, with Function Example typically generating more tests than Function Docstring due to its more concrete context.

In conclusion, the evaluation shows that different prompt strategies significantly impact test generation. The Function Body strategy provides the best results in terms of both statement and branch coverage, showing that giving the model full function details helps it generate better tests. The Base Prompt, which only gives the function signature, leads to the worst results, with lower coverage and fewer tests. Function Example and Function Docstring are better than the Base Prompt but not as good as the Function Body strategy. Overall, this highlights that providing richer, more detailed prompts allows LLMs to generate more accurate and comprehensive unit tests, which is essential for ensuring the quality of software testing.

## 6 Threat to validity

### 6.1 Internal Validity

Documentation miner extracts method signature, examples, docstring of each method from the source code. It searches for >>> line prefix inside docstring and considers that line as example. Sometimes there might be example that does not have that line prefix. Those examples are not extracted by documentation miner. After function body we observe (fig 6, fig 5) that prompt with function example produces good coverage. This indicates that this threat is not a serious limiting factor.

The examples in RQ1 result suggests that variable names in the generated unit tests are actually same as a developer would use.

So, user studies to asses the readability of the generated unit test would be an interesting future work.

## 6.2 External Validity

This study evaluates our approach on 5 pypi packages showing a high statement and branch coverage, other pypi packages might produce different outcomes. Also, the packages we selected for this study might have been used in training the LLM we used in this study (GPT-4o).

## 7 Conclusion

Our study presents PyTestPilot that levarages the pypi package's source code itself to generate unit tests automatically. PyTestPilot does not require any fine tuning or any parallel corpus of existing unit tests. This tool is not limited to these 5 pypi packages, in fact it can be used for any pypi packages. This tool provides four different prompt techniques to enrich the generated test suits achieving high coverage. It also provides an additional refined prompt technique to fix the previously generated incorrect unit tests. It leverages the source code to mine docstring, usage example. In future we will extend our study with combination of multiple prompt techniques to achieve better coverage.

After generating a vast test suite with 11k+ unit tests, we considered only the correct ones (7.7k+ unit tests) to report coverage. But we have found many cases through manual analysis that are actually not assertion errors. So, in future we will try to fix those other errors to enrich our generated test suite further.

Another interesting direction would be using fuzzing, feedback-directed random test generation combining with LLM directed unit test generation. For instance, LLM generated seed can be used to feedback-directed fuzzing to uncover further edge cases that we could not cover with PyTestPilot.

## 8 Artifacts

Our PyTestPilot's code can be found here.

## References

[1] Jacob Austin et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

[2] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

[3] Mark Chen et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[4] Christoph Csallner and Yannis Smaragdakis. 2004. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34, 11, 1025–1050.

[5] 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*, 416–419.

[6] Djamel Eddine Khelladi, Charly Reux, and Mathieu Acher. 2025. Unify and triumph: polyglot, diverse, and self-consistent generation of unit tests with llms. *arXiv preprint arXiv:2503.16144*.

[7] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40, 7, 633–649.

[8] Shuvendu K Lahiri et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*.

[9] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 21558–21572.

[10] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[11] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.

[12] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2111–2123.

[13] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 815–816.

[14] 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*, 50, 1, 85–105.

[15] 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*, 313–322.

[16] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.

[17] Michal Zalewski. 2017. American fuzzy lop. (2017).