Assessing CodePilot's Breadth

Justin Zhang*
JustinZhang@cmail.carleton.ca
Carleton University
Ottawa, Ontario, Canada

Kaya Gouin*
KayaGouin@cmail.carleton.ca
Carleton University
Ottawa, Ontario, Canada

Abstract

...

Keywords

Software engineering, prompt engineering, artificial intelligence, few-shot

ACM Reference Format:
Justin Zhang and Kaya Gouin. 2024. Assessing CodePilot's Breadth. In *Mining Software Repositories, December 2024, Ottawa, ON*. ACM, Ottawa, ON, Canada, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In software development, code coverage is a critical metric that provides insight into the extent to which source code is tested, helping to identify untested areas and enhancing overall code quality. Traditional code coverage tools require a program to be fully executable, meaning they are unable to analyze incomplete or partially developed code. This requirement limits their utility during iterative development stages, where obtaining coverage feedback early in the process could be valuable in guiding subsequent coding and testing efforts.

Recent advancements in large language models (LLMs) offer a promising alternative for addressing these limitations. By leveraging natural language processing capabilities, LLMs can analyze code structure and semantics to estimate code coverage, even in the absence of compilation or execution. This potential makes LLMs especially useful for providing insights into coverage for partial programs, thereby supporting developers in assessing and improving coverage throughout the development process.

Our study extends prior work in this area, specifically building upon Codepilot, a prompt-crafting technique that explored code coverage estimation through LLMs for a single programming language. In Codepilot, LLMs were evaluated using a combination of few-shot, one-shot, and zero-shot prompting techniques, as well as different phases of query structuring. The two-phase approach involved generating a natural language plan for code coverage in the first phase, then feeding the plan back into the LLM to retrieve the actual coverage. The single-phase approach directly prompted the LLM to generate both the plan and the coverage in a single prompt. Codepilot identified the few-shot one-phase method as achieving the best results, suggesting that a single, context-rich prompt could effectively guide the LLM in understanding the code and estimating its coverage.

Expanding on these findings, our study applies the few-shot, one-phase prompting technique to a broader set of programming languages, encompassing five paradigms to investigate the adaptability and robustness of LLM-driven code coverage estimation. Specifically, we examine languages from functional, procedural, object-oriented, logic-based, and scripting paradigms: Haskell, C, Java, Prolog, and Python, respectively. Each language presents unique syntactic and semantic characteristics that represent the diverse ways in which code structures and logic are expressed in programming. By testing the few-shot prompting method across languages of varied paradigms, we aim to evaluate how well LLMs can generalize across programming styles, potentially extending the applicability of LLM-based code coverage to a wide range of software development environments.

This study is motivated by the growing demand for flexible tools that can aid developers working in multi-paradigm environments, where traditional coverage tools may not be consistently effective or available. In particular, understanding the performance of LLMs in handling coverage for diverse languages could inform their use in real-world settings where developers often encounter multiple paradigms within a single project. Our research aims to uncover insights into the strengths and limitations of LLMs as tools for cross-language code coverage and to contribute to the body of knowledge on LLM applications in software engineering. By demonstrating the potential for LLMs to provide coverage feedback for programs that are incomplete or non-executable, we hope to support the development of tools that enhance software quality and reduce risks associated with untested code across programming paradigms.

2 Functional Programming

Functional programming is a paradigm that emphasizes computation through the evaluation of functions and avoids changing states or mutable data. In this paradigm, programs are structured around the application of mathematical functions, where each function takes an input and returns a new value without altering any existing data. Haskell, a prominent functional language, embodies these principles with features like pure functions, immutability, and lazy evaluation (i.e., values are computed only when needed). This paradigm is known for its high level of abstraction and expressiveness, making it suitable for complex computations and enabling more predictable, side-effect-free code. Haskell's functional characteristics allow for concise and expressive code, but

*Both authors contributed equally to this research.

COMP 5117, December 2024, Ottawa, ON
2024. ACM ISBN XXX-X-XXXX-XXXX-X/24/12
https://doi.org/XXXXXXX.XXXXXXX

these features can be challenging for conventional testing and code coverage tools to interpret due to the lack of mutable states and traditional control flow.

3 Procedural Programming

Procedural programming is a paradigm based on the concept of procedure calls, where a program is structured as a sequence of steps or instructions. It relies on routines, such as functions or subroutines, to perform tasks, and it emphasizes a clear, step-by-step structure that progresses through control flows like loops and conditionals. C is one of the most widely used procedural languages, offering low-level memory access and efficient execution. It is often favored in systems programming and applications requiring fine-grained control over hardware. Code in C is typically organized through procedures that manipulate data in variables, making code execution straightforward but sometimes complex in terms of tracking memory usage and managing pointer operations. Code coverage in procedural languages generally focuses on line or statement coverage, as well as function and branch coverage.

4 Object-Oriented Programming

Object-oriented programming (OOP) is centered around the concept of "objects"—data structures that combine data fields with methods for operating on the data. This paradigm promotes modularity and reusability by organizing code into classes and objects, each representing an instance of a class. Java is a quintessential object-oriented language that encapsulates data and methods within classes, using principles like inheritance, polymorphism, and encapsulation to model real-world entities and relationships. OOP facilitates code reuse, flexibility, and maintainability, making it popular in large-scale software development. However, the interconnected nature of objects and class hierarchies introduces additional complexity for code coverage, as tests must account for interactions among objects and potential inheritance chains that influence behavior across classes.

5 Logic Programming

Logic programming is a paradigm based on formal logic, where programs consist of a set of rules and facts. Computation in logic programming occurs through pattern matching and logical inference rather than sequential execution. Prolog, the primary language in this paradigm, uses a declarative approach in which problems are defined by relationships (facts) and rules, and the program searches for solutions that satisfy these logical constraints. This paradigm is well-suited to tasks in artificial intelligence and complex problem-solving, where solutions are derived through logical reasoning. In Prolog, code coverage involves evaluating how effectively rules and facts are tested, which can differ significantly from coverage metrics in imperative languages. Due to Prolog's declarative nature, coverage analysis focuses on ensuring that logical paths and rule conditions are adequately exercised.

6 Scripting and Dynamic Programming

Scripting languages, often dynamically typed, focus on automating tasks and performing higher-level operations. Python, a popular scripting language, supports multiple programming paradigms, including procedural, functional, and object-oriented styles, offering flexibility in writing code. Python is dynamically typed, meaning that data types are inferred at runtime, which allows rapid development but can introduce runtime uncertainties. Its versatility and ease of use make Python a favored choice in fields such as data analysis, web development, and machine learning. Code coverage in Python often involves testing code paths and ensuring that dynamic features like runtime type inference and exception handling are adequately covered. The language's dynamic nature and support for multiple paradigms present unique challenges for comprehensive coverage analysis, especially in understanding the interactions between different programming styles within the same codebase.

7 Assessing CodePilot's Breadth

Introduce our own work, and the motivation behind it, by talking about the fact that CodePilot is very new and has only been tested with Python, so very limited. Here we want to explore the breadth of CodePilot by testing it on different types of programming languages. We are interested in assessing both the quality of the plan that the model outputs, the quality of the code coverage prediction that the model outputs, and the link between the two (i.e. does it look like the model uses its own plan to predict code coverage). These are our research questions.

8 Methodology

Five different programming languages, each belonging to a different type of language class. Onephase. Two-shot, so we give two exemplars to the model. Two tests for each language. We match the exemplar and test language. We wanted small exemplars with some level of complexity. The two exemplars for all languages are implementations of the same algorithm, one with a certain level of complexity. The two tests for all languages are implementations of the same algorithm, one with a certain level of complexity. Talk about the exact algorithms. We run each experiment once. Temperature of 0.6. GPT instruct 3.5. The template that we use (the same as in CodePilot's original paper).

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

```
Input:

For the given code snippet, predict the code coverage.
The code coverage indicates whether a statement has been
executed or not.

> if the line is executed
! if the line is not executed

Example output:
> line1
! line2
> line3
...
> linen

You need to develop a plan for step by step execution of
the code snippet.

Do not answer unless instructed to do so

DISCLAIMER: Lines that are not executed are to be denoted
with a SINGLE '!' whereas lines that are executed are to be
denoted with a single '>'

Below are a couple examples of the process you need to follow
to predict the code coverage of a given code snippet and its
plan.

[example 1]

[example 2]

In a similar fashion, develop a PLAN of step by step execution
of the below code snippet and predict the CODE COVERAGE.

[code snippet]

Output:

[output plan]
[output code coverage]
```

Figure 1: Template

```
maxProfit :: [Int] -> Int
maxProfit prices = maxProfitHelper prices (maxBound :: Int) 0
  where
    maxProfitHelper [] _ maxProfit = maxProfit
    maxProfitHelper (p:ps) minPrice maxProfit =
      let newMinPrice = min minPrice p
          profit = p - newMinPrice
          newMaxProfit = max maxProfit profit
      in maxProfitHelper ps newMinPrice newMaxProfit
main = do
    let prices = [7, 1, 5, 3, 6, 4]
    let result = maxProfit prices
    putStrLn $ "Maximum profit: " ++ show result
```

Figure 2: Test

```
Given PLAN:
STEP 1: Define a function maxProfit which takes an array of
integers as input and produces an integer as output.
STEP 2: In the main function, let the "prices" variable hold
the input array of integers [7, 1, 5, 3, 6, 4].
STEP 3: In the main function, let the "result" variable hold
the output of maxProfit given the input array of integers.
STEP 4: In the maxProfit function, initiate the conditional
checks with parameters prices, (maxBound :: Int), and 0.
STEP 5: Define two checks. Check 1 verifies whether the first
parameter is an empty list and the second parameter is the
maximum bound of the integer type. Check 2 verifies whether
the first parameter is a non-empty list. These checks are
executed in the order in which they appear. In the case of the
[7, 1, 5, 3, 6, 4] input array, given the ordering of the checks,
only check 2 is executed.
STEP 6: In the maxProfit function, define a helper function
maxProfitHelper which takes three parameters: an array of
integers, an integer representing the minimum price, and an
integer representing the maximum profit. This helper function
recursively calculates the maximum profit by updating the
minimum price and maximum profit values based on the current
element of the input array.
STEP 7: In the main function, print the resulting maximum profit
to the screen.

So the CODE COVERAGE for the given code snippet will be:
> maxProfit :: [Int] -> Int
> maxProfit prices = maxProfitHelper prices (maxBound :: Int) 0
>   where
>     maxProfitHelper [] _ maxProfit = maxProfit
>     maxProfitHelper (p:ps) minPrice maxProfit =
>       let newMinPrice = min minPrice p
>           profit = p - newMinPrice
>           newMaxProfit = max maxProfit profit
>       in maxProfitHelper ps newMinPrice newMaxProfit
> main = do
>     let prices = [7, 1, 5, 3, 6, 4]
>     let result = maxProfit prices
>     putStrLn $ "Maximum profit: " ++ show result
```

Figure 3: Output

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

9 Threats to Validity

In any empirical study involving machine learning models, particularly large language models (LLMs), certain threats to validity may impact the reliability and generalizability of the findings. This study's exploration of LLM-driven code coverage across multiple programming paradigms faces several specific challenges, outlined below. One major threat to validity is the non-deterministic nature of LLMs. Due to the stochastic nature of LLMs, the same prompt may yield different outputs across multiple runs. This characteristic introduces variability in the code coverage results generated

```
Given CODE SNIPPET:
isValidParentheses :: String -> Bool
isValidParentheses str = check str 0
  where
    check [] 0 = True
    check [] _ = False
    check (')':xs) n = n > 0 && check xs (n - 1)
    check ('(':xs) n = check xs (n + 1)
    check (_:xs) n = check xs n
main = do
  let result = isValidParentheses "(()"
  print result

Given PLAN:
STEP 1: Define a function isValidParentheses which
takes a string as input and produces a boolean as output.
STEP 2: In the main function, let the "result" variable
hold the output of isValidParentheses given the string
"(()".
STEP 3: In the isValidParentheses function, initiate the
conditional checks with parameters str and 0.
STEP 4: Define five checks. Check 1 verifies whether the
parameters are an empty list and the value 0. Check 2 verifies
whether the parameters are an empty list and any value other
than 0. Check 3 verifies whether the first character of the
first parameter is a right parenthesis. Check 4 verifies
whether the first character of the first parameter is a left
parenthesis. Check 5 verifies whether the first character of
the first parameter is any character other than a right or a
left parenthesis. These checks are executed in the order in
which they appear. We stop executing the next check as soon as
we find one which satisfies the correct condition. In the case
of the "(()" input string, given the ordering of the checks,
all checks are executed.
STEP 5: In the main function, print the resulting array to the
screen.

So the CODE COVERAGE for the given code snippet will be:
> isValidParentheses :: String -> Bool
> isValidParentheses str = check str 0
>   where
>     check [] 0 = True
>     check [] _ = False
>     check (')':xs) n = n > 0 && check xs (n - 1)
>     check ('(':xs) n = check xs (n + 1)
>     check (_:xs) n = check xs n
> main = do
>   let result = isValidParentheses "(()"
>   print result
```

Figure 4: Exemplar

by the LLM, which could impact the consistency of coverage estimates. This variability presents a challenge in replicating results and verifying the reliability of the coverage analysis across programming languages. Although our study uses a few-shot, one-phase prompting method, which has been shown to yield more consistent outputs than zero-shot approaches, there remains a risk that minor fluctuations in output may lead to inconsistencies in coverage analysis, particularly across languages with vastly different syntax and structure. Another potential threat is prompt sensitivity, the results generated by LLMs are highly sensitive to prompt construction, which can influence the accuracy and quality of the code coverage estimations. Minor modifications to prompts can produce significant changes in LLM responses, potentially affecting the results' validity. In this study, we apply a carefully designed few-shot prompting technique; however, prompt engineering remains an inherently subjective process. The prompts used in our study might not generalize to other types of code or languages, limiting the applicability of our findings. Additionally, as prompt engineering evolves, new techniques may emerge that produce different results, posing a potential threat to the study's replicability.

10 Mitigation

To address these threats, we have standardized prompt templates and experimented with prompt refinement to minimize variability. Additionally, by using a few-shot, one-phase prompting approach, we aim to reduce inconsistencies across runs, as this method has demonstrated higher stability than zero-shot or multi-phase alternatives. However, we acknowledge that the inherent variability of LLMs limits the extent to which these mitigations can fully eliminate the outlined threats.

11 Future Directions

This study represents an initial exploration into the use of large language models (LLMs) for code coverage estimation across multiple programming languages and paradigms. Given the scope and limitations of our current research design, several potential extensions could further enrich the findings and address the limitations noted. In our current study, we used only two examples in our few-shot prompting approach, which provided minimal context for the LLM. A natural extension would be to increase the number of examples to further ground the LLM's understanding of code coverage expectations. Adding more examples across diverse programming constructs and paradigms could improve the accuracy and reliability of coverage estimations, as a broader context may help the LLM generalize better to new, unseen code snippets. This could include examples featuring more complex control flows, various levels of nesting, and different algorithmic structures within each programming language. Additionally, experimenting with multi-shot prompts could allow us to analyze how many examples are optimal for consistent coverage estimations across paradigms.

Our current analysis focused on simple, isolated code snippets to test the feasibility of LLM-driven coverage estimation. Extending the study to encompass more complex, real-world codebases—such as multi-module applications or scripts with extensive dependencies—would present a valuable next step. By evaluating the LLM's performance on these larger and more representative code samples, we could better assess the model's applicability to practical software development scenarios. Testing on complex programs would also allow for the exploration of how well the LLM handles dependencies, external libraries, and interactions between modules.

Given the inherent non-determinism of LLMs, running multiple trials for each code snippet would allow us to observe and document the variability in coverage estimations. By collecting and analyzing a distribution of coverage outputs for each snippet, we could assess the stability and reliability of LLM-driven code coverage. Plotting this distribution would provide insights into the consistency of the model's estimations and allow us to determine whether certain programming paradigms or language features result in higher variability. This extension could contribute valuable data for understanding how often LLMs yield inconsistent results and how best to mitigate this variability in practical applications.

Another valuable extension would involve benchmarking LLM-driven coverage against traditional code coverage tools (e.g., coverage.py, gcov) in multi-language, real-world applications. By comparing the coverage metrics generated by LLMs with those from execution-based tools, particularly on large codebases with mixed-paradigm languages, we could assess the feasibility of LLMs as a viable alternative to traditional coverage methods. This comparison would highlight the advantages and limitations of using LLMs in practical scenarios, identifying cases where LLMs might complement or substitute traditional tools, especially in early development stages.

Since LLMs are capable of handling multiple programming languages within the same prompt context, an extension of this study could focus on how well LLMs generalize code coverage techniques across paradigms in more nuanced ways. For instance, testing the LLM's ability to transfer knowledge of code coverage patterns from one language (e.g., Python) to another (e.g., Prolog) could provide insights into its cross-language adaptability. This extension would be particularly useful in evaluating whether the LLM can consistently apply code coverage logic across different paradigms or if paradigm-specific adjustments are necessary.

12 Results and Discussion

Achieved results, and our interpretation of them.

13 Conclusion and Future Directions

Summary of main contributions. Why this study matters, how can our findings be used. Generally speaking we see that prompt and exemplar engineering is necessary in the sense that it really has an impact on the model's output, and they must be tailored to the specific task, the specific language.