Exploring Claude AI's Capability in Unit Test Generation: An Investigative Study

2426234
University of The Witwatersrand
School of Electrical and Information Engineering
ELEN4010: Software Development |||
08 April 2024

***Abstract*—With the recent advancements in Artificial Intelligence, there is growing interest in exploring how AI can enhance various aspects of the software development life cycle. In this study, a dataset of Python programs known as HumanEval was used to prompt a Large Language Model (Claude AI) with the task of generating unit tests for these programs. Three key metrics were employed to evaluate the effectiveness of the generated unit tests: testing outcomes, branch coverage, and execution time. A Python plugin tool called `pytest-cov` was used as the evaluation tool, and the testing framework promoted to the AI was Pytest. The testing outcomes resulted in a 45.00% pass rate. The branch coverage achieved by the generated unit tests averaged an impressive 96.11%. Additionally, the average execution time of a single unit test was found to be 0.08 seconds, meeting the commonly accepted criterion of being under 1.00 seconds.**

***Index Terms*—Artificial Intelligence, Software Development, Large Language Model, Unit Testing, Branch Coverage, Execution Time**

1. Introduction

With the increased growth of programming languages over the past two decades [1], the demand for robust software testing has increased. Unit testing, an aspect of software testing, helps ensure software quality. This process involves testing individual units or modules of code within a system [2]. Traditionally, unit testing has been a manual process, relying on human programming [2]. In 2003, Michael Olan stated in a publication that a significant drawback of manual unit testing is its lack of automation [3]. Fast forward nearly two decades, and with recent advancements in artificial intelligence (AI) [4], an intriguing question emerges: can AI algorithms effectively generate meaningful unit tests, thus streamlining workflow and empowering developers to more efficiently detect faults?

This report explores this compelling question by tapping into the capabilities of readily available AI, particularly Large Language Models (LLMs). By leveraging an LLM, this study seeks to assess the effectiveness of unit test generation by AI across key metrics such as branch coverage, test execution success, and testing runtime.

The report is structured as follows: Section 2 provides a brief background on unit testing and the LLM used. Section 3 delves into existing literature and related work in the field. The methodology employed for this investigation is outlined in Section 4. The particular LLM used in the study is Anthropic's Claude AI. Section 5 presents the analysis of results, evaluating the performance of AI-generated unit tests. Finally, Section 6 offers a conclusion summarising the findings of the study and discussing potential implications for future research and industry practices.

2. Background

This section outlines the role of unit testing in software development, how unit tests can be measured and lastly, an introduction to an LLM known as Claude AI.

A. Unit Testing

Within a software development cycle, validation is a critical process aimed at ensuring that a program operates as intended [3]. This validation often takes the form of software testing, with unit testing being a prominent method. Unit testing involves exercising a "unit" of code in isolation and comparing the actual output with the expected results [3]. When a unit test fails, developers can pinpoint the specific area of code that is malfunctioning and rectify it accordingly [5].

Unit test quality is commonly assessed using coverage metrics, which indicate the extent to which a test suite exercises the source code [5]. Among these metrics, code coverage and branch coverage are the most popular [5]. In this report, the focus is on branch coverage, known for providing more precise results compared to code coverage [5]. Branch coverage scrutinises control structures, such as "if" and "switch" statements, revealing how many control structures are traversed by at least one test in a testing suite [5]. Additionally, this investigation evaluates the duration of unit tests. A general consensus is that a duration between 0.01 seconds to 1 second for a unit test in isolation is considered favourable [6].

B. Claude AI

Large Language Models (LLMs) serve as AI models trained on extensive text data, enabling them to generate human-like text, respond to questions, and perform various language-related tasks with remarkable accuracy [7]. These models can be fine-tuned for specific applications, adapting their capabilities accordingly [8]. For instance, LLMs can be tailored to function as chat-bots (ChatGPT) or even as code-generating systems (Codex) [8].

In recent years, there has been a notable advancement in LLM technology, leading to new possibilities within the software engineering domain [7]. These models are trained on vast repositories of code-related datasets, granting them the ability to interpret and process code-related prompts to a considerable extent. Notable LLMs as of early 2023 include OpenAI's GPT-3.5, DeepMind's Chinchilla, Google's Palm, and Anthropic's Claude [8].

This investigation focuses specifically on Anthropic's Claude AI. Notably, as explored further in Section 3, no prior related work to related to this investigation has utilised Claude, thus highlighting the unique contribution of this study.

3. Related Work

Provided within this section contains an overview of various investigations that have thus been conducted on LLM unit test generation.

A notable study by V. Guilherme and A. Vincenz [9] investigates the quality of Java unit tests generated by an OpenAI's GPT3.5 LLM model. The study aimed to evaluate the efficiency and effectiveness of the generated unit test set using metrics such as code coverage and mutation test scores. The authors selected thirty-three programs used by other researchers in automated test generation as a baseline for comparison. For each program, thirty-three unit test sets are generated automatically without human interference by adjusting OpenAI's API parameters. The evaluation included metrics such as code line coverage, mutation score, and the success rate of test execution. The results indicated that OpenAI's GPT3.5 LLM test sets demonstrate similar performance across all evaluated aspects compared to traditional automated Java test generation tools used in previous research.

Another study conducted by M. Schafer, S. Nadi, A. Eghbali, and F. Tip [10], presents a large-scale empirical evaluation of the effectiveness of LLMs in automated unit test production without requiring additional training or human effort. The study proposed a method where LLMs are provided prompts containing the signature and implementation of a function under test, along with usage samples extracted from the documentation. If the generated test fails, the authors attempted to rectify the issue by prompting the model with the failed test and error message. The study utilised TestPilot, an adaptive LLM-based JavaScript test generation tool designed to create unit tests for functions within a project's API. The evaluation of TestPilot employed OpenAI's GPT3.5-Turbo LLM. Notably, the tests produced by TestPilot achieved a median statement coverage of 70.2% and branch coverage of 52.8%.

Additionally, a study by M. L. Siddiq, J. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes [11] investigates the feasibility of using code generation models like GitHub Copilot for unit test generation without fine-tuning for strongly typed languages like Java. The investigation assessed the performance of three models (Codex, Open AI's GPT-3.5-Turbo, and StarCoder) on two benchmarks (HumanEval and Evosuite SF110) considering metrics such as compilation rates, test correctness, coverage, and test smells. Results revealed challenges in achieving high test coverage and the presence of test smells in the generated tests, suggesting areas for improvement in utilising code generation models for unit test generation tasks.

In continuation of this line of research, this investigation aims to test the unit test generation capabilities of Claude AI. Unlike previous investigations, branch coverage, test outcomes, and time to run unit test metrics are evaluated. Additionally, it's worth noting that while previous studies primarily focused on Java programs, this investigation evaluates a Python dataset. While inspired by the related works mentioned above, this investigation approach differs in its focus on assessing Claude AI's performance using the stated set of metrics.

4. Methodology

The methodology employed in this study involved selecting a set of Python programming problems from a publicly available dataset called HumanEval (Hand-Written Evaluation Set). Utilising Claude AI as the chosen LLM, a Python script, coupled with Claude's API, iterated through each Python program. For each program, a prompt was constructed based on the Python input, and unit tests were generated using Claude's responses.

To evaluate the automatically generated LLM tests, a bash script was utilised. This script iterated through each program once more, leveraging readily available tools to assess the outcomes, branch coverage, and execution time of the generated unit tests. The process outlined above provides a broad overview of the methodology, with detailed explanations provided in the subsections below.

A. LLM Selection

As discovered in Section 3, all the related work analysed make use of OpenAI's ChatGPT. To deviate from the norm and provide additional insight within this field, a different AI model was chosen. Anthropic's LLM, Claude AI, was selected for this investigation. A major factor in utilizing this LLM, aside from its distinctiveness, is its availability of an API, which streamlined the investigation by reducing the time required for generating responses. As of 2024, Claude 3.0 is available to users via Anthropic's website [12]. However, for the purposes of this investigation, Claude 2.0 was utilised, leveraging a free five-dollar API coupon.

B. Data Selection

Forty Python programming problems were extracted from a publicly available dataset called HumanEval. Despite the dataset containing one hundred and sixty-four written programming problems, only forty samples were utilised in this study. Each programming problem includes a description of the intended functionality of the code, along with a canonical solution.

For this investigation, the canonical solutions were employed as inputs to the LLM, serving as prompts for generating unit tests. This approach ensured that the generated tests were aligned with the intended functionality of the code, as described in the dataset.

C. Prompt Construction

The process of constructing prompts for unit test generation was facilitated by a Python script designed to extract forty Python programming problems from the HumanEval dataset. These problems were then stored in a directory corresponding to their respective program names. Subsequently, another Python script iterated through each extracted Python program.

During each iteration, a prompt was dynamically constructed. The prompt consisted of the following instruction:

```
Write unit tests for the following Python
code using Pytest. First, provide the
function followed by the test code. The
code is: program_code.
```

Here, `program_code` represents the specific Python programming problem being processed.

It is worth noting that although the program code was passed into the Large Language Model (LLM), the model was instructed to reproduce the code it received. This decision stemmed from the observation that the function names were omitted while extracting the data from the HumanEval dataset. As a result, attempting to run the Python script without function names would lead to errors. Thus, by asking the LLM to reproduce the original code, the process of manually editing each input prompt for evaluation purposes was simplified.

D. Unit Test Generation

In each iteration, as discussed in Section 43, the constructed prompt was passed to Claude 2.0 through its API. The resulting response from each iteration was stored in a new file within each program's repository, with an additional suffix of `_test` appended to the filename. Subsequently, all response files were aggregated and stored in a database.

Upon storage, the response files underwent manual editing to remove the original code. Additionally, the initial program files were updated with the generated code. This refactoring process was performed manually to ensure accuracy and consistency. Both the original and refactored code versions are stored in the database for reference and analysis.

E. Evaluation Framework & Metrics

The unit testing framework utilised in this investigation is Pytest, a mature and full-featured testing framework known for its versatility, supporting both small and large-scale functional testing [13]. To assess the effectiveness of the generated unit tests, Pytest was chosen for its robust testing capabilities.

To measure the quality of the generated unit tests, a Python plugin called `pytest-cov` was employed to produce coverage reports [14]. While `pytest-cov` can generate both code coverage and branch coverage reports, only the branch coverage functionality was utilised for this study [14]. Within the generated coverage report, metrics such as testing outcomes (i.e., the number of passed and failed tests) and the time taken to execute the tests are also included, providing a comprehensive assessment of the unit tests effectiveness.

F. Evaluation Procedure

The evaluation of the generated unit tests was conducted using a bash script to run the `pytest-cov` plugin. This script iterated through each program's folder and executed the plugin, capturing the output results. The results were stored in a `.txt` file and also in the database mentioned earlier. A detailed analysis of the evaluation results is provided in Section 5.

G. Experimental Setup

Table I below outlines the key components and specifications used during the study.

TABLE I: Experimental Setup Specifications

LLM	Claude 2.0
Python Kernel Version	3.11.7
Pytest Version	8.1.1
Processor	2.3 GHz Dual-Core Intel Core i5
RAM	8GB 2133 MHz LPDDR3

5. Results and Evaluation

The results of the investigation are presented and evaluated within this section. The branch coverage and test outcomes of the generated unit tests are analysed to assess the effectiveness of using Claude AI for automated unit test generation. Additionally, the performance of the unit tests in terms of execution time is evaluated.

A. Test Outcome Results

Of the forty programs for which unit tests were generated by Claude, two of the testing files produced errors when attempting to run the Pytest framework. Among the remaining thirty-eight programs, eighteen programs passed all unit tests, sixteen programs partially passed, and four programs failed all tests. "Partially passed" refers to cases in which some tests pass while others fail. Overall, the pass rate of the generated unit tests stands at 45.00%, reflecting a mixed but promising performance of Claude AI in automated unit test generation. Table II and Figure 1 provides a clear overview of the distribution of test outcomes.

TABLE II: Summary of Test Outcomes.

Outcome	**Number of Programs**
Passed all tests	18
Partially passed	16
Failed all tests	4
Errors encountered	2

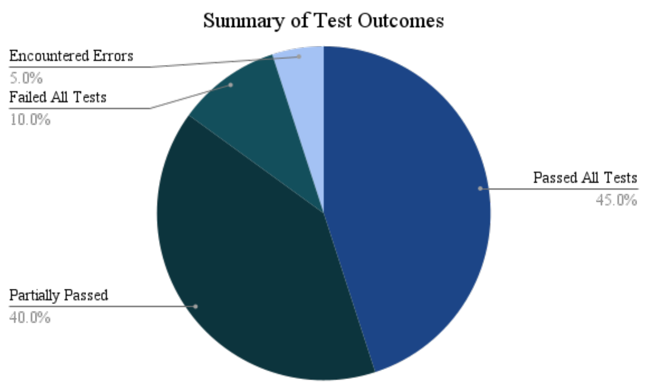


Fig. 1: Graphical Representation of Test Outcomes.

B. Branch Coverage & Execution Time Results

The branch coverage of the generated unit tests was measured using `pytest-cov`. On average, the branch coverage across the thirty-eight programs stands at an impressive 96.11%. Additionally, the average runtime of each program's unit testing process was found to be 0.23 seconds. Lastly, the average individual runtime for each unit test within a program is calculated at 0.08 seconds with an overview of each programs individual runtime shown in Figure 2.

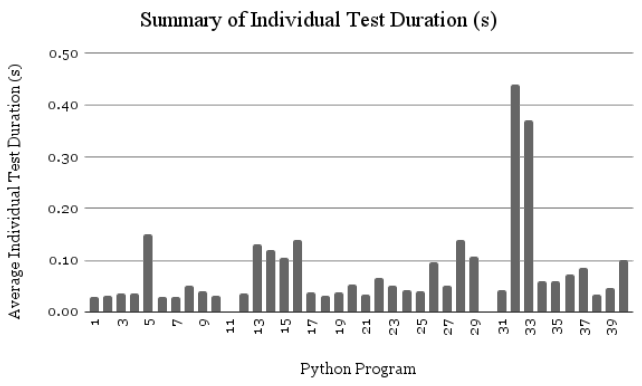


Fig. 2: Graphical Representation of Individual Unit Test Runtime.

C. Evaluation

Now that the results have been tabulated and mentioned, an analysis of Claude AI's performance is discussed. Firstly, the test outcomes reveal a mixed performance of Claude AI. While the pass rate of 45.00% indicates that nearly half of the generated unit tests successfully validate the functionality of the code, there are notable areas for improvement. The occurrence of errors in two testing files and the failure of four programs to pass any tests highlight issues or limitations in the generated test cases.

Despite these challenges, the branch coverage achieved by the generated unit tests is commendable, averaging at an impressive 96.11%. This indicates that the generated tests effectively exercise various control flow paths within the code, providing comprehensive testing of the program's logic and decision-making branches. The high branch coverage suggests that Claude AI can produce test cases that thoroughly evaluate the code's behaviour under different conditions.

However, it's crucial to recognise that branch coverage alone cannot be the sole metric used to assess test correctness [5]. While high branch coverage is indicative of thorough testing, it does not guarantee that all possible scenarios and edge cases are adequately addressed [5]. For instance, certain branches may still remain untested, or the tests may not accurately capture the intended behaviour of the code in certain scenarios.

Moreover, the efficient execution time of the unit testing process, with an average runtime for each unit test at 0.08s. Quick execution times facilitate rapid feedback during the development process, enabling developers to iterate and refine their code more efficiently. Generally, unit tests should individually not take more than one second [6]. In this regard, the criteria are met.

6. Threats to Validity

Threats to validity refer to factors that may affect the accuracy, reliability, and generalisations of a study's findings [15]. In this section, potential threats to the validity of the investigation are discussed as well as strategies to mitigate them in future research investigations.

A. Internal Threats

Internal threats pertain to factors within the study that may impact the validity of the results [15]. These threats include:

1) **Sampling Bias**: The selection of Python programs from the HumanEval dataset may introduce bias if the chosen programs do not represent a diverse range of code structures and functionalities. To mitigate this threat, efforts can be made to randomly select programs from the dataset and ensure a varied representation. For this investigation, no randomness in selecting of programs was conducted.
2) **Testing Framework Limitations**: The use of the Pytest framework for evaluating unit tests may introduce limitations or biases inherent to the framework itself. To address this, alternative testing frameworks could be considered for validation, or multiple frameworks could be used in conjunction to validate the results.
3) **Dependency on Claude AI**: The reliance on Claude AI as the primary tool for unit test generation introduces a potential internal threat, as the performance and capabilities of Claude AI may influence the outcomes of the study. To mitigate this threat, the experiment could be replicated using alternative LLMs.

B. External Threats

External threats relate to factors outside the study that may impact the generalisations of the findings [15]. Threats include:

1) **Generalisations to Other Languages**: The study focuses on Python programs, which may limit the generalisations of the findings to other programming languages. To address this, future research could replicate the experiment using different programming languages to assess any deviance's in the results.
2) **Limited Dataset**: The use of a specific dataset (HumanEval) may limit the external validity of the findings if the dataset does not accurately represent real-world programming scenarios. To mitigate this, future investigations could utilise a broader range of datasets to ensure a more comprehensive evaluation.
3) **Tool Dependency**: The reliance on specific tools and technologies, such as Claude AI and Pytest, may restrict the applicability of the findings to environments where these tools are not available or commonly used. To enhance external validity, the experiment could be replicated using different tools and environments to assess the robustness of the results.

7. Conclusion

In conclusion, this investigation assesses the LLM, Claude AI's capability in automatically generating unit tests for Python programs. Through the utilisation of the Pytest framework and the `pytest-cov` Python plugin, three key metrics were evaluated: testing outcomes, branch coverage, and execution time of unit tests.

While the analysis of the testing outcomes revealed a mixed performance of Claude AI, with notable areas for improvement highlighted by the occurrence of errors in some testing files and the failure of certain programs to pass any tests, the investigation also underscored several strengths. Particularly, Claude AI demonstrated commendable branch coverage, averaging an impressive 96.11% indicating its ability to effectively explore various control flow paths.

Furthermore, the efficient execution time of the unit testing process, with an average runtime of 0.08 seconds for each unit test, is indicative of Claude AI's potential to facilitate rapid feedback during a software development life cycle.

Overall, while there are areas for improvement, such as addressing errors and enhancing the pass rate of generated tests, the investigation sheds light on Claude AI's capabilities in automating unit test generation. As Claude AI continues to evolve and improve with improved training sets, it holds significant potential to streamline and enhance the software development process by automating unit testing tasks.

References

[1] P. Chakraborty, R. Shahriyar, and A. Iqbal, "Empirical analysis of the growth and challenges of new programming languages," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, (Milwaukee, WI, USA), pp. 191–196, 2019.

[2] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[3] M. Olan, "Unit testing: test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[4] W. Wang and K. Siau, "Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda," *Journal of Database Management (JDM)*, vol. 30, no. 1, pp. 61–79, 2019.

[5] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[6] J. Prickett, "How long should it take to run unit tests?." https://medium.com/@jakeprickett/how-long-should-it-take-to-run-unit-tests-5decd79679c5, July 2019. Accessed: 02 April 2024.

[7] F. Lin, D. J. Kim, Tse-Husn, and Chen, "When llm-based code generation meets the software development process," 2024.

[8] A. Korinek, "Language models and cognitive automation for economic research," Working Paper 30957, National Bureau of Economic Research, February 2023.

[9] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, pp. 15–24, 2023.

[10] 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*, vol. 50, no. 1, pp. 85–105, 2024.

[11] M. L. Siddiq, J. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Exploring the effectiveness of large language models in generating unit tests," *arXiv preprint arXiv:2305.00418*, 2023.

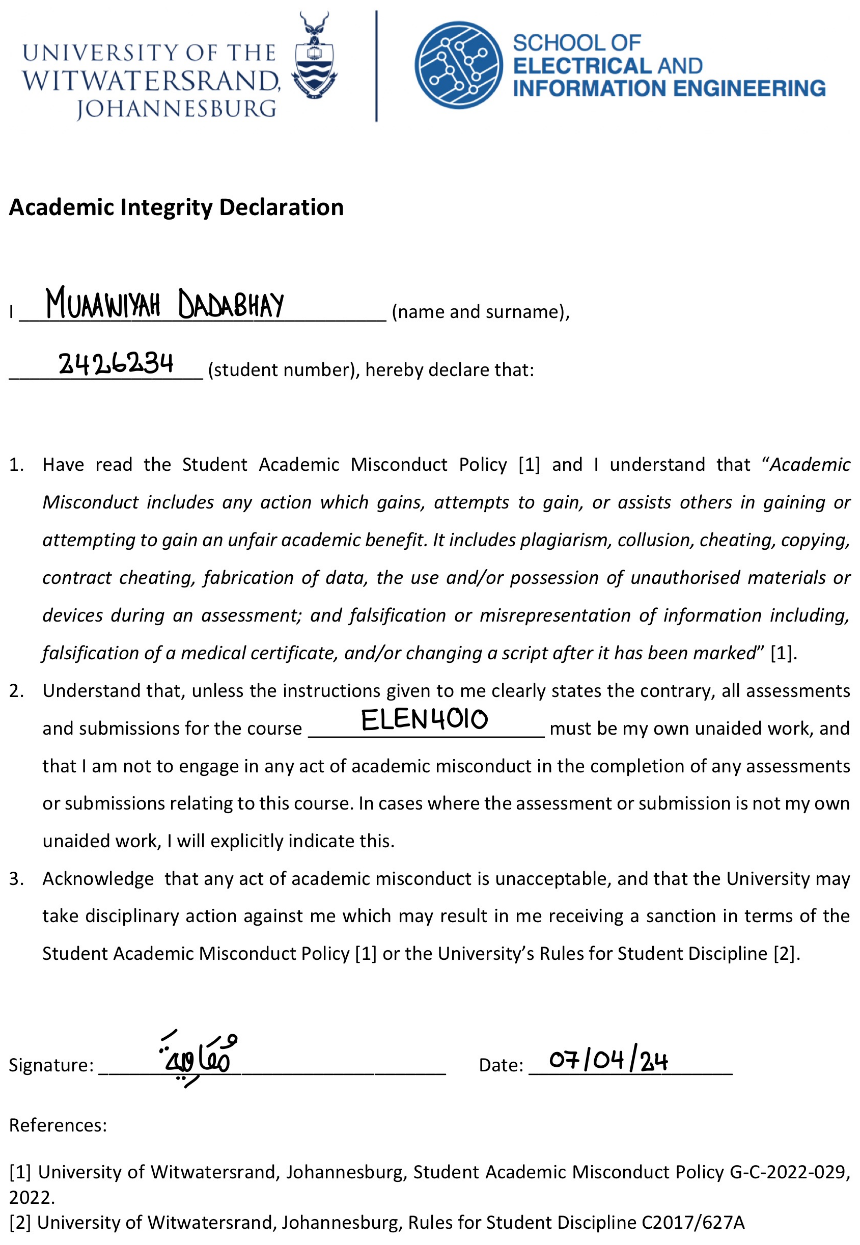
[12] "Claude." https://www.anthropic.com/claude, n.d. Anthropic.

[13] B. Oliveira, *pytest Quick Start Guide: Write better Python code with simple and maintainable tests*. Packt Publishing Ltd, 2018.

[14] pytest-cov developers, "pytest-cov documentation," n.d. Accessed: April 2, 2024.

[15] R. Malhotra and M. Khanna, "Threats to validity in search-based predictive modelling for software engineering," *IET Software*, vol. 12, no. 4, pp. 293–305, 2018.

APPENDIX A
ACADEMIC INTEGRITY DECLARATION



UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG | SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

Academic Integrity Declaration

I Muaawiyah Dadabhay (name and surname),

2426234 (student number), hereby declare that:

1. Have read the Student Academic Misconduct Policy [1] and I understand that "*Academic Misconduct includes any action which gains, attempts to gain, or assists others in gaining or attempting to gain an unfair academic benefit. It includes plagiarism, collusion, cheating, copying, contract cheating, fabrication of data, the use and/or possession of unauthorised materials or devices during an assessment; and falsification or misrepresentation of information including, falsification of a medical certificate, and/or changing a script after it has been marked*" [1].
2. Understand that, unless the instructions given to me clearly states the contrary, all assessments and submissions for the course ELEN4010 must be my own unaided work, and that I am not to engage in any act of academic misconduct in the completion of any assessments or submissions relating to this course. In cases where the assessment or submission is not my own unaided work, I will explicitly indicate this.
3. Acknowledge that any act of academic misconduct is unacceptable, and that the University may take disciplinary action against me which may result in me receiving a sanction in terms of the Student Academic Misconduct Policy [1] or the University's Rules for Student Discipline [2].

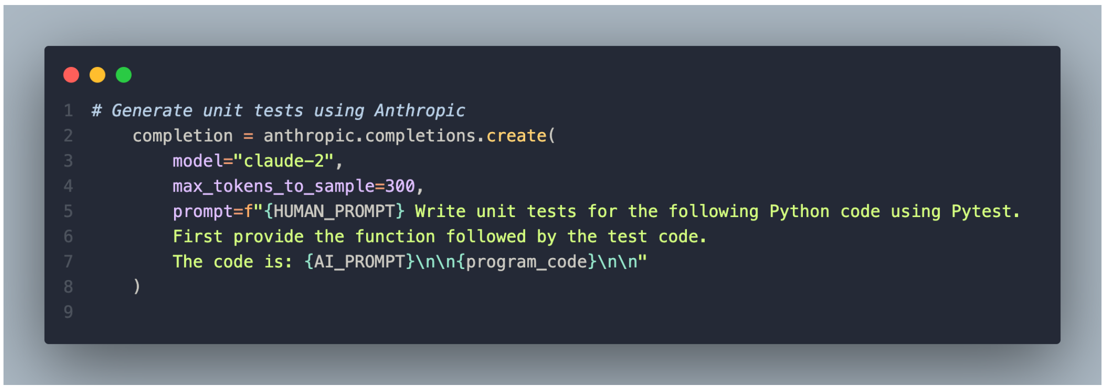
Signature: [illegible] Date: 07/04/24

References:

[1] University of Witwatersrand, Johannesburg, Student Academic Misconduct Policy G-C-2022-029, 2022.
[2] University of Witwatersrand, Johannesburg, Rules for Student Discipline C2017/627A

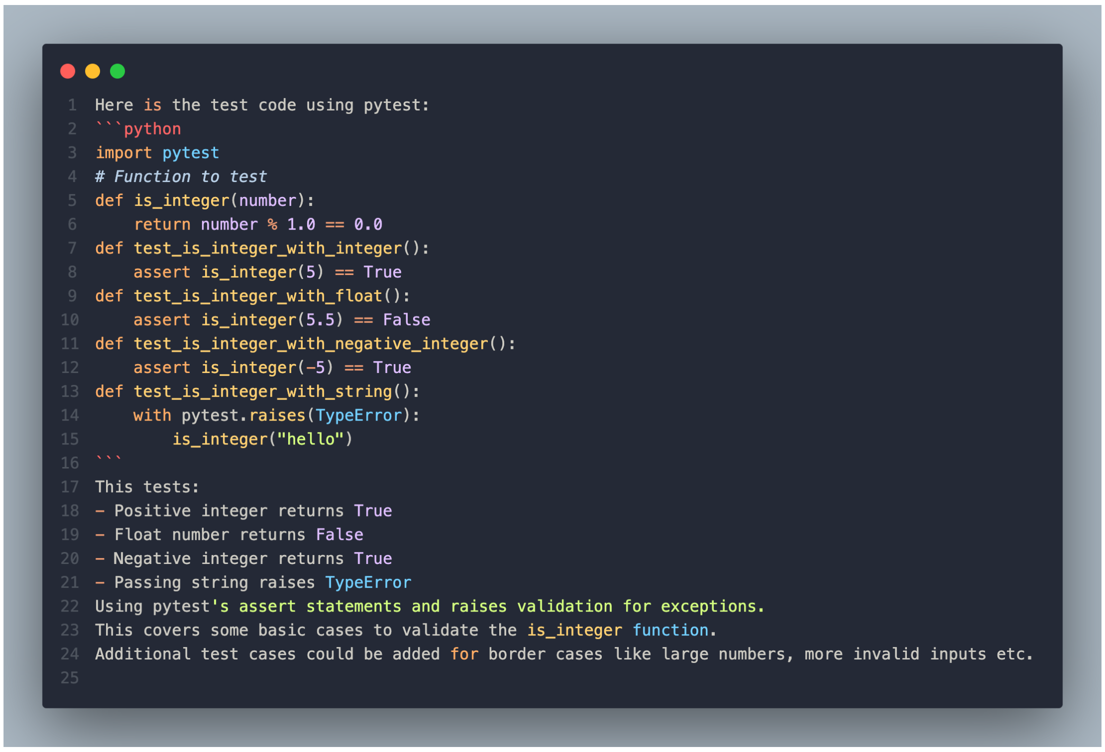
Fig. 3: Academic Declaration Form.

APPENDIX B
LARGE LANGUAGE MODEL PROMPTS & RESPONSES



```python
# Generate unit tests using Anthropic
    completion = anthropic.completions.create(
        model="claude-2",
        max_tokens_to_sample=300,
        prompt=f"{HUMAN_PROMPT} Write unit tests for the following Python code using Pytest.
        First provide the function followed by the test code.
        The code is: {AI_PROMPT}\n\n{program_code}\n\n"
    )

```

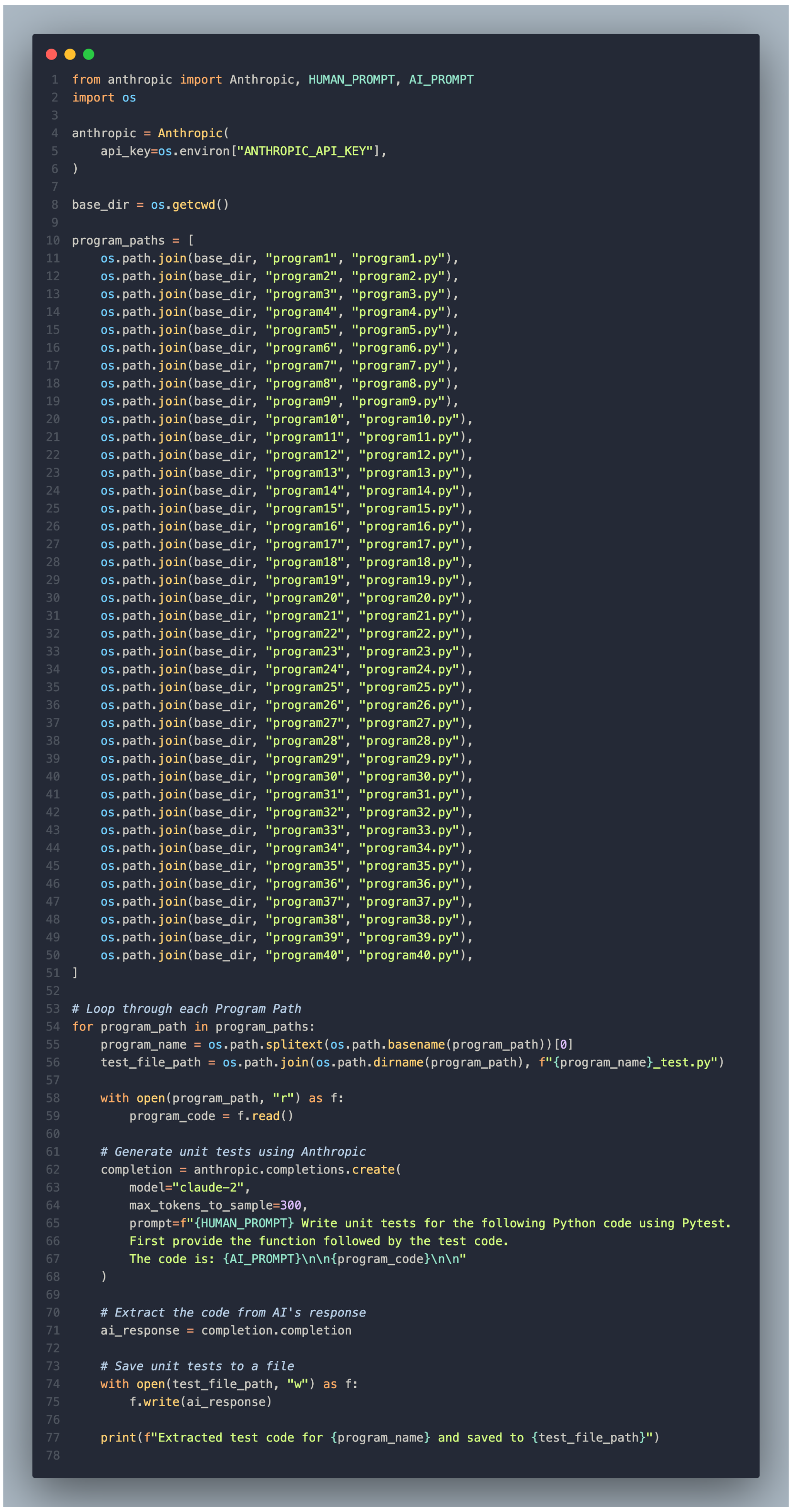
Fig. 4: Code snippet showing the prompt used to pass to the LLMs API.



~~~
Here is the test code using pytest:
```python
import pytest
# Function to test
def is_integer(number):
    return number % 1.0 == 0.0
def test_is_integer_with_integer():
    assert is_integer(5) == True
def test_is_integer_with_float():
    assert is_integer(5.5) == False
def test_is_integer_with_negative_integer():
    assert is_integer(-5) == True
def test_is_integer_with_string():
    with pytest.raises(TypeError):
        is_integer("hello")
```
This tests:
- Positive integer returns True
- Float number returns False
- Negative integer returns True
- Passing string raises TypeError
Using pytest's assert statements and raises validation for exceptions.
This covers some basic cases to validate the is_integer function.
Additional test cases could be added for border cases like large numbers, more invalid inputs etc.

~~~

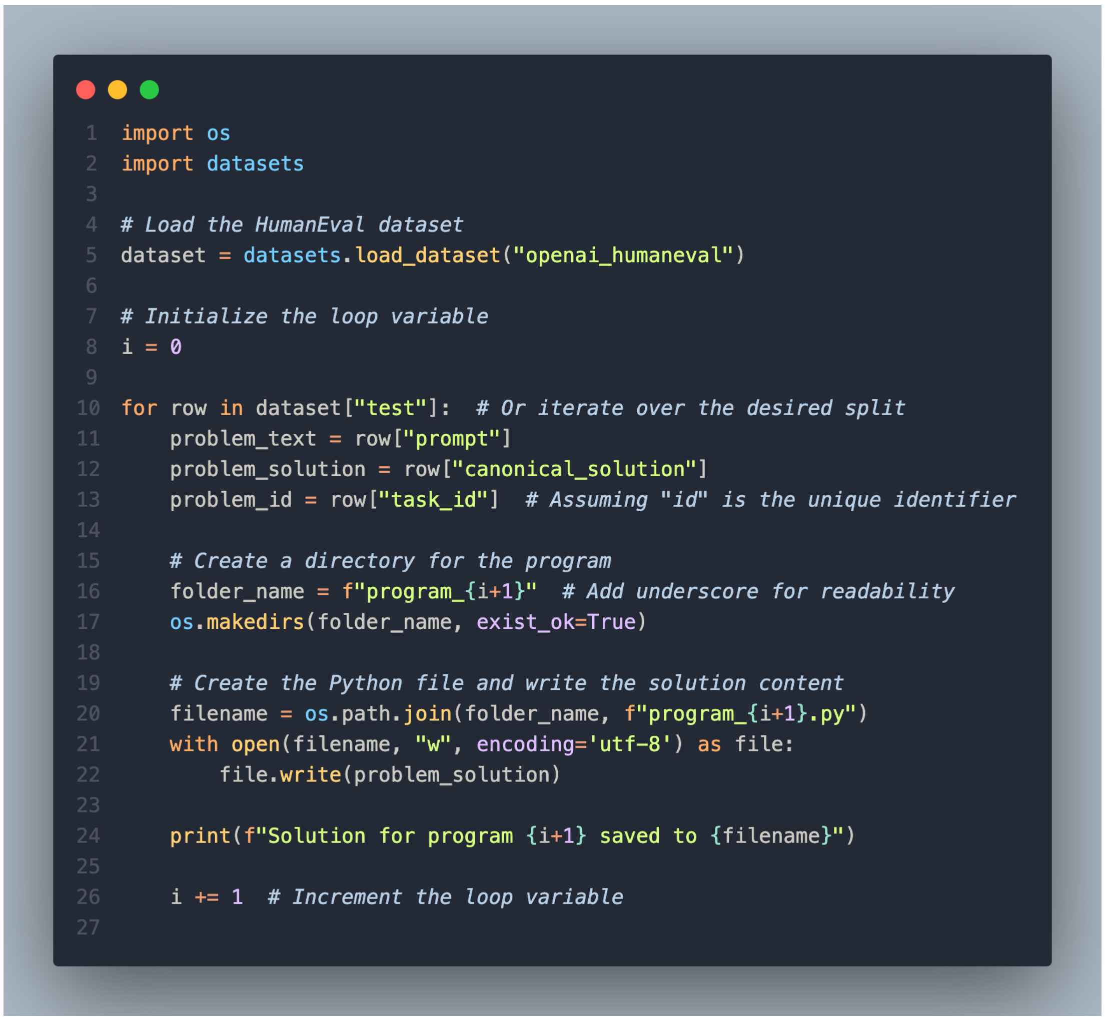
Fig. 5: Code snippet showing a response from the LLM for a single programs Unit Tests.

Appendix C
Scripts Used Within The Investigation



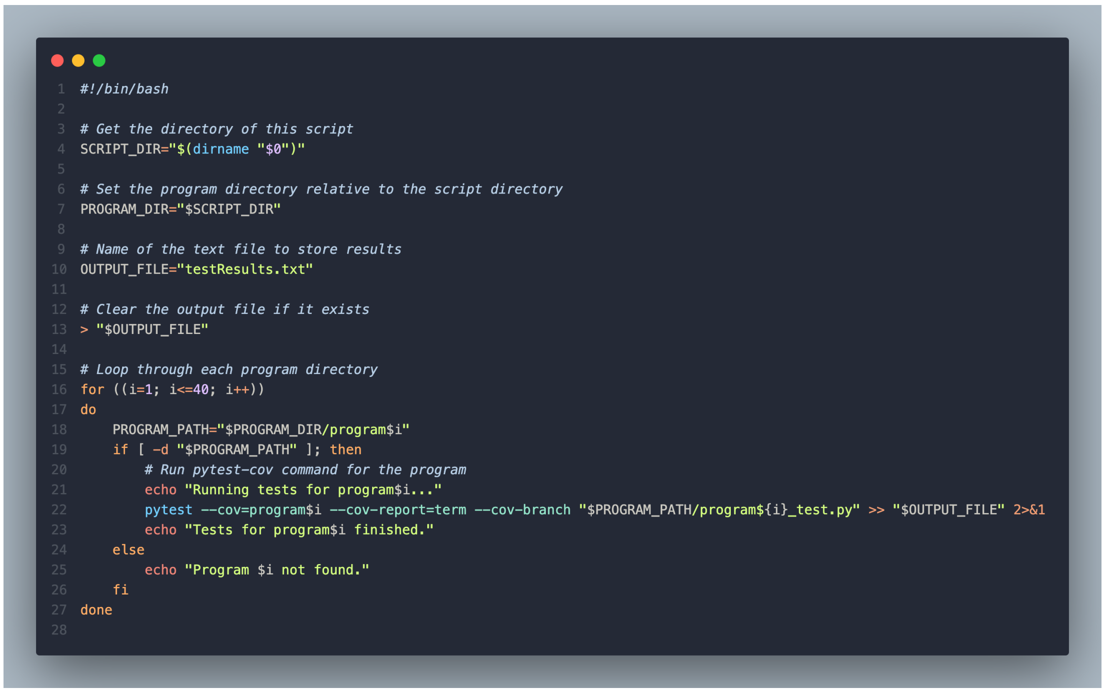
```python
from anthropic import Anthropic, HUMAN_PROMPT, AI_PROMPT
import os

anthropic = Anthropic(
    api_key=os.environ["ANTHROPIC_API_KEY"],
)

base_dir = os.getcwd()

program_paths = [
    os.path.join(base_dir, "program1", "program1.py"),
    os.path.join(base_dir, "program2", "program2.py"),
    os.path.join(base_dir, "program3", "program3.py"),
    os.path.join(base_dir, "program4", "program4.py"),
    os.path.join(base_dir, "program5", "program5.py"),
    os.path.join(base_dir, "program6", "program6.py"),
    os.path.join(base_dir, "program7", "program7.py"),
    os.path.join(base_dir, "program8", "program8.py"),
    os.path.join(base_dir, "program9", "program9.py"),
    os.path.join(base_dir, "program10", "program10.py"),
    os.path.join(base_dir, "program11", "program11.py"),
    os.path.join(base_dir, "program12", "program12.py"),
    os.path.join(base_dir, "program13", "program13.py"),
    os.path.join(base_dir, "program14", "program14.py"),
    os.path.join(base_dir, "program15", "program15.py"),
    os.path.join(base_dir, "program16", "program16.py"),
    os.path.join(base_dir, "program17", "program17.py"),
    os.path.join(base_dir, "program18", "program18.py"),
    os.path.join(base_dir, "program19", "program19.py"),
    os.path.join(base_dir, "program20", "program20.py"),
    os.path.join(base_dir, "program21", "program21.py"),
    os.path.join(base_dir, "program22", "program22.py"),
    os.path.join(base_dir, "program23", "program23.py"),
    os.path.join(base_dir, "program24", "program24.py"),
    os.path.join(base_dir, "program25", "program25.py"),
    os.path.join(base_dir, "program26", "program26.py"),
    os.path.join(base_dir, "program27", "program27.py"),
    os.path.join(base_dir, "program28", "program28.py"),
    os.path.join(base_dir, "program29", "program29.py"),
    os.path.join(base_dir, "program30", "program30.py"),
    os.path.join(base_dir, "program31", "program31.py"),
    os.path.join(base_dir, "program32", "program32.py"),
    os.path.join(base_dir, "program33", "program33.py"),
    os.path.join(base_dir, "program34", "program34.py"),
    os.path.join(base_dir, "program35", "program35.py"),
    os.path.join(base_dir, "program36", "program36.py"),
    os.path.join(base_dir, "program37", "program37.py"),
    os.path.join(base_dir, "program38", "program38.py"),
    os.path.join(base_dir, "program39", "program39.py"),
    os.path.join(base_dir, "program40", "program40.py"),
]

# Loop through each Program Path
for program_path in program_paths:
    program_name = os.path.splitext(os.path.basename(program_path))[0]
    test_file_path = os.path.join(os.path.dirname(program_path), f"{program_name}_test.py")

    with open(program_path, "r") as f:
        program_code = f.read()

    # Generate unit tests using Anthropic
    completion = anthropic.completions.create(
        model="claude-2",
        max_tokens_to_sample=300,
        prompt=f"{HUMAN_PROMPT} Write unit tests for the following Python code using Pytest.
        First provide the function followed by the test code.
        The code is: {AI_PROMPT}\n\n{program_code}\n\n"
    )

    # Extract the code from AI's response
    ai_response = completion.completion

    # Save unit tests to a file
    with open(test_file_path, "w") as f:
        f.write(ai_response)

    print(f"Extracted test code for {program_name} and saved to {test_file_path}")

```

Fig. 6: Code snippet showing the Automatic Unit Test generation utilising Anthropic’s API.



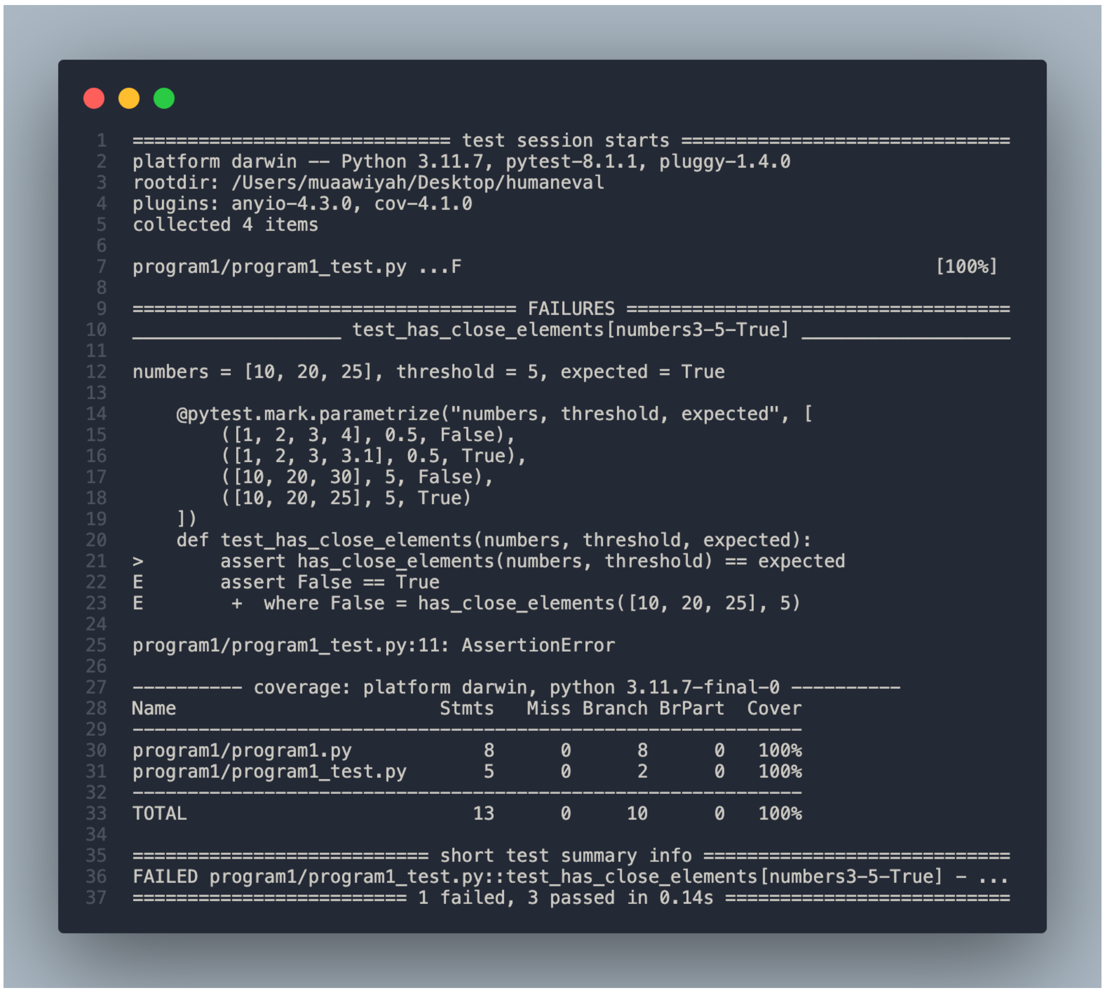
```python
import os
import datasets

# Load the HumanEval dataset
dataset = datasets.load_dataset("openai_humaneval")

# Initialize the loop variable
i = 0

for row in dataset["test"]:  # Or iterate over the desired split
    problem_text = row["prompt"]
    problem_solution = row["canonical_solution"]
    problem_id = row["task_id"]  # Assuming "id" is the unique identifier

    # Create a directory for the program
    folder_name = f"program_{i+1}"  # Add underscore for readability
    os.makedirs(folder_name, exist_ok=True)

    # Create the Python file and write the solution content
    filename = os.path.join(folder_name, f"program_{i+1}.py")
    with open(filename, "w", encoding='utf-8') as file:
        file.write(problem_solution)

    print(f"Solution for program {i+1} saved to {filename}")

    i += 1  # Increment the loop variable

```

Fig. 7: Code snippet showing the extraction of Python programs from the HumanEval dataset.



```bash
#!/bin/bash

# Get the directory of this script
SCRIPT_DIR="$(dirname "$0")"

# Set the program directory relative to the script directory
PROGRAM_DIR="$SCRIPT_DIR"

# Name of the text file to store results
OUTPUT_FILE="testResults.txt"

# Clear the output file if it exists
> "$OUTPUT_FILE"

# Loop through each program directory
for ((i=1; i<=40; i++))
do
    PROGRAM_PATH="$PROGRAM_DIR/program$i"
    if [ -d "$PROGRAM_PATH" ]; then
        # Run pytest-cov command for the program
        echo "Running tests for program$i..."
        pytest --cov=program$i --cov-report=term --cov-branch "$PROGRAM_PATH/program${i}_test.py" >> "$OUTPUT_FILE" 2>&1
        echo "Tests for program$i finished."
    else
        echo "Program $i not found."
    fi
done

```

Fig. 8: Code snippet of a bash script used to run `pytest-cov` for each programs Unit Tests.



```
============================= test session starts ==============================
platform darwin -- Python 3.11.7, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/muaawiyah/Desktop/humaneval
plugins: anyio-4.3.0, cov-4.1.0
collected 4 items

program1/program1_test.py ...F                                           [100%]

=================================== FAILURES ===================================
___________________ test_has_close_elements[numbers3-5-True] ___________________

numbers = [10, 20, 25], threshold = 5, expected = True

    @pytest.mark.parametrize("numbers, threshold, expected", [
        ([1, 2, 3, 4], 0.5, False),
        ([1, 2, 3, 3.1], 0.5, True),
        ([10, 20, 30], 5, False),
        ([10, 20, 25], 5, True)
    ])
    def test_has_close_elements(numbers, threshold, expected):
>       assert has_close_elements(numbers, threshold) == expected
E       assert False == True
E        +  where False = has_close_elements([10, 20, 25], 5)

program1/program1_test.py:11: AssertionError

---------- coverage: platform darwin, python 3.11.7-final-0 ----------
Name                           Stmts   Miss Branch BrPart  Cover
----------------------------------------------------------------
program1/program1.py               8      0      8      0   100%
program1/program1_test.py          5      0      2      0   100%
----------------------------------------------------------------
TOTAL                             13      0     10      0   100%

=========================== short test summary info ============================
FAILED program1/program1_test.py::test_has_close_elements[numbers3-5-True] - ...
========================= 1 failed, 3 passed in 0.14s ==========================
```

Fig. 9: Snippet of a `pytest-cov` reponse for a single programs Unit Tests.