Planning a Large Language Model for Static Detection of Runtime Errors in Code Snippets

***Abstract*—Large Language Models (LLMs) have been excellent in generating and reasoning about source code and the textual descriptions. They can recognize patterns, syntax, and semantics in code, making them effective in several software engineering tasks. However, they exhibit weaknesses in reasoning about the program execution. They primarily operate on static code representations, failing to capture the dynamic behavior and state changes that occur during program execution.**

In this paper, we advance the capabilities of LLMs in reasoning about program execution. We propose ORCA, a novel approach that instructs an LLM to autonomously formulate a plan to navigate through a control flow graph (CFG) for predictive execution of (in)complete code snippets. It acts as a predictive interpreter to "execute" the code. As a downstream task, we use ORCA to statically identify any runtime errors for online code snippets. Early detection of runtime errors and defects in these snippets is crucial to prevent costly fixes later in the development cycle after they were adapted into a codebase. In our novel technique, we guide the LLM to pause at the branching point, focusing on the state of the symbol tables for variables' values, thus minimizing error propagation in the LLM's computation. We also instruct the LLM not to stop at each step in its execution plan, resulting the use of only one prompt to the LLM, thus much cost-saving. Our empirical evaluation showed that ORCA is effective and improves over the state-of-the-art approaches in predicting the execution traces and in runtime error detection.

I. INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable capabilities in generating and reasoning about source code and textual descriptions. However, despite their strengths, LLMs have notable weaknesses when it comes to reasoning on program execution [1]. These models primarily operate on static code representations, as they do not effectively capture the dynamic behaviors and state changes that occur during program execution. This hinders their ability to accurately predict runtime behavior, handle computations, and manage control flows, ultimately affecting their performance in tasks that require a deep understanding of how code executes over time.

In the context of reasoning about program execution, LLMs can exhibit issues with hallucination and error propagation. Hallucination refers to the generation of incorrect or nonsensical information, leading to unrealistic or inaccurate representations of code behavior. This can result in flawed logic or erroneous predictions about program states and outcomes. Error propagation occurs when initial mistakes or inaccuracies in the model's predictions are compounded as the reasoning process continues, leading to the model's ineffectiveness in understanding the dynamic nature of program execution.

In this paper, we introduce a novel ML-based approach for predictive execution of (in)complete code. Given an (in)complete Python code snippet and its input, with a single prompt, ORCA instructs a LLM to autonomously formulate and execute a plan. With this plan, ORCA acts as a predictive interpreter to navigate through a control flow graph (CFG) and "executes" the statements within its blocks of code. By developing a planning strategy on CFG traversal, ORCA addresses the challenge of LLMs struggling to execute complex and intricate code structures, reducing the issues of hallucination and error propagation in predictive execution.

Our novel planning framework for code execution is named Observation, Reasoning, and Code-Driven, Action (ORCA). ORCA prompts the GPT model [2] to autonomously devising a plan for "executing" code by traversing the CFG. In ORCA, *actions* involve predicting execution, including predicting statement execution within a block and updating the symbol table to reflect variable values. *Observations* consist of retrieving variable values from the symbol table, which is maintained during computation as part of the plan. *Reasoning* entails evaluating conditions at the conclusion of a block and, based on observations from the symbol table, determining subsequent actions for the next block. This process involves updating the symbol table and proceeding to predictively execute statements in the next block. We introduce the concept of a *Pause* for *Reasoning* during the CFG traversal at the conditions, making the LLM focus on the observed variable values in the symbol table and make the decision on the next block. This aims to minimize value propagation error and enhance prediction accuracy. Importantly, despite being inspired by ReAct [3] planning (Reasoning+Action) for robotics and natural language processing in which the LLM is instructed to halt the execution following each cycle of reasoning and actions in a plan to observe the environment, ORCA simply requests the LLM to make a pause. This requires only prompt to the LLM, leading to more efficient than ReAct, which requires many API calls after cycles of reasoning.

To demonstrate its usefulness, we leverage ORCA during the predictive execution to statically detect runtime errors in (in)complete code snippets with a specific input, without requiring actual execution. Online forums like StackOverflow (S/O) serve as invaluable resources for developers seeking solutions to technical problems. However, copied code from S/O may harbor potential defects, runtime errors, and vulnerabilities, posing risks to the applications that adopt them [4], [5], [6], [7]. Early detection of runtime errors and defects in (in)complete code snippets from online forums offers advantages. If we identify runtime errors and bugs after integrating the code snippet, the effort invested in integrating the S/O code

```
1  # [n: 4, ans: '', total: 4, i: 0]
2  for i in range(n):
3     total += i
4     if (i + total) % 2 == 0:
5        ans = 'Even'
6        continue
7     else:
8        ans = 'Odd'
9        total = total + i
10    total = total + 1
11 total = 1 / (total - 13)
```

Fig. 1. An Example of An Exception in Python

```
1  Let us execute the code and provide the execution trace:
2  Symbol Table: n = 4, ans = '', total = 4, i = 0
3
4  Loop iteration 1:
5  i = 0
6  total += i => total = 4 + 0 = 4
7  (0 + 4) % 2 == 0 evaluates to True
8  ans = 'Even'
9  continue to next iteration
10
11 Loop iteration 2:
12 i = 1
13 total += i => total = 4 + 1 = 5
14 (1 + 5) % 2 == 0 evaluates to False <------ INCORRECT
15 ans = 'Odd'
16 total = total + i => total = 5 + 1 = 6
17 total = total + 1 => total = 6 + 1 = 7
18 continue to next iteration
19 ...
20 # (four iterations)
21
22 total = total + 1 => total = 18 + 1 = 19
23 The loop ends because all iterations for i in range(n)
       have been executed.
24
25 total = 1 / (total - 13) = 1 / (19 - 13) = 1 / 6
```

Fig. 2. Predictive Execution on the code in Fig. 1 by GPT-3.5

into existing code repositories would be wasted if the code proves to be error-prone. Moreover, determining whether the error stems from the code snippet, the existing source code, or a combination of both becomes challenging.

Analyzing online code snippets is crucial for identifying potential issues, but it is challenging due to their often incomplete nature, which reflects the assumed contextual understanding of contributors. These snippets often lack method declarations, class definitions, and import statements for external or built-in libraries, making it impossible to execute them to detect runtime errors. Completing the code and setting it up in a sandbox environment can also be time-consuming and labor-intensive. To address these challenges, we leverage ORCA for predictive execution and static runtime-error detection and localization, without the need of actual execution. Additionally, strategies like fuzzing can be used to automatically generate error-triggering inputs, which are fed into ORCA.

We conducted experiments to evaluate ORCA. Our findings indicate that it achieves an accuracy of **60.00%** and **47.85%** in detecting runtime errors at the statement level, for complete and incomplete code, respectively. It exhibits relative improvements of up to **1.44X** over the baselines. Moreover, ORCA has low false positive rates from **15.5%** and **25.1%** for complete and incomplete code. Notably, our results show that ORCA accurately predicts **38.90%** of execution traces that exactly match the actual execution traces. Moreover, **78.46%** of the prefix of a predicted trace matches correctly. It also achieves **73.26%** accuracy in tracking variable values at the end of each CFG block. In brief, we make the following contributions:

1. Predictive Execution Paradigm that predicts the execution of (in)complete code snippets to detect runtime errors.

2. A Planning Solution on CFG to guide the LLM to learn to traverse the CFG and predict the execution accordingly.

3. Empirical Evaluation to show that ORCA achieves high accuracy in both error detection and predictive execution. Code and data for ORCA is available in our project's website [8].

II. MOTIVATION

Let us use an example to elucidate the issue and motivate our proposed solution. Fig. 1 depicts a code excerpt inspired by a real-world Python code in CodeNetMut [9], with the initial input values of the variables at the beginning. This `for` loop, in actual execution, would iterate four times. The `if` statement at lines 4–9 would execute either its `true` (lines 5–6) or `false` branch (lines 8–9) in the sequence: `true`, `true`, `false`, and `true`, before concluding the loop. This code snippet would result in a runtime exception caused by "divide by zero" since the value of `total` after the loop exit is 13.

The execution path for the loop is not straightforward due to the `if` statement that makes the execution goes in different paths for different iterations and due to the occurrence of the `continue` statement at line 6, making the line 10 to be skipped.

A. Large Language Models (LLMs) and Predictive Execution

With the advances of the LLMs in understanding and generating source code, we set out to leverage LLM in predicting the execution for this code snippet and along the way, performing static detection of this runtime exception.

Let us use the phrases "predictive execution" or "predictively execute" to refer to the prediction of the execution of the code. Toward that goal, we provided the source code in Fig. 1 to GPT-3.5 [2] with a simple prompt "Can you execute this code and provide the execution trace?" The response from GPT is shown in Fig. 2. As seen, the model made an error at line 14 as it evaluated the expression `(1+5)%2==0` to `False`, leading to incorrect sequence of iterations `true`, `false`, `false`, and `false`. GPT's understanding of computation is based on patterns and associations learned from the vast amount of training data. While it can generate code and understand programming concepts to some extent, code complexity and the nuances of code constructs can lead to such inaccuracies.

A simplistic approach to address the intricacies of code execution complexity involves dividing it into segments and sequentially processing them through the LLM. However, this method proves ineffective due to several reasons: 1) the LLM lacks context from previous iterations or code segments for current computation, 2) the source code does not show the current "execution" or iteration (all iterations are "executed"

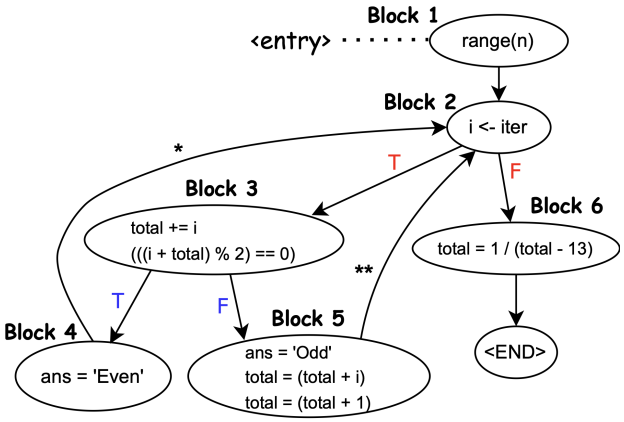

Fig. 3. Control Flow Graph for the code in Fig. 1

in the same body of the loop), and 3) breaking the source code into fragments that reflect the correct sequence of executed statements poses challenges, particularly concerning flow-altering statements (e.g., `continue`, `pass`, `return`, `exit`, etc.).

B. Key Ideas

In this work, we leverage Control Flow Graph (CFG), which represents the sequence of execution of *statements* or *code blocks* within a program and *the conditions* that decide the control flow between blocks, which are divided based on program semantics. Fig. 3 displays the CFG for the code in Fig. 1.

Definition 1 (Control Flow Graph (CFG))**.** *A control flow graph (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 flow of control between these blocks, typically based on conditions such as loops, conditional statements (e.g., `if-else`), or function calls.*

We introduce ORCA, a novel planning framework for code execution. ORCA prompts the GPT [2] to autonomously devising a plan for predictive execution by traversing the CFG and subsequently detecting the runtime errors. In designing ORCA, we have the following key ideas.

1) Teaching the LLM to 'execute' the CFG: Guiding a model to traverse and predict the execution in a CFG provides benefits to address the issues with the above naive solution, which 'executes' a loop by breaking it up into fragments. First, despite breaking the 'execution' into CFG blocks, the propagation of the values in the context of the previous block can be maintained. To reinforce this, we leverage the usage of a *symbol table* that tracks the variable values relevant to the current block. Second, the traversal through a circle in the CFG corresponds to the iterating through the corresponding loop in the source code, and the branching condition in the CFG represents the loop-entering and loop-exiting conditions. Third, the edges among the blocks automatically represent the control flows that are decided by the flow-altering statements. For example, the `continue` statement at line 6 of Fig. 1 is represented by the edge marked by '*' in Fig. 3.

2) Leveraging LLM-based Planning on CFG: The automated planning capacity of an LLM refers to its adeptness in devising a streamlined and productive method for crafting plans or sequences of actions. These plans are designed to deconstruct a complex task, enabling the achievement of a goal through a series of smaller, manageable steps [10], [11]. Researchers have introduced diverse planning frameworks tailored for LLMs, facilitating their effective handling of a multitude of tasks in various domains [12], [13], [3], [14]. The planning techniques have demonstrated efficacy in mitigating issues like hallucination and error propagation in LLMs when tackling complex tasks [3], e.g., program execution [1].

Our objective is to harness planning to guide an LLM in formulating its own plan for "executing" code by traversing the CFG and detect the runtime errors/exceptions in the process. Inspired by ReAct [3] planning technique, we aim to leverage three conceptual elements in modeling a plan, which operates in an interleaved manner: 1) *Observations*: These encompass the LLM's observations regarding the current state of the system and task. 2) *Reasoning*: This entails the cognitive processes involved in determining the next step forward within the plan. 3) *Actions*: These denote the sequence of actions to be undertaken based on the reasoning in the next step.

Within the context of our problem of predicting executions, *"actions"* ($\mathcal{A}$) entail predictive execution, involving the prediction of statement execution within a block, and updating the symbol table to reflect variable values. *"Observations"* ($\mathcal{O}$) involve retrieving variable values from the symbol table. *"Reasoning"* ($\mathcal{R}$) encompasses evaluating conditions at the conclusion of a block and, based on observations from the symbol table, determining the subsequent actions for the next block. This involves updating the symbol table and proceeding to predictively execute statements in the next block. We introduce the concept of a *Pause* for the LLM in reasoning during the CFG traversal at the condition of a block. This makes the LLM focus on the observations of the variables' values in the symbol table. We expect this to help minimize the error propagation in value computation in the LLM.

Fig. 4 partially shows the output plan, the execution and prediction of GPT after our prompt (will be explained later). As seen, it first takes the observation on the initial symbol table. Then, it starts predictive execution for Block 1, which contains the computation of `range(n)`. The reasoning at this branching condition is that it is always `true`, leading to the action of *'Go to Block 2'* and updating the symbol table. The interleaving process of observations, reasoning, and actions continue for Block 2. The action in Block 2 includes only the initialization of the iterator i. The condition is `true` leading to Block 3. The series of actions in Block 3 begin with the predictive execution of the statement `total += i`. At the branching condition, the LLM follows its plan and stops to evaluate it to `true`, leading to the reasoning of going to Block 4 and updating the symbol table. The process continues until `<end>` is encountered. As seen, the LLM was able to autonomously develop a plan to predict the execution of the loop with four iterations and finally detect the "divide-by-zero" runtime error.

```
Initial Symbol Table:
{n: (4, 'int'), total: (4, 'int'), i: (0, 'int')}

Block 1:
Statements: range(n)
- Execute range(n): [0, 1, 2, 3]

Condition: True (Always True for range(n))
- Go to Block 2

Updated Symbol Table:
{n: (4, 'int'), total: (4, 'int'), i: (0, 'int')}

Block 2:
Statements: i <- iterator
- Execute i <- iterator: i = 0

Condition: True (Always True iterator)
- Go to Block 3

Updated Symbol Table:
{n: (4, 'int'), total: (4, 'int'), i: (0, 'int')}

Block 3:
Statements: total += i, (i + total) % 2 == 0
- Execute total += i: total = 4 + 0 = 4
- Evaluate condition: (0 + 4) % 2 == 0 => 0 == 0

Condition: True
- Go to Block 4

Updated Symbol Table:
{n: (4, 'int'), total: (4, 'int'), i: (0, 'int')}
...
...
total = 1 / (total - 13) = 1 / (13 - 13)) = 1 / 0
The program crashes due to a divided by zero.
```

Fig. 4. Example on LLM Planning for Fig. 1

Another pivotal aspect of the ORCA framework for predictive code execution is its departure from instructing the LLM to halt the process after each cycle of observation, reasoning, and action, as seen in ReAct [3]. This would lead to an excessive number of API calls to prompt the LLM (one call for each block). For instance, in the scenario described, the process would pause at each block, prompting the LLM to retrieve variable values from the symbol table, which would then be written to a file. Subsequently, a new prompt would be requested for the LLM to begin a new cycle of observation (reading from that file), reasoning, and action for the next block. Instead, ORCA simply instructs the LLM to pause at the condition of a block to reduce error propagation. This streamlined approach also allows ORCA to make only one API call to prompt the LLM. ORCA lets the LLM autonomously operate the cycle of operations, reasoning, and actions without interruption. Note that ReAct [3], as being applied to robotics, needs to instruct the LLM to stop after each cycle to observe the external environment. ORCA instructs the LLM to manage the symbol table, instead of writing it to a file.

III. PROBLEM FORMULATION

A. Control Flow Graph Representation

Definition 1 (Control Flow Graph Representation)**.** *A control flow graph (CFG) can be described by a 5-element tuple: ($\mathcal{B}$, $\mathcal{S}$, $\mathcal{D}$, B_0, $\mathcal{E}$). B is a set of blocks of statements $s_1, s_2, ..., s_n$. $\mathcal{S}$ is a symbol table containing pairs of variables and their (values, types) within scopes $\{[v_1 : (l_1, t_1)], ...\}$. $\mathcal{D}$ is a set of transition functions δ: $\mathcal{B} \rightarrow \mathcal{B}$, each of which is between a pair of blocks. B_0 is the starting block, which belongs to $\mathcal{B}$. $\mathcal{E}$ is the set of ending block, which is the subset of $\mathcal{B}$.*

Each block $B \in \mathcal{B}$ contains a statement block in a sequential order without any branching statement (i.e., a statement with a branching instruction, e.g., `if`, `for`, `while`, or flow-altering statements such as `continue`, `break`, etc.). Each block might contain a condition C_B at the end of the block, which can be evaluated to either `true` (T) or `false` (F). The starting block B_0 is the starting point of the predictive execution and the ending block set $\mathcal{E}$ contains the ending ones. Before the start, the block B_0 has the corresponding symbol table $\mathcal{S}$ that is initialized with the input values and types $\{[v_1 : (l_1, t_1)], ...\}$.

As the predictive execution moves along in each block, the values & types of the variables are updated in the symbol table $\mathcal{S}$. The condition at the end of a block is evaluated based on the most updated variables' values and types at that point. Each transition function δ in $\mathcal{D}$ is defined between a pair of blocks B_i and B_j (B_i and $B_j \in \mathcal{B}$). There is exactly one transition function from one block to another with the label T and one with the label F. If a block has a condition C_B at the end, the transition function with T will be chosen if C_B is evaluated to T according to the symbol table, and the one with F will be chosen if C_B is evaluated to F. Except an ending block, if a block does not have a condition at the end, there must be exactly one transition function to the next block (without any condition). Mathematically, δ: $(B_i, T) \mapsto B_{j1}$ and δ: $(B_i, F) \mapsto B_{j2}$. The predictive execution stops either at an ending block in $\mathcal{E}$ or when an exception occurs at a block.

B. Guiding the LLM to 'execute' on the CFG via Planning

Our planning framework, ORCA, aims to guide the LLM to devise its own plan to predictively execute the CFG. We leverage the three elements to represent the following.

1) **Observation** $\mathcal{O}$: ORCA needs to "observe" the state of the symbol table $\mathcal{S}$ at the end of each block: Observation $\mathcal{S}$ = $\{[v_1 : (l_1, t_1)], ...\}$. It involves the values and types of the variables in their scopes. We could guide the LLM to observe $\mathcal{S}$ after the predictive execution of each statement. However, in our experiment, the number of tokens produced by the LLM becomes too large, limiting our model's effectiveness.

2) **Reasoning** $\mathcal{R}$: the reasoning involves the checking of the condition at the end of a block (if any) and decides on the next actions. If no condition is needed, the reasoning would involve the predictive execution actions at the next block.

3) **Action** $\mathcal{A}$: the key actions include 1) predictive execution of the statements in a block, 2) the evaluation of the condition at the end of the block, 3) making the transition to the next block based on the reasoning, and 4) stopping the predictive execution due to a runtime error or reaching the ending block.

In our implementation, while we use the prompts to GPT-3.5 [2] in the textual form, the descriptive content of our prompts can be expressed in the algorithm described in Listing 1. The content of the prompt will be explained later.

Listing 1. Algorithmic Desc. of Prompt to Guide LLM create a Plan

```
1  function PredictiveRun (Block B, SymbolTable symtab)
2    if (B ∈ E) return END
3    r = StatementsPredictiveRun(B.statements) // Action A
4    if (r == RTE) return RTE
5    else
6      Update(symtab) // Observation O and Action A
7      if (B.condition == ∅) // Reasoning (R)
8        r = PredictiveRun(δ(B,T), symtab) // Action A
9      else
10       C_B = Compute(B.condition,symtab) // Observation O and Action A
11       if (C_B == true) // Reasoning (R)
12         r = PredictiveRun(δ(B,T),symtab) // Action A
13       else
14         r = PredictiveRun(δ(B,F),symtab) // Action A
15   return r
16
17 function RuntimeErrorDetection (Code code, Input input)
18   AST ast = BuildAST (code)
19   CFG cfg = BuildCFG (ast)
20   SymbolTable symtab = initialize(input)
21   if ((RTE = PredictiveRun (cfg.B_0, symtab)) == END)
22     print ("No Runtime Error");
23   else print (Err)
```

In Listing 1, to detect runtime errors, ORCA first builds the Abstract Syntax Tree (AST) and the CFG for the code snippet (lines 18–19). AST and CFG can be built for incomplete code with preserved syntaxes and program semantics. The symbol table is initialized with the input values (line 20). ORCA will call the LLM to execute the function `PredictiveRun` on the starting block of the CFG. We will explain how we convert the CFG into a textual format to feed to the LLM later. If that function returns a runtime error, we report it.

The *pseudo-code of* `PredictiveRun` *represents the content of our prompt to guide the LLM* to develop its own plan for predictive execution. We guide it with the interleaving between observations, reasoning, and actions. We first instruct the LLM to end the predictive execution if it encounters an ending block (line 2). Otherwise, LLM should create a step to predictively execute the sequential statements in the current block B (line 3). During the "execution" of those statements, if detecting a runtime error, it should stop and report it. Otherwise, it must plan an Observation step to update and observe the symbol table (line 6). At line 7, we instruct the LLM to start the next step of reasoning on the condition of the block B. If the block does not have any condition, the LLM will proceed the predictive execution for the next block of B. If it does, we instruct the LLM to reason about the next block by first evaluating the condition (line 10) and then making decision to continue following the transition marked with T or F depending on the condition evaluation (lines 11–14).

IV. STATIC RUNTIME DETECTION WITH LLM PLANNING

To detecting runtime errors, ORCA has the following steps.

A. Building Code Representations

In ORCA, an initial step of transforming the code into a structured representation is essential. This process begins with the construction of the AST for the given code. The CFG is then built using the python-graph tool [15]. We then extract the essential information from each block, including the elements in their scopes in the symbol table, the enclosed statements, and the details concerning potential subsequent blocks.

```
1  Block 1:
2  Statement:
3     (range(n))
4  Next:
5     Go to Block 2
6
7  Block 2:
8  Statement:
9     i <- iterator
10 Next:
11    If True: Go to Block 3
12    If False: Go to Block 6
13 ...
14
15 Block 6:
16 Statement:
17    total = 1 / (total - 13)
18 Next:
19    <END>
```

Fig. 5. Textual Representation of the CFG, corresponding to Fig. 3

B. Transforming CFG to Text for LLM

In this step, the extracted CFG is transformed into textual descriptions and used in a user prompt for the LLM, highlighting the code's logical flow and decision points enabling the LLM to simulate the execution path. The textual representation of the CFG is integrated into the user prompt for API calls to the LLM. Fig. 5 shows the textual description for the CFG in Fig. 3. The portion of the prompt corresponding to the CFG includes each block's ID and statements, with conditional statements indicating the flow direction, e.g., "If True: go to Block x", or directly "Go to Block x" when no condition exists. The sequence ends with a `<END>` token (Fig. 3).

C. Prompting for LLM

Listing 1 shows our algorithm to guide the LLM in the development and execution of a plan in one single step. Given that the interaction with GPT primarily relies on prompts, we have streamlined the algorithm into a textual system prompt suitable for GPT [2]. Note that the CFG of the provided code is also presented to the LLM as a user prompt in textual format. Our system prompt is structured into sections, each corresponding to the core steps of the algorithm. This structure enables us to translate our algorithmic logic into actionable instructions for the LLM. Fig. 6 illustrates how the Listing 1's core steps are integrated into the system prompt to the LLM. Specifically, our prompt comprises the following sections:

Task and Input Details: This section informs the LLM about the task and provides the input details of the CFG. These instructions are in lines 1–10 of the system prompt in Fig. 6.

Block Traversal: This section of our prompt instructs the LLM to create a step in its plan to "execute"/"traverse" a block in the CFG including the introduction of sequential statements and conditions. This basically explains the sequential statements and conditions that the LLM will encounter in `PredictiveRun` (see Listing 1, lines, 3, 7, and 9).

```
1  Task:
2  Detect errors (runtime) by traversing the
3  Control Flow Graph (CFG) of a Python program. The
4  symbol table should be used to track variable states &
5  types and execute code within each block.
6  Input Details:
7  Block number: A unique identifier for the block.
8  Statement: Code lines.
9  Next, if True: Next block identifier if the condition
       evaluates to True.
10 Next, if False: Next block identifier if the condition
       evaluates to False.
11 Block Traversal:
12 - Begin the traversal at the first block of the CFG,
13 and document the symbol table with the initial variable
14 states and types.
15 - Each block includes two types of operations: executing
16 simple statements and evaluating conditional statements.
17 Statement Execution:
18 Observation: Carefully review the current block's
       statement and the symbol table, paying particular
       attention to the data types and values of
       variables involved.
19 Reasoning: Before executing any code, evaluate the
       statement for type compatibility and logical
       coherence, using the identified variables' types
       and values.
20 Action: If a runtime error is detected, immediately
       document the specific error type and the block
       number where it was identified. Stop the traversal
       process by adding <STOP> to the output.
21 Error Detection:
22 Precisely identify and document the errors
23 (RuntimeError), during code execution or
24 condition evaluation. This includes common runtime
25 issues like division by zero, accessing undefined
26 variables, and specifically, type errors resulting
27 from operations on incompatible data types.
28 Condition Evaluation:
29 The last line in each block is crucial as it contains a
30 condition determining the next block in the sequence.
31 To evaluate the condition, follow these steps:
32 Observation: Review the condition and get all the
       variables' names, their values, and types from the
       symbol table.
33 Reasoning: Put all the variables' values and types
       within the condition to evaluate the condition.
       Check for any type errors and logical coherence.
34 Action: Determine the condition's truth value (True or
       False) and proceed accordingly. Move to the next
       block as per the True or False path.
35
36 Before moving to the next block, update the symbol
37 table with the new variable states and types after
38 executing the current block's code and document it.
39 Continue this traversal process until the traversal
40 reaches the end block of the CFG (<END>) or anticipating
41 an error (runtime) during reasoning.
```

Fig. 6. System Prompt Guiding LLM Create/Execute a Plan for CFG traversal

Statement Execution: This part of our prompt aims to guide the LLM to observe, reason, and act during the predictive execution of sequential statements. The texts from lines 17–20 are the manifestation of the function call `Statements-PredictiveRun` at line 3 of Listing 1.

Error Detection: This part instructs the LLM to have a step in its plan to catch a runtime error if any during the "execution" of the sequential statements or the condition of a block. The portion of the prompt from lines 21–27 in Fig. 6 explains the details on the checking in lines 3–4 of Listing 1.

Condition Evaluation: ORCA's prompt instructs the LLM to have the steps in its plan when encountering a condition in a block. The description in the prompt is from line 33–39 of Fig. 6. This part instructs the LLM to follow observations, reasoning, and actions to evaluate conditions, detect potential errors, and select the subsequent block based on the condition outcome. This corresponds to the lines 7–14 in Listing 1.

Continuation Criteria: This section clarifies the stopping criteria for the plan such as reaching the end of the CFG or encountering an error. This instruction is in lines 39–41 of the prompt in Fig. 6. This corresponds to the explanation of the stopping conditions (lines 2 and 4 in Listing 1).

V. Empirical Evaluation

For evaluation, we seek to answer the following questions:

RQ1. [Runtime Error Localization Accuracy for Complete Code]. How accurately does ORCA detect and locate runtime errors for complete code?

RQ2. [Runtime Error Localization Accuracy for Incomplete Code]. How accurately does ORCA detect and locate runtime errors for incomplete code?

RQ3. [Execution Trace Prediction]. How well does ORCA predict the execution traces during runtime error detection?

RQ4. [Value Propagation Accuracy]. How effectively does ORCA update the symbol table during prediction?

RQ5. [Crash Location Profiling]. How accurately can ORCA identify the runtime errors from different types of locations?

RQ6. [Accuracy of Generated Plans for Error Detection]. How accurately does ORCA develop a plan in CFG traversal?

VI. Runtime Error Detection and Localization for Complete Code (RQ1)

A. Data Collection, Baselines, and Metrics

1) Dataset: We used the FixEval dataset [16], derived from Project CodeNet dataset [17]. FixEval is organized by problem IDs, each representing a unique programming challenge with multiple submissions with errors. The dataset comprises of 2,066 unique problems with 277,262 submissions of Python code snippets. Test cases were obtained for 800 of these problems from the Project CodeNet dataset [17]. With its input, each of these Python code snippets is executable and contains a runtime error. We used `pythonhunter` [18] to execute each code snippet to obtain execution traces, detailing the sequence of statement executions, variable state transitions, and the crash point. In 79,116 submissions, those failing execution by `python-hunter` or CFG construction by `python-graphs` [15] were excluded, leaving 7,458 buggy and 11,846 non-buggy submissions. The reasons for such failing include syntax errors, version incompatibility, scoping issues, input line inside loops and methods, etc. Since using GPT-3.5 [2] as the LLM, we pay attention to the cost from OpenAI [2]. Thus, to achieve a 95% confidence level and 5% margin of error on 7,458 submissions, we randomly sampled 374 Python code snippets. We calculated the average number of instances per problem across all submissions and randomly selected a similar proportion of instances from each problem to ensure that the sample is random and encapsulates the diversity. Moreover, we randomly selected the additional 374 bug-free instances from the FixEval dataset to form a balanced dataset of 748 instances.

TABLE I
ACCURACIES ON STATEMENT-LEVEL AND BLOCK-LEVEL FL (RQ1)

Approach	Statement-Level FL	Block-Level FL
B_1 (*GPT-3.5 w/o planning*)	24.60%	N/A
B_2 (ORCA *w/o CFG*)	33.42%	N/A
ORCA	**60.00%**	**61.5%**

TABLE II
ACCURACIES ON INSTANCE-LEVEL RUNTIME-ERROR DETECTION (RQ1)

Model	Actual	Tot	Predicted		Acc
			Buggy	Non-buggy	
B_1	**Buggy**	374	TP = 345 (92.25%)	FN = 29 (7.75%)	78.07%
	Non-buggy	374	FP = 135 (36.10%)	TN = 239 (63.90%)	
B_2	**Buggy**	374	TP = 353 (94.39%)	FN = 21 (5.61%)	78.61%
	Non-buggy	374	FP = 139 (37.17%)	TN = 235 (62.83%)	
ORCA	**Buggy**	374	TP = 256 (68.45%)	FN = 118 (31.55%)	**76.50%**
	Non-buggy	374	FP = 58 (**15.5%**)	TN = 316 (84.5%)	

2) Baselines: i. *CodeExecutor* [9] (B_0): primarily designed for execution trace prediction.

ii. GPT-3.5 (B_1) without any planning or the CFG representation: It receives only the code snippet, aiming to locate runtime errors with a detection prompt. We aim to compare with GPT's intrinsic capability in identifying runtime errors without any additional guidance or structured framework.

iii. A variant of ORCA (B_2): we used ORCA's prompting but excluded the CFG, i.e., *planning on source code*.

3) Metrics: For evaluation, we measure fault localization (FL) accuracies at the statement and block levels, and the error detection accuracies at the instance level. Statement-level accuracy is measured as the ratio between the number of instances that a model can correctly locate the errors and the total number of instances. Block-level accuracy is defined similarly. At the instance level, we use True Positive, True Negative, False Positive, False Negative, and Accuracy.

B. Empirical Results

1) Runtime Error Localization Accuracy: Table I shows the accuracies for statement-level and block-level. As seen, ORCA achieves the accuracies of **60.00%** at the statement level and **61.5%** at the block level. That is, in 60.00% of the instances, it can correctly locate the buggy statement. In 61.5% of the cases, it can correctly localize the block of sequential statements with a runtime error. ORCA has improvements of **143.90%** and **79.53%** over B_1 and B_2 in statement-level FL accuracies, respectively. Note that B_1 and B_2 work at the statement level, thus, block-level accuracy is not applicable.

Among the baselines, CodeExecutor [9] failed to detect any runtime exceptions. It provides the execution trace for a given code snippet with input until the end of execution without detecting or stopping at an actual crash point. That is, it predicts the execution beyond the crash point. This limitation may be attributed to the fact that CodeExecutor was not trained on the executions with crashes or runtime exceptions.

We observed a decrease in performance when using *GPT-3.5 without planning*, resulting in only 24.6% accuracy in statement level FL. Examining its result, we observed several reasons account for this performance. Firstly, GPT-3.5 struggles to predict execution orders for control structures like `if` statements and loops, leading to errors in variable value calculation and incorrect execution paths. ORCA addresses this issue by employing planning on the CFG, guiding the LLM to update the symbol table, and observe variable values, and types to make branching decisions. Secondly, the LLM often errors in iterator initialization, loop continuation or termination, resulting in inaccurate predictions of loop iterations. ORCA resolves this by representing loops as blocks and conditions in the CFG, simplifying planning for the LLM. Thirdly, in nested loops, LLM struggles to manage variable scopes accurately. ORCA handles scoping proficiently with a symbol table, akin to a compiler. Fourthly, the LLM fails to detect data type-mismatch errors, such as accessing a non-array variable as if it were an array. ORCA overcomes this by storing variable data types in the symbol table. Fifth, sometimes LLM "executes" both branches of an `if` statement. With planning on the CFG, ORCA guides the LLM to pause at the branching point, make observation, reasoning to go to the next block based on symbol table. Finally, GPT-3.5 exhibits hallucination and error propagation issues, which ORCA mitigates by its planning.

ORCA without CFG performs better than GPT-3.5 without planning. It achieves 33.42% FL accuracy at the statement level. Despite better than GPT-3.5 without planning, it still has key limitations. Firstly, when prompted with the same inputs as ORCA, but without CFG, the LLM struggles to reason effectively within loop structures containing nested condition statements. ORCA addresses this issue by instructing the LLM in planning on CFG traversal. Secondly, even with planning, the LLM occasionally fails to update the symbol table accurately, possibly due to hallucinations, resulting in incorrect execution paths at branching points. ORCA explicitly instructs the LLM to update the symbol table to overcome this issue. Thirdly, handling loops poses a challenge for the LLM without CFG representation, as it may fail to update loop variables properly in subsequent iterations, leading to incorrect loop entry or exit decisions. Lastly, when processing long code snippets, the LLM may compute incorrect values, propagating errors into later predictions. To mitigate this, ORCA employs a planning strategy on the CFG, breaking down longer code into smaller blocks and navigating block by block. Moreover, in some cases, it recovered from inaccuracies caused by previous blocks and accurately predict execution steps in subsequent parts where previous block values may not have an impact.

2) Runtime-Error Detection Accuracy: As seen in Table II, ORCA is slightly less accurate than the baseline (76.5 vs 78). However, it has the lowest false positive rate (15.5%). This indicates that for complete code, planning on CFG helps LLM in better localization at the finer-grained levels (statements) than detecting errors at the coarse-grained level (instances).

This is reasonable since planning on CFG, an advantage of ORCA over the baselines, is more beneficial in locating buggy statements/blocks. It has more impact on error detection for incomplete code (Section VII-B2). However, with lowest false positive rate, ORCA incorrectly flags the non-buggy code as buggy less often than the baselines, avoiding unnecessary actions. This can be atttributed to its better reasoning on statement-level execution via planning on CFG than the baselines.

VII. RUNTIME ERROR DETECTION AND LOCALIZATION FOR INCOMPLETE CODE (RQ2)

A. Data Collection, Baselines, and Procedure

To our knowledge, there is no existing runtime-error dataset of incomplete Python code snippets with the respective complete versions (to build the ground truth). Thus, we created an incomplete dataset by keeping the portion/entire method bodies and removing import statements, class and method declarations. We kept the statements in the method bodies to maintain syntax and semantics, ensuring code execution meaningful. For ground truth, we used the complete versions. Among 7,492 buggy and 11,846 non-buggy snippets, we randomly chose 374 buggy and 374 non-buggy code snippets (different from RQ1). We used the same baselines, setup, and metrics as in RQ1.

B. Empirical Results

1) Runtime Error Localization Accuracy: As seen in Table III, ORCA shows relative improvements of **231.37%** and **88.38%** over B_1 and B_2 in statement-level FL accuracies, respectively. The reasons for low accuracies in the baselines are similar to those discussed in Section VI, where B_1 struggles with the correct prediction of execution orders, variable types and value calculations, and complex control structures. B_2, while being better than B_1, still faces difficulties in handling loops, symbol tables, and nested conditions effectively.

The drop in performance for all approaches when operating on incomplete code is primarily due to the absence of import statements to the external/built-in libraries. It is non-trivial to infer the identities of the libraries. Baselines often incorrectly flag the line that uses a missing library as buggy because of the *module import error*, even though these are not the actual location of an error. Comparing Table III and Table I, ORCA's accuracy drops only *16.5%*, which is much less than the baselines, showing its effectiveness of its planning on CFG.

2) Runtime-Error Detection Accuracy: As seen in Table IV, ORCA achieves the highest accuracy (**67.51%**) and maintains the lowest false positive rate (**25.13%**). This discrepancy of the baselines stems from issues similar to those in Section VII-B1, where the baselines often misidentify lines that use the missing API libraries as buggy, even though these are not the error locations. Comparing to the results for complete code, ORCA is also affected by the missing import statements in incomplete code, leading to ineffectiveness at the usages of missing API libraries. However, ORCA's accuracy is still higher than those of the baselines (67.51% vs 56.42%). This shows that incompleteness impacts LLM's instance-level error detection, while ORCA's planning on CFG better handles code incompleteness.

TABLE III
ACCURACIES ON STATEMENT-LEVEL AND BLOCK-LEVEL FL (RQ2)

Approach	Statement-Level FL	Block-Level FL
B_1 (*GPT-3.5 w/o planning*)	14.44%	N/A
B_2 (ORCA *w/o CFG*)	25.40%	N/A
ORCA	**47.85%**	**48.66%**

TABLE IV
ACCURACIES ON INSTANCE-LEVEL RUNTIME-ERROR DETECTION FOR INCOMPLETE CODE (RQ2)

Model	Actual	Tot	Predicted		Acc
			Buggy	Non-buggy	
B_1	**Buggy**	374	TP = 365 (97.59%)	FN = 9 (2.41%)	55.08%
	Non-buggy	374	FP = 327 (87.43%)	TN = 47 (12.57%)	
B_2	**Buggy**	374	TP = 369 (98.66%)	FN = 5 (1.34%)	56.42%
	Non-buggy	374	FP = 321 (85.83%)	TN = 53 (14.17%)	
ORCA	**Buggy**	374	TP = 226 (60.43%)	FN = 146 (39.04%)	**67.51%**
	Non-buggy	374	**FP = 94** (25.13%)	**TN = 279** (74.60%)	

VIII. EXECUTION TRACE PREDICTION (RQ3)

In this study, we evaluate the accuracy of the execution traces produced by the models. We consider them at the block and the statement levels. The same baselines and procedures from RQ1 and RQ2 were used. We use the complete code dataset from RQ1 and the incomplete code dataset from RQ2.

A. Evaluation Metrics

1) Exact Match Accuracy: Exact-Match Accuracy is calculated as the ratio of cases where the predicted traces exactly match the actual traces to the total number of cases. Note that this metric remains consistent across both statement and block levels. This is because if a prediction matches all blocks in the trace, it inherently matches all statements, and vice versa.

2) Prefix Match: This metric evaluates the accuracy of the predicted trace up to the first incorrect block or statement. Suppose the actual trace has N elements (either statements or blocks), and the predicted trace has M elements. If the first n elements of the predicted trace correctly match with the actual trace, then Prefix-$\texttt{Rec} = \frac{n}{N}$ and Prefix-$\texttt{Pre} = \frac{n}{M}$. We have two separate *Prefix Match* metrics for statement and block levels.

3) Coverage: It measures the proportion of common elements (statements or blocks) between the predicted and actual traces relative to the total in the actual trace. Coverage is quantified using both *recall* and *precision*. Suppose the actual trace contains N unique elements and the predicted trace contains M unique elements. If $N \cap M$ represents the common elements between the actual and predicted traces, Cov-$\texttt{Rec} = \frac{N \cap M}{N}$ and Cov-$\texttt{Pre} = \frac{N \cap M}{M}$. We have separate *Coverage* metrics for statement and block levels.

4) Block Transition: This metric evaluates the number of accurately predicted transitions between two consecutive blocks reflecting a correct navigation. We assess this capability using *recall* and *precision* for block-to-block transitions. Suppose the actual trace has N blocks with n unique transitions,

TABLE V
ACCURACY (%) ON EXECUTION TRACES AT STATEMENT LEVEL FOR COMPLETE CODE (RQ3)

Approach	EM	Prefix Match		Coverage	
		Rec	Pre	Rec	Pre
B_0 (*CodeExecutor*)	18.72	70.33	70.43	82.28	88.37
B_1 (*GPT-3.5 w/o planning*)	16.84	52.18	75.91	63.93	80.61
B_2 (ORCA *w/o CFG*)	21.39	64.67	**82.94**	81.25	89.52
ORCA	**38.90**	**78.46**	72.65	**96.70**	**90.27**

TABLE VI
ACCURACY (%) ON EXECUTION TRACES AT STATEMENT LEVEL FOR INCOMPLETE CODE (RQ3)

Approach	EM	Prefix Match		Coverage	
		Rec	Pre	Rec	Pre
B_0 (*CodeExecutor*)	23.53	75.04	69.44	84.58	87.18
B_1 (*GPT-3.5 w/o planning*)	8.69	47.24	**87.19**	55.52	**89.35**
B_2 (ORCA *w/o CFG*)	14.57	57.18	85.02	68.87	88.90
ORCA	**36.76**	**77.45**	73.94	**95.56**	89.25

and the predicted trace has M blocks with m unique transitions. If $n \cap m$ represents the common transitions between the actual and predictions, Trans-`Rec`$=\frac{n \cap m}{n}$ and Trans-`Pre`$=\frac{n \cap m}{m}$.

B. Empirical Results

1) Statement-Level Accuracies:

a) ***Complete Code****:* Table V presents the results for predicting execution traces at the statement level for complete buggy and non-buggy code. **38.90%** of the predicted execution traces exactly match with the ground truth. With a prefix recall of **78.46%** and a prefix precision of **72.65%**, ORCA generates execution traces that match nearly 8 out of 10 statements in the correct order, with at most three statements missing in the resulting execution trace. With a coverage recall of **96.70%** and a coverage precision of **90.27%**, on average, ORCA misses only 3 out of 100 statements and correctly covers 90.27% of the statements in an execution trace. Compared to the baselines, ORCA demonstrates a relative improvement in all metrics except Prefix-Pre for statements. While the prefix of a trace is not as precise as B_1 and B_2, the number of exact-matches from ORCA is still much higher.

Upon examination of the results, we observe the following reasons for ORCA surpassing the limitations of the baselines.

Firstly, CodeExecutor [9] often fails to identify an exception in a buggy code snippet because it was not trained with runtime errors. The reported 18.72% accuracy for CodeExecutor primarily pertains to scenarios where code is non-buggy or a runtime error occurs at the last statement.

Second, GPT-3.5 without planning (B_1) exhibits a notably lower exact-match accuracy than ORCA. This underperformance can be attributed to the challenges highlighted in RQ1, including issues related to `if` and loop structures, symbol table updating, variable scopes, and value evaluation, all of which contribute to inaccuracies in the prediction of execution traces.

Thirdly, ORCA without CFG (B_2), which represents GPT-3.5 with ORCA's planning on source code, exhibits higher performance in most of metrics than B_0 and B_1. However, it achieves lower accuracies in all metrics compared to ORCA, except in prefix precision match. This underperformance is due to the same reasons as those explained in RQ1. Despite updating variable values in the symbol table and reasoning in source code, B_2 struggles with reasoning on iteration structures, nested conditions, and their combinations, all of which are effectively handled by ORCA operating on CFG. Moreover, B_2 sometimes fails to pause at the correct location in the source code and update the symbol table accurately, possibly due to hallucinations in longer code snippets. ORCA mitigates this issue by breaking down the code into smaller CFG blocks. While B_2 achieves higher prefix precision, it misses more statements than ORCA (lower prefix recall). Moreover, B_2 has a lower exact-match accuracy than ORCA (21.39% versus 38.24%). This suggests frequent misprediction of statements during the execution, resulting in lower accuracy in fault localization at the statement level than ORCA (Table I).

TABLE VII
PREDICTED EXECUTION TRACE ON BLOCK LEVEL (RQ3)

Code	EM	Prefix Match		Coverage		Block Transition	
		Rec	Pre	Rec	Pre	Rec	Pre
Complete Code	38.90	72.68	65.90	95.27	86.13	74.15	67.41
Incomplete Code	36.76	71.39	67.46	93.69	85.22	71.89	65.20

b) ***Incomplete Code****:* Table VI displays the results on statement-level execution trace prediction for incomplete, buggy and non-buggy code. ORCA achieves an exact-match accuracy of **36.76%** (i.e., **56.23%**, **323.01%**, and **152.30%** relatively higher than CodeExecutor, B_1, and B_2, respectively). CodeExecutor [9] has highest Exact-Match, Prefix-Recall, and Cov-Recall among the baselines due to its design for execution trace prediction. However, for incomplete buggy code, it produces the traces beyond the error locations. Second, for incomplete code, GPT-3.5 without planning (B_1) exhibits the lowest performance in 3/5 metrics. This is attributed to its inability to process code beyond certain points since it predicts module import errors at the API call sites in incomplete code (Section VII-B2). Third, ORCA without CFG (B_2) follows a similar pattern as B_1. Without planning on CFG, the baselines are affected much by the incomplete code (e.g., Exact-Match accuracy drops 48.40% for B_1 and 31.88% for B_2).

In contrast, the incomplete code affects ORCA less than the baselines with a drop of only **5.50%** in Exact-Match (Tables V -VI). We could attribute this to ORCA's planning strategy in the block-by-block prediction on CFG, which helps GPT deal better in smaller code blocks with only a couple of API calls, while the baselines must handle impacts from multiple ones.

2) Block-Level Accuracies:

a) ***Complete Code****:* Table VII presents the results for predicted execution traces at the block level for complete buggy and non-buggy code. Since all the baselines operate on

statements, their results are not available. Generally, ORCA's block-level accuracies surpass its statement-level accuracies, (38.90% of instances have correct execution order of blocks). Its result correctly matches the first 7 out of 10 blocks in the execution order (prefix-Rec of 72.68%), with 3–4 incorrect blocks (prefix-Pre of 65.90%). With high recall and precision in block transitions, ORCA makes correct decisions at branching points (67.41% of the cases). For buggy code, the incorrect inclusion of blocks can be attributed to incorrect traversal past the crash point. With 74.15 % recall, ORCA misses on average 2–3 block transitions in an execution trace.

*b) **Incomplete Code**:* Table VII shows the results on the block-level execution traces for incomplete buggy and non-buggy code. Notably, ORCA's block-level performance for incomplete code is slightly reduced than that for complete code (similar to the results on the statement level). However, it misses only 1/10 blocks (93.69% Cov-Rec) and correctly covers 85.22% blocks during the traversal. Recall and precision for block transitions are 71.89% and 65.20%, respectively, showing its effectiveness in handling branch decisions even in incomplete code. This high performance could be due to its block-by-block prediction on CFG (Section VIII-B1b).

IX. VALUE PROPAGATION ACCURACY (RQ4)

This study aims to assess how effectively ORCA computes the variables' values in a symbol table. We compare ORCA against CodeExecutor, which predicts the execution traces and all the variables' values along the execution. We compare the variables' values stored in the symbol table after traversing each block with the actual values obtained from ground truth. The assessment is performed only for blocks where the traversal matches the actual executed blocks. If the predicted value of a variable matches its actual value in the ground truth at that execution step, the prediction is considered correct; otherwise, it is deemed incorrect. Variable Value Accuracy is computed as the ratio between the correctly predicted variable values over the total number of them during the traversal.

TABLE VIII
VARIABLE VALUE ACCURACY (%) (RQ4)

Approach	Variable Value Accuracy
CodeExecutor [9]	23.90%
ORCA	**73.26%**

As seen in Table VIII (for complete code), ORCA accurately computes/updates the variable values in the symbol table after traversing each block more than 73.26% of the time. This can be attributed to its planning to instruct the LLM to pause to observe the symbol table at the end of each block, thus, avoiding the error propagation of the incorrect computation of values in GPT-3.5 without planning or GPT-3.5 planning on source code. ORCA also demonstrates a relative improvement of **2.06X** over CodeExecutor in variable value accuracy.

We examined the results and report the following. First, CodeExecutor [9] often did not have the correct variables' values in the loops, e.g., the loop's iterators, the values being updated from an iteration to another, the different values due to conditions, etc. ORCA instructs the LLM to execute a plan that pauses at a branching point (including the entering, bypassing, or exiting points of a loop), and observes the symbol table. Such pausing technique helps the LLM "focus" on the variables' values. Second, due to its design, CodeExecutor predicts the output values in the form of pairs of the variable name and its value. We notice cases that it produced incorrect variable names. In ORCA, we rely on the understanding capability of the LLM on source code and we use the concept of symbol table in the compiler technology to request the LLM to update the values. Third, sometimes, CodeExecutor did not produce correct computations of the statements, especially for strings.

X. CRASH LOCATION PROFILING (RQ5)

The code structures such as loops and condition statements could greatly affect ORCA's performance in predicting execution traces and in fault localization. In this RQ5, we aim to evaluate how well ORCA performs runtime error detection when the crashes occur within different types of code structures such as simple statements, conditional structures (`if`), or loop structures (`for`, `while`). We stratify the results of RQ1 according to the crash locations with respect to a different statement types and compute the corresponding accuracies.

TABLE IX
CRASH LOCATION PROFILING (RQ5)

	Crash Location		
	Simple Statement	Branch	Loop
Total Instances	151	99	123
Detected Crashes	124	58	47
Accuracy	**82.12**	**58.59**	**38.21**

As seen in Table IX, ORCA exhibits the highest accuracy in identifying runtime errors within simple statements at **82.12%**, followed by a slightly lower accuracy in branches at **58.59%**. Notably, the accuracy significantly declines to **38.21%** within loops. This is reasonable since it is more challenging to predict correct execution traces in a branching or loop structure. The reasons for such inaccuracies are reported in Section XII.

XI. ACCURACY OF GENERATED PLANS (RQ6)

ORCA follows Listing 1 via prompting to instruct the LLM to develop its plan to navigate on the CFG and detect runtime errors. This experiment aims to evaluate how effectively ORCA instructs the GPT in creating and following a plan.

For each given code snippet, when prompting the underlying LLM, GPT-3.5 [2], it generates a textual plan that guides predictive execution. To assess the accuracy of these plans, we manually scrutinized them, examining each step's observations, reasoning, and actions. Our evaluation of plan accuracy was based on the reasoning applied to each input code snippet. Notably, our dataset comprised 374 buggy code snippets, each containing up to 39 blocks during predictive execution on a CFG. Given this large volume of blocks and corresponding reasoning steps and considering much manual effort, our focus in this experiment was on the reasoning steps at the block of

TABLE X
ACCURACY OF GENERATED PLANS (RQ6)

Approach	Total Instances	Correct Planning	Accuracy
ORCA	**374**	**227**	**60.70**

TABLE XI
10 MOST COMMON RUNTIME ERRORS DETECTED BY ORCA

Runtime Error Category	Error Message
Operand Type Mismatch	unsupported operand type(s) for ** or pow(): 'str' and 'int'
NoneType Subscripting	'NoneType' object is not subscriptable
Argument Count	`replace` expected at least 2 arguments, got 1
Invalid Argument Type	ord() expected a character, but string of length 6 found
Type Conversion	not all arguments converted during string formatting
Object Not Callable	'int' object is not callable
Type Specific Operation	can't multiply sequence by non-int of type 'float'
Non Subscriptable	'int' object is not subscriptable
Non Iterable Type	argument of type 'int' is not iterable
Index Type	list indices must be integers or slices

code where the crash occurs for each snippet. This specific reasoning step was deemed crucial for error detection, as an incorrect assessment at this juncture could invalidate the entire plan. For instance, if GPT-3.5 outputs *"at line X, `replace()` expected at least 2 arguments, got 1"* at the crash block, and the executed code at that line X includes a method call to `replace` with only one argument, we consider it a correct step in crash detection decision-making. Accuracy in planning for crash detection decisions is quantified as the ratio between the number of correct instances (i.e., those with such correct steps) and the total number of instances.

As seen in Table X, we analyzed 374 instances to evaluate crash location prediction accuracy and reasoning correctness. ORCA predicted incorrect crash blocks in 144 instances, suggesting potential reasoning errors or symbol table inaccuracies, leaving 230 instances with correct crash locations. Further analysis revealed correct reasoning in 227 instances, with correct fault locations. Only 3 instances showed accurate crash identification but with incorrect reasoning. This indicates ORCA's effectiveness, with a **60.70%** accuracy rate in reasoning across all reasoning steps for runtime error detection.

During the manual examination of the runtime errors, we classified them into several categories. Table XI displays 10 most common runtime exceptions detected by ORCA.

XII. THREATS TO VALIDITY AND LIMITATIONS

Our dataset might not be representative. However, it was collected from real-world buggy Python code and has been used in prior research in bug detection. Our prompting approach is currently tested with Python code. The generalization to other languages requires further studies. We tested our planning with GPT-3.5, which could give less accuracy than GPT-4. The generalization to other LLMs requires more analysis.

We identified the following issues in ORCA. First, inaccurate evaluations of conditional statements occasionally occur due to incorrect symbol table values. Second, ORCA may fail to update iterator variables for the next iteration, resulting in the selection of an incorrect next block. This issue may arise from difficulty identifying the correct loop iterator or retrieving the accurate value. Third, while ORCA can accurately interpret the predefined method's effect, it may not consistently capture the precise data type or structure resulting from such operations. Fourth, there are instances where ORCA mispredicts a branching statement. However, if the mis-predicted computation does not affect the subsequent statements, ORCA can still detect runtime errors in later statements despite predicting an incorrect trace. Finally, we keep track of both data-types and values in the Symbol Table. However, we do not support the symbol table with objects or pointers to heap memory. In future, we will investigate the approaches in handling heap memory operations, potentially through integration with external tools or environments in a similar fashion as in Toolformer [19].

XIII. RELATED WORK

Predictive Code Execution: Transformer-based CodeExecutor [9] predicts execution trace. TRACE [20] is an execution-aware, pre-training strategy that leverages a combination of source code, executable inputs, and execution traces to pre-train UniXcoder [21]. This model is pre-trained to statically estimate value ranges and code coverage in C. While TRACED supports predicting branches in an execution path, it does not handle loops or predict entire execution traces.

LExecutor [22] executes code for arbitrary (in)complete code in an under-constrained manner. It utilizes a neural model to predict missing values, which are then injected into the execution to prevent program halting. TraceFixer [23] employs an encoder-decoder architecture for automated program repair by learning how to edit code to align with expected behavior. Although it trains with execution traces, its goal differs from ours as it uses editing code rather than predictive execution.

Fault Localization and Bug Detection: Early neural network-based FL approaches [24], [25], [26], [27], [28] predominantly rely on test coverage data. In contrast, recent deep learning-based approaches such as GRACE [29], DeepFL [30], CNNFL [31], and DeepRL4FL [32] have exhibited improved performance. Earlier learning-based FL techniques include MULTRIC [33], TrapT [28], and Fluccs [34]. Automated program repair approaches [35], [36] locate and fix bugs. FixLocator [37] can detect co-fixing locations, and TRANSFER [38] utilizes deep semantic features and transferred knowledge from open-source data to enhance FL. In comparison, these approaches require several passing/failing test cases, while ORCA works only on one test case and for incomplete code.

XIV. CONCLUSION

This paper presents ORCA, a novel ML-based approach to identify runtime errors in code snippets without execution. ORCA's planning scheme guides a LLM in auto-formulating a plan for the predictive execution of code snippets and localize the runtime errors. ORCA's planning directs the LLM to pause at branching points, focusing on the state of symbol tables for values, thereby minimizing error propagation in the LLM. Our evaluation shows ORCA is effective and outperforms the baselines in both error detection and predictive execution.

REFERENCES

[1] H. Dhulipala, A. Yadavally, and T. N. Nguyen, "Planning to guide llm for code coverage prediction," in *2024 IEEE AI Foundation Models and Software Engineering*. IEEE, 2024 (To appear).

[2] "OpenAI," https://openai.com/.

[3] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[4] H. Hong, S. Woo, and H. Lee, "Dicos: Discovering insecure code snippets from stack overflow posts by leveraging user discussions," in *Proceedings of the 37th Annual Computer Security Applications Conference*, ser. ACSAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 194–206. [Online]. Available: https://doi.org/10.1145/3485832.3488026

[5] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 121–136.

[6] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh, "An empirical study of C++ vulnerabilities in crowd-sourced code examples," *IEEE Trans. Software Eng.*, vol. 48, no. 5, pp. 1497–1514, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3023664

[7] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 560–581, 2021.

[8] "ORCA Website," https://github.com/sedoubleblinder/orca.git.

[9] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan, "Code execution with pre-trained language models," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 4984–4999. [Online]. Available: https://aclanthology.org/2023.findings-acl.308

[10] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, "On the planning abilities of large language models - a critical investigation," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=X6dEqXIsEW

[11] V. Pallagani, B. Muppasani, K. Murugesan, F. Rossi, B. Srivastava, L. Horesh, F. Fabiano, and A. Loreggia, "Understanding the capabilities of large language models for automated planning," 2023.

[12] Y. Zhuang, X. Chen, T. Yu, S. Mitra, V. Bursztyn, R. A. Rossi, S. Sarkhel, and C. Zhang, "Toolchain*: Efficient action space navigation in large language models with a* search," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=B6pQxqUcT8

[13] Z. Hu, A. Iscen, C. Sun, K.-W. Chang, Y. Sun, D. A. Ross, C. Schmid, and A. Fathi, "AVIS: Autonomous visual information seeking with large language model agent," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=7EMphtUgCI

[14] A. Prasad, A. Koller, M. Hartmann, P. Clark, A. Sabharwal, M. Bansal, and T. Khot, "Adapt: As-needed decomposition and planning with language models," 2023.

[15] "python-graphs," https://github.com/google-research/python-graphs.

[16] M. M. A. Haque, W. U. Ahmad, I. Lourentzou, and C. Brown, "Fixeval: Execution-based evaluation of program fixes for competitive programming problems," *arXiv preprint arXiv:2206.07796*, 2022.

[17] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021.

[18] "Python Hunter," https://github.com/ionelmc/python-hunter, accessed: 07/21/2023.

[19] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=Yacmpz84TH

[20] Y. Ding, B. Steenhoek, K. Pei, G. Kaiser, W. Le, and B. Ray, "Traced: Execution-aware pre-training for source code," 2023 (Accepted at ICSE'24 Early.

[21] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "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)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499

[22] B. Souza and M. Pradel, "Lexecutor: Learning-guided execution," 2023.

[23] I. Bouzenia, Y. Ding, K. Pei, B. Ray, and M. Pradel, "Tracefixer: Execution trace-driven program repair," 2023.

[24] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, 2016.

[25] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 2007, pp. 137–146.

[26] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng, and X. Chang, "Deep learning-based fault localization with contextual information," *Ieice Transactions on Information and Systems*, vol. 100, no. 12, pp. 3027–3031, 2017.

[27] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.

[28] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[29] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "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*, 2021, pp. 664–676.

[30] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 169–180.

[31] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 445–455.

[32] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE'21. IEEE, 2021.

[33] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 2014, pp. 191–200.

[34] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[35] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE'22. ACM Press, 2022.

[36] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1145/3360588

[37] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization to detect co-change fixing locations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 659–671. [Online]. Available: https://doi.org/10.1145/3540250.3549137

[38] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1169–1180.