Private GPTs for LLM-driven testing in automotive development of software and AI

Consuelo Rojas[1][0009−0000−4466−8625], Jakub Jagielski[2], Markus Quade[2], and Markus Abel[2][0000−0001−8963−6010]

[1]Universitat polytècnica de Catalunya, Spain,
[2]Ambrosys GmbH, Marlene-Dietrich Str. 16, 14482 Potsdam,
info@ambrosys.de, consuelo.belen.paz.rojas@upc.edu

Abstract. In this contribution, we examine the capability of privatly deploy GPTs to automatically generate executable test code based on requirements. More specifically, we use acceptance criteria as input, formulated as part of epics, or stories, which are typically used in modern development processes. This gives product owners, or business intelligence, respectively, a way to directly produce testable criteria through the use of LLMs. We explore the quality of the so-produced tests in two ways: i) directly by letting the LLM generate code from requirements, ii) through an intermediate step using Gherkin syntax. As a result, it turns out that the two-step procedure yields more reliable results - where we define better in terms of human readability and best coding practices.

Keywords: Component · formatting style · test code · RAG · LLM

1 Introduction

With the massification of Large Language Models (LLMs) and the rise of coding agents, their potential to speed up development by automating various tasks, like test case generation, has become a significant research area [1, 2]. LLMs excel at generating coherent content, including code snippets and human-like paragraphs, making them valuable tools for enhancing testing practices. This capability extends to generating diverse unit tests, which can significantly improve code coverage and reveal potential bugs early in the development cycle [3]. While automated unit test generation has been a focus of software engineering research for decades, existing tools often produce tests with poor readability and are predominantly available for a limited set of programming languages, such as Java, C, C#, and more recently, Python [3]. The integration of LLMs offers a novel approach to overcome these limitations, enabling the generation of more expressive and contextually relevant test cases [4].

recent studies have explored the effectiveness of LLMs in unit test generation, with some focusing on closed-source models and fixed prompting strategies. However, there remains a critical gap in understanding the performance of advances open-source LLMs under diverse prompting conditions for this task, particularly in generation robust and reliable test suites [5]. The use of technologies, such as

privately deployed GPTs offers a promising opportunity to tailored the LLMs to the customers specifications, while addressing concerns about data security, combined with technologies like Retrieval Augmented Generation (RAG), the test generation process could be significantly enhanced, enabling the generation of more comprehensive and contextually relevant test cases [6, 7].

In this study, we investigate how structured acceptance criteria can be paired with LLMs to generate test sets for software projects. Specifically, we compare results obtained when prompting the LLM in natural language (NL) versus intermediate formats such as Gherkin syntax (GS). We then evaluate the quality of the generated tests using metrics such as executability, pass rate, and coverage of the files created.

2 Methodology

RAG is a technique that improves the performance of LLMs by combining them with external information retrieval systems[7]. RAG allows the model to search and retrieve relevant documents or data from outside sources, instead of relying on the knowledge embedded within the model itself. This approach is especially useful for tasks which require highly specific or up-to-date domain knowledge. RAG does not require retraining the LLM on the data, instead it lets the model fetch and incorporate the information on the fly.

The physical setup of the system is described in Fig. 1. The LLM model is deployed on a server and is fronted by an HTTP interface. The workflow is as follows:

1. A developer submits code to the server.
2. A quality assurer prompts the model with acceptance criteria.
3. The developer's code is loaded into the model's context.
4. The server responds to the quality assurer with the prompt's result, namely the test code to be executed.
5. The resulting test code is executed against the developer's code, to check whether the acceptance criteria have been met.

3 Implementation

In the following section, we present the case study of test generated for a YOLO-based license plate recognition [8, 9]. The interface and the train scripts were given as context to the Deepseek-Coder-v2[10] LLM deployed on a server. All prompts instructed the model to follow good coding conventions and include the necessary libraries or package imports.

The quality of the generated test was evaluated using the following metrics: i) **Executable (%):** Percentage of files with one or more executable tests; ii) **Pass Rate (%)** Percentage of the files that have, at least, one executable test that pass; iii) **Coverage (%):** This metric represents the mean percentage of

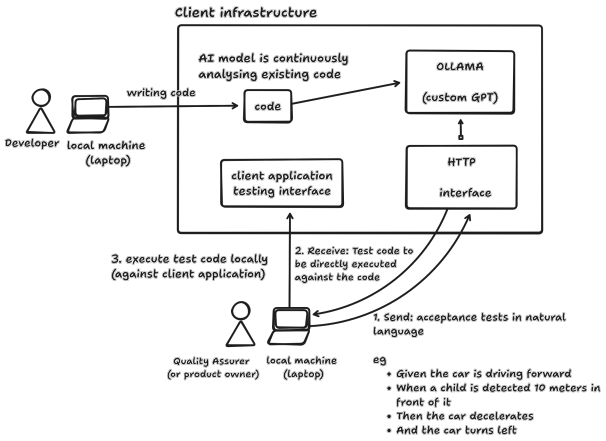

Fig. 1: Architectural setup of the system. It shows the client infrastructure and how the different actors interact with the system.

code lines executed during testing, considering only files with at least one executable test. Coverage is calculated as the ratio between the number of lines executed by the tests and the total number of executable lines in the original file. *Note:* For clarity, inline comments have been omitted from the code examples. Common import issues were resolved during post-processing to ensure that the module hierarchy did not cause errors and that all required elements were properly accessible.

4 Use case: License Plate Recognition

For the experiment, we prompt the LLM a 100 times per prompting style. In Figure 2 shows an example of the NL-generated test response. In this case, the test does not correctly use the pytest framework as requested and exported the training dataset, which can lead to artificial inflated metrics.

In comparison, the Gherkin (GS) prompt produced more structured and reliable tests [11]. Figure 3 shows an example. The GS-generated tests do not export the training dataset and produce focused test cases aligned with the acceptance criteria. Minor issues such as placeholder values or missing imports can be corrected through post-processing.

Table 1 summarizes the evaluation metrics for the NL and GS prompting strategies. The results show that NL prompts achieve higher pass rates and code coverage, indicating that more of the model's code was execute and a larger proportion of test passed. However, this performance is likely influenced by unintended behaviors such as semantic errors or inclusion of the training dataset.

In contrast, GS prompts produced tests with slightly lower executability, but the failures or issues with the GS-generated test were limited to minor

```
import ...

from license_plate_detector.detect_license_plate import ...
from license_plate_detector import train

def test_detect_license_plates():
    detector = DetectLicensePlate()

    image_paths = [path for path in
        train.images_test_dir.glob("*.jpg")]

    selected_image_paths = random.sample(image_paths,
        min(len(image_paths), 100))

    correct_detections = 0

    for img_path in selected_image_paths:
        predicted_bounding_box =
            infer_extract_license_plate_bounding_box(img_path)
        if is_detected_correctly(img_path, predicted_bounding_box):
            correct_detections += 1


    detection_rate = (correct_detections / len(selected_image_paths))
        * 100

    assert detection_rate > 80
```

Fig. 2: Sample of response to "License Recognition" with NL prompt

problems, such as import errors and use of placeholder that can be change in post-processing.

Table 1: Metrics for Case Study (Sec. 4)

Metric	**NL**	**Gherkins**
Executable (%)	98.00	71.72
Pass Rate (%)	86.73	15.49
Coverage (%)	83.49 ± 0.13	5.18 ± 0.01

Overall, the main errors in NL prompts were incorrect function signatures, unintended use of the training dataset, and failure to follow instructions. For GS prompts, the main issues were limited to placeholders, which are easily resolved during post-processing. These results demonstrate that structured prompting improves reliability and maintainability of automatically generated tests, even if apparent coverage is lower than that achieved with free-form NL prompts.

```python
import ...

@pytest.fixture
def model():
    return YOLO("license_plate_detector/YOLO_LICENSE_PLATE.pt")

@pytest.mark.parametrize(
    "image_path",
    [
        Path("test_image1.jpg"), ...
    ],
)
def test_license_plate_detection(model, image_path):
    detector = DetectLicensePlate()
    predicted_bounding_box = detector(image_path)
    assert is_detected_correctly(
        img_path=image_path,
        predicted_bounding_box=predicted_bounding_box,
    )

@pytest.mark.parametrize("threshold", [0.8])
def test_license_plate_detection_accuracy(threshold, model):
    image_paths = [
        Path("test_image1.jpg"), ...
    ]

    detector = DetectLicensePlate()
    correct_detections = sum(
        is_detected_correctly(img_path=img_path,
             predicted_bounding_box=detector(img_path))
        for img_path in image_paths
    )

    accuracy = correct_detections / len(image_paths)
    assert (
        accuracy >= threshold
    ), f"...%"
```

Fig. 3: Sample of response to "License Recognition" with GS prompt

5 Conclusion

This study demonstrate that is feasible generate code test using private GPT models. Our experiments show that prompting in a structured manner generally improve the quality of the tests generated.

A key finding is that interacting with LLMs through a structured language, specially Gherkin syntax, significantly enhances the quality and reliability of the produced test code.

Further exploration would include evaluate the use of a specialized LLM to translate natural language prompts into a intermediate representations like Gherkin, and employ a dedicated LLM, to generate the corresponding code from the structured prompt.

Ultimately, these approaches guide us toward a more robust and automated test generation workflow. We plan to further investigate the comparative performance of commercial versus non-commercial LLMs under established coding standards and more complex scenarios.

Acknowledgments

Part of this work was funded by the European Union, under HORIZON-MSCA-2022-DN, *Improving BiomEdical diagnosis through LIGHT-based technologies and machine learning* "BE-LIGHT" (GA nº 101119924 - BE-LIGHT) project.

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union nor UPC. Neither the European Union nor the granting authority can be held responsible for them.

The code and data used in this paper are publicly available at: https://github.com/consuelorojas/Exploratory-LLM.

References

1. S. Alagarsamy, C. Tantithamthavorn, W. Takerngsaksiri, C. Arora, and A. Aleti, "Enhancing Large Language Models for Text-to-Testcase Generation," *CoRR*, vol. abs/2402.11910, pp. 1–12, 2024, doi: 10.48550/arXiv.2402.11910.
2. R. Santos, I. Santos, C. Magalhães, and R. de Souza Santos, "Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing," in *Proc. IEEE Conf. Softw. Test., Verification Validation (ICST)*, Toronto, ON, Canada, 2024, pp. 353–360, doi: 10.1109/ICST60714.2024.00039.
3. R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, "Multi-language Unit Test Generation using LLMs," *arXiv preprint arXiv:2409.03093*, pp. 1–23, Sep. 2024. [Online]. Available: https://arxiv.org/abs/2409.03093
4. S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools," in Proc. 1st Int. Workshop Large Lang. Models Code (LLM4Code '24)*, Lisbon, Portugal, 2024, pp. 54–61, doi: 10.1145/3643795.3648396.
5. L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, "An Empirical Study of Unit Test Generation with Large Language Models," *arXiv preprint arXiv:2406.18181*, pp. 1–15, 2024. [Online]. Available: https://arxiv.org/abs/2406.18181
6. V. Alves, C. Bezerra, I. Machado, L. Rocha, T. Virgínio, and P. Silva, "Quality Assessment of Python Tests Generated by Large Language Models," *arXiv preprint arXiv:2506.14297*, pp. 1–18, 2025. [Online]. Available: https://arxiv.org/abs/2506.14297
7. S. Zhao, Y. Yang, Z. Wang, Z. He, L. K. Qiu, and L. Qiu, "Retrieval Augmented Generation (RAG) and beyond: A comprehensive survey on how to make your LLMs use external data more wisely," arXiv preprint, arXiv:2409.14924, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.14924
8. M. Yaseen, "What is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector," *arXiv preprint arXiv:2408.15857*, pp. 1–18, 2024. [Online]. Available: https://arxiv.org/abs/2408.15857
9. Roboflow Universe Projects, "License Plate Recognition Dataset," *Roboflow Universe*, Roboflow, Apr. 2025. [Online]. Available: https://universe.roboflow.com/roboflow-universe-projects/license-plate-recognition-rxg4e. Accessed: Sep. 29, 2025.
10. DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, W. Zeng, X. Bi, Z. Gu, H. Xu, D. Dai, K. Dong, L. Zhang, Y. Piao, Z.

Gou, Z. Xie, Z. Hao, B. Wang, J. Song, D. Chen, X. Xie, K. Guan, Y. You, A. Liu, Q. Du, W. Gao, X. Lu, Q. Chen, Y. Wang, C. Deng, J. Li, C. Zhao, C. Ruan, F. Luo, and W. Liang, “DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence,” *arXiv preprint arXiv:2406.11931*, pp. 1–39, 2024. [Online]. Available: https://arxiv.org/abs/2406.11931

11. Cucumber, "Gherkin reference," 2024. [Online]. Available: https://cucumber.io/docs/gherkin/reference/
12. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proc. IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. [Online]. Available: https://doi.org/10.1109/5.726791
13. Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, “A survey on evaluation of large language models,” *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, Art. no. 39, pp. 1–45, Mar. 2024, doi: 10.1145/3641289.