T-REX: Teaching Large Language Models to Reason with Verbalized Execution Semantics

YAN WANG✉, Central University of Finance and Economics, China
LING DING, Central University of Finance and Economics, China
JIECHEN SUN, Independent, China
TIEN N. NGUYEN, University of Texas at Dallas, USA
SHAOHUA WANG, Central University of Finance and Economics, China
AASHISH YADAVALLY, University of Central Florida, USA
XIN XIA, Zhejiang University, China
YANAN ZHENG, Yale University, USA

Large language models (LLMs) have shown strong performance in static code tasks like code search, summarization, and generation, but remain limited in dynamic code reasoning, which involves inferring how programs behave during execution without actually running them. In this paper, we present T-REX, a novel teacher-student framework for execution prediction that addresses these limitations by grounding LLM training in *actual execution* and corresponding *execution semantics*. T-REX uses a large teacher model (Explainer) to generate fine-grained, stepwise natural language rationales explaining how program state *transitions* from one statement to another during actual execution. These rationales are used to train a smaller student model (Reasoner) to predict next program states, enabling accurate simulation of program behavior with lower computational cost. Our execution-grounded, rationale-driven training aligns with *transition-aware execution semantics* at the statement level, enhancing prediction accuracy. Our experiments show that T-REX enables Reasoner to outperform much larger GPT-4o and GPT-4o-mini models across multiple dimensions of runtime behavior prediction, while also aiding in static detection of runtime errors as well as in debugging. Finally, we discuss how T-REX can be generalized to static emulation of any dynamic analysis through such a teacher-student distillation, illustrating with the specific case of dynamic program slicing in Python.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering**;

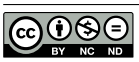
Additional Key Words and Phrases: AI4SE, Neural Networks, Execution Prediction, Execution Semantics

ACM Reference Format:
Yan Wang, Ling Ding, Jiechen Sun, Tien N. Nguyen, Shaohua Wang, Aashish Yadavally, Xin Xia, and Yanan Zheng. 2026. T-REX: Teaching Large Language Models to Reason with Verbalized Execution Semantics. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 320 (October 2026), 28 pages. https://doi.org/10.1145/3839452

1 Introduction

Large language models (LLMs) have exhibited remarkable reasoning capabilities across a range of source code-related tasks, including code search [5, 17, 41], code summarization [1, 13], and code

Authors' Contact Information: Yan Wang, Central University of Finance and Economics, Bejing, China, yanwang@cufe.edu.cn; Ling Ding, Central University of Finance and Economics, Bejing, China, 2021312380@email.cufe.edu.cn; Jiechen Sun, Independent, Bejing, China, lzhdjbsjc@126.com; Tien N. Nguyen, University of Texas at Dallas, Dallas, USA, Tien.N.Nguyen@utdallas.edu; Shaohua Wang, Central University of Finance and Economics, Bejing, China, davidshwang@ieee.org; Aashish Yadavally, University of Central Florida, Orlando, USA, aashish.yadavally@ucf.edu; Xin Xia, Zhejiang University, Hangzhou, China, xin.xia@acm.org; Yanan Zheng, Yale University, New Haven, USA, yanan.zheng@yale.edu.



ACM 2475-1421/2026/10-ART320
https://doi.org/10.1145/3839452

generation [23, 31, 34, 47]. However, they *remain significantly limited in dynamic code reasoning*, *i.e.*, reasoning about how a program behaves for specific test inputs *without actual execution* [6, 25, 32]. This stems primarily from LLMs being trained on static code representations, which lack explicit runtime contexts and do not capture runtime behaviors. As a result, their ability to simulate program execution by predicting program states and execution steps (collectively termed *execution prediction*) without actual execution is fundamentally constrained.

Addressing the problem of execution prediction has the potential to unlock several important applications where actual execution is infeasible or undesirable. First, while online forums are rich sources of reusable code, such code is often buggy and vulnerable [20, 40, 46], and can break client applications when adopted [12]. *Static detection of runtime errors in online code snippets* through execution prediction and dynamic code reasoning can help mitigate risks and reduce wasted adaptation effort. Second, execution prediction can assist developers in *debugging* in IDEs (all without actual execution) by simulating runtime behavior and predicting variable states for a statement or a sequence of statements up to a breakpoint in the current incomplete code. Third, in the case of *test case prioritization*, actual execution would defeat its key goal. Execution prediction can optimally identify untested or error-prone areas, thereby guiding the selection of test cases.

The state-of-the-art approaches for execution prediction can be broadly classified into three lines of work. The *first* category used the expressiveness of sequence-to-sequence-based models to learn execution patterns directly from code [4, 52]. Subsequent works adopted a *pretrain-then-finetune* paradigm with Transformer-based models [45], improving execution prediction through supervised training on curated run-time datasets [29]. *Recently*, CodeLlama [43] and GPT-4o [15] were used with Chain-of-Thought (CoT) [49] *prompting strategies* to simulate step-wise program execution [6, 25, 32]. NExT [33] enhances these prompting approaches with run-time contexts by annotating program states as comments. However, they remain suboptimal when reasoning through branching logic due to modeling input-output mappings, *without explicitly learning how state transitions occur at branching points*. This leads to hallucinations and error propagation [9, 44].

To improve how LLMs handle control flow, PredEx [27] introduces a *program analysis* (PA)-driven approach that combines LLMs with predictive backward slicing to first predict the branching points. Orca [37] incorporates control flow graphs (CFGs) into ReAct-based planning [51] to also enhance reasoning on branching statements. Both approaches require multiple inference steps over LLMs with billions of parameters, making them prohibitively expensive to deploy at scale [19].

Humans learn programming by reasoning about control flow, variable states, and dependencies, rather than by simply observing *input-output mappings*. The foundation of such reasoning is captured by *execution semantics*, a specification used in traditional interpreters and compilers to define dynamic program behaviors (Section 3.1). Recent work, e.g., SemCoder [10], has explored grounding LLMs in such semantics by training on *monologues*: natural language descriptions of how a program executes. However, it suffers two key limitations. First, *such monologue descriptions are predicted post-hoc from LLMs and not grounded in actual execution*, in which the LLMs can suffer from hallucinations and the monologues can contain logical inconsistencies in execution reasoning (Fig. 1), thereby limiting the effectiveness of monologue-based execution prediction. Second, these are still based on *coarse-grained, input-output operational semantics*, which abstract away the causal structure of execution. Specifically, it does not explicitly model two critical aspects of *execution semantics*: (1) **program state transitions**, and (2) updates to the program counter (*i.e.*, the **next statement** to be executed). As a result, the explanations are not aligned with stepwise state transitions, limiting the effectiveness of teaching an LLM in execution semantics.

In this work, we present T-REX, a framework that adopts the **teacher-student learning** paradigm (also called *knowledge distillation* [19]) to train LLMs using **fine-grained**, **stepwise execution rationales** that are **grounded in actual execution**. To this end, we first execute source code in

the training dataset with their test inputs. Then, we leverage the code understanding capabilities of a powerful *teacher* model (Explainer) to generate natural language explanations describing how program state evolves at each execution step for different statement types in those collected executions. Specifically, we prompt it to *explain the application of operational rules of execution semantics* (Section 3.1) for *each statement during execution* in the training phase, guiding it to focus on the two above aspects of execution semantics (program states and next statement). This mirrors how a programming language instructor teaches students, applying operational rules to specific statement examples. These capture changes to variable values and control flows, spanning a wide range of program constructs, including control structures (*e.g.*, conditionals, loops), operations (*e.g.*, assignments, updates), and function calls. We use these rationales to train a *smaller* model (Reasoner), which, given a program and its current state, learns to predict the next state. This enables Reasoner to simulate stepwise execution with lower computational overhead.

Each training instance with the rationales/explanations from the teacher model is an illustration of how an individual statement should be executed according to its applicable execution semantics rules, providing semantically structured supervision. Our training yields *execution-grounded*, *fine-grained*, *high-quality* representations of program behavior. In contrast, prior approaches treat execution as a *black-box process*, training LLMs on: raw execution data via pre-training or fine-tuning [29, 33], input-output pairs [26, 33], or *LLM-generated monologues on input-output operational semantics* over entire code blocks [10], without modeling such transitions. Note that our *execution-grounded rationales (for each actual execution step)* are more precise than LLM-generated monologues [10] (*e.g.*, SemCoder) since the latter requires the LLM to both *predict and explain* the execution of a complete program. Importantly, in addition to grounding on actual execution, we also establish **verification mechanisms** to ensure the high-quality generated rationales including 1) prompting the teacher model *to output a fixed structure* on program state, next statement, and transition rationales, 2) *reject sampling* by *filtering the incorrect samples*, 3) restricting a small context of only three statements for each training instance, and 4) manual sampling and verifying a statistically significant portion of the rationales. We posit that our training enables Reasoner to reduce hallucinations, *internalize* dynamic reasoning and *generalize effectively* across scenarios.

We also introduce a Stack-based Iterative Prompting Algorithm (SIPA) that prompts Reasoner to serve as a **predictive interpreter** that simulates *inter-procedural* program execution step by step, predicting both intermediate program states and full execution traces while maintaining the call stack. We conducted an evaluation of T-REX, comparing the student against its teacher as well as multiple state-of-the-art, code-specific LLMs in two popular benchmarks. T-REX helps achieve significant improvements, with the best model outperforming GPT-4o mini and GPT-4o by an average of 140.9% and 28.1% in next-statement prediction accuracy. Moreover, we assess T-REX's usefulness in predicting runtime behavior, including execution trace, code coverage, and program outputs, as well as in two downstream tasks: 1) static detection of runtime errors and 2) debugging.

In brief, this work makes the following contributions:

(1) ***Novelty***. A teacher-student framework for execution prediction with interpretable reasoning.
(2) A rationale-based supervision that grounds model training in *actual execution* and *transition-aware operational semantics*, incorporating transition reasoning about code behaviors.
(3) ***Empirical Evaluation***. An evaluation of T-REX, showing its usefulness in several tasks.
(4) ***Generalization***. A generalization of T-REX in a static emulation of dynamic analyses. We also illustrate our framework in a static approach for dynamic program slicing for Python.

2 Motivation

Several studies have explored using LLMs to simulate program execution [6, 32]. As an illustration, consider the program in Fig. 1, inspired by an instance in the CodeNetMut dataset [29]. This program

```
1   a = [5, 2, 9, 11]
2   maxa = max(a)
3   mini = [[] for _ in range(len(a))]
4   for i, v in enumerate(a):
5       r = maxa - 2 * v
6       if (v - r) % 2 == 1:
7           s = v - r + 1
8       else:
9           s = v - r + 2
10      mini[i] = [v, math.sqrt(s)]
11  mini.sort(key = lambda x: x[1])
```

Step	Line	Program State Updates
1	1	{a: [5,2,9,11]}
2	2	{maxa: 11}
3	3	{mini: [[],[],[],[]]}
4	4	{i: 0, v: 5}
5	5	{r: 1}
6	6	–
7	9	{s: 6}
8	10	{mini: [5,2.44945],[],[],[]}
9	4	{i: 1, v: 2}
10	5	{r: 7}
11	6	–
12	7	{s: -4}
13	10	⟨CRASH⟩

Forward Monologue: Coarse-Grained Operational Semantics

Given Python function for the provided input, let's analyze the code step by step. We have an array `a=[5,2,9,11]` and we need to evaluate the operations defined in the code.

(1) `maxa = max(a)` computes the maximum value of the list `a`, which is 11.
(2) `mini = [[] for _ in range(len(a))]` initializes a list called `mini` with empty inner lists, having the same length as `a`, so `mini` is `[[],[],[],[]]`.
(3) The code then enters a loop where it processes each value in `a` using its index `i` and value `v`. We need to evaluate the computations for each iteration in the loop.
(4) Iteration 0 (`i = 0, v = 5`):
  (a) `r = maxa - 2 * v` ⟶ `r = 11 - 2 * 5 = 1`
  (b) `(v - r) % 2` becomes `(5 - 1) % 2` ⟶ `4 % 2` ⟶ 0. So, execute the else block:
  `s = v - r + 2` ⟶ `s = 5 - 1 + 2 = 6`
  (c) `mini[0] = [v, math.sqrt(s)]` ⟶ `mini[0] = [5, sqrt(6)]`
(5) Iteration 1 (`i = 1, v = 2`):
  (a) `r = 11 - 2 * 2 = 7`
  (b) `(2 - 7) % 2` ⟶ `(-5) % 2` ⟶ 1. This is odd, go to if block: `s = 2 - 7 + 1` ⟶ `s = -4`
  (c) `mini[1] = [2, math.sqrt(-4)]` results in a potential issue here because the square root of a negative number is not defined in real numbers.
(6) Iteration 2 (`i = 2, v = 9`):
  …
(7) Iteration 3 (`i = 3, v = 11`):
  …
(8) After filling in `mini`, we have:
  `mini[0]=[5,sqrt(6)]`, `mini[1]=[2,math.sqrt(-4)]` (not a real number), `mini[2]=[9,sqrt(18)]`, `mini[3]=[11,sqrt(24)]`. Sorting `mini` using the second element of each inner list (which are square roots) will produce a sorted list. However, because of `math.sqrt(-4)`, this may result in an error or an invalid entry in a real scenario. At this point, the actual sorted output of `mini` (neglecting the square root of negative number for practical reasons) cannot be determined correctly.

[ANSWER] Error due to `math.sqrt(-4)` **[/ANSWER]**

Fig. 1. Motivating example: (*top-left*) Python code snippet, (*top-right*) its execution trace table tracking updates to program state, and (*bottom*) forward *monologue* representing *(input-output)* operational semantics [10].

crashes due to a *"ValueError: math domain error"*, arising from `s` attaining a negative value in `math.sqrt(s)` on line 10. In this case, `s` attains the value `-4` in the second iteration of the `for`-loop on line 4. To statically detect such runtime errors, LLMs need a deeper understanding of how each computation and conditional logic affect variable states at each step. However, state-of-the-art LLMs misinterpret complex control structures, such as treating mutually exclusive `if-else` branches as independent and sometimes predicting both branches as executed [6, 9]. That is, they are limited in understanding execution semantics–the operational rules that govern how programs behave at runtime.

Observation 1 (Limitations in Dynamic Code Reasoning). *LLMs tend to simulate code reasoning by relying on statistical co-occurrence patterns captured during pretraining, rather than modeling the stepwise transitions governing dynamic behaviors. As such, they hallucinate about control flow, misinterpret the sequences of statements, fail to update the program state across consecutive steps.*

To bridge this gap, recent approaches (e.g., SemCoder [10]) have proposed training on *LLM-generated monologues*, *i.e.*, high-level natural language texts produced by large models such as GPT-4o-mini [10]. However, the monologue approach has two key limitations. First, they are *predicted by GPT-4o-mini* explaining the execution of *entire coarse-grained* code snippets (e.g., a function), which makes them prone to *hallucinations*. For instance, SemCoder prompts an LLM to produce the monologue for the function in Fig. 1 (*top-left*). The model correctly identifies the loop structure, while also flagging a potential error due to computing `math.sqrt(-4)` (highlighted in yellow). Yet, it *fails to recognize that this error would terminate the program*, and continues analyzing subsequent iterations as if execution proceeds normally (highlighted in blue). Second, the monologues focus on *input-output operational semantics* and fail to capture the causal relations of program state observed during execution. As seen in Fig. 1 (*bottom*), the monologues do not explicitly reflect critical aspects of execution: (1) program state transitions and (2) the next statement to be executed. Thus, the model violates the exceptional behavior rule (Fig. 3) and continues execution even after encountering `sqrt(-4)`. In brief, *the monologues for execution prediction produced by LLMs without grounding in actual execution suffers from hallucinations, and lacks the details about causal relations of state transitions in execution*–making them coarse-grained and prone to inaccuracies.

We propose training LLMs to *internalize* execution semantics by grounding their reasoning in:

1) Operational rules, *i.e.*, formalisms that define how each statement (*e.g.*, conditionals, loops, function calls) transforms program state during execution (see Section 3.1). Rather than relying on coarse-grained, input-output mapping, monologue-style supervision, we utilize *fine-grained* natural language explanations aligned with the two aspects of *operational rules* (*program state transition and next statement*), which we refer to as **transition-aware execution rationales**.

2) Actual execution: to reduce the hallucinations from LLMs in generating monologues, we perform actual execution and leverage LLMs only to generate the rationales/explanations on stepwise execution for each statement focusing on the two above aspects. *Producing explanations for an actual, correct execution is considerably more accurate for LLMs than predicting the execution itself and explaining it as in SemCoder's monologues, which is a core limitation of these LLMs.*

In this example, we ran the Python source code to collect its execution trace. We then extracted the part of the execution trace corresponding to three statements at a time together with the entire source code (and its inputs). We used them to prompt GPT-4o-mini to *reason about each statement by considering the current program state, evaluating the execution of its sub-constituents (e.g., the condition expression, true/false branches), and determining the updates to variable values and the next statement to be executed.* Specifically, we guide the model to follow the logic of the relevant execution semantic rule—such as the `if` rule shown in Fig. 3—being applied to the `if` statement at line 6 in Fig. 1. This helps the model understand, e.g., why the condition `(v - r)%2==1` evaluates to `true` or `false`, and how that evaluation influences the updates to `mini[i]` and `s` during each iteration. Fig. 2 shows a simplified version of our verbalized execution rationales for the example, focusing on program state transitions and next statement. The full verbal details are shown in Fig. 4 (*right*).

Key Idea 1 (Grounding in Verbalized Operational Rules). *By post-training LLMs on natural-language execution rationales that are aligned with formal operational rules, we enable the models to internalize how a program changes its state during execution and transits to the next statement.*

As shown in Fig. 1, SemCoder [10] employs an LLM for predicting execution along with textual monologues, but it hallucinates beyond the actual crash point, thereby violating execution semantics. In contrast, when provided with the actual execution trace, asking the LLM to explain program states and the transitions to subsequent statements is simpler and reduces hallucinations (Fig. 2), thereby improving the learning effectiveness. The rationale is that producing execution rationales for an actual execution for an LLM is much easier than predicting that execution itself as in SemCoder.

T-REX: Fine-Grained, Stepwise Rationales

(1) Line 1: `a = [5, 2, 9, 11]`
  (a) *Before*: *a* undefined.
  (b) *Analysis*: assign list.
  (c) *State*: `a = [5, 2, 9, 11]`
  (d) *Next*: line 2
(2) Line 2: `maxa = max(a)`
  (a) *Before*: `a = [5, 2, 9, 11]`; `maxa` undefined.
  (b) *Analysis*: compute `max([5, 2, 9, 11]) = 11`.
  (c) *State*: `maxa` = 11.
  (d) *Next*: line 3.
(3) Line 3: `mini = [[] for _ in range(len(a))]`
  (a) *Before*: a has length 4; `mini` undefined.
  (b) *Analysis*: ...
  (c) *State*: `mini = [[], [], [], []]`
  (d) *Next*: line 4
(4) Line 4: `for i, v in enumerate(a):`
  (a) *Before*: `a` = [5, 2, 9, 11]
  (b) *Analysis*: ...
  (c) *State* (Iteration #0 begins): `i = 0, v = 5`
  (d) *Next*: line 5 (inside loop).
(5) **Iteration #0** (i = 0, v = 5)
(6) Line 5: `r = maxa - 2 * v`
  (a) *Before*: `maxa = 11, v = 5`
  (b) *Analysis*: ...
  (c) *State*: `r = 1`
  (d) *Next*: line 6
(7) Line 6: `if (v - r) % 2 == 1:`
  (a) *Before*: `v = 5, r = 1`
  (b) *Analysis*: `v - r` = 5 - 1 = 4; 4 % 2 = 0; condition 0 == 1
  (c) *State*: because the condition 0 == 1 is False, the else branch is next.
  (d) *Next*: line 9 (`else` branch).

T-REX: Fine-Grained, Stepwise Rationales

(9) Line 9: `s = v - r + 2`
  (a) *Before*: `v - r = 4`
  (b) *Analysis*: ...
  (c) *State*: `s = 6`
  (d) *Next*: line 10.
(10) Line 10: `mini[i] = [v, math.sqrt(s)]`
  (a) *Before*: mini = [[], [], [], []], i = 0, v = 5, s = 6
  (b) *Analysis*: ...
  (c) *State*: `mini` = [[5, 2.449489743...], [], [], []]
  (d) *Next*: loop continues back to line 4 header for next iteration.
(11) **Iteration #1** (`i = 1, v = 2`)
(12) Line 5: `r = maxa - 2 * v`
  (a) *Before*: `maxa` = 11, v = 2
  (b) *Analysis*: ...
  (c) *State*: `r = 7`
  (d) *Next*: line 6.
(13) Line 6: check condition `if (v - r) % 2 == 1:`
  (a) *Before*: `v = 2, r = 7`
  (b) *Analysis*: ...
  (c) *State*: condition `1 == 1` is `True`
  (d) *Next*: condition is true, line 7 (then-branch) is next.
(14) Line 7: `s = v - r + 1`
  (a) *Before*: `v - r = -5`
  (b) *Analysis*: ...
  (c) *State*: `s = -4`
  (d) *Next*: line 10.
(15) Line 10: `mini[i] = [v, math.sqrt(s)]`
  (a) *Before*: `i = 1, v = 2, s = -4`
  (b) *Analysis*: `math.sqrt(-4)`: raises `ValueError`
  (c) *State*: exception raised.
  (d) *Next*: loop is aborted. Stop.

Fig. 2. T-REX framework: An example of transition-aware, stepwise execution rationales for Python code snippet in Fig. 1, derived from grounding in actual execution information (see Section 4.2.2 for illustrations for different statement types).

Key Idea 2 (Grounding in Actual Execution). *By post-training a smaller LLM on natural-language execution rationales—generated by larger LLMs that explain actual program executions—we enable the model to ground its reasoning in correct program states and correct transitions to the next statements, substantially reducing hallucinations that arise when LLMs must predict execution behavior directly along with explanations.*

We adopt a *teacher-student* learning framework, where a large teacher model produces high-quality execution rationales from annotated examples, and a smaller student model learns to simulate stepwise execution from these rationales. In the example, the teacher explicitly verbalizes why condition `(v-r)% 2 == 1` evaluates as it does, and how that drives updates to `s`, eventually resulting in a runtime error. This *mirrors how a teacher instructs a student by applying an operational rule to specific statement example* (Fig. 4 (*right*)). These rationales serve as human-readable bridges between formal semantics and model comprehension, allowing a smaller model to learn not just *what* happens during execution, but also *why* and *how* it does. We posit that this enables the student model to extrapolate this knowledge, supporting cost-effective and execution-free dynamic analyses.

Key Idea 3 (Teacher-Student Learning to Distill Execution Rationales). *We enable efficient execution-free dynamic code reasoning by using a much larger teacher model to generate fine-grained, execution-grounded, transition-aware execution rationales, which are then distilled into a smaller student model via supervised fine-tuning.*

To realize this idea, we adopt a **teacher–student learning framework**. The central motivation is that a teacher can expose *structured supervision* beyond input–output pairs. By producing stepwise, transition-aware rationales, the more powerful teacher model provides an explicit training signal that encourages the student model to internalize operational semantics (e.g., state updates and control-flow transitions) rather than merely memorizing the program's input/output mappings.

This design choice is particularly important for execution-free reasoning, where ungrounded intermediate steps can easily drift. First, it improves **grounding**: because rationales are conditioned on the actual execution traces produced during offline instrumentation, the supervision remains faithful to program execution and reduces hallucinated intermediate states. Second, producing explicit rationales yields **interpretability**: they are human-readable, enabling direct inspection for debugging and error analysis. Third, the teacher–student split also delivers **lightweight deployment**. All instrumentation and rationale generation is done once offline, amortizing cost across training data, while the deployed student remains smaller, a practical requirement for general use.

Inference Granularity. During debugging, users examine the program state, such as variable values, to identify and fix errors [38]. In the example, this involves tracking the execution flow step-by-step leading up to the crash on line 10, specifically how variables `v`, `r`, and `s` are updated across loop iterations. Such stepwise information is essential for debugging root causes and providing actionable feedback. Beyond debugging, other dynamic analyses like dynamic slicing also rely on accurate, fine-grained predictions of control flow and program state transitions (see Section 11).

Observation 2 (Inference Granularity). *Generalizable dynamic code reasoning requires task-appropriate prediction granularity, ranging from fine-grained stepwise reasoning over individual statements to high-level program behaviors.*

We design the student model to reason iteratively, predicting the next executed statement and its effect on the program state. In the example, this involves tracking how `mini` and `s` evolve. This inference enables the construction of execution traces by stepwise prediction, capturing the evolving program state at a step. In turn, this enables dynamic code reasoning in a range of analyses.

Key Idea 4 (Stepwise Inference via Iterative Prompting). *Iteratively prompting the student model to stepwise predict program states enables an extrapolation of fine-grained dynamic behaviors.*

3 Important Concepts

3.1 Execution Semantics

The execution semantics of a program $P = \{s_1, s_2, \ldots, s_n\}$ defines how a statement s_i transitions from one program state to another during execution. Formally, a program state is represented as a pair $\langle s, \sigma \rangle$, where s is the current statement to be executed, and σ is the current store (*i.e.*, a mapping from memory locations l_v associated with variable v to its value), representing program states. Each memory location l uniquely identifies a variable and encodes sufficient scoping information to distinguish variables with the same name in different scopes. For example, two variables v defined in different blocks will be associated with distinct locations l, such as $l_v^{(1)}$ and $l_v^{(2)}$, to avoid ambiguity. The semantics of executing a statement s under store σ is formalized by the transition relation:

$$\langle s, \sigma \rangle \Rightarrow \langle s', \sigma' \rangle$$

indicating that s, when executed in state σ, takes a single computation step to s', producing an updated store σ'. In addition, execution may also result in terminal configurations:

$$\langle s, \sigma \rangle \Rightarrow \langle s', \sigma' \rangle | \langle s, \sigma \rangle \Rightarrow \mathbf{halt}(\sigma') | \langle s, \sigma \rangle \Rightarrow \mathbf{error}(exp, \sigma)$$

where $\mathbf{halt}(\sigma)$ denotes a successful termination and $\mathbf{error}(exp, \sigma)$ denotes an unexpected one due to an exception exp. Fig. 3 shows an excerpt of the execution semantics for statements in Python.

1. Assignment Statement $x := expr$

$$\frac{\sigma \vdash expr \Downarrow v}{\langle x := expr, \sigma\rangle \Rightarrow \langle s_{next}, \sigma[l_x \mapsto v]\rangle}$$

2. If Statement if $cond$ then s_{true} else s_{false}

- (*If–True*). If condition $cond$ evaluates to $True$:

$$\frac{\sigma \vdash cond \Downarrow \text{true}}{\langle \text{if } cond \text{ then } s_{true} \text{ else } s_{false}, \sigma\rangle \Rightarrow \langle s_{true}, \sigma\rangle}$$

- (*If–False*). If condition $cond$ evaluates to $False$:

$$\frac{\sigma \vdash cond \Downarrow \text{false}}{\langle \text{if } cond \text{ then } s_{true} \text{ else } s_{false}, \sigma\rangle \Rightarrow \langle s_{false}, \sigma\rangle}$$

3. Loop Statement while $cond$ do s_{body}

- (*If–True*). If condition $cond$ evaluates to $True$:

$$\frac{\sigma \vdash cond \Downarrow \text{true}}{\langle \text{while } cond \text{ do } s_{body}, \sigma\rangle \Rightarrow \langle s_{body}; \text{while } cond \text{ do } s_{body}, \sigma\rangle}$$

Here, ; denotes executing the body and then re-evaluating the loop.

- (*If–False*). If condition $cond$ evaluates to $False$:

$$\frac{\sigma \vdash cond \Downarrow \text{false}}{\langle \text{while } cond \text{ do } s_{body}, \sigma\rangle \Rightarrow \langle s_{next}, \sigma\rangle}$$

s_{next} is the next statement after the `while` loop in the code.

4. For Statement for _ in range ($expr$) s_{body}

$$\frac{\sigma \vdash expr \Downarrow v \text{ and } v >= 1}{\langle \text{for \_ in range } (expr), \sigma\rangle \Rightarrow \langle s_{body}; \text{for \_ in range } (v\text{-}1), \sigma\rangle}$$

$$\frac{\sigma \vdash expr \Downarrow 1}{\langle \text{for \_ in range } (expr), \sigma\rangle \Rightarrow \langle s_{body}, \sigma\rangle}$$

5. Block $s_1; s_2; ...; s_n$

- (*Seq–L*). If s_1 can take a step to s_1', then $s_1; s_2; ...; s_n$ takes a step to $s_1'; s_2; ...; s_n$:

$$\frac{\langle s_1, \sigma\rangle \Rightarrow \langle s_1', \sigma'\rangle}{\langle s_1; ...; s_n, \sigma\rangle \Rightarrow \langle s_1'; ...; s_n, \sigma'\rangle}$$

- (*Seq–R*). If s_1 completes (*i.e.*, reduces to a new state σ'), then $s_1; s_2; ...; s_n$ proceeds to execute s_2 in state σ':

$$\frac{\langle s_1, \sigma\rangle \Rightarrow \langle \emptyset, \sigma'\rangle}{\langle s_1; s_2; ...; s_n, \sigma\rangle \Rightarrow \langle s_2; ...; s_n, \sigma'\rangle}$$

Both *Seq-L* and *Seq-R* imply that s_1 must fully execute before s_2 begins.

6. Function Invocation $x := f(arg_1, ..., arg_n)$, where f has a body s_f

$$\frac{\sigma \vdash arg_1 \Downarrow val_1 ... arg_n \Downarrow\ val_n \forall i : \sigma' = [param_i \mapsto val_i]}{\langle x := f(arg_1, ..., arg_n), \sigma\rangle \Rightarrow \langle s_f; \text{return-to } x, \sigma'\rangle}$$

7. Return Statement return $expr$

$$\frac{\sigma \vdash expr \Downarrow v}{\langle \text{return } expr; \text{return-to } x, \sigma\rangle \Rightarrow \langle s_{next}, \sigma[x \mapsto v]\rangle}$$

8. Exception Statement raise exp

$$\frac{\sigma \vdash s \Downarrow exp}{\langle s, \sigma\rangle \Rightarrow \textbf{error}(exp, \sigma)}$$

9. Termination Rule

$$\frac{}{\langle s_{final}, \sigma\rangle \Rightarrow \textbf{halt}(\sigma)}$$

Notations:

- $\sigma \vdash expr \Downarrow v$ denotes evaluation of $expr$ under state σ, yielding value v.
- $\sigma[l_x \mapsto v]$ denotes the store obtained by updating σ so that location l_x now maps to v, *i.e.*, variable x is assigned with the value v.
- ; denotes sequential composition; *Seq-L* means left-hand side evaluation and *Seq-R* means completion and progression.
- $param_i$s are the formal parameters of function f.
- val_is are the evaluated actual arguments.
- return-to x denotes assigning the function's return value to the variable x.
- exp is the exception/error raised by the execution of s.

Fig. 3. An excerpt of execution semantics for procedural programming languages (*e.g.*, Python).

3.2 Execution Prediction

We aim to train LLMs to *internalize* the reasoning process on program state transitions via small-step operational rules (as in Section 3.1). That is, the model would learn from the explanation of the application of each formal semantic rule to the specific program statement to infer the program state changes and transitions to the next statement during execution.

Given a test input, *execution prediction* refers to the task of determining how a program evolves during its execution. In particular, we are interested in *predicting both the program state resulting from the execution of a given statement as well as the next statement that will be executed.* For a program $P = \{s_1, s_2, \ldots, s_n\}$, let $\mathcal{I} = \{I_1, I_2, ..., I_m\}$ represent the set of possible inputs. For each $I_i \in \mathcal{I}$, the execution of P produces a trace $T_i = \langle\langle s_i^1, \sigma_i^1\rangle, \langle s_i^2, \sigma_i^2\rangle, \ldots, \langle s_i^{|T_i|}, \sigma_i^{|T_i|}\rangle\rangle$, where in each pair, $s_i^k \in P$ corresponds to the statement executed at step k, and σ_i^k denotes the program state immediately after the execution of s_i^k. We define the state σ_i^0 as the initial state (the *values of variables*) prior

Prompt to Teacher LLM for Generating Fine-Grained Execution Rationales

Input Code.

```
⟨Line 1⟩ X = 13; Y = 3; Z = 1
⟨Line 2⟩ ans = 0
⟨Line 3⟩ X -= Z
⟨Line 4⟩ for _ in range(X):
⟨Line 5⟩   if X < (Y + Z):
⟨Line 6⟩       print(ans)
⟨Line 7⟩       return
⟨Line 8⟩   else:
⟨Line 9⟩       ans += 1
⟨Line 10⟩   X -= (Y + Z)
```

Current Program State. Suppose that the code is going to execute ⟨Line 1⟩ {X=13; Y=3, Z=1}. The current values of the variables are {}.

Number of subsequent executed statements. 3

Execution trace and the values of variables after executing each statement.
⟨Line 1⟩ {'X': 13, 'Y': 3, 'Z': 1}
⟨Line 2⟩ {'X': 13, 'Y': 3, 'Z': 1, 'ans': 0}
⟨Line 3⟩ {'X': 12, 'Y': 3, 'Z': 1, 'ans': 0}
⟨Line 4⟩ {This statement will **NOT** be executed}

Instruction. Given the code, current program state, and subsequent execution result, please analyze each statement's execution. For each statement in the three statements, indicate line number first and *describe variable value changes after executing this statement, and finally point out the next statement to be executed.* If the entire program ends after the statement execution, additionally indicate that the code execution has finished...

An Example of Execution Rationales Output by Teacher LLM

Line: 1
Analysis: The first line of code initializes the variables X, Y, and Z with the values 13, 3, and 1, respectively. Since these variables are being assigned their initial values, we can now track their states. After executing this statement, the variable dictionary contains X as 13, Y as 3, and Z as 1.
State: {'X': 13, 'Y': 3, 'Z': 1}
Next Statement: 2

Line: 2
Analysis: In this line, we initialize the variable ans to 0. This step sets up ans for use in the forthcoming loop to keep track of how many times we can reduce X by the sum of Y and Z. At this point, the variable states reflect that ans has been added, while X, Y, and Z remain unchanged.
State: {'X': 13, 'Y': 3, 'Z': 1, 'ans': 0}
Next Statement: 3

Line: 3
Analysis: This line executes the operation X -= Z, which decreases X by the value of Z. Consequently, X is updated from 13 to 12. This statement effectively prepares X for the loop that follows, ensuring that it accurately reflects the current state before entering the loop. The other variables Y, Z, and ans remain unchanged.
State: {'X': 12, 'Y': 3, 'Z': 1, 'ans': 0}
Next Statement: 4

Fig. 4. (*left*) Prompt to Explainer to generate fine-grained execution rationales; (*right*) corresponding output.

to the execution of the first statement s_i^1. Formally, given a program P, a corresponding input $\forall I_i \in I$, a statement s_i^k and its preceding state σ_i^{k-1} ($\forall k \in (1, |T_i|)$), the student LLM $\mathcal{M}$ in T-REX can correctly reason about the execution of s_i^k, *i.e.*,

$$\hat{\sigma}_i^k = \sigma_i^k$$
$$\hat{s}_i^{k+1} = s_i^{k+1}$$

where $\hat{\sigma}_i^k$ and $\hat{s}_i^{k+1}$ are the predictions of $k+1$ statement in P_i and the next state σ_i^{k+1} in T_i from $\mathcal{M}$.

4 T-REX: Reasoning on Program Execution

In this work, we post-train LLMs with execution rationales, *i.e.*, natural language explanations that are grounded in actual execution and explicitly verbalize the program state transitions for each statement in the training code.

We first execute complete source code with different inputs, capturing the execution trace and variables' values. We follow the execution order and extract the execution trace for three statements at a time. With that data and the source code of the currently processed method, we then prompt the larger Explainer LLM to generate fine-grained execution rationales that verbalize *transition-aware operational semantics* rules, illustrating how each rule is applicable to a specific program statement, resulting state transition to the next statement at runtime (Section 4.1). We use these rationales to post-train a smaller Reasoner LLM, which during inference, is tasked with predicting program behaviors including next statements and program states. For a program, this reasoning process is conducted stepwise using an inter-procedural, iterative prompting strategy (Section 4.2.3).

4.1 Explainer: Teacher LLM in T-REX

To verbalize the application of execution semantics and train Reasoner, we focus on three classes of statements: *sequential, branching*, and *exception*. For each category, we construct training samples that *follow localized execution steps* and *explain* intermediate program state transitions. Specifically, we follow a program's execution trace, process each method in the call graph, and its statements. For *sequential* and *branching* samples, we extract *three consecutively executed statements from the execution trace from actual execution*, and their *program states*. In the case of sequential samples, these are drawn from consecutive lines in the trace. For branching samples, we ensure that at least one control-flow-altering statement (*e.g.*, `if`, `for`) is included, resulting in non-consecutive line numbers. In the prompt (Fig. 4, (left), "Instruction"), we instruct Explainer to focus on program states, transitions, and next statement as in execution semantics. We provide execution results for the selected statements in the *"Execution trace and values of variables..."* section (blue box). To provide the context, we include the code of the currently processed method ("Input Code").

For *exception code samples*, we consider seven commonly encountered Python error types (listed in Fig. 5). Since such cases are under-represented in our benchmark datasets (in Section 6), we generate new errors by mutating existing programs, through insertion, deletion, or mutation of a statement, to trigger a specified runtime error. Fig. 5 illustrates the prompt used to introduce such modifications. Moreover, as with the sequential and branching samples, we aggregate the prompt to LLMs to generate natural language explanations for the exception-raising statements.

Prompt in T-REX for Injecting Runtime Errors

Input Program. . . . . . .

Execution Intermediate Result. They are the execution results for a statement, provided by the runtime environment: . . . . . .

Python Error Types.
- NameError: A name that is not defined
- ZeroDivisionError: Division or modulo by zero occurs
- IndexError: A sequence index is out of range
- KeyError: A dictionary key is not found
- TypeError: An operation/function is applied to an object of inappropriate type
- UnboundLocalError: Accessing a local variable before assigned
- ValueError: An operation/function receives an argument with inappropriate value

Instruction. Given the code and its intermediate execution result, modify, add, or remove statements in the code to produce one of the error types listed in Python error types. Provide your analysis in the following format for each statement:

Analysis. [. . .]

Modified Code. [. . .]

Check. [. . .]

Fig. 5. Prompt in T-REX for generating buggy programs.

Importantly, we utilize verification mechanisms to ensure the high-quality generated rationales. First, in addition to *grounding in actual execution*, we prompt the Explainer to output a fixed structure including program states, next statements, and transition rationales, serving as *per-statement training instances* (Fig. 4, (right)). Second, we then also apply *reject sampling* by *filtering the incorrect samples* whose *program states* and *next statements* do not match with ground-truth execution data. Third, we restrict a small context of only three already-executed statements for each training instance, reducing hallucinations. Finally, we randomly sample and verify a statistically significant portion of the generated rationales. Our verification schemes, combined with execution grounding, enable Reasoner to *generalize to the new testing dataset HumanEval* (Section 6), indicating robustness against residual noise in the generated texts.

4.2 Reasoner: Student LLM in T-REX

4.2.1 Training with Execution Rationales. To post-train Reasoner using execution rationales, we employ causal language modeling (CLM), a learning objective that trains it to predict the next token based on the preceding context (see the prompt in Fig. 6). To help Reasoner internalize reasoning about execution semantics and program state transitions, we supervise it with the execution rationales from Explainer (highlighted in yellow in Fig. 4 (right) and Fig. 6).

Formally, let $\mathcal{D} = \{\mathbf{tr}_1, \mathbf{tr}_2, ..., \mathbf{tr}_N\}$ denote the training dataset, where each training sample $\mathbf{tr}_i = (\mathbf{c}_i, \mathbf{r}_i)$ consists of the prompt context $\mathbf{c}_i$ (representing a specific statement) and the respective execution rationale $\mathbf{r}_i$ distilled from Explainer. Let $w_j^{\mathbf{c}_i}$ and $w_k^{\mathbf{r}_i}$ denote the j-th token in $\mathbf{c}_i$ and the k-th token in $\mathbf{r}_i$, respectively. The training objective is to minimize the negative log-likelihood of generating each token in $\mathbf{r}_i$, conditioned on the prompt and previously generated rationale tokens:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{|\mathbf{r}_i|} \log P(w_t^{\mathbf{r}_i} \mid \mathbf{c}_i, w_1^{\mathbf{r}_i}, \ldots, w_{t-1}^{\mathbf{r}_i}) \tag{1}$$

Note that we fine-tune Reasoner one statement at a time within the program context to promote localized, transition-aware learning. As a result, Reasoner learns to internalize verbalized execution rationales corresponding to the formal operational rules in Fig. 3.

We prioritize supervised fine-tuning (SFT) over Reinforcement Learning from Human Feedback (RLHF) due to the following. The Explainer produces *trace-conditioned, transition-aware* execution rationales, which provide explicit token-level targets; thus, maximizing negative log-likelihood (NLL) with causal language modeling (CLM) directly optimizing the same next-token generation behavior used at inference time. Moreover, RLHF would incur a larger engineering and compute burden. Beyond an SFT initialization, it requires preference data and reward modeling, and the subsequent RL stage (e.g., PPO) commonly requires training and serving several models simultaneously (Policy, Reference, Critic, Reward), which could increase GPU memory use and training cost. Since NLL-based distillation delivers strong improvements in our results (Section 6), we focus on SFT for efficiency, and leave RLHF/contrastive variants for future work.

Prompt to Reasoner for Stepwise Execution Prediction

Input Program. .....

Program State. .....

Number of subsequent executed statements. 1

Instruction. Given the code, current program state, and number of subsequent executed statements, analyze each statement's execution WITHIN the number of subsequent executed statements. For each statement, first indicate the line number, then describe the variable values after executing this statement, and finally point out the next statement to be executed. If a statement encounters an error during execution, analyze its cause and type. If the entire program ends after executing a statement, indicate that the code execution has finished. Provide your analysis in the following format for each statement:

Number. [Index number of executed statement]

Analysis. [Step-by-step explanation of the execution progress in a single paragraph without enumeration and enumeration symbols.]

State. [Values of the variables in a dictionary format if the statement in executable, the error type if the statement encounters any error, or the label 'completion' if the entire program ends after the statement execution.]

Next Statement. [Index of the statement to be executed next or the error type if the statement encounters any error or the label 'completion' if the entire program ends after the statement execution.]

Fig. 6. Prompt to Reasoner in T-REX.

4.2.2 Illustration: Code → Applied Operational Rule → Verbalized Rationale. Let us use different types of program statements from the Python code in Fig. 4 to illustrate the rules and the corresponding verbalized rationales/explanations. In the following, we will state the line of code and the position in the execution trace unless it is unique.

(1) Assignment: `X = 13; Y = 3; Z = 1` → Rule 1 is applied.
Rationale: The first line of code initializes the variables ... executing this statement, variable dictionary contains `X` as 13, `Y` as 3, and `Z` as 1 (see "Analysis" for Line 1 in Fig. 4, right).

(2) 'For' statement: `for _ in range(X)` (First iteration) → Rule 4 is applied.
Rationale: Before this statement, `X`=12. `range(X)` is evaluated once now as `range(12)`. The loop will attempt up to 12 iterations unless a break is hit. The loop variable _ is independent of X.

(3) 'If' statement: `if X < (Y + Z)` (First iteration) → Rule 2 (If-False) is applied.
Rationale: Before this statement, `X`=12, `Y+Z=4`". Check the condition: 12 < 4 → False.

(4) 'Block' statement: `print(ans); break` (Iteration 4, i.e., _ = 3) → Rule 5 is applied.
Rationale: `print(ans)` is executed first, outputting 3 to stdout.
(5) 'Return' statement: `return` → Rule 7 is applied.
Rationale: Exit the method immediately. There is no more code; program ends.

4.2.3 Stepwise Inference via Stack-based Iterative Prompting (SIPA). This section presents our iterative prompting strategy for Reasoner to simulate code execution, one statement at a time. Given the current statement and program state, it predicts both next statement and updated program state to which the execution flows. Note that a naive statement-by-statement strategy is insufficient to model execution flow across caller-callee boundaries. Thus, we design a *stack-based* prompting scheme that mimics inter-procedural execution in a Python interpreter (Algorithm 1, SIPA).

Given an initial program statement and its preceding program state, SIPA iteratively prompts Reasoner, using the current statement and program state (as in Fig. 6), to predict both the next program state and the next statement to execute.

Call stack and context management. To handle inter-procedural control flow, SIPA mimics a Python interpreter by following the call graph and maintaining a *call stack* $Stack$ to track the active function call and manage the function-level execution *context* c. Every time a function is called, SIPA creates a frame, which is a record for one active function call and contains the (function-level) context needed to run that function. In addition to the references to the global context such as heap states and global variables, two key pieces of data before execution in such a function-level context (*context* for short) within a frame are the current statement index i and program state p^{i-1}. Specifically, when encountering a call in s^i (Line 10), SIPA preserves (i, p^{i-1}) in the context c and pushes it in a new frame onto the stack $Stack$ (Lines 11–12). Then, at Line 13, the `update` function realizes the function-invocation rule in Fig. 3: the `update` function takes the call s^i and the caller's state p^{i-1} to evaluate the arguments and initialize the parameters and return the callee's entry index and its initial state. The process is repeated. Upon reaching a `return` statement s^i (Lines 14–18), SIPA restores the caller's context by popping $Stack$ into c and unpacking it to get $(index, ps)$ (Lines 15–16).

Algorithm 1. Stepwise Inference

Input: $C = \{s_1, \ldots, s_n\}$, stop condition r, index i (first statement to be executed), program state p^{i-1}, LLM M, prompt pr
Output: Execution record E

```
1:  flag_r ← False, E ← [ ], j ← 0, Stack ← [ ]
2:  while not flag_r do
3:      if not callingOrReturnCheck(s^i) then
4:          E.append((s^i, p^{i-1}))
5:          o ← M(pr(s^i, p^{i-1}))                    ▷ Call Reasoner
6:          flag_r ← CheckCondition(o, r, j)
7:          if flag_r then
8:              return E
9:          (i, p^{i-1}) ← ExtractInfo(o)
10:     if callingCheck(s^i) then
11:         c ← (i, p^{i-1})
12:         Stack.push(c)
13:         (i, p^{i-1}) ← update(s^i, p^{i-1})
14:     if returnCheck(s^i) then
15:         c ← Stack.pop()
16:         (index, ps) ← c
17:         p^{index} ← update(s^i, p^{i-1}, ps)
18:         i ← index + 1
```

Next, the function `update`(s^i, p^{i-1}, ps) at Line 17 (realizing the `return` rule in Fig. 3) uses the state p^{i-1} of the `return` statement s^i and the saved state ps to update the caller's program state so that the execution resumes at $index + 1$ (Line 18). This process continues until a stopping condition is met: either the model signals normal completion or raises an exception (indicated in model output with 'completion' label or the name of the exception type), or exceeds a predefined step limit.

5 Empirical Evaluation

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

(RQ_1) [Intrinsic Evaluation] Effectiveness in Predicting Execution Semantics. How well does Reasoner in T-REX learn to predict the execution flow and program state transitions?

(RQ_2) [Extrinsic Evaluation] Effectiveness in Runtime Behavior Prediction.

- **2.1** *Execution Trace.* To what extent do the predicted sequence of program statements and corresponding program states by T-REX align with the actual execution trace?
- **2.2** *Code Coverage.* Does the execution flow produced by T-REX approximate code coverage well, *i.e.*, can it predict which statements or branches are executed at runtime?
- **2.3** *Program Outputs.* How accurately can T-REX predict the final output of a program?

(RQ_3) Usefulness in Static Detection of Runtime Errors. How effective is T-REX in the detection of runtime errors/exceptions in code snippets, *without actual execution*?

(RQ_4) Usefulness in Debugging. How effective is T-REX in aiding the manual process of debugging a Python code snippet after it has crashed, *without actual execution*?

(RQ_5) Qualitative Evaluation.

- **5.1** Does T-REX improve LLMs' dynamic code reasoning capabilities?
- **5.2** Do the execution rationales produced by Reasoner LLM (*i.e.*, student) align with the much larger Explainer LLM (*i.e.*, teacher) in T-REX?

6 Effectiveness Evaluation in Predicting Execution Semantics (RQ_1)

6.1 Experiment Setup

6.1.1 Datasets. There exist datasets to benchmark execution tasks, e.g., CodeNet [39], Avatar [2], HumanEval [7], MBPP [3], and CruxEval [16]. We used CodeNetMut [29], a mutation-based extension of CodeNet [39] that spans various domains and degrees of complexity, and has been used in the state-of-the-art CodeExecutor [29]. It comprises 19,540 programs; each has a diverse set of test cases and is annotated with runtime data, including execution traces and program states. To assess the *generalizability* of T-REX, we also use HumanEval [7] for testing. HumanEval with 396 programs has been used in prior work to assess LLMs' dynamic code reasoning capabilities [6, 30].

(A) To build **non-exception** *execution training data* for Reasoner in RQ1 and RQ2, we randomly extracted from CodeNetMut where we reused test cases (*inputs*), execution traces, and per-statement program states. For RQ1 and RQ2, we applied the *random hold-out data partitioning strategy*. Specifically, we trained Reasoner on 45,000 program statements that were randomly extracted from 9,000 randomly selected CodeNetMut programs and associated inputs. Each training instance consists of three consecutive statements in the execution traces in those programs. We tested the models on all the statements from 500 and 250 programs for RQ1 and RQ2, respectively, which are *disjoint from the training set*. The same numbers of programs were selected for testing from HumanEval dataset.

Table 1. Statistics of statement and exception types in the training data.

Statement Types		Exception Types	
Type	**%**	**Type**	**%**
Variable Assignment (S_2)	32.70%	TypeError	16.50%
Expression (S_1)	23.80%	ZeroDivisionError	16.21%
If Statement (S_3)	18.70%	NameError	15.99%
For/While Loop (S_4)	16.93%	IndexError	15.42%
Method Calls (S_5)	7.87%	KeyError	14.94%
		ValueError	13.16%
		UnboundLocalError	7.78%

To justify our data selection and mitigate sampling bias, we report training-set statistics that quantify coverage across statement types. Since aiming to teach two control-flow modes, *sequential* and *branching* executions, we report the statistics on five categories: expressions (S_1) and variable assignments (S_2) as representative sequential operations; and `if` statements (S_3), iterative constructs

(`for`/`while`/`match`) (S_4), and method invocations (S_5) as representative branching behaviors. With a sufficient large randomly-selected samples, Table 1 illustrates the diversity of our training data. For statements, the high proportion of variable assignments (S_2, 32.7%) helps the model learn to track program states, while the substantial presence of control flow structures (S_3 and S_4, over 35%) is essential for reasoning about execution paths. Note that *the input diversity in our training data is inherited from CodeNetMut [29]*: for each of the 9,000 programs, we reused all available test cases, yielding a wide range of input values and corner cases. In brief, our diverse training data enables *Reasoner to generalize beyond CodeNetMut, transferring effectively to HumanEval* (Section 6.2).

(B) For **exception** training data (RQ3), we randomly sampled a different set of CodeNetMut programs as candidates for training. For *each program with an input*, we provided GPT-4o with the original code, all intermediate execution results, and a list of Python exception types (Fig. 5). We prompted GPT-4o to modify, add, or remove statements such that executing the program on the same input triggers exactly one exception. *We then executed the resulting code with that input to verify* 1) that an exception is raised, 2) that only one exception occurs, and 3) that its exception type matches GPT-4o's claimed label. If verification passed, we accepted the program and its input as an exception sample; otherwise, we re-prompted GPT-4o up to 5 times. We repeated this procedure until collecting *5,000 verified buggy programs*, each paired with concrete inputs that deterministically trigger a specific exception type. As seen in Table 1, the exception dataset is notably balanced across almost all exception types, each comprising 13–16% of the samples. This composition shows the diversity of coverage across exception behaviors and reduces bias toward any single exception type.

For testing, we sampled the disjoint programs from CodeNetMut and HumanEval and applied the same generation-and-verification pipeline to construct 100 exception-triggering samples. We also randomly selected 100 non-exception samples from the disjoint parts of the datasets. Importantly, the broad exception-type coverage supports *Reasoner's generalization to HumanEval* (Section 8).

6.1.2 Baselines. First, we used the following reference models:

(1) DeepSeek-R1 is a 671B parameter model based on transformer architecture, combining multi-head latent attention and mixture-of-experts. Due to the restriction in usage of DeepSeek-R1's APIs, we used the derived *DeepSeek-R1-Distill-Qwen-14B* variant [22, 28, 36] off-the-shelf.
(2) *GPT-4o* [21] is OpenAI's state-of-the-art LLM that excels in code-related tasks.
(3) *GPT-4o-mini* is a version with fewer parameters and requires less computational power.

We also considered (4) *CodeLlama*, the 7B-Instruct and 13B-Instruct variants, and (5) *Qwen2.5-Coder*, 7B-Instruct and 14B-Instruct variants as the LLMs in Reasoner, comparing against the base versions of both (*i.e.*, without supervised fine-tuning on execution rationales). For training, we used Adam optimizer with learning rate of 1×10^{-5}, on a server with 2 CPUs of Intel(R) Xeon(R) Golden 2.40GHz and 4 GPUs of Nvidia A800.

6.1.3 Evaluation Metrics. Given a program P, a statement s_i and the program state σ_{i-1} before the execution of s_i are the input to Reasoner $\mathcal{M}$, which produces the predicted next statement $\hat{s}_{i+1}$ and updated program state $\hat{\sigma}_i$; formally, $\mathcal{M}(P, s_i, \sigma_{i-1}) = (\hat{s}_{i+1}, \hat{\sigma}_i)$. If there are N instances in the test set, *next-statement accuracy* can be defined as $A_{\text{NS}} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(s_{i+1} = \hat{s}_{i+1})$, *program state accuracy* as $A_{\text{PS}} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(\sigma_i = \hat{\sigma}_i)$, and a combined *single-statement accuracy* as $A_{\text{NS+PS}} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}((s_{i+1}, \sigma_i) = (\hat{s}_{i+1}, \hat{\sigma}_i))$. Here, s_{i+1} and σ_i are the actual next statement and the updated program state following the execution of s_i within P, respectively.

6.2 Empirical Results

6.2.1 Comparing Performance in Predicting Execution Semantics. Table 2 shows the results in predicting the execution semantics for a given statement. The 14B-Instruct variant of Qwen-2.5-Coder learns to reason about program execution the best, achieving single-statement accuracies

Table 2. Comparing model performance in predicting execution semantics: next stmt. and prog. state (RQ_1).

Model	Variant	SFT	CodeNetMut				HumanEval			
			A_{NS}	A_{PS}	A_{NS+PS}	Time (s)	A_{NS}	A_{PS}	A_{NS+PS}	Time (s)
DeepSeek	Distill-14B	–	32.6	32.3	29.8	56.2	25.0	18.6	16.0	72.8
GPT	4o-mini	–	51.2	58.2	43.3	7.4	47.8	42.7	32.2	7.6
	4o	–	76.6	79.4	69.2	5.2	83.5	77.9	70.9	5.4
CodeLlama	7B-Instruct	✗	22.3	17.2	9.4	6.3	28.5	14.9	7.6	6.5
		✓	95.7	94.2	91.4	1.2	92.4	85.4	79.8	1.4
	13B-Instruct	✗	21.6	16.5	10.8	8.4	26.1	14.6	10.4	8.6
		✓	96.3	95.4	92.7	1.5	94.4	88.6	84.6	2.0
Qwen-2.5-Coder	7B-Instruct	✗	21.1	23.2	16.4	3.7	18.0	16.0	10.7	5.0
		✓	96.4	95.9	93.0	1.1	95.1	86.4	83.5	1.3
	14B-Instruct	✗	60.2	60.0	46.9	2.9	57.6	51.8	40.3	3.1
		✓	96.7	96.8	**94.2**	1.9	94.2	88.6	**85.1**	2.1

of 94.2% on CodeNetMut and 85.1% on HumanEval. In particular, it attains 96.7% *next-statement accuracy* and 96.8% *program-state accuracy* on CodeNetMut. On the more open-ended HumanEval dataset, while performance slightly declines in program state prediction with an accuracy of 88.6%, it predicts the next statement with 94.2% accuracy. Overall, these findings suggest that the learned dynamic reasoning capabilities in Qwen-2.5-Coder generalize well beyond its training distribution.

Across the board, all the fine-tuned variants of CodeLlama (7B- and 13B-Instruct) and Qwen-2.5-Coder (7B- and 14B-Instruct) outperform the baseline models, consistently achieving 91.4%–94.2% accuracy on CodeNetMut and 79.8%–85.1% on HumanEval. In particular, these fine-tuned models improve over their non-fine-tuned counterparts by 100.8%–950% in single-statement accuracy. These corroborate Key Ideas 1–2 (Section 2): grounding the LLMs in *verbalized operational rule rationales from actual execution enables them to internalize how a program state evolves during execution.*

Furthermore, larger models consistently outperform their smaller counterparts across both benchmarks, aligning with the scaling laws [24]. For instance, the 13B-Instruct variant of CodeLlama outperforms the 7B-Instruct variant by up to 1.4% on CodeNetMut and 6% on HumanEval in single-statement accuracy. Similarly, Qwen-2.5-Coder-14B also exhibits gains over its 7B counterpart, particularly in program state prediction. These improvements suggest that increased model capacity enhances code understanding as well as enables more precise reasoning over program execution.

As noted in Section 6.1.1, the execution rationales used to fine-tune all variants of CodeLlama and Qwen-2.5-Coder were distilled from GPT-4o-mini, while being grounded in actual execution traces. When compared against the same teacher model (*i.e.*, GPT-4o-mini) without such grounding, the fine-tuned models achieve 111.1%–117.6% higher accuracy on CodeNetMut and 147.8%–164.3% on HumanEval. This substantial improvement shows the benefits of *using execution-grounded explanations in our teacher-student framework* (Key Ideas 2–3, Section 2), enabling the student models to effectively learn dynamic code reasoning and outperform the teacher (in Section 10.2, we compare the quality of their explanations). Even when compared to the much stronger GPT-4o, they yield improvements of 32.1%–36.1% on CodeNetMut and 12.6%–20% on HumanEval.

Finally, we observe that despite being derived from the same Qwen-2.5-14B model, DeepSeek-R1-Distill-Qwen-14B significantly underperforms our fine-tuned Qwen-2.5-Coder-14B-Instruct model. This may be attributed to the latter's source code-focused pre-training followed by execution-specific post-training, which is better for dynamic code reasoning. These suggest that improving general reasoning ability, as in DeepSeek-R1 does not generalize effectively across domains.

In addition to improved accuracy, our models also offer *faster inference times* compared to larger general-purpose LLMs like GPT-4o-mini and GPT-4o, achieving 2.64×–5.8× speedups across both

Table 3. Stratified evaluation of model performance based on types of statements: S_1–S_5 denote 1) expressions, 2) variable assignments, 3) `if`-statements, 4) `for`/`while`-statements, and 5) method calls (RQ_1).

Model	Variant	CodeNetMut							HumanEval						
		S_1	S_2	Seq.	S_3	S_4	S_5	Branch	S_1	S_2	Seq.	S_3	S_4	S_5	Branch
DeepSeek-R1	Distill-14B	34.3	26.4	34.5	25.5	8.5	53.9	16.7	23.4	23.2	17.7	14.3	6.9	19.0	10.1
GPT	4o-mini	57.0	53.2	48.9	30.8	7.0	71.3	27.9	52.8	46.3	34.3	24.1	17.0	38.1	24.8
	4o	77.6	69.1	71.2	74.8	42.3	87.0	63.6	76.6	84.8	71.7	78.9	50.9	75.0	68.0
CodeLlama	7B-Instruct	88.7	97.1	94.5	88.5	85.6	95.7	84.9	75.9	88.4	81.4	83.4	79.7	60.7	74.3
	13B-Instruct	90.5	97.5	95.5	91.3	87.0	97.4	87.0	75.9	88.4	85.8	83.4	79.7	60.7	80.4
Qwen-2.5-Coder	7B-Instruct	92.1	97.3	95.1	92.2	85.6	97.4	87.2	87.5	92.7	82.8	89.9	71.8	91.7	86.2
	14B-Instruct	93.4	97.7	96.5	95.0	85.2	97.4	90.0	85.1	93.6	84.2	91.6	74.6	95.2	87.8

CodeNetMut and HumanEval. Notably, the inference times of the fine-tuned models is 20–30% less than the non-fine-tuned variants, as the latter typically generate excessive irrelevant outputs with high error rates. This highlights the practical utility of T-REX, offering models that are not only *more accurate in reasoning about program execution but also significantly more time-efficient.*

6.2.2 Stratified Evaluation Based on Types of Program Statements. To understand how models reason over different statement types (S_1–S_5) (presented in Table 1), we conduct a stratified evaluation.

In Table 3, we report the model performance over all statement types (S_1–S_5) across both CodeNetMut and HumanEval datasets. While CodeLlama and Qwen-2.5-Coder variants record consistently high performance across different types of sequential statements, we observe that branching statements benefit from model scaling. In particular, in CodeLlama, accuracy on branching statements improves by 2.5% in CodeNetMut (from 84.9% to 87.0%) and by 10.5% in HumanEval (from 74.3% to 80.4%) when scaling from 7B-Instruct to 13B-Instruct. A similar trend holds for Qwen-2.5-Coder, where branching accuracy improves by 3.2% on CodeNetMut and 1.9% on HumanEval as we scale from 7B-Instruct to 14B-Instruct. Notably, the 7B-Instruct variant of Qwen-2.5-Instruct is as good as the 13B-Instruct variant of CodeLlama on CodeNetMut and outperforms it by 7.2% on HumanEval for branching statements, showing the strength of Qwen-2.5-Coder's underlying code understanding. Overall, these results demonstrate that the performance gains of our fine-tuned models are *consistent across different statement types* and not limited to any specific syntactic constructs.

7 Effectiveness in Runtime Behavior Prediction (RQ_2)

We select three critical dimensions of dynamic behaviors [6]: *execution trace*, *code coverage*, and *program outputs*. These reflect the fundamental signals underlying a range of dynamic analysis tasks, including dynamic slicing and/or static detection of runtime errors (the latter, see Section 8).

7.1 Experiment Setup

7.1.1 Baselines. Apart from the LLM intrinsic baselines in Section 6, we also include two state-of-the-art dynamic code reasoning models: 1) CodeExecutor [29], built upon UniXCoder [17] and pretrained with execution traces; and SemCoder, fine-tuned on *monologues* predicted by LLMs without execution. For a fair comparison, we evaluate it in a zero-shot setting, i.e., without any additional fine-tuning. Note that SemCoder is limited to predicting final program outputs and does not explicitly model and predict intermediate execution steps, i.e., stepwise execution.

7.1.2 Evaluation Metrics. We use the following evaluation metrics:

(1) *Execution Trace.* To evaluate how closely the predicted trace matches the ground-truth, we adopt both strict and relaxed matching metrics. In particular, we report *prefix match*, which

Table 4. Comparing model performance in predicting runtime behaviors (RQ_2).

Model	Variant	CodeNetMut									HumanEval								
		Execution Trace				*Code Coverage*				*P/O*	*Execution Trace*				*Code Coverage*				*P/O*
		Prefix	$A_{0.5}$	$A_{0.8}$	$A_{1.0}$	P	R	$F1$	A_{EM}	A_{EM}	Prefix	$A_{0.5}$	$A_{0.8}$	$A_{1.0}$	P	R	$F1$	A_{EM}	A_{EM}
C/Executor.	–	20.1	14.0	4.2	3.4	60.9	83.2	67.9	19.0	4.6	14.3	6.2	2.3	1.0	49.4	74.7	57.2	10.9	2.4
DeepSeek-R1	Dist.-14B	16.7	6.8	0.0	0.0	24.5	99.6	25.7	1.2	0.9	9.2	2.1	0.3	0.3	20.9	99.7	31.5	0.6	0.5
GPT	4o-mini	41.4	33.6	18.9	16.4	64.9	97.3	73.8	23.4	14.9	24.3	13.3	5.1	4.8	64.1	94.8	73.7	15.5	7.9
	4o	50.2	50.6	26.5	16.9	71.7	97.8	78.2	27.4	16.2	33.1	25.9	11.1	5.2	70.8	97.7	77.8	20.8	5.2
CodeLlama	7B-Inst.	61.4	62.0	40.6	24.9	85.5	97.4	89.4	49.6	39.7	40.9	36.3	18.3	7.2	78.3	97.2	84.8	19.8	14.9
	13B-Inst.	67.3	69.1	47.9	41.1	87.6	98.5	91.2	55.9	51.6	48.1	43.3	29.8	16.5	83.4	97.3	88.0	34.9	23.4
Qwen-2.5 -Coder	7B-Inst.	67.3	67.1	50.6	41.8	89.7	99.0	**92.7**	**63.4**	51.8	46.0	42.2	25.4	13.1	83.5	97.9	88.2	35.0	30.1
	14B-Inst.	67.1	65.9	50.0	**42.7**	88.2	98.9	91.5	62.3	**52.3**	49.6	46.7	30.8	**18.5**	83.5	98.5	88.4	**39.3**	**32.6**

measures the proportion of statements (each consisting of a line number and its program state) that are predicted correctly and in order from the beginning of the trace. This metric is suitable for scenarios like debugging, where partial execution is still informative. We also report A_x scores ($x \in \{0.5, 0.8, 1.0\}$), which evaluates the accuracy of predicted traces at different thresholds of alignment (*e.g.*, 50% of the trace matched), providing a fine-grained perspective of performance. Among these, $A_{1.0}$ is the same as exact-match accuracy.

(2) *Code Coverage*. We report both statement-level and program-level metrics. For the former, we compute *precision* (P), *recall* (R), and *F1-score* ($F1$), assessing how correctly the model distinguishes between covered and uncovered statements across all examples. At the program-level, we report *exact match accuracy* (A_{EM}), which measures the percentage of programs for which the predicted code coverage exactly matches the ground-truth coverage across all statements. In particular, A_{EM} reflects whether the model can fully reproduce the coverage behavior of an entire program.

(3) *Program Outputs*. We report *exact match accuracy* (A_{EM}), which measures the percentage of cases where the final predicted output of the program exactly matches the true final output.

7.2 Empirical Results

Table 4 reports results in multiple runtime dimensions. First, for execution trace prediction, Qwen-2.5-Coder 14B-Instruct model achieves the best performance, exactly predicting traces in 42.7% and 18.5% of CodeNetMut and HumanEval datasets, respectively. The drop in the latter can be attributed to the execution traces in HumanEval being 2.6× longer than those in CodeNetMut on average, making exact prediction more challenging. On average, it correctly predicted 67.1% and 49.6% of each ground-truth execution trace in two datasets.

The GPT model variants perform competitively in predicting the initial execution steps (*e.g.*, Prefix and $A_{0.5}$), but struggle with predicting the later program state transitions. As noted in Section 6.2.2, this could be due to their inability to model branching statements well, leading to a performance drop in $A_{0.8}$ and $A_{1.0}$. These models rank in descending order of performance in predicting execution traces: GPT-4o > GPT-4o-mini > CodeExecutor > DeepSeek-R1.

While Section 6 focused on single-statement predictions (*i.e.*, the NS+PS setting), execution trace prediction serves as its cumulative extension, capturing model understanding of runtime behavior for multiple statements and the entire program. In a similar vein, predicting code coverage, which involves predicting whether the statements in a program shall be executed or not, can be seen as an aggregated extension of next-statement prediction (*i.e.*, the NS setting) while abstracting away execution order. Due to their better understanding of execution flow, all fine-tuned model variants exhibit high accuracy in statement coverage, observing gains of 18.5%–260.7% over the non-fine-tuned counterparts. Among these, both 7B and 14B variants of Qwen-2.5-Coder perform best, exactly predicting the code coverage in 63.4% and 62.3% of the programs, respectively, an improvement of 11.4%–27.8% over CodeLlama and 127.4%–170.9% over GPT models.

Table 5. Comparing output prediction (RQ_2)

Model	CodeNetMut	HumanEval
SemCoder	61.3%	64.4%
Qwen-2.5-Coder-14B Instruct (*non-fine-tuned*)	66.7%	58.5%
Qwen-2.5-Coder-14B-Instruct (*fine-tuned*)	68.2%	64.7%

Table 6. Comparing performance in static detection of runtime errors. Here, *Excpt* denotes *Exception* (RQ_3).

Actual (↓)	Reasoner		GPT-4o	
Predicted (→)	*Excpt*	*No Excpt*	*Excpt*	*No Excpt*
Excpt	TP=42	FN=58	TP=33	FN=67
No Excpt	FP=56	TN=44	FP=77	TN=23
Accuracy	**43%**		28%	

Table 4 shows the strength of our fine-tuned models in output prediction: they outperform GPT models and CodeExecutor on CodeNetMut, with the best-performing Qwen-2.5-Coder-14B-Instruct observing gains of 321%–1,383%. We also observed comparable improvements on HumanEval. Note that in Table 4, we aggregated single-statement predictions to derive the output prediction.

SemCoder is explicitly fine-tuned to predict only program inputs or outputs, limiting its generalization to other dimensions of runtime behavior. In contrast, the models within Reasoner, as reported in Table 4, derive runtime data by aggregating single-statement predictions to derive execution outputs. This enables broader applicability to downstream dynamic reasoning tasks. Thus, we did not compare with SemCoder in execution trace or code coverage prediction in Table 4.

Table 5 reports the output prediction comparison as we directly prompted the models for the output of a program for a fair comparison with SemCoder. Despite its targeted training, SemCoder exactly predicts program outputs for 61.3% of the programs in CodeNetMut and 64.4% of HumanEval. However, even the non-fine-tuned Qwen-2.5-Coder-14B-Instruct model achieves comparable or better performance: 66.7% on CodeNetMut and 58.5% on HumanEval. With fine-tuning through T-REX, this model achieves exact output prediction in 68.2% and 64.7% of the programs in both benchmarks, respectively. This improvement over SemCoder highlights the effectiveness of our training paradigm in generalizing across a range of runtime reasoning tasks.

Lastly, from the results in Tables 2–4, the fine-tuned 7B-Instruct CodeLlama model is significantly worse than the 13B-Instruct variant, while the 7B-Instruct and 14B-Instruct variants of Qwen-2.5-Coder are consistently similar in multiple dimensions, with the latter being better than the former.

8 Usefulness in Static Detection of Runtime Errors (RQ_3)

In this experiment, we tested Reasoner in static detection of runtime errors on 200 samples from CodeNetMut and HumanEval: 100 randomly selected non-exception samples and 100 containing exception statements in the test set (Section 6.1.1). In the case of non-exception testing samples, the prediction is considered to be correct if the model *accurately predicts all statement execution results*. For the exception testing samples, the prediction is considered to be correct if it accurately predicts *both execution results of statements before the exception statement and correctly identifies the line number and exception type of the exception-inducing statement*. In Table 6, we present the results of Reasoner and GPT-4o in statically detecting runtime errors. Here, TP and TN represent such correctly detected exception and non-exception samples, respectively, while FP and FN captures incorrect classifications. The most notable difference lies in non-exception samples: *Reasoner is better at recognizing programs that do not raise errors, leading to fewer false positives*. Moreover, it is 27.3% more precise than GPT-4o in pinpointing crash locations and the types of exceptions, resulting in a 54% boost in runtime-error detection accuracy – all without actual execution.

9 Usefulness in Aiding Debugging (RQ_4)

Given a program that crashes at a particular statement, debugging often requires developers to identify the underlying *fault statement(s)* that encapsulate the actual root cause of the error. In

contrast, the *crash statement* indicates where the interpreter fails. In this experiment, we aim to assess the usefulness of Reasoner in T-REX in aiding debugging. In doing so, it helps pinpoint the precise point in the program that needs to be fixed. To this end, we randomly selected 100 test samples as in RQ3 (Section 8). For each, we manually labeled the *fault-inducing line*, *i.e.*, the statement responsible for the observed crash. Given a model $\mathcal{M}$ and program $P_i \in \mathcal{D}$, let $\hat{T}_i$ denote the predicted execution trace and l_i the line number of the fault statement ($1 \le l_i \le |P_i|$). Here, we define accuracy as the proportion of cases where the fault line l appears in $\hat{T}$ with all the correctly predicted program states from the beginning to the faulty statement:

$$\text{Accuracy} = \frac{1}{|\mathcal{D}|} \sum_{P_i \in \mathcal{D}} \mathbf{1}\Big[\, l_i \in \hat{T}_i \;\wedge\; \big(\forall\, l_j \le l_i,\ \text{State}(l_j) \text{ is correct}\big)\Big]$$

Such a formulation measures how consistently the model "sees" the fault during its predicted execution. *An IDE could leverage such a simulated interpreter as a lightweight, execution-free substitute for an actual debugger.* This is particularly useful in settings where actual execution is not feasible.

In Table 7, we report the proportion of fault statements with all correctly predicted states that appeared in each model's predicted execution traces. We can see that Reasoner (with Qwen-2.5-Coder-14B-Instruct) achieves 77% accuracy, higher than the much larger GPT-4o (63%) and GPT-o4 mini (49%). That is, Reasoner helps pinpoint the faulty statement in 77% of the cases, showing that it not only reconstructs plausible execution paths but also preserves semantically accurate program states along those paths. Thus, it allows developers to inspect the program behaviors just as they would using a debugger.

Table 7. Comparing model performance in pinpointing fault locations, appearing in the predicted execution traces with correctly predicted program states (RQ_4).

Models	Reasoner	GPT-4o	GPT-o4 mini
Accuracy	77%	63%	49%

10 Qualitative Evaluation (RQ_5)

10.1 T-REX Scales Up Dynamic Code Reasoning

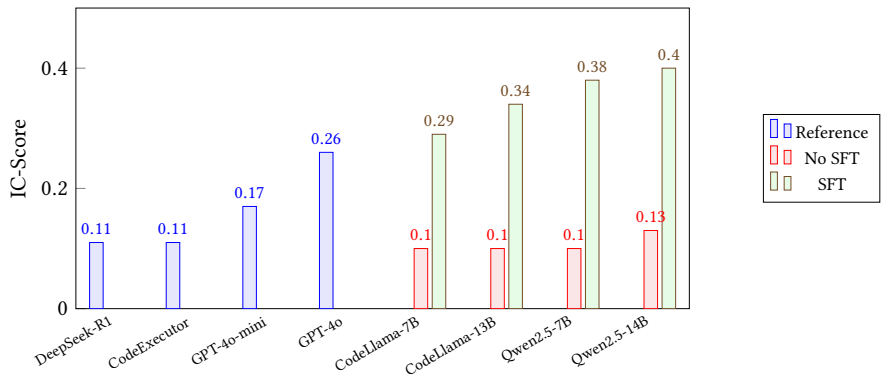
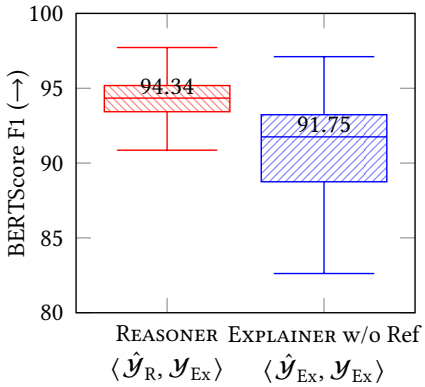


Fig. 7. Benchmarking LLMs for dynamic code reasoning using incremental consistency (IC-Score) [6] ($RQ_{5.1}$).

To evaluate the reasoning abilities in code execution prediction, Chen *et al.* [6] introduce *incremental consistency score* (IC-Score), which evaluates on a combination of sequentially-related runtime reasoning tasks of increasing difficulty. Given a program $P = \{s_1, \ldots, s_n\}$ and an input I, a model M is evaluated on its ability to predict: (1) whether a given statement s_i is executed, (2) the next statement to be executed after s_i, (3) the final output of the program, and (4) the type and value of a variable v relevant to s_i after its execution.



Measure	Average	
	Reasoner $\langle \hat{y}_R, y_{Ex} \rangle$	Explainer w/o Ref $\langle \hat{y}_{Ex}, y_{Ex} \rangle$
ROUGE-L	0.57	0.45
BLEU-4	53.0	39.0
BERTScore F1	94.3	91.7

Fig. 8. Explanation quality of Reasoner and Explainer *w/o* reference, to Explainer *w/* reference ($RQ_{5.2}$).

In this experiment, we use IC-Score to benchmark the dynamic code reasoning capabilities of T-REX and the baseline LLMs used in Section 7. Following their setup [6], we assess the models' predictions across all programs in HumanEval. As seen in Fig. 7, *those fine-tuned with execution rationales (highlighted in green) consistently outperform both their non-fine-tuned counterparts (red) as well as the reference LLMs (blue)*, including the significantly larger models such as GPT-4o-mini and GPT-4o. For example, the Qwen2.5-Coder-14B-Instruct variant achieves gains of 135.3% and 53.9% over GPT-4o-mini and GPT-4o, respectively. While the reference LLMs exhibit stronger execution reasoning than the base models without fine-tuning, the benefits of *execution-grounded, rationale-based supervision* are substantial: *fine-tuned variants show relative improvements of 190%–280% over their base versions without SFT*. These findings highlight the importance of *incorporating fine-grained, transition-aware reasoning* to enhance the LLMs' ability to reason about dynamic behaviors.

10.2 Reasoner Explains Similarly to Explainer

In this experiment, we assess the quality of Reasoner's execution rationales that guide its determination of program state transitions. For this purpose, we replicate the data collection process using Explainer on samples from the test set in Section 6, obtaining reference execution rationales. Let such rationales grounded in actual execution traces be denoted by y_{Ex}, while those from Reasoner be denoted by $\hat{y}_R$. This enables a side-by-side comparison of model-generated reasoning against reference explanations. We also collect execution rationales from Explainer when not grounded in actual execution traces (*i.e.*, without reference). Let these be denoted by $\hat{y}_{Ex}$.

In Fig. 8, we illustrate the distributions of BERTScore F1-measures [53], computed using DeBERTa-Base [18], by comparing $\langle \hat{y}_R, y_{Ex} \rangle$ (in red) and $\langle \hat{y}_{Ex}, y_{Ex} \rangle$ (in blue). This embedding-based similarity metric is useful for capturing *semantic equivalence* between execution rationale pairs. The former shows that execution rationales generated by Reasoner are closely aligned with the reference rationales. In contrast, the latter reveals that when Explainer is not grounded in actual execution information (blue box in Fig. 4), it fails to produce high-quality rationales despite being a significantly larger model (GPT-4o-mini). Overall, T-REX enables Reasoner to generate higher quality rationales, improving its effectiveness in enhancing dynamic code reasoning.

11 Static Emulation of Dynamic Analyses via Teacher–Student Distillation

In this section, we present a framework generalized from T-REX for static emulation of a dynamic analysis technique without actual execution.

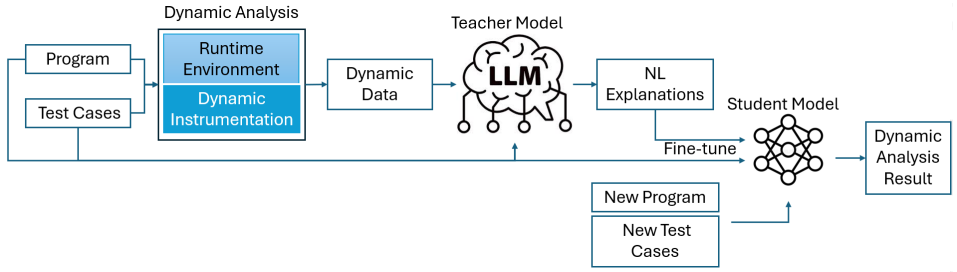


Fig. 9. Generalized T-REX Framework for Dynamic Analysis Learning

Our goal is to *teach a language model to statically perform a target dynamic analysis* (δ), i.e., to produce δ's outputs for a new program and test inputs *without executing them* by (1) collecting supervised dynamic data once from a dynamic analysis technique, (2) verbalizing *why* the analysis produced those results by an LLM, and (3) distilling that knowledge via verbalized explanations into a smaller student model. Intuitively, we train the model to "mentally run" the analysis using *verbalized operational/analysis rules*, extending the T-REX philosophy from execution semantics to a new dynamic analysis.

Algorithm 2. Static Emulation of Dynamic Analysis δ with T-REX

Require: Code corpus, test cases $C = \{(P, T)\}$, target dynamic analysis δ
Ensure: Student model S_ϕ predicts δ from code and test cases (w/o running)

procedure TRAIN(δ)
  $\mathcal{D} \leftarrow \emptyset$ ▷ training triples
  for all $(P, T) \in C$ **do**
    $D \leftarrow \text{EXECDELTA}(P, T, \delta)$ ▷ instrumented run once
    $X \leftarrow \text{TEACHEREXPLAIN}(P, T, D)$ ▷ NL rationales for δ
    $C \leftarrow \text{BUILDCONTEXT}(P, T)$ ▷ code, tests, light static cues
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(C, X, \text{TARGETSANALYSIS}(D))\}$
  $S_\phi \leftarrow \text{FINETUNE}(\mathcal{D})$ ▷ optimize rationale and structured losses
  return S_ϕ
procedure INFER(S_ϕ, P', T')
  $C' \leftarrow \text{BUILDCONTEXT}(P', T')$
  $(\widehat{D}, \widehat{X}) \leftarrow S_\phi(C')$ ▷ static emulation of δ
  return $(\widehat{D}, \widehat{X})$

11.1 Framework and Workflow

Let P be a program, T a set of test cases, and δ a dynamic analysis (*e.g.*, program slicing, taint, aliasing). The workflow (Fig. 9) has four stages. Our workflow is described in Algorithm 2:

11.1.1 Runtime Collection. We execute P under T inside an instrumented runtime for (δ):

$$D = \text{Exec}_\delta(P, T) \in \mathcal{D}_\delta,$$

where D is the *dynamic data* for (δ) (e.g., trace with dependence edges, program slices, etc.).

11.1.2 Verbalization by Teacher LLM. Given (P, T, D), a teacher ($\mathcal{T}_\Theta$) produces *natural-language rationales* that explain (δ)'s outputs in terms of local rules:

$$X = \mathcal{T}_\Theta(P, T, D) \in \mathcal{X}_\delta.$$

Note that the rationales align with small-step, transition-aware rules (e.g., *"the use of (x) at line (i) depends on the last definition at line (j) observed in (D)"*) as in execution semantics in Section 3.1.

11.1.3 Distillation into a Student Model. We fine-tune a student model ($\mathcal{S}_\phi$) on pairs (C, X), where C is a static context built from (P, T) (source code and test cases) and X are the teacher's rationales:

$$\phi^\star = \arg\min_\phi \mathbb{E}_{(P,T)} \left[-\log p_\phi\big(X \mid C(P, T)\big) \right].$$

Optionally, we jointly predict δ's structured outputs $\widehat{D}$ (e.g., edges/slices), with a multi-task loss:

$$\mathcal{L}(\phi) = \lambda_{\text{NL}} \cdot \mathcal{L}_{\text{rationale}} + \lambda_{\text{struct}} \cdot \mathcal{L}_{\text{struct}}.$$

11.1.4 Static Emulation at Inference. For a new program P' and tests T', the student produces $\widehat{D}$, which is the ouput of δ on P' and T', without execution:

$$\widehat{D} = \mathcal{S}_{\phi^\star}\big(C(P', T')\big),$$

and may also emit concise rationales ($\widehat{X}$) for interpretability.

11.2 Case Study: Static Emulation of Dynamic Program Slicing for Python

11.2.1 Formulation. Given an execution trace ($\tau = \langle s_1, \ldots, s_m \rangle$) under input ($t \in T$), a slicing criterion is ($\kappa = (k, V)$) with step index k and variable set V. The dynamic slice ($\text{Slice}_\tau(\kappa) \subseteq 1, \ldots, k$) is the set of statement instances whose executed definitions or control decisions actually influenced the values of V at step k.

We model a dynamic dependence graph (DDG) over executed instances $1..m$ with edges:

Data dependence ($j \xrightarrow{\text{data}} i$): instance j defines x and instance i reads that x on this run.

Control dependence ($j \xrightarrow{\text{ctrl}} i$): the execution of i is conditionally enabled by the outcome of a predicate at j on this run (e.g., taken branch).

$$\text{Slice}_\tau(\kappa) = \left\{ i \;\middle|\; i \text{ reaches } k \text{ via } \left(\xrightarrow{\text{data}} \cup \xrightarrow{\text{ctrl}} \right)^* \text{along variables } V \right\}.$$

11.2.2 Small-Step Program Semantics Rules. Let us illustrate a few important small-step semantic rules for dynamic slicing for Python code. These rules mirror T-REX's transition-aware alignment except that each executed step emits precise edges grounded in the local rule.

Below, σ is the store, ($\text{lastDef}_\tau(x, i)$) is the last executed instance ($j < i$) that wrote x. We annotate each executed instance with dynamic edges emitted during the step.

(1) DS-Assign Rule: `x = expr`

$$\sigma \vdash \texttt{expr} \Downarrow v, \qquad \langle \texttt{x=expr}, \sigma \rangle \Rightarrow \langle s_{\text{next}}, \sigma[x \mapsto v] \rangle,$$

emit data edges from every variable y read in `expr` to this instance i: $\text{lastDef}_\tau(y, i) \xrightarrow{\text{data}} i$.

(2) DS-Use Rule: Use in expression `y + z` at instance i.

For each read variable $u \in \{y, z, \ldots\}$: $\text{lastDef}_\tau(u, i) \xrightarrow{\text{data}} i$.

(3) DS-If-True / DS-If-False Rule: `if cond: S1 else: S2 executed at j`

Evaluate `cond` under σ. If True, control edges $j \xrightarrow{\text{ctrl}} i$ for each instance i executed within S_1 on this run; similarly for False and S_2. Also add data edges from definitions used in `cond` to j.

(4) DS-While Rule: `while cond: B at j`

Each iteration evaluates `cond`. For any instance i inside the iterations taken, add $j \xrightarrow{\text{ctrl}} i$; add data edges into j from the variables read in `cond`.

(5) DS-Call / DS-Return Rule: `x = f(a,b) at call site (c) and return r at (rtn)`

Data edges. From the actual's last defs to the entry of f; inside f, standard rules apply; at return, emit data edge $rtn \xrightarrow{\text{data}} c$ for x.

Control edges. the body's executed instances are control-dependent on the call site c.

(6) DS-Exception Rule: `'raise e' or runtime error at j`

Terminate the current region; any handler that actually caught the exception gets control-dependence from j.

11.2.3 Illustration: Code → Applied Rule → Verbalized Rationale. Let us use the examples in Python to illustrate the rules, the corresponding code examples, and the verbalized rationales/explanations that we could distill from a LLM.

(1) DS-Assign Rule: `x = y + 1`
At this step, x is assigned from the current value of y; therefore the last executed definition of y dynamically data-depends into this assignment.
(2) DS-If-True Rule: `if n % 2 == 0: parity = "even" else: parity = "odd"`
Because ‘n % 2 == 0‘ evaluated True, all instances inside the then-block are control-dependent on this condition; the condition data-depends on the last definition.
(3) DS-While Rule: `while i < n: s = s + a[i] i += 1`
Each executed body instance is control-dependent on the loop guard $i < n$ evaluated True for that iteration; uses of i, n, and $a[i]$ create data-dependence back to their last definitions.
(4) DS-Call/Return Rule: `def inc(z): return z + 1  x = inc(y)`
The call transfers the current value of y to z (data edge from y's last def to inc's entry). The return computes $z + 1$; the return instance data-depends on z and defines x at the call site.
(5) DS-Exception Rule: `r = 10 / d # may divide by zero`
If a `'ZeroDivisionError'` occurs, the step terminates abnormally; the handler (if any) is control-dependent on this event; d's last definition data-depends into the failing expression.

Learning objective. The student LLM predicts both the membership $\widehat{\chi} * i \in 0, 1$ for each executed instance i (or each static line approximating i), and the rationales $(\widehat{X})$:

$$\min_{\phi} \underbrace{\sum_{i} \mathrm{CE}\big(\widehat{\chi}_i, \mathbf{1}[i \in S^*]\big)}_{\mathcal{L}_{\text{slice}}} + \lambda \cdot \underbrace{\big(-\log p_\phi(X \mid C)\big)}_{\mathcal{L}_{\text{rationale}}}.$$

Static emulation at inference. Given (P', T', κ), the student LLM produces $(\widehat{S} = i : \widehat{\chi}_i = 1)$ and a compact explanation $(\widehat{X})$ citing DS-rules:

$$(\widehat{S}, \widehat{X}) = \mathcal{S}_{\phi^\star}\big(C(P', T', \kappa')\big).$$

12 Limitations and Threats to Validity

Our current model works only on Python. We need different execution semantics for other languages (Section 11). While SIPA works with CodeNetCodeMut and HumanEval, results may be different when handling intricate logic, lengthy loops, or multiple nested branches/loops in different datasets. Complex objects in parameter passing between statements or methods are handled via serialization/deserialization. Moreover, the integration with development workflows (debugging, continuous integration) is desirable. Finally, for web programming (with declarative web languages) or system programming (with event-driven programming), we need different execution models.

We evaluated T-REX mainly on Python programs, where execution prediction can be precisely validated against ground-truth traces. Performance on large codebases may vary. First, T-REX performs execution prediction for each statement at a time using a smaller LLM within the context of the current executed method. That is, it does not need to load the entire project's codebase. The methods are loaded one-by-one according to the method invocations. In other words, T-REX can naturally extend to large systems through compositional analysis, caching intermediate representations, or integrating with dependency and call graphs, suggesting practical scalability paths rather than inherent limits. Second, T-REX advances the state of the art in execution prediction by proving useful across multiple downstream tasks, e.g., code coverage estimation, output prediction, static runtime error detection, and debugging where local execution reasoning is as crucial as scalability.

The store for variables' values is realized similarly to the symbol table in Python interpreters, with supports for nested scopes to manage/resolve variables/names across different scope levels, and the references to the values. For complex objects, we serialized these values and stored them in a canonical form that mimics the memory for easy processing. We used `python-graphs` to build AST/CFG/PDG for individual functions, and connected the CFGs/PDGs from the callers' to callees' nodes via the symbol table as in the interpreters.

Our dataset may be public, but few execution traces are available, and existing inputs are neither exhaustive nor representative. The program 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, we posit that data leakage is unlikely. In Fig. 7 , the results for DeepSeek-R1, CodeExecutor, and GPT-4o are different from those reported in Chen *et al.* [6] since we performed stepwise statement execution (which aligns with T-REX). In contrast, they fed entire programs to the models to prompt for the traces, coverages, etc. We present our generalized framework on a theoretical ground, but it requires additional empirical experiments for validating the applicability of our solution to different concrete dynamic analysis techniques.

13 Related Work

Execution Prediction Approaches. Execution prediction approaches can be grouped into following categories. First, the mappings between the code representation and their outputs are learned by training an LSTM [52]. Graph neural networks (GNNs) are used for this task [4, 48] in which an attention-based GNN derived from a control flow predicts outputs.

The second category involves *pre-training language models* to incorporate execution semantics. CodeExecutor [29] introduces a pre-training strategy to include execution traces. TRACED [11], based on BERT/RoBERTa, is pre-trained on variables' value ranges and code coverage. With the success of Transformer [45], Transformer-based models [8, 29, 35, 50] have treated execution prediction as a sequence-to-sequence task. LExecutor [42] executes any (in)complete code in an under-constrained manner, instead of prediction. It utilizes a neural model to predict missing values.

Third, several work [14, 25] has simulated code execution using LLMs. Tufano *et al.* [44] use vanilla prompting to GPT-4 for code coverage prediction. NeXT [33] integrates Chain-of-Thought reasoning with execution traces inserted as inline comments. It fine-tunes an LLM as it naturalizes raw execution traces into natural-language rationales. However, NeXT annotates each executed line with variable values in the original code order rather than the actual execution order, which are different for loops. This may confuse the model. Additionally, it provides only the input and output values of variables without explaining the underlying execution semantics. Ultimately, it targets a different problem: NExT is an execution-aware program repair method trained and validated with runtime traces and unit-test oracles. The execution data supports the model in program repair, but it does not aim to predict execution traces. We did not perform an empirical comparison with NeXT as its fine-tuned LLM and code are not publicly available. Finally, reproducing NeXT's fine-tuning of an LLM would require substantial computational resources for large-scale fine-tuning.

CodePilot [9] and Orca [37] are centered on statement-specific semantics, guiding a LLM to autonomously plan an execution. Lyu et al. [32]'s Iterative Instruction Prompting executes code line-by-line. However, it struggles with interprocedural execution. See more related works in Section 1.

Static Analysis. Classical static analysis techniques are typically designed to estimate program properties across all possible inputs. While they can offer some forms of guarantees (e.g., soundness) and detect potential errors, they often rely on conservative over-approximations. This typically results in high false-positive rates, particularly for highly dynamic languages like Python (e.g., dynamic typing, reflection, first-class functions) where precise static modeling is challenging. In contrast, we positions T-REX as a data-driven *static emulation* approach that *is complementary to*

static analysis. Instead of seeking input-independent guarantees using engineered abstract domains, T-REX predicts the most likely *concrete, per-input* execution. This allows for the emulation of dynamic behaviors and the generation of step-by-step rationales for the likely runtime errors, a task that is difficult for classic static analysis which lacks tracking of per-input execution traces.

Alternative Designs. We also consider alternative designs and position T-REX accordingly. **1) Single end-to-end models** are trained directly on execution transitions and predict outputs without producing rationales (e.g., CodeExecutor [29]). While this is a natural baseline, our results (Section 6) showed that *execution-rationale post-training* is a major contributor to the gains over such "no-SFT" models (including CodeExecutor [29]), indicating that the explicit structured supervision in T-REX is critical. **2) Retrieval-augmented reasoning** can inject past traces or rules at inference time, but shifts complexity to deployment. While being complementary to T-REX, it requires maintaining and querying large external stores. In contrast, we *distill* trace-derived knowledge into model parameters to keep deployment self-contained. **3) Hybrids with static analysis** can be powerful, but would confound our goal of isolating what an LLM can learn from execution supervision alone; we thus keep static analysis lightweight (syntactic), making T-REX complementary to static tools rather than a replacement. In future, other static analyses (CFGs/slices/types) can be incorporated.

14 Conclusion

Novelty and Significance. We tackle execution prediction from a novel angle by grounding LLMs in **actual execution data** and **transition-aware execution semantics** instead of traditional input-output mappings. In our knowledge distillation, the teacher model generates execution rationales as training instances that illustrate how individual statements execute according to the applicable rules in the operational execution semantics. We then fine-tune a smaller model using these execution rationales to better predict dynamic program behavior. We found that execution rationales generated by a large LLM –grounded in actual execution data and transition-aware semantics – yield significantly more effective training than unguided monologues produced without such grounding. Importantly, with such training, Reasoner performs even better than Explainer (without fine-tuning) in the code execution prediction tasks.

Technically, Reasoner is a Python predictive interpreter that statically simulates execution when running code is unsafe, costly, or infeasible. For example, on online code snippets it can flag likely runtime errors without integrating or sandboxing the code. For test-case prioritization, Reasoner can estimate coverage for candidate tests and rank them without actually executing them, avoiding upfront execution cost. In an IDE, Reasoner can act as a static execution predictor—surfacing expected state transitions and next statements—to support manual debugging without a live run.
Impact. T-REX lays the groundwork for teaching operational semantics beyond execution prediction, enabling the training of smaller models on other dynamic behaviors. For example, dynamic analyses—such as dynamic slicing or vulnerability detection—can produce structured explanations that serve as supervision to fine-tune compact LLMs to reason about complex runtime phenomena (e.g., slices, exception flows, runtime errors, vulnerabilities). Our results show that T-REX effectively learns to detect runtime exceptions, highlighting its potential as a foundation for integrating operational semantics into LLM-based reasoning across diverse dynamic analyses. T-REX could work for incomplete/non-executable code. That said, more empirical studies are needed to assess generality and robustness. In addition, extending the framework to hardware-coupled runtime environments remains challenging and will likely require adapted execution semantics.

15 Data Availability Statement

Data and code is available at https://anonymous.4open.science/r/T-REX-4622.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[2] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590* (2021).

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2019. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems* 33 (2019), 8626–8637.

[5] Angelica Chen, David Dohan, and David So. 2024. EvoPrompting: language models for code-level neural architecture search. *Advances in Neural Information Processing Systems* 36 (2024).

[6] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning runtime behavior of a program with llm: How far are we?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 140–152.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. 2018. Universal transformers. *arXiv preprint arXiv:1807.03819* (2018).

[9] 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

[10] Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail E. Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. SemCoder: Training Code Language Models with Comprehensive Semantics Reasoning. In *NeurIPS*.

[11] 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 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[12] 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.

[13] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[14] Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. 2023. Looped transformers as programmable computers. In *International Conference on Machine Learning*. PMLR, 11398–11442.

[15] GPT-4o [n. d.]. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.

[16] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065* (2024).

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[18] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2021. Deberta: decoding-Enhanced Bert with Disentangled Attention. In *ICLR*. OpenReview.net.

[19] Namgyu Ho, Laura Schmid, and Se-Young Yun. 2023. Large Language Models Are Reasoning Teachers. In *ACL (1)*. Association for Computational Linguistics, 14852–14882.

[20] 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

[21] Raisa Islam and Owana Marzia Moushi. 2024. GPT-4o: The Cutting-Edge Advancement in Multimodal LLM. *Authorea Preprints* (2024).

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).

[23] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024),

1–30.

[24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[25] Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074* (2024).

[26] Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. CodeI/O: Condensing Reasoning Patterns via Code Input-Output Prediction. *CoRR* abs/2502.07316 (2025).

[27] Yi Li, Hridya Dhulipala, Aashish Yadavally, Xiaokai Rong, Shaohua Wang, and Tien N. Nguyen. 2025. Blended Analysis for Predictive Execution. In *Proceedings of the 2025 ACM International Symposium on the Foundations of Software Engineering (FSE'25)*.

[28] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's Verify Step by Step. *arXiv preprint arXiv:2305.20050* (2023).

[29] 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. *arXiv preprint arXiv:2305.05383* (2023).

[30] Changshu Liu, Shizhuo Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024. Can Large Language Models Reason About Code? *arXiv preprint arXiv:2402.09664* (2024).

[31] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS Datasets and Benchmarks*.

[32] Chenyang Lyu, Lecheng Yan, Rui Xing, Wenxi Li, Younes Samih, Tianbo Ji, and Longyue Wang. 2024. Large Language Models as Code Executors: An Exploratory Study. *arXiv preprint arXiv:2410.06667* (2024).

[33] 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 International Conference on Machine Learning (ICML)*.

[34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint* (2022).

[35] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114* (2021).

[36] Bhrij Patel, Souradip Chakraborty, Wesley A Suttle, Mengdi Wang, Amrit Singh Bedi, and Dinesh Manocha. 2024. AIME: AI System Optimization via Multiple LLM Evaluators. *arXiv preprint arXiv:2410.03131* (2024).

[37] Smit Patel, Aashish Yadavally, Hridya Dhulipala, and Tien Nguyen. 2025. Planning a Large Language Model for Static Detection of Runtime Errors in Code Snippets . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 639–639. doi:10.1109/ICSE55347.2025.00102

[38] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25 (2017), 83–110.

[39] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[40] 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

[41] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. 2024. Mathematical discoveries from program search with large language models. *Nature* 625, 7995 (2024), 468–475.

[42] 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

[43] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[44] Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement. 2023. Predicting code coverage without execution. *arXiv preprint arXiv:2307.13383* (2023).

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[46] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2022. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples. *IEEE Trans. Software Eng.* 48, 5 (2022), 1497–1514.

[47] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 1069–1088. doi:10.18653/V1/2023.EMNLP-MAIN.68

[48] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[49] 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 *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[50] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. 2020. Neural execution engines: Learning to execute subroutines. *Advances in Neural Information Processing Systems* 33 (2020), 17298–17308.

[51] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[52] Wojciech Zaremba and Ilya Sutskever. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

[53] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. In *ICLR*. OpenReview.net.

Received 2026-03-18; accepted 2026-06-10