CRISPE: Semantic-Guided Execution Planning and Dynamic Reasoning for Enhancing Code Coverage Prediction

HRIDYA DHULIPALA, University of Texas at Dallas, USA
AASHISH YADAVALLY, University of Texas at Dallas, USA
SMIT SONESHBHAI PATEL, University of Texas at Dallas, USA
TIEN N. NGUYEN, University of Texas at Dallas, USA

While LLMs excel in understanding source code and descriptive texts for tasks like code generation, code completion, etc., they exhibit weaknesses in predicting dynamic program behavior, such as code coverage and runtime error detection, which typically require program execution. Aiming to advance the capability of LLMs in reasoning and predicting the program behavior at runtime, we present CRISPE (short for Coverage Rationalization and Intelligent Selection ProcedurE), a novel approach for code coverage prediction. CRISPE guides an LLM in simulating program execution via an *execution plan* based on two key factors: (1) program semantics of each statement type, and (2) the observation of the set of covered statements at the current "execution" step relative to all feasible code coverage options. We formulate code coverage prediction as a process of semantic-guided execution-based planning, where feasible coverage options are utilized to assess whether the LLM is heading in the correct reasoning. We enhance the traditional generative task with the retrieval-based framework on feasible options of code coverage. Our experimental results show that CRISPE achieves high accuracy in coverage prediction in terms of both exact-match and statement-match coverage metrics, improving over the baselines. We also show that with semantic-guiding and dynamic reasoning from CRISPE, the LLM generates more correct planning steps. To demonstrate CRISPE's usefulness, we used it in the downstream task of statically detecting runtime error(s) in incomplete code snippets with the given inputs.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Software and its engineering**;

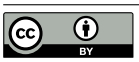
Additional Key Words and Phrases: AI4SE, Large Language Models, Planning, Predictive Code Coverage

ACM Reference Format:
Hridya Dhulipala, Aashish Yadavally, Smit Soneshbhai Patel, and Tien N. Nguyen. 2025. CRISPE: Semantic-Guided Execution Planning and Dynamic Reasoning for Enhancing Code Coverage Prediction. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE131 (July 2025), 22 pages. https://doi.org/10.1145/3729401

1 Introduction

Large language models (LLMs) [5, 23] excel at understanding source code and natural language texts, recognizing intricate patterns, syntax, and semantics. Typical use-cases include tasks involving text-based generation like code completion [27], code comprehension [32], *etc.* However, the LLMs primarily rely on static code representations, often lacking crucial context for reasoning about variable states, control flow along conditional branches or loop iterations, and their effects on other statements. This is further exacerbated in analyses involving program inputs, which require a deeper understanding of dynamic dependencies among statements at runtime. As a result, they are limited in predicting dynamic program behaviors and runtime state changes [8].

Authors' Contact Information: 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; Smit Soneshbhai Patel, University of Texas at Dallas, Dallas, USA, smitsoneshbhai.patel@utdallas.edu; Tien N. Nguyen, University of Texas at Dallas, Dallas, USA, Tien.N.Nguyen@utdallas.edu.



ACM 2994-970X/2025/7-ARTFSE131
https://doi.org/10.1145/3729401

Our general research aims to advance *the capabilities of LLMs in reasoning about dynamic program behaviors without requiring actual execution*. Toward that goal, in this paper, we explore a fundamental research question *"is the LLM capable of predicting whether a particular statement in a program would be covered if the program were executed"*.

In answering this question, we establish a proxy for the code coverage metric used in software testing, a crucial capability that can significantly benefit the testing process. *First*, it will enable an early detection of potential gaps in test coverage, thus allowing developers to address them before actual testing begins. *Second*, it can improve the testing's efficiency by providing insights into which parts of the code are likely to be tested and which are not, thus guiding the creation of more targeted and effective test cases, or guiding the prioritizing of certain test cases in more un-covered areas – all without execution. *Third*, it can be used to statically detect runtime errors in source code without execution. This is crucial for the early detection of exceptions in online code snippets before they are integrated into a project, especially when these are incomplete or non-executable. This is common in forums where code often lacks import statements for external libraries, and/or omits the definitions of enclosing methods/classes. With the knowledge of which statements in the code snippet will likely be covered during execution, one can determine if it will throw an error at runtime. In brief, in such scenarios of test coverage estimation and prioritization, and static detection of runtime errors, *code coverage prediction is valuable since execution is undesirable/impossible*.

Prior research on execution-free prediction of dynamic program behaviors can broadly be classified into two paradigms. The first one involves *pre-training language models* on large datasets with learning objectives to incorporate execution semantics-specific knowledge. CodeExecutor [16], based on UniXCoder [13], is a transformer-based, encoder-only architecture that is pre-trained on execution data to predict execution traces. From the traces, code coverage is computed. However, it is limited in generalization and scalability. TRACED [9], based on BERT/RoBERTa, is pre-trained on variables' values during the final execution of the last line within a loop to enhance execution knowledge. This is inadequate in learning how complex condition/iteration statements are handled.

The second paradigm leverages *LLMs' code understanding and reasoning capability*. Tufano *et al.* [26] circumvent value prediction with vanilla prompting to GPT-4 for code coverage prediction. However, due to the LLMs' susceptibility to hallucinations and error propagation, they fail when navigating a program space comprising multiple interdependent statements, including *conditional statements and loops*. To address those points, NExT [19] performs fine-tuning on execution traces to naturalize execution-specific knowledge into Chain-of-Thought (CoT) rationales via program state annotations. However, NExT sometimes violates program semantics in the reasoning steps, ending up 'executing' infeasible paths. To this end, CodePilot [8] leverages a CoT prompting strategy, centered on statement-specific semantics, guiding a LLM to autonomously plan the execution steps. However, it does not leverage the task-specific knowledge pertaining to code coverage prediction, leading to selecting infeasible paths as well. For instance, its step-by-step reasoning can incorrectly check all conditions within `if-elif-else` constructs in isolation, *without considering their mutually exclusive nature*, possibly predicting all conditions as true, which is logically incorrect.

In this paper, we present CRISPE (short for Coverage Rationalization and Intelligent Selection ProcedurE), a novel approach for code coverage prediction that *guides an LLM in autonomously planning predictive execution* by incorporating task-specific knowledge via *semantics-guided, observation-driven execution reasoning and action planning*. The observations include the current statement and the set of covered statements at the current "execution" step relative to all feasible options of code coverages (FOCCs). We formulate code coverage prediction as *a search through execution steps, using these FOCCs to determine whether the LLM is heading in the correct direction*. Such a design reinforces the LLM's understanding of program semantics towards predicting code coverage, discouraging it from considering execution steps that do not align with any of the FOCCs, and

encouraging when they do, thus mitigating error propagation as soon as an obvious error occurs. In short, combining program semantics on each statement and observation-driven reasoning/actions, we improve LLM's reasoning on code coverage, overcoming the infeasible path issue.

To derive the feasible options of block/code coverages (FOBC/FOCCs), we use the control flow graph (CFG), which maps the control flows among the statements. Using CFG provides several advantages. First, it enables the design of an algorithm that traverses a static CFG given an input to collect all possible sets of statements that will be covered. Second, in a CFG, all statements within the same block must be "executed" sequentially, thereby reducing hallucination and error propagation in LLMs on consecutive statements. Third, the coverage prediction problem is simplified from searching for a subset of $O(n)$ covered statements (where n is the total number of statements in a program) to searching for a subset of $O(m)$ covered blocks of statements (where m is the number of blocks in the CFG, and $m << n$). Finally, CFG enables the support of exception flow graph (EFG), which consists of the exception flows on the CFG from the beginning point of execution to all `throw` statements or exception exit points. This enables the support for static prediction of runtime errors.

To guide the LLM in predicting execution steps, we are inspired by ReAct planning (Reasoning and Action based on Observation) [31], which involves a dynamic interplay between reasoning and observation to guide decision-making. At each step, observations about the current FOCCs inform the reasoning process, which then determines the LLM's next action. This iterative process ensures that each decision is based on the most accurate and up-to-date information available, effectively narrowing down the execution paths to those that are most relevant. For our problem, by continuously refining the subset of applicable FOBCs/FOCCs, ReAct ensures that CRISPE remains focused and precise, thereby reducing error propagation and improving coverage prediction.

We conducted several experiments to evaluate CRISPE. Our experiments show that CRISPE achieves high accuracy in code coverage prediction in both exact-match and statement-match coverage metrics, improving over the baseline approaches. To demonstrate CRISPE's usefulness, we used it in a downstream task of predicting runtime errors for given inputs of (in)complete code snippets in online forums. The key challenge is the incompleteness/inexecutability of some online code snippets. To support that, we use an LLM to act as a fuzzer, generating inputs. Each is then fed into CRISPE with a code snippet to predict the code coverage. If the code coverage stops unexpectedly in an exception flow, and never reaches a normal exit point, it will report the runtime error. Our results show that it can predict the runtime errors with a high efficacy. We also showed that with the program semantics-guided execution planning and dynamic reasoning-based elimination strategy, CRISPE performs well in generating its plan for code coverage prediction.

In brief, this paper makes the following contributions:
a) CRISPE is a semantics-guided, observation-driven execution reasoning approach for LLMs to plan for *code coverage prediction*, leveraging feasible options of code/block coverages in an CFG.
b) Semantics-Guided, Observation-Driven Execution Planning and Dynamic Reasoning. CRISPE leverages program semantics-guided planning for a better execution semantics inference.
c) Empirical Evaluation. We conducted several experiments to show that CRISPE outperforms the existing code coverage prediction approaches. All code and data is available [3].

2 Motivation

2.1 Motivating Example and Observations

Fig. 1 shows a Python program that converts a numeric score into a letter grade. We input to the program different test cases with varying values for n and recorded the respective code coverages. We used the state-of-the-art CodePilot [8] to predict the code coverages for some test cases. CodePilot is a LLM-based agent that relies on Chain-of-Thought [28] to guide GPT-4 to create a stepwise prediction plan for the execution of statements. The coverage result is displayed in Fig. 1.

While the results are correct in several cases, GPT-based CodePilot violates the program semantics in execution, leading to choosing *infeasible execution paths* and reducing its prediction accuracy. For example, with the test case n=89 (Fig. 1), the condition (*B_grade_range*) evaluates to `true`, resulting in the assignment of grade *B*. Nevertheless, CodePilot's step-by-step reasoning mistakenly checks all conditions in isolation *without accounting for the mutually exclusive nature of the* `if-elif-else` *construct*, thereby predicting that all the later conditions are true, which is logically incorrect. Thus, CodePilot does not accurately simulate the execution of the conditions, leading to the erroneous conclusion that multiple conditions could be true simultaneously and eventually selects an infeasible path. In other cases (not shown), CodePilot chose the prediction of only one of two consecutive statements, which must be sequentially executed. A possible reason is that the underlying GPT model suffers from hallucination and error propagation. These cases highlight a limitation of classic prompting with LLMs in handling *sequential statements or complex conditions.*

```
1 >  n = int(input("Enter an integer number: "))
2 >  A_grade_range = range(90,101)
3 >  B_grade_range = range(80,90)
4 >  C_grade_range = range(70,80)
5 >  D_grade_range = range(60,70)
6 >  if n in A_grade_range:
7 !    grade = `A'
8 >  elif n in B_grade_range:
9 >    grade = `B'
10 >  elif n in C_grade_range:
11 >   grade = `C'
12 >  elif n in D_grade_range:
13 >   grade = `D'
14 >  else:
15 >   grade = `F'
```

Fig. 1. GPT-4 Incorrectly Covers `If-Else` Branches

Feasible Options of Code Coverage. As seen in the example, if we *could guide the LLM to avoid the infeasible paths*, we could help it improve code coverage prediction. However, while it is generally not possible to compute all of the feasible or infeasible paths for a given code, computing code coverage for a given input does not require such an enumeration. Let us take the example in Fig. 3a. One would not know how many iterations that the loop at line 4 would be executed. However, one can compute all FOCCs. In fact, one can formulate the code coverage prediction problem as *computing the set of statements that will be covered* when executing the code with that input.

Observation 1 (Feasible Option of Code Coverage – FOCC). *Instead of predicting the execution path as in CodePilot for a test case, one could model the code coverage prediction problem as predicting the feasible set of covered statements. We refer to each set as a Feasible Option of Code Coverage (FOCC) for the given test case.*

The total number of subsets of statements is 2^n, where n is the number of statements in the code. For large programs, this results in a vast number of subsets, making it impractical to list them all and select one of them. Moreover, in some cases, LLMs discard program semantics, e.g., selecting both branches of a condition statement or not executing consecutive statements that are supposed to be run sequentially. To capture program semantics in its execution and reduce the complexity involving 2^n subsets, one could leverage a Control Flow Graph (CFG). Fig. 2 displays the CFG for the code in Fig. 3(a). The CFG illustrates the sequence of execution of *statements* or *code blocks* within a program and *the conditions* that decide the control flow between two blocks, which are divided according to code semantics.

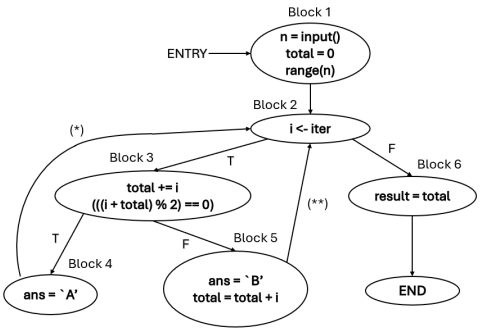


Fig. 2. Control Flow Graph for the Code in Fig. 3a

```
1  TEST CASE :  n = 1
2  n = int(input("Enter an integer: ")
3  total = 0
4  for i in range(n):
5    total += i
6    if (i + total) % 2 == 0:
7       ans = 'A'
8       continue
9    else:
10      ans = 'B'
11      total = total + i
12 result = total
```

(a) An Example with a `for` Loop and Conditions

```
1 FEASIBLE OPTIONS OF CODE COVERAGE (FOCC)
2 Set 1: Lines 1, 2, 3, 4, 12
3 Set 2: Lines 1, 2, 3, 4, 5, 6, 7, 11, 12
4 Set 3: Lines 1, 2, 3, 4, 5, 9, 10, 11, 12
5 Set 4: Lines 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

(b) All Feasible Options of Code Coverage (FOCCs) for the Code in Fig. 3a

```
1  Step 1: Input Processing: The program starts by taking an integer input. Since the test case directly
2  provides the integer n = 1, we consider that line 1 is executed to process this.
3  Observation 1: All sets of code coverage are applicable as they all include line 1.
4  Step 2: Variable Initialization: Line 2 will be executed.
5  Observation 2: All sets of code coverage remain applicable as they all account for the init of total.
6  Step 3: For Loop: Given that n = 1, the loop (line 3) will iterate once with i = 0.
7  Observation 3: All sets of code coverage remain applicable.
8  Step 4: First Iteration with i = 0: Within the loop, total will be updated by i, which is 0, so total
9  remains 0 (line 4). Then, the condition (i + total) % 2 == 0 is checked. Since (0 + 0) % 2 == 0, the
10 condition is true. Lines 4, 5, 6 are executed.
11 Observation 4: Sets 2 and 4 are applicable.
12 Step 5: 'if' block: Since the condition was true, ans is set to 'A', and the loop continues due to the
13 continue statement (lines 6 and 7).
14 Observation 5: Sets 2 and 4 are applicable.
15 Step 6: End of Loop and Final total Calculation: After the continue statement, the loop ends since n=1 and
16  there is only one iteration. Else block is not executed.
17 Observation 6: Set 2 is applicable
18 Step 7: Variable operation:Math operations are executed unless MathError is encountered.Lines 11 is covered.
19 Observation 7: Set 2 is applicable
20 FINAL CODE COVERAGE: Set 1: Lines 1, 2, 3, 4, 5, 6, 7, 11, 12
```

(c) The Exemplar Plan for the Code in Fig. 3a

Fig. 3. An Exemplar Plan guided by CRISPE for Code Coverage Prediction

Observation 2 (Control Flow Graph). *To capture program semantics in code execution, one could leverage a control flow graph (CFG). CFG is a graphical representation of the control flow within a program. Nodes in the graph represent basic blocks of code, such as individual or groups of statements that are executed sequentially, while edges represent the control flows between blocks, typically based on conditions such as loops (e.g.,* `for`, `while`*), conditional statements (e.g.,* `if-else`*), or function calls.*

Leveraging CFG, we can gain several advantages. First, we can reduce the consideration from 2^n subsets of statements to 2^m in which m is the number of blocks in the CFG. For the code in Fig. 3(a) and its CFG in Fig. 2, the number of subsets to be considered is reduced from 2^{12} to 2^6. Second, if we consider the control flows expressed in the CFG, we can automatically list all possible sets of blocks that are co-executed in the same execution (*i.e.*, order-agnostic). Note that the number of all possible sets is much smaller than that of execution paths. For Fig. 2, all such possible **sets** include:

(1) Block 1, block 2, block 6, END
(2) Block 1, block 2, block 3, block 4, block 6, END
(3) Block 1, block 2, block 3, block 5, block 6, END
(4) Block 1, block 2, block 3, block 4, block 5, block 6, END

Let us call them *feasible options of block coverages* (FOBCs). If we replace block i with its associated statements, we have feasible options of code coverages (FOCCs, Fig. 3(b)) at the statement level. Third, we can overcome the hallucination and error propagation issues with LLM-based CodePilot

where it mistakenly predicts the execution of consecutive statements. In a CFG, all the statements in the same block will need to be "executed" sequentially. Finally, we do not have to deal with the inaccuracy of the LLM in handling the control flow-altering statements (e.g., `continue`, `return`, etc.), because such flows are encoded within the edges. For example, `continue` in Fig. 3(a) is represented by the edge marked with (*). That leads to more correctly building of the FOBCs from the CFG.

2.2 Key Ideas

From the above observations, we design CRISPE (Coverage Rationalization and Intelligent Selection ProcedurE), a code coverage prediction approach, with the following key ideas:

2.2.1 Key Idea 1. [Semantics-Guided, Observation-Driven, Execution Reasoning and Action Planning]. We enhance LLM planning with guidance from retrieval-based framework via FOBCs/FOCCs. We teach the LLM to autonomously plan its own simulation of a step-by-step execution by guiding it via execution semantics on the CFG in accordance with the feasible options of code coverages (FOCCs). We formulate the code coverage prediction as a search through execution steps, *using FOBCs/FOCCs to decide if the LLM is headed in the correct direction.* The model is discouraged from taking steps that do not align with FOBCs/FOCCs, and encouraged as it does.

2.2.2 Key Idea 2. [Dynamic Reasoning by Leveraging ReAct Planning Strategy to Guide Predictive Execution]. The ReAct (Reasoning and Action based on Observation) planning procedure [31] involves a dynamic interplay between reasoning and observations to guide the decision-making process in the LLM. For the code coverage prediction task, in each step, we can incorporate observations about all feasible options of code coverages (FOCCs), which can help refine the reasoning process and drive the subsequent actions. Such an iterative process ensures that each decision is based on the most accurate and up-to-date information available, effectively narrowing down the code coverage space to those that are most relevant. For our problem, by continuously refining *the subset of applicable FOBCs/FOCCs*, ReAct ensures that CRISPE remains focused and precise, leading to an improvement in test coverage accuracy. This not only mitigates hallucinations, but also reduces the risk of taking infeasible paths, leading to a systematic exploration of viable paths and eventually a more precise code coverage prediction. Fig. 3(c) displays an ReAct exemplar plan.

3 CRISPE: Approach Overview

CRISPE Workflow. Fig. 4 displays CRISPE's overall workflow. It accepts an (in)complete code snippet and a given test case, and predicts the code coverage for the test case.

Overall, it has two phases:
(1) computing the feasible options of code coverages (FOCCs), and (2) leveraging LLM to predict code coverage via FOCC-guided planning following the ReAct scheme [31].

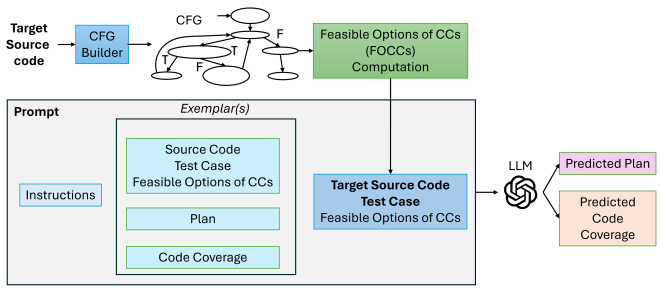


Fig. 4. CRISPE: Approach Overview

In the first phase, CRISPE takes the given source code and builds the static CFG. Then, it uses an algorithm (Section 4) that takes the test case as an input, traverses the CFG, and builds the feasible options of block coverages (FOBCs) and from which derives the feasible options of code coverages (FOCCs). In the second phase, the computed FOCCs are used in constructing the prompt following ReAct planning. The prompt has three parts,

beginning with an instruction that outlines the objectives and constraints of the reasoning steps we expect the LLM to follow. The second part of the prompt is an exemplar that provides the LLM with: (1) an exemplar code, (2) an input test case, (3) the FOCCs, (4) the ReAct plan, and (5) the expected output code coverage for this exemplar code and test case. This exemplar enables in-context learning in the LLM, providing an example showing how different elimination strategies and code coverage scenarios can unfold. The third part contains the target code, test case, and FOCCs.

ReAct Planning with FOCCs. An important part in the prompt is the instruction to the LLM to form its own plan. We leverage the ReAct scheme [31], which was originally designed in the domain of robotic planning. ReAct emphasizes deliberate and methodical actions combined with continuous observation and assessment to achieve optimal performance. We apply ReAct as follows. Within the context of code coverage prediction, *"actions"* ($\mathcal{A}$) entail a predictive execution step, involving the prediction of statement execution within a block, and updating the current set of covered statements. A possible series of planned actions include initializing variables, performing operations, evaluating conditions through branching and other statement types. *"Observations"* ($\mathcal{O}$) involve the retrieving of the *applicable FOCCs*. *"Reasoning"* ($\mathcal{R}$) encompasses evaluating the matching of applicable FOCCs and determining the subsequent actions for the next block and statement.

Illustrating Example. To illustrate CRISPE in operation, let us analyze a specific step involving an `if` statement using the running example in Fig. 3(a) and Fig. 2. To be specific, consider the `if` statement within the loop in Fig. 3(a). In the corresponding CFG in Fig. 2, this `if` statement corresponds to Block 3, which handles the conditional check, and Block 4 (Line 7) and Block 5 (Lines 10 and 11), which handle the actions taken if the condition is `True` or `False`, respectively. The continuation of Block 3 deals with the increment of the `total` variable.

CRISPE breaks down this process into a series of actions ($\mathcal{A}$) and observations ($\mathcal{O}$). Initially, the action involves evaluating the statement "`if (i + total) % 2 == 0`", corresponding to Block 3. This reasoning is visible in Step 2 of Fig. 3(c). The model then makes an observation to determine the feasible path based on the result (as noted in Observation 2 of Fig. 3(c)). If the condition is true (Step 3 of Fig. 3(c)), it transitions to Block 4 and sets the value of `ans` to `'A'`. If false, it transitions to Block 5 and sets the value of `ans` to `'B'`. This process exemplifies the predictive execution step, where it updates the current set of covered statements based on the evaluated condition.

Subsequently, the model observes the new state and checks which FOCCs match this updated state. Regardless of the branch taken, the next action involves incrementing `total` by `i`, which is part of the continuation in Block 3. The final observation checks the updated state against the FOCCs to ensure it matches one of the precomputed feasible options. For instance, when the loop runs with `i = 0`, the condition in `if ((i + total) % 2) == 0` is checked (Steps 4, 5 and 6 in Fig. 3c). Given that `total` is initially 0, the condition evaluates to `True`, resulting in `ans = 'A'` (Block 4). Next, the action increments `total` to `0 + 0 = 0`. The LLM then observes this updated state and verifies that it aligns with one of the feasible coverage paths (as shown in Observations 4, 5 and 6 in the plan listed in Fig. 3c), *i.e.*, Set 2 in the FOCCs which includes the lines {1, 2, 3, 4, 5, 6, 7, 11, 12}.

4 Feasible Options of Code Coverages (FOCCs) Building Algorithm

Let us explain our algorithm (Listing 1) to build the feasible options of code coverages (FOCCs) from the CFG of the given code. First, CRISPE constructs the Abstract Syntax Tree (*AST*) and the CFG (*CFG*) for that code (lines 12–13). The *CFG* is essential to capture all possible execution paths. Since the *CFG* can not capture the method call mapping, a *call_graph* is also constructed to integrate the method call mapping (line 14). We then connect the CFGs for individual methods together accordingly to the static call graph. We also extract the *blockLines* and the *connections* among all blocks (lines 15–16). CRISPE invokes the `FindPaths` function, with the *connections*, *startblock* (B_0), and *endblock* (B_{END}) parameters (line 17). This *recursive function explores all possible paths* from

the *startblock* (B_0) to the *endblock* (B_{END}) (lines 1–9) and returns feasible paths (line 17). Feasible paths are composed of *CFG* block IDs. Subsequently, the `MapPathsToLineNumbers` function transforms *CFG* blocks IDs into the corresponding statement numbers by using *blockLines* (line 18).

To accommodate the paths involving exception flows in an Exception Flow Graph (EFG), we also consider the throw statements or exception exit points as the end blocks (B_{END}s). The `FindPaths` function in this case would return an exception flow path to the abnormal exit points.

Listing 1. Feasible Options of Code Coverage Building Algorithm

```
1  function FindPaths(MAP edges, Block B_start, Block B_END, List path = [])
2      path.append(B_start)
3      if B_start == B_END then
4          return [path]
5      feasible_paths = []
6      for block connected to B_start in edges do
7          if block not in path or block = B_END then
8              feasible_paths.extend( FindPaths(edges, block, B_END, path) )
9      return feasible_paths
10
11 function GetFeasiblePaths(Code code)
12     AST ast = BuildAST(code)
13     CFG cfg = BuildCFG(ast)
14     call_graph = BuildCallGraph(cfg)
15     blockLines = ExtractCFGBlocks(cfg, call_graph)
16     connections = ExtractCFGraphEdges(cfg, call_graph)
17     cfg_paths = FindPaths(connections, B_0, B_END)
18     paths = MapPathsToLineNumbers(cfg_paths, blockLines)
19     return paths
```

For illustration, consider Fig. 2, which depicts the *CFG* of the Python code in Fig. 3(a). Following Listing 1, CRISPE constructs the *AST* and *CFG* (lines 12–13) for this code. Next, *blockLines* and *connections* are extracted (lines 15–16). Notably, in Fig. 2, the *startblock* is identified as block 1, and the *endblock* as *END*. It then executes the `FindPaths` function (line 17) with the *connections*, *startblock* and *endblock* parameters. The `FindPaths` function (lines 1–9) traverses the graph and returns four distinct paths. Subsequently, `MapPathsToLineNumbers` (line 18) converts these paths from CFG block paths into the following statement paths (listed below). These statement paths are converted into FOCCs and used in the prompt for the LLM (see Section 5):

(1) 1, 2, 3, 12, END
(2) 1, 2, 3, 4, 5, 6, 12, END
(3) 1, 2, 3, 4, 5, 9, 10, 11, 12, END
(4) 1, 2, 3, 4, 5, 6, 9, 10, 11, 12, END

5 Guiding LLM with ReAct Planning

This section explains how we use the FOCCs produced from Listing 1 to build the prompt to instruct the LLM to create its own plan in computing the code coverage. As outlined in Fig. 4, the key part of the prompt in CRISPE is an exemplar, which comprises an illustrative plan for a code snippet, its test case, and all the FOCCs. Based on the exemplar, the LLM follows the ReAct scheme [31] at each step to narrow down the set of FOCCs by eliminating the irrelevant ones, until it derives the final resulting code coverage for the input code snippet and test case. Specifically, the execution plan based on ReAct contains the three main components: *Reasoning*, *Actions*, and *Observations*.

5.1 Reasoning ($\mathcal{R}$) on Execution Semantics

To compute code coverage, the LLM must accurately reason through the execution steps of the code's statements. Beyond the sequential execution of statements, three types of statements can alter this flow: branching statements, loop statements, and method calls.

Branching statements (`if` or `switch` statements) are critical in an execution based on conditions. An `if` statement evaluates a condition and executes the corresponding block if it is true; if false, we move to the next `elif` condition or the `else` block if available. The exemplar plan must capture these nuances. For example, in Fig. 5(a) (line 4), the plan considers the `if-else` construct in Fig. 5(b) (Step 3) by outlining the condition and statements executed based on the evaluation of that condition.

```
1  (a) input_string =
       input("Enter a list of
       numbers: ")
2  numbers = [int(num) for num
       in
       input_string.split(',')]
3  for number in numbers:
4    if number > 0:
5      print(f"{number} is
         positive")
6    elif number < 0:
7      print(f"{number} is
         negative")
8    else:
9      print(f"{number} is zero")
10 print("End of program")
11
12 FEASIBLE OPTIONS OF CODE
       COVERAGE (FOCCs) -
13 (1) Error
14 (2) Lines 1, 2, 3, 4, 5, 10
15 (3) Lines 1, 2, 3, 4, 6, 7,
       9, 10
16 (4) Lines 1, 2, 3, 4, 6, 9, 10
17 (5) Lines 1, 2, 3, 4, 5, 6,
       7, 10
18 (6) Lines 1, 2, 3, 4, 5, 9, 10
19 (7) Lines 1, 2, 3, 4, 5, 6,
       7, 10
20 (8) Lines 1, 2, 3, 4, 5, 6,
       7, 8, 9, 10
21
22 GIVEN TEST CASE -
23 numbers = [7, -5]
```

```
1  (b) PLAN:
2  Step 1: Input Processing: The program starts with an input
       string that gets split into numbers based on commas. Since
       the test case provides the numbers [7, -5], we consider
       that lines 1 and 2 are executed to process this input.
3  Observation 1: All sets of code coverage are applicable as they
       all include lines 1 and 2.
4  Step 2: Looping through numbers: Given numbers are 7 and -5,
       the loop (line 3) iterates twice.
5  Observation 2: All sets of code coverage remain applicable as
       they all account for the loop execution (line 3).
6  Step 3: First Iteration with number = 7: For the first number
       (7), the condition in line 4 (if number > 0) is satisfied.
       Thus, line 5 will be executed, printing that 7 is positive.
7  Observation 3: Sets 2, 5, 6, and 8 are applicable as they
       include lines 4 and 5.
8  Step 4: Second Iteration with number = -5: For the second
       number (-5), the condition in line 4 is not satisfied, so
       the program will check the elif statement in line 6. Since
       -5 < 0, line 7 is executed, printing that -5 is negative.
9  Observation 4:Sets 5,7,8 are applicable, they include lines 6,7.
10 Step 5: End of Program: After iterating through all numbers,
       line 10 will be executed to print "End of program."
11 Observation 5: From observations, only sets that account for
       positive and negative number checks and include line 10
       are viable. This narrows down to sets 5, 7, and 8.
12 Step 6: Zero Check: The test case does not include 0, so the
       else block (line 8) is not executed. This eliminates set 8
       which includes lines 8--9.
13 Observation 6: Sets 5 and 7 are applicable, but since there's
       no condition where a number equals zero, set 5 is
       accurate. Therefore, the best set of statements that will
       be covered by the test case is:
14 Set 7: Lines 1, 2, 3, 4, 5, 6, 7, 10
```

Fig. 5. CRISPE Produces a Plan to Compute Code Coverage for a Code Example based on FOCCs

Loop statements, such as `for` and `while`, are iterative processes. The `for` statement is used when the number of iterations is known beforehand, consisting of initialization, condition, and increment/decrement parts. It executes as long as the condition is true. Conversely, the `while` statement is used when the number of iterations is uncertain, executing as long as its condition holds true, with the loop variable updated within the loop block. A `for` statement is more structured, while a `while` statement offers greater flexibility. For instance, the planning steps in Fig. 5(b) (lines 4-10) show instructions for executing `for` loop in Fig. 5(a) (line 3). The plan includes reasoning for executing the method calls in which the called method is executed. Other statements are also supported.

5.2 Actions ($\mathcal{A}$) on Execution Steps and FOCCs

The LLM is instructed to create its own plan with the steps for actions. First, the first type of actions include the predictive execution step after the reasoning based on the semantics of the current statement. This includes determining the type of current statement, whether branching conditions hold, etc. The actions guide the model in predicting the coverage of specific statements, evaluating conditions, and transitioning to subsequent parts of the code. For example, after a condition of an `if` statement is evaluated to `true`, the LLM is instructed to take an action to continue "executing" the `true` branch. Second, the second type of actions involves updating the relevant FOCCs after making an observation to compare the current step and the current coverage with the set of FOCCs.

5.3 Observations ($\mathcal{O}$) on FOCCs

Observations are based on the reasoning and actions taken. When a statement is predicted as covered within a particular step (Reasoning + Action), the LLM is instructed to observe the current

FOCCs. The FOCCs that are not consistent with the current execution steps or invalid blocks will be eliminated. Such elimination actions ensure that only the sets containing the covered statement are considered in subsequent steps. Observations of the succeeding steps further refine the selection of feasible sets of code coverage based on the outcomes of reasoning and actions in each step. Finally, the last one consistent with the final prediction step is returned as the resulting code coverage.

5.4 Prompt Structure and Illustrating Example

Fig. 6 illustrates the structure of CRISPE's prompt for predicting code coverage. To provide the model with only basic understanding, we included a simple exemplar (see Fig. 3). Let us consider the sample program in Fig. 5(a). The code snippet contains a series of `if-else` branching statements within a `for` loop, resulting in various FOCCs depicted in Fig. 5(a). For a given test case `n = [7, -5]`, the LLM is then prompted to use the CRISPE prompt to plan and predict code coverage.

```
1 COVERAGE PREDICTION
2 Given a program, test case and the possible sets of statements that
        will be covered by the test case, you are to create a step by
        step reasoning to choose slowly eliminate the non-applicable set
        of statements based on the reasoning for the test case.
3 Refer to the examples below -
4 « sample code, as in Figure 2(a) »
5 « Feasible sets of Code Coverage (FOCCs), as in Figure 2(b) »
6 « Exemplar plan and resulting code coverage, as in Figure 2(c) »
7
8 Make sure to follow the same format to predict code coverage.
9 Similarly, create a step by step reasoning and choose the best set of
10 statements that will be covered by the test case for the below program
11 « Test code... »
12 « FOCCs (acquired from CFG Builder)... »
13 « Test plan (generated by the Plan Formulation phase)... »
```

Fig. 6. Structure of CRISPE's prompt

The plan (depicted in Fig. 5(b)) created by CRISPE is systematically organized into steps that detail the actions to be taken based on reasoning during program execution. Each step is accompanied by observations that identify relevant FOCCs. As seen in Fig. 5(b), CRISPE effectively navigates the appropriate branching blocks by employing Reasoning + Action steps and Observations. This process is particularly evident in Steps 3, 4, and 6 in Fig. 5(b), where the model reasons about which branching statement will be executed based on the current variables' values. This reasoning guides the model in deciding whether a particular branching condition is satisfied. Then, the LLM uses Reasoning and Action in each step to observe which sets of code coverages contain the covered statements (corresponding to Observations 3, 4, and 6 in Fig. 5(b)) and eliminates the irrelevant sets.

As guided by the semantics from FOCCs, the model systematically performs reasoning and filters out the non-feasible options of code coverage, ultimately identifying the most accurate FOCC. This can be seen in Observation 6 of Fig. 5(b), and thus showing the model's refining its plan.

6 Empirical Evaluation

To evaluate CRISPE, we seek to answer the following research questions:

RQ1. [Code Coverage Prediction Comparison on Complete Code]. How well does CRISPE perform compared to the existing code coverage prediction approaches? This is equivalent to "how well does it perform in predicting whether a specific statement/branch is covered for a test case?".

RQ2. [Usefulness: Prediction of Runtime Errors for an Input of Incomplete Code]. How well does CRISPE perform in predicting if a given input triggers a runtime error in an incomplete code snippet? This ability in CRISPE is useful for *the static detection of runtime errors* in incomplete/non-executable online code snippets in which traditional program analysis might not work due to incomplete code. Such early detection is desirable before code snippets are adopted to a project.

RQ3. [Performance on Different Types of Statements]. How well does CRISPE perform code coverage prediction on different types of statements?

RQ4. [Performance of Planning]. How well does CRISPE generate plans for code coverage?
RQ5. [Efficiency in Code Coverage Prediction]. How is code coverage efficiency of CRISPE?
RQ6. [Sensitivity Analysis]. To which extent can CRISPE be applied to incomplete code?

7 Code Coverage Prediction on Complete Code (RQ1)

Dataset. We opt for the FixEval [14] dataset, which covers a wide variety of coding styles and comprises several nested structures, making it suitable for testing code coverage prediction approaches. In FixEval, each programming problem has multiple submissions. We selected one submission each for a randomly selected subset of 50 problems. The choice of the submissions was determined based on several criteria, including code length, branching factor, number of computed FOCCs, and program complexity. Subsequently, multiple test cases were created for each code snippet, aimed at covering different execution traces and edge cases. On average, for each of the 50 code snippets, 5–6 test cases were selected, resulting in a coverage prediction dataset with 275 unique data instances.
Baselines. We selected the state-of-the-art CodePilot [8] as a baseline because it is LLM-based and has demonstrated superior performance over CodeExecutor [16], Tufano *et al.* [26], and others. CodePilot leverages Chain-of-Thought, further improving effectiveness. Thus, we used CodePilot with GPT-4 (GPT-4+CodePilot) as a baseline, following its provided instructions and exemplars. We also included Claude-3.5 [2], pairing it with both CodePilot (Claude+CodePilot) and CRISPE (Claude + CRISPE). We compared the baselines with our approaches (GPT-4+CRISPE and Claude+CRISPE).
Metrics. We employed two metrics: *exact-match accuracy* and *statement-match accuracy*. *Exact-match accuracy* measures the number of programs for which the predicted sequence of statement/branch coverages exactly matches the target coverage sequence, indicating perfect accuracy across all statements/branches within the actual execution. Conversely, *statement-match accuracy* assesses the percentage of correctly predicted covered or not-covered statements. While *statement-match accuracy* evaluates accuracy at the individual statement level, *exact-match accuracy* provides an assessment at the program level. Both metrics are calculated as the averages over all instances.
Results. As seen in Table 1, GPT-4+CRISPE achieves highest exact-match and statement-match accuracies than other variants and the baselines.

Table 1. Accuracy in Code Coverage Prediction on Complete Code (RQ1)

	GPT-4 +CodePilot [8]	Claude +CodePilot	**GPT-4 +CRISPE**	**Claude +CRISPE**
Exact-Match	50.58%	42.21%	**54.54%**	50.83%
Stmt-Match	63.24%	59.88%	**84.55%**	66.88%

Comparing GPT-4+CRISPE with GPT-4+CodePilot and comparing Claude+CRISPE with Claude+ CodePilot, we can see that CRISPE is able to boost the performance of both the LLMs GPT-4 and Claude more than CodePilot. GPT-4+CRISPE relatively improves over GPT-4+CodePilot by 7.83%, and Claude+CRISPE over Claude+CodePilot by 20.4%. Upon examining the plans generated by the LLMs, we observed that CRISPE's higher accuracy can be attributed to its reliance on FOCC-guided, reasoning-based elimination of infeasible execution paths within the ReACT framework. *In the case of GPT-4+CodePilot, we observed that among the mispredictions, for 36.8% (50/136) of the instances, the predicted code coverage was not among any of the possible FOCCs.* In other words, *with CodePilot, GPT-4 failed to conform to program semantics in 18.2% of the instances.* In contrast, since CRISPE directs GPT-4 via program semantics-specific reasoning to choose from the FOCCs—the most suitable set of coverage options for the test case and program through reasoning-based elimination, *such non-program semantics-conforming errors are avoided.*

Furthermore, we observed that *CRISPE's focus on reasoning-based elimination rather than independent step-by-step reasoning helped it reduce the number of reasoning steps significantly.* Such a

reduction in the number of computations also helps *minimize the cascading error stemming from errors in intermediate steps*. For instance, the *average number of reasoning steps in CodePilot is 13.3 as opposed to 7.1 in CRISPE*. This observation *corroborates with our Key Ideas 1 and 2 (see Section 2.2) that incorporating FOCCs into the planning strategy helps mitigate LLM hallucinations and error propagation*, thus resulting in a more accurate prediction of code coverage.

Moreover, comparing GPT-4+CRISPE with Claude+CRISPE (columns 2 and 3) and comparing GPT-4+CodePilot with Claude+CodePilot (columns 4 and 5), we observe that with the same prompting strategy (*i.e.*, either CRISPE or CodePilot), GPT-4 performs coverage prediction better than Claude. Moreover, as seen, CRISPE is able to boost Claude's performance more than GPT-4.

In fact, comparing the result of GPT-4+CodePilot (50.58%) with that of Claude+CRISPE (50.83%), CRISPE enhances Claude (a weaker model in code coverage prediction) to perform at the same accuracy level with a stronger model GPT-4 in coverage prediction used in the CodePilot prompting. This further confirms that CRISPE can achieve better accuracy than the state-of-the-art CodePilot.

Finally, statement-match accuracies show similar trends as with exact-match accuracies in Fig. 1. CRISPE outperforms CodePilot by 33.7% with GPT-4 (GPT-4+CRISPE > GPT-4+CodePilot). At the statement-level, we can attribute this to the use of semantics-guided execution planning in FOCCs, which *enforces the co-occurences of predicted coverages of certain statements*, and reduces the number of incorrect statement coverages. This also corroborates with Key Idea 1 on FOCCs (Section 2.2).

RQ1. *CRISPE boosts the coverage prediction accuracies of GPT-4 and Claude more than the baseline.*

8 Static Detection of Runtime Errors in Incomplete/Non-executable Code (RQ2)

Online forums provide a good source for developers seeking solutions for their technical problems. However, they might contain runtime errors. Detecting such errors in online code snippets is challenging due to its incompleteness, which makes traditional program analysis non-applicable. In this experiment, we aim to show CRISPE's capability in predicting whether *a given input to a given incomplete code could cause a runtime error. This capability is useful since one can build a tool to statically detect runtime errors in inexecutable code snippets, which otherwise is not possible*. For example, one could first leverage a LLM-based test case generator (e.g., Fuzz4All [30]) to produce the inputs that potentially trigger runtime errors for the given code. Then, each generated input could be fed to CRISPE to predict if the input can trigger a runtime/exception in the incomplete code.

Dataset. We first used the FixEval dataset [14], derived from the CodeNet dataset [22]. FixEval is organized by problem IDs, each representing a unique programming challenge with multiple submissions with errors. With its input, each Python code is executable and could contain runtime errors. We used `pythonhunter` [1] to execute each code snippet to obtain execution traces. We randomly sampled 50 buggy and 50 non-buggy Python code snippets from the dataset to form a balanced dataset of 100 instances, which is distinct from those in RQ1. This subset comprises medium to large programs with varying degrees of complexity to ensure a diverse range of scenarios.

To obtain the incomplete code dataset, we chose to make the code in our dataset incomplete/uncompilable by removing the `import` statements to built-in/external libraries and removing enclosing methods/classes. The rationale is that missing `import` statements and enclosing methods/classes is common with online incomplete code [33]. Moreover, no dataset exists that contains incomplete code along with its complete versions, which we require for execution to gather ground-truth coverage.

Procedure. All code samples in FixEval are designed with user input prompts for variable initialization. For automation, we utilized GPT-4 to automatically generate test inputs, including those that would trigger runtime errors. The use of an GPT-4 aims to accelerate the test case generation process, as opposed to manual test creation. We ran the original complete code with the generated

test inputs to collect the ground-truth runtime errors. In total, we have 1,384 instances (inputs): 339 buggy instances (error-triggering inputs) and 1,045 non-buggy ones (non-error-triggering inputs).

Metrics. We used the following metrics. True Positive refers to the number of cases where CRISPE accurately predicts the presence of an exception in an error-triggering instance (i.e., the coverage includes the statement with exception). False Positive represents the number of cases where the input triggers an exception, but CRISPE predicts that it does not. True Negative and False Negative are defined similarly. Accuracy is calculated as the ratio of correctly identified instances (both TPs and TNs) over the total number of instances.

Results. As seen in Table 2, out of the 336 test cases that would have triggered an exception, CRISPE accurately predicted 276 cases (TP=81.42%) and incorrectly predicted 63 cases (FN=18.58%). The high TP rate shows that with FOCCs, CRISPE can recognize the paths that lead to exceptions in buggy code. This could be due to its effective training on patterns that typically cause exceptions (e.g., division by zero, null pointer exceptions). Moreover, it has a low FN rate (18.58%), perhaps due to ambiguous code or conditions.

Table 2. Accuracy on Static Detection of Runtime Errors (RQ2)

Actual	**Predicted**		**Accuracy**
	Buggy	*Non-Buggy*	
Buggy	TP = 276/339 (81.42%)	FN = 63/339 (18.58%)	**86.63%**
Non-Buggy	FP = 122/1045 (11.68%)	TN = 923/1045 (88.33%)	

Conversely, among the 1,045 test cases that would not trigger an exception or runtime error, CRISPE made correct predictions in 923 cases (with 100% Exact-Match Accuracy) and incorrect predictions in 122 cases. This suggests that CRISPE has a good understanding of normal coverage behavior without exceptions. Moreover, CRISPE has a low FP rate of 11.68%. We investigated and found that in those cases, the conditions are not obvious or involve intricate dependencies.

The high True Positive and True Negative rates suggest that CRISPE is effective at recognizing both the presence and absence of exceptions in most scenarios, particularly when the code patterns are straightforward and follow common exception-handling practices. If a test case triggers an exception or a runtime error in a code snippet, CRISPE can detect it with an accuracy of 86.63%.

RQ2. *Overall, the accuracy of CRISPE in predicting runtime errors in a code snippet is 86.63%.*

9 Performance on Different Statement Types (RQ3)

We analyzed the results in RQ1 to assess the *GPT-4's comprehension via CRISPE's prompting strategy*, of the three types of branching constructs: `if-else` conditions, `for` and `while` loops, and method calls.

Metrics. We use the following metrics. *Branch-match Accuracy* is defined as the ratio of correctly predicted `if-else` branch coverage (i.e., the model correctly identifies the executed branch given a condition) to the total number of branches present in the code. *Loop-match accuracy* is calculated as the ratio of correctly predicted loop coverage to the total number of distinct loops in the code. A loop's coverage is deemed accurate if all the statements within the loop are correctly predicted as covered or non-covered. *Inter-pro-match accuracy* is the ratio of accurately predicted *inter-procedural flow* for a method call to the total number of method calls. The inter-procedural flow for a method call is considered as correct if the first statement in the called method is predicted to be covered.

Results. As seen in Table 3, the FOCCs-guided prompting strategy in CRISPE helps GPT-4 improve over CodePilot in all three branching constructs. The relative improvement is **41.71%, 41.70% and 35.40%** in *branch-match accuracy, loop-match accuracy and inter-pro-match accuracy*, respectively.

The improvement in coverage prediction can be attributed to two key factors. Firstly, by computing and utilizing all FOCCs, CRISPE provides the model with a set of plausible execution paths. This

guided planning significantly enhances the model's ability to predict accurate code coverage, as it narrows down the possibilities and focuses the model's reasoning process. Furthermore, CRISPE's integration of observations, reasoning, and actions ensures that the model can methodically evaluate and update its predictions based on observed outcomes, leading to higher accuracy.

The CRISPE's relatively higher branch-match accuracy indicates that is particularly effective in handling conditional statements, which are critical for accurately predicting program flow. With a relative improvement of 41.70% in loop-match, CRISPE demonstrates better performance in predicting the coverage of loops. This improvement suggests that it is adept at understanding iterative constructs, ensuring that all statements within loops are correctly identified as (non-)covered. Finally, the 35.40% improvement in inter-pro-match accuracy highlights CRISPE's ability to correctly predict the coverage of method calls and their execution flows. This capability is crucial for *inter-procedural analysis*.

Table 3. Accuracy on `If-Else`, Loops, and Method Calls (RQ3)

Approach	Evaluation Metrics (%)		
	Branch-match	*Loop-match*	*Inter-pro-match*
GPT-4+CodePilot	54.23	41.65	54.35
GPT-4+CRISPE	**76.85**	**59.02**	**73.59**

The performance metrics of CRISPE can be explained relative to each other considering the nature of the code structures involved. Firstly, the loop-match accuracy (59.02%) is significantly lower compared to branch-match and inter-pro-match accuracies. This suggests that it faces challenges in accurately predicting the coverage of loop constructs. Loops often contain nested `if-else` statements and other nested loops, which add layers of complexity. Secondly, the higher branch-match accuracy indicates that CRISPE is more effective at predicting the coverage of branching statements compared to loops. While branches can also be complex, they are typically less dynamic than loops because they represent single points of decision as opposed to multiple points over multiple iterations. Lastly, the inter-pro-match accuracy (73.59%) is close to branch-match accuracy, suggesting that *CRISPE handles inter-procedural flow relatively well.* This is because that method calls often follow a more structured and predictable flow compared to loops with nested conditions.

RQ3. *With CRISPE, GPT-4 performs better on all three branching constructs than with the baseline.*

10 Correctness of Generated Plans (RQ4)

Procedure. The objective of this experiment is to evaluate the correctness of the plans generated by GPT-4 with CRISPE's prompting.

Out of the total 275 instances in RQ1, the plans for 100 randomly selected programs were checked manually for correctness. Specifically, the reasoning and observations of each plan were manually evaluated by three judges to determine whether the model correctly linked *the reasoning for a statement being covered or not and the elimination of certain coverage sets based on this reasoning.*

Metrics. We define the following metrics to evaluate the plans generated by CRISPE. For a given code instance, a plan of coverage prediction is considered as correct if *all the steps* of observations, reasons, actions, and the above linking between the reasoning and elimination of irrelevant FOCCs are correct. *Plan accuracy* is the ratio of the instances with correct plans over the total number of instances. A step of either observation, reasoning, or action is considered correct if it is correct with respect to the program semantics at the current statement. *Step accuracy* for an instance is the

Table 4. Correctness of Generated Plans (RQ4)

Approach	Evaluation Metrics (%)	
	Plan accuracy	*Step accuracy*
GPT-4+CRISPE	82	92.4

```
(a) TARGET PROGRAM - TEST CASE : 5 5 5
import sys
sys.setrecursionlimit(300000)
def solve(x,y,k):
  if x==y==0: return []
  if x<0: return [(-x,y) for x,y in solve(-x,y,k)]
  if y<0: return [(x,-y) for x,y in solve(x,-y,k)]
  if x>y: return [(x,y) for y,x in solve(y,x,k)]
  if (x+y)%k==0: return [(k,0)]*(x//k)+[(0,k)]*(y//k)+...
  if x+y>=2*k:
    if x%k+y%k>=k: return
          [(k,0)]*(x//k)+[(0,k)]*(y//k)+solve(x%k,y%k,k)
    else: return
          ([(k,0)]*(x//k)+[(0,k)]*(y//k))[:-1]+solve(.)
  if x+y<k and (x+y)%2==1: return solve(x,y+k,k)+[(0,-k)]
  if k&1 and x+y>k and (x+y)%k%2==0: return solve(.)+...
    z=(x+y)//2
  return [(k-z+x,z-x),(z-k,z)]...
// (b) LLM response when prompted using CodePilot
Step 1: Import Statement : ...
Step 2: sys.setrecursionlimit() : ...
Step 3: Function Definition :
Step 4: ... Each if-else statement will be executed one
        by one ... There are multiple conditions in this
        code snippet, so it is difficult to predict the
        exact execution path without knowing input values.
Step 5: Variable initialization and math operations:...
Step 6: Print Statement : ...
So the code coverage for the given snippet cannot be
accurately predicted without knowing input values
```

```
//(c) LLM response as prompted with CRISPE
PLAN:
Step 1: Import Statement : ...
Observation 1: ...
Step 2: Check for k%2==0 and (X+Y)%2==1:
Observation 2: ...
Step 3: ... It proceeds to execute the
        `solve` function with X=5, Y=5, and
        K=5 (line 20)... Line 4 is the
        first conditional check, but X and
        Y are not both 0, so it is not
        executed. Lines 5 and 6 checks for
        negative values of X and Y, which
        are not applicable here since both
        X and Y are positive. Line 7 checks
        if X is greater than Y, which is
        not true in this case since X=Y.
        Line 8 checks if (X+Y) % k == 0,
        which is true since (5+5) % 5 == 0.
        This condition leads to the
        execution of line 8.
Observation 3: Since the specific
        conditions leading to line 8
        execution are met with our input,
        we can focus on sets ....
Step 4: Conclusion on the Applicable Code
        Coverage Set: ...
Observation 4:
FINAL CODE COVERAGE -
Set 4: Lines 1, 2, 3, 4, ...
```

Fig. 7. Illustrative Example for CRISPE's Planning

ratio of the number of correct steps over the total number of steps for the coverage prediction for that instance. *Step accuracy* for the entire dataset is such ratio but for all the instances.

Results. As shown in Table 4, CRISPE's accuracy in planning is **82%**, meaning that out of the predictions for 100 programs, it predicts the plans for approximately 82 programs with 100% accuracy in all the steps. Compared to the results in RQ1, in 27.46% of the instances, despite making the right reasoning until the last step, CRISPE incorrectly made the last decision on generating the correct FOCC. This could be due to LLM's hallucination. Moreover, the step accuracy, as detailed in Table 4, is **92.4%**, indicating that for a plan consisting of 10 steps, GPT-4 accurately performs observations, reasoning, and actions through 9 of those steps. Consensus was reached in all manual evaluations.

Comparison with CodePilot in planning. CodePilot involves generating new predictions based on previous steps, which can lead to inaccuracies building up as the reasoning progresses. In contrast, by minimizing the need for iterative loops of reasoning and verification, CRISPE reduces the risk of compounding errors. *Its elimination strategy is less iterative, focusing on direct and logical decision-making on selecting/eliminating feasible options of code coverages.*

Illustrative Example. Consider the code snippet in Fig. 7(a) with the test case input `= 5 5 5`. The code includes nested `if-else` conditions (Lines 11–13), independent loops (Lines 25-28), loops containing `if-else` branching (Lines 7–10) as well as method calls and execution, thereby showing typical challenges in code coverage prediction. As shown in Fig. 7(b), CodePilot follows a step-by-step reasoning approach to predict code coverage. The model lists the steps involved in the code execution and provides observations at each step. However, it experiences an error in Step 4 of the plan. This step involves interpreting the control flow through `if-else` branches within the `solve()` function. Here, CodePilot highlights difficulties, such as handling multiple conditions and maintaining the variable states without knowing input values. Finally, CodePilot is unable to provide a final code coverage for the given test case, claiming a lack of sufficient information.

For CRISPE, in steps 2 and 3 which involves `if-else` branching in both `main()` and `solve()`, CRISPE breaks down the control flow of the solve function by explicitly considering the conditions `(K2==1 and (X**2)==1)`. In doing so, the LLM updates the state of variables and considers both the static and dynamic flow within the `solve` function. It makes deliberate decisions based on the FOCCs, avoiding compounding errors that occur in step-by-step prediction.

RQ4. *CRISPE's semantic-guided planning achieves 82% plan accuracy and 92.4% step accuracy.*

11 Time Efficiency and Token Counts in Code Coverage Prediction (RQ5)

Procedure. The code coverage prediction time was recorded for a program and test case including that for CFG and FOCC computation under the same setting. We also counted the number of tokens for each data instance including the source code, the exemplar plan, the FOCCs, and the output coverage prediction. These time values and token counts were averaged across all data instances.

Results. Table 5 presents the results. As shown, GPT-4+CRISPE, on average, is *52.58* seconds faster than GPT-4+CodePilot. This time reduction can be attributed to CRISPE's *reasoning-based elimination strategy*, which streamlines decision-making processes and reduces computational complexity. By utilizing predefined coverage sets, CRISPE leverages existing structures, *minimizing the need for generating new reasoning paths and thereby reducing the complexity and time required for predictions*. The prediction time of LLMs can be impacted by network latency and traffic. Moreover, for an instance, the token counts for all approaches are the same for the given source code, exemplar plan, and FOCCs (776 tokens on average, not shown). For the number of output tokens, GPT-4+CRISPE produces shortest outputs (Table 5) and avoids lengthy and incorrect reasoning on incorrect paths.

Table 5. Average Prediction Time and Token Counts (RQ5)

Approach	GPT-4 +CodePilot	Claude +CodePilot	**GPT-4 +CRISPE**	**Claude +CRISPE**
# O. Tokens	514	613	467	536
Avg. Time	84.29s	104.8s	31.71s	98.9s

Our utilization of FOCCs entails direct matching with existing options, fewer computational operations, and reduced iterative re-evaluation, thereby facilitating faster decision-making and smaller output token counts. Thus, CRISPE's emphasis on reasoning-based elimination is not only effective but also more efficient. In contrast, CodePilot's iterative step-by-step reasoning approach entails multiple loops of reasoning and verification, resulting in a computational overhead. It necessitates recalculations at each step, leading to increased computational and time complexity. Comparing the time for GPT-4 and Claude, we can see that in general, the prediction time with Claude is much longer. Claude+CRISPE has more than triple the prediction time with GPT-4+CRISPE.

RQ5. *CRISPE is more time-efficient and requires smaller numbers of tokens than the baselines.*

12 Sensitivity Analysis (RQ6)

The goal of this experiment is to study the extent that CRISPE performs on incomplete code.

Procedure. We utilized the FixEval dataset [14] to select 100 complete code snippets of medium to large sizes with varying complexity based on the metrics relevant to the degree of incompleteness: (1) the number of missing `import` statements, (2) the number of method calls to the methods *without corresponding method bodies*, and (3) the number of method calls in the driver method.

To evaluate the extent that CRISPE performs on the code snippets missing `import` statements, we performed the following procedure. For each complete code snippet, we systematically removed one, two, or more `import` statements at random. We then ran the best model, GPT-4+CRISPE, to

predict code coverage on the modified code, measuring statement-coverage accuracies. To assess the impact of missing `import` statements, we further stratified the results based on *the number of API calls that were affected by those missing* `import` *statements in a driver method.*

To assess CRISPE's performance on code snippets containing method calls *without corresponding bodies*, we followed this procedure. For each complete code instance, we first constructed a static call graph of the involved methods. We then identified *sink nodes*, i.e., the callee methods that are invoked by at least another method but do not call any other methods. We also identified *sink cliques*, where a clique is a set of methods that are called by a method (inside or outside the clique) or not called at all, but do not invoke any methods outside the clique. Each such sink method and sink clique serves as a candidate for method body removal in our experiment. By removing the bodies of the sink nodes and sink cliques, we ensure that we evaluate the impact of missing method bodies without breaking the semantics of calling executions in the call graph. For example, if A calls B, which in turns calls C. We remove either the body of C or the bodies of both B and C. If we remove the body of B while retaining the bodies of A and C, we disrupt the execution flow, leading to incomplete execution semantics and potentially misleading coverage predictions.

For each complete snippet, we systematically removed one, two, or more method bodies of sink nodes and sink cliques. We then ran CRISPE on the modified code and measure statement-coverage accuracies across different numbers of missing method bodies. Similarly, we further stratified results based on the *number of method calls in each caller of the callee whose body is removed.*

Results. Table 6 shows how accuracy is affected as `import` statements are missing. The performance also degrades gracefully as more `import` statements are removed and more API calls depend on them. The performance decrease is a result of the compounded impact of missing dependencies and how they affect execution paths. Our analysis reveals the two key factors mainly contributing to this:

Table 6. Impact of Missing `import` Statements on Accuracy (RQ6). Blank cell: no instance for that case.

# of Method Calls	# of Removed `Import` Stmts			
	1	*2*	*3*	*4*
1 API call	87.84%	-	-	-
2 API calls	85.96%	84.61%	-	-
3 API calls	83.15%	79.32%	76.34%	-
4 API calls	80.36%	77.89%	72.21%	-
5 API calls	82.02%	72.76%	68.15%	64.94%
6 API calls	-	-	65.24%	-

1. *Direct Impact on API Calls (26% of Incorrect Predictions)*: With missing `import` statements, the returned values and intermediate assignments directly linked to API calls become incorrectly classified as (un-)reachable. Thus, the assignments to variables using these returned values later in the code often are incorrectly predicted as (non-)covered statements.

2. *Indirect Impact on Code Execution (68% of Incorrect Predictions)*: When `import` statements are removed, some variables fail to get initialized properly because they depend on missing APIs. These incorrect values propagate through the code, affecting branching conditions, loop execution, and dependent computations. This inherently leads to false negatives in coverage predictions when branches that should be executed are incorrectly predicted as infeasible.

As seen in Table 6, accuracy decreases as more `import` statements required for API calls are removed. This is because missing `import` statements result in missing dependencies, causing API calls to return incorrect values, leading to errors in subsequent computations and conditional evaluations.

Table 7 shows how accuracy is affected as method bodies are missing. Sink-only removals (removing only terminal methods) have a lesser impact on coverage prediction because they affect only the final execution steps, leaving intermediate control structures intact. Sink-clique removals (both terminal methods and their immediate caller) alter the execution flows (e.g., conditions, loops). As seen across the diagonal (Table 7), the steady decline from 82.02% (1 method removed, 1 method

call) to 68.34% (3 methods removed, 3 method calls) suggests that CRISPE's reasoning on FOCCs significantly limits accuracy degrading since it systematically avoids infeasible coverages.

In the cases where multiple calls to the same removed methods (more method calls than the removed methods), its accuracy also reduces gracefully. The values ending with (*) represent cases where the number of removed methods exceeds the number of method calls because that the sink methods and their immediate calling cliques are removed together. Unlike sink method removals (which only eliminate terminal methods), removing both sinks and cliques disrupts more execution logic, as entire subgraphs of execution paths are missing, leading to lower accuracy in CRISPE's coverage prediction.

Table 7. Impact of Missing Methods on Accuracy (RQ6)

# of Method Calls	# of Methods Removed		
	1 method	*2 methods*	*3 methods*
1 method call	82.02%	79.36%*	78.17%*
2 method calls	80.96%	76.61%	75.63%*
3 method calls	77.93%	72.32%	68.34%
4 method calls	73.35%	70.89%	62.21%

Importantly, CRISPE still performs better than a naïve model because it avoids hallucination towards non-existent paths when faced with missing methods. With reasonably high accuracy, CRISPE's use of FOCCs ensures that only valid coverage options are considered, improving accuracy even when code is incomplete, e.g., in the tasks involving online code snippets or code under editing.

Comparing Tables 6 and 7, we observe that the impact of missing `import` statements is slightly less negative than that of missing method bodies. This is due to the following perceived factors:

1. *LLM training on external APIs and library implementations:* LLMs like GPT-4 are highly likely to have been trained on codebases using these APIs or the API libraries with similar functionality. This *pre-trained knowledge* helps CRISPE infer missing dependencies even when some `import` statements are missing, leading to less accuracy loss than in the case of missing method bodies.

2. *More robust generalization for missing* `import` *statements to libraries:* GPT-4+CRISPE can sometimes estimate expected outputs based on its learned understanding of common or similar APIs. This allows it to partially recover missing execution paths when dependencies are missing, leading to more stable performance than when entire method bodies are removed.

3. *Semantics-guided execution planning via FOCCs:* CRISPE helps the LLM reason on the feasible coverage paths. This could partially mitigate the incorrect returned values from the API calls that miss the `import` statements when CRISPE avoids the paths that are not in the FOCC sets.

RQ6. *As the number of involved methods in a driver method increases, the missing of those methods' bodies have more negative impact on performance. The impact of missing* `import` *statements is slightly less negative than that of missing method bodies. CRISPE's FOCC-guided execution planning partially helps avoid infeasible coverage paths, leading to gracefully degrading in CRISPE's accuracy.*

13 Threats to Validity

External Validity. Our chosen benchmark might not be representative of real-world codebases. However, they have been used in the existing work for code coverage prediction, which enables a fair comparison. We also evaluated CRISPE with two LLMs. The results may not hold for other LLMs with different architectures or reasoning capabilities. However, we chose GPT-4 and Claude, which are the two state-of-the-art LLMs. Our experiments were conducted only on Python. More experiments are needed for other programming languages with different execution semantics.

Internal Validity. The results in time might be vary due to the network traffics, latency, server loads when interacting with LLMs. Thus, we also counted the tokens being sent/received from the LLMs' servers. For error detection, we used LLMs to generate error-triggering inputs. While LLMs

may not generate such inputs, they were solely used as fuzzers to generate inputs. If a generated input did not trigger an error, we simply tried a new one. The third-party tools to build FOCCs may introduce errors. However, we used well-established tools. Manual checking of planning could have human errors. Thus, we checked the generated plan in texts along with the resulting coverage.

Construct Validity. The inputs generated from the LLMs for the runtime error detection experiment might not be comprehensive. They might not cover all edge cases, thus, we may not accurately assess error detection capabilities. However, GPT-4 performs well in fuzz testing. In future, we could consider other types of fuzz testing. Varied prompts may lead to varied input distributions, potentially affecting results. To address this, we define the template with well-specified structure.

14 Related Work

CRISPE is closely related to the following research. Tufano *et al.* [26] uses GPT-4 [5], to predict code coverage, but has inherent limitations. CodeExecutor [16] is a Unixcoder-based neural network model pre-trained on a diverse set of programs to predict execution traces. CodePilot [8] is a LLM-based solution that instructs the LLM to build its own plan based on the execution semantics. Compared to CRISPE, it does not use the guidance from FOCCs to have the benefits. NExT [19] teach LLMs to inspect the execution traces (variable states of executed lines as annotations) and reason about runtime behavior through CoT rationales. With CoT, both approaches sometimes violates program semantics during predictive execution. See Section 1 for detailed comparison.

Code coverage is crucial in software testing, supporting techniques like fuzzing [24, 29] and fault localization. Studies such as Böhme *et al.* [4] and Fioraldi *et al.* [10] highlight its role in advancing fuzzing. In industry, Chen *et al.* [6] estimate coverage from execution logs, while Ivanković *et al.* [15] examine Google's practices. Gopinath et al. [12] emphasize its value in evaluating test suites.

ML has been used code execution prediction [16]. Lou *et al.* [17] enhance coverage-based fault localization using graph-based learning. Gay [11] focuses on test suite effectiveness through combined coverage criteria, while Wei *et al.* [29] question branch coverage as a reliable testing measure. Aligning with these insights, CRISPE enhances coverage prediction via LLMs.

To compute code coverage, Pavlopoulou *et al.* [20] propose removing instrumentation after data collection, tracking Java bytecode execution. Nagy *et al.* [18] apply binary instrumentation to modify code at runtime. Tikir *et al.* [25] extend DyninstAPI for native code analysis, while Chilakamarri *et al.* [7] introduce disposable instrumentation for JVM bytecode. SlipCove [21] optimizes dynamic instrumentation by de-instrumenting covered lines, improving efficiency.

15 Conclusion, Limitations, and Implications for Future Work

Error Analysis. We have analyzed the results and reported the following *categories of cases where CRISPE fails in prediction*. First, it is still challenging for CRISPE in *effectively predicting for loops*, particularly those containing nested `if-else` statements or loops. Other cases involve nested loops. To address this, future work could focus on improving its planning algorithms for traversing, stopping, and analyzing each nested, dependent control structure. An alternative solution could be the integration of external program analysis (PA) tools in a multi-agent framework to facilitate the analysis of those complex structures. Second, CRISPE incorrectly detects some runtime errors which involve different paths with the same FOCC. In these cases, it could choose an incorrect path, leading to incorrect detection. Third, in some cases, the LLM in use incorrectly evaluated the condition in an `if` statement due to hallucinations. The integration of program analysis could help in such computations. Finally, in runtime error detection, CRISPE incorrectly identified the exception statements to be covered due to error propagation from an incorrect evaluation of a condition.

Limitations. There are also rooms for improvement in other aspects. First, for GPT-4, scalability could be an issue due to a lower limited number of tokens in the input. To make GPT-4 to work

on real-world, large codebases, one could design algorithms to process source code method by method in an on-demand manner. Despite that, with the current capability, CRISPE is useful for the scenarios involving online code snippets with small to medium sizes (in which the traditional PA tools fail to detect). Second, despite that CRISPE improves over the state-of-the-art approaches in *time efficiency*, its prediction time is still slower than the time of actual execution. However, CRISPE at least provides a good solution when the actual execution is undesirable or even impossible (see Section 1). The main bottleneck is the time for sending/receiving requests from the LLMs. With the emerging of LLMs operating in a more local fashion, it would help reduce the running time. Third, the integration with existing development workflows is desirable. Seamless integration with continuous integration and deployment pipelines could make CRISPE more valuable for developers. Finally, for web programming (with declarative web languages) or system programming (with event-driven programming), we need different semantics-specific guidance with more capability than traditional CFGs to capture those execution flows and integrate into our framework.

Significance of CRISPE's contributions. CRISPE is a novel step toward advancing the LLMs' capabilities in understanding/reasoning on the dynamic program behaviors. In this work, we leverage *semantics-guided, observation-driven execution planning and dynamic reasoning to enhance the LLMs in predicting code coverage for both complete/incomplete code snippets*. We found that integrating the task-specific semantics in FOCCs could guide the LLM to better autonomously develop and execute a coverage prediction plan. Combining semantic insights and observation-driven actions, we improve LLMs' reasoning on code coverage, overcoming the infeasible path issue.

This work also has good **impacts** on software development practice and future research.

For software development practice: first, in test case prioritization, actual execution would undermine its purpose, thus is undesirable. CRISPE can rank test cases by estimating code coverage—without execution—prioritizing those that target uncovered code areas. Second, CRISPE could estimate the code coverage quality of test cases generated by an automated test generation engine, such as a fuzzer, enabling early detection of potential code coverage gaps and allowing developers to address them. Third, as shown, CRISPE can statically detect runtime errors in online code snippets (which traditional program analysis does not support well), before they are adapted into users' codebases, avoiding the risk of running buggy code. For future work, one could apply semantic-guided execution planning for *vulnerability detection* in which vulnerability knowledge can be used to guide an LLM toward the paths to discover them. Finally, CRISPE can be used as a *program assistant tool* in IDEs to predict (un)reachable code as code under editing is often incomplete.

For future research: first, from our framework, one could enhance an LLM to predict the execution traces by providing other semantic guidance obtained via external PA tools. Multi-agent framework between LLM-based agents and PA tools could be a good combination. Second, we could also apply the semantic-guided execution planning for vulnerability detection. Third, for dynamic behavior reasoning, we could use dynamic information collected during the runtime to guide the LLM in predicting or reasoning on runtime behaviors such as memory leaking, performance bottleneck, pointer analysis, taint analysis, etc. Finally, Section 8 showed that CRISPE can be integrated into fuzz testing to statically detect runtime errors for (in)complete code. An LLM-based fuzzer can be used to produce the potential error-triggered inputs. Then, for each generated input, CRISPE can predict whether that input could trigger any runtime error in the code snippet without actual execution.

16 Data Availability

All code and data is available [3].

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grant CNS-2120386.

References

[1] [n. d.]. Python Hunter. https://github.com/ionelmc/python-hunter. Accessed: 07/21/2023.

[2] 2024. *Claude.ai*. https://claude.ai

[3] 2024. *CRISPE*. https://github.com/crispe-prompt-engineering/crispe

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

[5] ChatGPT [n. d.]. OpenAI. https://openai.com/.

[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 *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. 305–316. doi: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. doi:10.1002/stvr.347

[8] Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. 2024. Planning to Guide LLM for Code Coverage Prediction. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering* (Lisbon, Portugal) *(FORGE '24)*. Association for Computing Machinery, New York, NY, USA, 24–34. doi:10.1145/3650105.3652292

[9] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. doi:10.1145/3597503.3608140

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[11] 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.

[12] 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. doi:10.1145/2568225.2568278

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.499

[14] Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. FixEval: Execution-based Evaluation of Program Fixes for Programming Problems. arXiv:2206.07796 [cs.SE] https://arxiv.org/abs/2206.07796

[15] 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. doi:10.1145/3338906.3340459

[16] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code Execution with Pre-trained Language Models. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 4984–4999. doi:10.18653/v1/2023.findings-acl.308

[17] 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. doi:10.1145/3468264.3468580

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

[19] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NExT: teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1540, 28 pages.

[20] Christina Pavlopoulou and Michal Young. 1999. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 277–284. doi:10.1145/302405.302637

[21] 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. doi:10.1145/3597926.3598128

[22] Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian T Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/a5bfc9e07964f8dddeb95fc584cd965d-Paper-round2.pdf

[23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

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

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

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

[27] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. doi:10.1145/3491101.3519665

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought prompting elicits reasoning in Large Language Models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[29] Yi Wei, Bertrand Meyer, and Manuel Oriol. 2012. *Is branch coverage a good measure of testing effectiveness?* Springer-Verlag, Berlin, Heidelberg, 194–212.

[30] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. doi:10.1145/3597503.3639121

[31] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*.

[32] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation. arXiv:2308.01240 [cs.CL] https://arxiv.org/abs/2308.01240

[33] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 316–327. doi:10.1109/ICSE.2019.00046

Received 2025-02-26; accepted 2025-04-01