Semantic Coverage: A Contextual Approach to Software Testing Using LLMs

Muhammed Bilal [1] **Rouvin Rebello** [1] **Marium Nur** [1] **Shihab Khateeb** [1] **Naser Ezzati-Jivan** [1] **Majid Babaei** [1]

Abstract

Traditional software testing metrics - such as statement, branch, and function coverage, are inadequate in evaluating the semantic importance of tested logic. These metrics treat all lines equally, disregarding the actual purpose or risk weight of a code segment. In this paper, we propose a novel semantic coverage framework using large language models (LLMs) to categorize each line of a function into logical blocks such as core functionality, error handling, and user interface interactions. We then compute a weighted semantic score by combining block-level test coverage with domain-informed weights. Our methodology first segments Python functions using an LLM-based prompt into logical blocks and then maps execution coverage to each block using `coverage.py`. This enables us to prioritize blocks based on criticality rather than raw line count. We evaluated the performance using real-world and course-assigned Python projects and compared semantic scores with traditional statement coverage. Our results reveal that semantic coverage highlights undertested critical logic more effectively, making it a promising direction for quality assurance and test optimization.

1. Introduction

Software testing is vital to ensure system reliability, yet many traditional test coverage metrics fall short of capturing the true effectiveness of test suites. Statement coverage, branch coverage, and function coverage all rely on syntactic execution paths and treat all lines of code equally. This can misrepresent the actual criticality of code logic: testing a logging statement weighs the same as testing core business logic.

We propose a novel approach called *semantic coverage*, which leverages large language models (LLM) to segment code into semantically meaningful blocks, such as core functionality, error handling, or user interface logic, and weights their coverage accordingly. This weighted method provides a more insightful and contextual view of the code logic tested.

Using LLMs such as GPT-4, we categorize function contents line by line, applying predefined semantic categories. The execution data from tools such as `coverage.py` is then mapped to these blocks, and we compute a semantic score by multiplying the coverage ratio with the importance weights of the blocks. This metric not only highlights untested high-risk logic, but also prioritizes meaningful test development.

Our contributions include:

- A method for segmenting code into logical blocks using LLMs.
- A semantic weighting formula for test coverage.
- An evaluation of semantic vs. statement coverage on Python projects.
- Insights into how semantic scores align better with test relevance.

2. Background and Theory

2.1. Traditional Coverage Metrics

Traditional coverage metrics such as **statement**, **branch**, and **function** coverage have been widely used to quantify how thoroughly software is tested. Statement coverage checks whether each line of code has been executed at least once. Branch coverage measures whether every possible branch (ex: if/else conditions) has been taken. Function coverage verifies whether all defined functions were called during testing.

These metrics are commonly collected using tools like `coverage.py` for Python-based projects. Despite their popularity, these metrics operate under a fundamental limitation: they treat all lines of code as equal in importance. For example, a simple print statement and a core business logic line are counted the same in statement coverage, which may not accurately reflect the true quality of test effort.

*Equal contribution [1]Department of Computer Science, Brock University, St. Catharines, Canada. Correspondence to: Muhammed Bilal <bb18hb@brocku.ca>.

2.2. Use of LLMs in Software Testing

Recent advances in large language models (LLMs) such as GPT-4, CodeBERT, and Mistral have enabled new capabilities in software engineering tasks including code summarization, generation, and defect detection. These models understand the structure and semantics of code at a deep level, making them suitable for tasks that require contextual reasoning beyond just code syntax.

While LLMs have shown a lot of promise in automated documentation and code review, their application to software testing still remains underexplored. In this work, we leverage LLMs not for generating test cases, but for analyzing the semantic structure of functions. By categorizing each line of a function into logical blocks (ex: core functionality, error handling), we introduce a new paradigm - **semantic coverage** - that emphasizes what is specifically being tested, not just how much.

This semantic perspective allows us to identify whether the most critical parts of a function, such as core logic and error handling, are sufficiently tested by existing test suites. By revealing gaps in testing focus and providing contextual information, semantic coverage provides a practical tool for enhancing test suite effectiveness, especially in real-world contexts where quality assurance and risk mitigation are essential.

3. Methodology

3.1. Organizing Source Code and Test Files

To support semantic analysis across projects of varying size and complexity, we design our tooling to dynamically traverse project directories. Our scripts recursively scan all subdirectories to collect:

- **Source files**: All Python files (`.py`) that do not start with a dot.
- **Test files**: Any file prefixed with `test_` or contained within a `tests/` folder.

This generalization ensures compatibility with medium to large-scale codebases and automates function extraction without requiring a rigid structure.

Each function from the source files is then segmented and saved as a uniquely identified unit. Corresponding test coverage data is also collected on a per-line basis using `coverage.py`.

3.2. Function Segmentation Using LLMs

We apply GPT-4 to semantically segment functions into logical blocks based on their intent. Every function is passed into the LLM using a prompt that preserves original line numbers. Each line is then classified into one of the following semantic categories:

- **Core Functionality**: Primary business logic and main outputs.
- **Boundary Conditions and Edge Cases**: Handling input extremes and corner cases.
- **Error Handling**: Exceptions, validation, fail-safe behavior.
- **Integration Points**: External system/API interactions.
- **User Interface Interactions**: Input/output, user prompts, GUI actions.
- **Security Features**: Sensitive data checks, access control.
- **Performance and Scalability**: Resource limits, batch handling.
- **Configuration and Environment**: Logging, setup, system compatibility.
- **Output Consistency**: Return formatting, deterministic responses.

This segmentation enables fine-grained understanding of what each part of the code is trying to achieve, improving the alignment between testing focus and real-world risks.

Furthermore, rather than sending all functions in a project to the GPT-4 LLM at once, we adopted a batch-based approach, analyzing and segmenting one function per prompt. While this method is slower, it significantly improves the segmentation accuracy, especially for medium to large-scale projects.

Initial testing revealed that prompting the LLM to segment multiple functions simultaneously often resulted in incomplete outputs. This was particularly problematic for longer or more complex functions, where entire blocks or lines would be omitted from the final categorization.

By segmenting each function individually, we ensured:

- Every line of the function was accounted for and categorized.
- No logical block was skipped due to prompt truncation or token limits.
- Each resulting block could be saved into a separate file reliably.

This per-function batching strategy ensures consistent semantic granularity, which is critical for accurate semantic coverage scoring.

In addition, to improve the quality and consistency of the LLM's segmentation, we implemented a few-shot learning approach using GPT-4. A curated set of Python function examples was used to guide the model in accurately identifying and labeling semantic blocks. Each example included an original function and a manually labeled block segmentation.

These examples were derived from real-world open-source Python repositories, which were accessed using the GitHub API. The selected functions were representations of a lot of diverse logic patterns and use cases (ex: utility functions and I/O handlers) to ensure that the model can generalize more efficiently. The structured few-shot prompt enabled the model to better understand the semantic distinctions among blocks such as Core Functionality, Error Handling, and Boundary Conditions.

This approach significantly improved the reliability of the block segmentation produced by the LLM, increasing the average recall, precision, and f-1 score to 90% and even 80% for longer complex functions.

3.3. Line Numbering as Semantic Identifiers

During preprocessing, we append line numbers to each line of every extracted function to create a unique identifier per line. This serves multiple purposes:

- Prevents ambiguity when multiple lines contain similar syntax (e.g., `return x`).
- Enables accurate matching between execution coverage and semantic blocks.
- Ensures block-level granularity even in long or repeated functions.

Only executable lines (i.e., not comments or multiline strings) are assigned numbers. For example, a function:

```
def deal_card():
    cards = [11, 2, 3, 4, 5]
    return random.choice(cards)
```

Becomes:

```
def deal_card():
#1      cards = [11, 2, 3, 4, 5]
#2      return random.choice(cards)
```

This guarantees consistent references during semantic mapping and evaluation.

3.4. Statement Coverage Collection

We use `coverage.py` to collect which lines of code are executed during test runs. This is done using:

```
coverage run -m pytest
coverage json -o coverage.json
```

The resulting JSON file lists all covered line numbers per file. We cross-reference these line numbers with the LLM's block categorization to determine per-block test coverage.

3.5. Semantic Coverage Computation

We define a set of weights per block category to reflect their criticality to program correctness, safety, and business logic. The weights are:

Table 1. Logical Block Weights

Block Type	Weight
core_functionality	10
error_handling	9
boundary_conditions_and_edge_cases	8
integration_points	7
security_features	6
performance_and_scalability	5
output_consistency	4
configuration_and_environment	3
user_interface_interactions	2

Based on Table 1, weights are inspired by relative risk profiles in software engineering standards. For instance, failure in *core functionality* or *error handling* typically leads to system-critical bugs, whereas UI misalignment is less dangerous to the system. Future work will explore automated and data-driven weight calibration using datasets such as Defects4J or industrial fault repositories.

Given weights and test coverage per block, we compute each function's semantic score as:

$$\text{semantic\_score}_f = \sum_{b \in B_f} w_b \cdot \text{coverage}_b \qquad (1)$$

Where:

- B_f: all semantic blocks in function f
- w_b: the weight for block type b
- coverage_b: Statement coverage: the fraction of lines in block b covered during tests

This quantifies how well a function's core responsibilities are covered by its test suite, not just how many lines are covered.

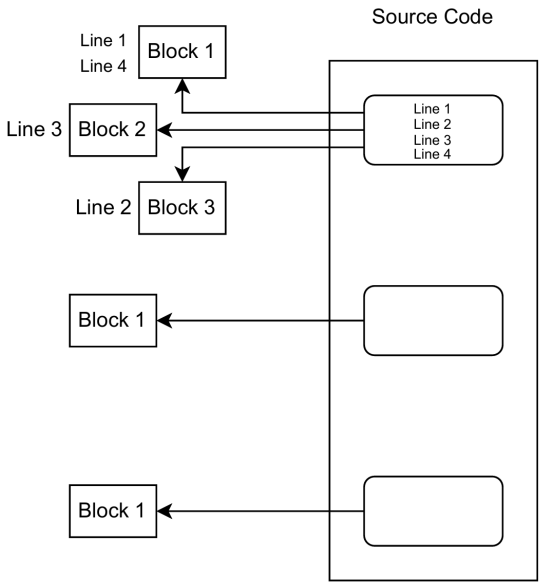

Figure 1. Source Code in Functions is Divided into Distinct Logical Blocks

3.6. Comparison Method

To compare semantic and traditional testing signals, we evaluate each function based on:

- **Statement Coverage:** $\frac{\text{\# lines executed}}{\text{\# total lines}}$
- **Semantic Score:** Weighted score based on covered blocks.
- **Normalized Semantic %:** $\frac{\text{sem\_score}}{\text{max possible score}}$

At the project level, we compute the mean of all individual functions' statement and semantic coverage values. This allows project-wide trends to be visualized and analyzed.

3.7. Implementation Details and Workflow

This section outlines the complete process for computing semantic coverage from a Python project. Each step ensures the system remains modular, flexible, and scalable across small- and large-scale code-bases.

3.7.1. Pipeline Overview

Step 1: Project Preparation.

- The source code and test files of the Python project are placed within a structured directory.
- The directory is passed to the main driver script via a command-line argument.

Step 2: Function Extraction.

- The script recursively traverses the source directory and collects all `.py` files, excluding hidden or test files.
- Using abstract syntax tree (AST) parsing, all function definitions are extracted from the collected files.

Step 3: Line Numbering.

- Each function is assigned line numbers for each line of executable code (excluding comments and docstrings).
- This allows each code line in any function to be uniquely identified during semantic block segmentation and coverage matching.
- Example:

```
#1 def validate_user(data):
#2     if not isinstance(data, dict):
#3         raise ValueError("Expected a dictionary.")
#4     return True
```

Step 4: Logical Block Segmentation via GPT-4.

- Numbered functions are converted into a specific format ("Function X: ...") and batched.
- Each batch is submitted to GPT-4 using a carefully designed prompt that instructs the model to categorize each line into one of the following logical blocks:
  - Core Functionality
  - Boundary Conditions and Edge Cases
  - Error Handling
  - Integration Points
  - Security Features
  - Performance and Scalability
  - Output Consistency
  - Configuration and Environment
  - User Interface Interactions
- To maintain full traceability, each line of code retains its unique line number (e.g., `#17 return x`). This line numbering plays a crucial role in distinguishing identical lines that may appear in different contexts within the same or multiple functions. For example, two identical lines like `return True` will each have a unique identifier based on their position, ensuring accurate coverage tracking even if one is tested and the other is not.

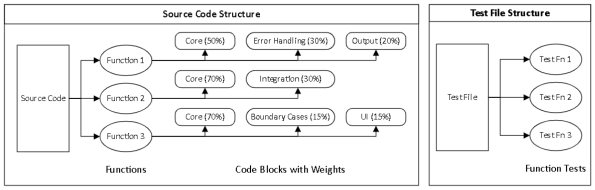

Figure 2. Overall File Structure

- The full prompt given to GPT-4 enforces strict formatting, forbids skipping lines, and ensures that lines are grouped into no more than five blocks per function. The output is parsed to extract block definitions for each function.
- After receiving the categorized output from GPT-4, the system creates individual files for each logical block for all functions as shown in Figure 1. These block files are stored in the `output_blocks/` directory. Every file is named using the following convention:

  `<function_name>_<block_type>.py`

  For example:

  - `game_play_core_functionality.py`
  - `deal_card_error_handling.py`

  This naming structure makes it extremely convenient to locate and map test coverage to the right semantic block. Since each file represents a distinct logical objective, we can now evaluate the extent to which that specific purpose is covered by the test suite.

Step 5: Statement Coverage Collection.

- Unit tests are executed using `coverage.py` to gather line-level statement coverage.
- The generated `.coverage` file is parsed to extract all lines executed during testing.
- These lines are mapped back to their corresponding functions and logical blocks using the line numbers from Step 3.

Step 6: Semantic Coverage Computation.

- Each logical block is assigned a predefined weight based on criticality:

Listing 1. Semantic coverage formula for each block

```
stmt_coverage = (covered_count / len
    (func_lines))
block_type =
    extract_block_type_from_filename(
    block_file)
weight = BLOCK_WEIGHTS.get(
    block_type)
block_semantic_score = stmt_coverage
     * weight
```

  - `stmt_coverage = (covered_count / len(func_lines))`: Calculates the raw statement coverage of the block by dividing the number of covered lines by the total number of executable lines in the block.
  - `extract_block_type_from_filename (block_file)`: Extracts the logical block type (ex: `core_functionality`, `error_handling`) from the block file's name.
  - `BLOCK_WEIGHTS.get(block_type)`: Retrieves the predefined importance weight assigned to the block type based on its criticality.
  - `block_semantic_score = stmt_coverage * weight`: Computes the final semantic score for the block by multiplying its statement coverage with the importance weight.

Step 7: Normalized Comparison.

- According to Figure 2, functions semantic scores are calculated after running the test files.
- Both semantic and statement coverage are normalized to percentage scores for each function.
- Aggregated project-level averages are also computed to compare test effectiveness across the entire codebase.

4. Results

This section presents semantic and statement coverage findings for two projects of varying complexity:

- **Small-scale project:** A Blackjack-style card game implemented in Python.
- **Large-scale project:** A graphical Python-based Bubble Shooter game.

4.1. Small-Scale Project: Card Game

This project included four core functions with accompanying unit tests. We applied our semantic segmentation and weighting method to analyze coverage effectiveness.

Table 2. Function-Level Semantic vs. Statement Coverage

Func	Stmt %	Sem	Sem %
`deal_card`	90.00	25.40	94.07
`compare`	86.67	24.14	89.42
`calculation_total`	100.00	10.00	100.00
`game_play`	95.65	18.67	93.33

4.1.1. Function-Level Coverage Scores

From the data shown in Table 2 and the Bar Chart in Figure 3, it is clear that despite lower statement coverage in some functions (ex: `compare()`), semantic scores remained high due to critical logic blocks being tested. In contrast, in `game_play()`, we observed a relatively high statement coverage (95%) but a lower semantic score (93%). This highlights a key insight: high statement coverage does not guarantee that the most important or complex logic is thoroughly tested. While the semantic coverage is still reasonable in this case, it reveals gaps in critical-path testing that may be obscured by raw line execution metrics.

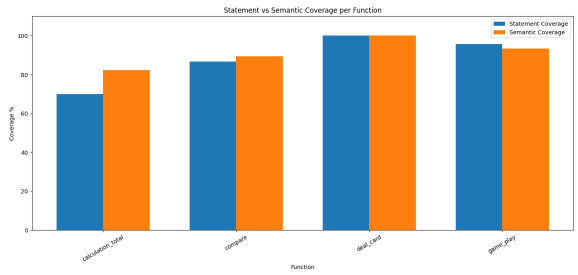

Figure 3. Comparison of Statement and Semantic Coverage for the Smaller-Scaled Project

4.1.2. Project-Level Summary

- **Avg. Statement Coverage:** 93.08%
- **Avg. Semantic Coverage:** 94.21%

This highlights that the project had its critical components tested out quite thoroughly.

4.2. Large-Scale Project: Bubble Shooter Game

This project includes multiple modules totaling over 200 functions and 150 unit test.

4.2.1. Function-Level Coverage Scores

Through Table 3 as well as Figure 4, it is evident that despite `run` exhibiting a high traditional statement coverage of 95.45%, its semantic score is drastically lower at 16.67%. This contrast reveals that raw line execution does not necessarily reflect meaningful testing. Upon examining the corresponding block files and coverage data, it was observed that only the `user_interface_interactions` block was mostly executed during the tests. While this block contributed the majority of lines, the more critical `core_functionality` block, though smaller in size, was not tested at all. As a result, the semantic coverage metric appropriately penalized this discrepancy, showing the lack of coverage for critical logic. This shows the advantage of semantic coverage: it uncovers testing blind spots masked by the inflated statement coverage, allowing more informed quality assurance.

Table 3. Semantic vs. Statement Coverage for Select Functions

Function	Stmt %	Sem	Sem %
`run`	95.45	2.00	16.67
`scale_message`	0.00	0.00	0.00
`scan_bubbles`	100.00	18.00	100.00
`select_func`	83.33	15.33	85.19

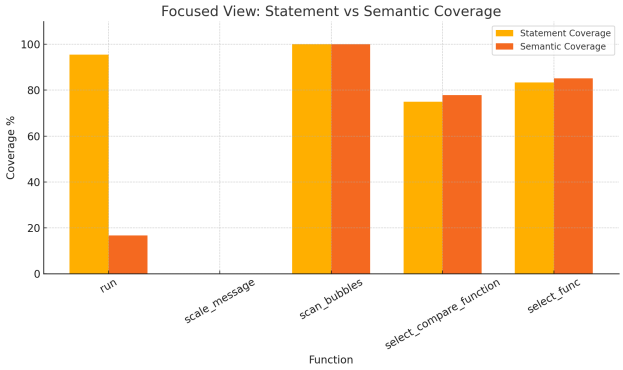

Figure 4. Comparison of Statement and Semantic Coverage for the Large-Scaled Project

4.2.2. Project-Level Summary

- **Avg. Semantic Coverage:** 76.58%

This highlights that the project had its critical components tested out moderately.

4.3. Insights & Interpretation

- **Semantic coverage reveals critical gaps:** Even with low statement coverage, functions testing win conditions or edge cases scored high semantically.
- **UI-heavy modules inflate statement metrics:** These skew traditional coverage results by emphasizing setup or rendering logic over functional correctness.
- **Block-weighted metrics support better prioritiza-**

tion: QA efforts can now target untested but high-weight blocks first.

5. Future Work

While this work introduces a promising shift toward context-aware test analysis, several areas remain for refinement and expansion:

- **Block Type Exploration:** Current block categories are derived from domain knowledge and intuition. Further research is needed to systematically study how functions should be segmented. Are there additional or alternative block types that better reflect function intent?
- **Formal Segmentation Metrics:** A deeper dive is necessary into what makes an effective segmentation. Can we create evaluation metrics for segmentation quality beyond accuracy and recall? How can we measure the semantic boundaries of logic blocks?
- **Error Robustness:** The system occasionally fails due to minor syntax issues or unconventional symbols in the code. More robust parsing and preprocessing techniques are needed to support a wider range of source formats and edge cases.
- **Multilingual Code Support:** Future iterations should extend the framework to support languages beyond Python, such as JavaScript, Java, or C++.
- **Larger and Real-World Datasets:** A more diverse dataset, especially pulled from more varied repositories could strengthen model generalization.
- **Block Weight Justification:** Another promising direction is the dynamic validation or adjustment of block importance weights. In the current approach, weights assigned to code segments (ex: Core Functionality, Error Handling) are hard-coded based on intuition and domain knowledge. However, a more robust method could involve using Large Language Models (LLMs) or static analysis tools to detect patterns in how different block types influence the function's overall behavior.

  For example, constructing a function-level dependency graph could highlight which segments other blocks rely on most heavily. By analyzing these dependencies, the model could automatically infer importance scores, assigning higher weights to blocks with greater centrality or impact. Alternatively, an LLM trained on annotated data could learn to predict block importance based on context, usage, and patterns observed across large code-bases. This would enable a data-driven reweighting mechanism, enhancing both the accuracy and adaptability of semantic coverage computations.

6. Conclusion

Traditional coverage metrics such as statement, branch, and function coverage often fail to capture the semantic significance of different parts of a program. In this paper, we introduced *Semantic Coverage*, a novel approach that leverages Large Language Models (LLMs) to segment code into logical blocks and assign weighted importance to each based on its role within the function.

By combining LLM-based segmentation with coverage tools like `coverage.py`, a weighted semantic score that provides a more meaningful reflection of test effectiveness was calculated. The results show that semantic coverage can identify cases where low statement coverage still yields high-value testing by focusing on critical functionality.

In this paper, We also explored the challenges associated with LLM-generated block categorization, including accuracy and consistency. Additionally, we discussed the potential for future enhancements, such as dynamically reweighting block importance using data-driven techniques like dependency graphs or learned heuristics from labeled corpora.

Ultimately, Semantic Coverage provides a deeper understanding of software test quality. Rather than measuring how much of the code is tested, it focuses on what exactly is being tested more importantly. This approach opens up promising pathways for more intelligent test prioritization, particularly in safety-critical systems such as medical devices.

References

Dormuth, J., Gelman, B., Moore, J., and Slater, D. Logical segmentation of source code. In *Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering (SEKE 2019)*, pp. TBD, Lisbon, Portugal, 2019. KSI Research Inc. doi: 10.18293/SEKE2019-026.

Santos, R., Santos, I., Magalhaes, C., and de Souza Santos, R. Are we testing or being tested? exploring the practical applications of large language models in software testing. In *Proceedings of the 2023 International Conference on Software Engineering (ICSE 2023)*, pp. TBD, Melbourne, Australia, 2023. IEEE.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. Software testing with large language models: Survey, landscape, and vision. In *Proceedings of the 2024 IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*, pp. TBD, Curitiba, Brazil, 2024. IEEE.