Unit Test Generation Multi-Agent AI System for Enhancing Software Documentation and Code Coverage

1st Dimitrije Stojanović
University of Novi Sad
Faculty of Technical Sciences
Departmant of Computing and Control Engineering
Novi Sad, Serbia

2nd Bogdan Pavković
University of Novi Sad
Faculty of Technical Sciences
Departmant of Computing and Control Engineering
Novi Sad, Serbia

3rd Nenad Četić
University of Novi Sad
Faculty of Technical Sciences
Departmant of Computing and Control Engineering
Novi Sad, Serbia

4th Momčilo Krunić
LabSoft
Novi Sad, Serbia

5th Luka Vidaković
LabSoft
Novi Sad, Serbia

2024 32nd Telecommunications Forum (TELFOR) | 979-8-3503-9106-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/TELFOR63250.2024.10819096

***Abstract*—Software development necessitates a robust testing plan though test development can be laborious and non-appealing task. We explore the utilization of the application artificial intelligence agents for generating and executing unit tests, enhancing the "Mostly Basic Python Problems" dataset. We employ behavior-driven development within a three-agent system to generate user stories and unit tests. Empirical results indicate improvements in branch coverage, illustrating the effective utilization of large language models in software testing and development processes.**

***Index Terms*—AI agents, unit test generation, code generation, SW test automation, transformers, BDD**

I. Introduction

The continuous advancement of AI technologies presents opportunities to improve software development processes significantly. From helping engineers research problems, answer questions about different libraries to code generation tasks. One notable application is the employment of AI agents to generate unit tests. Testing of code is crucial for verifying the functionality of code, stability, robustness, completness, fulfilment of requirements. Automatic generation of unit tests using Agents can speed up the development process while offering a comprehensive test suit that will empower further code upgrades in the future. Another problem with AI is training LLM (large language mode). Data sets used for training LLM on code generation tasks can lack comprehensive unit tests. While they offer a possible solution for code, more crucial are unit tests that verify the functionality of code and cover edge cases. By executing a unit test suit on generated code we can be sure that LLM generates a correct code for a task.

As large language models (LLMs) continue to evolve and new applications are discovered, innovative methods for utilizing these models have also emerged. The need for LLMs to communicate with each other and the need for LLMs to interact with the environment led to the development of AI agents. The agentic way of working involves software programs that wrap LLM functionality into higher abstraction. This allows the parsing of LLM and communication between multiple LLMs and allows LLM to interact with the environment. Agents introduced new way of working where LLMs can call user defined tools for specific tasks. Moreover, agents can talk to each other and decide what next step to take. Also, agents can write code and execute a generated code to solve specified tasks.

We examine the potential of AI agents to enhance datasets by generating comprehensive unit tests for existing code base. We will present:

- Agentic system that uses BDD and iterations to generate unit tests
- Case study of an agentic system for improving test coverage of MBPP (mostly basic python programs) [1] data set.

Our goal is to develop an AI agent system capable of generating comprehensive unit tests. We hypothesize that with AI Agents we can generate unit tests to enhance legacy code and improve existing datasets for futher training of LLM. We expect to generate comprehensive unit tests that cover existing legacy code. Our research aims to benefit software developers by improving both legacy and new code, and AI reserachers by providing generated unit tests to validate AI models that generate code.

This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-65/2024-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project "Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad" (No. 01-3394/1).

II. Related Work

Significant research has been conducted in utilizing AI for software testing and dataset improvement, particularly in the generation of unit tests. The emergence of large language models has accelerated advancements in these areas, demonstrating the capacity of AI tools to enhance both software testing and the quality of datasets. More recently, the rise of advanced LLMs, such as GitHub Copilot and OpenAI Codex, has led to various studies on their potential for unit test generation and code understanding [2], [3]. These studies emphasize the need for high-quality datasets, which are essential for maximizing the accuracy and utility of AI-generated tests. Our research builds on these insights by incorporating systematic methods for dataset enhancement via AI agents.

Meta's TestGen-LLM [4] system exemplifies the industrial application of LLMs for unit test generation. The system enhances existing test suites by creating additional test cases aimed at improving coverage, particularly in edge-case scenarios. It ensures non-regression by maintaining and validating original tests. In large-scale deployments, such as Instagram and Facebook, TestGen-LLM increased test coverage by 11.5%, with 73% of generated test cases integrated into production. This highlights the practical utility of LLMs in generating scalable and reliable tests.

OpenDevin [5] utilizes a combination of dynamic program analysis and machine learning algorithms to automatically generate test cases that cover a wide range of code paths. The framework is built on the concept of test case synthesis, where the AI agent learns from existing test cases and code repositories to create new, effective tests that can identify potential bugs.

In summary, while significant progress has been made in AI-driven unit test generation and dataset enhancement, gaps remain, particularly in addressing the complexity of real-world applications and the iterative refinement of tests.

Compared to previous approaches, our proposal allows AI agents to iterate over their answers and generate better tests through multiple iterations. Additionally, by first generating user stories to guide the LLMs, we aim to cover more edge cases and increase unit test coverage.

III. Triple-Agent Testing Solution

We developed an AI agent system designed to improve unit tests for the selected function through a structured, multi-agent approach. We have chosen an Autogen framework [6] from Microsoft for our solution. Autogen is an evolving framework for developing AI Agent and it offers a built-in code executor that we used for improvement of generated unit tests. We combined two mechanisms: Multi-Agent Collaboration and Hierarchical Agents to create a multi Agent system for unit test generation. The overall structure of the AI Agenets system consists of three agents (Fig 1):

- User Story Writer Agent
- Unit Test Writer Agent
- Code Executor Agent

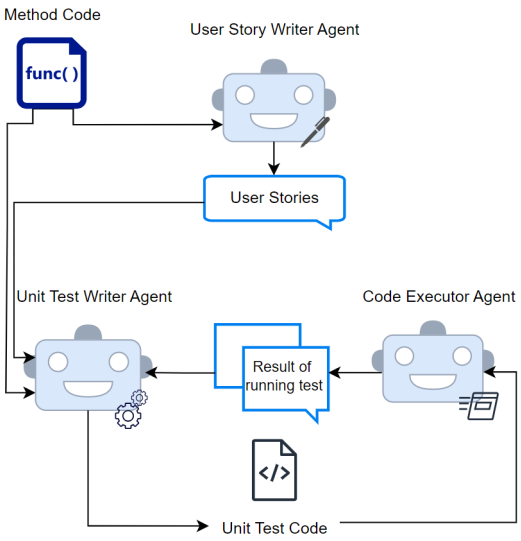

Fig. 1. System of AI agent used to generate unit tests

```
user_story_prompt = f"""
Based on the following Python method in ###,
write BDD-style user stories in Gherkin format.
Each user story should capture different scenarios that
this method handles. Include the title, description, and examples
for typical use cases and edge cases. Write Gherkin scenarios
for different cases.
###
{method_code}
###
"""
```

Fig. 2. User Story Writer Agent Prompt

A. User Story Writer Agent

The system takes as input a Python function and generates a unit test suite for that function. The first agent generates user stories based on the provided Python function. By employing behavior-driven development principles, this agent ensures that the generated user stories provide a concrete structure that guides the subsequent unit test generation process.

System prompt for User Story Writer Agent is setting up context for agent:"You are experienced python developer that writes user stories for a given Python method".

With a prompt for User Story Writer Agent (Fig. 2), we are guiding the agent to write a series of BDD-styled user stories in Gherkin [7] format for a given method, ensuring to cover edge cases. These user stories (Fig.3) will guide the Unit Test Writer Agent to get general structure and cover specified edge cases in order to generate better tests.

By generating user stories and scenarios, we enhance the understanding of legacy code by practically presenting user requirements. Additionally, we are improving the code documentation with the goal of facilitating future work.

B. Unit Test Writer Agent

Following the generation of user stories, the second agent creates unit tests that correspond to the given method and

```
Feature: Undulating Sequence Checker

  As a user,
  I want to determine if a sequence is undulating,
  So that I can verify patterns in my data.

  Scenario: Sequence of length less than or equal to 2
    Given the sequence "[]"
    When I check if it is undulating
    Then the result should be "False"

  Scenario: Sequence of length 2
    Given the sequence "[1, 2]"
    When I check if it is undulating
    Then the result should be "False"

  Scenario: Undulating sequence with three elements
    Given the sequence "[1, 2, 1]"
    When I check if it is undulating
    Then the result should be "True"

  ...

  Scenario: Large undulating sequence
    Given the sequence "[1, 2, 1, 2, 1, 2, 1, 2, 1]"
    When I check if it is undulating
    Then the result should be "True"

  Scenario: Complex non-undulating sequence
    Given the sequence "[0, 1, 0, 1, 0, 2, 0]"
    When I check if it is undulating
    Then the result should be "False"
```

Fig. 3. Example of a generated user story to check if the sequence is undulating. The agent generated a clear requirement of what the user expected and gave a comprehensive scenario list covering wide use cases and edge cases. We shortened the example for space.

```
code_writer_prompt = f"""Based on the following user stories:
\n{user_stores}\n
generate comprehensive unit tests suit to validate the behavior
of the method:\n{method_code}\n\n
Make sure to cover all edge cases, and to have full code coverage.
Save response to a file {task_id}.py
Format of the respond: {respond}"""
```

Fig. 4. Unit Test Writer prompt

previously generated user stories.

We are giving the agent instructions (Fig. 4) to follow user stories, generate comprehensive unit test 1 suits and cover edge cases that exist in code. Utilizing advanced prompts, this agent formulates initial unit test code, which may not always be fully functional.

C. Code Executor Agent and Feedback Loop

The Code Executor Agent is responsible for running the code that the Unit Test Writer Agent produces and creating a feedback loop for the Unit Test Writer Agent. Code Executor Agent is written using Autogen's Code Executor, which creates a temporary file that will be executed. The response of this agent indicates the status of the executed file and if there are

```
class TestIsUndulating(unittest.TestCase):
    def test_empty_sequence(self):
        self.assertFalse(is_undulating([]))
    def test_sequence_length_two(self):
        self.assertFalse(is_undulating([1, 2]))
    def test_undulating_three_elements(self):
        self.assertTrue(is_undulating([1, 2, 1]))
    def test_undulating_with_more_elements(self):
        self.assertTrue(is_undulating([2, 3, 2, 3,
        ↪  2]))
    def test_non_undulating_larger_sequence(self):
        self.assertFalse(is_undulating([1, 2, 1, 3,
        ↪  1, 2]))
    def test_repeating_undulating_pattern(self):
        self.assertTrue(is_undulating([5, 6, 5, 6,
        ↪  5, 6]))
    def test_identical_elements(self):
        self.assertFalse(is_undulating([7, 7, 7,
        ↪  7]))
    def test_edge_case_identical_three(self):
        self.assertFalse(is_undulating([8, 8, 8]))
    def test_large_undulating_sequence(self):
        self.assertTrue(is_undulating([1, 2, 1, 2,
        ↪  1, 2, 1, 2, 1]))
    def test_complex_non_undulating_sequence(self):
        self.assertFalse(is_undulating([0, 1, 0, 1,
        ↪  0, 2, 0]))
```

Listing 1: Example of generated tests for undulating function. Tests cover all scenarios provided by user stories and ensure that functionality works and that there is high code and branch coverage.

any errors in the execution. In our case, the response is a test report with passed and failed tests or errors in execution. The Agent creates a feedback to Unit Test Writer Agent by providing output of test execution. If any of the test fails, the Unit Test Writer Agent incorporates this feedback and refines the unit tests iteratively until test become green.

IV. Use Case Study on MBPP Dataset

While analyzing different datasets for code generation and unit test generation, we noticed that Google's MBPP dataset [1] had room for improvement in the unit test section. MBPP dataset consists of 1000 examples of prompts for a Python problem. Each task consists of a prompt that is input for LLM, code example that LLM should generate and a test example that verifies the solution. We noticed that all tests are assert statements that verify the problem. When we run statistical analyses on dataset, we measured the overall statement coverage of 97%, while overall branch coverage is 92%. We had a closer look at branch coverage per file and found more than 100 examples that have less than 90% branch coverage, including 67 example less then 80% branch coverage and 25 examples having 50% branch coverage. To improve examples with insufficient branch coverage, we engaged our agentic system.

Our system took 100 examples with coverage less than 90% from MBPP dataset, taking only a code part as input to the system. As a base model we used OpenAi's ChapGpt4o-mini, with temperature set to zero. For measuring the pass rate and overall coverage we run python coverage module to generate appropriate reports.

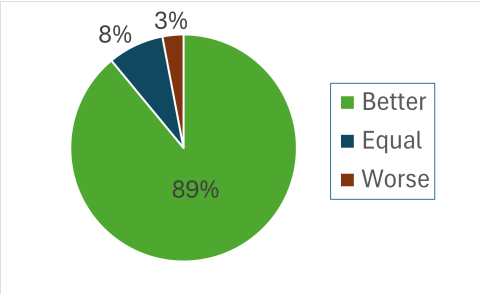

Fig. 5. Statisical results of improved branch coverage on original dataset and after running AI Agents system.

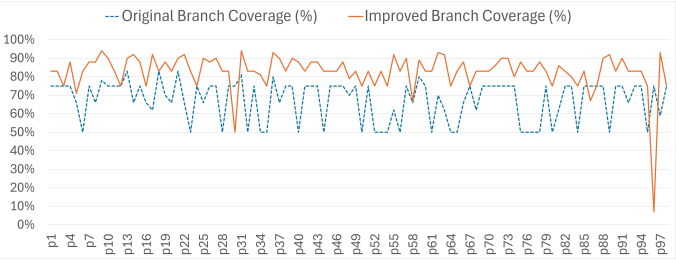

Fig. 6. Original branch coverage (in blue) and AI Agenet unit test breach coverage (in orange)

V. Results Discussion

After running AI Agents, we got better or equal branch coverage in 96/100 examples (Fig. 5).

While running the AI agent one case generated an infinite loop, 3 cases had lower branch coverage, and 8 examples had same branch coverage as original (Fig. 6). In two examples it was due to a minor errors in code itself, such as not closing quotation mark and in one example LLM confused build-in python function with code example. Errors in generated code can happen because LLM is trained to generate the next tokens in natural languages first. We assume that with more iteration in the feedback loop and better underlying models, these can be resolved. We observed a test pass rate of 85%, which, although relatively high, indicates significant room for improvement. This limitation arises from the constraints of the underlying model and the automation process, which currently operates without human intervention. By incorporating additional feedback into the AI agent system and allowing for potential human oversight, we can enhance the pass rate with the goal of achieving 100%. Our analysis particularly focused on branch coverage without human interaction with the system.

Our system made improvements to median branch coverage from 75% to 84% with a increase of 11%. Average branch coverage is from 67% to 83% with an increase of 23%.

With agentic workflow we gotten a higher branch coverage without human intervention. Whole generaion of 100 examples cost less then 0.2 dollars per current OpenAI prices, and executed under 5 minutes.

Results show that implementing AI agents with the proposed architecture generates tests with high branch coverage, saving time and enabling researchers and developers to enhance their work.

VI. Conclusion and Future Work

The integration of AI agents into the unit test generation process showcases the potential for achieving higher code quality and improved datasets for training language models. By automating the generation and refinement of unit tests, we contribute to more reliable software development practices. The implications of this methodology extend to various coding frameworks and programming languages, further emphasizing the relevance of robust datasets in training advanced AI models. This paper highlights the transformative impact of AI agents in generating effective unit tests and improving datasets. Our findings advocate for the adoption of AI-driven methodologies to enhance software testing processes, ultimately leading to more reliable models and applications. Future work will expand our framework to support additional programming languages and explore the incorporation of user feedback in the test-generation process. Also, future work will focus on increasing test quality and test coverage to resolve identified issues. In addition, we plan to develop a system that fully generates both code, unit tests and acceptance tests for specified functionalities while adhering to Behavior-Driven Development (BDD) principles, starting with generating user stories then generating tests followed by generation of code. In this paper, we have explored a subset of functionalities as part of our broader objectives.

References

[1] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[2] K. El Haji, C. Brandt, and A. Zaidman, "Using github copilot for test generation in python: An empirical study," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, ser. AST '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 45–55. [Online]. Available: https://doi.org/10.1145/3644032.3644443

[3] D. M. Zapkus and A. Slotkienė, "Unit test generation using large language models: A systematic literature review," *Vilnius University Open Series*, p. 136–144, May 2024. [Online]. Available: https://www.journals.vu.lt/open-series/article/view/35383

[4] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," 2024. [Online]. Available: https://arxiv.org/abs/2402.09171

[5] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "Opendevin: An open platform for ai software developers as generalist agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.16741

[6] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," in *COLM*, 2024.

[7] Cucumber, "Gherkin reference - cucumber documentation." [Online]. Available: https://cucumber.io/docs/gherkin/reference/