Blended Analysis for Predictive Execution

YI LI, University of Texas at Dallas, USA
HRIDYA DHULIPALA, University of Texas at Dallas, USA
AASHISH YADAVALLY, University of Texas at Dallas, USA
XIAOKAI RONG, University of Texas at Dallas, USA
SHAOHUA WANG, Central University of Finance and Economics, China
TIEN N. NGUYEN, University of Texas at Dallas, USA

Although Large Language Models (LLMs) are highly proficient in understanding source code and descriptive texts, they have limitations in reasoning on dynamic program behaviors, such as execution trace and code coverage prediction, and runtime error prediction, which usually require actual program execution. To advance the ability of LLMs in predicting dynamic behaviors, we leverage the strengths of both approaches, Program Analysis (PA) and LLM, in building PredEx, a predictive executor for Python. Our principle is a *blended analysis* between PA and LLM to use PA to guide the LLM in predicting execution traces. We break down the task of predictive execution into smaller sub-tasks and leverage the deterministic nature when an execution order can be deterministically decided. When it is not certain, we use predictive backward slicing per variable, i.e., slicing the prior trace to only the parts that affect each variable separately breaks up the valuation prediction into significantly simpler problems. Our empirical evaluation on real-world datasets shows that PredEx achieves 31.5–47.1% relatively higher accuracy in predicting full execution traces than the state-of-the-art models. It also produces 8.6–53.7% more correct execution trace prefixes than those baselines. In predicting next executed statements, its relative improvement over the baselines is 15.7–102.3%. Finally, we show PredEx's usefulness in two tasks: static code coverage analysis and static prediction of run-time errors for (in)complete code.

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

Additional Key Words and Phrases: AI4SE, Large Language Models, Blended Analysis, Predictive Execution

ACM Reference Format:
Yi Li, Hridya Dhulipala, Aashish Yadavally, Xiaokai Rong, Shaohua Wang, and Tien N. Nguyen. 2025. Blended Analysis for Predictive Execution. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE132 (July 2025), 22 pages. https://doi.org/10.1145/3729402

1 Introduction

Large Language Models (LLMs) [6, 15] or pre-trained language models (PLMs) [11, 14] have demonstrated remarkable strengths in natural language processing tasks, including code generation, summarization, questions-answers, etc. However, despite these strengths, they struggle with reasoning about dynamic program behaviors, such as predicting execution traces [7, 14]. This limitation arises because execution traces depend on runtime contexts, variable states, and the intricate flow of

Authors' Contact Information: Yi Li, University of Texas at Dallas, Dallas, USA, yi.li@utdallas.edu; 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; Xiaokai Rong, University of Texas at Dallas, Dallas, USA, xiaokai.rong@utdallas.edu; Shaohua Wang, Central University of Finance and Economics, Bejing, China, davidshwang@ieee.org; Tien N. Nguyen, University of Texas at Dallas, Dallas, USA, Tien.N.Nguyen@utdallas.edu.

ACM 2994-970X/2025/7-ARTFSE132
https://doi.org/10.1145/3729402

control within a program that are not directly encoded in the static representations on which they are trained. They primarily learn from static code, which lack the execution relations necessary for accurate dynamic analysis. As a result, they may fail to anticipate how subtle changes in input can affect the program's behavior. This highlights the need for specialized models or hybrid approaches that can integrate the strengths of LLMs with tools better suited for dynamic analysis.

Our research seeks to enhance the ability of LLMs/PLMs to predict program execution traces at runtime without requiring actual execution, a capability we term *predictive execution* or *static code execution.* This predictive ability could significantly benefit software testing and quality control by offering insights into which parts of the code are likely to be exercised and which parts may be overlooked. It enables the analysis on dynamic behaviors without actual execution such as *dynamic slicing, dynamic dependence analysis, test case coverage analysis, runtime error detection, etc.*

Importantly, predictive execution could be instrumental in statically detecting runtime errors in source code without execution. While online forums such as StackOverflow (S/O) provide excellent sources of knowledge for developers, they potentially pose significant risks to the applications that adopt them [9, 13, 19]. Detecting early those buggy or vulnerable online code snippets is crucial to avoid several risks of integrating vulnerabilities and wasting adaptation efforts. However, due to the context of discussions, such online code snippets are often incomplete and lack of necessary `import` statements for built-in or external libraries, and the definitions of enclosing methods and classes, rendering them non-executable in their form [27]. Traditional static analyses and symbolic execution techniques [5, 10, 20] fall short in these scenarios because they rely on having a complete program, with all necessary source code and libraries. Therefore, predictive execution for incomplete code emerges as a desirable alternative. This detection task can tolerate for a degree of imprecision in static analysis of code execution, enabling the model to approximate the dynamic behavior.

The approaches to predictive execution can be classified into two categories. The first one follows the pre-trained language model (PLM) paradigm. CodeExecutor [14] requires the training data including source code, input values, and full execution traces together with the values at each execution step. As cascading errors, incorrect predictions in values lead to incorrect predictive behaviors at branches and iterations. TRACED [8] proposes pre-trained strategies on execution data, but it struggles with predicting for *conditions and iterations.* It exclusively relies on the final execution of the last line within a loop to determine the program states via variable value ranges. The second category leverages the LLMs' code understanding capability. Tufano *et al.* [22] relies on GPT-4's reasoning on source code to predict code coverage. However, due to the LLM's hallucinations and error propagation, it struggles with complex structures. CodePilot [7] uses Chain-of-Thought (CoT) planning [24] to guide an LLM in autonomously generating execution plans for code coverage. NExT [16] teaches LLMs via CoT to inspect the execution traces (variable states of executed lines as annotations). With CoT, both approaches violate program semantics during predictive execution.

In this paper, aiming to advance LLMs and PLMs in predictive execution, we propose PredEx, a blended analysis, *to combine the strengths of Program Analysis (PA) and LLMs/PLMs to predict the next executed statement and execution trace given a code snippet and its input.* In the discussion of predictive execution, we refer to both types of models, LLMs and PLMs, as LLMs for short unless we need to distinguish them in specific contexts.

In PredEx, we posit that by providing the LLM with support from PA and feeding it relevant statements, we can guide LLM in reasoning on the execution of a smaller subset of statements without executing them. We decompose the static code execution into smaller sub-tasks and allocate each task to either PA or LLM based on their respective strengths. At the coarse-grained level, we divide the predictive execution problem into predicting the next executed statement at each step. When PA can make deterministic decisions, such as determining the next executed statement within a block, we leverage PA for that to alleviate the potential inaccuracy of the LLM. However, when PA is unable to ascertain the execution flow, particularly at branching points (e.g., `if`, `while`, `for`, etc.), we enlist the LLM to handle tasks critical for the decisions at those branching points. At the fine-grained level of the task of LLM-based branching decisions, we harness PA to assist the LLM in focusing on the pertinent statements as deciding on the branching direction.

Specifically, the two sub-tasks at a branching point for the LLM include *value evaluator* and *condition evaluator*. First, the value evaluator focuses on determining the values of variables involved in decision-making at a branching point. By using backward slicing starting from those variables, we can help the LLM focus only on the relevant, important statements that influence the values of the variables. We build the backward slices and use the current predicted statements and paths in the trace to approximate the dynamic backward slice. Let us call it *predictive backward slice*. By such slices and predicted values, we enable the LLM to recognize patterns and dependencies in the code that affect the values of the current variables. Our idea is that backward slicing per variable, i.e., slicing the prior trace to only the parts that affect each variable separately breaks up the valuation prediction into significantly simpler problems. Moreover, by training the LLM via few-shot learning or fine-tuning on the external API calls, we enable it to derive the output values of such calls. Second, the condition evaluator, builds on the values obtained from the value evaluator. Given the values of the variables at a branching point, the condition evaluator can predict the outcome of the condition expression. For this evaluation, one can have different strategies. First, one can leverage the LLM as in the value evaluator. Second, one can leverage the expression evaluator in the compiler. Third, one could build his/her own probabilistic condition evaluator to estimate the probability of a condition being true. Finally, one can combine multiple strategies.

We build PredEx as the interpreter for Python, with (1) the combination of PA and the PLM CodeExecutor (i.e, `PA+CodeExecutor`), and (2) the combination of PA and GPT-4 (`PA+GPT-4`). For complete code snippets, with PA support, PredEx boosts CodeExecutor's and GPT-4's accuracies by 31.5% and 47.1% relatively in predicting full execution traces. It produces 8.6%–53.7% more correct prefixes of execution traces than CodeExecutor and GPT-4 without PA. In predicting the next executed statement, its relative improvement over the baselines is from 5.7–102.3%. For incomplete code snippets, with PA support, PredEx boosts CodeExecutor's and GPT-4's accuracies by 36% and 41.2% relatively in execution trace prediction. We also show PredEx's usefulness in two tasks: static code coverage computation and static prediction of run-time errors. PredEx (`PA+GPT-4`) is able to boost the coverage prediction accuracy of GPT-4 by 33.5% and 26.2% in statement and branch coverage accuracies. PredEx with GPT-4 (`PA+GPT-4`) also predicts run-time errors for a given input of incomplete code with a high accuracy of 89.5%. In brief, we make the following contributions:

a) [Predictive Execution]. PredEx is a Python predictive interpreter without execution that predicts the execution traces for Python code and its input values for both (in)complete code.

b) [Blended Analysis Framework]. We introduce the blended analysis approach between program analysis and LLM for predictive execution that performs better than individual approaches. Other static/dynamic analysis could be blended with an LLM in the same manner.

c) [Applications of Predictive Execution]. We showed PredEx's usefulness in static code coverage prediction and in static prediction of run-time errors for (in)complete code snippets.

d) [Empirical Evaluation]. Our results show that PredEx improves over the state-of-the-art baselines in effectiveness and downstream tasks. Our code and data is available at [17].

2 Motivation

2.1 A Motivating Example and State-of-the-Art Approaches

Let us use an example to explain the problem and motivate our solution. Fig. 1 shows a code snippet missing the `import` statement for the external library `numpy`. It could accept three inputs for `X`, `Y`, and `Z`.

```
0 // Input: X = 12;Y = 3;Z = 1
1 count = 0
2 width = 2*Z+Y
3 while X >= width:
4    count +=1
5    width +=numpy.square(Z)+Y
6 myPrint(count)
```

Execution Trace: <line><0>, <line><1>, <line><2>, <line><3>, <line><4>, <line><5>, <line><3>, <line><4>, <line><5>, <line><3>, <line><6>

Training data for the state-of-the-art approach CodeExecutor:

```
<line><0><state>X:12<sep>Y:3<sep>Z:1</state>
<line><1><state>X:12<sep>Y:3<sep>Z:1<sep>count:0</state>
<line><2><state>X:12<sep>Y:3<sep>Z:1<sep>count:0<sep>width:5</state>
<line><3><state>X:12<sep>Y:3<sep>Z:1<sep>count:0<sep>width:5</state>
<line><4><state>X:12<sep>Y:3<sep>Z:1<sep>count:1<sep>width:5</state>
<line><5><state>X:12<sep>Y:3<sep>Z:1<sep>count:1<sep>width:9</state>
<line><3><state>X:12<sep>Y:3<sep>Z:1<sep>count:1<sep>width:9</state>
<line><4><state>X:12<sep>Y:3<sep>Z:1<sep>count:2<sep>width:9</state>
<line><5><state>X:12<sep>Y:3<sep>Z:1<sep>count:2<sep>width:13</state>
<line><3><state>X:12<sep>Y:3<sep>Z:1<sep>count:2<sep>width:13</state>
<line><6><state>X:12<sep>Y:3<sep>Z:1<sep>count:2<sep>width:13
<sep>output:2</state>
```

Fig. 1. A Motivating Example of Predictive Execution on Incomplete Code (missing '`import numpy`')

Note that a regular Python interpreter for classical execution would fail to execute this code snippet. The missing of the '`import`' statement to the `numpy` library prevents the classical Python interpreters from resolving the API call `numpy.square` at line 5. This is exacerbated in the case of third-party libraries. In general, the classical execution from a traditional interpreter will fail in the cases with missing variable declarations, missing input values, resulting missing variable values. Thus, it is preferred to achieve predictive execution, i.e., to predict the full execution trace (as shown).

The state-of-the-art prediction approaches exhibit limitations when applied to this example. Firstly, the static *program analysis* and *symbolic execution* approaches require the complete code, thus, do not work on this example. Secondly, due to its trade-off made in favor of performance, the pre-trained model *TRACED* [8] relies on the final execution of the last line within the `while` loop (line 5) to finalize the program states via variable value ranges. Consequently, it *does not handle well condition and iteration statements.* Moreover, while aiming to produce execution traces, its pre-training objective focuses only on code coverage learning. Thus, valuable execution trace information, such as the execution orders between line 4 and line 5 in two consecutive iterations, is not fully and correctly leveraged, leading to inaccuracies. Finally, it employs variable coverage learning, i.e., labeling variable occurrences as covered or not. Thus, it may not effectively capture the branching behavior in the scenarios where within a loop, a variable occurrence in a `true` branch occurs in one iteration but not in another.

Another state-of-the-art approach, *CodeExecutor* [14] uses the transformer-based UniXcoder model [11]. It requires the training data including the source code, input values, and the full execution trace together with the values at each execution step (Fig. 1). *The model heavily relies on the underlying UniXcoder to learn how to transform source code along with its input into an execution trace.* For this example, since the value of `width` at the condition is changed inside the loop (line 5), CodeExecutor has error propagation from incorrect values, leading to incorrect prediction.

We attempted to use GPT-4 [6] on the example in Fig. 1. It correctly produced the execution trace. To evaluate it in a longer execution trace, we changed the input to: `X=102`, `Y=3`, and `Z=2`. The result is correct only until the 4^{th} iteration (`count=4`). This shows that *while being effective for short programs, due to hallucination and error propagation, GPT-4 is limited in predicting a complex execution.*

In brief, each individual of PA and ML approaches is not sufficient to support static code execution for incomplete code. From this example, we make the following observations, aiming to *combine the strengths of both directions* to improve predictive execution.

2.2 Observations and Key Ideas

2.2.1 Unconditional Execution. Instead of putting a burden on a LLM model to predict the entire execution trace, *we aim to predict the next statement to be executed given the current one.* Depending on the current statement, the decision on the next statement can be made by following the control flow in the program. Specifically, the next executed statement can be determined either unconditionally or conditionally based on the program's state, specifically the values of its variables. For example, the statements at lines 1–2 do not contain a condition for the next statement. Thus, the next statement is the statement 2 and then the statement 3, respectively. The next statement for the "unconditional" case is not necessarily the next statement in the code. For example, if the current statement is a method call, the next one will belong the callee method. If the current statement is either `return` or `continue`, the next one is deterministically decided based on program semantics.

Observation 1 (Execution Decision). *The decision on the next statement to be executed can be determined unconditionally or conditionally on the control flow and the program's state.*

This observation leads us to an idea to avoid the random errors in LLMs in predicting those obvious outcomes by relying on *program analysis*.

Key Idea 1 (Unconditional Execution Decision). *Via* **program analysis***, a model can follow the control flow graph (CFG) to decide the execution order for unconditional execution cases.*

2.2.2 Conditional Execution. After the statement 3 was executed the first time, the next executed statement (either the statement 4 or statement 6) depends on the evaluation of the condition (`X >= width`). At the first iteration, `X` is evaluated to 12 and `width` to 5, thus, the condition is `True`.

Observation 2 (Conditional Execution – Branching). *At a branching point (e.g.,* `for`*,* `while`*,* `if`*), the next statement to be executed depends on the condition expression, determined by two factors: the values of the relevant variables and the evaluation of the (sub-)condition(s) based on those values.*

Key Idea 2 (Value Evaluator and Condition Evaluator). *We break down the task of deciding the next executed statement at a branching point into two sub-tasks. First, we use* **a LLM to act as a *value evaluator*** *for the non-boolean sub-expressions and the variables involved in the (sub-)condition(s). Second, based on those predicted values, we use a* condition evaluator *to evaluate the sub-conditions and then the condition as* `True` *or* `False`.

As seen in Key Ideas 1–2, we leverage program analysis (PA) and LLM to help each other in deciding the execution order for the statements. Next, let us explain our next idea to leverage PA to help a LLM in the variable-value evaluation.

2.2.3 Program Slicing. Let us consider the condition expression '`X >= width`' at line 3 at the first iteration. A naive approach is to feed all the statements up to the condition and the input values, and request the LLM to compute the values of the variables `X` and `width`. In more complex programs, that could make the LLM ineffective due to the large number of statements. In contrast, if we feed each statement one-by-one and maintain the values at each step, we face a very high number of requests to the LLM, which can be cost-ineffective and time-consuming. To balance between both solutions, we could feed to the LLM only the statements that might affect the values of the variables in the condition: `X` and `width`. That is, we provide to the LLM only the *dynamic backward slice* from the variables `X` and `width`. This treatment also helps the LLM reduce the random impact of irrelevant statements to its prediction. In Fig. 1, the backward slice for `width` at this iteration includes the statement 2. The slice for `X` is empty, i.e., involving only the input value for `X`.

Note that dynamic slicing cannot be performed because PredEx operates without actual execution. Thus, we introduce a technique called *predictive backward slicing*. We first build a static program dependence graph (PDG). During the predictive execution, all the predicted nodes/statements and edges are marked. During slicing for a variable, we traverse the graph. The nodes/edges that are not marked are avoided as they cannot be a part of the predictive backward slice.

Key Idea 3 (Predictive Backward Slicing – PBS). *We model the LLM as the value evaluator for the variables involved in a condition (sub-)expression. We build the static dependency graph and prune it during predictive execution so that during backward slicing for each variable in that condition (sub-)expression, only the predictively executed statements are included in the PBS.* **Program Analysis via Predictive Backward Slicing** *helps the LLM reduce the random impact of irrelevant statements.*

Assume that the LLM predicts the values for `X=12` and `width=5` at line 3 at the first iteration. Let us consider the same condition at the second iteration. Because the execution in the first iteration proceeded inside the `for` loop, the statement 5 of the first iteration belongs to the predictive backward slice of the variable `width` at line 3 at the second iteration.

2.2.4 Slicing Optimization. To avoid the inclusion of a long PBS as the predictive execution goes on or an iteration grows, to predict the values of the variables at the iteration $n + 1$, we artificially create the assignment statements for all the involved variables with the predicted values in the previous prediction at iteration n. For example, at the first iteration of the `while` loop, we artificially create two statements at the beginning of the loop: `X = 12` and `width = 5` (assume that they are the predicted values). At the second iteration, when building the PBS for `X` and `width`, we only consider the statements back to the prior prediction point with those artificial statements. For example, we do not need to include the statements from the beginning of the program to the loop statement (i.e., lines 0–2). In this case, the statement 2 will be excluded in the PBS at this iteration.

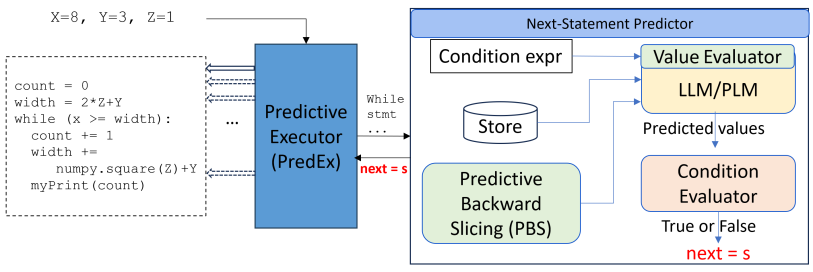
Key Idea 4 (Optimization for Predictive Backward Slicing). *As traversing the PDG to compute a Predictive Backward Slice, we stop at the artificially created assignment statements for the variables involved in the condition at the previous prediction point (e.g., the beginning of a loop). That is, slicing the prior trace to only the parts that affect each variable separately allows us to break up the valuation prediction into significantly simpler problems.*

2.2.5 Third-party APIs. To deal with external libraries, we apply few-shot learning for the LLM and fine-tuning for the PLM on the external APIs. We limit to the APIs that the LLM has seen.

Key Idea 5 (Few-shot Learning or Fine-Tuning for Third-party APIs). *By training the LLM via few-shot learning and fine-tuning the PLM on external APIs, we help it recognize the calls that affect the return values.*

3 PredEx: Predictive Execution

3.1 Framework Overview



```
count = 0
width = 2*Z+Y
while (x >= width):
  count += 1
  width +=
     numpy.square(Z)+Y
myPrint(count)
```

Fig. 2. PredEx: Framework Overview

Fig. 2 shows PredEx's overview. It accepts a Python code snippet and input values, and produces the predictive execution trace, i.e., the order of the statements that would be executed. The principle is a **blended analysis** between **program analysis** and **LLM** in which PA guides the LLM in

Listing 1. **PredEx: Predictive Execution Algorithm**

```
1  function PredictiveExecution (SourceCode p, Location entry, Input input)
2    ExTrace = $\emptyset$
3    AST ast = BuildAST (p)
4    SYMTAB symtab = BuildSymtab(ast)
5    Store $\theta$ = BuildStore(symtab, input)
6    CFG cfg = BuildCFG (ast, symtab)
7    PDG pdg = BuildPDG (ast, cfg, symtab)
8    Stmt current = Lookup (cfg, entry)
9    while ((current != END(cfg)) && (!ERROR)) //not yet ending and no error
10     ExTrace.add(current)
11     MarkPredStmtAndEdgeInPDG(current, cfg)
12     if (current is a function call (as an expression statement))
13       next = TargetFunction(cfg, current)
14     else
15       if (current is a simple_stmt) //simple statement
16         next = NextInControlFlowOrder(current, cfg)
17       else // compound statement
18         if (current in [if_stmt, for_stmt, while_stmt, match_stmt]) //branch
19           next = NextStmtPredictor(ast, cfg, pdg, ExTrace, current, $\theta$)
20         else // with_stmt, try_stmt, function_def, class_def
21           next = NextInControlFlowOrder(current, cfg)
22     current = next
23   return ExTrace
```

predictive execution. The blended analysis is expressed in two folds. First, when PredEx is certain on the execution order of the next statement s based on the current statement, it will add s into the resulting predictive execution trace. Only when it is not certain, e.g., at a branch of an `if` statement with an uncertain condition, it requests the next-statement predictor (NSP) to provide the prediction of the potential next statement. Second, even though we leverage LLM to predict the values of relevant variables and non-boolean sub-expressions in the condition expression of a statement with a branching instruction, we use **program analysis**, specifically the **predictive backward slicing**, to help the LLM pay attention only to the statements that might affect the evaluation of the condition. The backward slice for a variable might be long, thus, we design a strategy to optimize the slice by storing the predicted values of relevant variables at the latest prediction point.

In brief, with blended analysis, we break down the predictive execution into a smaller sub-tasks and leverage the deterministic nature of some sub-tasks (e.g., unconditional execution). In fact, our solution can boost both GPT-4's (LLM) and CodeExecutor's (PLM) performance (see Section 4).

3.2 Predictive Execution Algorithm

3.2.1 PredEx Algorithm. The pseudo-code for PredEx is listed in Listings 1 and 2. The input of the algorithm includes a program, the entry location (e.g., the `main` function), and the input (e.g., the values of the variables as in Fig. 1). The initialization tasks (lines 2–5) include setting up the initial execution trace set and building the abstract syntax tree (AST).

After building the AST, PredEx constructs a nested symbol table with scoping as in an interpreter or compiler (line 4). It traverses the AST to collect and organize information about *identifiers*, such as *variables, functions, and their scopes*. During the traversal, it enters a new scope when encountering a block or function declaration, pushing a new table onto the symbol table stack. Identifiers declared within the scope are added to the current table, and lookups are performed starting from

Listing 2. **Next-Statement Prediction Algorithm**

```
function NextStmtPredictor(AST $ast$, CFG $cfg$, PDG $pdg$, $ExTrace$, Stmt $stmt$, Store $\theta$)
  $C$ = ExtractConditionExpr($stmt$)
  $V = [v_1, v_2,...,v_n]$ = IdentifyRelevantVariables($C$)
  for all $v_i \in V$
      $\mathcal{S}_i$ = PredictiveBackwardSlicing($ast$, $pdg$, $ExTrace$, $v_i$, $\theta$)
      SliceOptimize ($\mathcal{S}_i$, $pdg$, $ExTrace$, $\theta$)
      $Val_{v_i}$ = ValEvaluator($C$, $\mathcal{S}_i$, $v_i$, $\theta$) // via LLM
      if ($Val_{v_i}$ = ERROR) return ERROR
  $Val = [Val_{v_1}, Val_{v_2},...,Val_{v_n}]$
  $\theta$.update($Val$)
  $\mathcal{R}$ = CondEvaluator($stmt$, $\theta$)
  If ($\mathcal{R}$ = ERROR) return ERROR
  return NextStatement($\mathcal{R}$, $cfg$)
```

the innermost scope outward to handle nested definitions. Once the traversal exits the scope, the corresponding table is popped from the stack, ensuring proper handling of the visibility of identifiers. Next, it builds the store in the similar structure as the symbol table and initializes the store with the input values (line 5). After that, PredEx constructs the control flow graph (CFG) (line 6) and static program dependence graph (PDG) (line 7). In our current implementation, we use `python-graphs` [1] for building AST, CFG, and PDG. With incomplete code, we use the ML-based tool, NeuralPDA [26], to predict CFG/PDG to supplement the missing edges by `python-graphs`.

For the code spanning multiple functions, we connect the CFGs and PDGs for individual functions together from the callers to the respective callees using the proper scope in the symbol table to enable the inter-procedural predictive execution with the proper dispatching to the right callees.

PredEx then looks up for the entry statement in the CFG (line 8). It starts iterating over the statements one-by-one until the end or a runtime error is encountered (line 9). It collects the current statement into the resulting execution trace (line 10). It marks the current statement and edge as "predictively executed" in the static PDG for later predictive backward slicing (line 11). From the current statement, it makes a prediction for the next predictively executed statement (lines 9–22).

If the current statement is an expression statement in the form of a function call, the next statement is the beginning of the target function (lines 12–13).

If the current statement is a simple statement (lines 15–16), the next one is the statement in the control-flow order (computed via `NextInControlFlowOrder()`).

The next statement in the control-flow order is not necessarily the next one in the sequential order. For example, if the current statement is `continue`, the remaining code inside the loop at the current iteration is skipped, and the next statement is the one at the loop's condition expression. The simple statements are defined as

`simple_stmt: assignment | star_expressions | return_stmt | import_stmt | raise_stmt | 'pass' | del_stmt | yield_stmt | assert_stmt | 'break' | 'continue' | global_stmt | nonlocal_stmt`

If the current one is a compound statement, PredEx needs to check if it is a branching statement (line 18, Listing 1): `if_stmt | for_stmt | while_stmt | match_stmt`

If yes, PredEx relies on the value predictor (via LLM) to predict the values of the relevant program elements (Section 3.2.2, Listing 2). If it does not have a branching condition, i.e., `with_stmt`, `try_stmt`, `function_def`, `class_def` (line 20), the next one in the control-flow order is used (line 21).

3.2.2 *Next-Statement Prediction.*
Listing 2 displays the pseudo-code for our LLM-based next-statement prediction algorithm. It accepts as input the AST, the CFG, the static PDG with marks,

the current resulting predictive execution trace, the current branching statement, and the store containing the current predicted values of variables. Initially, it extracts the condition expression C from the statement (line 2). Next, it identifies the set V of the variables involved in C (line 3). It then iterates over all the variables in V to predict their values (lines 4–8). Specifically, it calculates the predictive backward slice $\mathcal{S}_i$ from each variable v_i (line 5). If it processes all relevant variables at the same time, the resulting PBS might contain many more statements. Moreover, the number of relevant variables are often small, rendering our solution reasonable. To avoid including a long backward slice, we perform an optimization step (line 6 via `SliceOptimize`) (Section 3.2.3).

After obtaining the slice for the variable v_i, we feed into the value evaluator the following inputs (line 7): the condition C, the slice $\mathcal{S}_i$, the variable v_i, and the current store θ with the updated values. PredEx creates a request to send to the LLM to predict the value Val_{v_i} of the variable v_i. Details on *Value Evaluator via LLM* are in Section 3.2.4. In the evaluation of a variable, if the LLM predicts a run-time error, a special code (`ERROR`) is returned. We then update the store θ with the newly predicted values of the variables v_is (lines 9–10). This process is repeated for all variables v_is.

Condition Evaluator. With the predicted values in Val from the value evaluator, PredEx is ready to examine the stored values in θ to evaluate the condition expression C in the current statement $stmt$ (line 11). For this condition evaluation, one could have different strategies. First, one can leverage the LLM as in the value evaluator (Section 3.2.4). Second, another strategy can leverage Python expression evaluator (we currently choose this). Third, one could build a probabilistic condition evaluator. If the condition contains the API calls to third-party libraries, LLM would benefit from the few-shot learning or fine-tuning from training instances with those APIs (Section 3.2.4). With such strategy, the process of condition expression evaluation is in a similar bottom-up fashion as the evaluation for an expression represented as a sub-tree in an AST in a modern programming language. The key difference is as follows. If a node as a sub-expression represents a variable or a function call, PredEx will invoke the LLM-based value evaluator (Section 3.2.4). When a sub-expression is neither a variable nor a function call, if it is of a boolean type, PredEx will use the condition evaluator to continue the process. If it is not of the boolean type, the LLM-based value evaluator or Python expression evaluator is used. Let us consider, x + `numpy.square`(y) > 9. It is evaluated from the bottom-up fashion of the corresponding AST subtree. The LLM-based value evaluator is used for evaluating the sub-expression x, the sub-expression y, and then `numpy.square`(y) because they contain either a variable or function call. Next, x + `numpy.square(`y`)` is evaluated by the Python expression evaluator or LLM-based value evaluator from the previously computed values. Finally, the condition evaluator is used for the entire condition from the previously computed values.

With the result $\mathcal{R}$ from the condition evaluator, PredEx decides the next statement based on the current statement in the CFG. For example, if the condition of an `if` statement is predicted as `True`, the `True` branch will be chosen. PredEx also reports runtime errors in the condition if any (line 12).

3.2.3 Predictive Backward Slicing and Optimization. Our predictive backward slicing algorithm is adapted from the dynamic backward slicing algorithm by Agrawal and Horgan [3]. The difference is that our PBS algorithm keeps tracks of the *predictive* executed path and statements in the static PDG, rather than the *actual* executed path and statements.

The algorithm has two phases. In the first phase, the algorithm gradually marks the edges/nodes in the static PDG as predictively executed. Specifically, we build the static PDG. Initially, all the nodes (statements) and edges (program dependencies) are not marked (i.e., not predictively executed). Along the predictive execution from the beginning of the program, the edges corresponding to the new dependencies that occurred are changed to be "marked" (line 9, Listing 1).

In the second phase, i.e., slicing phase, from the criterion, the PDG is traversed backward following different types of relations among variables and references including

```
0: X = 8
1: Y = 3
2: Z = 1
3: width = 2 * Z + Y
```

(a) PBS as Input to LLM/PLM

```
<line><0><state>X:12</state>
<line><1><state>X:12<sep>Y:3</state>
<line><2><state>X:12<sep>Y:3<sep>Z:1</state>
<line><3><state>X:12<sep>Y:3<sep>Z:1<sep>width:5</state>
```

(b) Output from the PLM CodeExecutor

```
width = 5
```

(c) Output from GPT

Fig. 3. Input and Output for Value Evaluator via LLM/PLM

1) `def-use` (def. to ref.): a definition d and a reference r of a variable v have a `def-use` relation if there exists a control flow from the statement containing d to the statement containing r without any intervening redefinition of v.

2) `info-flow` (ref. to def.): for a statement S, a reference r of variable v_1 has an info-flow relation with a definition d of variable v_2 if the value of v_1 on entry to S may affect v_2's value on exit from S.

During the traversal on PDG, we traverse only along the marked edges (i.e., predictively executed statements), and the nodes (statements) reached are marked. The final set of all marked statements provides the resulting backward slice (line 6, Listing 2). This helps PredEx include only *the statements that were predictively executed and came from the predictively executed path.*

Optimization (line 6, Listing 2). To avoid including unnecessarily long backward slice all the way to the beginning, we perform an optimization *to keep only the needed portion of the slice by slicing only to the prediction point that has the recorded value for the variable under slicing*. In Fig. 4, when the condition of the `while` statement is visited at the second time, we keep the backward slice for `width` at line 5 backward to only the immediately prior iteration where we recorded the previously predicted values for `X` and `width` (Section 3.3). Specifically, the optimization step is as follows.

1) If the branching statement is the first one, the slice goes all the way to the beginning.

2) If the variable v at the current criterion does not appear in the list of variables whose values were predicted and recorded in the previous branching (prediction) point, we continue looking for v in the further prediction point. If found, we keep only the portion of the backward slice from v to that prediction point. Otherwise, we must use the entire backward slice from the beginning.

3) In the case that we found a previous prediction point containing the predicted value for v, we artificially create a statement $v = V$ (V is the predicted and recorded value for v at the previous prediction point), and add it as the first statement of the optimized predictive backward slice.

4) If the entire PBS must be used, we create an artificial statement for each input, e.g., $v_i = V_i$, as V_i is the input value for v_i. We add those statements to the beginning of the PBS.

3.2.4 *Value Evaluator.*

It is not trivial to analyze the complex code directly using an LLM. A reason is that predicting variable values at specific statements is complicated by numerous irrelevant statements that do not affect the variables' values. Additionally, the LLMs often have input length limitations. To maintain computational accuracy and efficiency for LLM, we feed to it the shorter code consisting of the statements in the PBS. In PredEx, we choose CodeExecutor [14] (with UniXcoder [11] architecture) as the underlying PLM and combine it with PA in PredEx, taking advantage of its pre-training on execution traces and variables' values. One can use *any underlying LLM*, e.g., GPT-4 [6] (Section 10) with other interface mechanisms, e.g., prompts for value evaluation. PredEx feeds the PBS into the LLM/PLM in the described process. Specifically, it combines the pieces of information including the values of relevant variables in the store θ, the PBS, and the artificial statements into a code fragment, and then feed it to the LLM/PLM. In Fig. 4, in the case of predicting the value for variable `width` at line 3 for the first iteration, the input to the LLM/PLM (i.e., the PBS), is shown in Fig. 3a. The PLM uses this PBS to produce an execution trace prediction for the backward slicing statements along with the estimated values for each variable in each statement

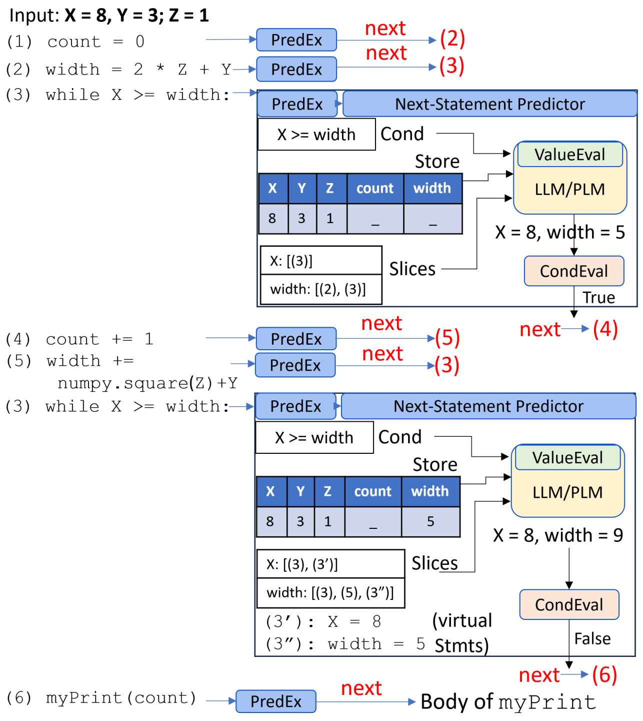


Fig. 4. PredEx: An Illustrating Example

as in Fig. 3b. From the output, we select only the values for the variables involving in the condition expression, e.g., width=5. For the GPT-4, the output is simply the variables' values (Fig. 3c).

With the nested scopes in the store θ, we can store and retrieve the proper values of the involved variables in their scopes. For complex values such as custom objects and lists, we use the pickle module, which is a built-in Python library used for serializing and deserializing Python objects. Serialization converts Python objects into a binary format that can be saved to a file (pickle.dump). This mechanism ensures that the structural and attribute data of the objects are preserved, allowing for their accurate reconstruction. This process is particularly advantageous for PredEx that needs to persist the values/states of the objects. We then de-serialize them to reconstruct the original objects with their original attributes and states from the binary data file (pickle.load). This mechanism can support any complex data structures (e.g., lists, dictionaries) or user-defined classes.

To support third-party libraries, we employ fine-tuning and few-shot learning on a small dataset to enhance PLM (e.g., CodeExecutor) and LLM (e.g., GPT-4). The dataset includes the input and output pairs for each API (similar to the ones in Fig. 3). The APIs must be known a priori for PredEx to perform predictive execution. To build the fine-tuning dataset for PLMs, we selected the API methods of the libraries of interest and diversified the inputs to each API to produce the corresponding outputs via API calls. In our experiment, we used a number of test cases for each API in the fine-tuning dataset. For GPT-4, we used few-shot learning for third-party or built-in APIs. GPT-4 is expected to generalize and predict the output for the unseen combinations of inputs.

3.3 Illustrating Example

Fig. 4 illustrates how PredEx works on our motivating example. For brevity, we use the new input X=8, Y=3, and Z=1. Assume that, PredEx starts with the assignment at line (1). Because line (1) is a simple statement, PredEx decides the statement (2) to be the next one. With the same reason, PredEx decides the statement (3) as the one after that.

At line 3, encountering a `while` statement, PredEx passes the information on to the next-statement predictor. This LLM-based value evaluator requires three essential pieces of information as input. First, PredEx extracts the condition expression `X >= width`. Second, the current state of the value store θ includes the initial values for the variables `X=8`, `Y=3`, and `Z=1`, and the undefined values for `count` and `width` because those are the values from the latest prediction point (also referred to as *check point*). If there is no prediction point yet, the latest check point is at the beginning. Third, it computes two PBSes for the two variables involving in the condition `X>=width`. The slice for `X` includes only the statement at line (3), while that for `width` includes two statements at lines (2) and (3).

From the three pieces of information, the LLM-based value evaluator predicts the values of `X=8` and `width=5`. After the store is updated, the condition evaluator (via Python expression evaluator) will use it to evaluate `X>=width` and return $True$. To reduce the number of requests to the LLM to update the values for all variables, we only update the values of the variables involving in the condition. For example, the value of the variable `count` is not updated. The next prediction for the second iteration will use the updated values in the store θ. Because the condition is predicted as $True$, the next-statement predictor will return the statement (4) as the next predictively executed one. Because the statement (4) is a simple statement, PredEx decides the statement (5) as the next one. With the same reason, it moves on to the statement (3) after that.

At the second iteration for the `while` statement at line 3, PredEx also utilizes the next-statement evaluator. The condition is the same: `X>=width`. However, the store θ is updated with the latest predicted values: `X=8`, `Y=3`, `Z=1`, `count=_`, and `width=5`. To compute the backward slices for `X` and `width`, PredEx performs differently from the last time. Because in the previous prediction point, `X` was evaluated to 8 and `width` was evaluated to 5, we create two virtual statements: 1) statement (3'): `X=8`, and 2) statement (3"): `width=5` at the beginning of the loop (right after the statement (3)). The predictive backward slice for `X` is computed as containing the statement (3) and the new one (3'). Due to the new virtual statement with the updated value of `width`, we only need to include into the predictive backward slice for `width` the statement (3), (5), and (3"). We do not need all the statements from the beginning up to that point. That is, we do not include the statement (2) into the Predictive Backward Slice (PBS) for `width` at this iteration.

With the condition `X>=width`, the store θ, and the above slices, the LLM-based value evaluator predicts the values `X=8` and `width=9`. Note that for statement (5), to compute the function `square` from the `numpy` library, PredEx relies on the fine-tuning of CodeExecutor on `square` or on the pre-training knowledge of GPT-4 on function `square` of `numpy`. Therefore, at this iteration, the condition evaluator returns $False$, and the statement (6) is next, i.e., the loop ends.

The statement (6) is an expression statement, which is an expression in the form of a function call to `myPrint(...)`. PredEx will move on to predict for the body of the `myPrint` function. The process continues until the end node in the CFG or `ERROR` occurs (see Section 10 for error scenarios).

4 Empirical Evaluation

To evaluate PredEx, we aim to answer the following research questions:
RQ1. Execution Trace Prediction on (In)complete Code. How well do PredEx (`=PA+GPT-4`) and PredEx* (`=PA+CodeExecutor`) boost the accuracies of the baselines: GPT-4 [6] and CodeExecutor [14]?
RQ2. Performance Analysis on Different Statement Types. How well does PredEx perform on different statement types?
RQ3. Impact of Predictive Backward Slicing. How does PBS in PredEx affect its performance?
RQ4. Code Coverage Analysis. How well does PredEx perform code coverage prediction?
RQ5. Scalability. How well does PredEx perform in predictive execution for larger programs?

RQ6. [Usefulness] Runtime Error Prediction. How well does PredEx perform in static detection of runtime errors with a given input for an *incomplete* code snippet?

5 Execution Trace Prediction on (In)complete Code Snippets (RQ1)

Datasets. We performed our empirical evaluation on the CodeNetMut dataset, which was used in the state-of-the-art (SOTA) execution trace prediction approach CodeExecutor [14]. It contains 19K Python code snippets for testing with their source code, inputs, and the respective execution traces. Each data point comprises Python code snippets and the execution traces for different inputs.

In total, we have 19,541 complete code snippets in CodeNetMut. Regarding *code complexity*, the numbers of lines of code are 1–146. There are 13,254 code snippets with 1-5 *maximum nesting depths*, and 228 of them have 5-10 maximum nesting depths. Regarding *decision points* including `if`, `for`, `while`, etc. 11,762 code snippets have 1–5 decision points; 1,698 have 5-10; 209 have 10-15; 48 have 15-20; 12 have 20-50; and 3 code snippets have 50-100 decision points.

To evaluate PredEx on incomplete code, we chose to sample and make the code snippets in CodeNetMut incomplete/uncompilable by removing the `import` statements to built-in/external libraries and removing enclosing methods/classes. The rationale for choosing this dataset and procedure has three folds. First, this dataset was used in the SOTA CodeExecutor [14]. Second, to the best of our knowledge, there is no available dataset of incomplete code with the corresponding complete code versions. We need the complete code versions because we need to actually execute them to collect the ground-truth execution traces. Thus, we chose the complete code snippets and made them incomplete. Finally, developers often perform the adaptation for online code snippets by adding the `import` statements and enclosing methods/classes [27]. We did not remove other parts of the code, otherwise, that would break its syntax/semantics, rendering the prediction execution meaningless.

After the removal, we have *2,975 incomplete code snippets*: 2,887 snippets have 1-5 `import` *statements*; 44 have 5-10; 41 have 10-15; and 3 have 15-20 `import` statements. Regarding *built-in function calls*: 2,155 snippets have 1-5 calls; 75 have 5-10 calls; and 11 have 10-15 calls. For *external library calls*: 4,374 snippets have 1-5 calls; 294 have 5-10 calls; 9 have 10-15 calls, and 3 have 15-20 calls. Finally, in our experiment, we have *16,566 complete code snippets and 2,975 incomplete ones.*

Baselines and Procedure. We compared PredEx* (=`PA+CodeExecutor`) and PredEx (=`PA+GPT-4`) with 1) CodeExecutor [14] (no PA), and 2) GPT-4 [6] (no PA) (used in Tufano *et al.* [22]). We did not compare with the concrete program execution because we focus on the scenarios that the execution is undesirable or impossible (Section 1). For CodeExecutor, the code snippets with the inputs being encoded as the assignments at the start were used as input. For GPT-4, we provided it with a code snippet (having its input as the assignments and with the line numbers) and a prompt directing it to produce the execution trace. For each API of the third-party libraries, we mined additional 30-40 examples from GitHub for fine-tuning in CodeExecutor and few-shot learning in GPT-4.

When handling multiple methods, CodeExecutor segments the source code into blocks, predicting for each one sequentially. The final trace results from linking these individual traces. However, for larger programs, CodeExecutor and GPT had poor performance as they faced the issue of limited input sizes. This *issue do not occur with PredEx since it feeds in a predictive backward slice (PBS) with a limited length at any given time.* At most, an PBS equals a method's length, but most often, it is considerably shorter. However, for a comparison, we applied the same method to CodeExecutor and GPT, feeding a model one method at a time based on control flow graph.

Evaluation Metrics. We used following metrics:

(1) *Full Execution Trace Prediction Accuracy* (**ETA-F**): When a model predicts the full execution trace correctly for an input (exact order and line numbers for the entire trace), we consider the result as correct. Otherwise, we consider it as incorrect. Then, $ETA\text{-}F = \frac{\#ofCorrectPredictions}{\#ofTotalPredictions}$.

Table 1. Execution Trace Prediction on (In)complete Code. ETA-F: full, ETA-P: prefix, ETA-S: statement (RQ1)

(a) Complete Code	ETA-F	ETA-P	ETA-S
CodeExecutor [14]	35.2%	82.4%	68.8%
Tufano [22] (GPT-4)	36.7%	61.5%	43.2%
PredEx* (=PA+CodeExecutor)	46.3%	89.5%	79.6%
PredEx (=PA+GPT-4)	**54.0%**	**94.5%**	**87.4%**

(b) Incomplete Code	ETA-F	ETA-P	ETA-S
CodeExecutor [14]	33.4%	80.8%	67.2%
Tufano [22](GPT-4)	35.2%	60.2%	41.9%
PredEx* (=PA+CodeExecutor)	42.1%	85.3%	76.3%
PredEx (=PA+GPT-4)	**49.7%**	**89.6%**	**82.3%**

(2) *Prefix of Execution Trace Prediction Accuracy* (**ETA-P**): This measures the correct *prefix* of the output trace. Assume that there are N statements/steps in an actual execution trace T for an input. We counted the number n of the predicted steps in the corresponding predicted execution trace T_p before the first erroneously predicted statement in T_p. $ETA\text{-}P$ is computed as $ETA\text{-}P = \frac{n}{N}$, i.e., the ratio of the overlapping prefix (partial trace) of T_p over the full trace T.

(3) *Next Executing Statement Prediction Accuracy* (**ETA-S**): On an predicted trace T, for each step/statement, if a model predicted correctly *the next executed statement*, we consider it as correct. Otherwise, it is incorrect. $ETA\text{-}S$ is defined as $\frac{\#ofCorrectSteps}{\#ofStepsInT}$.

Empirical Results. For complete code, as seen in Table 1(a), the results for PredEx and the baselines are slightly higher, which is reasonable. Importantly, *our blended analysis (BA) relatively improves CodeExecutor (w/o BA) by 31.5% and improves GPT-4 (w/o BA) by 47.1% in ETA-Fs.*

As seen in Table 1(b), as predicting execution traces for incomplete code, PredEx* and PredEx yield ETA-F scores of **42.1%** and **49.7%**. This shows that with blended analysis, PredEx* and PredEx improve the performance of CodeExecutor and GPT-4 relatively *26.0%* and *41.2%*. Considering both correct/incorrect predictions, the ETA-P scores for PredEx* and PredEx could reach as high as **89.6%**. That is, on average, *almost 9/10 of the execution steps from the prefix of a predicted execution trace are correct.* For the statement level analysis, when predicting the next executed statement, PredEx* and PredEx are successful in **76.3%** and **82.3%** of the cases.

PredEx's and PredEx*'s relatively higher ETA-S scores than the baselines, i.e., higher accuracy in next-statement prediction, could be attributed to the help of *blended analysis* for sequential statements and the better support for the LLM with the predictive backward slicing, thus improving the decisions at the branching statements. This leads to PredEx's supporting better for branches and loops. Section 6 will show the detailed results for different statement types.

PredEx* and PredEx are able to boost CodeExecutor's and GPT-4's accuracies in the following (from now on, we use PredEx to refer to both variants for short unless we need to distinguish them):

(1) When there are many *sequential ordered statements or deterministic-ordered statements* (e.g., function call, `return`, `continue`, etc.), CodeExecutor exhibits a low, albeit non-zero, possibility to predict the next executed statement incorrectly. GPT-4 produces incorrect results in predicting non-sequential statements such as `continue`, `return`, `exit`, `break`, etc. This leads to the lowest ETA-S of GPT-4 (41.9%). Contrarily, with the PA decision on sequential statements, PredEx* and PredEx boost CodeExecutor and GPT-4 in correct predictions, respectively.

(2) When dealing with a *branching* statement (e.g., `if`, `for`, and `while`), if a high number of statements precede it, CodeExecutor and GPT-4 struggled with accurately predicting the values in that statement. This is primarily because excessive information can make it challenging for a LLM to select pertinent details. In contrast, PredEx slices the prior trace to only the parts that affect each variable separately in the condition. This breaks up the valuation prediction into much simpler problems by reducing the number of irrelevant statements (Section 7).

(3) For a code snippet with less complex or shorter execution traces, PredEx typically generates better results compared to more intricate ones. See Section 6 for the results in complex statements.

(4) When processing *loops*, CodeExecutor and GPT-4 frequently struggle to accurately predict the number of iterations. However, PredEx often fares better, as it considers only the relevant backward slice between the current and the previous branching statements at the beginning of the loop, rather than including all statements from the beginning.

(5) For incomplete code, with pre-training knowledge, our model is able to handle popular built-in and external API calls without `import` statements. For less popular APIs, fine-tuning and few-shot learning allow PredEx* and PredEx to adapt quickly with examples. Moreover, by breaking down long execution prediction into smaller program slices, the LLM only needs to predict the output of *fewer individual API method calls*, rather than multiple of API calls in a single prompt.

(6) For some code snippets missing the input values, leading to undefined variable values, by running LLMs on per-variable slices for variables in conditions, PredEx can still evaluate the conditions not depending on the missing values. Classical interpreters may crash or halt unexpectedly.

RQ1. *With blended analysis (BA), PredEx* (BA+CodeExecutor) and PredEx (BA+GPT-4) can boost CodeExecutor's and GPT-4's performance in predictive execution for both (in)complete code snippets.*

6 Performance Analysis on Different Statement Types (RQ2)

Procedure. We separately calculated different metrics for each statement type from the best result (of PredEx) in RQ1. We categorize the branch statements based on their types, e.g., `if`, `while`, `for`. We computed the accuracy for the combination of both complete and incomplete code snippets.

Table 2. Accuracy on Statement Types for (In)complete Code (ETA-F: full, ETA-P: prefix, ETA-S: stmt) (RQ2)

	GPT-4			CodeExecutor			**PredEx**		
Statement Type	ETA-S	ETA-F	ETA-P	ETA-S	ETA-F	ETA-P	ETA-S	ETA-F	ETA-P
`If`	53.5	53.5	53.5	78.7	78.7	78.7	**85.3**	**85.3**	**85.3**
`For`	39.8	43.9	38.4	53.8	27.8	78.3	**59.2**	**48.1**	**83.6**
`While`	46.8	36.1	45.1	47.2	22.8	82.6	**63.4**	**43.3**	**90.4**

Empirical Results. As seen in Table 2, for the `if` statements, PredEx achieves an ETA-S score of **85.3%** (*85.3% of the branching decisions are correct*, Table 2). This can be helpful for branch coverage estimation (Section 8). Our blended analysis boosts CodeExecutor's and GPT-4's performance on `if` statement execution prediction with *8.4%* and *59.4%*, respectively. Since the `if` statements require only one-step prediction, the ETA-F and ETA-P scores coincide with the ETA-S score in this particular scenario. Note that there is no `match` statement in our dataset.

For the `for` and `while` statements, ETA-F and ETA-P are considered as the accuracy on the entire execution trace *within* a `for`/`while` statement. As seen in Table 2, with the ETA-F scores for the `for` and `while` statements, the estimation for the entire execution traces with the correct numbers of iterations is still challenging to all three approaches. However, PredEx still manages to deliver performance that, while lower, remains within a promising range. Moreover, PredEx performs well in predicting the prefix of a loop: on average, **83.6%** and **90.4%** of the entire `for` and `while` loop's execution from the start of the loop are correct, respectively. That is, for initial iterations, it performs effectively, but as more iterations continue, it becomes inaccurate. For ETA-S scores, PredEx correctly predicts 59.2% and 63.4% of the next executed statements within a `for` and `while` loop, respectively. This is lower than ETA-S (85.3%) for `if` statements as expected since in addition to the `if` branches in a loop, a model must predict the conditions to enter, continue or exit the loop.

RQ2. *PredEx enhances CodeExecutor and GPT-4 in branching and loop decision predictions.*

7 Impact of Predictive Backward Slicing (RQ3)

Procedure. To evaluate the impact of predictive backward slicing on performance, we built a new variant PREDEX$^-$ of PREDEX (=PA+GPT-4) such that we used all the statements in the execution trace before the current statement, instead of using only the statements in the *optimized* predictive backward slice. We then compare PREDEX$^-$ against PREDEX to evaluate the contribution of our PBS.

Table 3. Impact of Predictive Backward Slicing on Performance (RQ3)

	ETA-F	ETA-P	ETA-S
PREDEX$^-$ (w/o slicing)	36.5%	82.9%	69.7%
PREDEX (w/ slicing)	**49.7%**	**89.6%**	**82.3%**

Empirical Results. The comparison in Table 3 between PREDEX$^-$ (full trace) and PREDEX (optimized backward slice) highlights a key design in PREDEX. When compared to PREDEX$^-$, this represents a relative increase in ETA-F, ETA-P, and ETA-S by *36.1%, 8.0%*, and *18.1%*, respectively. These underscore the valuable contribution of our PBS, which not only refines the input to the LLM for more efficient, but also boosts the performance since the LLM deals with shorter code.

Upon investigating the results, we identified several reasons for the performance improvement achieved with predictive backward slicing (PBS). First, PBS eliminates statements that are irrelevant to the condition expressions, thereby reducing the number of statements the LLM must process. This helps minimize error propagation and hallucinations. Second, PBS allows the LLM to evaluate condition expressions that are independent of undefined variable values, which are often caused by missing input values or missing intializations of some variables (see Section 5).

Note that the accuracies of PREDEX$^-$ are even higher than those of CodeExecutor and GPT-4 (Table 1), indicating the positive contribution of *our design strategy on breaking down the execution prediction into predicting the next executed statement* (since PBS was not used in PREDEX$^-$).

RQ3. *The combination of LLM and PA via predictive backward slicing (PBS) in PREDEX helps improve predictive execution performance. PBS is essential in improving PREDEX's performance.*

8 Code Coverage Prediction Application (RQ4)

Procedure and Evaluation Metrics. To show PREDEX's usefulness, we used its predicted traces to compute code coverage for a test case. We compared the predicted coverage with the coverage from actual execution. We compared PREDEX* and PREDEX with CodeExecutor [14] and GPT-4 [6] in code coverage prediction on 100 randomly chosen test cases. We used Statement Coverage Accuracy (**SCA**) and Branch Coverage Accuracy (**BCA**). SCA is computed as the ratio between the number of correctly predicted statements as covered or non-covered ones over the total number of the covered or non-covered statements. To calculate BCA, we considered the branches in the conditional statements (`for`, `while`, `match`, *etc.*).

Table 4. Performance on Code Coverage Prediction (RQ4)

	Statement Coverage Acc. (SCA)	Branch Coverage Acc. (BCA)
CodeExecutor	66.7%	72.8%
GPT-3.5	59.6%	66.8%
PREDEX*	75.3%	79.9%
PREDEX	79.6%	84.3%

Empirical Results. As seen in Table 4, PREDEX* enhances the coverage prediction's performance of CodeExecutor. For a test case of a given code, on average, *75.3%* of the statements and *79.9%* of the branches are correctly marked as covered based on the predicted execution traces. Similarly, PredEx boosts the performance of GPT-4 by 33.5% and 26.2% in statement and branch coverage accuracies, respectively. With good accuracies (>75%) in statement/branch coverage predictions, PredEx and PredEx* are useful in code understanding, manual debugging, coverage estimation, test case prioritization or applications that tolerate a degree of inaccuracies, without actual execution.

RQ4. *PredEx's execution traces are useful in predicting code coverages for test cases with the high accuracies in statement coverage and branch coverage prediction.*

9 Scalability (RQ5)

In this experiment, we aim to evaluate how well PredEx (we chose the better variant of PredEx) supports real-world Python code: 1) with API calls and 2) with larger sizes.

Dataset. Our dataset in this RQ5 is BugsInPy, which is part of Widyasari *et al.* [25]. It contains 493 bugs with 112,602 test cases from 17 Python projects (see their paper for the full statistics of BugInPy [25]). Since the dataset is large, in this experiment, we sampled it with the confidence level of 99% and the confidence interval of 0.97. The sampled dataset contains 1,820 complete programs ranging from 4.3K to 292.2K LOCs. The code uses both built-in and third-party Python libraries. We use Hunter [2] to collect the execution traces from those test cases as the ground truth.

Table 5. Execution Trace Prediction on BugsInPy dataset (FSL: Few-Shot Learning, FT: Fine-tuning) (RQ5)

	ETA-F	ETA-P	ETA-S
GPT-4	30.0%	49.3%	58.8%
GPT-4 + FSL	34.0%	52.7%	72.0%
CodeExecutor	29.0%	45.7%	62.7%
CodeExecutor + FT	33.0%	50.3%	67.1%
PredEx	40.9%	52.5%	72.6%
PredEx + FSL	**50.7%**	**66.1%**	**75.4%**

Empirical results without few-shot learning or fine-tuning. As seen in Table 5, when predicting the execution traces for the larger programs in the BugsInPy dataset, PredEx achieves slightly less accurate than predicting for smaller programs in CodeNetMut. In 40.9% of the cases, it predicts correctly the entire execution traces. The average for prefix matching is 52.5%, i.e., among the incorrect predicted traces, they contain about half of the traces that are correct. Regarding the statement level, when predicting the next execution statement, PredEx is successful in *72.6%* of total cases (an ETA-S score of 72.6%). PredEx also performs better in all metrics than the baselines GPT-4 and CodeExecutor without few-shot learning and fine-tuning.

The results also show that all the metrics for the programs in BugsInPy are slightly lower than those for CodeNetMut. This can be attributed to several factors: (1) The programs in BugsInPy are more complex than those in CodeNetMut; (2) The execution traces for each run in the BugsInPy are longer than in CodeNetMut; and (3) There are more external API calls in BugsInPy.

Empirical results with few-shot learning or fine-tuning. To mitigate the impact of these third-party libraries, we applied fine-tuning/few-shot learning to the models. The outcomes show that few-shot learning and fine-tuning relatively enhance all the metrics for the models. As seen in Table 5, in half of the cases, PredEx predicts correctly the execution traces. The prefixes of the traces are correct in 66% on average. In 75% of the traces, all the next statements are predicted correctly.

As seen, even with FSL and FT, CodeExecutor+FT and GPT-4+FSL are less effective than PredEx+FSL. We attribute this improvement to the way that we break down the predictive execution of the entire program into smaller predictive executions of smaller program slices. This can significantly reduce the complexity of learning, as the LLM only needs to predict the output of fewer individual API calls, reducing the error propagation of the LLM in predicting the results of multiple API calls.

RQ5. *By breaking down the predictive execution into those for smaller program slices, PredEx scales to large programs. Few-shot learning and fine-tuning are helpful in dealing with external API calls.*

10 Usefulness in Static Prediction of Run-time Errors (RQ6)

Experimental Setup. In this experiment, we demonstrate the usefulness of PredEx in a task of predicting a runtime error for a given input of an incomplete code snippet. We chose PredEx (`PA+GPT-4`) because it performs better than PredEx* as seen in RQ1. PredEx can be beneficial for this task because it can tolerate the imprecision of full execution traces. Given an input to a Python (in)complete code snippet, PredEx predicts whether the code will encounter a runtime error when executed with that input. With that capability, PredEx can be used in the process of statically detecting runtime errors. To complete that process, a fuzzer can be used to generate inputs that might trigger a runtime error, which are then fed into PredEx for a prediction of its presence.

Dataset. We utilized the FixEval dataset [12], which is derived from the Project CodeNet dataset [18]. FixEval is organized by problem IDs, each corresponding to a unique programming challenge with multiple erroneous submissions. Each Python code snippet within the dataset is executable and may contain runtime errors. To execute each code snippet and obtain execution traces, we used `pythonhunter` [2]. To create a dataset of incomplete code snippets, we modified the complete code in our dataset by removing `import` statements for built-in and external libraries and by removing enclosing methods and classes, rendering the code incomplete or uncompilable. This was necessary because, to our knowledge, no existing dataset contains incomplete code alongside their complete code versions, which is needed for executing the code to gather ground-truth coverage data.

We randomly sampled 100 Python code snippets that produce runtime errors with specific inputs. Additionally, we randomly selected 100 bug-free instances (i.e., with inputs that do not cause runtime errors) to create a balanced dataset of 200 instances. Each instance includes an incomplete code snippet and its corresponding input. To ensure a random and diverse sample, we calculated the average number of instances per problem across all submissions and randomly selected a proportional number of instances from each problem.

Procedure. The input for a code snippet is encoded as the assignments of the input values to the corresponding variables. The algorithm in Section 3.2, which aims to predict the execution trace, needs to be enhanced to predict the runtime error in such a code snippet. First, when an end node in the CFG is reached, we will compute the PBSes for all the active variables in the current scope and feed to the LLM/PLM to make the predictions on their values as well as the prediction on whether any runtime error could occur. This is needed because for the execution trace prediction, we do not need to perform the "execution" of those sequential statements at the end of the code snippet (the execution order is always decided). Second, a special case of that scenario is when a code snippet contains no branching statements (only sequential statements), and PredEx encounters the end node. PredEx will feed the statements into the LLM to predict the execution and any runtime error. In this scenario, we break the sequential statements into smaller chunks of code to feed to the LLM. Finally, in the prompt for GPT-4, we added an instruction to not only perform the predictive execution, but also recognize any runtime error(s)/exception(s).

Evaluation Metrics. We measured the traditional metrics: True Positive, True Negative, False Positive, and False Negative. Accuracy is measured as the ratio between the number of instances that a model can correctly detect over the total number of instances. Accuracy = (TP+TN)/(TP+TN+FP+FN).

Table 6. PredEx's Performance on Static Detection of Runtime Errors (RQ6)

Model	Actual	Total	Predicted		Accuracy
			Buggy	Non-buggy	
PredEx	**Buggy**	100	TP = 94 (94%)	FN = 6 (6%)	**89.5%**
	Non-buggy	100	FP = 15 (15%)	TN = 85 (85%)	

Empirical Results. As seen in Table. 6, PredEx predicts correctly runtime errors for given inputs of incomplete code snippets with TP of **94%**. Among the buggy instances, our tool did not detect any runtime error in **6%** of them (FN). Looking further into those instances, we found that the main reasons for such inaccuracy include 1) incorrect computation of variables' values in the PBSes simply due to hallucination even with some short PBSes, 2) incorrect computation of API calls to built-in or third-party API calls, 3) incorrectly handling the objects as values in the heap (we could use the external PA tools to handle the store and values), and 4) incorrect error types.

For the non-buggy instances, PredEx correctly detects them as non-buggy in **85%** cases. To count a correct detection for a non-buggy case, beside correctly identifying the non-buggy instance as non-buggy, it must follow the correct execution path. The false positive rate is **15%**. We further examined these FP cases and reported the same reasons as in the above FN cases. We also counted a case as incorrect if the detected error type does not match with the error type in the ground truth. The most common runtime errors that were detected by PredEx include: *Operand Type Mismatch, NoneType Subscripting, Argument Count, Invalid Argument Type*, and *Type Conversion.*

RQ6. *PredEx statically predicts runtime errors in (in)complete code with high accuracy (**89.5%**).*

11 Related Work

Several approaches enhance ML models' reasoning on program behaviors at runtime, following two main paradigms. First, language models are pre-trained with execution data to improve their performance on downstream tasks. For instance, CodeExecutor [14] pre-trains UniXcoder [11] on execution data including source code with inputs, respective traces, and program states. However, this places a significant burden on the pre-trained language model (PLM) to learn the transformation from source code to long execution traces, leading to error propagation and reduced accuracy. In contrast, PredEx employs blended analysis to divide long executions into smaller, manageable sequences, simplifying the learning process. TRACED [8] is another approach that utilizes the PLM strategy. The authors pre-train UniXcoder [11] with execution data. However, their pre-training objectives are limited to tasks such as code generation, code coverage, and value range prediction, which make it less effective in handling iterations with conditional execution. In comparison, PredEx decomposes loops into sequential statements and branching points, thereby improving the LLMs' prediction accuracy. TraceFixer [4] uses an encoder-decoder for program repair by learning how to edit source code. The model is trained with the buggy code, execution traces, desired values, and the bug-fixed code. In comparison, its goal is not to predict execution traces as in PredEx.

The second category focuses on improving LLMs' reasoning about program runtime behaviors. Tufano *et al.* [22] use GPT-4 with prompting to predict code coverage, but it struggles with complex nested structures and conditions due to hallucinations and error propagation, resulting in low accuracy. CodePilot [7] enhances prompting through LLM-based planning with Chain-of-Thought (CoT) [24], guiding the LLM to autonomously generate execution plans for code coverage. In contrast, our approach uses PA with PBS to enable deterministic decisions and reduce the prediction burden.

NExT [16] incorporates variable states as annotations of executed lines into the prompt and applies CoT to teach LLMs predictive execution. However, both CodePilot and NExT violate program semantics when executing statements. By comparison, PredEx leverages PA to maintain semantic correctness when it is possible, significantly reducing errors and their propagation.

LExecutor [21] enables under-constrained execution of (in)complete code by using a neural model to predict and inject missing values for undefined variables and absent function return values. Leveraging CodeT5 [23], it predicts runtime values based on the surrounding context. This capability can assist PredEx in deriving the variable values and return values from unknown API calls.

12 Conclusion, Discussions and Implications for Future Work

Novelty. PredEx is a novel step toward advancing the LLMs' capabilities in reasoning on program execution and dynamic behaviors. In this work, our principle is a blended analysis between PA and LLM to use PA to guide the LLM in predicting execution traces. We found that a PA process can be enhanced with the LLMs when the decisions cannot be derived effectively via PA. Moreover, PredEx is useful in predicting code coverage and predicting the runtime error(s) for a given input of an incomplete code, in which traditional PA cannot support such detection.

Limitations and Threats to Validity. There are rooms for improvement in PredEx. First, PredEx does not support general programs, e.g., event-driven programs, framework-based programs, GUI-oriented code, client-server architecture, etc. Second, it can support code with third-party libraries, but does not work well for uncommon APIs on which the LLM was not pre-trained or fine-tuned. Third, we rely on the LLM to detect crashes and exceptions. In future, we could enhance this aspect with more PA tools. Fourth, if a program has only sequential statements, while the trace is easily predicted, the LLM might struggle in computing variables' values. For this, we could break the code into smaller fragments and feed to the LLM. Finally, the prediction time is still slow due to the API calls to the LLM. However, PredEx provides a good solution when actual execution is undesirable.

We currently support Python. However, our blended analysis is general for any language. Our datasets might not be representative. CodeExecutor and GPT-4 might not be representative for all LLMs. Evaluation with other LLMs is in our future work. While the code in our dataset is available online, few execution traces of those snippets are accessible. Moreover, the inputs are not exhaustive and generalized in all domains. Thus, considering the lack of both source code and execution traces for all (potentially infinite) input spaces, the soundness is not affected by data leakage.

Impacts on Future Research. First, our findings indicate that an LLM can be enhanced to predict execution traces by *incorporating semantic guidance obtained via external program analysis (PA) tools.* Currently, only the program execution semantics and program slicing are used to guide the LLMs. Other types of PA can be used for different tasks in dynamic analysis. A multi-agent framework that combines LLM-based agents with PA tools could be an effective approach. Second, as shown, PredEx can be used to help detect runtime errors. We could use different strategies for detecting different types exceptions and security vulnerabilities, and leverage LLMs in predicting different directions leading to the issues. Third, for reasoning about dynamic behavior, dynamic information collected during runtime can be used to guide the LLM in such reasoning and prediction. Finally, PredEx can be integrated into the fuzzing process to statically detect runtime errors in incomplete code. An LLM-based fuzzer can generate potential error-triggering inputs, and for each generated input, PredEx can predict whether it might trigger a runtime error without actual execution.

13 Data Availability

Our code and data is available at [17].

Acknowledgments

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

References

[1] [n. d.]. python-graphs. https://github.com/google-research/python-graphs.

[2] [n. d.]. Python Hunter, howpublished = https://github.com/ionelmc/python-hunter, note = Accessed: 07/21/2023.

[3] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. *SIGPLAN Not.* 25, 6 (jun 1990), 246–256. doi:10.1145/93548.93576

[4] Islem Bouzenia, Yangruibo Ding, Kexin Pei, Baishakhi Ray, and Michael Pradel. 2023. TraceFixer: Execution Trace-Driven Program Repair. arXiv:2304.12743 [cs.SE]

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

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

[7] 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

[8] 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

[9] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. 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, 121–136.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[11] 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)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.499

[12] 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

[13] Hyunji Hong, Seunghoon Woo, and Heejo Lee. 2021. Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. In *Proceedings of the 37th Annual Computer Security Applications Conference* (, Virtual Event, USA,) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. doi:10.1145/3485832.3488026

[14] 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

[15] Llama [n. d.]. Llama. https://llama.meta.com/.

[16] 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.

[17] Predictive Execution [n. d.]. Predictive Execution. https://github.com/predictiveexecution/predictive-execution.

[18] 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

[19] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. 2021. Toxic Code Snippets on Stack Overflow. *IEEE Transactions on Software Engineering* 47, 3 (2021), 560–581. doi:10.1109/TSE.2019.2900307

[20] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 263–272. doi:10.1145/1081706.1081750

[21] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1522–1534. doi:10.1145/3611643.3616254

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

[23] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

[24] 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.

[25] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[26] Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2501–2513. doi:10.1109/ICSE48619.2023.00209

[27] 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