Master Thesis

Title

Constructing a Dataset of Functionally Equivalent Methods in Python Using Automated Test Generation Techniques

Supervisor
Professor Yoshiki Higo

Author
Yusheng Guo

2025/1/28

Software Engineering Laboratory, Department of Computer Science
Graduate School of Information Science and Technology, Osaka University

令和 6 年度 Master Thesis

Constructing a Dataset of Functionally Equivalent Methods in Python Using Automated Test Generation Techniques

Yusheng Guo

Abstract

As a popular programming language in modern software development, Python boasts an extensive open-source codebase on GitHub. Code reuse is common across these vast repositories. This study leverages open-source Python projects from GitHub and applies automated testing techniques to discover functionally equivalent method pairs. The research involved collecting and processing methods from 5.1k Python projects on GitHub. Due to the lack of type checking in Python, grouping methods present specific challenges. To address this, we performed detailed type inference on the methods and grouped them based on the inferred types, providing a structured and comprehensive foundation for further analysis. Automated test generation techniques were applied to create unit tests for each method. These methods were executed against one another within their respective groups to identify candidate method pairs that produced identical outputs given the same inputs. Finally, through manual checking, we identified 130 functionally equivalent method pairs and 731 functionally non-equivalent method pairs. These method pairs were compiled into a comprehensive dataset, which served as the basis for further analysis. The dataset enabled a detailed examination of performance differences among the functionally equivalent pairs. Additionally, it was used to evaluate the ability of large language models (LLMs) to recognize functional equivalence, focusing on both their accuracy and the challenges they face when handling diverse implementations. The results highlight the potential of LLMs in identifying functionally equivalent methods and point to areas where further advancements can be made.

Keywords

functionally equivalent methods
source code analysis
dataset

Contents

1 Introduction

With the evolution of programming languages, modern languages have become increasingly rich and complex in their syntactical features, particularly dynamic languages like Python. Python, a popular programming language, is widely embraced by developers worldwide due to its concise and readable syntax and powerful standard library. It has a vast codebase and active user communities on platforms like GitHub. Python supports multiple programming paradigms, including object-oriented, functional, and imperative programming. This versatility enables the implementation of identical functionality through different approaches, depending on developers ' coding preferences.

The open-source community provides a vast array of software projects containing rich and diverse code resources. It's common to find methods within these codebases that, while functionally equivalent, are implemented in different ways. Collecting such functionally equivalent code snippets is highly valuable for software engineering research. These snippets can be utilized to create datasets of equivalent methods, which can, in turn, drive advancements in areas like code optimization, refactoring, and test generation. However, identifying and collecting functionally equivalent methods remains a complex challenge due to the variations in their structure.

Many existing code clone detection tools rely on identifying repetitions in code snippets to detect clones, such as the token-based SourcererCC [18] and the tree-based DECKARD [9]. These tools are generally effective at identifying syntactically similar code fragments, such as directly copied-and-pasted code or code with minor changes in variable names. However, they often struggle to detect functionally equivalent but structurally different codes. This is because these tools primarily focus on superficial code similarities and overlook functionality. Therefore, there is an urgent need to develop new techniques and tools capable of identifying functionally equivalent code pairs rather than just syntactically similar fragments.

This study ' s primary goal is to collect functionally equivalent method pairs from open-source projects. Functionally equivalent methods, referred to as FE methods, are defined as pairs of methods that return the same output given the same input (parameters). The key idea of this research is to use Pynguin [11] to automatically generate test cases for the extracted methods, followed by mutual execution to identify methods that exhibit identical behavior under the generated test cases. Subsequently, we manually check all potential FE method pairs to identify the valid FE method pairs. This study selects the ManyTypes4Py [12]dataset as the target for detecting FE method pairs in Python. ManyTypes4Py contains approximately 5.1K type-checked Python repositories, comprising around 1.5 million methods. From this dataset, we extract methods and perform type

inference, followed by grouping based on the type inference results. Test cases are then automatically generated, and mutual execution is conducted within each group. Ultimately, we obtained 7415 candidate FE method pairs and manually checked a part of them.

Subsequently, we constructed a meticulously reviewed dataset, which contains 130 functionally equivalent method pairs and 731 functionally non-equivalent method pairs. In this dataset, we conducted an in-depth performance analysis of the functionally equivalent method pairs, categorizing the main performance differences into three types.

We also used this dataset to evaluate the functional equivalence recognition ability of large language models (LLMs). In the experiment, we selected GPT-4 as the test model to validate its ability to recognize functionally equivalent method pairs. The results indicated that GPT-4 could identify some method pairs that were implemented differently but functionally equivalent, highlighting the significant potential of large language models in this task. Despite some challenges, especially for method pairs with substantial structural differences, GPT-4 demonstrated considerable potential and prospects in functional equivalence recognition.

2 Background

2.1 Definition of FE Methods

FE methods refer to methods that may differ in implementation but are equivalent in functionality. FE methods are characterized by producing identical outputs when provided with identical inputs. Although their code structures may vary, such as using different algorithms, data structures, or coding styles, they ultimately achieve the same functionality. The concept of functional equivalence is especially important in code optimization, refactoring, and clone detection, as identifying FE methods can help developers understand potential redundant code or opportunities for improvement within a codebase.

Code clone detection is typically categorized into four types based on similarity and structural differences:

Type-1 Clones: Identical code fragments, except for variations in whitespace, comments, or identifier names.

Type-2 Clones: Code fragments that are largely similar but may include changes in identifiers, such as variable names or function names, and some code formatting modifications.

Type-3 Clones: Structurally similar code with some differences in code fragments or logic modifications.

Type-4 Clones: Code fragments that have the same functionality but different implementations, also known as semantic clones. These clones are not based on superficial code similarity but rather on the behavior or functionality of the code.

Traditional code clone detection tools primarily focus on detecting Type-1 and Type-2 Clones, which depend on code structure and syntax similarity. These tools identify clones by searching for similar code fragments, but their limitation lies in their inability to effectively detect code fragments that are functionally identical but structurally different.

This research aims to overcome this limitation by using automatic generation techniques to detect functionally equivalent but differently implemented code clones, specifically Type-4 Clones. Type-4 Clones are particularly challenging, as they do not depend on syntactical similarities but instead require an analysis of method behavior to establish functional equivalence. Detecting these clones is crucial for code refactoring and optimization, as it reveals code fragments that are entirely different in implementation but identical in functionality.

2.2 Key Idea for Automatically Identifying Candidate FE Method Pairs from FEMPDataset

Previous research in this field has explored techniques such as automatic test case generation and mutual execution methods. The literature [5] proposed an approach to obtaining a set of FE methods by mutually executing the generated test cases. Additionally, a dataset of FE method sets was constructed using Borge ' s dataset [1]. It contains 276 FE method pairs. Similarly, the FEMPDataset [4], created using similar approaches, consists of 1,342 FE method pairs in Java, validated by three independent programmers.

In the research on FEMDataset, FE method pairs are automatically collected by leveraging both the static features (e.g., method signatures) and dynamic behavior (test results) of Java methods. Static features include return types and parameter types, with methods sharing the same features grouped together. The EvoSuite tool is then used to generate test cases for methods within the same group. Test cases generated by automated test generation techniques have the property that the test cases always succeed. Mutual execution of these test cases is performed to determine whether the methods exhibit equivalent behavior. If a method can pass the test cases generated for another method, and vice versa, the two methods are considered functionally equivalent. Finally, manual checking is conducted to confirm the valid functional equivalence of method pairs, despite differences in implementation.

The success of the FEMPDataset highlights the effectiveness of utilizing automatic test case generation tools and mutual execution techniques to identify FE methods. Its success primarily stems from the ability to verify whether two methods produce identical outputs given the same inputs.

2.3 The Rapidly Developing Python Language

Python has rapidly developed in recent years, becoming one of the most popular and widely used programming languages in the world [19]. Its simple and readable syntax, along with its powerful features, have made it a go-to language across various industries. In software development, Python is widely used in fields such as web development, data analysis, artificial intelligence, and automation testing. According to the TIOBE [7] index and developer surveys conducted by platforms like Stack Overflow, Python has consistently ranked among the top programming languages and continues to climb, particularly in the fields of data science and artificial intelligence, where its usage has surged. Its open-source nature has led to contributions from numerous communities and companies, providing developers with an abundance of tools and resources.

One of the key reasons for Python's popularity is its relatively low learning curve. The lan-

guage's straightforward and intuitive syntax lowers the barriers to entry for programming. Many universities and educational institutions have chosen Python as the primary language for teaching programming, especially in fields such as computer science, engineering, and data science. Python has become the language of choice for students learning to code, and its vast user community and wealth of online learning resources have greatly facilitated the growth and skill development of self-learners.

On GitHub, the world's largest open-source code hosting platform, Python has also achieved remarkable success. According to GitHub's annual reports, Python is one of the most popular programming languages, with many open-source projects and libraries built using Python. Developers can easily access, modify, and optimize these codebases. The open and shared nature of these resources has fostered innovation and collaboration, contributing to Python's continued growth in the tech industry. With strong support from third-party libraries such as NumPy, Pandas, and TensorFlow, Python has become the leading language in data science, machine learning, and artificial intelligence, widely used in both enterprise and research projects.

2.4 Type Inference Techniques in Python

Python, as a dynamic language, lacks static type checking when compared to statically-typed languages like Java and C++. While this characteristic enhances development speed and flexibility, it also presents challenges for code analysis. In statically-typed languages, the types of variables and methods are determined at compile-time, which allows code analysis tools to easily check type consistency, identify potential type errors, and conduct more in-depth static analysis. However, in Python, the types of variables are determined at runtime and can change during the execution of the program, making it more difficult for code analysis tools to perform comprehensive static type inference.

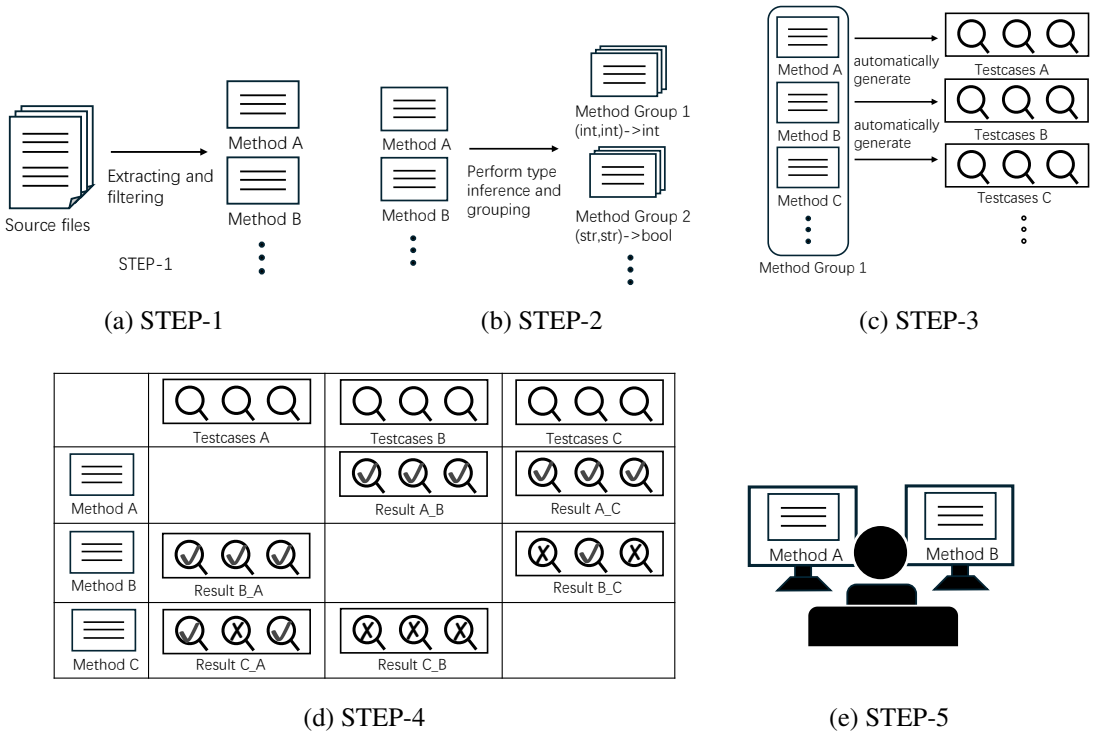
This dynamic nature of Python makes it challenging for static analysis tools to be directly applied to Python code, particularly when performing tasks such as code clone detection and functional equivalence analysis. Many static analysis tools rely on type information and symbol tables for inference and optimization, but in Python, due to the absence of type constraints and explicit type declarations, these tools often fail to deliver the same level of precision. Therefore, analyzing and optimizing Python code requires more flexible and dynamic approaches, such as type inference and dynamic execution techniques, to address the challenges posed by its dynamic characteristics.

Despite the challenges posed by Python's dynamic characteristics, its ecosystem offers robust

support for addressing these issues. With the continuous development of artificial intelligence and machine learning technologies, existing Python type inference tools have made significant progress. These tools combine static analysis with reasoning algorithms to infer the types of variables and methods within code. For example, tools like Type4Py [13], CodeT5 [25] [22], and TypeT5 [24], based on deep learning techniques. And on this basis, formal methods and static analysis are used as supplementary tools [14] [15]. They have shown high accuracy in inferring basic Python built-in types such as int, str, and list, even though they still face some limitations in handling complex types. Particularly, TypeT5 uses seq2seq technology and integrates static analysis as a supporting technique to accurately infer the types in most Python programs, providing a strong support platform. It is precisely because of the advancements and maturation of these tools that this study can build upon the FEMPDataset framework and incorporate type inference techniques, further expanding the study to Python and enhancing the precision and efficiency of code equivalence analysis.

2.5 Key Idea of This Study

Therefore, based on the research findings from FEMPDataset, this study decides to use Python as the research language and extend the detection of functionally equivalent method pairs by adding an additional type inference step. By combining type inference with automatic test case generation, we can more accurately identify Python method pairs that are functionally equivalent but structurally different. Especially in the context of Python as a dynamic language, integrating type inference and automated testing techniques will provide stronger support for detecting functional equivalence. Ultimately, this study will create a dedicated dataset of functionally equivalent method pairs for Python, which will provide valuable resources and data support for subsequent research in areas such as code optimization, refactoring, and code review.



(a) STEP-1 (b) STEP-2 (c) STEP-3

(d) STEP-4 (e) STEP-5

Figure 1: Steps to obtain pairs of functionally equivalent Python methods.

3 Procedure of Dataset Construction

In this study, the following process is used to construct a dataset of FE method pairs:

STEP-1: Extract Python methods from open-source projects on GitHub and perform an initial filtering.

STEP-2: Perform type inference on each method and group them accordingly.

STEP-3: Generate test cases for each method.

STEP-4: Mutually execute methods within the same group to identify candidate FE method pairs.

STEP-5: Manually check each candidate FE method pair to confirm if they are valid FE method pairs.

Figure 1 provides an overview of the five steps described above. STEP-1 through STEP-4 is automatically performed by the developed tool, while only Step 5 is executed manually. The detailed process for each step is as follows:

3.1 STEP-1

To build the dataset of FE method pairs, we selected the ManyTypes4Py [1] dataset as the basis for our experiment. This is a Python benchmark dataset for machine learning-based type inference, containing 5,382 Python projects sourced from GitHub. It offers a diverse collection of open-source Python projects, covering various types and application scenarios and providing an abundant sample of methods. A key reason for choosing this dataset is its suitability for type inference.

From the selected Python projects, we extracted all Python methods. Initially, we used Python ' s Abstract Syntax Tree (AST) module to parse all `.py` files within the projects. This allowed us to construct the abstract syntax tree for each file and extract method definitions. We then traversed these abstract syntax trees to gather method-related information. In total, we obtained 1,500,000 Python methods. For each method, the following information was collected and recorded in a dataset: method name, (original) source code, normalized source code, the number of statements and conditional predicates, file path, start line, and end line.

After processing the extracted Python methods, we normalized the source code. This process involved standardizing all variables, string constants, and code indentation using the `ast` library. These steps helped eliminate formatting differences and excluded the impact of different variable names. Due to the extensive standard library of built-in types in Python, we chose not to normalize the names of method calls.

Figure 2 shows an example of normalization. Subfigure 2a shows the original method code, and subfigure 2b shows the code after normalization. During the normalization process, we first removed the type hints and default values from the method declarations. Next, we renamed all variables, attribute names, and string literals. For all method calls, only the method itself and its recursive calls were renamed.

The reason for not renaming other method calls is that Python has many powerful built-in methods, and renaming all method calls might result in different methods being incorrectly identified as duplicate code. Additionally, any code that calls user-defined external methods is excluded in subsequent steps, as it is beyond the scope of this study.

After completing the code normalization, we generated a unique hash value for each normalized method code. By calculating the hash value of the method code, we could effectively detect and identify duplicate methods. The primary purpose of this step is to eliminate any potential duplicate methods in the dataset, ensuring that each method pair in the dataset is unique.

[1] It was presented in the data showcase of the MSR ' 21 conference.

```
def get_open_business_day(business, day):
    "\n    Helper function which returns 'day' dictionary of
corresponding day for\n given business dictionary. If the day
is not found, returns None.\n    "
    if (len(business.open_hours) == 0):
         return None
    for open_day in business.open_hours:
         if (open_day.day == day):
             return open_day
    return None
```

(a) Original Method

```
def func(val1, val2):
    if len(val1.attr1) == 0:
         return None
    for val3 in val1.attr1:
         if val3.attr2 == val2:
             return val3
    return None
```

(b) Normalized Method

Figure 2: Example of normalization

Next, we filtered the remaining methods to remove those not meeting our research requirements. This step ensured that only relevant methods were retained in the dataset. The following types of methods were excluded:

- **Methods with no parameters or return values**: These methods cannot be effectively evaluated through test cases, as their behavior cannot be adequately judged. Consequently, they did not provide helpful information for functional equivalence analysis and were excluded.
- **Methods with `self` in their parameters**: Methods containing the `self` parameter are typically instance methods in object-oriented programming. Since our research focuses on generating unit tests for standalone methods, instance methods were removed from the dataset.
- **Methods that invoke external classes or methods**: To ensure that the methods in our dataset are self-contained and can be independently analyzed, any method that calls external classes or methods was filtered out. These methods would fail during automatic test case generation, so they were removed in advance to better facilitate the detection of FE method

```
def crossOff(possible, prime):
    nextPrime = None
    for i in range(prime, len(possible)):
        if possible[i] % prime == 0:
            possible[i] = 0
        if possible[i] and (not nextPrime):
            nextPrime = possible[i]
    return nextPrime
```

(a) Original Method

```
def crossOff(possible: list, prime: int) -> int:
    nextPrime = None
    for i in range(prime, len(possible)):
        if possible[i] % prime == 0:
            possible[i] = 0
        if possible[i] and (not nextPrime):
            nextPrime = possible[i]
    return nextPrime
```

(b) Method after type inference

Figure 3: Example of type inference

pairs.

By applying these filtering criteria, we ensured that the remaining methods in the dataset are better suited for further analysis and the identification of candidate FE method pairs. After completing these steps, 28,353 methods were retained for subsequent analysis.

3.2 STEP-2

Since Python is a dynamically typed language and lacks static type checking, this poses challenges for subsequent operations. To improve the effectiveness of automated analysis, it is necessary to perform type inference on methods and group them based on the inferred types. Methods with explicit type information tend to perform more reliably in automated test case generation, making type inference essential.

In this study, we used the TypeT5 [24] tool for type inference. TypeT5 is a Transformer-based model specifically designed for type inference in Python code. It can infer the types of variables and parameters and return values directly from the source code, particularly excelling when explicit type hints are absent. Leveraging TypeT5 allows for a better understanding of method type

```
@pytest.mark.xfail(strict=True)
def test_case_2():
    int_0 = -1741
    int_1 = 2067
    int_2 = module_0.inv(int_1, int_0)
    assert int_2 == -1740
```

(a) Test case with the 'xfail' marker

```
def test_case_0():
    bool_0 = True
    set_0 = {bool_0, bool_0, bool_0}
    module_0.set_add(set_0, set_0)
```

(b) Test case without assert statements

Figure 4: Example of removed test cases.

information, providing a solid foundation for subsequent test generation and functional equivalence detection.

In the subsequent automated test case generation, we will rely on type hints to generate test cases. If the type hints for a method include Python non-built-in types, the test case generation will fail outright. Therefore, before performing inference, developer-provided type hints were initially processed, with non-built-in type hints being removed. Next, we used the TypeT5 tool to infer the types of parameters and return values for methods lacking explicit type hints. After completing the type inference, We rechecked each method's parameters and return values to ensure they all belong to Python's built-in types. Methods that still contain non-built-in types in parameter types or return value types will be removed. This step ensured that the final retained methods had clear built-in type information. Ultimately, 21,503 methods were preserved for grouping and the next step of automated test case generation.

Figure 3 presents an example of type inference. In the original code, the methods lack any type hints. After applying type inference, as shown in the red-highlighted section of the figure, we annotate the variable possible as a list, prime as an int, and the return value as an int. Based on these results, we grouped the methods accordingly. This method is grouped together with other methods that have parameter types (list, int) and a return type of int.

Following this, we grouped all methods based on their parameters and return types. Only methods with identical parameters and return types were grouped together. In the end, there were 726 groups, and each group contained at least two methods. In STEP-4, we will perform mutual

execution within these groups.

3.3 STEP-3

In this step, our primary goal was to generate corresponding test cases for all the methods. We chose the Pynguin [11] for automatic test case generation. Pynguin is a Python-based automatic test case generation tool that can produce a comprehensive set of test cases for a given method, ensuring that all functional aspects of the method are thoroughly tested. By utilizing Pynguin, we could automatically generate test cases for all methods obtained in the previous steps, ensuring each method was adequately validated.

After generating the test cases, we performed initial filtering to remove those test cases that contained 'xfail' markers or lack of assert statements. In Figure 4, we present a test case marked with 'xfail' and another test case lacking an assert statement. The 'xfail' marker indicates that the test case is expected to fail, which generally means that it cannot effectively validate the correctness of the method, making it unsuitable for functional equivalence analysis. On the other hand, test cases lacking 'assert ' statements cannot verify whether the method's actual output matches the expected results. They only checked whether the method could run correctly under given inputs, and therefore, these cases were also excluded. This filtering process aims to ensure that the test cases effectively validate the behavior of the methods rather than merely executing code without actual verification.

Next, we conducted coverage testing on the remaining test cases using the coverage component provided by pytest. The purpose of coverage testing is to assess the extent to which the test cases cover the method's code. To ensure the effectiveness and comprehensiveness of the test cases, we retained only those with 100% coverage. This means that these test cases can cover all code branches and paths in the method, ensuring no code segments that could impact functionality are omitted. Test cases with 100% coverage provide the most thorough validation, ensuring that the functional equivalence analysis in subsequent steps is based on complete and accurate test results.

Ultimately, after this filtering and coverage testing step, we obtained a high-quality set of test cases, including a total of 6,500 test cases for methods. This step took approximately 40 hours. These test cases can comprehensively and accurately validate the functional behavior of each method. We will use these methods and their corresponding test cases for mutual execution in the next step.

3.4 STEP-4

In this step, we utilize the high-quality test cases and method information retained from Step-3, as well as the grouping information from Step-2, to perform mutual execution among methods within the same group. The primary objective of this step is to validate whether the methods are functionally equivalent through cross-testing.

The detailed process is as follows:

1. **Preparation of Test Cases and Methods**:

   Suppose we have method A and method B, each with corresponding test cases A and test cases B. These test cases were rigorously filtered in Step-3 to ensure that they have 100% coverage, thereby thoroughly assessing the functionality of the methods.

2. **Execution of Tests**:

   First, we execute method A using test cases B. This step aims to evaluate the performance of method A under the test cases from method B, confirming whether method A can pass all the tests in test cases B. If method A passes all the tests in test cases B, we proceed to the next step: executing method B using test cases A. This step is intended to verify the performance of method B under the test cases from method A.

   Through this approach, we conduct cross-testing between methods A and method B to ensure that both methods can pass under different test cases.

3. **Result Analysis**:

   If method A passes all tests in test cases B and method B passes all tests in test cases A, it indicates that methods A and B are likely functionally equivalent under the given test cases. This result suggests that both methods produce identical outputs for the same inputs and may be functionally equivalent. Therefore, we mark this pair of methods as candidate FE method pairs.

   Conversely, if any test case fails during the testing process, it indicates a functional discrepancy between methods A and B under the given test cases. In this case, we exclude this method pair from the candidate FE method pairs to ensure that only method pairs that consistently perform similarly across all test cases are considered functionally equivalent.

For method pairs marked as candidate FE method pairs, further validation and analysis are required. Although the preliminary mutual execution provides initial evidence of functional equivalence, the test cases are limited, and this only indicates functional equivalence under specific

conditions. In practical applications, additional verification steps are necessary to ensure that these method pairs exhibit consistent behavior across all possible input conditions.

3.5 STEP-5

In this phase, we perform a detailed manual checking of all candidate FE method pairs. The primary objective of this step is to confirm whether these method pairs indeed exhibit functional equivalence through human judgment.

After completing the mutual execution step, we obtained a list of candidate FE method pairs. Each pair was selected based on the cross-testing results, where both methods passed all tests in each other's test cases. Although this result provides preliminary evidence of functional equivalence, further validation is required to confirm this equivalence.

For candidate method pairs identified as functionally non-equivalent during manual checking, we created new test cases to highlight their functional differences. We designed test inputs that could potentially cause the two methods to produce different results and executed these newly created test cases on both methods. If the methods produce different outputs for the same test case, it indicates that their behavior diverges under certain conditions, thereby demonstrating a functional difference between them.

Finally, we record those method pairs that are confirmed to be valid FE method pairs and use these pairs to construct a dataset.

4 Dataset

In this section, we describe the dataset we constructed. The dataset is built using source code from ManyTypes4Py [3], which includes approximately 5.1k open-source Python projects. From these, we extracted a total of 1,500,000 methods to build our dataset. In STEP-1, based on research requirements, we retained 28,356 methods for type inference. In STEP-2, these methods were grouped into 726 categories according to the results of the type inference. After filtering out groups with only one method, we proceeded with the subsequent steps. In STEP-3, we used Pynguin to generate test cases for the remaining methods automatically. After processing the test cases and removing those with 'xfail' markers and those without assert statements, we rechecked that the coverage of the test cases was 100%. Ultimately, we had 2,434 methods and their corresponding test cases that met the requirements for mutual execution. These were divided into 129 groups, with the largest group containing 528 methods. In STEP-4, we identified a total of 7,415 potential FE method pairs. These pairs were then subjected to manual checking. We manually checked 750 candidate FE method pairs and identified 130 valid FE method pairs.

The number of candidate FE method pairs obtained in STEP-4 is quite large. Due to the limitations of automatically generated test cases, it is challenging to detect functional differences in methods that involve string manipulations. This also applies to methods that return boolean values, as it is difficult to capture edge cases with a limited number of test cases. This introduces some challenges for manual checking.

To address this, we adopted the following approach to extract a subset of candidate FE method pairs for manual checking: For each method pair, if neither method has been inspected in any previous pair, the pair is selected for checking. The current pair is skipped if either method has already been included in a previously inspected pair. After this filtering process, the number of method pairs requiring checking was reduced to 750. We spent approximately 10 hours manually checking these pairs, ultimately identifying 130 valid FE method pairs.

Table 1: Schema for the `methods` Table

Column Name	Data Type	Description
signature	STRING	Method signature
name	STRING	Method name
rtext	BLOB	Raw text of the method
ntext	BLOB	Normalized text of the method
size	INT	Lines of code
branches	INT	Number of branches
hash	BLOB	Hash of ntext
path	STRING	File path of the method
start	INT	Start line of the method
end	INT	End line of the method
repo	STRING	Project repository name
revision	STRING	not used in this dataset
compilable	INT	The methods used in STEP-4
tests	INT	not used in this dataset
Target_ESTest	BLOB	Automatically generated test cases
Target_Tesecase	BLOB	The test cases used in STEP-4
groupID	INT	Group identifier for the method
id	INTEGER	Method ID

In the end, We constructed the dataset and published it on GitHub. The dataset consists of three tables: `methods`, `pairs`, and `verifiedpairs`. The methods table records all relevant information about each method, including the original method, its normalized version, the number of lines in the method, the test cases generated for the method, and the method's grouping information (see Table 1). The pairs table contains all the candidate equivalent method pairs obtained in STEP-4. Each method pair is linked to the corresponding original methods in the methods table based on the pair's information. Additionally, every candidate method pair has a unique ID. The verifiedpairs table records the IDs of the method pairs that have been manually verified as functionally equivalent.

Figure 5 shows an example of FE method pairs identified in STEP-5. Both methods calculate the sum of all integers from s to `e-1` but differ in their implementation.

The `sum1d` method uses `a` for loop to iterate over all integers from `s` to `e-1`, with range `(s,`

```
def sum1d(s:int, e:int) -> int:
    c = 0
    for i in range(s, e):
         c += i
    return c
```

(a) Method sum1d

```
def while_count(s:int, e:int) -> int:
    i = s
    c = 0
    while i < e:
         c += i
         i += 1
    return c
```

(b) Method while_count

Figure 5: Example of FE method pairs.

`e)` generating a sequence of integers from `s` to `e-1`. The for loop automatically iterates through this sequence.

In contrast, the `while_count` method uses a while loop for accumulation. The while loop requires manual updating of the loop variable `i` and checking the loop condition `i < e`. Here, `i` starts from `s`, and `i` is incremented by 1 in each iteration.

Figure 6 shows an example of functionally non-equivalent method pairs identified in STEP-5. Both methods are designed to calculate the greatest common divisor (GCD) of two integers using the same algorithm—the Euclidean algorithm. However, differences in specific implementation details lead to divergent outputs for some inputs.

The `gcd` method is a classical implementation of the Euclidean algorithm, which iteratively swaps `(a, b)` in a while loop under the condition `a` is not equal to 0 until `a` becomes 0, at which point it returns `b`. The `mutated_gcd` method also implements the Euclidean algorithm but adds an if statement at the start to ensure that `a` is always greater than or equal to `b`. If `a` is less than `b`, the values of `a` and `b` are swapped. The swapping continues inside the while loop until `b` equals 0, at which point it returns `a`.

When both input parameters are either positive or negative, the two methods return the same result regardless of the values of `a` and `b`. However, when `a` and `b` have opposite signs and `a` is greater than `b`, the results differ in terms of their signs. This discrepancy arises because the termination condition of the while loop in the two methods depends on different variables `gcd`

```
def gcd(a: int, b: int) -> int:
    while a != 0:
        (a, b) = (b % a, a)
    return b
```

(a) Method gcd

```
def mutated_gcd(a: int, b: int) -> int:
    if a < b:
        (a, b) = (b, a)
    while b != 0:
        (a, b) = (b, a % b)
    return a
```

(b) Method mutated_gcd

Figure 6: Example of functionally non-equivalent method pairs.

relies on `a`. In contrast, `mutated_gcd` relies on `b`. For instance, with the input `(12, -8)`, the `gcd` method returns `4`, while the `mutated_gcd` method returns `-4`. This difference went undetected in the test cases primarily because the `mutated_gcd` method only checks the relative magnitude of `a` and `b` and does not account for the signs of the numbers.

5 Performance Evaluation of Functionally Equivalent Methods

In modern software development, functional correctness and execution efficiency are two critical dimensions for evaluating code quality. Functionally equivalent methods exhibit identical functional behavior and correctness, yet their execution performance often varies due to differences in implementation. This performance disparity is particularly significant in Python, a relatively slower language, when handling large-scale data [26]. By evaluating the efficiency of functionally equivalent method pairs, developers can gain insights into the performance differences across various implementations, thereby laying a theoretical foundation for writing more efficient code.

Python ' s simplicity and flexibility attract developers from diverse backgrounds, resulting in varying programming habits and practices. While professional software engineers may prioritize performance-optimized implementations, developers from non-computer science fields, such as scientists or analysts, often focus more on code readability and ease of implementation. This diversity leads to multiple implementation approaches for the same functionality, with method selection often based on personal experience or readily available library support, rather than systematic performance analysis. Furthermore, in Python's open-source ecosystem, many codebases offer multiple implementations of the same functionality [21]. For instance, removing duplicates from a list can be achieved using a `set` conversion, iterative loops, or third-party libraries, while string processing and data structure operations present even more alternatives. These variations may arise from evolving requirements or updates in library versions. While such flexibility underscores Python's versatility, it also leaves developers without clear criteria for choosing the most efficient implementation.

Evaluating the performance of functionally equivalent methods provides not only objective references for developers from different backgrounds but also valuable insights for tool development [2]. For example, the results of performance testing can be integrated into code analysis tools, enabling them to recommend more efficient code snippets to developers.

Through this study, we have quantified the performance differences among functionally equivalent method pairs across various scenarios and analyzed the underlying causes of these disparities. These findings help developers better understand the strengths and weaknesses of different implementations, allowing them to avoid unnecessary performance bottlenecks caused by suboptimal code choices.

5.1 Execution Speed Measurement

In the performance evaluation process, the first step is to accurately and reliably measure the execution time of each method. To ensure the validity and precision of the measurement results, we employed Python's built-in performance measurement tool, the `timeit` [17] module. The `timeit` module allows for precise recording of the execution time of code blocks or methods, eliminating errors caused by external environment factors or system load. This makes it a widely used tool for performance testing of Python methods.

Specifically, for each method to be evaluated, we used multiple test cases and set the number of executions for each method to 10,000. This approach helps reduce the impact of incidental fluctuations or system load by increasing the number of executions, ensuring that the measurement results are representative. By executing the tests multiple times, we can also avoid errors caused by occasional system fluctuations or other external factors during testing. During the process, we recorded the total execution time of each method while executing all test cases.

To ensure better test coverage, we utilized the test cases previously automatically generated by Pynguin. When performing performance evaluation, we merged the test cases of two methods in each functional equivalent method pairs into a unified test set, ensuring that both methods were compared under identical test conditions. This process not only guarantees the diversity of method executions but also ensures test case coverage.

Throughout the execution, we strictly controlled the testing environment to ensure that each method ran under the same hardware and software conditions, avoiding environmental differences that could interfere with execution time. With these measures in place, we were able to obtain accurate and comparable execution time data, providing a solid foundation for subsequent performance comparisons, optimization analysis, and result interpretation.

5.2 Comparison of Method Execution Times

Then we compare the execution time differences between two methods. To quantify the performance differences between methods, we calculate the time ratio by using the execution times of the two methods. Specifically, for each pair of functionally equivalent methods, we use the longer execution time as the numerator and the shorter execution time as the denominator to compute their time ratio. This method allows us to visually understand the time differences between two methods and provides a numerical measure of the difference.

For each pair of methods M_1 and M_2, we first measure their execution times, denoted as T_1 and T_2, respectively. Then, we calculate the time ratio using the following formula:

$$\text{Time Ratio} = \frac{\max(T_1, T_2)}{\min(T_1, T_2)}$$

Here, $\max(T_1, T_2)$ represents the longer execution time, and $\min(T_1, T_2)$ represents the shorter execution time. This ratio clearly reflects the time difference between the two methods. A ratio close to 1 indicates that the performance of the two methods is similar, while a larger ratio suggests a significant performance difference.

Table 2: Execution Time Ratio Distribution

Execution Time Ratio	**Count**	**Percentage**
Greater than 1 and less than 1.5	96	73.85%
Greater than 1.5 and less than 2	20	15.38%
Greater than 2 and less than 5	10	7.69%
Greater than 5	4	3.08%
Total	130	100.00%

In the analysis of performance differences, we calculated the execution time ratio for each pair of methods and categorized the results based on the magnitude of the ratio. These calculations provide a clear view of the execution time disparities between different method pairs. Table 2 illustrates the distribution of execution time ratios.

It can be observed that the vast majority of method pairs have execution time ratios between 1 and 1.5, accounting for approximately 73.85%. Additionally, 15.38% of method pairs fall within the 1.5 to 2 range. Larger performance disparities, represented by ratios greater than 2, are relatively rare, with 7.69% of method pairs falling in the 2 to 5 range and only 3.08% exceeding a ratio of 5.

5.3 Analysis of Performance Differences

Building on the execution time analysis, we further utilized Python's performance profiling tool, `cProfile` [16], to conduct a detailed performance analysis of each method pair. By examining the function call patterns within the methods, we gained a clearer understanding of which operations had a decisive impact on performance differences. Based on the results of our analysis, we identified three primary factors contributing to these performance disparities:

1. Generator expressions or list comprehensions

```python
def method1(n: int) -> bool:
    return not any((n // i == n / i for i in range(n - 1, 1, -1)))
```

(a) Method 1

```python
def method2(x: int) -> bool:
    for i in range(2, int(x ** 0.5)):
        if x % i == 0:
            return False
    return True
```

(b) Method 2

Figure 7: Examples of differences in Generator expressions

Generator expressions and list comprehensions are unique Python syntax features. When dealing with large-scale data, the lazy evaluation of generators can improve efficiency. However, for smaller problems, both approaches introduce some additional overhead compared to using a direct for-loop, as they inherently involve function calls.

For example, in the case shown in Figure 7, both methods are used to check whether a number is prime. Method 1 employs a generator expression, while Method 2 opts for a direct for-loop. During this process, Method 1 incurs multiple function calls, including the call to the built-in any function. These additional function calls result in a significant performance difference between the two methods.

2. Excessive calls to built-in functions

```python
def method1(char: str) -> bool:
    return char.isascii() and char.isalpha()
```

(a) Method 1

```python
def method2(input_str: str) -> bool:
    flag = [False] * 26
    for char in input_str:
        if char.islower():
            flag[ord(char) - 97] = True
        elif char.isupper():
            flag[ord(char) - 65] = True
    return all(flag)
```

(b) Method 2

Figure 8: Examples of differences in built-in function call

Frequent calls to built-in functions within a method can lead to significant performance differences. Built-in functions often involve additional checks and computations, especially when handling complex data or performing multiple operations. Repeatedly calling a built-in function not only incurs the overhead of method calls but also directly affects the performance of the entire method, as the efficiency of the function itself plays a crucial role.

In the example shown in Figure 8, Method 1 calls 'isascii()' and 'isalpha()' only once. In contrast, Method 2 repeatedly calls 'islower()' and 'isupper()' within a loop, resulting in a significant increase in the overhead of function calls. This is the main cause of the performance difference between the two methods.

3. Differences in algorithm or calculation details

   Different methods may employ entirely distinct algorithms or computational logic. Even when functionally equivalent, differences in algorithmic complexity. Additionally, some methods may share the same computational goals but use different calculation processes, which can also impact execution efficiency.

```
def f1(b: int, e: int, m: int) -> int:
    if e == 0:
        return 1
    t = f1(b, e // 2, m) ** 2 % m
    if e & 1:
        t = t * b % m
    return t
```

(a) Method 1

```
def f2(x: int, n: int, m: int) -> int:
    res: int = 1
    if n > 0:
        res = f2(x, int(n / 2), m)
        if n % 2 == 0:
            res = res * res % m
        else:
            res = res * res % m * x % m
    return res
```

(b) Method 2

Figure 9: Examples of differences in computational details

We can refer to the example shown in Figure 9. The two methods are functionally equivalent, as they both implement modular exponentiation, which computes the result of $b^e \mod m$ (where b is the base, e is the exponent, and m is the modulus). However, there are certain differences in the calculation details that lead to performance discrepancies. Method 1 uses Python's power operator for squaring, which involves an implicit function call. In contrast, Method 2 directly multiplies the variable by itself.

6 Accuracy Evaluation of Large Language Models

In recent years, with the rapid development of natural language processing technologies, particularly the significant achievements of large language models (LLMs) in various domains, artificial intelligence has shown tremendous potential in code analysis, automated programming, and program comprehension. State-of-the-art LLMs, such as GPT-4, have demonstrated outstanding performance in a wide range of programming tasks, especially in code generation [6], bug fixing [8], code summarization [20], and code explanation [3]. These models possess the ability to understand and generate code, aiding developers in writing and debugging programs more efficiently.

However, despite the impressive accomplishments of LLMs in code handling, their performance in more complex code analysis tasks, particularly their ability to recognize functional equivalence between code snippets, still warrants deeper investigation. This is especially true for dynamic languages like Python. The challenge lies in correctly identifying functionally equivalent methods that may differ in implementation. Such capabilities are of significant practical value in fields like code optimization, refactoring, and code review.

By evaluating the performance of LLMs in recognizing functionally equivalent pairs, we not only gain a deeper understanding of their capabilities in handling complex programming tasks but also provide theoretical support for their applications in program comprehension, automated refactoring, and code optimization. If LLMs can accurately recognize functionally equivalent pairs, they will have far-reaching implications. Developers could leverage LLMs to automate code reviews and refactoring processes, saving considerable manual inspection time and improving code quality. In the educational field, the ability of LLMs to determine equivalence could make computer programming instruction more efficient, especially in automated code reviews and programming exercises, helping students identify and understand potential issues in their code in real-time. For automated tools, integrating LLMs into code optimization and static analysis tools would enhance their functionality, thus better supporting development teams in decision-making during software maintenance and upgrades.

6.1 Model Selection

This study selected the GPT-4o model as the subject of investigation. GPT-4o is one of the most advanced language models currently available, excelling in tasks such as code generation, bug fixing, and code explanation. It exhibits exceptional capabilities in natural language processing and understanding programming tasks. The reasons for selecting GPT-4o are as follows:

- **Performance Superiority**: GPT-4o has demonstrated outstanding performance across a

wide range of programming tasks. Its multimodal understanding capabilities make it highly effective in handling complex code analysis problems.

- **Scalability**: GPT-4o supports multiple languages and tasks, showing strong adaptability, especially for dynamic languages like Python.
- **Contextual Understanding of Code**: Compared to other language models, GPT-4o has a stronger ability to capture the semantics and logic of code, making it suitable for tasks such as identifying functional equivalence.
- **Zero-Shot Learning Capability**: GPT-4o has demonstrated remarkable reasoning ability under zero-shot conditions, enabling it to perform complex tasks without requiring additional fine-tuning.

By selecting GPT-4o, this study aims to leverage its state-of-the-art capabilities to evaluate its performance in identifying functionally equivalent method pairs, thereby providing insights into the potential of language models in complex programming tasks.

6.2 Prompt Design

This study employs a zero-shot prompt to evaluate GPT-4o's ability to recognize functionally equivalent code pairs. We chose a zero-shot prompt [10] as the basis for our prompting approach.Under a zero-shot setting, the model relies solely on its pre-trained knowledge to make judgments about the input code pairs without requiring additional fine-tuning. The prompt is designed to be concise and clear, ensuring that the model understands the task requirements and that the experiment remains consistent and reproducible.

The core components of the prompt are as follows:

- **Task Description**: Provide a definition of functional equivalence and briefly explain the objective of the task.
- **Input Format**: Present the raw code of two Python methods.
- **Output Requirement**: The model is required to answer only "Yes" or "No".

The template for the prompt used in the experiment is as Firgue9.

In the experiment, {Method_1} and {Method_2} are replaced with two Python code snippets from the dataset. This prompt design is straightforward and directs the model to focus on the task of equivalence judgment while providing a foundation for analyzing its results.

Imagine you are an experienced software developer. Your task is to analyze the functionality of the following two methods and determine whether they are functionally equivalent.

Functionally equivalent methods refer to methods that may differ in implementation but are equivalent in functionality. Functionally equivalent methods are characterized by producing identical outputs when provided with identical inputs.

Here are the source code of two methods:
Method 1:
```python
{Method_1}
```
Method 2:
```python
{Method_2}
```
Are these two methods functionally equivalent? Please respond with either "Yes" or "No".

Figure 10: Template of prompt

6.3 Evaluation Results

In this subsection, we present the evaluation results of GPT-4o on the task of recognizing functional equivalence between Python methods using the manually verified method pairs in the third table of our dataset. The dataset includes 130 functionally equivalent method pairs and 621 functionally non-equivalent method pairs. To assess the model ' s performance, following [23] ,we used three evaluation metrics: **Precision**, **Recall**, **Accuracy**.

- **Precision**: The proportion of true positive predictions among all positive predictions.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

- **Recall**: The proportion of true positive predictions among all actual positive instances.

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

- **Accuracy**: The proportion of correct predictions among all predictions.

$$\text{Accuracy} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total Samples}}$$

The test results for all method pairs are as follows:

Pair Type	**Actual Count**	**Predicted as Equivalent**	**Predicted as Non-Equivalent**
Equivalent	130	100 (TP)	30 (FN)
Non-Equivalent	621	84 (FP)	537 (TN)
Total	**751**	**184**	**567**

Using the above results, the metrics were computed as follows:

- **Precision**:
$$\text{Precision} = \frac{100}{100 + 84} = 0.543\,(54.3\%)$$
- **Recall**:
$$\text{Recall} = \frac{100}{100 + 30} = 0.769\,(76.9\%)$$
- **Accuracy**:
$$\text{Accuracy} = \frac{100 + 537}{130 + 621} = \frac{637}{751} = 0.848\,(84.8\%)$$

The results indicate that GPT-4o achieves high accuracy (84.8%) in distinguishing functionally equivalent and non-equivalent methods. While the recall of 76.9% suggests that the model successfully identifies most equivalent pairs, the relatively lower precision of 54.3% highlights a tendency to misclassify non-equivalent pairs as equivalent. This imbalance warrants further investigation and refinement of the prompt design or model behavior for enhanced precision in real-world applications.

7 Related work

This research is inspired by the FEMPdataset study [4]. In FEMPdataset's work, a dataset of 1,342 FE method pairs in Java was constructed by automatically generating test cases and mutual execution. This paper primarily extends that approach to Python, with several key differences outlined below.

- The IJADataset used in FEMPdataset contains approximately 314 million lines of code, from which 23 million methods were extracted. In contrast, this study uses the ManyTypes4Py database, extracting 1.5 million methods. Additionally, the size of the constructed datasets differs: FEMPdataset includes 1,342 FE method pairs, whereas the dataset in this paper contains 130 FE method pairs.
- In FEMPdataset, Java method types were directly used for grouping. However, since Python lacks static type checking, this study uses TypeT5 for type inference to facilitate the grouping and mutual execution process.
- In FEMPdataset, test execution was skipped when fewer than five test cases were generated. In this study, no such limitation was imposed. Due to differences in the test case generation tools, this study checked test case coverage, retaining only those test cases with 100% branch coverage.
- During the final manual checking phase, FEMPdataset ' s candidate functionally equivalent pairs were evaluated independently by three individuals. In this study, I conducted the visual checking alone. However, for pairs deemed non-equivalent, I generated new test cases to demonstrate their functional differences.

8 Conclusion

In this study, we extracted Python methods from open-source projects and automatically generated test cases for them. These generated test cases were then mutually executed to identify candidate functionally equivalent method pairs. We manually checked a subset of these candidate FE method pairs. Ultimately, from all candidate functionally equivalent pairs, a total of 731 pairs were selected for manual verification, of which 130 pairs were confirmed to be functionally equivalent.

We then conducted performance difference tests on the functionally equivalent method pairs in the dataset and performed an in-depth analysis of the main factors contributing to these differences. Additionally, we leveraged this dataset to evaluate the ability of large language models (LLMs) in identifying functionally equivalent method pairs. The experimental results showed that GPT-4 was able to recognize some method pairs with different implementations but functionally equivalent, highlighting the significant potential of large language models in this task.

Currently, one of the primary challenges of this research is the large number of candidate functionally equivalent method pairs. Manual checking of all these pairs is impractical. This issue primarily arises from the low quality of the automatically generated test cases. To address this, we plan to develop a better filtering process for the test cases to reduce the number of method pairs requiring manual checking. Enhancing the quality of the test cases will be crucial in improving the efficiency and accuracy of identifying functionally equivalent method pairs.

Acknowledgement

As I complete this thesis, I would like to express my heartfelt gratitude to everyone who has supported and guided me throughout my graduate studies.

First and foremost, I am deeply grateful to my advisor, Professor Yoshiki Higo, for giving me the invaluable opportunity to study and conduct research at Osaka University. Your insightful guidance, encouragement, and support have been instrumental in shaping my academic journey and inspiring me to strive for excellence.

I would also like to extend my sincere thanks to my senior colleague, Yang Shiyu, and my tutor, Tabata Akihiro, for their unwavering support and invaluable advice during my studies. Additionally, I am truly thankful to Mrs. Mizuho Karube, for the care and assistance you provided in my daily life. Your warm support ensured that I could focus fully on my studies and research.

I am deeply indebted to my parents for their unwavering love and support. Your encouragement and sacrifices have been the foundation of my journey, and I am forever grateful for everything you have done for me.

Lastly, I want to thank everyone who has helped me grow over the past two years. Whether through academic guidance, companionship, or words of encouragement, your kindness has made a lasting impact on my life and filled me with hope and determination for the future.

Thank you all from the bottom of my heart!

References

[1] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[2] Arthur Crapé and Lieven Eeckhout. A rigorous benchmarking and performance analysis methodology for python workloads. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 83–93, 2020.

[3] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A comparative analysis of large language models for code documentation generation, 2024.

[4] Yoshiki HIGO. Dataset of functionally equivalent java methods and its application to evaluating clone detection tools. *IEICE Transactions on Information and Systems*, E107.D(6):751–760, 2024.

[5] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, and Kazuya Yasuda. Constructing dataset of functionally equivalent java methods using automated test generation techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 682–686, 2022.

[6] Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. Knowledge-aware code generation with large language models, 2024.

[7] TIOBE Index. Tiobe index for january 2025, 2025. [Online; accessed 2025-01-28].

[8] Nafis Tanveer Islam, Mohammad Bahrami Karkevandi, and Peyman Najafirad. Code security vulnerability repair using reinforcement learning with large language models, 2024.

[9] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, 2007.

[10] Mohamad Khajezade, Jie JW Wu, Fatemeh Hendijani Fard, Gema Rodríguez-Pérez, and Mohamed Sami Shehata. Investigating the efficacy of large language models for code clone detection, 2024.

[11] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE*, pages 168–172. ACM/IEEE, May 2022.

[12] A. M. Mir, E. Latoskinas, and G. Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 585–589. IEEE Computer Society, May 2021.

[13] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.

[14] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE ' 22, page 2019–2030. ACM, May 2022.

[15] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R. Lyu. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors, 2023.

[16] Python Software Foundation. cProfile: Profile execution time of a program.

[17] Python Software Foundation. timeit: Measure execution time of small code snippets.

[18] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[19] K. R. Srinath. Python – the fastest growing programming language. 2017.

[20] Chia-Yi Su and Collin McMillan. Distilled gpt for source code summarization, 2024.

[21] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 644–655, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*, 2023.

[23] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. Comparison and evaluation of clone detection techniques with different code representations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 332–344, 2023.

[24] Jiayi Wei, Greg Durrett, and Isil Dillig. Typet5: Seq2seq type inference using static analysis. In *The Eleventh International Conference on Learning Representations*, 2023.

[25] Shafiq Joty Steven C.H. Hoi Yue Wang, Weishi Wang. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.

[26] Farzeen Zehra, Maha Javed, Darakhshan Khan, and Maria Pasha. Comparative analysis of c++ and python in terms of memory and time. *Preprints*, December 2020.