Treefix: Enabling Execution with a Tree of Prefixes

Beatriz Souza
University of Stuttgart
Germany
beatrizbzsouza@gmail.com

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

***Abstract*—The ability to execute code is a prerequisite for various dynamic program analyses. Learning-guided execution has been proposed as an approach to enable the execution of arbitrary code snippets by letting a neural model predict likely values for any missing variables. Although state-of-the-art learning-guided execution approaches, such as LExecutor, can enable the execution of a relative high amount of code, they are limited to predicting a restricted set of possible values and do not use any feedback from previous executions to execute even more code. This paper presents Treefix, a novel learning-guided execution approach that leverages LLMs to iteratively create code prefixes that enable the execution of a given code snippet. The approach addresses the problem in a multi-step fashion, where each step uses feedback about the code snippet and its execution to instruct an LLM to improve a previously generated prefix. This process iteratively creates a tree of prefixes, a subset of which is returned to the user as prefixes that maximize the number of executed lines in the code snippet. In our experiments with two datasets of Python code snippets, Treefix achieves 25% and 7% more coverage relative to the current state of the art in learning-guided execution, covering a total of 84% and 82% of all lines in the code snippets.**

I. Introduction

Executing code is essential for reasoning about the runtime behavior of code, e.g., in a dynamic program analysis, when extracting runtime data to train a model, or when trying to understand code during manual debugging. However, getting code to actually run is challenging, both at the small and the large scale. At the small scale, individual code snippets, e.g., found in documentation or online forums, may contain undefined variables, functions, or classes, which prevent the code from executing. At the large scale, setting up a complex project is often difficult due to the diversity of build systems, missing dependencies, configuration issues, etc. Even when a project is perfectly set up, executing a specific code location will require a specific set of inputs, which may not be available.

To enable the execution of arbitrary code snippets, either stand-alone snippets that simply are not executable on their own or code snippets extracted from a larger project, researchers have proposed *learning-guided execution* [1]. The basic idea to predict likely values for any missing variables in a code snippet with a neural model, preventing the execution from getting stuck. Learning-guided execution has numerous applications because it enables dynamic analysis of code snippets in isolation. Since its introduction in 2023, the community has started to explore several applications, such as detecting runtime type errors [2], reproducing bugs [3], and checking whether a code change preserves the semantics [4]. Learning-guided execution may also serve as a mechanism for validating code or code changes produced by an auto-programming technique, e.g., LLM-based code completion [5], [6] and code editing [7]–[9]. The ability of learning-guided execution to execute arbitrary code without requiring the full setup of the project should make it relatively easily applicable to a wide range of codebases.

While the current state of the art in learning-guided execution, LExecutor [1], can enable the execution of a relative high amount of code, it suffers from two key limitations: (1) LExecutor predicts values sampled from a *limited set of runtime values*. Specifically, their neural model predicts one out of 23 abstract values, such as "non-empty string", which then are concretized into a hard-coded concrete value, such as `"a"`. These values may not match realistic values, as they would occur in a real execution of the given code snippet, and they may not be diverse enough to reach all branches. (2) LExecutor is designed for and evaluated on the task of executing a code snippet *once*, which may not be sufficient to cover all branches. Furthermore, executing the snippet only once prevents the approach from leveraging feedback from previous executions to improve the environment in which the given code snippets gets executed. Overall, these two limitations curtail the effectiveness of the state-of-the-art approach at covering the lines in a given code snippet.

This paper presents Treefix, a novel learning-guided execution approach that introduces several ideas. First, instead of training a custom model to predict abstract values, Treefix builds upon a large language model (LLM) to predict code that constructs concrete values. We refer to the code that constructs these values as *prefixes*, because the code gets prepended to the given code snippet before executing them together. By generating code prefixes, Treefix can create a much larger and more diverse set of values, including domain-specific strings, complex objects, and even values returned from third-party libraries. Second, Treefix reasons about the problem in a multi-step fashion, where each step uses feedback about the code snippet and its execution to instruct the LLM to improve a previously generated prefix. The prefixes generated by Treefix form a tree, where each node represents a prefix and an edge represents a refinement of a prefix in the next step. Finally, Treefix produces not only a single execution, but also returns a minimal set of prefixes that maximize the number of

cumulatively executed lines in the code snippet.

The approach consists of three fully automated steps, which are designed to mimic the way a human may approach the problem of enabling the execution of a code snippet.

- Step 1: Statically identify undefined references in the given code snippet and then query the LLM for prefixes that initialize them.
- Step 2: Execute the code snippet with the prefixes, observe any runtime errors that may occur, and then query the LLM for refined prefixes that address these errors.
- Step 3: Execute the code snippet with the refined prefixes, keep track of lines that are not yet covered, and then query the LLM for prefixes aimed at covering these lines.

We evaluate Treefix by applying it to two sets of code snippets from prior work [1]: functions extracted from popular open-source projects and code snippets extracted from Stack Overflow posts. As baselines, the evaluation compares Treefix with six alternative approaches for executing arbitrary code snippets: regular execution, a state-of-the-art type predictor [10], a state-of-the-art function-level test generator [11], LExecutor [1], Incompleter [3], and SelfPiCo [2]. We show that Treefix enables the execution of 84% and 82% of all lines in the open-source code and Stack Overflow snippets, respectively, which improves over the best baseline by 25% and 7%. Moreover, we show that each step of our approach contributes to its effectiveness and that Treefix produces, on the evaluated snippets, 16528 unique values, which is 16505 more values than LExecutor. In case studies we find that Treefix's ability to generate adequate imports, complex objects, and diverse primitive values contribute the most to Treefix's improvements over LExecutor.

In summary, our contributions are as follows:

- We propose Treefix, a multi-step learning-guided execution approach leveraging LLMs for enabling code execution.
- Through experimentation on two datasets, we show that Treefix can substantially improve code coverage and is superior to the state-of-the-art approaches.
- We release our open-source implementation of Treefix upon acceptance of the paper.

II. Motivating Example

To motivate our work and illustrate the challenges of enabling code execution, consider the example in Figure 1a. The code could be part of a complex project or a code snippet extracted from an online resource. Our goal is to execute this code snippet while covering as many of its lines as possible. However, executing this code snippet is challenging due to various missing pieces of contextual information. At first, the code tries to read the attribute `self.user_type`, but `self` does not exist, which will cause the code to crash. In case `self` and `self.user_type` were defined, the code would try to call the function `get_register_func`, which is also undefined, as are the attributes `self.name` and `self.alias`. Beyond undefined variables and functions, covering all code lines in this snippet is even more challenging due to the multiple

```python
try:
    register = get_register_func(self.user_type)
    if register is not None:
        self.email = register(self.name, self.alias)
        if '@' in self.email:
            result = 0
        else:
            result = -1
    else:
        result = -2
except SystemExit:
    result = 1
```

(a) Python code to execute.

```python
get_register_func = DummyObject
self = DummyObject()
user_type = "a"
name = "a"
alias = "a"
```

(b) Values predicted by LExecutor [1].

```python
def dummy_register_func(name, alias):
    return f'{name}@example.com'
get_register_func = dummy_register_func
self = type('Mock', (object,), {
    'user_type': 'standard', 'name': 'JohnDoe',
    'alias': 'jdoe', 'email': None})()
```

(c) Prefix from step 1 of Treefix (undefinedness guidance).

```python
def dummy_register_func(user_type):
    def register(name, alias):
        return f'{name}@example.com'
    return register
get_register_func = dummy_register_func
self = type('Mock', (object,), {
    'user_type': 'standard', 'name': 'JohnDoe',
    'alias': 'jdoe', 'email': None})()
```

(d) Prefix from step 2 of Treefix (error guidance).

```python
class MockRegisterFunction:
    def __call__(self, name, alias): return ''
def get_register_func(user_type):
    return MockRegisterFunction()
self = type('MockSelf', (object,), {
    'user_type': 'mock', 'name': 'test_name',
    'alias': 'test_alias'})()
```

(e) One prefix of step 3 of Treefix (coverage guidance).

```python
def get_register_func(user_type): return None
self = type('Mock', (object,), {
    'user_type': 'invalid', 'name': 'John Doe',
    'alias': 'jdoe', 'email': ''})()
```

(f) Another prefix of Step 3 of Treefix (coverage guidance).

```python
class MockRegisterFunction:
    def __call__(self, name, alias): raise SystemExit
def get_register_func(user_type):
    return MockRegisterFunction()
self = type('MockSelf', (object,), {
    'user_type': 'system_exit', 'name': 'test_name',
    'alias': 'test_alias'})()
```

(g) Yet another prefix of Step 3 of Treefix (coverage guidance).

Fig. 1: Example of code to execute and predicted values.

branches and the `try` and `except` blocks. To cover all lines, a single execution is not enough: A non-exceptional execution would miss out on the code in the `except` block, while an exceptional execution would miss out on any lines after the line that raises an exception in the `try` block.

The current state-of-the-art approach for executing arbitrary Python code snippets is LExecutor [1], which has introduced the concept of learning-guided execution. To enable the execution of arbitrary code snippets, LExecutor first trains a neural model that learns from real executions what values are typically used in a given code context. LExecutor then modifies the code snippet so that whenever it would usually refer to a non-existing value and crash, the code instead queries the model to predict the most likely kind of value. The approach then concretizes this kind of value, injects it into the running code, and continues the execution. For example, a predicted kind of value could be "non-empty string", which LExecutor concretizes into a string `"a"`.

While LExecutor enables the execution of a relative high amount of code, it suffers from two keys limitations:

- *Restricted set of runtime values*. The kinds of values that LExecutor predicts are restricted to a fixed set of 23 abstractions, such as "non-empty string", "negative integer", and "empty list". For each of them, LExecutor has a single concrete value, such as `"a"`, `-1`, or `[]`, respectively. That is, the injected values may not match realistic values, as they would occur in a real execution of the given code snippet, and they may not be diverse enough to reach all branches. For the example in Figure 1a, in case LExecutor predicts that the value assigned to `self.email` is a "non-empty string", the concrete value assigned to `self.email` would be `"a"`. In this case, it will not be able to execute the code in the second `if` statement, which checks if `"@"` is in `self.email`. Moreover, `"@"` would not be present in any other concrete value used by LExecutor.
- *Single execution*. LExecutor is designed for and evaluated on the task of executing a code snippet once, which may not be sufficient to cover all branches. In particular, any code snippet that contains mutually exclusive branches, such as two `if-else` branches without any loop or recursion around it, such as in Figure 1a, cannot be fully covered in a single execution.

Figure 1b shows the values predicted by LExecutor for the code snippet in Figure 1a. Their approach predicts the return value of `get_register_func` to be a "callable", and hence, its concrete value is a `DummyObject` class, which is assigned to the `register` variable. As `register` is not `None`, the execution proceeds to the assignment of `self.email`. LExecutor predicts `self` to be an "object", i.e., the concrete value is an instance of the `DummyObject` class. Moreover, LExecutor predicts `self.user_type`, `self.name`, and `self.alias` to be "non-empty strings", so they all get the concrete value `"a"`. When `register` is called, a `DummyObject` instance is assigned to `self.email`. This way, when the code tries to check if `"a"` is in `self.email`, a "TypeError: 'DummyObject' object is not iterable" exception is raised, crashing the code execution. As a result, only the first three lines in the `try` block are executed, but neither any of the remaining branches nor the `except` block, giving a line coverage of only 30%.

III. Approach

The following presents Treefix, a multi-step, LLM-based approach to enable the execution of arbitrary code snippets. We address the limitations of the state-of-the-art learning-guided execution approach, LExecutor, as follows. Instead of predicting a restricted set of possible values, we leverage LLMs to predict code prefixes that can produce arbitrary values, such as domain-specific strings, complex objects, and even values returned from third-party libraries. By prepending such a code prefix to the given code snippet, Treefix significantly increases the likelihood of executing the code snippet with realistic values that reach branches guarded by non-trivial conditions. To address the limitation of using only a single execution, our approach creates a set of prefixes that iteratively increase the number of executed lines in the code snippet. Finally, the approach yields a set of prefixes that complement each other in terms of coverage, e.g., by executing different branches of the code snippet.

A. Problem Statement

Before presenting Treefix, we start by defining the problem that we address. The input to our approach is a syntactically valid piece of code, which we refer to as a *code snippet* s. The goal is to execute the code snippet, one or more times, to maximize the number of successfully executed lines. To enable the execution of the code snippet, we generate one or more *prefixes* p, where a prefix is a syntactically valid piece of code that consists of two parts: a possibly empty list of import statements and a possibly empty list of assignment statements that initialize variables used in s.

Because a single prefix p may be insufficient to execute all lines in s, we aim to generate a set P of prefixes that maximizes the number of executed lines in s. We refer to those lines of s that are executed when running $p + s$ as the *coverage* achieved by p, and to those lines of s that are executed when running $p+s$ for each $p \in P$ as the *cumulative coverage*. Using this terminology, the goal is to maximize the cumulative coverage of s by generating the prefixes P, where one $p_{best} \in P$ will have the highest coverage achieved by any individual prefix. The set P and the single-best prefix p_{best} are useful for different usage scenarios. For example, p_{best} can be used to understand the execution of s, e.g., by inspecting the execution in an interactive debugger. Instead, P can be used to dynamically analyze different executions of s, e.g., to detect behavioral anomalies.

While related, the problem addressed here differs from fuzzing [12]–[14] and test case generation [11], [15], [16]. One difference is the assumptions about the given code to execute. Both fuzzing and test case generation assume to have a complete and fully installed project, i.e., without any missing code and with all dependencies set up and ready to run. In

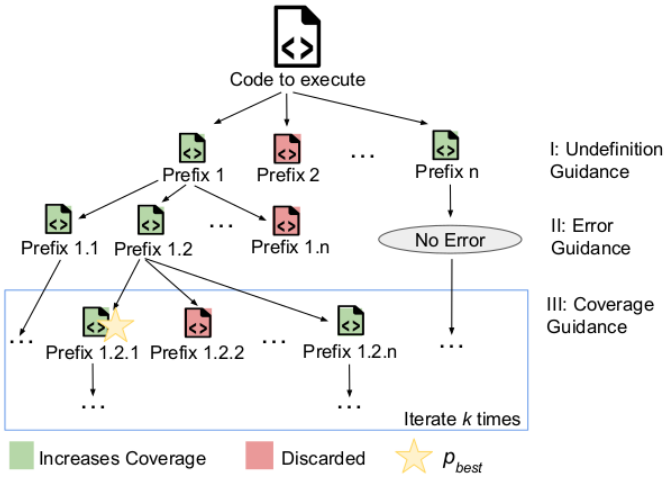


Fig. 2: Tree of prefixes generated in different steps of Treefix. Prefixes highlighted in green are added to P. The best prefix p_{best} is highlighted with a star.

contrast, the code snippets that Treefix aims to execute are incomplete, e.g., due to undefined variables and functions, and may require additional dependencies to be installed. Another difference is the interface used to provide values for the code to use. Fuzzing typically provides values at the application interface, e.g., as command line arguments, and test case generation typically provides values at the function or method interface. Instead, Treefix aims to fill in values that are missing at an arbitrary point in the given code snippet, without any well defined interface for providing these values.

B. Overview and Running Example

To address the problem of generating prefixes that maximize the cumulative coverage of a code snippet, Treefix reasons about the problem in a multi-step fashion. The basic idea is to iteratively generate and refine prefixes until reaching full coverage or exceeding a configurable budget. To generate and refine prefixes, Treefix queries an LLM with information obtained by statically and dynamically reasoning about the code snippet and already generated prefixes. The approach provides three kinds of information to the model: *undefinedness guidance*, i.e., variables, attributes, and methods that are undefined in the code snippet; *error guidance*, i.e., runtime errors observed when executing the code snippet with a specific prefix; and *coverage guidance*, i.e., lines in the code snippet that are not yet covered by the currently known prefixes.

The prefixes generated by Treefix form a tree, as illustrated in Figure 2. The root is the given code snippet and each of the other nodes represents a prefix. An edge represents a refinement of a prefix in the next step. In addition to the root, the nodes in the tree are grouped into three levels, which correspond to the three kinds of guidance provided to the LLM. The approach iteratively generates the tree of prefixes in a breadth-first manner. That is, Treefix first generates prefixes based on undefinedness guidance, then refines them based on error guidance, and finally further refines the prefixes based on coverage guidance. During this process, the approach keeps track of the prefixes S that maximize cumulative coverage, highlighted in green in the figure, and of the single-best prefix p_{best} that achieves the highest coverage, highlighted with a star.

The following illustrates the approach on the motivating example from Figure 1a.

Undefinedness guidance: The first step is to statically identify any undefined values in the code and ask the LLM to predict code that initializes them. Figure 1c shows an example of prediction made in the first step of Treefix. The LLM predicts `get_register_func` to be the also predicted `dummy_register_func` function, which receives two arguments and returns a string. Moreover, the model predicts `self` to be a `Mock` object containing the attributes accessed in the snippet. While these values look legitimate at first sight, executing the prefix and the snippet shows that, when `get_register_func` is called on the first line of the `try` block, a "TypeError: dummy_register_func() missing 1 required positional argument: 'alias'" exception is raised.

Error guidance: The second step of Treefix uses the feedback from the execution of the code snippet to refine the prefix, providing the invaluable information provided in the error message to the LLM. Figure 1d shows the refined prefix obtained in this step. Notice that only the value of `dummy_register_func` changed. Now, the LLM predict `dummy_register_func` to return another function, which receives two arguments and returns a string containing `"@"`. Given this prefix, the code snippet runs without raising any exceptions. The execution covers the two `if` blocks, covering 60% of the lines.

Coverage guidance: To further increase the coverage of the given code snippet, the third step of Treefix guides the LLM toward prefixes that execute those lines that were not executed in the previous steps. For our example, those lines are the `except` and `else` blocks. Figures 1e, 1f, and 1g show three prefixes obtained in the third step. The prefixes contain a value for `self.email` without `"@"`, a `register` set to `None`, and a statement that will raise a `SystemExit` exception, respectively. The union of these prefixes successfully executes all lines in the code snippet, achieving 100% line coverage.

C. Main Algorithm

Algorithm 1 summarizes the main steps of the approach. In addition to the code snippet s, the algorithm takes two parameters as input: the number n of prefixes to generate per prompt with the LLM and the number k of coverage guidance attempts. The output of the algorithm is the set P of prefixes that maximize cumulative coverage and the single-best prefix p_{best} that achieves the highest coverage. The remainder of this section describes the three steps of the approach in detail.

D. Step 1: Undefinedness Guidance

Given the code snippet s, the first step of Treefix aims to predict a prefix that initializes any undefined references

Algorithm 1 Main algorithm of Treefix.

Input: Code snippet s, number n of prefixes to generate per prompt, number k of coverage guidance attempts
Output: Set P of prefixes that maximize cumulative coverage and $p_{best} \in P$

1: $P, p_{best} \leftarrow \emptyset$, None
▷ **Step 1: Undefinedness guidance**
2: $V_{undefined} \leftarrow$ GETUNDEFINEDREFS(s)
3: $prompt_1 \leftarrow$ GENPROMPT1(s, $V_{undefined}$)
4: $P_1 \leftarrow$ QUERYLLM($prompt_1$, n)
5: $execRes$, $covRes \leftarrow$ EXECUTE(P_1, s)
6: $P, p_{best} \leftarrow$ UPDATEPREFIXES(P, p_{best}, P_1, $covRes$)
▷ **Step 2: Error guidance**
7: $P_{error} \leftarrow$ GETERRORPREFIXES(P_1, $execRes$)
8: **for** p_{error} in P_{error} **do**
9: $prompt_2 \leftarrow$ GENPROMPT2(s, p_{error})
10: $P_2 \leftarrow$ QUERYLLM($prompt_2$, n)
11: $execRes$, $covRes \leftarrow$ EXECUTE(P_2, s)
12: $P, p_{best} \leftarrow$ UPDATEPREFIXES($P, p_{best}, P_2, covRes$)
▷ **Step 3: Coverage guidance**
13: $P_3 \leftarrow P_2$
14: **while** CUMULATIVECOV(P) < 1 and $k > 0$ **do**
15: $execRes$, $covRes \leftarrow$ EXECUTE(P_3, s)
16: $s_{annot} \leftarrow$ ANNOTATEUNCOVEREDLINES(s, $covRes$)
17: $prompt_3 \leftarrow$ GENPROMPT3(s_{annot})
18: $P_3 \leftarrow$ QUERYLLM($prompt_3$, n)
19: $execRes$, $covRes \leftarrow$ EXECUTE(P_3, s)
20: $P, p_{best} \leftarrow$ UPDATEPREFIXES($P, p_{best}, P_3, covRes$)
21: $k \leftarrow k$ - 1
22: **return** P, p_{best}

1
Provide self-contained and concrete Python values to initialize the undefined variables in the code snippet.

2
```
# begin code snippet
(see Figure 1a)
# end code snippet
```

3
```
# begin undefined variables
self
get_register_func
# end undefined variables
```

4
```
# begin undefined attributes and methods
self.user_type
self.name
self.alias
# end undefined attributes and methods
```

5
Response specification (see Figure 4)

Fig. 3: Prompt for undefinedness guidance.

Respond strictly with JSON. The JSON should be compatible with the TypeScript type "Response":

```ts
interface Response {
  // Python import statements, one string per import
  imports: string[];

  // Python code to initialize undefined variables,
       one string per variable
  initialization: string[];
}
```

Fig. 4: Response specification.

in s. These steps are summarized in lines 2 to 6 of Algorithm 1. To identify all undefined references in the given code snippet, Treefix calls GETUNDEFINEDREFS, which is based on static analysis. GETUNDEFINEDREFS first parses the source code of the code snippet into an AST. Then, within the AST, it identifies the scopes of all variables in the code. Finally, GETUNDEFINEDREFS iterates over each scope and then over each variable access within that scope. If a variable is being accessed but not defined in the current scope or any enclosing scopes, it is considered undefined. For example, in a code snippet `a = b + foo()` the variables `b` and `foo` are undefined. Besides undefined variables, GETUNDEFINEDREFS also identifies undefined attributes and methods. For example, in `y = d.year - p.init()`, the attribute `d.year` and the method `p.init()` are undefined, besides the undefined variables `d` and `p`. For any undefined variable, any of its attributes or methods are considered to be undefined as well. To determine the undefined attributes and methods, GETUNDEFINEDREFS identifies the locations of the nodes of the undefined variables and visits the attributes and methods being used in s. Whenever the base object of an undefined attribute or method matches with one of the undefined variables, the undefined attribute or method name with its base object, e.g., `d.year`, is added to a list, which is returned by GETUNDEFINEDREFS.

Based on the statically determined set of undefined references, Treefix generates a prompt aimed at predicting code to define those references (GENPROMPT1). Figure 3 shows the prompt (slightly modified for readability), which has the following structure: 1) a request to provide the missing values; 2) the code snippet s with comments indicating its beginning and end; 3) the list of undefined references in s; 4) the list of undefined attributes and methods in s; 5) a specification of the expected response, as described in Figure 4. Next, Treefix calls QUERYLLM, which queries the model with the prompt to obtain n prefixes. The rationale for generating multiple prefixes for the prompt is to increase the diversity of the prefixes generated, and hence, the chance to find prefixes that successfully cover the code in s.

Given the prefixes returned by the LLM, which are stored in set P_1 in Algorithm 1, Treefix executes them and updates the coverage information. Because this part of the approach is used for all three of Treefix's steps, we describe it in more detail in Section III-G. In short, EXECUTE post-processes each prefix, automatically installs any third-party dependencies required to execute the prefix, prepends the prefixes to the code snippet

1 When trying to execute the code snippet with the provided imports and initialization, the following error happens:

2
```
# begin error message
Execution error at line 14:
    register = get_register_func(self.user_type)
TypeError: dummy_register_func() missing 1 required
     positional argument: 'alias'
# end error message
```

3 Provide a fixed version of the imports and initialization to solve the error and make the code snippet executable.

4 Response specification (see Figure 4)

Fig. 5: Prompt for error guidance.

s, and then measures the line coverage achieved by it. Finally, Treefix updates the set of prefixes P and the single-best prefix p_{best} based on the coverage achieved by the prefixes in P_1.

E. Step 2: Error Guidance

The values predicted in step 1 of Treefix may be incomplete or contain values that lead to execution errors, as in Figure 1c. To fix any errors, the second step of Treefix uses the error messages observed during the execution of the prefixes generated in step 1 as feedback to refine any erroneous prefixes (lines 7 to 12 in Algorithm 1). The helper function GETERRORPREFIXES checks the execution results of the prefixes in P_1 and returns those prefixes that raised an exception. Each exception contains the exception type, message, and line number where it happened.

For each prefix that raises an exception, Treefix formulates a prompt (GENPROMPT2) aimed at predicting a fixed version of the prefix. Figure 5 shows the prompt structure, which contains the following: 1) a description of the problem; 2) the exception type, message, and line number where it happened; 3) a statement of the task; 4) a response specification indicating the response content and its format, as described in Figure 4. Similar to step 1, Treefix queries the LLM with the prompt to obtain n refined prefixes, which are stored in set P_2 in the algorithm. Because LLMs tend to be more effective when given meaningful context, Treefix keeps the conversation history that has led to the erroneous prefix in step 1 as part of the prompt for step 2. Finally, the approach executes the prefixes in P_2 and updates the coverage information (again, Section III-G will provide the details).

F. Step 3: Coverage Guidance

Steps 1 and 2 are often successfully at finding one or more prefixes that enable executing the given code snippet without errors. However, there may still be lines in the code snippet that are not executed by any of the prefixes generated in the previous steps, such as the multiple branches in Figure 1a.

1 When trying to execute the code snippet with the provided imports and initialization, the lines commented with "uncovered" are not executed.

2
```
# begin code snippet
try:
    register = get_register_func(
        self.user_type)
    if register is not None:
        self.email = register(
            self.name, self.alias)
        if '@' in self.email:
            result = 0
        else:
            result = -1 # uncovered
    else:
        result = -2 # uncovered
except SystemExit: # uncovered
    result = 1 # uncovered
# end code snippet
```

3 Provide a modified version of the imports and initialization to execute one of the uncovered paths in the code snippet.

4 Response specification (see Figure 4)

Fig. 6: Prompt for coverage guidance.

To increase the coverage of the code snippet, the third step of Treefix aims to predict prefixes that exercise any not yet covered lines, as summarized in lines 13 to 21 in Algorithm 1.

The basic idea of step 3 is to iteratively generate additional prefixes until either reaching full coverage or exhausting a budget of k attempts. In each iteration, Treefix calls ANNOTATEUNCOVEREDLINES, which identifies all the lines in s that have not been covered by the previous predictions and marks them using a special comment `# uncovered`. The approach then formulates a prompt aimed at predicting a prefix that exercises the uncovered lines. Figure 6 shows the prompt structure, which contains: 1) a description of the problem; 2) the annotated code; 3) a statement of the task; 4) the response specification. As in Steps 1 and 2, Treefix calls the LLM to generate n prefixes. For each prefix, the approach executes it and updates the coverage information.

G. Execution and Coverage Measurement

A key component of Treefix is to obtain feedback by executing the prefixes generated by the LLM. The following presents code execution and coverage measurement in more detail, which corresponds to the helper functions EXECUTE and UPDATEPREFIXES in Algorithm 1. These helper functions are used in all three steps of Treefix.

1) Installing Third-Party Dependencies: The predicted prefixes may contain imports from dependencies that are not currently installed in the environment where Treefix is being executed. Our approach automatically identifies and installs any missing dependencies. To this end, we use *pipreqs*, a Python library that identifies the dependencies based on the

TABLE I: Datasets used for evaluation.

Dataset	Snippets	LoC
Open-source functions	1,000	9,653
Stack Overflow snippets	462	3,580
Total	1,462	13,233

imports in the code. For example, the predictions made by Treefix in Figure 8b start with two `import` statements. In this case, Treefix identifies that `pandas` and `numpy` are dependencies and installs them. Because installing dependencies is one of the most time-consuming processes of Treefix, we keep a shared environment between snippets. This environment contains all installed dependencies and avoids repeatedly installing the same libraries.

2) Post-Processing of Prefixes: Some of the prefixes generated by the LLM may contain errors, yet other parts of the same prefix are useful. Treefix post-processes the prefixes to heuristically remove any lines that raise an execution error. Specifically, the approach iteratively attempts to execute each prefix up to 10 times and removes any lines that raise an execution error. If this process results in a prefix that runs without errors, it will be concatenated with the code snippet s and executed to measure the coverage achieved, as described below. Otherwise, the prefix is discarded. Another problem is that the predicted prefixes may contain an infinity loop or take very long to run. To work around this problem, Treefix also removes prefixes that take more than 30 seconds to execute.

3) Measure Coverage: For all prefixes that, when executed on their own, neither raise an error nor time out, Treefix measures the coverage achieved when prepending the prefix to the code snippet. To measure the coverage achieved by a prefix in s, Treefix uses the same strategy to measure coverage used by LExecutor. It instruments s by adding a call to a special function `_l_` after every line in the code. `_l_` receives a unique ID, which identifies the line above, as argument. Whenever `_l_` is called, the previous line was successfully executed. Treefix combines, in this order, the predicted and post-processed prefix with the instrumented version of s into a program, and then executes it to record the executed line numbers. Unlike reports by popular coverage tools, this measurement considers a line "covered" only if the entire line executes without crashing.

IV. Evaluation

We structure our evaluation along five research questions.

- RQ1: How much code does an execution guided by Treefix cover, and how does it compare to prior work?
- RQ2: How do the three steps in Treefix contribute to its effectiveness?
- RQ3: Qualitatively, why does Treefix achieve different coverage results than existing work?
- RQ4: How diverse are the values predicted by Treefix?
- RQ5: What are the costs of executing code with Treefix?

A. Experimental Setup

1) Benchmark Datasets: We evaluate on two datasets, described in Table I, containing Python code snippets used in previous work [1]. The *Open-source functions* dataset contains 1,000 randomly selected functions from five large and diverse open-source Python projects. The *Stack Overflow snippets* dataset contains 462 code snippets from the answers to Python questions on Stack Overflow.

2) Baselines: We compare Treefix with six alternative approaches for executing arbitrary code snippets: 1) LExecutor [1], for which we use the most effective variation, i.e., the fine-grained value abstraction, using top-1 predictions from the CodeT5 model. 2) SelfPiCo [2], an LLM-based approach developed concurrently with this work, which guides code execution in an interactive loop. We apply their approach to the datasets we use here, which is the same as in the LExecutor paper, but different from the dataset used in the SelfPiCo paper. As their fine-tuned Code Llama model is not available, we use their approach with GPT-3.5, which they report to achieve similar performance than the version with Code Llama. 3) Incompleter [3], a rule-based, feedback-driven approach. It measures coverage using *coverage.py*, which – unlike our coverage metric – counts a line as covered even if that line crashes. For a fair comparison, we modify Incompleter to measure coverage as we do for Treefix and all the other baselines, i.e., considering a line as covered if it is successfully executed. We apply Incompleter to the datasets we use. In addition to these three state-of-the-art baselines, we also consider the baselines that LExecutor has been compared with: 4) "As Is", i.e., trying to execute a code snippet as it is without making any value predictions. 5) Pynguin, a function-level test generator for Python [11]. As in previous work [1], for a fair comparison, we run Pynguin on a single function at a time, which contains only the code to execute. 6) Type4Py, a neural model that predicts types for all local variables, parameters, and return values [10]. We concretize the predicted types as done for this baseline in previous work [1].

3) Evaluated Models: We use OpenAI's latest flagship models: GPT-4o, the largest available model, and GPT-4o mini, a more lightweight and cheaper model.[1]

4) Algorithm Parameters: We set the maximum number of completions to $n = 10$ for each query to the model. Moreover, in step 3, we set the maximum number of iterations $k = 10$.

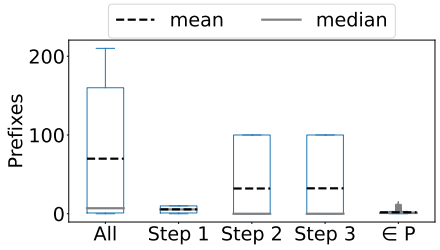
B. RQ1: Effectiveness at Covering Code

Table II (left side) shows the line coverage achieved by Treefix and by the baselines on the two datasets. We use the Wilcoxon signed-rank test to compare the significance of coverage differences between techniques, at p = 0.05. For the open-source functions, on average, executing the code as it is and the type (Type4Py) predictor cover only 4.1% and 13.3% of the lines, respectively. LExecutor and Incompleter increase the mean coverage to 51.6%. Then, SelfPiCo further increases the coverage to 59%. Finally, Treefix covers 76% of

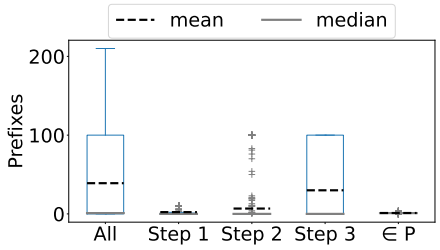
[1] https://platform.openai.com/docs/models

TABLE II: Effectiveness achieved by Treefix and baselines.

Approach	Coverage				Full Execution Rate	
	Open-source functions		Stack Overflow snippets		Open-source functions	Stack Overflow snippets
Treefix (GPT4o)	P = **0.84**	p_{best} = **0.76**	P = **0.82**	p_{best} = 0.72	**0.69**	**0.71**
Treefix (GPT4o-mini)	P = 0.79	p_{best} = 0.73	P = 0.79	p_{best} = **0.78**	0.62	0.66
SelfPiCo	0.59		0.75		0.40	0.60
Incompleter	0.51		0.69		0.35	0.53
LExecutor	0.51		0.65		0.35	0.49
Type4Py	0.13		0.46		0.08	0.32
Pynguin tests	0.04		-		0.02	-
As Is	0.04		0.43		0.02	0.30



(a) Open-source functions.



(b) Stack Overflow snippets.

Fig. 7: Prefixes explored by Treefix.

the lines with p_{best} and even 84% with P, which is higher than all considered baselines (statistically significant) and an improvement of 25% over SelfPiCo, i.e., the currently best available approach. Comparing the two LLMs shows that using the larger GPT4o model is more effective than using GPT4o-mini. Nevertheless, even with GPT4o-mini, Treefix is still significantly more effective than SelfPiCo. On the Stack Overflow code snippets datasets, Treefix covers 82% of the lines with P, an improvement of 7% over SelfPiCo.

We also measure the full execution rate, i.e., how many of all code snippets achieve 100% line coverage. The results are presented in Table II (right side). Overall, Treefix outperforms the baselines and fully executes the code of 69% of the open-source functions and of 71% of the Stack Overflow code snippets. In contrast, SelfPiCo fully executes the code of 62% of all open-source functions and of 66% of all Stack Overflow code snippets.

C. RQ2: Design Choices

1) Effectiveness per Step: Treefix uses three steps to enable code execution. We evaluate how each of these steps contributes to the improvements in effectiveness. To this end, we measure the line coverage and full execution rate achieved after each step, again for both datasets and models. Table III presents the results. On average, Treefix, with either of the considered models, in the first step already achieves comparable or higher effectiveness than the best baseline, SelfPiCo (Table II). Steps 2 and 3 consistently further increase effectiveness on both considered datasets, regardless of the model.

To understand when Treefix typically finds the single-best prefix, Table IV shows the number of snippets where p_{best} is found in a specific step of the approach. We see that p_{best} is most frequently found on step 1, yet steps 2 and 3 also contribute to finding the best prefix in many cases. Taken together, steps 2 and 3 contribute 21.7% and 17.0% of all p_{best} prefixes for the open-source functions and Stack Overflow datasets, respectively.

Overall, these results indicate that each step in Treefix consistently adds to its effectiveness, with step 1 being the most important. The low number of prefixes in P indicates that Treefix effectively determines a small number of prefixes that maximize coverage, which is useful to keep the number of executions performed in a downstream applications of Treefix manageable.

2) Trees of Explored Prefixes: Treefix generates the prefixes that form a tree. In total, every tree of prefixes could contain up to 210 nodes: 10 nodes from step 1, 100 nodes from step 2, and 100 nodes from step 3. We investigate how many of these nodes are visited, on average, before Treefix terminates, e.g., because it has already fully covered the given snippet. Figure 7 shows the distribution of the number of prefixes explored in total ("all"), and in each step of the approach. We also show the number of prefixes in the set P. On average, Treefix explores 70 prefixes for the open-source functions dataset, and only two of these prefixes end up in P, i.e., are required to achieve maximum coverage. The maximum observed size of P is twelve. Across the 1,000 snippets in the open-source functions dataset, Treefix achieves full coverage for 531 in step 1. Then, 442 go to step 2, and 392 go to step 3. Figure 7b shows the corresponding results on the Stack Overflow dataset. Here, Treefix explores 39 prefixes, and only one is added to P, on average. The maximum observed size of P is four. Out of the 462 snippets, Treefix achieves full coverage for 275 in step 1. Among the remaining snippets, 52 have errors that Treefix tries to fix in step 2, and 166 go to step 3 for further increasing coverage.

3) Impact of Resolving Dependencies: As described in Section III-G1, Treefix uses pipreqs to identify and install missing

TABLE III: Effectiveness achieved on each step of Treefix.

Model	Dataset	Coverage			Full Execution Rate		
		I	II	III	I	II	III
GPT4o	Open-source functions	0.72	0.78	**0.84**	0.54	0.61	**0.69**
	Stack Overflow snippets	0.73	0.77	**0.82**	0.59	0.64	**0.71**
GPT4o (mini)	Open-source functions	0.64	0.74	0.79	0.46	0.56	0.62
	Open-source functions w/o pipreqs	0.60	0.70	0.74	0.44	0.53	0.58
	Stack Overflow snippets	0.70	0.72	0.79	0.56	0.59	0.66
	Stack Overflow snippets w/o pipreqs	0.67	0.71	0.77	0.54	0.58	0.64

TABLE IV: Snippets where p_{best} is found in a specific step.

Dataset	Step		
	I	II	III
Open-source functions	**719**	129	70
Stack Overflow snippets	**356**	28	45

dependencies. On the 1,462 snippets used in our evaluation, pipreqs is invoked 1,216 times and succeeds in 1,206 of these invocations. The ten failures are caused by syntax errors on the predicted prefixes. After identifying dependencies with pipreqs, Treefix tries to install them using pip install. In total, pip install gets invoked 48 times, as we do not try to install the same dependency twice, which is successful in 38 out of the 48 cases. The failures are caused by versions of the libraries suggested by pipreqs that are not available for the version of Python we use, i.e., Python 3.8.

To understand the importance of resolving dependencies, we investigate the effectiveness of Treefix without pipreqs. For cost reasons, we only consider GPT-4o mini. The results are in Table III, in the lines containing "w/o pipreqs". We observe that for both datasets, not resolving dependencies leads to a decrease in effectiveness, i.e., this part of our approach contributes to achieving high coverage.

D. RQ3: Case Studies

We qualitatively analyze the strengths and weaknesses of Treefix by inspecting samples of examples.

1) Reasons for Higher Coverage: Across all 1,462 analyzed code snippets, Treefix achieves higher coverage than LExecutor in 707 code snippets: 561 from the open-source functions dataset and 146 from the Stack Overflow dataset. We randomly inspect a sample of 20, 10 from each dataset, leading the the following observations.

Adequate imports and usage of dependencies: For 15/20 code snippets, Treefix increases coverage by importing a dependency and using it to create adequate values. LExecutor does not add any imports but always injects a value from a fixed set.

Complex objects: For 13/20 code snippets, Treefix predicts code that creates complex objects, e.g., with `type("Mock", bases, dict)`. These objects usually contain the attributes and methods used in the code snippet. Figure 8 shows an interesting combination of using imported dependencies and complex objects. Notice that the code snippet in Figure 8a tries to access a multi-index dataframe with random data. The prefix predicted by Treefix (Figure 8b)

```
df = multiindex_dataframe_random_data.T
expected = df.values[:, 0]
result = df["foo", "one"].values
tm.assert_almost_equal(result, expected)
```

(a) Code snippet.

```
import pandas as pd
import numpy as np

index = pd.MultiIndex.from_product(
    [['foo', 'bar'], ['one', 'two']],
    names=['first', 'second'])
data = np.random.rand(4, 4)
multiindex_dataframe_random_data = pd.DataFrame(
    data, index=index, columns=index)
tm = type('Mock', (object,),
          {'assert_almost_equal': lambda x, y: None})
```

(b) Prefix predicted by Treefix.

Fig. 8: Example of adequate imports and usage of dependencies and complex values predicted by Treefix.

```
x=[1,2,3,4]
y=iter(x)
y=[1,2,3,4]
y.next()
...
```

Fig. 9: Example of problematic code.

correctly imports `pandas` and `numpy`, and then instantiates a multi-index dataframe with random values. Moreover, Treefix assigns to `tm` a mock object, which contains an `assert_almost_equal` method receiving two arguments, as used in the last line of the given snippet.

Diverse primitive values: For 11/20 code snippets, Treefix predicts domain-specific primitive values, e.g., meaningful string and integers. This differs from LExecutor, which predicts values from a fixed set only.

Multiple paths covered: For 1/20 code snippets, the multi-step algorithm of Treefix yields multiple snippets, that cover different paths. Since LExecutor makes only one prediction, it fails to fully cover any examples with multiple paths.

2) Reasons for Lower Coverage: Across all 1,462 code snippets Treefix has lower coverage than LExecutor in 58 code snippets: 47 from the open-source functions dataset and 11 from the Stack Overflow dataset. We randomly selected and inspect 10 snippets from each dataset, with the following observations.

Missing dependencies and language versions: 12/20 code snippets have problems with missing dependencies. In these cases, the LLM predicts a correct prefix, but its execution fails because Treefix fails to install the correct dependency. Moreover, one of the code snippets simply fails because it requires an older version of Python than the one used in our experimental environment.

Problematic code: 4/20 code snippets are problematic and cannot be fully executed, no matter what the model predicts. Figure 9 shows an example. Notice that initially `y` is an `iterable` and then is changed to a `list`, which does not contain the `next()` method. Therefore, the fourth line will always fail. Given the the corresponding error message in step 2, Treefix creates a prefix with `y = iter([1, 2, 3, 4])`. However, as by design we always add the predictions to the top of the file, `y` will always be a list when `y.next()` is called.

Unparseable string: We specify the output of the model to be a JSON according to Figure 4. This helps the model to produce predictions that are often more structured and easier to parse than plain text. However, for three of the 20 code snippets, the LLM consistently produced strings on the JSON responses, across all samples we get from the model, that could not be parsed.

E. RQ4: Diversity of Values

As observed in RQ3, one of the reasons for the effectiveness of Treefix is the diversity of values predicted by the LLM. We further study this diversity by analyzing how many unique types and values the approach predicts. To this end, we consider each of the prefixes predicted by Treefix. For each prefix, we identify the undefined variables in the corresponding code snippet and the values assigned to them in the prefix. We consider two primitive values to be equal if they have the same string representation. For non-primitive values, e.g., complex objects, we consider two values to be equal if they share the same set of attributes and methods, as determined by Python's built-in `dir` function. We then count the number of unique types, and the number of unique values per type, across all prefixes predicted by Treefix.

In total, for the predictions for the 1,462 snippets on the two datasets, Treefix predicted 1,120 unique types and 16,528 unique values. These numbers are in stark contrast to the 23 fixed values predicted by LExecutor. We attribute this difference to the LLM's ability to predict context-dependent and domain-specific values that fit naturally to the given code. Table V presents the most commonly predicted types, ordered by the number of unique values. Interestingly, there is a mix of primitive and non-primitive types. The type with most predicted unique values is `str`, with 2,962 unique values. The prefixes predicted by Treefix in Figure 1a illustrate the importance of domain-specific strings. Another commonly predicted type is `type`, which means that Treefix dynamically creates objects on the fly based on the content of the code snippet. An example of a predicted `type` is presented in Figure 8b. Another interesting observation are the 181 unique values for type `module`. This type refers to all imported dependencies, such as `pandas` and `numpy` in the prefix in Figure 8b.

TABLE V: Diversity of values predicted by Treefix.

Type	Unique values
str	**2962**
list	**2567**
Mock	**1383**
type	**1083**
dict	**936**
object	**919**
set	657
MockSelf	406
ContextVar	342
tuple	322
SimpleNamespace	320
ndarray	298
bytes	256
Pattern	196
module	181
Line	148
int	110
...	
Total	16528

F. RQ5: Efficiency and Costs

Table VI shows the efficiency and costs of Treefix by measuring (i) the time it takes to execute a code snippet, and (ii) the monetary expenses to query the LLMs, based on OpenAI's pricing for GPT4o and GPT4o-mini as of July 2024. On average, the first step is the most time-consuming. We attribute this to the time required to install dependencies on the prefixes predicted in step 1. Most times, the prefixes predicted in steps 2 and 3 contain imports previously predicted in step 1, and hence, are already installed. Overall, on average, Treefix takes 18.6 and 21.2 seconds to execute on an open-source function, and 7.8 and 7 seconds to execute on a Stack Overflow code snippet with GPT4o and GPT4o-mini, respectively.

As presented on the right side of Table VI, the price to query both models in Treefix increases along the steps for both datasets. OpenAI's pricing depends on the amount of tokens consumed and produced by the model. The amount of tokens in the prompts is higher as the steps in Treefix increase. On average, Treefix with GPT4o costs USD 0.425 to execute on an open-source function and USD 0.212 to execute on a Stack Overflow code snippet. Using the smaller, less expensive GPT4o-mini models reduces costs significantly, with USD 0.016 to execute on an open-source function and USD 0.008 to execute on a Stack Overflow code snippet.

These results allows for several observations. First, both Treefix's execution time and monetary costs depend on the amount of missing values, as the open-source functions dataset contains almost double the amount of missing values of the Stack Overflow dataset. Second, there is a clear trade-off between the cost of a model and its effectiveness. Treefix with GPT4o is much more expensive than with GPT4o-mini. However, the higher expense is paid with higher effectiveness, as presented in Table II. For example, on the open-source functions, using the newer model increases costs by 27x, while

TABLE VI: Average costs per code snippet imposed by Treefix.

Model	Dataset	Time (seconds)				Price (USD)						
		Step 1	Step 2	Step 3	All	Step1		Step 2		Step 3		All
						Input	Output	Input	Output	Input	Output	
GPT4o	Open-source functions	13.7	2.7	2.2	**18.6**	0.002	0.019	0.056	0.106	0.132	0.109	**0.425**
	Stack Overflow snippets	5.2	0.7	1.9	**7.8**	0.001	0.006	0.012	0.013	0.119	0.059	**0.212**
GPT4o-mini	Open-source functions	14.2	3.8	3.2	**21.2**	6.67x10-5	0.0007	0.0018	0.0045	0.004	0.004	**0.016**
	Stack Overflow snippets	4.4	0.4	2.2	**7.0**	3.62x10-5	0.0003	0.0005	0.0008	0.004	0.003	**0.008**

increasing coverage from 79% to 84%. Third, another trade-off exists between the costs of letting the approach perform all three steps and the benefits obtained in terms of higher coverage. While the time taken by Treefix increases only slightly in steps 2 and 3, the costs increase significantly. For example, when using GPT4o on the open-source functions dataset, 94.8% of the total costs are imposed by steps 2 and 3, which contribute only 14.3% of the total coverage achieved. These trade-offs are important to consider when using Treefix in practice, as they provide users with the flexibility to choose the most suitable model and number of steps to balance costs and effectiveness.

V. Threats to Validity

The two datasets we use are diverse and representative of real-world scenarios, but may not cover all possible types of Python code. Following prior work [1], we use line coverage as our primary metric of effectiveness, which does not capture other important aspects of code execution, such as execution time and memory usage. Our experiments are conducted using specific versions of the LLMs (GPT-4o and GPT-4o-mini). Future updates to these models could impact the reproducibility of our results. To mitigate this threat, we make logs of running our experiments available. Likewise, our results may not generalize to other LLMs. We reduce this threat by evaluating on the currently best available model (GPT4o), but also on a smaller model (GPT4o-mini). The comparison with SelfPiCo is based on the best LLM used in their work (GPT-3.5), which differs from the models we use. We leave a comprehensive comparison of different approaches with a range of models as future work. Finally, the results depend on the specific prompts used to query the LLMs. We design the prompts to describe the problem to the LLM in a way similar to how a human would understand it and refine the prompts during preliminary experiments.

VI. Related Work

Arbitrary code execution: Prior work explores ways of executing arbitrary pieces of code. Micro-execution [17] enables executing arbitrary x86 code by injecting binary values into memory on demand in a virtual machine. Undercon-strained symbolic execution [18] applies symbolic execution to individual functions in isolation. Forced execution [19] forces the execution of uncovered paths. LExecutor [1] enables the execution of arbitrary code snippets using a neural model. Our work differs from the above by predicting code prefixes that create missing values to maximize coverage.

Test generation and fuzzing: Test generation and fuzzing automatically produce tests and inputs to dynamically trigger software errors. Many approaches use traditional techniques, e.g., symbolic execution [20]–[22] or search-based algorithms [11], [23]. Recent work [14], [24]–[26] rely on AI and LLMs. Both fuzzing and test case generation assume that the target code is complete and executable, e.g. in an environment with all dependencies installed, and that the input values are provided at well defined locations. In contrast, Treefix enables the execution of incomplete code and predicts values to be used at arbitrary locations.

Automated program repair: APR aims to automatically generate patches for buggy programs. Recently, researchers have started to apply LLMs for APR [27]–[29]. Unlike APR, our approach does not modify the to be executed target code snippet. Instead, we produce prefixes that are added at the top of the code snippet to enable its execution.

Learning and LLMs in software engineering: Learning-based approaches and LLMs have been applied to various software engineering tasks [30], including type prediction [31]–[34], test generation [24], [25], [35], [36], fuzzing [14], [26], code completion [5], [37], [38], and automated repair [28], [39]–[42] and coding agents [29], [43]–[46]. Our work is the first to propose an LLM-based approach for learning-guided code execution.

VII. Conclusions

Motivated by the recurring problem of executing snippets of code, this paper presents Treefix, a novel approach toward learning-guided execution. Starting from a code snippet, the approach iteratively creates a tree of code prefixes aimed at initializing any undefined references, running the code snippet without errors, and covering as many lines of the code snippet as possible. Our empirical evaluation shows improvements over the previous state of the art, raising line coverage from 59–75% to 82–84%. We envision Treefix to provide a basis for various dynamic analysis applications, as well as a starting point for developer's trying to understand and debug code.

VIII. Data Availability

The code and data associated with our work is available: https://github.com/sola-st/Treefix

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreements 851895 and 101155832) and by the German Research Foundation within the ConcSys, DeMoCo, and QPTest projects.

References

[1] B. Souza and M. Pradel, "LExecutor: Learning-guided execution," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1522–1534. [Online]. Available: https://doi.org/10.1145/3611643.3616254

[2] Z. Xue, Z. Gao, S. Wang, X. Hu, X. Xia, and S. Li, "Selfpico: Self-guided partial code execution with llms," in *ISSTA*, 2024.

[3] I. Hayet, A. Scott, and M. d'Amorim, "Feedback-directed partial execution," in *ISSTA*, 2024.

[4] L. Gröninger, B. Souza, and M. Pradel, "ChangeGuard: Validating code changes via pairwise learning-guided execution," in *International Conference on the Foundations of Software Engineering (FSE)*, 2025.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[6] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[7] J. Wei, G. Durrett, and I. Dillig, "Coeditor: Leveraging contextual changes for multi-round code auto-editing," *arXiv preprint arXiv:2305.18584*, 2023.

[8] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Language models meet code edits," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1483–1495. [Online]. Available: https://doi.org/10.1145/3611643.3616253

[9] M. Dilhara, A. Bellur, T. Bryksin, and D. Dig, "Unprecedented code change automation: The fusion of llms and transformation by example," in *FSE*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.07138

[10] A. M. Mir, E. Latoškinas, S. Proksch, and G. Gousios, "Type4py: practical deep similarity learning-based type inference for python," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2241–2252. [Online]. Available: https://doi.org/10.1145/3510003.3510124

[11] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *SSBSE*, 2020, pp. 9–24. [Online]. Available: https://doi.org/10.1007/978-3-030-59762-7_2

[12] M. Zalewski, "American fuzzy lop (afl)," https://lcamtuf.coredump.cx/afl/, 2013. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[13] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Trans. Software Eng.*, vol. 45, no. 5, pp. 489–506, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2785841

[14] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 126:1–126:13. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[15] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 919–931. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00085

[16] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," 2024. [Online]. Available: https://arxiv.org/abs/2403.16218

[17] P. Godefroid, "Micro execution," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 539–549.

[18] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[19] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, R. Barrett, R. Cummings, E. Agichtein, and E. Gabrilovich, Eds. ACM, 2017, pp. 897–906. [Online]. Available: https://doi.org/10.1145/3038912.3052674

[20] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2008, pp. 209–224.

[21] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 213–223.

[22] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005, pp. 263–272.

[23] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, 2011, pp. 416–419.

[24] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[25] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," 2024.

[26] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[27] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning." New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3540250.3549101

[28] ——, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 819–831. [Online]. Available: https://doi.org/10.1145/3650212.3680323

[29] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in *International Conference on Software Engineering (ICSE)*, 2025.

[30] M. Pradel and S. Chandra, "Neural software analysis," *Commun. ACM*, vol. 65, no. 1, pp. 86–96, 2022. [Online]. Available: https://doi.org/10.1145/3460348

[31] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*, 2018, pp. 152–162. [Online]. Available: https://doi.org/10.1145/3236024.3236051

[32] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript function types from natural language information," in *Proceedings*

of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, 2019, pp. 304–315. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00045

[33] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: Neural type prediction with search-based validation," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, 2020, pp. 209–220. [Online]. Available: https://doi.org/10.1145/3368089.3409715

[34] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*, 2020, pp. 91–105. [Online]. Available: https://doi.org/10.1145/3385412.3385997

[35] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[36] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage guided test generation in regression setting using llm," in *FSE*, 2024.

[37] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in *ICSE*, 2023.

[38] A. Eghbali and M. Pradel, "De-hallucinator: Iterative grounding for llm-based code completion," *CoRR*, vol. abs/2401.01701, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.01701

[39] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1943–1959, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2940179

[40] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *ICSE*, 2024.

[41] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," 2024.

[42] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," in *FSE*, 2024.

[43] J. Yang, C. E. Jimenez, K. Lieret, S. Yao, A. Wettig, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024.

[44] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *ISSTA*, 2024.

[45] W. Tao, Y. Zhou, W. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," *arXiv preprint arXiv:2403.17927*, 2024.

[46] I. Bouzenia and M. Pradel, "You name it, I run it: An LLM agent to execute tests of arbitrary projects," 2024. [Online]. Available: https://arxiv.org/abs/2412.10133