LLMLOOP: Improving LLM-Generated Code and Tests through Automated Iterative Feedback Loops

Ravin Ravi*, Dylan Bradshaw*, Stefano Ruberto†, Gunel Jahangirova‡, and Valerio Terragni*
*University of Auckland, Auckland, New Zealand
{rrav468, dbra157, vter674}@aucklanduni.ac.nz
†JRC European Commission, Ispra, Italy
Stefano.RUBERTO@ec.europa.eu
‡King's College London, London, United Kingdom
gunel.jahangirova@kcl.ac.uk

Abstract—**Large Language Models (LLMs) are showing remarkable performance in generating source code, yet the generated code often has issues like compilation errors or incorrect code. Researchers and developers often face wasted effort in implementing checks and refining LLM-generated code, frequently duplicating their efforts.**

This paper presents LLMLOOP, a framework that automates the refinement of both source code and test cases produced by LLMs. LLMLOOP employs five iterative loops: resolving compilation errors, addressing static analysis issues, fixing test case failures, and improving test quality through mutation analysis. These loops ensure the generation of high-quality test cases that serve as both a validation mechanism and a regression test suite for the generated code.

We evaluated LLMLOOP on HUMANEVAL-X, a recent benchmark of programming tasks. Results demonstrate the tool effectiveness in refining LLM-generated outputs. A demonstration video of the tool is available at https://youtu.be/2CLG9x1fsNI.

Index Terms—**AI4SE, software testing, program synthesis, automated test generation, Large Language Models**

I. INTRODUCTION

Recent years have seen remarkable progress in large language models (LLMs), leading to their adoption across a wide range of domains [1]. Beyond generating textual responses, LLMs have shown great potential in generating source code [2]–[5], promising to improve the productivity of software engineers [6].

Researchers and practitioners are studying and evaluating the effectiveness of LLMs in generating source code from textual prompts. While the results are promising [2], [3], [7], several challenges remain [3], [5]. For instance, such code often fails to compile, as LLMs do not inherently perform compilation checks, requiring more sophisticated pipelines to handle such tasks. Additionally, LLM-generated code frequently suffers from dependency issues [5], such as missing libraries. These models are also limited by the quality and of their training data [8], which may include outdated or buggy code samples.

We observed that researchers and practitioners consistently encounter these issues. This leads to significant wasted effort in implement automated techniques to fix LLM-generated code. A commonly adopted strategy involves **feedback loops** that integrate LLMs with source code analysis tools (e.g., compilers) [9]. These tools identify issues, which are then fed back to the LLM, providing contextual clues to refine the code based on the reported problems. This iterative approach allows LLMs to refine their output, correct mistakes, and produce more robust and contextually appropriate code. However, setting up these feedback loops often requires significant effort to implement and setup properly.

This paper presents **LLMLOOP**, a framework to enhance the quality and reliability of LLM-generated JAVA code. LLMLOOP serves as an intermediary between developers/researchers and LLMs, providing automated mechanisms to fix, validate, and improve both source code and test cases produced by LLMs. The tool is fully automated and highly configurable, enabling researchers and developers to efficiently refine LLM-generated code. LLMLOOP avoids the repeated effort in implementing feedback loops and can be seamlessly integrated in any tool or workflow that uses LLMs to generate JAVA source code.

The framework is for JAVA and runs within a dedicated DOCKER image, functioning as a secure sandbox environment. This ensures that the framework operates in isolation, protecting the host system from the execution of any potentially harmful code generated by the LLM during test case execution.

LLMLOOP currently implements five iterative self-refinement loops: (i) to fix compilation errors, (ii) to improve code quality issues reported by static analysis, (iii) to fix failures of automatically generated test cases, (iv) to improve the quality of the test cases via mutation analysis. These test cases serve a dual purpose: identifying bugs in LLM-generated code and providing an associated test suite for the generated code.

We evaluated LLMLOOP by assessing its effectiveness in generating programming solutions of the HUMANEVAL-X [2] benchmark, as compared to a baseline approach that invokes the LLM only once (without incorporating any feedback loop). The results show that LLMLOOP improves the quality of LLM-generated code: pass@10 of 90.24% versus pass@10 of 76.22% for the baseline. We believe that LLMLOOP provides substantial benefits to the developer and research communities that are using LLMs to generate code. A demonstration video on how to use it is available at `https://youtu.be/2CLG9x1fsNI`

We publicly released LLMLOOP's code and experimental data: `https://github.com/ravinravi03/LLMLOOP`

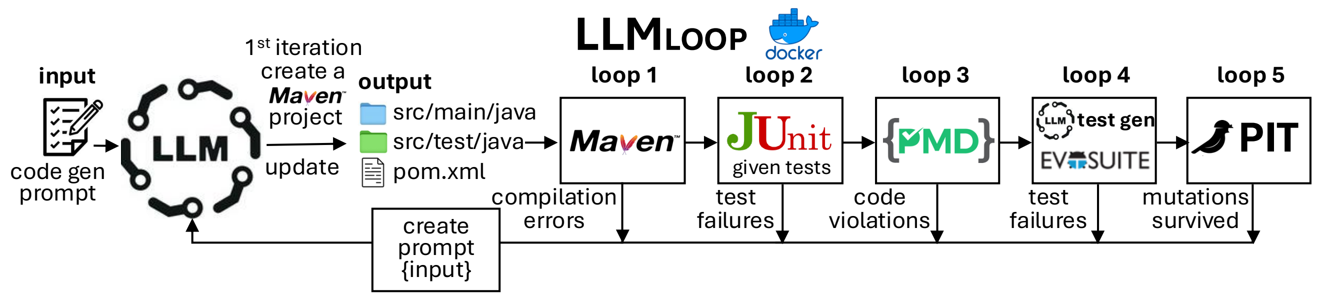

Fig. 1. Logical architecture of LLMLoop

II. LLMLoop

Figure 1 illustrates the logical architecture of LLMLoop, which leverages iterative feedback loops and multiple analysis methods to enhance LLM-generated Java code. For each type of feedback, LLMLoop generates a dedicated prompt to guide the LLM in improving the code (production or test) based on the feedback (see "create prompt {input}" in Figure 1).

Inputs. The inputs to LLMLoop are: 1) a prompt to generate a program, 2) a series of command-line arguments (see Table I), 3) an optional test suite to validate the generated code.

The framework dynamically adjusts the LLM's temperature to optimize the generation process. It begins with the default deterministic setting (temperature = 0) or the temperature specified in input (flag `-t`) but increments the temperature by 0.1 in successive runs if errors persist. This introduces variability, potentially resolving repeated issues. Studies like Liu et al. [3] suggest an optimal temperature for generating code is around 0.2, guiding LLMLoop's adaptive adjustments.

Outputs. The output of LLMLoop is a Maven project containing the improved code generated by the LLM along with an automatically generated test suite in addition to the given test suite (if any).

While the canonical input is a prompt and arguments, LLMLoop can also start from an existing Maven project. In such cases, it can automatically fix and improve a specific class under test or integrate newly generated code into the project. This functionality is particularly useful as it relies on dependencies already declared in the project. Additionally, users can use LLMLoop to create a new project from scratch, fix bugs in an existing project, or enhance test suite coverage by providing the root directory and the path to a test class. For simplicity, the rest of the paper will assume the canonical use of LLMLoop.

Implementation Details. LLMLoop is implemented for Java programs, leveraging the **Maven ecosystem** and the Maven build automation tool for dependency resolution.

To address security concerns related to running LLM-generated code, LLMLoop operates within a **Docker** container. This isolated environment safeguards the host system from potential vulnerabilities, including malicious code that might originate from open-source training data. Using a sandbox environment ensures the protection of sensitive information and maintains system integrity [2], [10].

TABLE I
COMMAND-LINE FLAGS OF LLMLOOP

Flag	Additional Argument	Description
-e	-	Enable EvoSuite
-p	Path to Java project (String)	Path to an existing Java project
-d	-	Enable debug logging
-t	-	Enable test generation using LLMs
-r	-	Enable code coverage report
-n	Number of retries (Int)	Set the number of attempts in the feedback loop
-s	-	Enable static analysis of code generated by LLM
-mut	-	Enable mutation analysis of tests generated by LLM
-temp	Temperature value (Float)	Set the temperature of the LLM model
-ts	Path to test suite (String)	The relative path of the test suite
-depth	Dependency depth (Int)	The depth of the dependency tree
-m	Number of minutes (Int)	Number of minutes that EvoSuite runs for

LLMLoop's implementation uses the OpenAI APIs[1] to interact with LLMs, leveraging OpenAI's extensive range of powerful LLMs and its position as a leading provider in the LLM field. However, it could be easily adapted to any LLM.

A. Feedback Loops in LLMLoop

Loop 1: Fixing Compilation Errors The first feedback loop ensures all generated or modified code is compilable. The process begins with an initial prompt to the LLM, supplemented with comments to direct it to produce Java 11-compliant code. We choose Java 11 because EvoSuite currently supports up to Java 11[2]. The LLM's response is structured as a nested JSON object containing:

- `src`: A map of file paths to code strings.
- `main`: Configuration for running the project.
- `dependencies`: A list of required dependencies.

The framework parses the JSON object, updates the Maven project, and attempts to compile it. If compilation errors are detected, their details (location and type) are fed back to the LLM for refinement. This loop continues until the code compiles successfully or a set limit is reached (`-n`).

Loop 2: Test Failures After generating compilable code, the framework runs the given test cases (if any). The feedback provided to the LLM consists of the test cases that fail, along with the corresponding stack traces for each failure. In this

[1]https://platform.openai.com/docs/models
[2]https://github.com/EvoSuite/evosuite

loop, we assume the given tests are correct and only ask the LLM to fix the generated code so that the tests pass.

Loop 3: Static Analysis When all given test pass, or the budget is reached, the framework performs static analysis using the PMD MAVEN plugin. PMD[3] is an extensible multilanguage static code analysis tools that detects common programming flaws like unused variables, empty catch blocks, unnecessary object creation, etc. It comes with 400+ built-in rules and can be extended with custom rules. It parses source files into abstract syntax trees (AST) and runs rules against them to find violations. Once the PMD report is generated, LLMLOOP performs the following steps:

1) Parses the PMD report for violation details.
2) Prompts the LLM to fix these issues.
3) Iterates until no violations remain or the iteration limit is reached.

To ensure compatibility, the framework dynamically verifies and adds required plugins and dependencies to the `pom.xml` file using MAVEN.

Loop 4: Test Generation LLMLOOP supports two test generation strategies:

- **EVOSUITE:** A search-based tool that generates unit tests with high code coverage [11], [12].
- **LLM-Generated Tests:** The LLM creates test cases based on provided prompts and project context [13]–[16].

The two modes of test generation offer different approaches to testing. The test suite generation by EVOSUITE focuses on achieving a set of coverage objectives guided by a fitness function. The produced test cases contain test oracles that capture the implemented behaviour (but not the intended behaviour), leading to test cases applicable only for a regression scenario [17], so all tests would pass by construction.

The test generation by LLMs, however, is not guided by any pre-determined algorithm, but is expected to produce test code similar to the one produced by humans (due to being trained on a large set of human-produced data) and often produces test oracles that capture the expected behaviour instead of the implemented behavior [17], [18]. The prompt used by LLMLOOP explicitly instructs to generate test cases that achieve high coverage, to ensure that both positive and negative scenarios are covered and to take into account exceptional cases and boundary values.

Both types of generated test cases are validated through the MAVEN test lifecycle. If failures occur for the LLM-generated tests, the details are sent back to the LLM for refinement. This iterative process continues until the test suite pass all the test. Note that in order to make the test suites pass the LLMs can change either the generated source code, the generated test code, or both. This means that LLMLOOP lets the LLM to autonomously determine whether the test cases reveal a bug in the generated code or if the test cases themselves contain issues that need to be fixed. As such, loop 4 can potentially improve both the code under test and the associated automatically LLM-generated test suite.

Loop 5: Mutation Analysis Once LLMLOOP ensures that all the generated test cases are passing, it proceeds with the further improvement of the test suite quality. For this purpose, we use mutation testing, which evaluates the test suite effectiveness by introducing errors (mutants) into the code. These mutants are slight variations of the original code. The test suite is then executed against these mutants to determine whether it can detect the seeded errors. If a test suite fails to identify a mutant, it indicates a potential weakness or gap in the tests. This approach provides a quantitative measure of test suite quality. Loop 5 needs that all generated tests are passing on the given version, as mutation testing requires a 'green' test suite were all test pass. Note that this step is only performed for the generated tests, not for the given tests (used in loop 2).

LLMLOOP uses the PIT[4] mutation testing tool and performs the following steps:

1) Generate a report of surviving and killed mutants.
2) Provide details of surviving mutants to the LLM for refinement.
3) Iterate until all mutants are addressed or the iteration limit is reached.

III. EVALUATION

Benchmark. To evaluate LLMLOOP, we used **HUMANEVAL-X** [19], an extension of the popular HUMANEVAL [2] benchmark. Unlike the original benchmark, which focuses solely on PYTHON, HUMANEVAL-X expands to multiple programming languages, including JAVA, and introduces 80× more test cases [3]. HUMANEVAL-X contains **164 coding problems in JAVA**, each comprising a function signature, a docstring describing the function's intended behavior (which we will use as the initial LLM prompt), and a test suite of unit tests. These tests serve as a widely accepted proxy for correctness. Following prior work on program synthesis, we consider a solution correct if it passes all the provided test cases [19]. The test suite is divided into example and validation tests. *Example tests* (used in Loop 2) are designed to guide the program synthesis tool in solving the task; *validation tests* are exclusively used to assess the correctness of the generated code and kept hidden during program synthesis.

Experimental Setup. We developed a PYTHON script to convert the function signatures and docstrings from HUMANEVAL-X into class skeletons. Additionally, we transformed HUMANEVAL-X's tests, originally written as standalone main functions, into JUNIT tests for compatibility with the framework. After reformatting the problems, LLMLOOP launches a DOCKER container based on the framework's image, with the project files mounted into the container for execution.

[3]https://docs.pmd-code.org/latest/index.html

[4]https://pitest.org/

TABLE II
PASS@1 OF HUMANEVAL-X PROBLEMS (MEAN OVER 10 RUNS) AT EACH STEP OF LLMLOOP.

LLMLOOP stage	**pass@1**	**pass@1**
(mean ± std)	**(mean %)**	
#1 Baseline (no feedback loop)	117.50 ± 1.20	71.65%
#2 Compilation (Loop 1)	125.30 ± 1.55	76.40%
#3 HUMANEVAL-X Given Tests (Loop 2)	130.40 ± 2.91	79.51%
#4 Static Analysis (Loop 3)	130.50 ± 3.11	79.57%
Loops 4 and 5 repeated for LLM and EVOSUITE-generated test cases		
#5 LLM Tests (Loops 4 + 5)	132.50 ± 3.14	80.55%
#6 EVOSUITE Tests (Loops 4 + 5)	132.60 ± 3.10	80.85%

For our experiments, we used OpenAI's GPT-4O-MINI LLM. The terminal prompt provided to the DOCKER container included all the necessary flags to perform every stage of the framework (see Table I). The number of retries (`-n`) and the depth of the dependency tree (`-d`) were both set to five. We presented each of the 164 JAVA problems to the LLM using a standardized initial prompt for code generation. To account for the non-deterministic nature of LLM output, we ran LLMLOOP ten times for each problem.

The second column in Table II gives the order of loops used in our experiments. We started with the Compilation Loop (Loop 1 in Figure 1) because successful compilation is necessary for executing tests and performing static analysis. We then used the HUMANEVAL-X example tests to guide the feedback loop (Loop 2), applied static analysis (Loop 3), and finally generated additional tests using both LLM and EVOSUITE (Loops 4-5). It is important to mention that Loop 1 is triggered every time the code changes, even in subsequent loops. This is because Loop 1 must always ensure that the code is compilable to perform the static analysis and run the test suites.

We evaluated the framework's effectiveness with **pass@k** [2]: The probability that at least one out of k independently generated code samples for a given problem passes all the test cases. We computed *pass@1* to *pass@10*, aligned with the ten runs for each problem in input. This metric is particularly relevant for LLMs due to their non-deterministic nature, where identical inputs may produce different outputs. Considering multiple samples, *pass@k* provides a robust measure of performance [2].

How to run the experiments. To run the HUMANEVAL-X evaluation, the following steps should be followed:

1) Build the DOCKER image and tag it as `framework`.
2) Run the `generate.py` script to begin generating solution attempts for the problems in the dataset.
3) Once the script completes, the solutions will be in the `results/result.json` file, and the logs will be in the `framework_logs` directory.

There are various scripts to analyze the data. Each of the analysis scripts takes a list of directories or runs as command-

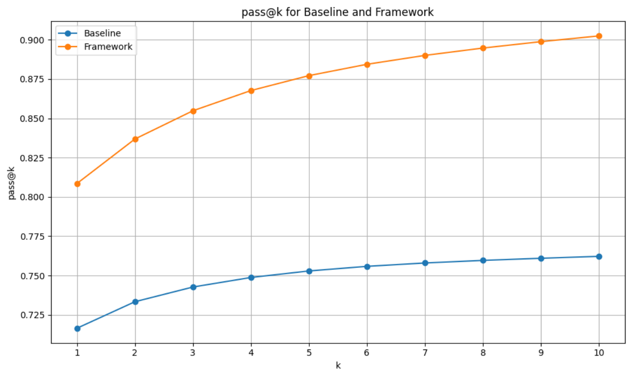

Fig. 2. Pass@K for the ten runs, baseline vs LLMLOOP

line arguments, each containing the framework JSON logs for the problem. The scripts include:

- `aggregate.py`: Calculates the average number of problems passed at each stage, both individually and cumulatively, with standard deviations.
- `stats.py`: Displays this data as a bar graph.
- `pass_k.py`: Calculates the pass@k values for the baseline and framework and plots them on a line graph.

Results. Table II shows the average number of problems solved at each stage of the framework across ten runs. The framework solved an average of 132.6 problems, a 9.2% improvement over the baseline, which solved 117.5 problems. Average pass@1 increased from 71.65% for the baseline to 80.85% after completing all loops of LLMLOOP.

Figure 2 illustrates the improvement in *pass@k* metrics. The baseline achieved a *pass@1* of 71.65%, rising to 76.22% at *pass@10*. In comparison, LLMLOOP achieved a *pass@1* of 80.85% and a *pass@10* of 90.24%. The framework consistently outperformed the baseline, with a 9.2% increase at *pass@1* and a peak difference of 14.02% at *pass@10*. The iterative feedback loops in LLMLOOP amplified the baseline's improvements as the number of code samples increased.

The results show that the performance of GPT-4O-MINI benefited from the iterative refinement provided by LLMLOOP. The largest improvement came from the Compilation Loop, which ensures that code compiles successfully, enabling subsequent tests and analyses. The feedback of the HUMANEVAL-X's example tests (Loop 2) also contributed significantly, aligning the generated code more closely with the problem requirements.

Later stages of the framework provided smaller gains. The LLM-generated tests (Loop 4) showed (small) improvements, as they could targeted expected behavior [17] (rather than implemented one). Differently, EVOSUITE-generated tests had zero gain in pass@1, as expected. Indeed, EVOSUITE generates tests with regression oracles to validate implemented logic (for regression testing), rather than checking program correctness [11], [17].

IV. LIMITATIONS AND FUTURE WORK

One limitation of LLMLOOP is the time required to complete all loops, which increases computational costs due to the frequent invocation of the LLM (often taking several minutes). This makes the approach less **sustainable**, as repeatedly invoking the LLM is very expensive in terms of both energy and computation. Future work could focus on reducing the number of LLM interactions by prioritizing feedback that is most likely to significantly improve the code. For example, LLMLOOP could begin by addressing the compilation error associated with the most issues in the code.

Another important future work is **evaluating the quality of the tests** generated by LLMLOOP (in particular the improvement of Loop 5). Due to space limitations, this was not included in this paper. It will be interesting assessing both LLM-generated and EVOSUITE tests generated and improved via feedback loops [20].

Future work could also integrate LLMLOOP as **a plugin for IDEs** to encourage adoption by developers. While researchers may continue using the command-line version, an IDE plugin would improve usability, bringing LLMLOOP closer to a mature, IDE-friendly tool.

Finally, extending LLMLOOP to **support PYTHON** is also important. This could replace EVOSUITE with PYNGUIN [21] to generate PYTHON test cases, and PIT with MUTPY.

V. RELATED WORK

Using feedback loops is a common approach to improving LLM-generated code [10]. However, LLMLOOP is the first framework designed to reduce the repeated effort typically required to set up such loops. Additionally, LLMLOOP introduces new types of feedback loops that are not explored in other techniques, which focus solely on handling compilation errors and test failures [13]–[16].

The work most closely related is SELFEVOLVE [9], which employs a self-correcting loop with a PYTHON interpreter to detect and fix errors in the generated code, including test failures. The loop iterates until all errors are resolved or a maximum number of iterations is reached. However, SELFEVOLVE is limited to PYTHON and does not implement feedback loops using static analysis output or automated test generation, as LLMLOOP does. Also, SELFEVOLVE is not currently publicly available, and its companion paper is an ArXiv preprint, which may not have been peer-reviewed yet.

VI. CONCLUSION

This paper presented LLMLOOP, a framework to automatically improve LLM-generated JAVA code through iterative feedback loops. By open-sourcing LLMLOOP, we aim to empower researchers and developers with an important asset to reduce repeated effort when generating code with LLMs. Moreover, the community could extend its capabilities, optimize its performance, and explore new feedback mechanisms. We hope that LLMLOOP will serve as a useful starting point for improving the quality and reliability of LLM-generated code.

REFERENCES

[1] M. U. Hadi, Q. Al Tashi, A. Shah, R. Qureshi, A. Muneer, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, J. Wu *et al.*, “Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects,” *Authorea Preprints*, 2024.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[3] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[4] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, “Improving llm code generation with grammar augmentation,” *arXiv preprint arXiv:2403.01632*, 2024.

[5] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “Evaluating large language models in class-level code generation,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[6] V. Terragni, A. Vella, P. Roop, and K. Blincoe, “The future of ai-driven software engineering,” *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025.

[7] W. Hou and Z. Ji, “A systematic evaluation of large language models for generating programming code,” *arXiv preprint arXiv:2403.00894*, 2024.

[8] L. Perez, L. Ottens, and S. Viswanathan, “Automatic code generation using pre-trained language models,” *arXiv preprint arXiv:2102.10535*, 2021.

[9] T. T. Quoc, D. H. Minh, T. Q. Thanh, and A. Nguyen-Duc, “An empirical study on self-correcting large language models for data science code generation,” *arXiv preprint arXiv:2408.15658*, 2024.

[10] J. Wang and Y. Chen, “A review on code generation with llms: Application and evaluation,” in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.

[11] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” in *ESEC/FSE*, 2011, pp. 416–419.

[12] G. Jahangirova and V. Terragni, “Sbft tool competition 2023 - java test case generation track,” 2023, pp. 61–64.

[13] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *IEEE Transactions on Software Engineering*, 2023.

[14] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, “Llm-powered test case generation for detecting tricky bugs,” *arXiv preprint arXiv:2404.10304*, 2024.

[15] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, “Llm for test script generation and migration: Challenges, capabilities, and opportunities,” in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 2023, pp. 206–217.

[16] S. Kang, J. Yoon, and S. Yoo, “Large language models are few-shot testers: Exploring llm-based general bug reproduction,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[17] M. Konstantinou, R. Degiovanni, and M. Papadakis, “Do llms generate test oracles that capture the actual or the expected program behaviour?” *arXiv e-prints*, pp. arXiv–2410, 2024.

[18] S. Ruberto, J. Perera, G. Jahangirova, and V. Terragni, “From implemented to expected behaviors: Leveraging regression oracles for non-regression fault detection using llms,” in *IEEE Conference on Software Testing, Verification and Validation Workshop*, 2025.

[19] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li *et al.*, “Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x,” *arXiv preprint arXiv:2303.17568*, 2023.

[20] W. C. Ouedraogo, K. Kabore, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyande, “Llms and prompting for unit test generation: A large-scale evaluation,” in *Proceedings of the 39th IEEE/ACM Inter. Conf. on Automated Software Engineering*, 2024, pp. 2464–2465.

[21] S. Lukasczyk and G. Fraser, “Pynguin: Automated unit test generation for python,” in *ICSE*, 2022, pp. 168–172.