PAPER

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

Shiyu YANG[†a)], **Yusheng GUO**[†b)], **Akihiro TABATA**[†c)], ***Nonmembers***, ***and*** **Yoshiki HIGO**[†d)], ***Member***

SUMMARY As one of the most widely used programming languages in modern software development, Python hosts a vast open-source codebase on GitHub, where code reuse is widespread. This study leverages open-source Python projects from GitHub and applies automated testing to discover pairs of functionally equivalent methods. We collected and processed methods from 5,100 Python repositories, but Python's lack of static type checking presented unique challenges for grouping these methods. To address this, we conducted detailed type inference and organized methods based on their inferred types, providing a structured foundation for subsequent analysis. We then employed automated test generation to produce unit tests for each method, running them against one another within their respective groups to identify candidate pairs that yielded identical outputs from the same inputs. Through manual verification, we ultimately identified 68 functionally equivalent method pairs and 683 functionally non-equivalent pairs. These pairs were compiled into a comprehensive dataset, serving as the basis for further examination. With this dataset, we not only evaluated the ability of large language models (LLMs) to recognize functional equivalence, evaluating both their accuracy and the challenges posed by diverse implementations, but also conducted a systematic performance evaluation of equivalent methods, measuring execution times and analyzing the underlying causes of efficiency differences. The findings demonstrate the potential of LLMs to identify functionally equivalent methods and highlight areas requiring further advancement.

key words: *Functionally equivalent methods, Code clones, Program analysis, Automated test generation*

1. Introduction

With the evolution of programming languages, modern languages —particularly dynamic ones like Python—have grown increasingly rich and complex in their syntactical features. Python has become a popular choice for developers worldwide due to its concise, readable syntax and powerful standard library. It has a vast open-source codebase on GitHub and an active user community. Python also supports multiple programming paradigms, including object-oriented, functional, and imperative programming, enabling developers to achieve the same functionality using various approaches.

Open-source communities host a large number of software projects containing diverse code examples. In these codebases, it is common to find methods that, while functionally equivalent, are implemented in different ways. Collecting such functionally equivalent code snippets is highly valuable for software engineering research, as they can be used to create datasets of equivalent methods. These datasets, in turn, facilitate advancements in areas such as code optimization, refactoring, and test generation. In addition, functionally equivalent methods may differ substantially in runtime efficiency depending on their implementation, and analyzing such performance variations provides practical guidance for developers when selecting among alternative implementations. However, identifying and collecting functionally equivalent methods remains a challenging task because of the wide variations in their structural implementations.

Many existing code clone detection tools, such as SourcererCC [1] and DECKARD [2], detect clones by focusing on syntactic similarities or repeated patterns in code snippets. While these tools effectively identify copied-and-pasted code or fragments with minor modifications (e.g., variable name changes), they often struggle to detect functionally equivalent yet structurally different code. Because they rely primarily on superficial similarities, deeper functional equivalences are frequently overlooked. Consequently, there is a pressing need for novel techniques and tools capable of detecting truly functionally equivalent code pairs, rather than focusing solely on syntactic resemblance.

This study's main goal is to collect functionally equivalent (FE) method pairs from open-source projects. We define FE methods as pairs of methods that return the same output for the same input. Our approach uses Pynguin [3] to automatically generate test cases for extracted methods, followed by mutual execution to identify methods exhibiting identical behavior. We then manually verify all candidate FE method pairs. In this work, we focus on the ManyTypes4Py [4] dataset, which comprises approximately 5.1k Python repositories (1.5 million methods) with type annotations. We extract methods from this dataset, perform type inference, and group the methods based on those inference results. We then automatically generate and execute test cases within each group, ultimately identifying 7,415 candidate FE method pairs and manually validating a subset of them.

Through this process, we constructed a curated dataset containing 68 confirmed FE method pairs and 683 non-equivalent pairs. We used this dataset to conduct an in-depth performance comparison among the FE pairs, identifying three main categories of performance differences.

Furthermore, we leveraged this dataset to evaluate the

Manuscript received October 8, 2015.
Manuscript revised April 1, 2024.

[†]The author is with the Graduate School of Information Science and Technology, The University of Osaka, Suita-shi, 565-0871, Japan
a) E-mail: yangsy@ist.osaka-u.ac.jp
b) E-mail: guoysh@ist.osaka-u.ac.jp
c) E-mail: a-tabata@ist.osaka-u.ac.jp
d) E-mail: higo@ist.osaka-u.ac.jp

DOI: 10.1587/transfun.E107.A.1

ability of large language models (LLMs) to recognize functional equivalences. Specifically, we tested GPT-4 to assess its effectiveness in identifying pairs of methods that differ in implementation yet produce the same output. Our results indicate that GPT-4 shows significant potential in recognizing functionally equivalent methods, even in the face of structural variations. However, challenges remain, especially for pairs with substantial structural differences. Despite these limitations, GPT-4 demonstrated considerable promise in functional equivalence recognition, suggesting productive avenues for further exploration in this area.

The main contributions of this research are as follows: (1) construction of a dataset of functionally equivalent Python methods, (2) a detailed evaluation of execution-time performance differences among equivalent implementations, including both execution-time measurement and analysis of performance factors, and (3) evaluation of LLMs in recognizing functional equivalence.

2. Background

2.1 Definition of FE Methods

FE methods are those that differ in implementation yet provide the same functionality. They are distinguished by generating identical outputs when given identical inputs, despite variations in algorithms, data structures, or coding styles. This notion of functional equivalence is especially significant in areas such as code optimization, refactoring, and clone detection. By identifying FE methods, developers can more easily pinpoint redundant code and uncover opportunities for improvement within a codebase.

A concept closely related to functional equivalence is the code clone. Code clones can be classified as follows.

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 identifier names as well as minor formatting modifications.

Type-3 Clones.
Code fragments that exhibit structural similarity but contain syntax differences to some extent.

Type-4 Clones.
Code fragments that provide the same functionality but use different implementations (often called semantic clones). These are determined based on behavior or functionality rather than superficial syntactic similarity.

Traditional clone detection tools primarily focus on identifying Type-1 and Type-2 Clones, relying on structural and syntactic resemblance. While these tools detect clones based on syntax similarity, they cannot effectively identify functionally equivalent code that differs significantly in structure.

To address this limitation, our research employs automatic generation techniques to detect functionally equivalent clones with distinct implementations, specifically Type-4 Clones. Because Type-4 Clones are defined by behavioral rather than syntactic similarity, their detection poses a significant challenge. Nonetheless, identifying these clones is crucial for refactoring and optimization efforts, as it highlights code fragments that diverge in implementation yet deliver identical functionality.

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

Previous research in this domain has examined various techniques, including automatic test case generation and mutual execution. For instance, the work presented in literature [5] utilizes mutually executed, automatically generated test cases to identify sets of FE methods. Additionally, by drawing on Borge's dataset [6], a dataset containing 276 FE method pairs was constructed. Building on similar methodologies, FEMPDataset [7] was created, consisting of 1,342 FE method pairs in Java, each validated by three independent programmers.

In studies related to FEMPDataset, FE method pairs are automatically collected by integrating static features (e.g., method signatures) and dynamic behavior (i.e., test results) of Java methods. Static features, such as return and parameter types, are used to group methods with matching signatures. EvoSuite [8] is then employed to generate test cases for methods within the same group. Since automatically generated test cases always pass for the method they are created for, such test cases can be mutually executed across methods to verify behavior. If one method passes the test cases generated for another, and vice versa, those two methods are deemed functionally equivalent. Finally, a manual inspection is performed to confirm the validity of functional equivalence despite differences in implementation.

The success of FEMPDataset underscores the effectiveness of combining automatic test case generation and mutual execution to detect FE methods. Its primary strength lies in verifying whether two methods produce identical outputs for the same inputs, thereby revealing functional equivalence even when structural implementations differ.

2.3 The Rapidly Developing Python Language

Python has experienced rapid growth in recent years, establishing itself as one of the most popular and widely used programming languages worldwide [9]. Its clear, readable syntax and powerful features have made it a go-to choice across various industries. In software development, Python is extensively used for web development, data analysis, artificial intelligence, and automated testing. According to the TIOBE [10] index and developer surveys on platforms such as Stack Overflow, Python consistently ranks among the top programming languages. This upward trend is particularly evident in the domains of data science and artificial intelligence, where its adoption has surged. Python's open-source

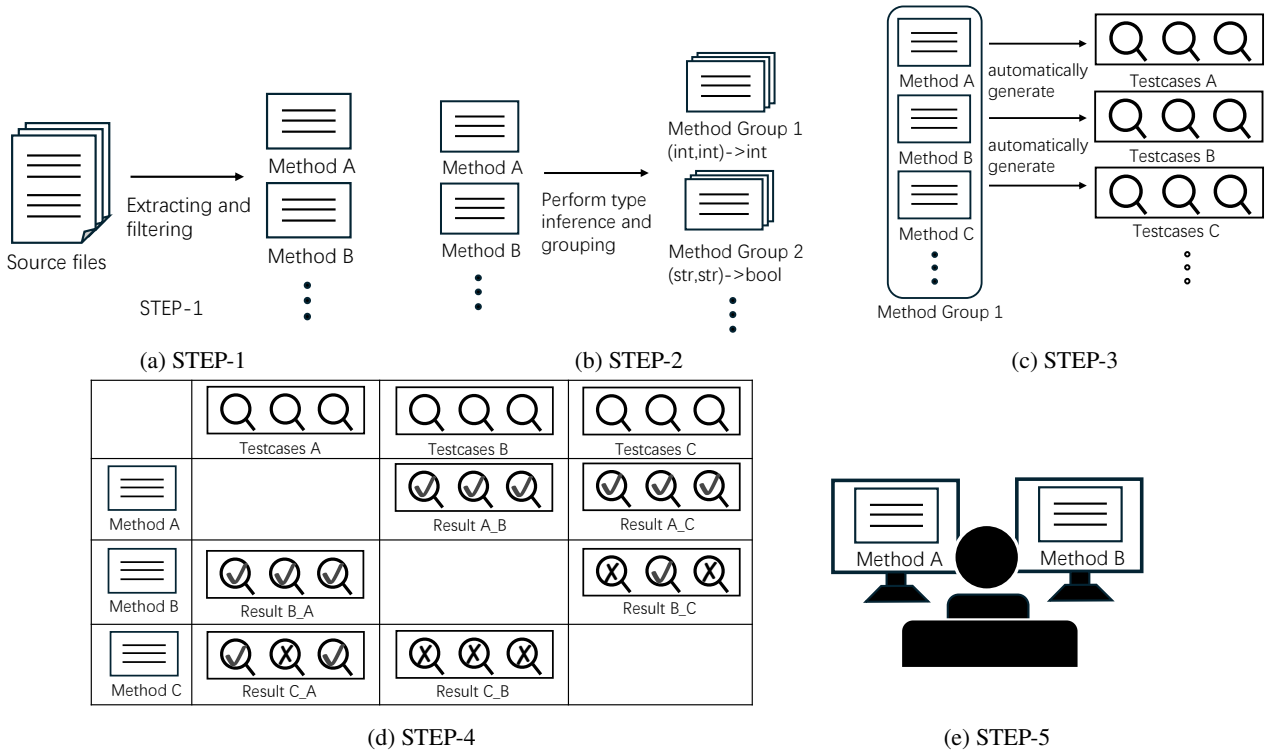

Fig. 1: Steps to obtain pairs of functionally equivalent Python methods.

nature has further encouraged contributions from diverse communities and organizations, providing developers with an abundance of tools and resources.

A key factor in Python's popularity is its relatively low learning curve. Its straightforward and intuitive syntax lowers barriers to entry for new programmers. As a result, many universities and educational institutions now use Python as the primary language for teaching programming, particularly in computer science, engineering, and data science programs. In addition, Python's extensive user community and wealth of online learning materials have greatly supported self-learners in developing and refining their skills.

On GitHub, the world's largest open-source code hosting platform, Python has also demonstrated remarkable success. Annual reports from GitHub consistently list Python among the most widely adopted languages, with many open-source projects and libraries built using it. The open and collaborative nature of the platform enables developers to easily access, modify, and optimize these codebases, fostering innovation and contributing to Python's sustained growth in the tech industry. Moreover, robust support from third-party libraries such as NumPy, Pandas, and TensorFlow has solidified Python's position as a leading language in data science, machine learning, and artificial intelligence, making it indispensable for both enterprise and research applications.

2.4 Type Inference Techniques in Python

Python, being a dynamic language, lacks the static type checking found in statically typed languages like Java and C++. Although this design choice accelerates development and offers greater flexibility, it also poses challenges for code analysis. In statically typed languages, the types of variables and methods are determined at compile time, allowing analysis tools to readily verify type consistency, detect potential type errors, and conduct in-depth static analysis. In Python, however, variable types are determined at runtime and can change during program execution, complicating attempts to perform comprehensive static type inference.

Because of Python's dynamic nature, directly applying traditional static analysis tools can be difficult, especially for tasks like clone detection and functional equivalence analysis. Many static analysis tools rely on type information and symbol tables for inference and optimization. Without explicit type declarations or constraints, these tools often fail to achieve the same level of precision in Python that they do in statically typed languages. Consequently, analyzing and optimizing Python code requires more adaptive and dynamic methods—such as type inference and runtime analysis—to address the unique challenges introduced by its dynamic features.

Nevertheless, Python's ecosystem offers robust support for tackling these issues. With the ongoing advancement of artificial intelligence and machine learning, Python type inference tools have seen significant progress. These tools combine static analysis with advanced reasoning algorithms to infer variable and method types within the code. Notable examples include Type4Py [11], CodeT5 [12], [13], and TypeT5 [14], which all utilize deep learning techniques and supplement them with formal methods and static anal-

ysis [15], [16]. They have demonstrated high accuracy in inferring basic built-in types such as *int*, *str*, and *list*, though they still encounter limitations when handling more complex types. In particular, TypeT5 leverages sequence-to-sequence technology and incorporates static analysis to accurately infer types in most Python programs, providing a robust support platform. These advancements and the increasing maturity of such tools have made it possible to build upon frameworks like FEMPDataset and integrate type inference techniques, thereby expanding research efforts to Python and enhancing both the precision and efficiency of code equivalence analysis.

2.5 Key Idea of This Study

Building on the findings from FEMPDataset, this study focuses on Python as the target language and enhances the detection of functionally equivalent method pairs by incorporating an additional type inference step. By combining type inference with automated test generation, we can more accurately identify Python method pairs that are functionally equivalent. Given Python's dynamic nature, integrating type inference and automated testing is especially crucial for improving functional equivalence detection. Ultimately, this study produces a dedicated dataset of functionally equivalent Python method pairs, providing a valuable resource and data foundation for subsequent research in areas such as code optimization, refactoring, and code review.

3. Procedure of Dataset Construction

In this study, we use the following procedure to construct a dataset of functionally equivalent (FE) method pairs:

STEP-1 Method Extraction and Initial Filtering.
Collect Python methods from open-source GitHub projects and apply an initial filtering process to remove irrelevant or trivial methods.

STEP-2 Type Inference and Grouping.
Perform type inference on each method and group them based on the inferred types.

STEP-3 Test Case Generation.
Automatically generate test cases for each method to facilitate functional verification.

STEP-4 Identification of Candidate FE Method Pairs.
Execute methods within the same group to identify candidate pairs that exhibit functional equivalence.

STEP-5 Manual Validation.
Manually verify each candidate FE method pair to confirm their functional equivalence.

Figure 1 provides an overview of the five steps. 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 in the following subsections.

3.1 STEP-1

To construct the dataset of functionally equivalent (FE) method pairs, we used the ManyTypes4Py dataset [4] as our experimental foundation. ManyTypes4Py is a Python benchmark dataset for machine learning-based type inference, consisting of 5,382 Python projects sourced from GitHub. It offers a diverse range of open-source Python projects covering various application scenarios, thereby providing a rich corpus of methods. A key rationale for choosing this dataset is its strong suitability for type inference.

```
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

Fig. 2: Example of normalization

(1) Method Extraction and Preliminary Processing.

We began by extracting all Python methods from the selected GitHub projects. First, we used Python's Abstract Syntax Tree (`AST`) module to parse every `.py` file, allowing us to generate the corresponding abstract syntax trees and identify method definitions. We then traversed these trees to collect relevant method information. In total, we extracted about 1,500,000 Python methods. For each method, we recorded the following information in our dataset:

- method name,
- original source code,
- normalized source code,
- number of statements and conditional predicates,
- file path, start line, and end line.

(2) Code Normalization.

After extracting the methods, we normalized their source code using the `ast` library. This process standardized all variables, string constants, and code indentation, helping to eliminate formatting inconsistencies and mitigate the effect of differing variable names. Due to Python's extensive standard library of built-in types, we chose not to rename external method calls.

Figure 2 illustrates this process. Subfigure 2a shows the original method, whereas subfigure 2b shows the method after normalization. During normalization, we first removed type hints and default parameter values from method declarations. Next, we renamed all variables, attribute names, and string literals. Only the method itself and its recursive calls were renamed for method calls.

```python
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

```python
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

Fig. 3: Example of type inference

```python
@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

```python
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

Fig. 4: Example of removed test cases.

Since Python has numerous built-in methods, renaming all method calls could mistakenly conflate distinct methods as duplicates. Moreover, code invoking user-defined external methods was excluded from later stages, as such calls fall outside the scope of our study.

(3) Duplicate Detection.

We generated a unique hash value for each normalized method to detect and remove duplicates. By computing this hash, we effectively identified and eliminated any repeated methods in the dataset, ensuring that every method pair would be truly distinct.

(4) Filtering Criteria.

The remaining methods were then filtered to retain only those relevant to our research. The following categories of methods were excluded.

Methods without parameters or return values.
These methods are difficult to evaluate through test cases since their behavior cannot be adequately assessed.

Methods with `self` in their parameters.
Such methods are typically instance methods in object-oriented programming. Because our study focuses on generating unit tests for standalone functions, these methods were removed.

Methods invoking external classes or methods.
To ensure each method could be analyzed in isolation, any method relying on external classes or methods was filtered out. These methods would fail during automatic test-case generation, so they were excluded in advance to streamline the process of detecting FE method pairs.

By applying these criteria, we refined our targets to 28,353 methods, all of which were better suited for subsequent analyses and for identifying candidate FE method pairs.

3.2 STEP-2

Since Python is a dynamically typed language lacking static type checking, subsequent operations can be challenging. To enhance the effectiveness of automated analysis, we first perform type inference on methods and then group them based on the inferred types. Methods with explicit type information generally yield more reliable results in automated test case generation, making the type inference step crucial.

In this study, we use the TypeT5 [14] tool for type inference. TypeT5 is a Transformer-based model specifically designed for Python type inference. It can infer the types of variables, parameters, and return values directly from source code, performing particularly well when explicit type hints are absent. By leveraging TypeT5, we obtain deeper insights into the type information of each method, establishing a solid foundation for subsequent test generation and functional equivalence detection.

For automated test case generation, we rely on type hints. If a method's type hints contain non-built-in Python types, test case generation fails immediately. Therefore, before running the inference, we preprocess developer-provided type hints to remove any non-built-in types. Next, we apply the TypeT5 tool to infer types for methods lacking explicit hints. Once type inference is complete, we recheck each method's parameters and return values to ensure that they all belong to Python's built-in types. Methods still containing non-built-in types at this stage are removed. Consequently, 21,503 methods remain, all of which have clear built-in type information and are suitable for the next step of automated test case generation.

Figure 3 illustrates a typical example of type inference. In the original code, the methods lack type hints. After applying inference (shown in red), we annotate the variable `possible` as a `list`, `prime` as an `int`, and the return value as an `int`. Based on these inferred types, we group this method with others that have parameter types `(list, int)` and a return type of `int`.

Following type inference, we group all methods by their parameter and return types, ensuring that only those sharing identical parameter and return types are placed together. In total, we obtain 726 groups, each containing at least two methods. In **STEP-4**, we will perform mutual execution of these methods within their respective groups.

3.3 STEP-3

In this step, we aimed to generate test cases for all the target methods. To accomplish this, we selected Pynguin [3], a Python-based automatic test generation tool capable of producing comprehensive test suites that thoroughly exercise a given method's functionality. A natural question arises as to why we rely on automatically generated test cases rather than the original test code in the repositories. The main rationale is that repository-level test suites vary greatly in availability, quality, and coverage, and are often written at the module or integration level, making them unsuitable for method-level equivalence checking. In contrast, Pynguin systematically generates unit tests for each extracted method, achieving high coverage (100% branch coverage in our filtering step) and ensuring that every method is evaluated under comparable conditions. This guarantees both consistency across different projects and the ability to assess functional equivalence at the granularity of individual methods, independent of external dependencies. By leveraging Pynguin, we generated test cases for all methods identified in the previous steps, ensuring that each method was adequately validated.

After generating these test cases, we performed a filtering to remove those that contained *xfail* markers or lacked `assert` statements. Figure 4 shows examples of such cases. The *xfail* marker indicates an expected failure, making these test cases unsuitable for functional equivalence analysis. Meanwhile, test cases without `assert` statements merely check whether a method executes without errors, rather than verifying its output correctness; consequently, these were also excluded. This filtering step ensures that our remaining test cases effectively validate each method's behavior rather than just running code.

Next, we conducted coverage testing on the remaining test cases using pytest's coverage component. Test coverage indicates the extent to which a test suite exercises the code. To guarantee thoroughness, we retained only those test cases that achieved 100% coverage, ensuring that all possible branches and paths of each method were tested. Test cases meeting this criterion offer a robust foundation for subsequent functional equivalence analysis.

Upon completing the filtering and coverage assessments, we obtained a high-quality set of about 6,500 test cases, a process that took roughly 40 hours. These test cases provide comprehensive validation for each method and will be used in the next step, where we perform mutual execution to identify functionally equivalent method pairs.

3.4 STEP-4

In this step, we leverage the high-quality test cases and method information retained from STEP-3, along with the grouping information from STEP-2, to conduct mutual execution among methods within each group. The main objective is to validate whether these methods are functionally equivalent through cross-testing.

1. Preparing Tests and Methods.
Consider two methods, A and B, each with its own corresponding test suite: test cases A and test cases B. These test suites were rigorously filtered in STEP-3 to ensure 100% coverage, thereby thoroughly evaluating each method's functionality.

2. Executing Tests.
First, we run method A with test cases B. This step checks whether method A can pass all of B's test cases. If it does, we then run method B with test cases A to verify whether B can pass all of A's test cases. Through this reciprocal approach, we assess both methods' behavior under each other's test conditions.

3. Analyzing Results.
If method A passes all of B's tests and method B passes all of A's tests, we consider A and B likely to be functionally equivalent under the given test cases, and mark them as *candidate FE method pairs*. Conversely, if either method fails any test case, we exclude that pair from further consideration, ensuring only pairs that consistently meet all test conditions are retained.

For the method pairs identified as *candidate FE method pairs*, additional validation and analysis are required. While the cross-testing provides preliminary evidence of functional equivalence, the scope of the test cases is inherently limited. In practical scenarios, further verification is advisable to confirm consistent behavior across all possible input conditions.

3.5 STEP-5

In this phase, we conduct a detailed manual review of all *candidate FE method pairs*. The primary goal is to determine, through human judgment, whether these pairs genuinely exhibit functional equivalence.

During the manual review, if we find that certain candidate pairs are not truly functionally equivalent, we create new test cases to highlight their functional differences. Specifically, we design inputs that might cause the methods to produce different outputs, then execute these test cases on both methods. If the two methods yield different results for the same input, it demonstrates a functional discrepancy, indicating that they are not equivalent.

Finally, we document and include in our dataset only those method pairs that are definitively confirmed as functionally equivalent.

4. Dataset

In this section, we present the dataset constructed in this study. The dataset is built from source code in ManyTypes4Py [4], which includes approximately 5.1k open-source Python projects. From these projects, we extracted about 1,500,000 methods. In **STEP-1**, based on our research objectives, we retained 28,356 methods suitable for type inference. In **STEP-2**, these methods were grouped into 726 categories according to their inferred types; groups containing only one method were excluded from subsequent steps.

In **STEP-3**, we used Pynguin to automatically generate test cases for the remaining methods. After filtering out test cases marked with `xfail` or lacking `assert` statements and ensuring 100% code coverage, we were left with 2,434 methods, each paired with its corresponding test suite. These methods were divided into 129 groups, with the largest group containing 528 methods. In **STEP-4**, we identified 7,415 potential FE (functionally equivalent) method pairs, which were then manually inspected. Ultimately, in **STEP-5**, we manually examined 751 of these pairs and confirmed that 68 pairs were truly functionally equivalent.

(1) Manual Checking Approach.

The number of *candidate FE method pairs* in **STEP-4** was large. Due to limitations in automatically generated test cases—particularly for methods involving string manipulations or boolean return values—detecting functional differences can be challenging with limited test coverage. This complexity necessitated a pragmatic manual inspection strategy: if neither method in a pair had appeared in any previously checked pair, we included that pair in our manual verification process. If one or both methods had already been inspected, we skipped that pair. This approach reduced the number of pairs requiring manual review to 751, a task that took roughly 10 hours. Of these, we identified 68 valid FE method pairs.

(2) Dataset Organization.

Finally, we published our dataset on GitHub[†]. The dataset comprises the following three tables.

Methods. Records all pertinent information about each method, including the original and normalized source code, method length (in lines), the generated test cases, and group information.

Pairs. Contains all candidate equivalent method pairs from **STEP-4**, linking each pair to the corresponding original methods in the `methods` table. Each pair is assigned a unique ID.

VerifiedPairs. Lists the IDs of method pairs that were manually confirmed to be functionally equivalent.

To facilitate the use of the dataset, Appendix A provides example SQL queries demonstrating how to access and manipulate the database.

(3) Examples of FE and Non-FE Method Pairs.

Figure 5 illustrates two methods identified in **STEP-5** as FE. Both compute the sum of all integers from `s` up to (but not including) `e`, yet employ different implementation strategies. Method sum1d uses a `for` loop and `range(s, e)` to automatically iterate over integers from `s` to `e`-1. On the other hand, method while_count uses a `while` loop for accumulation, manually incrementing the loop variable `i` and checking '`i < e`' in each iteration.

Figure 6 shows an example of two methods deemed

[†]https://github.com/Scepter4Qing/PyFuncEquivDataset

```
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

Fig. 5: Example of FE method pairs.

non-equivalent in **STEP-5**, both of which use the Euclidean algorithm to compute the greatest common divisor (GCD). Although they apply the same algorithm, the additional `if` statement in `mutated_gcd` leads to discrepancies when handling input parameters with opposite signs. Method gcd iteratively swaps `(a, b)` while `a` is not zero, then returns `b`. On the other hand, method mutated_gcd ensures `a` is always greater than or equal to `b` before entering the loop and continues until `b` is zero, then returns `a`. When `a` and `b` have the same sign, both methods yield identical results. However, if `a` and `b` have opposite signs (e.g., `(12, -8)`), gcd returns `4`, whereas mutated_gcd returns `-4`. This discrepancy went undetected by the automatically generated test cases, highlighting the importance of thorough manual inspection for certain edge cases.

5. Performance Evaluation of Functionally Equivalent Methods

In modern software development, both functional correctness and execution efficiency are vital metrics for evaluating code quality. While functionally equivalent methods exhibit the same functional behavior and correctness, their execution performance can vary substantially due to differences in implementation. In Python—known for relatively slower execution—this performance gap is particularly noticeable when handling large-scale data[17]. By comparing the efficiency of functionally equivalent method pairs, developers gain insights into performance variations across different implementations, thereby establishing a theoretical basis for writing more efficient code.

Python's simplicity and flexibility appeal to developers with diverse backgrounds, resulting in varied programming habits and practices. Professional software engineers may strive for performance-optimized implementations, whereas developers from non-computer science domains (e.g., scientists or analysts) often emphasize code clarity and simplicity. Consequently, multiple approaches emerge for the same functionality, with method selection guided more by experience or existing library support than by systematic performance analysis. Moreover, in Python's open-source ecosystem, numerous codebases contain redundant implementations of identical functionality[18]. For example, eliminat-

```
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

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

ing duplicates in a list can be handled by a `set` conversion, iterative loops, or third-party libraries, and operations on strings or data structures offer even more alternatives. These variations may stem from evolving requirements or shifts in library versions. While such diversity showcases Python's adaptability, it also leaves developers without clear, data-driven criteria for choosing the most efficient implementation.

Assessing the performance of functionally equivalent methods not only offers objective guidance to developers across different fields but also generates valuable insights for tool development[19]. For instance, performance benchmarks can be incorporated into code analysis tools, enabling them to suggest more efficient code snippets.

In this study, we evaluated performance differences among functionally equivalent method pairs under various scenarios and investigated the root causes of these variations. Our findings help developers better understand each implementation's strengths and limitations, reducing the likelihood of performance bottlenecks introduced by suboptimal code choices.

5.1 Execution Speed Measurement

In the performance evaluation process, the first task is to accurately and reliably measure each method's execution time. To ensure validity and precision, we used Python's built-in *timeit* module, which precisely measures the execution time of code blocks and methods by minimizing errors introduced by external environment factors or system load. As a result, *timeit* is widely adopted for performance testing in Python.

Concretely, for each method under evaluation, we employed multiple test cases and set the execution count to 10,000. Increasing the number of executions helps mitigate incidental fluctuations and system load variations, thereby producing representative results. Repeated executions also reduce the impact of occasional system irregularities or external factors on the measurements. During testing, we recorded each method's total execution time across all applicable test cases.

To further ensure comprehensive testing, we reused the Pynguin-generated test cases from earlier steps. In evaluating each pair of functionally equivalent methods, we merged their respective test sets into a single unified collection, ensuring both methods were compared under identical conditions. This approach not only guarantees diverse execution paths but also maintains consistent test coverage.

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

(a) Method 1

```
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

Fig. 7: Examples of differences in Generator expressions

Throughout the tests, we controlled the environment to ensure that all methods ran under identical hardware and software configurations, minimizing environmental discrepancies that could affect execution time. These measures allowed us to gather accurate, comparable timing data, forming a reliable basis for subsequent performance comparisons, optimization analysis, and result interpretation.

5.2 Comparison of Method Execution Times

Next, we compare the execution times between pairs of methods to quantify their performance differences. Specifically, we define a *time ratio* by dividing the longer execution time by the shorter one. This approach offers a clear numerical measure of the performance gap between two functionally equivalent methods.

(1) Calculation of the Time Ratio.

For each pair of methods M_1 and M_2, we first measure their execution times, denoted by T_1 and T_2, respectively. The time ratio is then computed as follows:

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

In this formula, $\max(T_1, T_2)$ and $\min(T_1, T_2)$ represent the longer and shorter execution times, respectively. A ratio close to 1 indicates near-equal performance, whereas larger ratios suggest more pronounced performance differences.

(2) Distribution of Time Ratios.

To analyze the performance gap between different method pairs, we calculated the time ratio for each pair and organized the results based on the ratio's magnitude. Table 1 illustrates the distribution of time ratios across all evaluated pairs. Most pairs (72.06%) exhibit ratios ranging from 1

Table 1: Execution Time Ratio Distribution

Execution Time Ratio	Count	Percentage
Greater than 1 and less than 1.5	49	72.06%
Greater than 1.5 and less than 2	11	16.18%
Greater than 2 and less than 5	6	8.82%
Greater than 5	2	2.94%
Total	68	100.00%

```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

Fig. 8: Examples of differences in built-in function call

to 1.5, while an additional 16.18% lie between 1.5 and 2. Ratios greater than 2 are comparatively rare, with 8.82% of pairs falling between 2 and 5, and only 2.94% exceeding 5. These findings underscore that, although many functionally equivalent methods perform similarly, certain implementations can result in substantial execution-time variations.

5.3 Analysis of Performance Differences

Building on the execution-time findings, we utilized Python's performance profiling tool, *cProfile* [20], to conduct a more fine-grained performance analysis of each method pair. By examining function call patterns, we identified which operations significantly influenced execution time. From this analysis, three primary factors emerged as drivers of performance discrepancies.

Generator Expressions or List Comprehensions.
Generator expressions and list comprehensions are distinctive Python constructs. While generator-based lazy evaluation can enhance efficiency for large-scale data, it also introduces overhead due to multiple function calls, making a simple `for` loop more efficient in smaller cases. For instance, in Figure 7, both methods determine whether a number is prime, but Method1 uses a generator expression along with the built-in `any` function, incurring additional function call overhead compared to the direct `for` loop in Method2.

Excessive Calls to Built-in Functions.
Frequent built-in function calls can markedly affect performance. Built-in functions often execute extra checks and computations, particularly for complex data or repeated operations. Thus, repeated calls not only add overhead from multiple function calls but also depend on the intrinsic efficiency of the built-in function itself. In Figure 8, Method 1 calls `isascii()` and `isalpha()` only once, whereas Method 2 repeatedly invokes `islower()` and `isupper()` in a loop, leading to noticeably higher function-call overhead.

Differences in Algorithms or Calculation Details.
Even methods that achieve the same computational goal can employ different algorithms or vary in calculation details, resulting in disparate execution times. As illustrated in Figure 9, both methods perform modular exponentiation, which

```python
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

```python
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

Fig. 9: Examples of differences in computational details

computes the result of $b^e \ mod \ m$ (where b is the base, e is the exponent, and m is the modulus). Method 1 uses Python's power operator for squaring, which invokes an implicit function call, while Method 2 multiplies the variable directly by itself. These implementation details ultimately yield different performance characteristics.

6. Accuracy Evaluation of Large Language Models

In recent years, the rapid advancement of natural language processing (NLP) technologies—particularly the significant achievements of large language models (LLMs) in various domains—has unveiled the immense potential of artificial intelligence (AI) in code analysis, automated programming, and program comprehension. State-of-the-art LLMs, such as GPT-4, have exhibited remarkable performance across diverse programming tasks, including code generation [21], bug fixing [22], code summarization [23], and code explanation [24]. These models can comprehend and generate code, thereby assisting developers in writing and debugging programs more efficiently.

Despite these impressive accomplishments, the performance of LLMs in more complex code-analysis tasks—particularly in recognizing functional equivalences among code snippets—remains underexplored. This challenge is especially pronounced in dynamic languages like Python, where two methods may implement the same functionality but exhibit different structures. Accurately identifying such functionally equivalent methods has practical relevance in areas like code optimization, refactoring, and code review.

By evaluating LLMs' ability to detect functionally equivalent pairs, we not only deepen our understanding of their capacity to handle complex programming tasks but also provide theoretical support for their application in program comprehension, automated refactoring, and code optimization. If LLMs can reliably identify functionally equivalent pairs, the implications are far-reaching. Developers could

leverage LLMs to automate code reviews and refactorings, significantly reducing the time spent on manual inspections and enhancing overall code quality. In education, accurate equivalence detection could improve the effectiveness of automated code review and programming exercises, helping students identify and understand potential issues in their work more quickly. For automated tool development, integrating LLMs into code optimization and static analysis utilities would enhance their capabilities, supporting development teams in making better-informed decisions during software maintenance and upgrades.

6.1 Model Selection

This study selects GPT-4o as the primary subject of investigation. GPT-4o is among the most advanced language models currently available, excelling in code generation, bug fixing, and code explanation tasks. Its robust natural language understanding and programming proficiency make it an ideal candidate for exploring the challenges of FE method identification. The key reasons for choosing GPT-4o are as follows.

Superior Performance.
GPT-4o consistently demonstrates exceptional results across a broad spectrum of programming tasks. Its multimodal understanding capabilities make it especially adept at handling complex code analysis problems.

Scalability.
GPT-4o supports multiple languages and tasks with strong adaptability, making it particularly well-suited to dynamic languages like Python.

Deep Contextual Understanding of Code.
Compared to other models, GPT-4o offers enhanced insight into code semantics and logic, enabling it to more effectively identify functional equivalence.

Zero-Shot Learning Potential.
GPT-4o displays remarkable reasoning ability in zero-shot settings, allowing it to tackle complex tasks without additional fine-tuning.

By examining GPT-4o, we aim to leverage its state-of-the-art capabilities to evaluate how effectively it can identify FE method pairs, thereby shedding light on the broader potential of language models for tackling complex programming challenges.

6.2 Prompt Design

This study evaluates GPT-4o's capability to recognize FE method pairs using a zero-shot prompt[25]. In a zero-shot setting, the model relies solely on its pre-trained knowledge to judge the input code pairs, without any additional fine-tuning. We designed a concise and clear prompt to ensure the model understands the task and to maintain consistency and reproducibility throughout the experiment. The prompt structure is as follows.

Task Description.

````
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".
````

Fig. 10: Template of prompt

Defines functional equivalence and briefly outlines the primary goal of the task.

Input Format.
Presents the raw code for two Python methods.

Output Requirement.
Instructs the model to respond only with `Yes` or `No`.

Figure 10 shows the template for the prompt used in this experiment. In practice, we replace Method_1 and Method_2 with Python code snippets from our dataset. This straightforward prompt design directs the model's attention to the task of equivalence judgment, forming a clear basis for subsequent result analysis.

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 68 FE method pairs and 683 non-FE method pairs. To assess the model's performance, following literature [26], we used three evaluation metrics: Precision, Recall, and 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.

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

Table 2: Experimental Results

Pair Type	Actual Count	Predicted as Equivalent	Predicted as Non-Equivalent
Equivalent	68	64 (TP)	4 (FN)
Non-Equivalent	683	107 (FP)	576 (TN)
Total	**751**	**171**	**580**

Table 3: Cross-tabulation of false positives by Functional Domain and Misclassification Reason

Functional Domain	Boundary	Error handling	Input domain	Input+Error	Partial equiv.	SB-Ann	SB-Form	SB-True	Total
String processing	2	1	1	3	1	8	17	12	45
Math / Numeric operations	4	0	1	3	3	1	9	21	42
Other / Miscellaneous	0	0	0	0	2	4	1	7	14
Container / Collection manipulation	0	0	0	1	0	3	0	1	5
File / I/O handling	0	0	0	0	0	1	0	0	1
Total	6	1	2	7	6	17	27	41	107

The test results for all method pairs are shown in Table 2. Using the above results, the metrics were computed as follows.

$$\text{Precision} = \frac{64}{64+107} = 0.374\,(37.4\%)$$

$$\text{Recall} = \frac{64}{64+4} = 0.941\,(94.1\%)$$

$$\text{Accuracy} = \frac{64+576}{68+683} = \frac{640}{751} = 0.852\,(85.2\%)$$

The results indicate that GPT-4o achieves high accuracy (85.2%) in distinguishing functionally equivalent and non-equivalent methods. While the recall of 94.1% suggests that the model successfully identifies most equivalent pairs, the relatively lower precision of 37.4% highlights a tendency to misclassify non-equivalent pairs as equivalent.

This imbalance warrants further investigation. To this end, we analyzed the false positives (non-equivalent pairs misclassified as equivalent) and grouped them into functional domains and misclassification reasons (see Appendix B for the categorization schema). Table 3 summarizes the distribution.

The results show that false positives are not random: over 70% fall into *semantic similarity bias*, where GPT-4o overgeneralizes from similar names or superficial structures and ignores subtle differences. Other systematic causes include *boundary-condition differences* (6 cases) and *input-/error-handling mismatches* (9 cases), which indicate that GPT-4o often fails on rare but critical edge cases. In terms of domains, most errors appear in *string processing* (45 cases) and *numeric operations* (42 cases), both of which are prone to subtle semantic divergences.

By grouping these non-equivalent pairs, we highlight concrete sources of misclassification. These groups not only explain the low precision but also provide actionable clues for improvement, such as designing prompts that stress edge-case handling or integrating training examples that differentiate semantically similar but functionally distinct methods.

7. Related Work

7.1 FEMPDataset

This research draws on the work of FEMPdataset [7], which constructed a dataset of 1,342 functionally equivalent (FE) method pairs in Java through automated test-case generation and mutual execution. We extend that approach to Python, with several notable distinctions outlined below.

Dataset Scale and Composition.
FEMPdataset's IJADataset includes approximately 314 million lines of code, yielding 23 million extracted methods. By contrast, we use ManyTypes4Py, extracting 1.5 million methods. Consequently, the dataset sizes also differ: FEMPdataset contains 1,342 FE method pairs, while our dataset includes 68 FE pairs.

Type Inference.
FEMPdataset directly employs Java's static types to group methods. Given that Python lacks static type checking, we use TypeT5 to infer types and facilitate grouping and mutual execution.

Test Case Execution Criteria.
In FEMPdataset, test execution was skipped if fewer than five test cases were generated. We place no such limitation. Instead, owing to the differing test-generation tools, we verify test coverage and retain only those with 100% branch coverage.

Manual Validation Process.
FEMPdataset's candidate FE pairs were evaluated by three independent reviewers. In this study, we conducted the final manual checks individually. However, for pairs deemed non-equivalent, we generated additional test cases to demonstrate their functional discrepancies.

7.2 SeqCoBench

SeqCoBench [27] constructs a benchmark for functional equivalence by applying semantic-preserving and semantic-altering transformations to MBPP Python functions, then evaluating LLMs and match-based metrics on the resulting pairs. In contrast, our dataset mines naturally occurring equivalent methods from real GitHub projects, identified through type inference, automated high-coverage test generation, cross-testing, and manual validation. Moreover, beyond LLM evaluation, we also analyze performance differ-

ences among implementations, making our dataset complementary to SeqCoBench's controlled, transformation-based setting.

7.3 Functionally Similar Clones in C/Java

A prior study systematically investigated functionally similar clones (FSCs) in C and Java using Google Code Jam submissions, finding that such clones rarely manifest syntactically and released a 58-pair benchmark [28]. In contrast, our work targets Python and constructs method-level, test-based FE pairs from GitHub via type-aware grouping, automated tests with 100% coverage, mutual execution, and manual verification. This design provides executable test suites for each pair, enabling controlled runtime comparisons of FE implementations (absent in prior work) and supporting LLM-based equivalence recognition. Overall, prior evidence on the scarcity of syntactic similarity motivates our test-based verification, and our dataset complements earlier work in language, granularity, and evaluation focus.

8. Discussion

While our study successfully constructed a dataset of functionally equivalent Python methods, the current design is constrained by the choice of Pynguin as the sole test generation tool. Pynguin only supports line and branch coverage, and we required 100% branch coverage in order to retain a method. This strict criterion ensures consistency but also contributed to the limited database size, as many methods could not meet this adequacy requirement. More advanced adequacy metrics, such as mutation scores, may offer stronger behavioral guarantees and potentially reduce the need for manual inspection, and represent a promising direction for future work.

Another limitation is that Pynguin-generated tests are based on a single automated perspective. Although effective for structural coverage, these tests may fail to expose subtle semantic differences. Complementary approaches, such as generating additional test cases with large language models (LLMs), could increase input diversity and provide broader behavioral coverage. Such hybrid strategies may help address cases where current test suites miss edge conditions.

Finally, our pipeline excluded methods that invoke external classes or methods, as these dependencies would cause failures in isolated execution. This design choice simplified analysis and ensured test reproducibility, but it also reduced the practicality and scope of the dataset. Future work could explore strategies for selectively retaining such methods, for example through mocking or dependency injection, to construct a larger and more representative dataset.

9. Conclusion

In this study, we extracted Python methods from open-source projects and automatically generated test cases for them. Using these test cases, we performed mutual executions to identify potential functionally equivalent (FE) method pairs. A subset of these pairs was manually inspected, ultimately leading to 751 candidate pairs, of which 68 were confirmed as functionally equivalent.

Next, we conducted performance evaluations on these functionally equivalent pairs to analyze the key factors contributing to their performance discrepancies. Additionally, we employed the resulting dataset to assess the ability of large language models (LLMs) to recognize functionally equivalent methods. Our experiments revealed that GPT-4 can indeed identify certain pairs of methods with differing implementations yet equivalent functionality, underscoring the considerable potential of LLMs in this domain.

A major challenge facing this research is the sizable number of candidate FE method pairs, which makes comprehensive manual validation impractical. The primary cause is the limited quality of automatically generated test cases. Moving forward, we plan to develop an improved filtering mechanism for test cases, thereby reducing the number of pairs that require manual inspection. Enhancing test-case quality will be crucial for boosting both the efficiency and precision of identifying functionally equivalent method pairs.

Acknowledgments

This research was supported by JSPS KAKENHI Japan (JP24H00692, JP21K18302, JP22H03567, JP22K11985).

References

[1] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.1157–1168, 2016.

[2] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," 29th International Conference on Software Engineering (ICSE'07), pp.96–105, 2007.

[3] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE, pp.168–172, ACM/IEEE, May 2022.

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

[5] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods using automated test generation techniques," 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), pp.682–686, 2022.

[6] H. Borges, A. Hora, and M.T. Valente, "Understanding the factors that impact the popularity of github repositories," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.334–344, 2016.

[7] Y. HIGO, "Dataset of functionally equivalent java methods and its application to evaluating clone detection tools," IEICE Transactions on Information and Systems, vol.E107.D, no.6, pp.751–760, 2024.

[8] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE), pp.416–419, ACM,

2011.
[9] K.R. Srinath, "Python – the fastest growing programming language," International Research Journal of Engineering and Technology (IRJET), vol.4, no.12, pp.354–357, 2017.
[10] T. Index, "Tiobe index for january 2025," 2025. [Online; accessed 2025-01-28].
[11] A.M. Mir, E. Latoškinas, S. Proksch, and G. Gousios, "Type4py: practical deep similarity learning-based type inference for python," Proceedings of the 44th International Conference on Software Engineering, pp.2241–2252, 2022.
[12] Y. Wang, W. Wang, S. Joty, and S.C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," EMNLP, 2021.
[13] Y. Wang, H. Le, A.D. Gotmare, N.D. Bui, J. Li, and S.C.H. Hoi, "Codet5+: Open code large language models for code understanding and generation," arXiv preprint, 2023.
[14] J. Wei, G. Durrett, and I. Dillig, "Typet5: Seq2seq type inference using static analysis," The Eleventh International Conference on Learning Representations, 2023.
[15] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu, "Static inference meets deep learning: a hybrid type inference approach for python," Proceedings of the 44th International Conference on Software Engineering, ICSE '22, p.2019–2030, ACM, May 2022.
[16] Y. Peng, S. Gao, C. Gao, Y. Huo, and M.R. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," 2023.
[17] F. Zehra, M. Javed, D. Khan, and M. Pasha, "Comparative analysis of c++ and python in terms of memory and time," Preprints, December 2020.
[18] M. Valiev, B. Vasilescu, and J. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem," Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, New York, NY, USA, p.644–655, Association for Computing Machinery, 2018.
[19] A. Crapé and L. Eeckhout, "A rigorous benchmarking and performance analysis methodology for python workloads," 2020 IEEE International Symposium on Workload Characterization (IISWC), pp.83–93, 2020.
[20] Python Software Foundation, "cProfile: Profile execution time of a program." https://docs.python.org/3/library/profile.html, 2024.
[21] T. Huang, Z. Sun, Z. Jin, G. Li, and C. Lyu, "Knowledge-aware code generation with large language models," 2024.
[22] N.T. Islam, M.B. Karkevandi, and P. Najafirad, "Code security vulnerability repair using reinforcement learning with large language models," 2024.
[23] C.Y. Su and C. McMillan, "Distilled gpt for source code summarization," 2024.
[24] S.S. Dvivedi, V. Vijay, S.L.R. Pujari, S. Lodh, and D. Kumar, "A comparative analysis of large language models for code documentation generation," 2024.
[25] M. Khajezade, J.J. Wu, F.H. Fard, G. Rodríguez-Pérez, and M.S. Shehata, "Investigating the efficacy of large language models for code clone detection," 2024.
[26] Y. Wang, Y. Ye, Y. Wu, W. Zhang, Y. Xue, and Y. Liu, "Comparison and evaluation of clone detection techniques with different code representations," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp.332–344, 2023.
[27] N. Maveli, A. Vergari, and S.B. Cohen, "What can large language models capture about code functional equivalence?," Findings of the Association for Computational Linguistics: NAACL 2025, ed. L. Chiruzzo, A. Ritter, and L. Wang, Albuquerque, New Mexico, pp.6865–6903, Association for Computational Linguistics, April 2025.
[28] S. Wagner, A. Abdulkhaleq, I. Bogicevic, J. Ostberg, and J. Ramadani, "How are functionally similar code clones syntactically different? an empirical study and a benchmark," PeerJ Computer Science, vol.2, p.e49, 2016.

Appendix A: Examples of Basic Dataset Operations

The dataset (*PyFuncEquivDataset*) is provided as an SQLite database file (`PyFuncEquivDataset.db`), available on GitHub[†]. It can be opened with the standard `sqlite3` command-line tool or any SQLite-compatible client. Please refer to the GitHub page for detailed information and download instructions. The database contains three tables: `methods`, `pairs`, and `verifiedpairs`.

Below we present example SQL queries demonstrating how to explore and utilize the dataset.

A.1 Basic Queries

(a) Count the total number of methods:

```
SELECT COUNT(*) FROM methods;
```

(b) Count the number of FE method pairs validated by the reviewer:

```
SELECT COUNT(*)
FROM verifiedpairs
WHERE reviewer = 1;
```

A.2 Retrieving Detailed Information

(a) Retrieve the source code of one verified FE method pair:

```
SELECT m1.rtext AS Method1,
       m2.rtext AS Method2
FROM verifiedpairs AS v
JOIN pairs   AS p  ON p.id = v.pairid
JOIN methods AS m1 ON m1.id = p.leftMethodID
JOIN methods AS m2 ON m2.id = p.
↪ rightMethodID
WHERE v.reviewer = 1
LIMIT 1;
```

(b) Retrieve all methods in a specific type-inferred group (e.g., group ID = 5):

```
SELECT id, name, signature
FROM methods
WHERE groupID = 5;
```

(c) Count the number of methods in each group:

```
SELECT groupID, COUNT(*)
FROM methods
GROUP BY groupID
ORDER BY COUNT(*) DESC;
```

Appendix B: False Positive Categorization Schema

Functional Domain Categories

- **Math / Numeric operations**: Functions performing

[†]https://github.com/Scepter4Qing/PyFuncEquivDataset

arithmetic, numeric calculations, or bitwise operations.
Example: gcd, prime_check, XOR on bytes, RMSE computation
- **String processing**: Functions manipulating or comparing strings.
Example: Prefix removal, case conversion, difference counting between strings
- **Container / Collection manipulation**: Functions operating on lists, sets, dicts, or iterables.
Example: Checking if two sets overlap, iterating with zip, list removal
- **File / I/O handling**: Functions dealing with file operations, path handling, or SQL/query generation.
Example: Constructing SQL statements, reading/writing files
- **Data structure / Algorithmic utilities**: Functions implementing or using classic data structures/algorithms.
Example: Binary search with bisect, heap operations with heapq, tree traversal
- **Other / Miscellaneous**: Functions that do not clearly fall into the above categories.
Example: Header formatting, checking boolean value of first argument

Reason Categories for Misclassification

- **Boundary condition difference**: Functions behave the same on typical inputs but diverge on edge cases.
Example: `zip` truncates shorter string vs `range(len())` causes `IndexError`
- **Input domain mismatch**: One function enforces input constraints, the other accepts a broader or different domain.
Example: `assert 0 <= b < 128` vs accepting any integer
- **Error handling difference**: Functions differ in handling invalid inputs: one raises an exception, the other returns a value.
Example: Returning original string if prefix not found vs raising `TypeError`
- **Partial equivalence only**: Equivalent only on a restricted sub-domain, not in general.
Example: Two XOR implementations equal only when lengths match
- **Semantic similarity bias**: GPT overgeneralizes from similar names or structures, overlooking subtle differences.
Example: Two nearly identical `col()` implementations with different boundary handling
- **Performance / Implementation detail**: Functions logically equivalent, differing only in performance or coding style (rare for FP).
Example: Loop vs built-in sum producing same result