Quality Evaluation of Large Language Models Generated Unit Tests: Influence of Structured Output

Dovydas Marius Zapkus, Asta Slotkienė

Vilnius University,
Universiteto g. 3, Vilnius
marius.zapkus@mif.stud.vu.lt

Summary. Unit testing is critical in software quality assurance, and large language models (LLMs) offer an approach to automate this process. This paper evaluates the quality of unit tests generated by large language models using structured output prompts. The research applied six LLMs in generating unit tests across different classes of cyclomatic complexity of C# focal methods. The experiment result shows that LLMs generated results according to a strict structure output (Arrange-Act-Assert pattern) that significantly influences the quality of the generated unit tests.

Keywords: large language model, unit test, quality metrics, structured prompt output

1 Introduction

Unit testing is crucial to ensuring the quality of software code units. Consequently, various automated unit test generation tools and large language models (LLMs) have demonstrated promising results and capabilities in unit test generation [1]. Nevertheless, LLM-based unit test generation encounters challenges in generating robust unit tests.

This paper investigates the quality of unit tests generated by LLMs (Gemini-2.0-flash, GPT-4o, GPT-4o-mini, Llama-3.3-70 b-versatile, Qwen-2.5-32b, Qwen-2.5-coder-32b) when prompted to produce output in a structured format. There have already been attempts to investigate the structures of the prompts, and it was shown to influence the reliability and accuracy of the output [3, 4, 6]. Our experiment showed that LLMs generated results according to a strict structure output (Arrange-Act-Assert pattern) that significantly influences the quality of the resulting tests.

DOI: https://doi.org/10.15388/LMITT.2025.32

2 Research Methodology

The research investigated the effectiveness of LLMs in generating unit tests by addressing three research questions (RQs). These research questions allow us to systematically evaluate the capabilities of six different LLMs in generating unit tests by considering key performance and quality metrics. The research questions are as follows.

- RQ1: How effective are the LLMs in unit test generation according to focal method complexity?
- RQ2: How effectively do the generated unit tests cover the focal method?
- RQ3: How robust are the generated unit tests?

Investigating RQ1, we try to analyze whether the cyclomatic complexity of the focal method under test impacts the different LLMs abilities to produce effective unit test cases, helping to understand the relationship between method complexity and test generation performance. The second research question (RQ2) will assess the capability of generated tests to fully cover code lines and code branches and show the carefulness of automatically generated test suites. Developing robust unit tests for various types of changes so that they can be retested with small changes and run across a series of system versions is significant for the improvement of the software development process [2]. This is the answer we are looking for with RQ3. To develop unit tests that are robust to various types of changes so that they can be executed with small changes and across a series of system versions

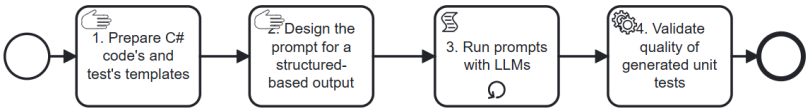

Figure 1. Research methodology diagram

The research utilized real-world projects implemented in the C# programming language from the GitHub repository. The research began with manual retrieval of C# code segments (see Figure 1, first step), and with the realized template, the code was divided into focal methods and categorized according to the values of cyclomatic complexity (CC). When the cyclomatic complexity threshold is 10, the method is considered simple and straightforward to test. Between 11-19 threshold, is the moderate method,

and a threshold of 25 indicates that the method is overly complex and may require refactoring to improve testability. To test evaluation metrics on C# code, an entire project containing focal methods and focal method tests is required; thus, we created a project template into which focal methods and generated unit test methods can be injected. For the testing environment, we used Python with the PydanticAI library [7], which allowed us to define the structure of an object (see Figure 2). In this case, the object was a unit test using the Arrange-Act-Assert template for the structured response. This object could then be further used to inject generated unit tests into the C# testing project (see Figure 1, 2nd step).

The prompt engineering for an LLM involves the context and instruction of the task and should be well designed to guide the result of the desired output and have a marked effect on the responses generated by the model [3]. As shown in Figure 2, the prompt consists of these components: instructions (task and structure relevant), input and output [5]. The instruction part clearly states the task, setting the tone for the model response, while the input provides the necessary context or specific examples. In this context, the instruction is structure-related and shapes the format and structure of the LLM response.

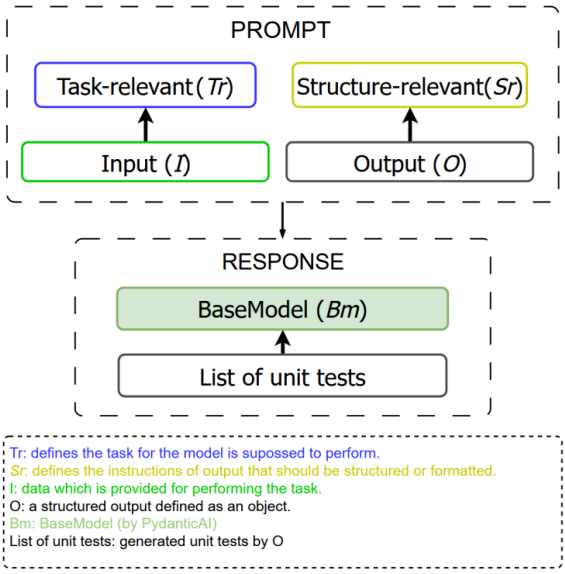

Figure 2. LLM prompt definition

In this research, six large language models were selected: GPT-4o-mini, GPT-4o, Gemini-2.0-flash, Llama.3.3-70b-versatile, Qwen-2.5-32b, Qwen-2.5-coder-32b [8-12]. None of the parameters of these models were modified, meaning they were used as is out of the box. We used three different LLM API providers to integrate and execute prompts on selected models in our research: OpenAI API, Gemini Developer API, Groq API (see Figure 1, 3rd step). Unit tests collected from these providers were validated with the PydanticAI library and then injected into the C# testing project.

In this research, the quality of the generated unit tests was leveraged to widely used metrics to find answers to the findings.

RQ1: the count of unit tests generated using the method of cyclomatic complexity;
RQ2: line coverage, branch coverage, mutation coverage;
RQ3: killed mutants, surviving mutants.

All generated unit tests were validated for code compilation errors, and afterwards, the tools for code quality metric analysis were executed (see Figure 1, 4th step). For unit test code lines and branch coverage evaluation, the C#/.NET coverage tool was used, and for performing robustness testing of various mutations, Stryker.NET was applied.

3 Results

The analysis of experimental results began to find the answer to RQ1. In Table 1, we present the LLM test count generated for each method depending on the class of cyclomatic complexity. It was noticed that the Gemini 2.0 flash model generated most unit tests for simple focal methods. Moving onto the medium-cyclomatic complexity tasks, the qwen-2.5-coder model has generated more tests than any other model. For the high-complexity methods, once again, qwen-2.5-coder-32b was able to generate more tests, followed by the Gemini-2.0-flash model, 38 and 35 tests, respectively. The number of tests generated cannot be an accurate measure of the unit test quality; therefore, in a further paragraph, the quality metrics of these generated unit tests are discussed.

Regarding the RQ2, we evaluated the coverage of the lines, branches, and mutations, which were measured by how many codes from the focal method were covered by the unit tests generated. On simple cyclomatic complexity, all investigated models could achieve 100% code lines, branch

and mutation coverage of focal methods. The challenges arise when the focal method has a cyclomatic complexity value greater than 10. The results of the coverages of moderate and complex CC of the focal methods are presented in Figure 3.

Table 1. LLMs generated test counts for each varying cyclomatic complexity task.

	Gemini-2.0-flash (1)			**GPT-4o (2)**			**GPT-4o-mini (3)**			**Llama-3.3-70b-versatile (4)**			**Qwen-2.5-32b (5)**			**Qwen-2.5-coder-32b (6)**		
Cyclomatic complexity (class)	S	M	C	S	M	C	S	M	C	S	M	C	S	M	C	S	M	C
Generated tests (count)	12	12	35	8	7	26	9	10	17	9	9	28	7	12	24	9	26	38

Where S is simple cyclomatic complexity (CC≤10), M is moderate cyclomatic complexity (CC is between 11-19), and C is complex cyclomatic complexity (CC≥ 20).

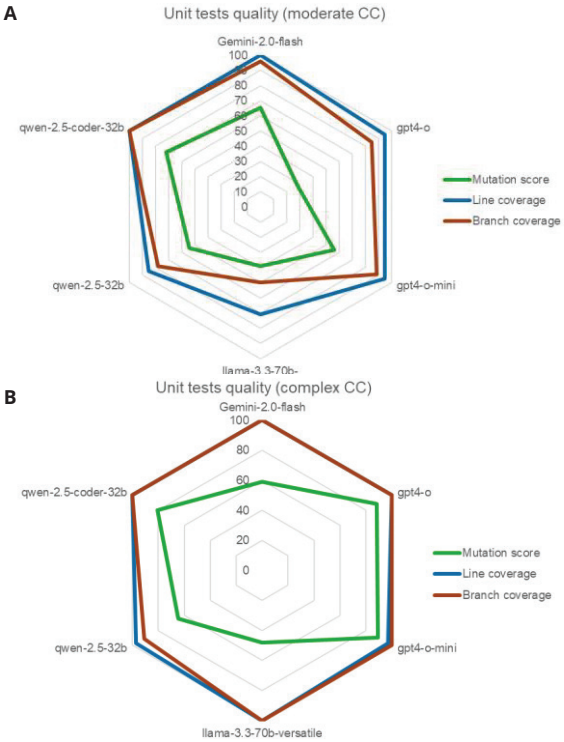

Figure 3. Unit tests quality metric results with moderate (3a) and complex (3b) CCs class of focal methods.

From Figure 3a, we can observe that for the moderate complexity code segments, only two models achieved 100 % code coverage, and only one of the models, qwen-2.5-coder-32b, achieved 100 % branch coverage for moderate difficulty methods. The qwen-2.5-coder-32b model also received the highest mutation score for moderate difficulty unit tests at 71.74 %. Unexpectedly, the GPT-4o model scored lower in branch coverage and mutation score than its less capable model, GPT-4o-mini, scoring 85 and 89 % in branch coverage and 28.26 and 56.52 % in mutation score, respectively. When analyzing the results of the coverage of the complex focal method, the highest percentage of mutation coverage received by GPT-4o and GPT-4o-mini at ~89%, and the qwen-2.5-coder-32b model achieved the third highest mutation score at 80.39 % (see Figure 3b). It was noticed that the llama-3.3-70b-versatile model could not compete with other models like GPT and Gwen when introduced to moderate and complex CC of focal methods, usually scoring lower in mutation coverage than competitors.

Looking for the answer to research question RQ3, we analyze the robustness of generated unit tests when evaluating the count of generated unit tests with killed and survived mutants of them.

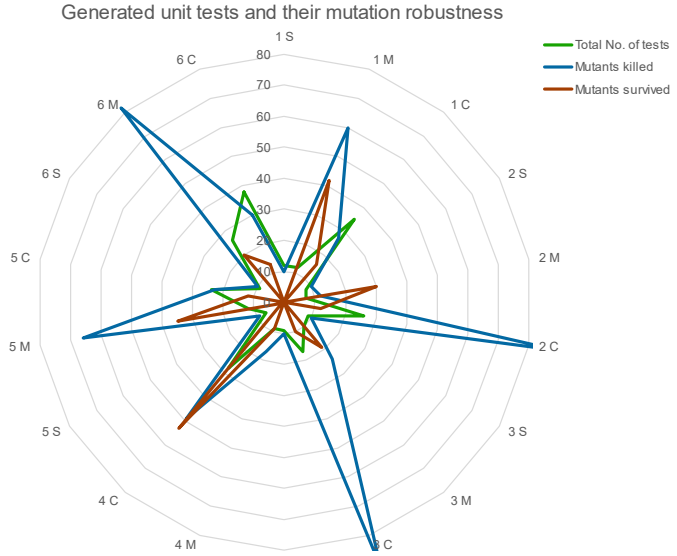

Figure 4. Generated unit tests and their mutation robustness

where: 1- Gemini-2.0-flash, 2 - GPT-4o, 3 - GPT-4o-mini, 4 - Llama-3.3-70b-versatile, 5- Qwen-2.5-32b, 6 - Qwen-2.5-coder-32b.

In Figure 4, it was observed that most mutants were killed by the GPT-4o-mini, GPT-4o and Qwen-2.5-coder-32b models, meaning that these models should have a higher mutation score than other models which did not have as many mutants killed or had many of the mutants survive. Since the mutation score measures the 'strength' of a test suite and characterizes its bug detection abilities, it is highly likely that these tests generated by the GPT-4o-mini, GPT-4o and qwen-2.5-coder-32b models are of higher quality, meaning they are more likely to sufficiently cover focal methods under test. The highest number of mutants that survived mutations were from models from Llama-3.3-70b-versatile and Gemini-2.0-flash. Llama-3.3-70b-versatile model had more mutations survive than mutations killed, indicating that the unit tests generated by this model may not be ideal to adequately cover test code.

Another observation was made when further analyzing the results; it seems that the LLM mutation score on medium complexity task was lower than for the high complexity task. It could be the case that the LLM had more knowledge about testing the selected complex method; thus, the medium method was not recognized as such. Another case could be that the medium cyclomatic complexity task had more complex inner workings for the LLM to test, compared to the high cyclomatic complexity task.

4 Conclusions

This paper analyzes the capabilities of unit tests generated by six LLMs utilizing a structured output validation library (PydanticAI) and evaluates their quality. The results of the experiment carried out indicate that all unit tests generated achieve 100% code, branch, and mutation coverage when the focal method has a simple cyclomatic complexity. The quality varies considerably when faced with moderate and complex cyclomatic complexity of methods. This research distinguishes the LLM model Gwen-2.5-coder-32b, which achieves high branch coverage and mutation robustness of generated unit tests across different method complexities.

Future work on the quality evaluation of LLMs-generated unit tests could extend to assessment of assertion quality, including density, diversity, and logical soundness.

References

[1] Pan, R., Kim, M., Krishna, R., Pavuluri, R., & Sinha, S. (2024). Multi-language Unit Test Generation using LLMs. arXiv preprint arXiv:2409.03093.

[2] Elbaum, S., Chin, H. N., Dwyer, M. B., & Jorde, M. (2008). Carving and replaying differential unit test cases from system test cases. IEEE Transactions on Software Engineering, 35(1), 29-45.

[3] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM computing surveys, 55(9), 1-35.

[4] Ryan, G., Jain, S., Shang, M., Wang, S., Ma, X., Ramanathan, M. K., & Ray, B. (2024). Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. Proceedings of the ACM on Software Engineering, 1(FSE), 951-971.

[5] Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). A systematic survey of prompt engineering in large language models: Techniques and applications. arXiv preprint arXiv:2402.07927.

[6] Bhatia, S., Gandhi, T., Kumar, D., & Jalote, P. (2024, April). Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In Proceedings of the 1st International Workshop on Large Language Models for Code (pp. 54–61).

[7] Pydantic Team. (2024). PydanticAI: Agent Framework for Generative AI. Pydantic. https://ai.pydantic.dev/

[8] OpenAI. (2024, July 18). GPT-4o mini: Advancing cost-efficient intelligence. OpenAI. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[9] OpenAI. (2024, May 13). Hello GPT-4o. OpenAI. https://openai.com/index/hello-gpt-4o/

[10] Kavukcuoglu, K. (2025, February 5). Gemini 2.0 model updates: 2.0 Flash, Flash-Lite, Pro Experimental. Google. https://blog.google/technology/google-deepmind/gemini-model-updates-february-2025/

[11] Meta. (2024, December 7). Model Cards and Prompt Formats – Llama 3.3. Llama. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/

[12] Qwen Team. (2024). Qwen2.5 Technical Report. arXiv preprint arXiv:2412.15115. https://arxiv.org/pdf/2412.15115