# Readability Comparison of Generated Test Suites

Akshaya Bathula

## I. Introduction

This report compares the readability of test files generated by Pynguin (original) against three refactored versions produced by different models: DeepSeek, Gemini, and LLaMA. Examined test suites for three modules (`queue_example`, `string_utils.validation`, and `httpie.sessions`), focusing on how each version improves (or not) the clarity of assertions, code structure, naming conventions, and use of comments. The goal is to illustrate which approach yields the most human-readable and maintainable tests, highlighting improvements over the original auto-generated tests.

## II. Original Files

Three files were processed:

- `test_queue_example.py`
- `test_httpie_sessions.py`
- `test_string_utils_validation.py`

## III. LLM Automation

### A. Scripts used:

- `deepseek.py`
- `gemini.py`
- `meta.py`

Each applies a structured prompt to rewrite and enhance test suites.

### B. Prompt Used

*Improve the following Python test file by:*

- *Refactoring test structure and naming*
- *Adding edge cases and comments*
- *Ensuring readability and clear assertions*

The simple prompt is used to yield better results. A simple prompt may or may not be used for a large test suite. Should be changed accordingly. For better readability and understandability by the model, I have chosen to generate this simple style prompt.

## IV. Execution

1) Insert your OpenRouter API key
2) Run: `python deepseek.py`, `gemini.py`, `meta.py`
3) Outputs saved in model-specific folders

## V. Overall Summary and Ranking

All three refactored versions (DeepSeek, Gemini, and LLaMA) improved the readability of the test suites compared to the original Pynguin output, but to very different extents. Taking into account structure, naming, clarity of assertions, and helpful documentation, I rank the versions as follows:

### A. DeepSeek – Most Readable Overall

DeepSeek's test suites are well-structured (often using multiple classes to organize tests by feature or scenario), with descriptive names and minimal unnecessary complexity. Each test tends to focus on one concept, which makes the code easier to understand. The addition of docstrings and comments is just right — enough to clarify intent without overwhelming the reader. DeepSeek also introduced sensible edge-case tests, enhancing clarity about how the code should behave in unusual scenarios. Across all modules, DeepSeek's output is consistent and highly maintainable.

### B. Gemini – Close Second

Gemini's refactorings also feature clear naming and thorough documentation. In some cases (such as `string_utils.validation`), Gemini preserved more of the original test combinations, which leads to a few very large test methods. These can be dense, but Gemini offsets this with extensive comments and docstrings explaining each part. Gemini's use of helper constants and grouping of expected failures into separate tests shows attention to readability and completeness. While just slightly more verbose or complex in places than DeepSeek's version, Gemini's test suites are still very reader-friendly and well-organized.

### C. LLaMA – Somewhat Improved

LLaMA's refactored tests add basic documentation and make small naming tweaks, which provide some help, but they do not substantially reorganize or simplify the original tests. Many of the structural issues from the Pynguin output (like multiple concerns in one test, and non-intuitive variable naming) remain. Inconsistencies and minor errors in the refactoring further reduce clarity. LLaMA's versions demonstrate the importance of the more in-depth restructuring that DeepSeek and Gemini performed — without it, tests remain somewhat hard to read even if comments are added.

TABLE I
COMPARISON OF READABILITY ASPECTS ACROSS TEST GENERATORS

| Aspect | Original | DeepSeek | Gemini | LLaMA |
|---|---|---|---|---|
| **Assertions** | Verbose, multi-line asserts (e.g., f-string type checks); not clearly explaining intent. | Clear and direct asserts (e.g., `assert not queue.full()`) instead of indirect checks. | Clear asserts with straightforward comparisons (often uses direct equality and boolean checks). | Mostly direct asserts similar to original structure (some still use f-string type comparisons). |
| **Structure** | Flat sequence of `test_case_0...N` functions with no grouping; tests appear in a somewhat arbitrary order. | Organized into a `TestQueue` class grouping related tests; each test targets a specific behavior or edge case. | Organized into a `TestQueue` class; each test method covers one aspect (e.g., initialization, enqueue, etc.). | Remains as standalone functions (`test_case_0`, `test_case_1`, ...); no class grouping. |
| **Naming** | Non-descriptive names (`test_case_0`, `int_0`, `bool_0`) that reveal nothing about purpose. | Descriptive method names (e.g., `test_queue_initialization`) and clear variable names (`queue_size`, `item`). | Descriptive names (e.g., `test_queue_initialization`, `test_invalid_size`) and meaningful variable names (`size`, `element1`). | Partially improved: still uses generic function names (`test_case_n`); variable names remain generic (`queue_0`, `int_0`). |
| **Comments/Docs** | None (no comments or docstrings explaining tests). | Each test method has a docstring and inline comments (e.g., filling the queue to capacity); the class is documented. | Includes class-level and test-level docstrings, plus top-of-file comment block describing overall coverage. | Minimal improvement: lacks docstrings or top-level descriptions. |

TABLE II
COMPARISON OF TEST FILES FOR `STRING_UTILS_VALIDATION` MODULE

| Aspect | Original | DeepSeek | Gemini | LLaMA |
|---|---|---|---|---|
| **Assertions** | Many asserts chained in single tests; some comparing unrelated constants and built-ins; uses generated values like random strings, making intent unclear. | Simplified to focus each test on one function's return value. Assertions clearly check `True`/`False` or equality. Constants verified separately. | Generally clear and logically grouped. Verifies constants alongside results. A few blocks combine multiple asserts for completeness. | Assertions are present but some are incorrect or confusing (e.g., referencing `module_0`). Concerns are not well separated. |
| **Structure** | Flat list of `test_case_0...N` functions. Some (e.g., `test_case_9`) marked xfail, mix unrelated checks. | Still flat functions, but each focused on a single validation function. Grouped by naming. Edge cases like empty strings included. | Uses a `TestStringUtilsValidation` class. Some methods combine related checks. Structure retains logic of original while improving readability. | Flat functions like original. Some improvement in function names, but structure remains muddled. Mixed multiple checks still occur. |
| **Naming** | Uninformative names (`test_case_0`, `str_0`, etc.). No context of function under test. | Descriptive function names (e.g., `test_is_palindrome_with_empty_string`, `test_is_palindrome_with_valid_input`). Semantic variable names. | Clear method names within class (e.g., `test_is_string_10_with_invalid_input`). Uses meaningful constants and variables. | Mixed: some improved names (e.g., `test_is_valid_number`), but others remain generic. Inconsistencies like reusing `module_0` confuse context. |
| **Comments/Docs** | None, aside from a multi-line string used as input (not an actual comment). | Each test includes a brief inline comment. No top-level docstring, but naming helps clarify intent. | Class docstring summarizes suite. Each method has docstrings explaining test purpose. Constants grouped at top with comments. | Minimal documentation. No docstrings or summary. Code intent is unclear in places. |

### *D. Conclusion*

Refactoring the Pynguin-generated test suites yielded markedly more readable and maintainable tests. The original tests, while functional, were not reviewer-friendly: they lacked narrative structure, had cryptic naming, and offered no insight into what was being verified. In contrast, the DeepSeek and Gemini versions show how thoughtful grouping of tests, clear naming conventions, and concise documentation can transform an automated test suite into something approaching human-written quality. Test functions are now self-describing, and each suite is organized logically, allowing a developer to quickly understand the purpose and outcome of each test. Even the less comprehensive LLaMA refactoring demonstrates some benefits of adding comments and better names, though it stops short of a full cleanup.

Overall, the readability improvements across these refactored tests highlight the value of maintaining clarity in test code. A more readable test suite makes it easier to trust the tests, extend them, and catch issues, ultimately improving the software development process. The DeepSeek

TABLE III
COMPARISON OF TEST FILES FOR `HTTPIE_SESSIONS` MODULE

| Aspect | Original | DeepSeek | Gemini | LLaMA |
|---|---|---|---|---|
| **Assertions** | Direct but lengthy dictionary comparisons; some tests lack assertions. Overuses constants in every test. | Clean assertions; defines expected outputs as variables. Redundant checks are minimized. Focused outcomes per test. | Similar to DeepSeek; maintains expected session dicts as constants. xfail tests are structured and complete. | Assertions mostly copied from original; some are incorrect or confusing due to refactoring issues. |
| **Structure** | Flat functions (`test_case_0...8`), non-sequential, hard to follow. xfail usage is unclear. | Organized into multiple logical classes (e.g. `TestSessionInitialization`). Clear functional separation. | Single class `TestHTTPSession` with logically grouped test methods. Dedicated xfail section. | Flat structure retained. Some docstrings added, but tests remain mixed and loosely structured. |
| **Naming** | Uninformative numeric test names. Variables like `str_0`, `session_0`. | Descriptive class and method names. Clear variable naming (e.g. `request_headers`). | Clear method names and constants for repeated values. Variables are meaningful and descriptive. | Function names remain generic (`test_case_n`). Inconsistencies like undefined aliases confuse naming. |
| **Comments/Docs** | No comments or docstrings. Expected behavior unclear. | Docstrings for every class and method. Inline comments explain key logic. Very well documented. | Class and method docstrings provided. Constants documented at top. xfail tests are separated clearly. | Some docstrings added as pseudo-names. Overall limited improvement in clarity or documentation. |

model in particular produced the most straightforward and well-structured tests, showing the greatest contrast to the original. However, all three refactored versions underscore the important takeaway: auto-generated tests greatly benefit from a post-generation pass focused on human readability, turning them from opaque verification scripts into clear, communicative examples of intended behavior.

In conclusion, I would like to say that different LLMs have different advantages, we can flexibly use the LLM that our test suite suits well to yield the best results. This research has a few limitations that I would like to mention: As it is just a basic technical report, the LLMs used here are free versions available in openrouter; the paid version may perform better than these versions. There is always a better option when compared to these models, as this is just a basic study instead of extensive research.

I would also like to mention the reason why I chose more models instead of a single approach; As I already mentioned, these versions are free, and they may have a few limitations. I want to overcome them by comparing more models instead of a single model. I also personally felt like single multi-model analysis would be helpful in research instead of a single model analysis. This thorough process is made for this report to obtain the optimal results and provide a comparative analysis for the top models available in the LLM market currently.

*Note: No actual validation of test cases generated by gemini, meta and deepseek is done. Purely used for comparative analysis*