Planning to Guide LLM for Code Coverage Prediction

Hridya Dhulipala
University of Texas at Dallas
Dallas, USA
hridya.dhulipala@utdallas.edu

Aashish Yadavally
University of Texas at Dallas
Dallas, USA
aashish.yadavally@utdallas.edu

Tien N. Nguyen
University of Texas at Dallas
Dallas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Code coverage serves as a crucial metric to assess testing effectiveness, measuring the degree to which a test suite exercises different facets of the code, such as statements, branches, or paths. Despite its significance, coverage profilers necessitate access to the entire codebase, constraining their usefulness in situations where the code is incomplete or execution is not feasible, and even cost-prohibitive. In this paper, we present CodePilot, a plan-based prompting approach grounded in program semantics, which collaborates with a Large Language Model (LLM) to enhance code coverage prediction. To address the intricacies of predicting code coverage, CodePilot employs planning by discerning various types of statements in an execution flow. Planning empowers GPT to autonomously generate plans based on guided examples, and then CodePilot prompts the GPT model to predict code coverage (Action) based on the plan it generated (Reasoning). Our experiments evaluating CodePilot demonstrate high accuracy, achieving up to 55% in exact-match and 89% in statement-match. It performs relatively better than the baselines, achieving up to 33% and 19% relatively higher in those metrics. We also showed that due to highly accurate plans (90%), GPT model predicts better code coverage. Moreover, we show CodePilot's utility in correctly predicting the least covered statements.

CCS CONCEPTS

• Computing methodologies → Neural networks; **• Software and its engineering → Software reliability**;

KEYWORDS

AI4SE, Large Language Models, Planning, Code Coverage Analysis

ACM Reference Format:
Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. 2024. Planning to Guide LLM for Code Coverage Prediction. In *AI Foundation Models and Software Engineering (FORGE '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3650105.3652292

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated considerable success in various code-related tasks, showcasing their ability to generate coherent and contextually relevant code snippets. These models, such as GPT [12], have been particularly effective in tasks like code completion, summarization, and translation. However, their success has limitations when it comes to understanding the intricacies of program semantics for code execution [30]. Issues such as program execution exploration, value changes, loop unrolling, inter-procedural calls, memory handling and pointers pose challenges that current LLMs struggle to grasp adequately.

The limitations in LLMs' prediction of program semantics on code execution are analogous to the challenges faced in complex tasks within robotics or natural-language processing in which the LLMs must *reason and select a large range of decisions and actions.* To address these limitations, machine learning (ML) research explores the integration of *planning* techniques to guide LLMs [13, 22, 25, 32, 34]. Planning serves as a strategic tool to help navigate and guide the LLM through the complexities of intricate tasks.

In the realm of complex tasks involving program semantics for code execution, we advocate for the planning strategies grounded in program analysis (PA), that we call **PA-based planning**. This planning aims to leverage insights from program analysis to guide the LLM in navigating the intricate exploration space inherent in complex prediction tasks for code. By incorporating PA-based planning, we anticipate that LLMs can better comprehend and address the nuanced aspects of program behavior, thereby improving their performance in tasks involving intricate code execution.

One of such complex tasks is in the area of software testing. A widely adopted technique for gauging testing efficacy is *code coverage*, which evaluates the comprehensiveness of testing endeavors and provides a level of assurance that the system will adhere to predefined specifications. Various code coverage metrics exist, each offering a unique perspective. Line coverage measures the percentage of executed lines in tests compared to the total lines in the code. Statement coverage concentrates on the number of individual statements covered. Another code coverage type is branch coverage, commonly known as decision coverage. This metric assesses the degree to which different decision points in a program's source code have been tested or covered by test cases.

Measuring code coverage for a specific code snippet requires access to the entire program containing that snippet. This constraint becomes apparent in scenarios where only partial code is available, such as in a code snippet posted on an online forum, a commit log or code diff, or when transmitting partial code to a server due to security considerations. In other instances, executing tests may be undesirable or excessively resource-intensive. For example, developers may need to prioritize and run only a subset of a test suite due to constraints like limited time or resources. It becomes crucial for them to understand which test cases cover specific parts of the source code before actual execution. The decision regarding test case prioritization could hinge on the code coverage information, e.g., which areas of the code base are least or most covered by the existing test suites *before actual execution.*

FORGE '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0609-7/24/04
https://doi.org/10.1145/3650105.3652292

Tufano *et al.* [30] propose a method that leverages zero-shot and few-shot prompts within LLMs, particularly GPT [12], to compute code coverage for any code snippet without actual execution. However, this approach exhibits reduced accuracy and experiences performance decline when dealing with the challenges of navigating a large and intricate space of multiple interdependent execution steps within a program. CodeExecutor [16] is a Unixcoder-based neural network model that is pre-trained on the execution of a large number of programs to predict the execution traces. Despite pre-training on execution traces, CodeExecutor still suffers low accuracy in understanding complex execution spaces.

In this paper, we introduce CodePilot, a prompting approach based on program semantics that collaborates with the LLM model to predict code coverage. Specifically, to tackle the intricacies of predicting code execution for coverage, we leverage *planning*. Planning is pivotal, enabling an LLM to autonomously generate plans based on exemplars. CodePilot's planning for code coverage is grounded in program semantics, capturing the subtleties of the execution of each statement. CodePilot integrates the synthesis of Reasoning and Acting to predict the code coverage of a given code snippet. The reasoning component encompasses the program-flow steps that would have occurred had the code been executed. Each step in the reasoning provides a concise explanation of whether a statement would or wouldn't have been executed. This reasoning process guides the LLM, directing its attention to detailed execution steps in its own plan first by discerning various types of statements in a control flow. Then, the LLM is requested to predict code coverage (Action) based on the plan generated by itself (Reasoning).

Our experiments demonstrate CodePilot's high accuracy, achieving up to 55% in exact-match and 89% in statement-match. CodePilot performs relatively better than the baselines, achieving up to 33% and 19% relatively higher in those metrics. We also showed that with CodePilot, the GPT model produces highly accurate plans (90%), leading to better code coverage prediction. Moreover, we showcase CodePilot's utility in accurately predicting the least covered statements. In brief, the contributions of this paper include:

1. [CodePilot: Planning with LLM for Code Coverage]. CodePilot is the first using planning with LLM to support execution-aware tasks. It leverages GPT+Planning for code coverage.

2. [PA-based Planning with Reasoning and Actions] Our program semantics-based planning enables an LLM to harness its capacity to generate its own plan for code coverage computation.

3. [Empirical Evaluation] We conducted several experiments to show that CodePilot performs better than the state-of-the-art code coverage prediction models. Data and code is available [1].

2 MOTIVATION

2.1 Motivating Example

Let us start with a small experiment to illustrate the problem with the Large Language Models (LLMs) and motivate our approach. Figure 1(a) displays our first experiment with a zeroshot prompt to GPT-3.5 requesting it to compute the code coverage for the listed Python code at lines 17–37. The output from GPT-3.5 is displayed in Figure 3 in the column marked by `"Zero"` (zeroshot). As seen in Figure 3, GPT-3.5 predicted the code coverage incorrectly. It predicted all the statements to be executed, including both branches of the `if-else` statements at line 18 and line 31. This is unacceptable since only one branch will be covered for a user number (line 29).

In the next step of our experiment, we attempted to aid GPT-3.5 with an exemplar, which is referred to as a one-shot prompt. The exemplar is shown in Figure 1(b), which differs from the test code in Figure 1(a). The inclusion of an exemplar in the prompt ideally aims to assist the LLM in comprehending the problem statement and the test code more precisely. However, the code coverage result from the one-shot prompt was still largely incorrect (see the column marked with `"One"` in Figure 3). In fact, GPT-3.5 successfully identified the 'else' statements as not-executed (line 5, Figure 3) but erroneously predicted the statements within the 'else' block as executed (line 6, Figure 3). This is a failure from a code execution perspective.

While LLMs have demonstrated impressive performance across tasks in source code and language understanding, it is still challenging for them to capture and reason on the nuances of program execution, especially with very large exploration spaces of execution paths [30]. Aiming to deal with such complexity, we leverage an advanced prompt engineering technique, called *planning* [13, 22, 25, 32, 34], which helps the model devise a reasoning plan to guide itself through reasoning traces and actions.

Since planning has been achieving success in the robotics [25, 34] and natural language processing fields (NLP) [32] to overcome the prevalent issues of hallucination and error propagation in LLMs, we aim to bring the planning concept into guiding GPT-3.5 in an execution-aware task. Specifically, we leverage program semantics to manually draft a reasoning-and-action plan in an exemplary code. For this example, we created such a plan shown in Figure 2 for the exemplary code in the one-shot prompt in Figure 1(b). This plan serves as an illustrative and guiding reference for GPT-3.5, outlining a step-by-step procedure for processing the test code, in which the steps that would have been taken had the program or code snippet been actually executed. Each step encompasses the execution of a code snippet and accompanied by a concise rationale explaining why a statement or set of statements will or will not be executed. For example, the plan could focus on 1) the branching statements (e.g., `if`, `switch`) in which the execution could go to either branch, 2) the iteration statements (e.g., `for`, `while`) in which the execution could be repeated), and 3) the method calls in which the execution could become inter-procedural. The manually drafted, exemplary plan depicted in Figure 2(a) corresponds to a code snippet that encompasses the guidance for those types of statements.

As illustrated in Step 5 of Figure 2(a), the plan explains the reasoning steps to predict the execution of a `for` loop with the running variable `i` from 1 to N. The plan for guiding GPT-3.5 with respect to the `if-else` statements can be seen in Step 6 and Step 9 of Figure 2(a). The plan succintly outlines the condition and the statement executed accordingly to the condition mentioned in the `if-else` statement. Moreover, the plan regarding a method invocation is shown in Steps 10–11 of Figure 2(a). Upon invoking a method, the plan guides the model to progress to the subsequent step, involving the hypothetical execution of the called method.

In addition to the guidance for those statements, the plan accommodates supplementary statements found in a code snippet, such as variable initialization and print statements.

In this experiment, we first provided the prompt consisting of the exemplary plan in Figure 2(a) for the exemplar code in Figure 1(b).

```
1  For the given code snippet, Predict the code coverage. The code coverage indicates
          whether a statement has been executed or not.
2  > if the line is executed
3  ! if the line is not executed
4
5  Example output:
6  > line1
7  ! line2
8  > line3
9  ...
10 > linen
11
12 Note: Executed Lines must be denoted with a SINGLE '>' and non-executed Lines must be
          denoted with a SINGLE '!'
13
14 DISCLAIMER: DO NOT PROVIDE AN EXPLANATION. JUST PROVIDE THE FINAL PREDICTED CODE
          COVERAGE FOR THE GIVEN CODE SNIPPET
15 Only give the final predicted code coverage of the following code.
16
17 def check_even_odd(number):
18   if number % 2 == 0:
19     return "even"
20   else:
21     return "odd"
22 def display_results(number, result):
23   print("Number: {}".format(number))
24   print("Result: {}".format(result))
25   print("--------------------")
26 def main():
27   num_iterations = 3
28   for i in range(num_iterations):
29     user_number = 4
30     result = check_even_odd(user_number)
31     if result == "even":
32       print("It's an even number!")
33     else:
34       print("It's an odd number!")
35     display_results(user_number, result)
36 if __name__ == "__main__":
37     main()
```

(a) Zero-shot prompt for GPT 3.5 to predict the code coverage for a given test code from lines 17–37

```
1  « Instructions, as on Lines 1-12 in Figure 1(a) »
2  Below is an illustration of the process to predict the code coverage of the given
          code snippet.
3
4  Exemplar code snippet:
5  from math import factorial
6  def comb(n, r):
7      return factorial(n)
8  def main():
9      N = 2
10     P = 0
11     A = [1, 3]
12     even, odd = 0, 0
13     for i in range(N):
14         if A[i] % 2 == 0:
15             even += 1
16         else:
17             odd += 1
18     a, b = 0, 0
19     b = 2 ** even
20     for i in range(odd + 1):
21         if i % 2 == P:
22             a += comb(odd, i)
23     print(a * b)
24 if __name__ == "__main__":
25     main()
26
27 So the code coverage for the example code snippet will be:
28 > from math import factorial
29 > def comb(n, r):
30     ...
31 >           even += 1
32 !       else:
33 !           odd += 1
34 >    a, b = 0, 0
35     ...
36 >    main()
37
38 In a similar fashion, predict the code coverage for the below code snippet:
39 « Code, as on lines 17-37 in Figure 1(a) »
```

(b) A one-shot prompt containing an exemplar (different from the test code) that contains the code coverage

Figure 1: Zero-shot prompt and one-shot prompt *without Planning* to predict the code coverage for a test code

```
PLAN for the exemplar code in Figure 1(b)
Step 1: Import Statement : factorial is a packahe being imported from math library.
        Import statements are always executed. Statement "from math import factorial"
        will ne executed.
Step 2: Check __name__ : Check if the script is being run as a main program. Since it
        is, "if __name__ == "__main__": " will be executed.
Step 3: Main method call: Since the script is being run as the main program, call the
        main method. "main()" will be executed
Step 4: Main Method Execution: Contents inside mthod will be executed. "def main(): "
        will be executed. Since "N = 2", "P = 0", "A = [1, 3]" and "even, odd = 0, 0"
        are all variable initialization statements, they will be executed.
Step 5: First For loop in Main Function: Enter the for loop and Iteratively assign
        values to i from 1 to range(N). "for i in range(N):" will be executed.
Step 6: If - Else Branching in For Loop: For each iteration of i, the element at A[i]
        is checked if its even or odd. Both elements in list A are odd, so the if
        statement is not satisfied and the else statement is satisfied so contents of
        else block is executed. Statements "if A[i] % 2 != 0:" and "odd += 1" are
        executed. Statements "else: " and "even += 1" is not executed.
Step 7: Variable initialization and Math operations : Statements "a, b = 0, 0" and "b
        = 2**even" are variable declarations so they both will be executed.
Step 8: Second For Loop in Main Function : Enter the for loop and Iteratively assign
        values to i from 1 to range(odd + 1). "for i in range(odd + 1):" is executed.
Step 9: If-else block withing for loop in main method: The if statement checks
        whether the remainder of variable i when divided by 2 is equal to value of P.
        Since the value of i in the second iteration is 2, the remainder is 0 which
        is equal to P, so the if condition is satisifed. Statement "if i % 2 == P:"
        is executed
Step 10: Method call in if block of for loop: When the if statement is satisfied, the
        variable a is incremented with the value return by method comb. Since it is a
        function call, statement "a += comb(odd, i)" will be executed.
Step 11: Called Method execution: Inside comb(), The math operation of calculating
        the factorial occurs. Since its a simple math operation and return statement,
        statement "return factorial(n)" will be executed. The statement containing
        the method declaration "def comb(n, r): " will also be executed.
Step 12: Print Statement: Print statements are always executed. Statement "print(a *
        b)" will be executed.
```

(a) Manual exemplary plan for the exemplary code in Figure 1(b)

```
PLAN generated by GPT-3.5 for the running example in Figure 1(a)
Step 1: Check __name__ : Check if the script is being run as a main program. Since it
        is, "if __name__ == "__main__": " will be executed.
Step 2: Main method call: Since the script is being run as the main program, call the
        main method. "main()" will be executed
Step 3: Main Method Execution: Contents inside mthod will be executed. "def main(): "
        will be executed.
Step 4: For loop in Main Function: The for loop will iterate from 0 to
        num_iterations. Since num_iterations is 3, the for loop will execute 3 times.
        Statement "for i in range(num_iterations):" will be executed 3 times.
Step 5: Variable Initialization in For Loop of main method: The variable user_number
        is initialized to 4 in each iteration of the for loop. Statement "user_number
        = 4" will be executed 3 times.
Step 6: Method call in main method: The method check_even_odd is called with the
        value of user_number as the argument. Since it is a method call, "result =
        check_even_odd(user_number)" will be executed 3 times.
Step 7: Method Execution: Inside check_even_odd(), the variable number is checked if
        it is even or odd. The number 4 is even so the if condition is satisfied.
        Statement "if number \% 2 == 0:" will be executed 3 times.
Step 8: If-Else branching in check_even_odd() method: Since the number 4 is even, the
        if condition is satisfied and the value "even" will be returned. So statement
        "return "even"" will be executed 3 times. Statement "else" and "return "odd""
        will not be executed.
Step 9: If-Else branching in main() method: Since the result is "even", the if
        condition is satisfied and the print statement is executed. Statement "if
        result == "even":" will be executed 3 times. Statement "else" will not be
        executed.
Step 10: print() statement: The print statement will always be executed. So
        "'print("It's an even number!")' will be executed 3 times.
Step 11: Method call in main method: The method display_results is called with the
        values of user_number and result as the arguments. Since it is a method call,
        "display_results(user_number, result)" will be executed 3 times.
Step 12: Method Execution: Inside display_results(), the values of number and result
        are formatted and printed. So both print statements will be executed 3 times.
        Statement "print("--------------------")" will be executed 3 times.
```

(b) GPT-generated plan for test code snippet in Figure 1(a)

Figure 2: One-shot with *Planning* for the test code snippet

We requested GPT-3.5 to generate its own plan for the given code in Figure 1(a). Subsequently, in another prompt, we requested GPT-3.5 to follow its plan to perform the computation of code coverage.

The result of the first prompt, which is the plan produced by GPT-3.5 for our running example code in Figure 1(b), is shown in Figure 2(b). Upon closer examination, the plan produced by GPT-3.5

```
Zero  One P Oracle  TEST CODE
 >     >  >   >     def check_even_odd(number):
 >     >  >   >         if number % 2 == 0:
 >     >  >   >             return "even"
 >     !  !   !         else:
 >     >  !   !             return "odd"
 >     >  >   >     def display_results(number, result):
 >     >  >   >         print("Number: {}".format(number))
 >     >  >   >         print("Result: {}".format(result))
 >     >  >   >         print("-------------------")
 >     >  >   >     def main():
 >     >  >   >         num_iterations = 3
 >     >  >   >         for i in range(num_iterations):
 >     >  >   >             user_number = 4
 >     >  >   >             result = check_even_odd(user_number)
 >     >  >   >                 if result == "even":
 >     >  >   >                     print("It's an even number!")
 >     !  !   !                 else:
 >     >  !   !                     print("It's an odd number!")
 >     >  >   >                 display_results(user_number, result)
 >     >  >   >     if __name__ == "__main__":
 >     >  >   >         main()
```

Figure 3: Code coverage results for different strategies: zeroshot (`Zero`), one-shot (`One`), one-shot *with Planning* (`P`)

encompasses the precise steps that account for the rationale behind the execution of `if-else` branching blocks (refer to Steps 8 and 9 in Figure 2(b)), the `for` loop (Step 4 in Figure 2(b)), and the method calls (illustrated in Steps 6 and 11 in Figure 2(b)).

The code coverage result of the second prompt in our planning is displayed in the column marked by `P` (planning) in Figure 3. Comparing with the results without planning, we can see that with planning, GTP-3.5 achieved better performance, with the 100% correct coverage prediction for all statements. Additionally, through its plan (Figure 2(b)), the LLM adeptly articulates the steps that would have been followed had the code been executed.

2.2 Key Ideas

Drawing upon the above observations, we present CodePilot, a planning approach that involves prompting GPT to assist in devising a plan and subsequently executing it to compute code coverage.

CodePilot is formulated based on the following key ideas.

2.2.1 Key Idea 1 [Leveraging Planning Ability of Large Language Models for Code Coverage task]. The utilization of planning techniques in Large Language Models (LLMs) has achieved significant success in the domains of robotics, NLP, and machine learning [13, 22, 25, 32, 34]. When dealing with the intricacies of predicting code execution for code coverage, planning becomes a valuable vehicle for *enabling an LLM to harness its capacity to autonomously generate its own plans based on guided examples and plans.* This approach leverages the LLM's inherent capability to analyze sequences of actions and decisions, empowering it to formulate comprehensive plans that capture the rationale and navigate through the action sequences essential for execution-aware tasks.

2.2.2 Key Idea 2 [Enhancing Code Execution-Flow Understanding with Program-Semantics-based Planning]. Statically predicting the code execution and calculating code coverage poses a challenge in the realm of program analysis when actual execution is infeasible. Unlike planning in NLP, which depends on the LLM's proficiency in text comprehension, we propose a novel approach—planning for code coverage rooted in program analysis, which encompasses the nuances in code execution. In this context, we present a guided exemplary plan that discerns various types of statements, treating them differently to enhance code execution understanding.

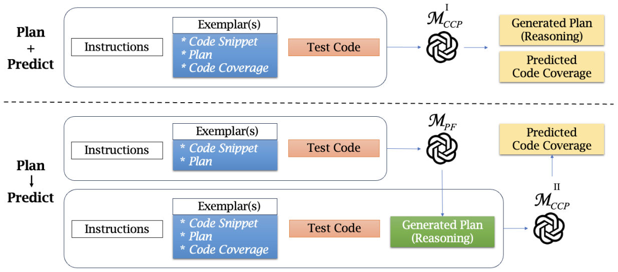

Figure 4: Overview of CodePilot Workflow for *One-Prompt* (top) and *Two-Prompt* (bottom) settings.

2.2.3 Key Idea 3 [PA-based Planning through Reasoning and Actions]. CodePilot combines the synthesis of Reasoning and Acting to predict the code coverage of a given code. The reasoning component of the approach encompasses the program flow steps that would have been taken had the code been executed. Each step in the reasoning provides a brief explanation of why a statement would or wouldn't have been executed. This reasoning serves as a guide for the LLM, first directing its attention to the detailed execution steps in its own plan. Subsequently, the LLM is tasked with code coverage prediction (Action) based on the generated plan (Reasoning).

3 OVERVIEW OF CODEPILOT WORKFLOW

This paper presents CodePilot with two approaches to prompting: a unified *Plan+Predict* design; and a two-phase *Plan⟶Predict* design. Figure 4 illustrates the workflow of both prompting approaches. For a given code snippet C_T comprising the test input, CodePilot facilitates the systematic prediction of code coverage by formulating a PA-based plan to navigate the code and attain an understanding of its execution flow. In the rest of the paper, we refer to them as *one-prompt* and *two-prompt* approaches, respectively.

3.1 One-Prompt CodePilot

In this approach, for a given code snippet C_T comprising the test inputs, CodePilot facilitates the systematic prediction of code coverage by: [Step I. *Plan Formulation*] constructing a plan rooted in program semantics to navigate the code and attain an understanding of the execution flow; [Step II. *Code Coverage Prediction*] determining the code coverage based on such a plan.

For this purpose, we leverage an LLM $\mathcal{M}^{I}_{CCP}$ that takes as an input prompt: (a) a set of instructions $\mathcal{I}_{CCP}$ describing the task; (b) an exemplar comprising code snippet C (different from C_T), a manually-crafted, examplary plan $\mathcal{P}$, its code coverage Cov; and (c) the test code snippet C_T. Here, $\mathcal{M}^{I}_{CCP}$ utilizes the exemplar to guide the LLM to first reason about C_T and construct a program semantics-guided code execution plan P_T, following which, it predicts the code coverage Cov_T. This can be formulated as:

$$\langle \mathcal{P}_T, Cov_T \rangle = \mathcal{M}^{I}_{CCP} \{ \mathcal{I}_{CCP}, \langle C, \mathcal{P}, Cov \rangle, C_T \} \quad (1)$$

3.2 Two-Prompt CodePilot

Unlike in one-prompt CodePilot, in this approach, we divide both *Plan Formulation* and *Code Coverage Prediction* in two prompts. The goal of the initial *Plan Formulation* phase is to guide the LLM to _reason_ about the given code snippet and construct a plan by itself, that is integral for navigating the code snippet and attaining an understanding of the execution flow. For this purpose, we leverage an LLM $\mathcal{M}_{PF}$ that takes as an input prompt: (a) a set of instructions $\mathcal{I}_{PF}$ describing the task; (b) an exemplar comprising a code snippet C (different from C_T) and its corresponding plan $\mathcal{P}$; (c) the given code snippet C_T. Here, $\mathcal{M}_{PF}$ utilizes the exemplar to guide the LLM to generate a similar plan $\mathcal{P}_T$ for C_T. This can be formulated as:

$$\mathcal{P}_T = \mathcal{M}_{PF}\ \{\ \mathcal{I}_{PF}, \langle C, \mathcal{P} \rangle, C_T\ \} \tag{2}$$

In the next *Code Coverage Prediction* phase, the goal is to _act_ on the code snippet C_T as per the LLM-generated plan $\mathcal{P}_T$ to enhance its execution-flow understanding, and predict the code coverage accordingly. For this purpose, we leverage an LLM $\mathcal{M}^{II}_{CCP}$ that takes as an input prompt: (a) a set of instructions $\mathcal{I}_{CCP}$ describing the task; (b) an exemplar comprising code snippet C and its plan $\mathcal{P}$ (same as in the *Plan Formulation* phase), as well as its code coverage *Cov*; (c) the test code snippet C_T; and (d) the LLM-generated plan $\mathcal{P}_T$. Here, $\mathcal{M}^{II}_{CCP}$ utilizes the exemplar to guide the LLM to learn to determine the code coverage for the code example based on the program semantics-guided code execution plan. Subsequently, based on the $\mathcal{M}_{PF}$-generated plan $\mathcal{P}_T$ for the test code snippet, it predicts the code coverage Cov_T. This can be formulated as:

$$Cov_T = \mathcal{M}^{II}_{CCP}\ \{\ \mathcal{I}_{CCP}, \langle C, \mathcal{P}, Cov \rangle, C_T, \mathcal{P}_T\ \} \tag{3}$$

We will elaborate on both prompting approaches in Sections 4 and 5, respectively. We experiment with both One-Prompt and Two-Prompt solutions and compare their results (Section 7).

4 ONE-PROMPT CODEPILOT

In this section, we present our design of the unified *Plan+Predict*.

4.1 Basic Structure

In the one-prompt setting (i.e., *Plan+Predict*), we design a single, consolidated prompt comprising three primary segments: (1) the instructions to the LLM to compute the code coverage for the given code snippet, (2) the exemplar(s) comprising code snippet, corresponding manually-crafted examplary plan, corresponding code coverage, and (3) the test code snippet. The output from the LLM includes: (1) the generated plan, and (2) the predicted code coverage for the test code snippet.

4.1.1 Instructions. This segment contains specifications for the LLM to first generate a plan for understanding the execution flow in the given test code, using the manually-crafted plan for the code snippet in the exemplar(s) as a guide. Then, it instructs the LLM to follow the plan to compute the code coverage for the test code snippet, drawing parallels from the manually-crafted plan and the code coverage for the code snippet in the exemplar(s). Furthermore, the included code coverage guides the LLM to format the output code coverage prediction in the prescribed format.

4.1.2 Exemplar(s). To guide the LLM as described in Section 4.1.1, we include example(s) for the few-shot setting comprising the code snippet, manually-crafted plan, and corresponding code coverage.

4.1.3 Given Code Snippet. This is the code for which LLM has to create a code execution plan and subsequently predict its coverage.

4.2 Manually-Crafted Plan in Exemplar(s)

The essence of designing the plan resides in the formulation of a step-by-step reasoning for understanding execution-flow in an exemplar code snippet. The structure of each step within the plan is composed of three fundamental elements: the step number, the type of statement(s), and the rationale behind their potential execution.

Firstly, the step numbers denote the sequence in which the code would have been executed. In Figure 2(a), Step 4 (pertaining to the exemplar code in Figure 1(b)) labeled as 'Main Method Execution' consistently follows Step 3, designated as 'Main Method Call'.

Secondly, the label assigned to each step serves as the primary distinguishing factor for that set of statement(s) in the code. For example, the statements 18-19 in Figure 1(b) are categorized as 'Variable Initialization' and are consequently grouped together in Step 7 of the exemplary plan in Figure 2(a). It is noteworthy that certain labels, such as 'Variable Initialization', 'Method Call', 'If-else Statement', among others, may directly indicate whether the associated statements will be executed. Details are given in Section 4.3.

Lastly, the final element of each step in the plan is the justification for the possible execution or non-execution of the specific set of statements. For instance, let us consider Step 9 in Figure 2(a), which analyzes the `if-else` statement in the `for` loop within the `main()` function (lines 20-22 of Figure 1(b)). Step 5 briefly explains that due to all the elements in list `A[]` being odd, only the condition in the `if` statement holds true, which leads to the statement(s) within the `if` branch to be executed. It also clearly mentions that since the condition of the `if` branch holds true through all iterations of the `for` loop, the `else` branch and its associated statements would have never been executed. This concise yet accurate reasoning is essential to guide the LLM to follow the step to design its own plan.

4.3 Reasoning on Program Semantics

To compute code coverage, the LLM needs to reason correctly on the execution steps of the statements in the code. In addition to the statements whose executions follow a sequential order, there are three types of statements that could *alter such sequential execution*: *branching statements, loop statements, and method calls.*

In programming, the execution of branching statements (`if` or `switch` statements) are pivotal for controlling the flow of a program based on certain conditions. When encountering an `if` statement, the program evaluates a specified condition and, if true, executes the corresponding block of code; if false, it either moves to the next `elif` condition or proceeds to the `else` block if provided. In contrast, a `switch` statement is designed to evaluate an expression against multiple possible constant values. It provides a concise way to handle multiple cases, each with its own set of code. The exemplary plan needs to explain the nuances in the branching statements. For example, the plan in Figure 2(a) considers the `if-else` construct in the code snippet (Figure 1(b)) by briefly outlining the condition and the statement(s) executed based on the condition mentioned in the condition. This is expressed in Steps 6 and 9 of Figure 2(a).

The execution of a `for` statement and a `while` statement are both iterative processes. The `for` statement is typically employed when the number of iterations is known beforehand. It consists of three parts within its parentheses: initialization, condition, and increment/decrement. The loop executes as long as the specified condition remains

```
 1  ONE-PROMPT SETTING
 2  For the given code snippet, Predict the code coverage. The code coverage indicates
            whether a statement has been executed or not.
 3  > if the line is executed
 4  ! if the line is not executed
 5
 6  Example output:
 7  > line1
 8  ! line2
 9  > line3
10  ...
11  > linen
12
13  You need to develop a plan for step by step execution of the code snippet.
14  Below is an illustration of the process you need to follow to predict the code
            coverage of the given code snippet.
15
16  Example - Given Code Snippet:
17  number = 15
18  if number % 2 == 0:
19     print("{} is an even number.".format(number))
20  else:
21     print("{} is an odd number.".format(number))
22
23  PLAN :
24  Step 1: Variable Initialization: Initialize the number variable with a specific
            value. In this case, it's set to 15. Statements "number = 15" will be
            executed. "if number % 2 == 0:" will be executed
25  Step 2: Operation : Use the % (modulo) operator to check if the number is divisible
            by 2. If the result is 0, the number is even. If not, it's odd.
26  Step 3: Branching if-else Block: Enter the if block if the number is even. Since 15%2
            results in 1, statement "print("{} is an even number.".format(number))" will
            not be executed but the else block will if the number is odd. Statements
            "else:" and "print("{} is an odd number.".format(number))" will be executed
27  Step 4: Output: The required print statement will be executed based on the output of
            the if-else block
28
29  So the code coverage for the given code snippet will be:
30  > from math import factorial
31  > number = 15
32  > if number % 2 == 0:
33  !    print("{} is an even number.".format(number))
34  > else:
35  >    print("{} is an odd number.".format(number))
36
37  In a similar fashion, develop a plan of step by step execution of the below code
            snippet and predict the code coverage -
38  « Test Code... »
```

Figure 5: Example on one-prompt setting with CodePilot

true. In contrast, a `while` statement is more versatile and is used when the number of iterations is uncertain or depends on a certain condition. The `while` loop continues to execute as long as the specified condition holds true, and the programmer is responsible for updating the loop variable within the loop block. While both constructs facilitate iteration, the `for` statement is more structured and concise, while the `while` statement offers greater flexibility in handling variable loop conditions. Such nuances of a loop statement need to be incorporated into the exemplar plan. For example, let us consider Step 5 of Figure 2(a). The plan for the exemplary code snippet in Figure 1(b) explains the number of iterations for each of the statements in the `for` loop. Since the `for` loop is not conditional, line 21 in Figure 1(b) will be executed. Lastly, the plan also includes the reason behind the execution of a statement containing a method call. Upon calling a method, the plan progresses to the subsequent step, involving the execution of the called method.

In addition to the statement-specific guiding, the exemplary plan accommodates additional statements found in a code snippet, such as variable initialization and print statements. This can be seen in Steps 1, 4 and 12 of Figure 2(a), which have been created primarily for accommodating the package import statements, variable initialization within a method and print statements, respectively.

4.4 Illustrating Example

The example presented in Figure 5 outlines the process of predicting code coverage using one-prompt CodePilot for a given code snippet. The objective is to first predict a plan for the test code, and then determine the executed or non-executed status of each line, denoted by '>' for executed lines and '!' for non-executed lines. The detailed instruction is illustrated in lines 2–11.

The example code snippet (lines 18-22 of Figure 5) initializes a variable `number` to `15`, followed by a conditional statement checking if the number is even or odd. The provided plan (lines 25-28) details the execution steps, such as variable initialization, modulo operation, branching in the `if-else` block, and the corresponding output. Following the outlined steps, the final predicted code coverage is presented (lines 31–36), highlighting the lines that are expected to be executed or skipped based on the given plan. The subsequent request prompts a similar analysis for a different test code (line 40 onwards), encouraging a comprehensive exploration of code coverage prediction within the generated plan.

5 TWO-PROMPT CODEPILOT

In this section, we present our design of the two-phase *Plan⟶Predict*.

5.1 Plan Formulation

5.1.1 *Basic Structure.*

This section presents the first phase, i.e., *Plan Formulation*, in *Plan⟶Predict*. In this phase, we design a prompt comprising three primary segments: (1) instructions to the LLM to devise a PA-based plan, (2) exemplar(s) comprising code snippet and corresponding plans, (3) test code snippet. The output from the LLM comprises of the generated plan for the test code snippet.

Instructions. The first segment consists of the problem statement explained in natural language, which instructs the LLM to predict/build its own plan to be pursued for predicting the code coverage of the given test code (see example in Section 5.1.2).

Exemplar(s). The second segment of the prompt incorporates exemplar(s) to enable the few-shot setting, each comprising a code snippet (distinct from the given test code) and a manually-crafted plan. Each step within that plan provides a succinct elucidation of the reasons behind the (non)-execution of the associated statement(s). The procedure for crafting the plan is same as in Section 4.2. The LLM uses this as a guide to understand the execution flow.

Given Test Code Snippet. The last segment is the test code for which LLM has to create the code execution plan.

5.1.2 *Illustrating Example.*

An example prompt for the *Plan Formulation* phase is provided with a single exemplar in Figure 6(a). Note that the code snippet and the corresponding manually-crafted plan in the exemplar are the same as in Section 4.4.

Similar to as in one-prompt CodePilot, the prompt in this phase guides the LLM to draw parallels from the execution rationale in the exemplar plan by learning to reason about the exemplar code, and subsequently, generate a similar plan for the test code. Such a break down of the test code into comprehensible steps enables the LLM to reason about the code execution and make predictions regarding which statements will be executed or skipped. Thus, such a systematic approach encapsulated within the *Plan Formulation* phase instills a structured method for later predicting the code coverage. By design, this approach provides a guide to understand the code coverage prediction, improving the LLM's interpretability.

```
PLAN FORMATION
For the given code snippet, give the plan to predict the code coverage. The code
        coverage indicates whether a statement has been executed or not.

You need to develop a plan for step by step execution of the code snippet.
Below is an illustration of the process you need to follow to predict the code
        coverage of the given code snippet.

« Exemplar code, as on Lines 16-22 in Figure 5 »

« Exemplar plan, as on Lines 24-28 in Figure 5 »

In a similar fashion, develop a plan of step by step execution of the below code
        snippet -
« Test Code... »
```

(a) One-shot prompt used for *Plan Formation*

```
COVERAGE PREDICTION
For the given code snippet and plan, give the code coverage that follows the plan.
        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 give the code with its coverage for the given plan.
Below is an illustration of the process you need to follow to predict the code
        coverage of the given code snippet and its plan.
« Exemplar code, as on Lines 16-22 in Figure 5 »

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 '>'

« Exemplar plan, as on Lines 24-28 in Figure 5 »

« Exemplar code coverage, as on Lines 30-36 in Figure 5 »

In a similar fashion, give the code coverage of the below code snippet based on the
        given plan -

« Test code... »
« Test plan (generated by the Plan Formulation phase)... »
```

(b) One-shot prompt used for *Code Coverage Prediction*

Figure 6: Prompts with a single exemplar (i.e., one-shot) for *Plan Formulation* (top) and *Code Coverage Prediction* (bottom) phases in two-prompt CodePilot.

5.2 Code Coverage Prediction

5.2.1 Basic Structure. This section presents the second phase, i.e., *Code Coverage Prediction*, in *Plan⟶Predict*. In this phase, we design a prompt comprising four primary segments: (1) instructions to the LLM to compute the code coverage for the code snippet, (2) exemplar(s) comprising code snippet, corresponding manually-crafted exemplary plan and code coverage, (3) test code snippet, (4) generated plan from *Plan Formulation* phase. The output from the LLM comprises the predicted code coverage for the given test code.

Instructions. This segment in the second prompt requests the LLM to adhere to the *generated plan* to predict the code coverage for the given test code, using the exemplar(s) to guide this process.

Exemplar(s). In the two-prompt setting, the exemplar(s) are exactly the same as in one-prompt CodePilot (see Section 4.1.2). The inclusion of this serves the purpose of guiding the LLM towards achieving a code coverage prediction in the prescribed format.

Given Test Code Snippet. This segment is same in both *Plan Formulation* (Section 5.1.1) and *Code Coverage Prediction* phases.

Generated Plan. In the two-prompt setting, the plan generated from the *Plan Formulation* phase is incorporated into the prompt in this phase to guide the code coverage prediction. Note that this differs from the one-prompt CodePilot where one LLM is tasked with generating both the plan and the code coverage sequentially.

Table 1: Comparison among CodePilot's Settings (RQ1)

Model Setting	Evaluation Metrics (%)	
	Exact-match	*Statement-match*
One-prompt CodePilot with one-shot	51.00	87.47
Two-prompt CodePilot with one-shot	44.37	87.00
One-prompt CodePilot with few shots	**55.20**	**89.20**
Two-prompt CodePilot with few-shots	47.27	88.53

5.2.2 Illustrating Example. An example prompt for the *Code Coverage Prediction* phase is provided with a single exemplar in Figure 6b. Note that the exemplar's structure is same as that in one-prompt CodePilot, including the same code snippet and manually-crafted plan in the exemplar as in the *Plan Formulation* phase (Section 5), as well as the corresponding code coverage. Next, along with the test code, we include the test plan generated in the *Plan Formulation* phase. By design, this prompt facilitates the prediction of code coverage for the test code from the generated plan, inherently learning such associations between alike components in the exemplar(s).

6 EMPIRICAL EVALUATION

For evaluation, we seek to answer the following research questions:

RQ1. [Comparison on Code Coverage Prediction]. How well does CodePilot perform compared with the existing ML models?

RQ2. [Performance of Planning]. How well does GPT perform with CodePilot in generating its own plans for code coverage?

RQ3. [Performance on Different Statement Types]. How well does CodePilot perform on different types of statements?

RQ4. [Performance on Least-covered Statement Prediction]. How accurate is CodePilot in predicting least-covered statements?

RQ5. [Exploration Space in Execution]. Does CodePilot help in reducing the exploration space in execution paths?

Dataset. We used CodeNetMut dataset, provided in CodeExecutor [16], containing the mutated versions of a collection of the submissions to competitive programming problems. In total, it has 19,541 data examples, each of which has code and execution traces.

7 CODE COVERAGE PREDICTION (RQ1)

7.1 Comparison among CodePilot's settings

7.1.1 Procedure. As explained in Section 3, we have two variants: one-prompt and two-prompt CodePilot. One-shot means one examplary code and plan, while few-shot refers to multiple ones. All four settings of CodePilot in Table 1 consist of two parts: planning and code coverage prediction (organized in one or two prompts).

7.1.2 Metrics. To assess the performance of CodePilot, we used two metrics: *exact-match accuracy* and *statement-match accuracy*. The *exact-match accuracy* quantifies the number of programs for which the predicted sequence of statement/branch coverages precisely matches the target coverage sequence, indicating perfect accuracy for all statements/branches within the program. In contrast, the *statement-match accuracy* measures the percentage of correctly predicted covered/not-covered statements. While *statement-match accuracy* is aimed to evaluate accuracy at the individual statement level, *exact-match accuracy* provides an assessment at the entire code level. Two metrics complement to each other on evaluating the quality of entire coverage and individual statements.

7.1.3 Results. As seen in Table 1, CodePilot achieves highest accuracies with the few-shot and one-prompt setting. In *more than half* of the code snippets, CodePilot predicts correctly the entire code coverages. Considering the statements individually, it correctly predicted the coverages of *9 out of 10 statements*.

When prompted with a single prompt, the GPT-3.5 model exhibits higher exact-match accuracy compared to the scenario where it is prompted sequentially with two prompts. The observed relative increase of 14.94% in exact-match accuracy, coupled with a relative increase of 0.54% in statement-match accuracy when compared to the two-prompt strategy in a one-shot setting, suggests the possibility that predicting step-by-step reasoning and subsequent code coverage within the same prompt may facilitate the retention of information within the model between the two predictions (plan and code coverage). This also applies for the few-shot setting where the one-prompt strategy has a relatively higher exact-match accuracy of 16.77% and a relatively higher statement-match accuracy of 0.75%. In brief, with the one-prompt setting, CodePilot could save the number of tokens sent to GPT as well as achieve higher code coverage prediction accuracy.

In the comparison between the one-shot and few-shot single-prompt strategies, the few-shot prompting shows a relatively higher exact-match accuracy of 8.2% and a statement-match accuracy of +2%. This observed increase in both accuracy metrics is potentially attributable to the incorporation of additional exemplars in the prompt, thereby enhancing the model's understanding.

7.2 Comparison with state-of-the-art Models

7.2.1 Baselines. We compared CodePilot to Tufano *et al.* [30], that directly used GPT model in both zero-shot and one-shot settings. We also compared CodePilot to CodeExecutor [16], a Unixcoder-based neural network model to predict execution traces. From the traces, we compute the code coverages for the code snippets.

7.2.2 Procedure and Metrics. Following CodeExecutor's [16] and Tufano *et al.*'s [30], we executed them on our dataset. For CodePilot, we followed the workflow in Figure 4 for one-shot setting. For few-shot setting, the workflow is the same but with additional examplary code and plans. We used the same metrics as in RQ1.

7.2.3 Results. As the one-prompt setting give higher accuracy, we use it in comparison with the state-of-the-art models. As shown in Table 2, CodePilot emerges as a highly effective approach for code coverage prediction, outperforming both [16] and Tufano *et al. et al.* [30] in all settings. As seen in Table 1, CodePilot exhibits notable performance advantages in a *one-shot setting*, achieving an *23.21% relatively higher exact-match accuracy* and an *16.16% relatively higher statement-match accuracy* compared to CodeExecutor. This increase in performance can be attributed to the distinction in their core objectives. CodeExecutor places an emphasis on predicting the precise order of statement execution as well as the values of the variables at each execution step. Minor inaccurate value or execution trace order prediction could result in more widespread incorrect code coverage prediction. CodePilot focuses on predicting code coverage regardless of the statement execution order.

Table 2: Code Coverage Prediction Accuracy (RQ1)

Model	Evaluation Metrics (%)	
	Exact-match	*Statement-match*
CodeExecutor [16]	41.39	75.30
Tufano *et al.* zero-shot [30]	41.31	84.09
Tufano *et al.* one-shot [30]	42.99	84.60
CodePilot w. GPT-3.5 one-shot	51	87.47
CodePilot w. GPT-3.5 few-shots	**55.20**	**89.20**

Regarding the LLM-based solutions, CodePilot's *one-shot setting* demonstrates an approximate *23.45%* relative increase in exact-match accuracy and a *4.02%* relative higher statement-match accuracy compared to Tufano *et al.* [30] with zero-shot. Compared with Tufano *et al.* in a one-shot setting, CodePilot outperforms with *18.63%* and *3.39%* relatively higher accuracies in exact-match and statement-match, respectively. The positive differences in accuracies are attributed to the methodological variation in prompting, where CodePilot, via planning, guides the model to create a detailed step-by-step execution plan before predicting code coverage, resulting in reduced model hallucination and helping it navigate better in the large exploration space of execution paths.

In the *few-shot* setting using CodePilot, a relative increase of *33.36%* in exact-match accuracy and *18.45%* in statement-match accuracy is observed compared to CodeExecutor. Additionally, compared to Tufano *et al.* in a zero-shot setting, CodePilot's few-shot (with planning) achieves a relative increase of *33.62%* and *6.08%* in exact-match and statement-match, respectively. Furthermore, CodePilot's few-shot (with planning) surpasses Tufano *et al.*'s one-shot setting (without planning) by *28.40%* relatively in exact-match accuracy and *5.43%* relatively in statement-match accuracy. With planning, *the LLM computes correctly the coverage for more individual statements (5.4–18.5% relatively higher), as well as the coverage for more code snippets as a whole (*28.6–33.6%*)*. These improvements can be attributed to the combination of incorporating more exemplars and guiding the model in creating a stepwise plan.

8 ACCURACY OF PREDICTED PLANS (RQ2)

Table 3: Accuracy of Predicted Plans (RQ2)

Model	Evaluation Metrics (%)	
	Plan accuracy	*Step accurarcy*
One-prompt CodePilot with few-shots	75	90

In this experiment, our objective is to evaluate the accuracy with which the LLM formulates the plans via the help of CodePilot.

Procedure. We chose the few-shot, one-prompt setting for CodePilot due its highest accuracy in RQ1. A set of 100 random data examples was drawn from the CodeNetMut dataset, and the GPT-generated plans were subject to manual evaluation for accuracy.

Metrics. Two metrics were used to assess the plans generated by GPT for predicting code coverage for a code snippet - *plan accuracy* and *step accuracy.* Plan accuracy (micro accuracy) is defined as the accuracy of the entire plan, wherein each code snippet achieves 100% plan accuracy if the entirety of the plan is generated accurately, including the precise order of the steps. Step accuracy (macro accuracy) is defined as the ratio of accurately predicted steps within a particular plan to the total number of steps generated in that plan.

Results. As described in Table 3, the plan accuracy for CodePilot’s few-shot double prompting strategy stands at 75%, indicating that for every 100 code snippets, GPT predicts 75 plans with 100% accuracy. Furthermore, the step accuracy (refer to Table 3) is 90%, which means that if a single plan contains 10 steps, GPT is able to describe and reason 9 of them with 100% accuracy.

9 PREDICTION OF STATEMENT TYPES (RQ3)

Table 4: Accuracy on If-Elses, Loops, and Method Calls

Model	**Evaluation Metrics (%)**		
	Branch-match	*Loop-match*	*Method-match*
Tufano *et al.* zero-shot [30]	57.91	33.85	79.05
Tufano *et al.* one-shot [30]	58.44	33.96	79.68
CodePilot GPT 3.5 one-shot	**63.51**	**35.21**	85.98
CodePilot GPT 3.5 few-shots	**65.17**	**35.50**	87.49

In our design of an examplary plan to guide GPT, we focus on three types of statements (`if-else`, `loop`, and `method call`) as in Section 4.3. In this experiment, we aim to evaluate how well CodePilot performs for those statements involving in those types.

Metrics. We utilize three following metrics: *branch-match accuracy*, *loop-match accuracy* and *method-match accuracy.* Branch-match accuracy is calculated as the ratio of accurately predicted `if-else` branches’ coverages (i.e., the model predicts the correct branch at a condition) to the total number of branches in the code. Loop-match accuracy is defined as the ratio of accurately predicted loop coverages to the total number of distinct loops in the code. A loop is considered to be correctly predicted for code coverage if all the statements within it are correctly predicted as covered/non-covered. Method-match accuracy is the ratio of accurately predicted method call coverages to the total number of method calls in the code. An inter-procedural flow for a method call is considered to be correctly predicted if the first statement in the called method is covered. (Note that we excluded the Python built-in methods, e.g., ‘`print()`’.)

Results. CodePilot with GPT 3.5 one-shot setting exhibits enhancements in all metrics when compared with Tufano *et al.* [30] with zero-shot, showing a relative increase of 9.67%, 4.01% and 8.76% in branch-match and loop-match and method-match accuracies, respectively (Table 4). Moreover, CodePilot with GPT 3.5 few-shot setting manifested improvements in all metrics compared to Tufano *et al.* [30] with zero-shot, with a relative increase of 12.53%,, 4.87% and 10.67% in branch-match, loop-match and method match accuracies, respectively. CodePilot with GPT-3.5 one-shot and few-shot settings also improve over Tufano *et al.* with one-shot.

As seen, with planning, all three metrics are improved. *Regarding branches*, with 65.17% of branch-match, almost 2 out 3 branching decisions are predicted correctly with CodePilot. Without planning, 57% of them are correct. *Regarding the loop understanding*, it is still challenging for all the approaches with only about 5% relative improvement in loop-match accuracy. However, with CodePilot, in 35.5% of the loops, all of their statements are predicted correctly on their coverages. Regarding *inter-procedural method calls*, CodePilot achieves 10.7% method-match accuracy relatively higher than the baselines. That is, *with planning, the GPT-3.5 model has better understanding of the inter-procedural flows with method calls.*

```
 1  Ranking of statements based on execution
 2  GT  PR  CODE
 3  1   1   s = *string*
 4  1   3   k = *integer*
 5  1   2   l = *string*
 6  1   2   ans = "" *string*
 7  1   3   for i in range(len(s) - 1):
 8  1   4       num = (26 - l.index(s[i])) % 26
 9  2   7       if k >= num:
10  2   8           ans += "a"
11  2   9           k -= num
12  2   10      else:
13  3   11          ans += s[i]
14  1   6   ans += l[(l.index(s[-1]) + k) % 26]
15  1   5   print(ans)
```

Figure 7: Test Code for Section 10

10 LEAST COVERAGE PREDICTION (RQ4)

In this experiment, we aim to show CodePilot’s usefulness in predicting the least covered statements for a test suite without execution. The capability to predict the least covered statements within a test suite holds significant usefulness in test case prioritization. Test case prioritization aims to optimize the testing process by identifying and executing critical test cases early, thus improving the efficiency of the testing cycle. Predicting the least covered statements allows testers to focus on areas of the code that have received minimal attention during testing by the current test suite.

We randomly selected one code snippet from our dataset (Figure 7). We utilized Google Atheris [10], a Python based coverage-guided fuzzer, to automatically produce a test suite comprising 1,000 test cases. Each test case within the generated test suite imparted distinct values to the variables in the code snippet (lines 3–6), ensuring the coverage of all conceivable branches during execution. We collected the ground truth by manually executing each test case on the code snippet and recording the code coverage.

We used CodePilot to predict the code coverage for all the test cases for the code in Figure 7. The columns `GT` and `PR` show the ranking results for the statements based on the ground-truth and the predicted coverage, respectively. For example, the statement at line 13 attained the lowest coverage in the ground truth (3rd rank).

As seen in Figure 7, CodePilot predicted correctly the least-covered statement at line 13. Moreover, according to the ground-truth, the top-5 least-covered statements include lines 9–13, and CodePilot also predicted correctly those with the ranks from 7–11. This is reasonable because those statements are conditioned via the `if` statement (line 9) inside the `for` loop (line 7).

11 PLAN AND EXECUTION SPACE (RQ5)

The planning from CodePilot is instrumental in efficiently curbing the expansive execution space that the LLM would otherwise

```
1  Code and Predicted Coverage
2  GT PR
3  >   > h = 1
4  >   > w = 100
5  >   > n = 200
6  >   > count1 = 0
7  >   > count2 = 0
8  >   > if h > w:
9  !   !     while w > 0:
10 !   !         count1 += h
11 !   !         if count1 >= n:
12 !   !             count2 += 1
13 !   !             break
14 !   !         else:
15 !   !             count2 += 1
16 !   !             w -= 1
17 >   > else:
18 >   >     while h > 0:
19 >   >         count1 += w
20 >   >         if count1 >= n:
21 >   >             count2 += 1
22 >   >             break
23 >   >         else:
24 >   >             count2 += 1
25 >   >             h -= 1
26 >   > print(count2)
```

(a) Test Code

```
Generated Plan
Step 1: Variable initialization : Assign values to
        variables h, w, n, count1, count2. Statements "h
        = 2", "w = 100", "n = 200", "count1 = 0",
        "count2 = 0" will be executed.
Step 2: If-else branching : Check if variable h is
        greater than variable w. If this condition is
        satisfied, execute the statements inside the if
        block. Else, execute the statments inside the
        else block. Since variable h is less than
        variable w, the if condition is not satisfied.
        So statements inside if block are not executed;
        statments inside else block are executed.
Step 3: While loop inside else block : Enter the while
        loop and iteratively check if variable h is
        greater than 0. Since the value of h is 2, the
        condition is satisfied and execute the
        statements inside the while loop.
Step 4: If-else branching inside while loop: Execute the
        statements inside the if block only if the value
        of count1 is greater than or equal to n. Since
        the value of count1 is 0, the if condition is
        not satisfied and execute the statements inside
        else block. Statements inside the else block are
        executed; statement "count2 += 1" is executed.
Step 5: Update the value of w: Decrement the value of
        variable w by 1. Statement "w -= 1" is executed.
Step 6: Repeat steps 3-5 until the condition in the
        while loop is not satisfied. The condition in
        the while loop will be satisfied for a maximum
        of 2 iterations since the value of h is 2.
Step 7: Print Statement: Print statements are executed.
        Statement "print(count2)" will be executed.
```

(b) Plan generated by CodePilot

Figure 8: Plan helps reduce the exploration space in execution

have to explore. Without the guidance of a plan, the provided code snippet presents numerous conditional branches and loops, each representing a potential pathway through the code. For example, let us consider the code in Figure 8. The total number of branches to be explored completely via execution to compute code coverage is $100 \times 2^2 = 400$ execution paths.

With CodePilot, the generated plan directs the LLM to discern crucial decision points (as seen in Steps 2 and 4 in the plan of Figure 8(b)) and relevant code segments, reducing the vast array of potential branches that would otherwise need exploration. Specifically, the LLM is strategically guided using the plan in Figure 8)(b) to navigate through these complex structures with a targeted focus and has successfully reduced the exploration space to 2 exact branches - the `if-else` block contained within lines 20–25. Importantly, it is crucial to note that with reduced exploration space, the LLM is able to accurately predict the code coverage (Figure 8)(a).

12 THREATS TO VALIDITY AND LIMITATIONS

The dataset we used might not be representative. However, this dataset has been used in the state-of-the-art CodeExecutor [16]. Our approach is tested only for Python and with GPT-3.5. The results for different datasets, languages, and other LLMs might vary. The experiments in RQ4 and RQ5 were conducted with single code snippets. The generalization requires larger datasets. However, our goal is to illustrate a CodePilot's application and a case study showing the benefit of planning in reducing a large execution space.

There are notable areas for improvement in CodePilot. Firstly, it faces challenges in accurately predicting cases where input values lead to runtime errors, stemming from its design that generates predictions for statement coverages without considering the validity of inputs. Secondly, there is difficulty in identifying control-flow statements, indicating potential issues with training data on runtime exceptions. Thirdly, CodePilot encounters challenges with recursive functions. Fourthly, its accuracy is reduced when handling programs with external libraries, suggesting the need for fine-tuning. Complex programs remain a difficulty for CodePilot.

13 RELATED WORK

In the realm of neural networks, Tufano *et al.* [30] utilize LLMs, particularly GPT [12] to predict statements requiring coverage without actual program execution. However, their approach, as discussed in Section 1, faces inherent limitations. CodeExecutor [16] is a Unixcoder-based neural network model pre-trained on the execution of a diverse set of programs to predict execution traces.

Code coverage has found application in various testing, fuzzing, and fault localization techniques and has been explored through machine learning [2, 3, 5, 6, 8, 9, 11, 14, 15, 17–19, 26–28, 31, 33].

The literature on computing code coverage is extensive and can be categorized into static and dynamic instrumentation.

In static instrumentation, Pavlopoulou and Young [23] introduced the concept of removing instrumentation after recording coverage data. They instrument bytecode in Java class files to track the execution of each basic block. For native code, Nagy and Hicks [21] employ binary instrumentation, triggering software interrupts upon first-time reach of basic blocks, and then rewriting the binary on disk to de-instrument that specific block.

Dynamic Instrumentation for native code includes Tikir and Hollingsworth [29], who develop a code coverage analyzer by extending DyninstAPI [4], a library for native code instrumentation. Chilakamarri and Elbaum [7] introduce a "disposable" coverage instrumentation for Java, involving instrumenting JVM bytecode and de-instrumenting probes once they are no longer required by overwriting them with NOPs (no-operation). Another Java-based approach by Misurda *et al.* [20] implements dynamic probe insertion and removal. Similar to Tikir and Hollingsworth [29], it pre-instruments by placing seed probes, which subsequently instrument basic blocks upon reaching them. Their method involves instrumenting the x86 code generated by the JIT compiler, relying on support from Jikes. SlipCover [24] operates in the same domain of dynamic instrumentation and program monitoring, improving by de-instrumenting lines/branches that have already been covered.

14 CONCLUSION

We introduce CodePilot that combines program analysis and planning to enhance code coverage prediction. Through collaboration with LLMs, CodePilot achieves high accuracy, with up to 55% in exact-match and 89% in statement-match, outperforming baselines by up to 33% and 19%, respectively. The effectiveness of CodePilot is demonstrated not only in predicting code coverage but also in accurately identifying the least covered statements. *This work advocates for the use of planning in combination with program analysis to guide the LLM in better navigating in the complex intricate tasks. We demontrates CodePilot as a proof-of-concept PA-based planning scheme guilding the LLMs in better code coverage prediction.*

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grant CNS-2120386 and the National Security Agency (NSA) grant NCAE-C-002-2021.

REFERENCES

[1] 2024. *Planning to Guide LLM for Code Coverage Prediction.* https://github.com/code-planning/code-coverage-planning

[2] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/3368089.3409748

[3] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1621–1633. https://doi.org/10.1145/3510003.3510230

[4] Bryan Roger Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329. https://doi.org/10.1177/109434200001400404

[5] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Opportunities. *IEEE Software* 38, 3 (2021), 79–86. https://doi.org/10.1109/MS.2020.3016773

[6] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jack Jiang. 2018. An Automated Approach to Estimating Code Coverage Measures via Execution Logs. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 305–316. https://doi.org/10.1145/3238147.3238214

[7] Kalyan-Ram Chilakamarri and Sebastian G. Elbaum. 2006. Leveraging Disposable Instrumentation to Reduce Coverage Collection Overhead. *Software Testing, Verification and Reliability* 16, 4 (2006), 267–288. https://doi.org/10.1002/stvr.347

[8] Andrea Fioraldi and Michael Maier. 2020. Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[9] Gregory Gay. 2017. Generating effective test suites by combining coverage criteria. In *Proceedings of the 11th International Workshop on Search-Based Software Testing (SBST)*. ACM, 65–82.

[10] Google Atheris [n. d.]. Google Atheris. https://github.com/google/atheris.

[11] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 72–82. https://doi.org/10.1145/2568225.2568278

[12] GPT [n. d.]. OpenAI. https://openai.com/.

[13] Ziniu Hu, Ahmet Iscen, Chen Sun, Kai-Wei Chang, Yizhou Sun, David A Ross, Cordelia Schmid, and Alireza Fathi. 2023. AVIS: Autonomous Visual Information Seeking with Large Language Model Agent. arXiv:2306.08129 [cs.CV]

[14] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 955–963. https://doi.org/10.1145/3338906.3340459

[15] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[16] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code Execution with Pretrained Language Models. arXiv:2305.05383 [cs.PL]

[17] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 371–383. https://doi.org/10.1109/ICSE48619.2023.00042

[18] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/3468264.3468580

[19] Breno Miranda and Antonia Bertolino. 2020. Testing Relative to Usage Scope: Revisiting Software Coverage Criteria. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–30.

[20] J. Misurda, J.A. Clause, J.L. Reed, B.R. Childers, and M.L. Soffa. 2005. Demand-driven Structural Testing with Dynamic Instrumentation. In *Proceedings of the 27th International Conference on Software Engineering, 2005*. 156–165. https://doi.org/10.1109/ICSE.2005.1553558

[21] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy, SP 2019*. IEEE, San Francisco, CA, USA, 787–802. https://doi.org/10.1109/SP.2019.00069

[22] Siqi Ouyang and Lei Li. 2023. AutoPlan: Automatic Planning of Interactive Decision-Making Tasks With Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 3114–3128. https://doi.org/10.18653/v1/2023.findings-emnlp.205

[23] C. Pavlopoulou and M. Young. 1999. Residual test coverage monitoring. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 277–284. https://doi.org/10.1145/302405.302637

[24] Juan Altmayer Pizzorno and Emery D. Berger. 2023. SlipCover: Near Zero-Overhead Code Coverage for Python. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1195–1206. https://doi.org/10.1145/3597926.3598128

[25] Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. 2023. ADaPT: As-Needed Decomposition and Planning with Language Models. *arXiv* (2023).

[26] Kostya Serebryany. 2021. *libFuzzer - a library for coverage-guided fuzz testing.* https://llvm.org/docs/LibFuzzer.html

[27] Robert Swiecki. 2021. *Honggfuzz.* https://github.com/google/honggfuzz

[28] Clang Team. 2023. *Source-based Code Coverage.* https://clang.llvm.org/docs/SourceBasedCodeCoverage.html Clang 15 Documentation.

[29] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2005. Efficient Online Computation of Statement Coverage. *Journal of Systems and Software* 78, 2 (2005), 146–165. https://doi.org/10.1016/j.jss.2004.12.021

[30] Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement. 2023. Predicting Code Coverage without Execution. arXiv:2307.13383 [cs.SE]

[31] Yi Wei, Bertrand Meyer, and Manuel Oriol. 2012. Is Branch Coverage a Good Measure of Testing Effectiveness?. In *Empirical Software Engineering and Verification*. Springer Berlin Heidelberg, 194–212. https://doi.org/10.1007/978-3-642-25231-0_5

[32] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2022).

[33] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Seyit Ahmet Çamtepe, and Yang Xiang. 2020. DeFuzz: Deep Learning Guided Directed Fuzzing. *arXiv preprint arXiv:2010.12149* (2020).

[34] Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A. Rossi, Somdeb Sarkhel, and Chao Zhang. 2023. ToolChain*: Efficient Action Space Navigation in Large Language Models with A* Search. arXiv:2310.13227 [cs.CL]