Software Testing with Large Language models: Semantic Coverage

Naser Ezzati-Jivan Marium Nur Rouvin Rebello Shihab Khateeb

Introduction

In traditional software testing, code coverage metrics such as statement, branch, and function coverage have been the primary means of assessing the thoroughness of tests. However, these methods often overlook the semantic importance of different parts of the code, leading to scenarios where a high coverage percentage does not necessarily equate to effective testing. To address this, we propose a novel approach to software coverage, termed **Semantic Coverage**, which introduces a more meaningful way to measure code coverage by assigning weightages to different segments of a function based on their significance. Each function is divided into distinct code splits—Core Functionality, Boundary Conditions and Edge Cases, Error Handling, Integration Points, User Interface Interactions, Security Features, Performance and Scalability, Configuration and Environment, and Output Consistency. Each of these splits is then assigned a weight based on its importance to the overall function. The total coverage for a function is calculated by aggregating these weighted splits, offering a more nuanced view of coverage.

For instance, consider a 'withdraw' function in a banking application. The core functionality, which updates the account balance, is crucial and thus given a higher weight (e.g., 70 out of 100). In contrast, the pre- and post-conditions, while still important, are assigned lower weights. By evaluating coverage through this lens, we can recognize that a test case with lower statement coverage but targeting core functionality might be more valuable than one with higher coverage but focusing on less critical aspects. Semantic Coverage thus provides a more accurate and meaningful metric for assessing test effectiveness, bridging the gap between coverage numbers and real-world software reliability.

Background

Code coverage is a metric that indicates (as a percentage) the amount of the source code that is executed when the test suite being measured is run. Test suites with higher code coverage indicate that more of the source code is being tested by the suite, and is therefore less likely to contain bugs. There exist different methods of calculating code coverage, such as statement, branch, function, basic block, etc, but shared between these methods is that all code executed is treated as equal, and contributes to the final coverage equally.

Large language models are models created for natural language processing and generation tasks. Our experiment uses a mixture of GPT 3.5, 4, and 4o to split the code into the discrete blocks and assign weights to each.

Semantic coverage aims to divide the source code being tested into discrete 'blocks', each of which is assigned a weight using a GPT model. This weight indicates how significant the block is to the functioning of the source code as a whole, with more important blocks receiving higher weights than less critical blocks. Given these weights, code is no longer treated equally, and test coverage of higher priority code contributes more towards the final coverage score than that of low priority code. [table to be inserted]

Statement coverage is a measure of how many of the executable statements are executed when the program is run. Branch coverage checks to ensure that every (true/false) decision's outcome is executed at least once. Semantic coverage splits the code, assigns weights, obtains statement coverage for each block of split code, and then alters that statement coverage value with respect to weightage to obtain a more accurate value of coverage.

Methodology

Evaluating Semantic Coverage requires a systematic organization of both the project's source code and associated test files. Unit testing plays a crucial role in this process, as it verifies the accuracy and functionality of individual

software components. As outlined in The Role of Testing Professionals in Unit Testing (Neves et al., 2024), each function in the source code is accompanied by corresponding test cases within structured test files. These test cases rigorously validate the performance and correctness of the function, ensuring the system meets high-quality standards.

1. Source Code and Test Files

- **Source Code Division:** The source code can be segmented into discrete functions, each with specific objectives. These functions are the focus of unit testing. However, it is not necessary.
- **Test Files:** Each function is linked to test cases within organized test files. These tests validate the function's performance and correctness, ensuring that all code paths are adequately covered and the function behaves as expected under various conditions.

2. Semantic Coverage Process

The semantic coverage process involves categorizing the source code into logical blocks, each reflecting a distinct objective of the function. The key categories include:

- **Core Functionality:** Testing key business logic and expected outputs.
- **Boundary Conditions and Edge Cases:** Ensuring proper handling of boundary values and invalid inputs.
- **Error Handling:** Validating exception management and system behavior under unexpected conditions.
- **Integration Points:** Assessing the reliability of interactions with external systems, such as APIs and databases.
- **User Interface (UI) Interactions:** Testing user inputs, navigation, and overall user experience.
- **Security Features:** Verifying secure handling of data and prevention of vulnerabilities.
- **Performance and Scalability:** Evaluating system behavior under varying loads and stress conditions.
- **Configuration and Environment:** Ensuring the system operates correctly across different environments.
- **Output Consistency:** Validating the accuracy and consistency of system outputs, such as logs and reports.

For example, core functionality typically holds the highest weight, while boundary conditions Each block is assigned a weight based on its importance to the function's overall objective. This weighting is essential for a balanced evaluation of the function's impact on system reliability. and error handling are crucial for ensuring software robustness, especially in edge-case scenarios.

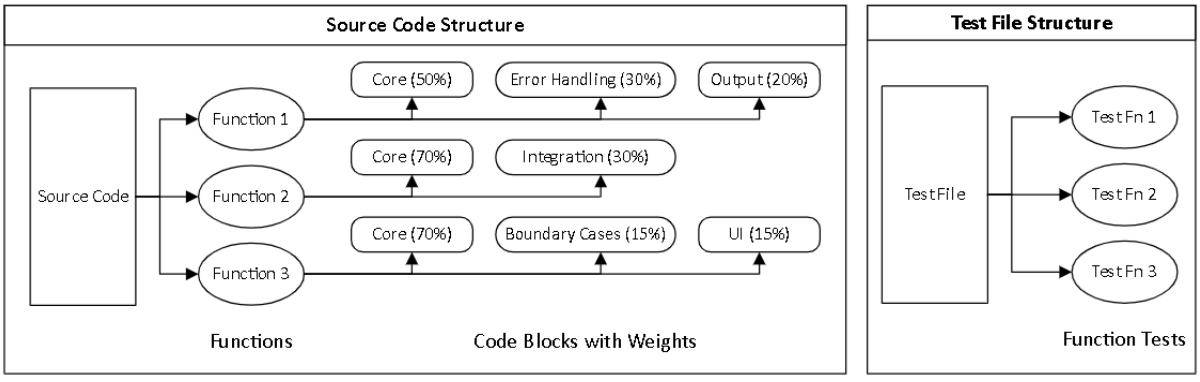


Figure 1: File Structure

3. Recent Advancements in Large Language Models (LLMs)

Recent advancements in Large Language Models (LLMs) like ChatGPT and Codex have introduced more efficient methodologies for software testing. LLMs possess the capability to understand and analyze static code, enabling their application in tasks such as test case generation, bug reporting, and code analysis, as explored in Exploring the Practical Applications of Large Language Models in Software Testing (Santos et al., 2023). LLMs can automate the categorization of code blocks and the assignment of appropriate weightings, leading to more systematic semantic coverage.

Additionally, LLMs support the reorganization of source code into modular blocks to optimize test coverage each of which are also assigned weights. By dividing a function into testable units aligned with the defined categories (e.g., core functionality, error handling), developers can ensure isolated testing of each block, thus improving overall coverage and enhancing code reliability. As noted in Software Testing with Large Language Models: Survey, Landscape, and Vision (Wang et al., 2024), models like ChatGPT and Codex excel in generating and refining unit tests via structured prompts, significantly contributing to test accuracy and software resilience.

4. Evaluation Process Using a Python Script

The evaluation of semantic coverage involves the following steps:

- **Function-Level Evaluation:**
  - Extract the corresponding test file for each function and identify the logical code blocks.
  - Evaluate the statement coverage for each block using a Python script to determine the extent to which each block is tested by the unit tests.
  - Multiply the statement coverage score of each block by its assigned weight to calculate the block score.
  - Sum the block scores to derive the Function Semantic Coverage.
- **Project-Level Evaluation:**
  - Repeat the above steps for all functions within the project.
  - Calculate the average of the Function Semantic Coverage scores to determine the overall Project Semantic Coverage.

This framework provides a comprehensive approach to evaluating semantic coverage in software projects. By integrating traditional unit testing methodologies with the advanced capabilities of LLMs, this approach ensures a thorough analysis of both function-level and project-level coverage. The use of LLMs not only automates complex tasks but also enhances the accuracy and efficiency of the testing process, leading to higher software quality and reliability.

5. Traditional Coverage Metrics

Semantic coverage scores are compared against traditional coverage metrics like function coverage, statement coverage, and branch coverage to assess their effectiveness in ensuring comprehensive software testing.

- **Statement Coverage (C0 Coverage):**
  - Measures whether each executable statement in the code has been executed at least once.
  - It is calculated as the percentage of program statements executed by the test cases out of the total program statements.
- **Branch Coverage:**
  - Measures whether each possible branch (e.g., if-else conditions) in the code has been executed.
  - This ensures that all logical branches in the code are tested, revealing potential issues in decision-making structures (Guo et al., 2024).
- **Function Coverage:**
  - Tracks how many of the functions defined in the code have been called during testing.

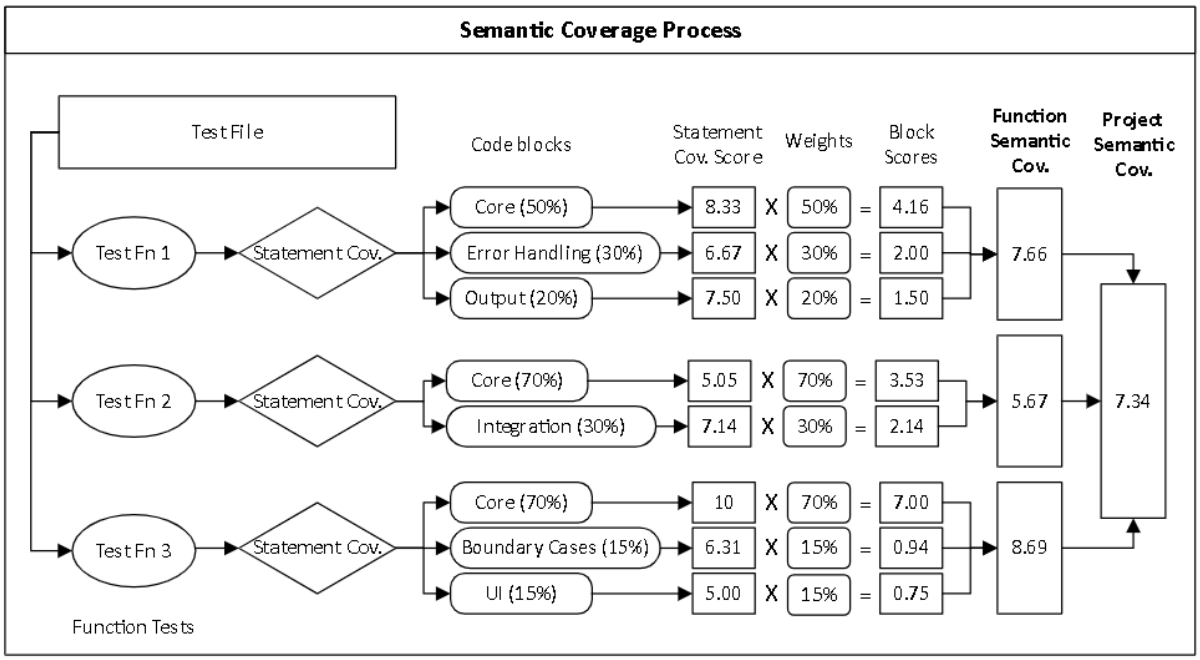


Figure 2: Semantic Coverage Process

- It assesses the completeness of function testing, ensuring that every function has been tested at least once (Chakrabarty and Huang, 2012).

These traditional metrics focus on different levels of code execution, from individual statements to broader logical paths, providing a quantitative measure of test coverage.

Semantic coverage code

Click here for GitHub Repository
Click here for Ranking Spreadsheet

This semantic coverage script analyzes the coverage of a specific function within a source file by evaluating the lines executed during test cases and comparing them to the total lines within the function. The script first identifies the lines corresponding to the specified function and then measures the coverage by determining which lines are executed by the tests. It provides a detailed breakdown of covered and missed lines and calculates the statement coverage as a percentage. This coverage metric is then used to derive a coverage score, offering a more meaningful evaluation of test effectiveness than traditional methods.

The script uses the coverage.py library to measure the statement coverage of a specific function within a Python source file. It starts by collecting coverage data while executing a test file, then analyzes the coverage results to determine which lines of the target function were covered, missed, or unaccounted for. It calculates the coverage percentage by comparing the covered lines against the total lines in the function, and provides detailed output on the coverage, including the percentage and a derived coverage score. This allows for focused analysis of how well a specific function is tested.

To apply this script to your project, you need to update the `source_file` and `test_file` variables to point to your specific source code and test files. Additionally, you should modify the function name in the `measure_coverage` call to match the function you wish to analyze. For example:
`coverage_percent = measure_coverage(source_file, test_file, "your_function_name").`
If you encounter an error message stating, "Error analyzing coverage data: 'NoneType' object cannot be interpreted as an integer," it likely indicates that the function name specified does not exist in the source file. Ensure the function name exactly matches the one in your code, including case sensitivity.

Data Analysis

Project 1 - Shopping Cart Unit Tests

The GitHub project "shopping-cart-unit-tests" by AutomationPanda is a tutorial repository focused on teaching unit testing in Python using the `pytest` framework. It features a simple shopping cart scenario in `shopping_cart.py` with functionalities like adding items, calculating totals, and applying discounts. The `test_shopping_cart.py` file demonstrates writing unit tests to verify these functions' behavior. Aimed at beginners, the project covers various testing scenarios, such as handling different item types, calculating discounts, and testing for exceptions. Users can clone the repository, install Python and `pytest`, and run tests to learn about test-driven development (TDD) and effective testing practices. This repository is an excellent educational resource for understanding how to write clean, comprehensive unit tests to maintain code quality.

Function	Semantic Coverage	Traditional Coverage
calculate_total	5.34	5.41
calculate_item_total	5.80	7.03
add_item	6.07	7.13
calculate_subtotal	5.77	7.24

Table 1: Coverage Comparison for Shopping Cart Unit Tests Functions

Project 2 - Poketerminal Game

PokéTerminal Game is a command-line based, interactive Pokémon-inspired game developed in Python. The project emphasizes object-oriented programming and modular design principles to simulate a turn-based battle system. Players can choose from various Pokémon, each having unique stats and abilities, to engage in battles against either a computer-controlled opponent or another player. The game implements features such as HP management, attack types, and randomized outcomes, offering a simple yet strategic gameplay experience. By using Python's standard libraries alongside custom classes for Pokémon and attacks, the project demonstrates a balance between code simplicity and effective game logic. The code repository showcases clear documentation and examples, making it accessible for users who want to understand or modify the game's mechanics. This project serves as an educational tool for learning Python programming, game development fundamentals, and object-oriented concepts in a text-based environment.

Function	Semantic Coverage	Traditional Coverage
calcular_ataque	8.50	8.11
definir_vantagens	6.16	7.73
_restaurar_pokemons	7.42	8.94
batalhar	10.00	8.34
capturar_pokemon	10.00	8.82
get_inimigo	9.20	8.71
_resultado_da_batalha	9.82	8.44
apresentar_pokemons	9.78	8.94
atacar	5.85	8.17
mudar_estrategia_para_luta	9.00	8.49
recuperar_vida	10.00	8.78
subir_nivel	6.84	7.62
_reconfigurar_status	10.00	8.80
apresentar_pokemon	5.00	5.89
get_pokemon	9.25	5.44

Table 2: Coverage Comparison for PokéTerminal Game Functions

Project 3 - Tic-Tac-Toe

The **Tic-Tac-Toe Project** is a Python-based implementation of the classic Tic-Tac-Toe game with a focus on object-oriented design. The code is modular, with separate functions for game logic, including checking for

win conditions, updating the board, and parsing player input. The project also includes test coverage analysis, comparing semantic and traditional test methods across key functions. The goal of the project is to provide a simple but effective demonstration of game logic design, code coverage techniques, and Python programming practices. The repository includes detailed documentation, making it accessible for developers seeking to study basic AI techniques and code testing methodologies.

Function	Semantic Coverage	Traditional Coverage
diagonals	10.00	6.65
columns	10.00	7.48
update_board	5.65	6.17
check_win	7.44	6.44
parse_coordinates	6.92	5.44
play	6.67	6.13

Table 3: Coverage Comparison for Tic-Tac-Toe Functions

Project 4 - Blackjack

The **Python Blackjack Game** is a terminal-based implementation of the classic card game Blackjack, written in Python. The project is designed with simplicity in mind while adhering to core principles of object-oriented programming. Players can interact with the game through the command line, making decisions such as hitting or standing while competing against the dealer. The game employs various functions, including dealing cards, calculating hand totals, and comparing outcomes. Test coverage analysis is provided, highlighting both semantic and traditional testing approaches for key functions. This project is an excellent starting point for developers looking to explore game development, card logic, and testing techniques in Python.

Function	Semantic Coverage	Traditional Coverage
deal_card	6.04	6.83
calculation_total	5.81	6.89
compare	6.14	6.37
game_play	7.51	6.68

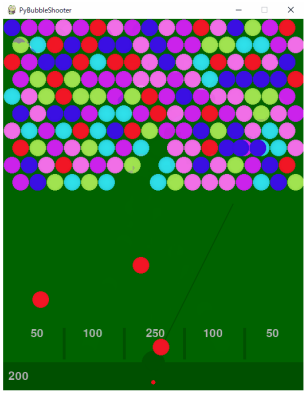
Table 4: Coverage Comparison for Blackjack Functions

Project 5 - Bubble Shooter Game

The Python bubble shooter game uses the Pygame library and involves launching colored bubbles to form groups and clear the board. Bubbles are managed within a grid of cells, and the player controls a launcher at the bottom of the screen. The game includes collision detection, angle calculations for bouncing, and status management for various game states like PLAY, WIN, and GAMEOVER. The semantic coverage analysis for the Python bubble shooter game offers a more comprehensive evaluation of the project's code than traditional coverage methods. For instance, functions like `simulate_shoot_right` and `simulate_shoot_left` achieved a perfect score of 10.00 in semantic coverage, indicating that their critical aspects are thoroughly covered by the tests. On the other hand, traditional coverage metrics, which combine branch, statement and function coverage, offer lower scores and do not distinguish between the importance of different parts of the code.

Function	Semantic Coverage	Traditional Coverage
calculate_sides	8.00	7.19
calculate_center	7.60	7.41
move_bubble	7.83	7.51
decide_positions	8.11	7.04
simulate_shoot_right	10.00	7.33
simulate_shoot_left	10.00	7.33
update	9.87	9.55
change_bubbles	7.17	7.22

Table 5: Bubble Shooter game

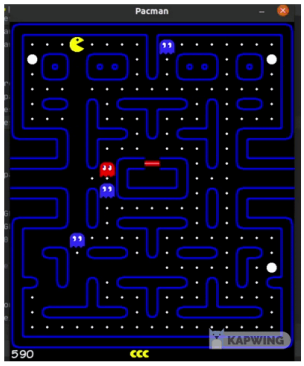


Project 6 - Pacman game

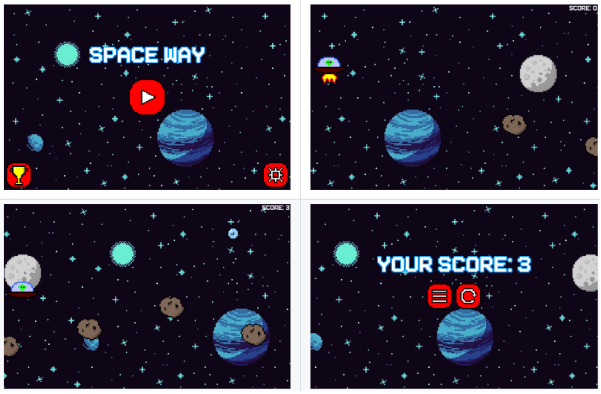
The Pacman game project is a classic arcade-style game where players control Pacman to navigate through a maze, eating pellets and avoiding ghosts. It features various game modes and levels, where Pacman can collect power-ups to turn ghosts vulnerable. The game integrates sound effects and animations to enhance the experience, with a focus on player control, ghost AI, and dynamic maze interactions. Key elements include handling collisions, managing game states, and updating graphics and scores in real-time. Functions like `draw` and `start_game` received high semantic coverage scores of 8.10 and 8.08, respectively, reflecting a strong focus on their essential components. Traditional coverage metrics, which merge branch and statement coverage, show lower scores for these functions, suggesting that while the code is executed, the significance of each segment is not fully captured.

Function	Semantic Coverage	Traditional Coverage
move	7.75	5.58
draw	8.10	5.33
start_game	8.08	5.64
event_loop	5.50	5.33
draw_texts	6.94	6.08
set_mode	7.40	5.86
check_ghosts_state	6.84	6.14

Table 6: Pacman game



Project 7 - Space Way game



Function	Semantic Coverage	Traditional Coverage
clamp_ip	7.65	6.43
clip	7.65	6.43
union_ip	7.65	6.43
fit	10.00	6.43
colliderect	9.33	6.25
fy	10.00	6.54
update	10.00	5.93

Table 7: Space Way game

Space Way is an arcade-style game built with Pygame. It features multiple scenes, including a lobby, settings, and gameplay area, with customizable sound and full-screen modes. The game includes real-time updates, smooth transitions, and interactive elements, supported by debugging tools and configuration management.

The comparison between semantic coverage and traditional coverage for the Space Way game highlights distinct differences in test evaluation. Semantic coverage, which assesses critical aspects of each function, shows high scores for functions like fit, fy, and update (10.00), reflecting thorough testing of their key components. Traditional coverage, which combines branch and statement coverage, reports lower scores across all functions, with the highest being 6.54 for fy and the lowest at 5.93 for update.

Project 8 - Card War Game

The **Card War Game** is a Python-based simulation of the classic card game ”War,” implemented with object-oriented principles. The game involves two players who draw cards from a shuffled deck, comparing their values to determine a winner. The project features a modular structure with methods for initializing the game, shuffling cards, dealing cards, comparing values, and determining the overall winner. Additionally, this project includes test coverage analysis, providing insights into the effectiveness of semantic and traditional coverage for the various

functions used in the game. The project serves as an educational tool for Python developers interested in game design, card handling logic, and testing methodologies.

Function	Semantic Coverage	Traditional Coverage
__init__	10.00	7.12
__str__	6.50	6.64
__repr__	6.60	6.64
__gt__	8.45	7.11
__eq__	5.50	6.90
new_game	6.17	6.28
get_winner	6.98	7.64
cards_shuffle	5.79	7.00
deal_one	7.71	7.73
play_game	7.73	6.31
set_hand	8.10	9.04
get_card	7.50	8.95
add_card	6.75	6.05

Table 8: Coverage Comparison for Card War Game Functions

Project 9 - Pytube

Pytube is a Python based library and command line tool that allows downloading of videos hosted on Youtube. It offers a variety of options for downloading videos alone, from a playlist, from a channel, etc. The command line interface allows users to take advantage of its functionally without needing to write a program. The program has an extensive test suite that was used to calculate traditional and semantic coverage for comparision.

Function	Semantic Coverage	Traditional Coverage
xml_caption_to_srt	6.058	
cache_tokens	9.94	
refresh_bearer_token	6.534	
fetch_bearer_token	6.492	
_call_api	6.737	
calculate_n	6.5255	
get_throttling_function_name	7.575	
get_throttling_function_code	5.535	
get_throttling_function_array	5.47	
get_throttling_plan	6.117	
get_initial_function_name	5.3345	

Table 9: Coverage Comparison for Pytube Functions

Results

Project-Level Semantic vs. Traditional Coverage

The comparison of semantic and traditional coverage across eight projects reveals significant variations in performance between the two metrics (Table 9). Semantic coverage ranged from 5.815 to 8.898, while traditional coverage ranged from 5.710 to 8.844

- **Project 1** (shopping-cart-unit-tests) exhibited lower semantic coverage (5.815) compared to traditional coverage (6.844).
- **Project 2** (poketerminal-game) displayed higher semantic coverage (8.452) than traditional coverage (8.081), indicating that semantic measures may better capture certain behaviors in game simulations.
- **Project 7** (space way game) had the highest semantic coverage (8.898), significantly exceeding traditional coverage (6.348).

Overall, semantic coverage outperformed traditional coverage in several projects, particularly in game-related applications (poketerminal-game, bubble shooter game, space way game).

Projects	Semantic Coverage	Traditional Coverage
Project 1 - shopping-cart-unit-tests	5.815	6.844
Project 2 - poketerminal-game	8.452	8.081
Project 3 - Tic-Tac-Toe	7.779	6.385
Project 4 - Blackjack	6.376	6.693
Project 5 - bubble shooter game	8.572	7.573
Project 6 - pacman game	7.229	5.710
Project 7 - space way game	8.898	6.348
Project 8 - Card-war-game-python	7.213	7.187
Project 9 - Pytube	6.574	

Table 10: Project Level Summary: Semantic vs. Traditional Coverage

Project-Level Semantic Coverage Statistics

Table 10 summarizes the statistics for semantic coverage across all projects, including the mean, median, standard deviation, minimum, and maximum values.

- **Project 7** (space way game) had the highest mean semantic coverage (8.898) and a maximum score of 10.0, indicating consistently high coverage.
- **Project 2** (poketerminal-game) exhibited the largest variation in coverage, with a standard deviation of 1.741
- Projects such as **Project 1** (shopping-cart-unit-tests) had lower variability, with a standard deviation of only 0.225, suggesting more uniform coverage.

Project	Mean	Median	Standard Deviation	Min	Max
Project 1 - shopping-cart-unit-tests	5.815	5.843	0.225	5.338	6.071
Project 2 - poketerminal-game	8.452	9.195	1.741	5.000	10.000
Project 3 - Tic-Tac-Toe	7.779	7.178	1.816	5.650	10.000
Project 4 - Blackjack	6.376	6.090	0.771	5.808	7.514
Project 5 - bubble shooter game	8.572	8.053	1.181	7.168	10.000
Project 6 - pacman game	7.229	7.400	0.914	5.500	8.101
Project 7 - space way game	8.898	9.334	1.191	7.650	10.000
Project 8 - Card-war-game-python	7.213	6.980	1.220	5.500	10.000
Project 9 - Pytube	6.754	6.492	1.291	5.535	9.940

Table 11: Project Level Semantic Coverage Statistics

Function-Level Semantic vs. Traditional Coverage

The function-level analysis (Table 11) highlights how semantic coverage captures deeper behavior across various projects. Key findings include:

- **Project 2** (poketerminal-game) functions such as batalhar and capturar_pokemon achieved the maximum possible semantic coverage of 10.0, outperforming traditional coverage.
- In **Project 5** (bubble shooter game), functions like simulate_shoot_right and simulate_shoot_left also achieved the maximum semantic coverage of 10.0, significantly higher than traditional scores of 7.327.
- **Project 3** (Tic-Tac-Toe) functions diagonals and columns reached a perfect semantic coverage score of 10.0, compared to traditional coverage of 6.647 and 7.480 respectively.

The function-level results demonstrate that semantic coverage is more capable of detecting subtle behavioral nuances and capturing comprehensive test coverage, particularly in gaming applications.

Project	Function	Semantic Coverage	Traditional Coverage
Project 1 - shopping-cart-unit-tests	calculate_total	5.338	5.407
	calculate_item_total	5.800	7.027
Project 2 - poketerminal-game	batalhar	10.000	8.337
	capturar_pokemon	10.000	8.823
Project 3 - Tic-Tac-Toe	diagonals	10.000	6.647
Project 5 - bubble shooter game	simulate_shoot_right	10.000	7.327
Project 6 - pacman game	move	7.749	5.583
Project 7 - space way game	update	10.000	5.925
Project 8 - Card-war-game-python	__init__	10.000	7.123

Table 12: Function Level Summary: Semantic vs. Traditional Coverage

Future Works

Fine-tuning of LLM for Automated Code Splitting and Weight Assignment: We aim to fine-tune an existing language model to automatically analyze a given function, correctly splitting it into predefined semantic categories (such as Core Functionality, Error Handling, Integration Points) and assigning accurate weightage to each split. Our approach involves creating a database that includes function splits and code weight assignments from our current and any potential future projects. By incorporating this data into the fine-tuning process, we will enhance the model's ability to provide consistent and precise categorization and weighting across various codebases. This will streamline the application of semantic coverage, ensuring more reliable and accurate results.

Enhancement of Traditional Coverage Code: Future work will focus on improving the accuracy of branch coverage analysis. Currently, tools like `coverage.py` may not always recognize conditional statements accurately, resulting in incomplete or inaccurate branch coverage results. One potential solution is to enhance the parsing logic of these tools to better identify complex conditional expressions, including nested and compound conditions. Additionally, integrating static analysis techniques could help to cross-reference and validate identified branches, leading to more reliable and comprehensive branch coverage calculations.

Development of an Automated Semantic Coverage Tool: The final step involves creating a comprehensive tool capable of handling an entire project, which may include multiple '.py' files and multiple test files, to generate semantic coverage with a single click as follows:

1. **Identify All Source Files:** Detect and list all `.py` files in the project directory.
2. **Identify All Functions:** Identify and extract all functions within each source file.
3. **Process Each Function:**
    a. **Fine-tuned LLM:** Pass each function to a fine-tuned LLM to get code splits.
    b. **Save Splits:** Automatically save each split into separate `.py` files using Python packages.
    c. **Semantic Coverage Analysis:** Process each split through the semantic coverage script.
    d. **Calculate Function Coverage:** Compute the semantic coverage for the function by averaging the coverage results of all splits. For example, if Core has a semantic coverage of 9 with a weightage of 0.7, and Output Consistency has a semantic coverage of 4 with a weightage of 0.3, then the semantic coverage for the function is calculated as $(9 \times 0.7 + 4 \times 0.3) = 7.5$.
4. **Repeat for All Functions:** Apply the above steps to all functions across all source files.
5. **Compute Project Coverage:** Calculate the overall project semantic coverage by averaging the semantic coverage results of all functions, providing a comprehensive measure of coverage for the entire project.

Extra - Failed attempts

Contributions

Rouvin

- Conducted research and cited key papers in software testing, including:
  - *Testing Coverage Functions*
  - *Optimal Test Case Generation for Boundary Value Analysis*
  - *Elevating Software Quality in Agile Environments: The Role of Testing Professionals in Unit Testing*
  - *Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing*
  - *Software Testing with Large Language Models: Survey, Landscape, and Vision*
- Created scripts to test coverage (Statement, Branch, and Function) and traditional metrics (Code Metrics, Smell Tests, Static Analysis, Test Execution Time).
- Developed prompts to decompose source code into function files and divide functions into distinct blocks (core, boundary, etc.).
- Created the Ranking System Excel file that compares:
  - Semantic coverage: (Mean, Median, Standard Deviation, Min, Max)
  - Coverage methods: Statement, Branch, Function
  - Manual Testing Methods: Code Metrics, Smell Tests, Static Analysis, Test Execution Time
  - Summary Analysis Sheet
- Calculated Semantic Coverage, Traditional Coverage, and Manual Testing Methods for 5 projects (46 functions):
  - Project 1: *Shopping Cart Unit Tests*
  - Project 2: *Poketerminal Game*
  - Project 3: *Tic-Tac-Toe*
  - Project 4: *Blackjack*
  - Project 8: *Card-War-Game-Python*
- Enhanced data analysis sections by adding projects, creating workflow diagrams, and developing methodology and results sections.

Marium

- Worked on the `code_block_prompt.py` in GitHub: it uses an OpenAI API key, takes in a provided function which is passed on to GPT to divide into categories, and automatically places each category in separate files. (Better results can be obtained from this process if the LLM is fine-tuned.)
- Worked on some possible improvements to avoid common errors in the semantic coverage script written by Rouvin. Solved common errors like: `Error running test file: name 'mock' is not defined`, `UnicodeEncodeError: 'utf-8' codec can't encode character '\udc80' in position 27: surrogates not allowed`, and much more.
- Calculated semantic coverage for 3 projects, including Bubble Shooter, Pacman, and Space Way games.
- For this report, I worked on the introduction, future works along with an intital step by step process to how semantic coverage can be implemented to work with a single click, and a description of the three projects that I worked on.
- I read the entire paper and suggested additions to make it more detailed.

Shihab

- Worked on the semantic coverage script, troubleshooting common errors with it not being able to read files.
- Calculated semantic coverage for Project 8: Pytube.
- Wrote the Background and Failed Attempts sections, and proofread the paper.

References

Chakrabarty, D. and Z. Huang (2012). *Testing Coverage Functions*. Springer. Accessed: August 24, 2024.

Guo, X., H. Okamura, and T. Dohi (2024). Optimal test case generation for boundary value analysis. *Software Quality Journal 32*, 543–566.

Neves, L., O. Campos, R. Santos, C. C. de Magalhaes, I. Santos, and R. de Souza Santos (2024). Elevating software quality in agile environments: The role of testing professionals in unit testing. Accessed: August 24, 2024.

Santos, R., I. Santos, C. Magalhaes, and R. de Souza Santos (2023). Are we testing or being tested? exploring the practical applications of large language models in software testing.

Wang, J., Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang (2024). Software testing with large language models: Survey, landscape, and vision.