# Analyzing Code Compatibility, Equivalence, and Similarity to Support Software Maintenance and Evolution

Submitted to
Graduate School of Information Science and Technology
The University of Osaka

January 2026

Shiyu YANG

# Abstract

Software has become a fundamental component of modern society, supporting critical infrastructure, industries, and everyday life. As software systems continue to expand in scale and complexity, maintaining and evolving them over time while ensuring reliability and adaptability has become a central challenge. In this context, developers increasingly rely on reusing existing code from open repositories and online platforms such as Stack Overflow. While such reuse accelerates development, it also introduces new risks, particularly when the reused code is incompatible with the target environment or semantically misunderstood. Addressing these challenges requires a deeper understanding of how compatibility, functional equivalence, and structural similarity interact during software evolution.

This dissertation systematically investigates how these three dimensions, compatibility, equivalence, and similarity, can be jointly analyzed to support sustainable software maintenance and evolution. The research first examines the reliability of reused online code, focusing on Python: a widely used programming language whose major versions are intentionally not backward compatible. An empirical study of Python code snippets collected from Stack Overflow demonstrates that a substantial portion of widely referenced examples fail to execute due to version incompatibilities between Python 2 and Python 3. This finding exposes hidden maintenance risks in online code reuse and underscores the critical need for automated version-aware analysis.

Building upon this empirical necessity, the dissertation introduces the browser-based tool **PyVerDetector**. It automatically detects the Python version compatibility of reused Stack Overflow code snippets in real time. By leveraging multiple Abstract Syntax Tree (AST)–based grammars, the tool identifies version-specific syntax issues and visualizes the compatible Python versions directly within the browser. Evaluation results indicate that PyVerDetector achieves accuracy comparable to existing offline analyzers while still providing lightweight, immediate feedback to developers. This demonstrates that automated compatibility checking can effectively prevent subtle version-related faults during code reuse.

Building on the insights from compatibility analysis, the dissertation extends its focus to code equivalence and similarity, exploring whether syntactically different implementations can exhibit identical functional behavior. To this end, the **PyFuncEquivDataset**, a representative dataset of Python functions, was systematically constructed from GitHub, encompassing diverse projects and implementation styles. Through automated test generation, mutual execution, and manual validation, functionally equivalent pairs were identified and analyzed. The results reveal that equivalent functions often differ substantially in structure, performance, and readability, suggesting that equivalence detection can facilitate refactoring, redundancy reduction, and performance optimization. Furthermore, by evaluating large language models (LLMs) on code equivalence and semantic similarity recognition, this study provides new insights into the emerging role of artificial intelligence in understanding software semantics.

Taken together, this research establishes a unified research framework that connects code compatibility, equivalence, and similarity as interdependent dimensions of software evolution. The dissertation contributes (1) empirical evidence of real-world compatibility issues in reused code, (2) a practical browser-based tool for real-time version-aware maintenance, and (3) a representative dataset and analytical methods for identifying functionally equivalent and similar code fragments. Collectively, these outcomes advance the understanding of software reuse and lay the foundation for intelligent, automated approaches that support reliable maintenance and sustainable evolution of large-scale software systems.

# List of Publications

## Major Publications

1. Shiyu Yang, Tetsuya Kanda, Daniel M. German, Yoshiki Higo. "Unveiling Python Version Compatibility Challenges in Code Snippets on Stack Overflow." IEICE Transactions on Information and Systems, vol. E107-D, no. 8, pp. 1007-1015, August 2024.

2. Shiyu Yang, Tetsuya Kanda, Davide Pizzolotto, Daniel M. German, Yoshiki Higo. "PyVerDetector: A Chrome Extension Detecting the Python Version of Stack Overflow Code Snippets." The 31st IEEE/ACM International Conference on Program Comprehension (ICPC 2023), Melbourne, Australia, 2023, pp. 25-29.

3. Shiyu Yang, Yusheng Guo, Akihiro Tabata, Yoshiki Higo. "Constructing a Dataset of Functionally Equivalent Python Methods Using Test Generation Techniques." IEICE Transactions on Information and Systems, 2026 (to appear).

## Related Publications

1. Shiyu Yang, Tetsuya Kanda, Katsuro Inoue. "The Effect of Python Version Upgrades on the Compilability of Code Snippets Posted on Stack Overflow." IPSJ SIG Technical Report, vol.2022-SE-211, no.28, pp.1-8, Hokkaido, Japan, July 2022.

2. Ramita Deeprom, Shiyu Yang, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul. "Challenges in Adopting LLaMA: An Empirical Study of Discussions on Stack Overflow." The 12th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2024), pp. 35–42. Chongqing, China, December 3–6, 2024.

3. Yusheng Guo, Shiyu Yang, Akihiro Tabata, Yoshiki Higo. "Finding Functionally Equivalent Methods in Python Using Automated Test Generation Techniques." IEICE Tech. Rep., vol. 124, no. 217, SS2024-16, pp. 10-15, Oct. 2024.

# Acknowledgement

First of all, I would like to express my deepest gratitude to my supervisor, Professor Yoshiki Higo, for his steady guidance, encouragement, and unwavering support throughout my doctoral studies. His profound insights, patient mentorship, and gentle but firm way of leading have shaped not only my research, but also my confidence and growth as a young scholar. I am truly fortunate to have been under his supervision; his kindness, wisdom, and the sense of reassurance he has always given me have carried me through many challenging moments. The influence of his mentorship will remain with me throughout my career, and I will always be grateful for the opportunity to learn and grow under his guidance.

I would also like to express my sincere appreciation to Professor Katsuro Inoue of Ritsumeikan University, whose vision and leadership laid the foundation for my research path. My heartfelt thanks go to Professor Daniel M. German at the University of Victoria for his thoughtful guidance and stimulating research discussions, which greatly enriched this work. I am equally grateful to Associate Professor Tetsuya Kanda at Notre Dame Seishin University for his early support and encouragement when I first embarked on this research journey, and I remain deeply appreciative of the guidance he offered at that formative stage.

I am also thankful to the other members of my thesis committee, Professor Shinji Kusumoto and Professor Tatsuhiro Tsuchiya, for their valuable feedback and insightful suggestions throughout the development of this dissertation.

I am deeply grateful to the faculty members of my laboratory: Professor Raula Gaikovina Kula, Associate Professor Makoto Matsushita, and Assistant Professor Olivier Nourry, for their generous support, constructive feedback, and continuous encouragement throughout my studies. Their guidance has played an important role in my academic growth, and I truly appreciate all that I have learned from them.

My sincere thanks also go to the former senior members of our laboratory and dear friends, Shi Qiu and Davide Pizzolotto, who generously offered advice, encouragement, and technical insights during my research.

Their support was indispensable, and I am deeply grateful for the time and effort they devoted to helping me.

I would like to express my heartfelt appreciation to our laboratory secretary, Ms. Mizuho Karube, for her tireless assistance and warm support throughout the years. I am also grateful to all members of Higo Laboratory, whose kindness, collaboration, and friendship created a stimulating and welcoming environment for both study and research.

My heartfelt thanks go to my best friend, Wendi Li, for always being by my side and bringing comfort and encouragement during challenging times. I am also indebted to my partner, Yilin Cao, whose unwavering support, patience, and understanding have sustained me throughout this journey.

Finally, I would like to express my deepest gratitude to my family: my aunt and uncle, Fenrong Xue and Lusheng Guo, and especially my parents, Shujing Xue and Pu Yang, for their unconditional love, spiritual support, and constant encouragement throughout my life. Their belief in me has been the greatest source of strength in completing this dissertation.

# Contents


**1 Introduction** **1**
- 1.1 Background and Significance of Software Maintenance . . . 1
- 1.2 Challenges in Modern Software Evolution . . . . . 5
- 1.3 Key Concepts: Compatibility, Equivalence, and Similarity . 9
- 1.4 Research Motivation and Questions . . . . . 15
- 1.5 Research Approach and Methodology . . . . . 19
- 1.6 Contributions and Significance . . . . . 23
- 1.7 Structure of the Dissertation . . . . . 25

**2 Empirical Analysis of Code Compatibility on Stack Overflow** **27**
- 2.1 Introduction . . . . . 27
- 2.2 Background . . . . . 29
  - 2.2.1 Stack Overflow . . . . . 29
  - 2.2.2 The Evolution of Python and Version Compatibility 29
- 2.3 Study Approach . . . . . 32
  - 2.3.1 Data Collection . . . . . 32
  - 2.3.2 Code Snippet Analysis . . . . . 33
- 2.4 Results . . . . . 34
  - 2.4.1 S1-RQ1: How many Python code snippets have version compatibility issues in the good answers to Stack Overflow questions? . . . . . 34
  - 2.4.2 S1-RQ2: How many of the code snippets interpretable only by Python 2 or only by Python 3 have Python version-specific identification? . . . . . 36
  - 2.4.3 S1-RQ3: How do users on Stack Overflow react and adapt to the introduction of new Python releases? . 38
- 2.5 Threats to Validity . . . . . 40
- 2.6 Related Work . . . . . 42
- 2.7 Conclusion . . . . . 43

**3 Tool Support for Python Version Compatibility Detection** **45**
- 3.1 Introduction . . . . . 45

3.2 PyVerDetector . . . 46
3.2.1 Frontend . . . 46
3.2.2 Backend . . . 47
3.2.3 Python Grammars and Extensibility . . . 48
3.3 Usage Scenarios . . . 49
3.3.1 Displaying the latest version as default . . . 49
3.3.2 Accurate display of results for selected versions . . . 49
3.4 Evaluation of Accuracy . . . 49
3.5 Related Work . . . 50
3.6 Conclusion . . . 52

**4 Detecting Functionally Equivalent or Similar Code for Software Evolution 55**
4.1 Introduction . . . 55
4.2 Background . . . 57
4.2.1 Definition of FE Methods . . . 57
4.2.2 Key Idea for Automatically Identifying Candidate FE Method Pairs in FEMPDataset . . . 58
4.2.3 The Rapidly Developing Python Language . . . 59
4.2.4 Type Inference Techniques in Python . . . 60
4.2.5 Key Idea of This Study . . . 61
4.3 Procedure of Dataset Construction . . . 61
4.3.1 STEP-1 . . . 62
4.3.2 STEP-2 . . . 64
4.3.3 STEP-3 . . . 66
4.3.4 STEP-4 . . . 67
4.3.5 STEP-5 . . . 68
4.4 Dataset . . . 68
4.5 Performance Evaluation of Functionally Equivalent Methods 71
4.5.1 Execution Speed Measurement . . . 72
4.5.2 Comparison of Method Execution Times . . . 73
4.5.3 Analysis of Performance Differences . . . 74
4.6 Accuracy Evaluation of Large Language Models . . . 75
4.6.1 Model Selection . . . 76
4.6.2 Prompt Design . . . 77
4.6.3 Evaluation Results . . . 78
4.7 Related Work . . . 80
4.7.1 FEMPDataset . . . 80
4.7.2 SeqCoBench . . . 81
4.7.3 EqBench . . . 81
4.7.4 EquiBench . . . 81
4.7.5 S4Eq (proof-oriented equivalence) . . . 82
4.7.6 Functionally Similar Clones in C/Java . . . 82

4.8 Discussion . . . . . 82
4.9 Conclusion . . . . . 83

**5 Integrated Discussion and Implications 85**
5.1 Overview . . . . . 85
5.2 Common Challenges from Version Compatibility to Functional Equivalence . . . . . 85
5.3 Scalability and Extensibility of Datasets and Tools . . . . . 86
5.4 Implications for Software Maintenance and Evolution . . . . 87
5.4.1 Bridging Online Knowledge and Real-World Codebases 87
5.4.2 Toward Behaviorally Aware Maintenance Tools . . . 87
5.4.3 Integrating Multi-Level Analyses for Evolutionary Understanding . . . . . 87

**6 Conclusion and Future Work 89**
6.1 Conclusion . . . . . 89
6.2 Key Contributions . . . . . 90
6.3 Limitations . . . . . 91
6.4 Future Directions . . . . . 91
6.4.1 Cross-Language and Cross-Ecosystem Compatibility Analysis . . . . . 91
6.4.2 Semantic-Aware Refactoring and Maintenance Support 91
6.4.3 Automated Dataset Expansion and Benchmarking . 91
6.4.4 Integrating Multi-Level Analyses into Evolution Analytics . . . . . 92

**Appendices 103**

**A Supplementary Dataset Details for Functionally Equivalent Python Methods 103**
A.1 Examples of Basic Dataset Operations . . . . . 103
A.1.1 Basic Queries . . . . . 103
A.1.2 Retrieving Detailed Information . . . . . 103
A.2 False Positive Categorization Schema . . . . . 104

# List of Figures


2.1 An example of a Q&A post on Stack Overflow . . . . . 30
2.2 Responses by Python version on Stack Overflow. . . . . 39

3.1 Overview of PyVerDetector. . . . . 47
3.2 Example of how the extension works in the default version. (Left): Parse error, (Right): Pass. . . . . 49
3.3 User manually selects a version of Python, incompatible with the given snippet. . . . . 50

4.1 Steps to obtain pairs of functionally equivalent Python methods. . . . . 62
4.2 Example of normalization. . . . . 63
4.3 Example of type inference. . . . . 65
4.4 Examples of removed test cases. . . . . 67
4.5 Examples of FE method pairs. . . . . 70
4.6 Example of functionally non-equivalent method pairs. . . . 71
4.7 Examples of differences in generator expressions. . . . . 73
4.8 Examples of differences in built-in function calls. . . . . 75
4.9 Examples of differences in computational details. . . . . 76
4.10 Template of prompt. . . . . 78

# List of Tables


2.1 Python code snippets facing compatibility issues in Stack Overflow. . . . . 36
2.2 The results of version annotation for code snippets in the good answers on Stack Overflow, which are only compatible with either Python 2 or Python 3. . . . . 37

3.1 Comparison results of PyVerDetector (PyVer) and PyComply (PyC). . . . . 50

4.1 Execution time ratio distribution. . . . . 73
4.2 Experimental results. . . . . 79
4.3 Cross-tabulation of false positives by Functional Domain and Misclassification Reason. . . . . 80

# Chapter 1

# Introduction

## 1.1 Background and Significance of Software Maintenance

In the modern world, software has become deeply embedded in the infrastructure that sustains human life. From communication networks and transportation systems to medical devices, finance, and education, software forms the invisible backbone that enables modern civilization to function. Virtually every industry, organization, and social service now depends on complex software systems for its daily operation. As a result, the reliability and long-term sustainability of these systems have become a matter of social importance. A single failure in a large-scale software system can lead not only to economic loss but also to disruptions of essential services, safety hazards, and the erosion of public trust in technology.

Software is inherently different from traditional physical artifacts. Unlike a machine or a bridge, it does not deteriorate through physical wear. Instead, software *ages* because the world around it changes [51]. Programming languages evolve, libraries are updated, hardware platforms are replaced, and user requirements shift in response to technological and societal trends. This continual change creates an ongoing need to adapt existing software to new contexts. As Lehman's seminal laws of software evolution indicate, any successful software system that is actively used must be continuously modified, or it will become less useful and eventually obsolete [41, 42]. Therefore, the maintenance and evolution of software are not peripheral activities; they are at the very core of the software lifecycle.

### The Expanding Scope of Maintenance

Historically, software maintenance was often viewed narrowly as the task of fixing bugs after delivery. However, decades of empirical research have shown that maintenance encompasses a far broader range of activities. Ac-

cording to the IEEE Standard for Software Maintenance [1], maintenance refers to *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.* In other words, maintenance is not merely reactive; it is a proactive and strategic process that ensures the long-term viability of software.

Scholars and practitioners generally classify software maintenance into four main categories [2]: **corrective**, **adaptive**, **perfective**, and **preventive** maintenance.

- **Corrective maintenance** involves fixing defects discovered in the system after it has been released.
- **Adaptive maintenance** refers to the modification of software to keep it usable within a changing environment, such as adapting to new operating systems, databases, or programming language versions.
- **Perfective maintenance** focuses on improving performance, maintainability, usability, or other qualities to better satisfy user needs.
- **Preventive maintenance** aims to detect and correct potential issues before they cause system failures, such as refactoring code to prevent future bugs or vulnerabilities.

These four categories are complementary, and their relative prominence depends on a system's context, lifecycle stage, and operating environment. In contemporary ecosystems—where programming languages, libraries, frameworks, and platforms evolve continuously—environmental change is a particularly common driver of maintenance work, making **adaptive maintenance** unavoidable for many long-lived systems.

This dissertation focuses on ecosystem-driven **adaptive maintenance** in the presence of widespread code reuse. In particular, when developers reuse external code (e.g., online snippets), they often need foresight to anticipate version-related incompatibilities prior to integration. This need underscores the importance of understanding software, its dependencies, and how it evolves over time—an understanding that we operationalize through *analyses* of compatibility, functional equivalence, and structural similarity.[1]

## Software Evolution as a Continuous Process

The concept of *software evolution* extends maintenance from a set of activities into a broader theoretical framework. While maintenance describes

[1] In practice, migrations may co-occur with defect correction or restructuring activities. In this dissertation, we treat such co-occurrence as context and focus our contributions on adaptive-maintenance risks under ecosystem evolution.

what developers do to keep software operational, evolution emphasizes the dynamic nature of software itself. Every modification, whether a bug fix, an update, or a refactoring, changes the software's structure, often in ways that accumulate over the years. The architecture that once seemed clean and modular can become tangled through numerous small adaptations. Dependencies among components multiply, and undocumented assumptions become embedded in the code.

Software evolution is thus not a discrete event but a continuous process of adaptation and reorganization. It reflects the co-evolution between software and its environment: as technology and user expectations evolve, so must the software. Managing this process effectively requires balancing two conflicting goals, preserving system stability while enabling change. The more software evolves, the harder it becomes to understand, yet understanding is precisely what is needed to evolve it safely. This paradox makes software evolution one of the most enduring challenges in software engineering.

The practical implications are profound. Studies have shown that maintenance and evolution activities typically account for more than 60% of the total cost of software over its lifetime, and in large organizations, this proportion can exceed 80% [17, 43]. As software systems grow larger, distributed, and interconnected, the difficulty of ensuring consistent behavior across versions and environments increases dramatically. Consequently, research into tools, techniques, and methodologies that facilitate maintenance and evolution is central to both academic inquiry and industrial practice.

### Understanding Code as the Foundation of Maintenance

At the heart of maintenance lies a deceptively simple question: how well do developers understand the code they must modify? Maintenance tasks are often performed not by the original authors but by later teams who inherit complex codebases with limited documentation. Even when source code is available, understanding its intent, assumptions, and dependencies can be daunting. Maintenance is therefore a cognitive activity as much as a technical one; it requires reconstructing the mental model of a system through its code.

This challenge is compounded by modern development practices such as reuse, modularization, and open collaboration. Developers today frequently incorporate external components, libraries, and snippets drawn from vast online ecosystems. While this reuse accelerates development and reduces duplication, it also introduces hidden dependencies and potential incompatibilities. Understanding how a piece of external code behaves, whether it is safe to integrate, and whether it will remain compatible with future updates has become a crucial aspect of maintenance work.

Thus, *code comprehension* is not merely a preliminary step but the very foundation upon which effective maintenance depends. Without understanding, developers risk introducing subtle faults, misusing APIs, or duplicating existing functionality. These issues accumulate over time, eroding software quality and increasing maintenance costs, a phenomenon often referred to as *software entropy* [16, 51].

### The Changing Landscape of Maintenance in the Era of Reuse

The widespread availability of open-source repositories and online Q&A platforms has profoundly changed how software is developed and maintained [9, 87, 88]. Platforms like GitHub and Stack Overflow have become invaluable sources of reusable knowledge. Developers can instantly access millions of code examples, libraries, and solutions contributed by others. This collective knowledge accelerates innovation and lowers barriers to entry.

However, the convenience of reuse comes with new risks [61, 87, 88]. Code snippets shared online are often context-specific: they may rely on particular versions of libraries, frameworks, or programming languages. A snippet that works flawlessly in one environment may fail in another. Furthermore, such code fragments are seldom accompanied by metadata describing their dependencies or compatibility. Developers copying them into their projects must manually adapt them, often without realizing that the snippet was written for a different version or platform. These subtle mismatches can lead to defects that are difficult to detect through traditional testing.

From a maintenance perspective, this phenomenon blurs the boundary between *development* and *reuse*. Software is no longer built entirely from scratch; it is assembled from a mosaic of components, some of which are poorly understood. The traditional view of maintenance as post-delivery modification must therefore be expanded to include the continuous process of managing and integrating reused code throughout the software lifecycle. This shift calls for new analytical frameworks and tools capable of dealing with the diversity and dynamism of modern code ecosystems.

### The Need for Systematic Analysis of Software Change

Given these realities, understanding software maintenance today requires more than tracking version histories or counting bug fixes. It demands a systematic analysis of how code changes, interacts, and evolves. Researchers have long studied *code clones* [7, 30, 62], repeated or similar code fragments, as indicators of redundancy and potential maintenance problems. Others have investigated *refactoring* [23] as a disciplined way to improve code structure without altering behavior. More recently, attention

has turned to the semantic dimension of code [4, 25], understanding what a piece of code does, not just how it looks.

This shift toward semantic understanding aligns closely with the goals of sustainable maintenance. To maintain software effectively, developers must identify when two code fragments are equivalent in function, when a reused component is compatible with its environment, and when structural similarity implies potential reuse or refactoring opportunities. Each of these perspectives, compatibility, equivalence, and similarity, offers a different lens for understanding software change. Together, they form a conceptual triad that captures the multifaceted nature of software evolution.

Therefore, advancing maintenance research requires bridging these perspectives. Compatibility concerns whether code remains *usable* under a target environment (e.g., a specific language/runtime version and its APIs), serving as a prerequisite for safe reuse and migration. Equivalence concerns whether alternative implementations exhibit indistinguishable behavior under the same inputs (as approximated by test-based evidence in this dissertation). Similarity provides a means to detect patterns, redundancies, and relationships that inform refactoring and reuse. By analyzing these aspects together, we can move closer to the long-standing goal of *sustainable software evolution* (Definition 1.1).

**Definition (Sustainable software evolution).** In this dissertation, *sustainable software evolution* refers to a state in which software can accommodate continual *ecosystem change*—such as programming-language and dependency upgrades, as well as reuse-driven integration—without compromising acceptable functional correctness or maintainability. This notion is operationalized in this dissertation through joint analyses of *compatibility*, *functional equivalence*, and *structural similarity*, which support safer migration, reuse, and refactoring decisions.

## 1.2 Challenges in Modern Software Evolution

As software has grown to unprecedented scales and become integral to all aspects of society, the task of maintaining and evolving it has become increasingly complex. Modern software systems are not isolated programs but ecosystems composed of interdependent modules, libraries, services, and external APIs. They operate in heterogeneous environments that constantly evolve, driven by technological advances and user demands. Managing such systems poses formidable challenges for developers, researchers, and organizations alike. This section discusses the major challenges that characterize contemporary software evolution and the motivations behind the analytical perspectives addressed in this dissertation.

### Growing Complexity and Interdependence

One of the most fundamental challenges in software evolution is the explosion of system complexity [48]. Software today is rarely developed from scratch; instead, it is composed of thousands of interacting components written in multiple languages, maintained by distributed teams, and deployed across diverse platforms [38]. Each layer, from the operating system and runtime environment to application frameworks and user interfaces, evolves at its own pace. A seemingly minor update in one layer may trigger unforeseen side effects in another, creating a ripple effect of maintenance tasks.

The resulting web of dependencies makes reasoning about software behavior increasingly difficult [38]. As systems evolve, the number of interactions among modules grows nonlinearly, and understanding the global impact of local changes becomes nearly impossible without automated analysis. Moreover, modern development practices such as continuous integration and continuous deployment (CI/CD) have shortened release cycles dramatically. While this accelerates innovation, it also reduces the time available for manual review and long-term architectural reflection, increasing the risk of unintended degradation.

This complexity is not only structural but also social [71]. Large software projects often involve contributors from around the world with varying levels of expertise and different goals. Knowledge about design decisions, dependencies, or compatibility constraints is frequently implicit, existing only in developer discussions or issue trackers. As contributors come and go, this knowledge decays, leading to the phenomenon known as *software aging*. Thus, the challenge of managing evolution is as much about sustaining human understanding as it is about maintaining technical consistency.

### The Burden of Dependencies and Technology Transitions

Software rarely exists in isolation; it depends on a constantly shifting technological foundation. Frameworks, programming languages, and third-party libraries are updated continuously, often introducing breaking changes. When a dependency evolves, downstream projects must evolve with it, creating a cascade of adaptation tasks. In some cases, developers are forced to upgrade not because they want new features but because older versions become unsupported or incompatible with modern platforms.

These dependency transitions can be costly and error-prone. A notable example is the migration from Python 2 to Python 3, one of the most widely discussed language transitions in modern software history [45, 46]. Python 3 introduced numerous syntactic and semantic changes that intentionally broke backward compatibility, thereby requiring developers to modify existing codebases to remain functional [73]. Although such tran-

sitions aim to improve the language, they also create massive maintenance overhead for projects that rely on legacy code or community-contributed snippets.

Importantly, disruptive, backward-incompatible transitions are not unique to Python. Comparable ecosystem-wide breaks have also been observed in other mainstream languages, though manifested in different forms and to different extents. Examples include the Swift 2 to Swift 3 migration [69], major upgrades in PHP (e.g., PHP 5 to PHP 7 [55] and PHP 7 to PHP 8 [56]), and Ruby's major-version shifts (e.g., the historically disruptive Ruby 1.8 to Ruby 1.9 transition [63] and the more recent Ruby 2.7 to Ruby 3.0 migration [64] involving planned backward-incompatible adjustments). These examples illustrate a general reality of modern software evolution: as programming languages evolve, developers must repeatedly assess whether legacy or reused code remains compatible with the target environment. In this dissertation, the transition from Python 2 to Python 3 serves as a representative and data-rich case that enables systematic empirical analysis and tool support.

Beyond language migrations, dependency evolution also affects open-source ecosystems [71]. Many projects depend on one another in complex hierarchies, where an update in a core library can affect thousands of downstream users. Tools such as package managers (e.g., `pip`, `npm`, `Maven`) simplify distribution but also propagate compatibility issues at unprecedented speed. Ensuring that all components in such dependency graphs remain mutually compatible has become a major concern for sustainable software evolution. These transitions underline the importance of **compatibility analysis**, a foundational theme in this dissertation.

### Code Reuse and Its Hidden Risks

Modern software engineering encourages reuse as a means to reduce duplication and improve productivity. Developers can integrate open-source libraries, frameworks, and even small code snippets shared on online Q&A platforms such as Stack Overflow [9]. This reuse culture embodies one of the core principles of software engineering: "Don't Repeat Yourself." However, while reuse accelerates development, it introduces new and subtle maintenance challenges.

Code fragments available online are typically written for specific environments and versions [87, 88]. When reused blindly, they may not behave as expected in a different context. In the case of dynamic languages such as Python, a code snippet that executes correctly under one interpreter version might raise errors or produce unexpected results under another. Moreover, the vast majority of shared snippets lack explicit metadata describing their dependencies, supported versions, or execution constraints.

The problem is further compounded by the phenomenon of *copy-and-paste reuse* [21]. Developers often copy small code examples directly into their own projects without verifying their correctness or compatibility. These copied fragments can propagate bugs, security vulnerabilities, and outdated practices across projects. Once integrated, they become part of the system's evolution history, often without clear attribution or traceability.

From a maintenance perspective, reused code must be understood, validated, and maintained just like any other part of the system. However, because its origin and assumptions are often unclear, it becomes a source of uncertainty. Detecting whether such code is compatible with the current environment or functionally equivalent to existing implementations is therefore an essential capability for modern maintenance tools.

### Challenges of Understanding Functional Equivalence

Even when reused code executes correctly, determining whether it behaves as intended or matches the behavior of existing implementations remains challenging. Large systems routinely contain multiple implementations of similar functionality, arising from independent development or incomplete refactoring efforts.

Syntactic similarity alone is insufficient for reasoning about behavior: implementations that look different may behave identically, while those that look similar may differ in subtle but important ways. Traditional clone detection techniques cannot reliably capture such distinctions. Although dynamic and semantic analysis techniques exist [24, 26], they often face scalability limitations.

A deeper examination of **functional equivalence** and how it differs from syntactic similarity is provided in Section 1.3, where the concept is formally defined and contextualized within this dissertation.

### Structural Similarity and the Complexity of Change

Software evolution also involves understanding how code changes structurally. Patterns of structural similarity across versions can indicate refactoring opportunities, divergence caused by independent evolution, or copy-and-paste reuse that increases maintenance cost.

However, structural similarity can be obscured by identifier renaming, restructuring, or refactoring. A variety of techniques, ranging from token- or AST-based comparison to graph representations and learned embeddings [5, 11, 20, 36, 40], attempt to capture these relationships, each with trade-offs between precision and scalability.

Section 1.3 provides more precise definitions of structural similarity and explains its relationship to compatibility and equivalence. Here, the focus

is on the challenge: understanding structure is essential for tracking how code evolves, yet difficult due to continuous and diverse transformations.

### Interdependence of Compatibility, Behavior, and Structure

These challenges are deeply interconnected. A change intended to restore compatibility may break existing behavior; conversely, a behavior-preserving refactoring may significantly alter structure. Copy-and-paste reuse may yield structural similarity without behavioral correctness. Understanding these interactions requires tools and methodologies that consider compatibility, functional behavior, and structural patterns jointly.

Section 1.3 introduces these three dimensions formally and explains how they form the analytical foundation of this dissertation.

### Summary

In summary, modern software evolution is characterized by rapid change, pervasive reuse, and intertwined dependencies. These factors make maintenance more challenging than ever. Issues of compatibility, functional behavior, and structural similarity are central to understanding and managing this complexity. Section 1.3 formalizes these concepts, which serve as key analytical lenses for the studies presented in this dissertation.

## 1.3 Key Concepts: Compatibility, Equivalence, and Similarity

The previous sections outlined the growing complexity of modern software systems and the diverse challenges that arise during their maintenance and evolution. Among these challenges, three interrelated concepts—**compatibility**, **equivalence**, and **similarity**—play a particularly crucial role. Each represents a different lens through which software change can be analyzed: compatibility captures how software interacts with its environment, equivalence concerns the preservation of functionality across versions or implementations, and similarity addresses the structural relationships among code fragments. This section defines these concepts, discusses their distinctions and overlaps, and positions them as the theoretical foundation of this dissertation.

### Software Compatibility

**Compatibility** broadly refers to the ability of software to operate in a target environment or alongside other software components [48]. It reflects the alignment between code and its surrounding technologies, including programming-language versions, libraries, operating systems, and hardware

platforms. When compatibility is broken, code that previously worked may fail to run, or it may run but behave unexpectedly.

In software engineering, compatibility is often discussed in several forms:

- **Backward compatibility** means that a newer version of a system or language can correctly process (and ideally execute) code written for older versions.
- **Forward compatibility** refers to the ability of older software to tolerate input or interaction from newer versions, typically through flexible or extensible design.
- **Cross-version compatibility** concerns interoperability between different versions of the same software or between distinct yet related components.
- **Platform compatibility** denotes the ability of software to run across different operating systems, architectures, or hardware environments.

**Compatibility levels and scope in this dissertation.** Orthogonally to the above forms, compatibility can also be assessed at different layers. *Syntax-level* incompatibilities prevent code from being parsed or compiled; *API/library-level* incompatibilities arise when dependencies change or are removed; and *behavioral* incompatibilities occur when code executes but its runtime behavior diverges (e.g., different outputs, raised exceptions, or other runtime failures).

**Operationalization in this dissertation.** To enable large-scale measurement and lightweight tool support, Studies 1–2 (Chapters 2–3) operationalize compatibility at the *syntax level*: a Python snippet is treated as *syntactically compatible* with a given Python version if it can be successfully parsed/compiled by that version (as implemented in the corresponding study). This criterion captures a clear, necessary precondition for execution and provides a direct signal of version-related breakage. Accordingly, if a snippet parses successfully but later exhibits runtime errors or behavioral divergence, it is considered *syntactically compatible* but potentially *behaviorally incompatible*. Behavioral compatibility is outside the primary scope of Chapters 2–3 and is discussed as a limitation and future direction.

Maintaining compatibility is essential for sustainable software evolution. Without it, systems risk fragmentation, as users and developers must maintain multiple versions of the same functionality for different environments. The issue is particularly visible in programming languages that evolve rapidly. For instance, the transition from Python 2 to Python 3 represents a deliberate break in backward compatibility to improve language design [73]. While this decision modernized Python, it also caused

substantial maintenance overhead across the ecosystem: countless libraries and Stack Overflow code snippets became unusable in newer environments, leaving developers uncertain about which examples would work for their target version.

From a maintenance perspective, compatibility analysis involves detecting and understanding these incompatibilities. It requires tools capable of identifying which language features, syntax forms, or APIs are supported by different versions, and how those differences impact code execution. The ability to automatically assess compatibility thus plays a critical role in ensuring that reused or legacy code can be safely incorporated into evolving systems.

### Functional Equivalence

**Functional equivalence** concerns whether two implementations are indistinguishable with respect to a chosen *observation model* (i.e., what we regard as observable behavior) [82]. In its strongest form, equivalence can be defined (i) *with respect to a formal specification*, or (ii) as *observational/behavioral equivalence* over the entire input domain. In real-world maintenance settings, however, explicit formal specifications are often unavailable, and input-universal proofs are typically infeasible.

Accordingly, equivalence is commonly discussed along two orthogonal dimensions: (1) *what* is observed (the observation model), and (2) *over which inputs* the equivalence is required (universal vs. a finite test suite).

- **Specification-level equivalence:** two implementations are equivalent if they satisfy the same formal specification/contract, when such a specification is available.

- **Observational (behavioral) equivalence, universal:** two implementations are equivalent if, for all inputs in the domain, they exhibit the same observable behavior under the chosen observation model.

- **Observational equivalence w.r.t. a test suite (test-suite / empirical equivalence):** two implementations are treated as equivalent if they exhibit the same observable behavior for a finite set of test inputs, providing an empirical approximation widely adopted in testing and empirical software engineering studies.

Depending on the observation model, equivalence may range from pure input–output behavior (e.g., return values and raised exceptions) to operational effects such as state transitions and side effects (e.g., I/O actions).

**Operationalization in this dissertation.** Study 3 (Chapter 4) adopts a *test-based, observational* notion of functional equivalence at the method level, primarily under an input–output observation model. Concretely, given an input, we treat the observable outcome of a method execution as its normal return value or an uncaught exception; abnormal termination (e.g., timeouts/crashes), when applicable in the experimental setup, is treated as a distinct observable outcome.

We operationalize FE labels through automated test generation and mutual execution: given two methods $A$ and $B$ with their respective generated test suites, we cross-run $A$ on $B$'s tests and $B$ on $A$'s tests; if both exhibit the same observable outcomes on all executed tests, the pair is treated as a *candidate* FE pair under the given test suites. Because test suites are inherently finite, this provides empirical evidence rather than a proof for all possible inputs; therefore, we further conduct manual validation and, when necessary, add new tests to expose functional differences. Only pairs that are ultimately confirmed are included as FE in the constructed dataset. Consequently, the equivalence labels used in this dissertation correspond to *observational equivalence w.r.t. a test suite* under a controlled execution environment, rather than formal specification-level equivalence or universal behavioral equivalence.

Determining equivalence is important for a variety of maintenance tasks. When developers refactor code, optimize algorithms, or replace components, they must ensure that the changes do not alter the intended behavior. Equivalence checking thus forms the theoretical basis of *regression testing*, *compiler optimization*, and *program verification*.

In practical software systems, however, exact equivalence is difficult to prove. The space of possible program inputs is infinite, and even slight variations in implementation can produce subtle differences. As a result, researchers often approximate equivalence using empirical methods such as dynamic testing, differential analysis, or symbolic execution. In recent years, the availability of large code repositories has enabled data-driven approaches: instead of proving equivalence formally, tools can learn from examples of functionally equivalent methods across projects.

Functional equivalence also provides insight into the redundancy and diversity of software ecosystems. On platforms like GitHub, it is common to find multiple implementations of the same algorithm, written independently by different developers. Studying these equivalent pairs reveals not only alternative design patterns but also performance trade-offs and stylistic variations. Moreover, equivalence detection enables *semantic clone analysis*, identifying code fragments that perform the same task but differ syntactically. This concept, known as *Type-4 clones*, extends beyond the scope of traditional clone detection, which primarily targets syntactic similarity (Type-1 to Type-3) [62]. Understanding functional equivalence is thus a

prerequisite for higher-level reasoning about software reuse, redundancy, and optimization.

### Structural Similarity

While equivalence focuses on behavior, **similarity** concerns structure. Structural similarity refers to the degree to which two code fragments share common syntactic, lexical, or architectural patterns [11]. It is the foundation of many program analysis tasks, including code clone detection, plagiarism analysis, and refactoring support.

Similarity can be measured at different granularities and representations [40]:

- **Lexical similarity:** Based on token or character sequences, capturing superficial resemblances in code text.
- **Syntactic similarity:** Based on Abstract Syntax Trees (ASTs) or parse structures, reflecting shared syntactic constructs.
- **Semantic or structural similarity:** Based on control-flow or data-flow graphs, focusing on relationships between program elements rather than their surface form.

Detecting similarity is valuable for several reasons. It allows developers to locate duplicated logic, identify opportunities for code reuse, and detect inconsistent modifications across versions. At a higher level, structural similarity supports the study of software evolution: by comparing code fragments across commits, researchers can track how structures diverge or converge over time.

However, similarity alone does not guarantee equivalence. Two fragments may look nearly identical but differ in subtle semantic ways, for instance, in boundary conditions or exception handling. Conversely, functionally equivalent fragments may look entirely different due to refactoring, optimization, or coding style. This disconnect between syntactic and semantic similarity lies at the heart of many software analysis challenges. Bridging it requires integrating multiple perspectives, combining structural comparison with semantic understanding.

### Relationships and Interactions among the Three Concepts

Although compatibility, equivalence, and similarity originate from different contexts, they are deeply intertwined in practice. Compatibility concerns the relationship between code and its environment; equivalence, the relationship between multiple implementations; and similarity, the relationship

between structures within or across systems. Together, they define the space in which software evolves.

Conceptually, these relationships can be understood as follows:

- Compatibility constrains the boundary conditions for correct execution.
- Equivalence ensures behavioral preservation under change.
- Similarity provides structural signals that enable reasoning across versions.

Changes in one dimension often affect the others. For example, a refactoring operation that preserves functional equivalence may alter structural similarity. Upgrading a dependency to ensure compatibility may break equivalence with previous behavior. Conversely, detecting structural similarity across incompatible versions may indicate opportunities for adaptation. Understanding these interactions allows researchers to develop comprehensive analyses that capture the multidimensional nature of software change.

In the context of this dissertation, these three concepts serve as the analytical pillars for studying sustainable software evolution. Study 1 (Chapter 2) focuses on compatibility by empirically investigating how Python version differences affect code reuse on Stack Overflow, primarily through syntax-level interpretability analysis. Building on this empirical evidence, Study 2 (Chapter 3) develops and evaluates PyVerDetector, a browser-based tool that detects and visualizes Python version compatibility for online code snippets in real time. Finally, Study 3 (Chapter 4) extends the investigation beyond environment-dependent compatibility to behavior- and structure-aware analysis by constructing and analyzing a dataset of functionally equivalent method pairs, examining how functional equivalence relates to structural similarity and other properties, and evaluating the capability of large language models to recognize equivalence/similarity relationships.

### Toward a Unified Perspective

Treating compatibility, equivalence, and similarity as independent topics has yielded valuable insights, but a fragmented understanding limits progress. Each captures only one facet of software evolution. A unified perspective, however, can reveal patterns that remain invisible when studied in isolation.

For example, compatibility analysis alone can detect whether a snippet will execute under a given environment, but it cannot determine whether alternative snippets would produce the same results. Equivalence analysis can verify functional identity, but without considering structural similarity, it may miss recurring patterns that indicate broader design principles.

Structural similarity, in turn, identifies patterns but cannot judge correctness or environmental fit. Integrating these perspectives allows for more holistic reasoning about software change: we can ask not only "does this code run?" but also "does it behave the same?" and "how is it related to other code in the system?"

This integrated view forms the conceptual foundation of this dissertation. By connecting compatibility, equivalence, and similarity, we move toward a comprehensive understanding of how software can evolve safely and intelligently. The following sections build upon these definitions to articulate the research motivation, objectives, and methodologies adopted in this study.

## 1.4 Research Motivation and Questions

The preceding sections established that software maintenance and evolution are indispensable yet challenging activities in the software lifecycle. Modern systems evolve under constant technological, organizational, and social change. Developers must continuously adapt existing code to new versions of languages, frameworks, and environments while ensuring that system functionality and quality remain intact. As discussed through the lenses of compatibility, equivalence, and similarity (Section 1.3), maintaining such continuity across versions and implementations is far from trivial.

This dissertation is motivated by the need to better understand and support **sustainable software evolution**, i.e., enabling software to accommodate continual ecosystem change without compromising acceptable correctness or maintainability (Definition 1.1). Achieving this goal requires a systematic understanding of (i) how code interacts with evolving environments, (ii) how alternative implementations coincide or diverge in behavior, and (iii) how structural patterns relate to maintainability and reuse. Despite decades of progress, existing research and tools remain fragmented across these dimensions. Bridging them is the central motivation of this dissertation.

**Structure and scope.** To ensure consistency between the dissertation-level questions and the remainder of the manuscript, the dissertation is organized into **three main studies** (Chapters 2–4), each centered on one dissertation-level research question. The three questions below are the only *dissertation-level* RQs; later chapters may introduce *study-specific* sub-questions for measurement or evaluation, but they will be labeled with distinct identifiers to avoid confusion (see the naming rule at the end of this section).

### Motivation 1: Understanding Compatibility in Code Reuse

As the software engineering community increasingly relies on code reuse, compatibility has emerged as one of the most pervasive yet underexplored challenges in real-world development. Online platforms such as Stack Overflow host millions of code snippets that developers frequently copy, adapt, and integrate into their projects. These snippets serve as both a valuable knowledge resource and a potential source of maintenance problems.

Programming languages evolve through version updates that introduce changes to syntax, semantics, and library APIs. While such changes may improve expressiveness and safety, they can also break backward compatibility. In Python, code written for one version may no longer be interpretable by another. Moreover, developers who consult online snippets rarely know which version a snippet targets, and most examples lack explicit version annotations.

This situation raises an important empirical question:

> **D-RQ1:** *How prevalent are version compatibility issues in reused code snippets on online platforms such as Stack Overflow, and how do they affect developers' ability to reuse code safely?*

In line with the compatibility scope clarified in Section 1.3, Studies 1–2 primarily operationalize version compatibility at the *syntax level* (interpretability), which is a necessary precondition for execution and a direct indicator of version-related breakage in reused snippets. Addressing D-RQ1 requires large-scale empirical evidence rather than anecdotal reports. By systematically analyzing Python snippets from Stack Overflow and assessing their version interpretability, we quantify the scope of the problem and provide insights into how language evolution impacts community knowledge sharing. These findings constitute the focus of **Study 1**.

### Motivation 2: Automating Compatibility Detection and Support

Empirical evidence alone cannot solve the compatibility problem; developers need actionable support. Manual verification of compatibility is infeasible given the volume of online code. Therefore, the second motivation of this dissertation is to design automated methods that help developers detect and understand version compatibility risks at the moment of code reuse.

Existing static analyzers and linters can detect some issues, but they are not designed for real-time analysis of arbitrary online code fragments, and most operate inside development environments rather than directly within the platforms where reuse occurs. Developers encountering a snippet on

Stack Overflow should ideally be able to assess its version compatibility *before* copying it into their projects.

To address this gap, **Study 2** introduces **PyVerDetector**, a browser-based tool that detects Python version compatibility of Stack Overflow snippets in real time. The tool parses code fragments using multiple version-specific grammars and visualizes which Python versions can interpret each snippet, thereby providing immediate feedback in the reuse context.

Accordingly, the second research question is formulated as follows:

> **D-RQ2:** *How can automated tools effectively detect and visualize version compatibility issues in online code snippets, and to what extent can such tools assist developers in avoiding version-related faults?*
>
> **Interpretation of D-RQ2.** We answer D-RQ2 along two complementary aspects:
>
> - **Detection/visualization feasibility and correctness:** What design enables accurate, real-time detection and clear visualization of *syntax-level* version-compatibility signals (interpretability) for online snippets?
> - **Usefulness in reuse settings:** To what extent do such in-context signals provide actionable support that helps developers avoid version-related reuse faults?

**Why evaluating a single tool answers a general question.** We address D-RQ2 by proposing **PyVerDetector** as a concrete, browser-integrated instantiation of automated compatibility tooling and evaluating it as an *existence proof* of feasibility and utility in the target setting (online snippet reuse). The goal is not to exhaustively evaluate all possible tools, but to demonstrate a realizable design and empirically assess its effectiveness and practical implications.

### Motivation 3: Exploring Equivalence and Similarity in Software Evolution

While compatibility concerns whether code can operate within a given environment, it does not guarantee that alternative implementations behave identically or evolve consistently. In large software ecosystems, multiple implementations of similar functionality often arise through independent development, partial refactoring, or reuse. Identifying such relationships is crucial for understanding redundancy, ensuring consistency, and guiding maintenance.

Traditional clone detection captures syntactic similarity but often misses deeper semantic relationships. In dynamic languages such as Python,

structural variations may conceal functionally identical behavior. Detecting *functionally equivalent* and *semantically/structurally related* code fragments can reveal refactoring opportunities, performance trade-offs, and knowledge transfer patterns. Recent advances in large language models (LLMs) further motivate systematic investigation of such relationships.

These observations lead to the third research question:

> **D-RQ3:** *How can functionally equivalent and semantically similar code fragments be identified, analyzed, and leveraged to understand redundancy and diversity in large software repositories?*

To investigate D-RQ3, **Study 3** constructs and analyzes a dataset of Python methods extracted from open-source projects. Through automated test generation, mutual execution, and manual validation, candidate functionally equivalent pairs are identified and studied. This dataset enables analysis of behavioral equivalence, structural variation, and performance differences, and supports evaluation of LLMs for equivalence/similarity recognition.

### Integrating the Three Perspectives

The three studies form a coherent progression from observation to automation to broader semantic reasoning. **Study 1** establishes empirical evidence of real-world version compatibility problems in reused snippets (D-RQ1). **Study 2** translates these insights into a practical automated tool and evaluates its feasibility and utility (D-RQ2). **Study 3** expands the focus from environment-dependent compatibility to intrinsic behavioral and structural relationships in code (D-RQ3). Together, they support a unified understanding of sustainable software evolution through the joint lenses of compatibility, equivalence, and similarity.

### Naming rule for research questions

To avoid confusion between the dissertation-level research questions and the Study 1 sub-questions, we use the following naming convention throughout the remainder of this dissertation:

- **D-RQ1**–**D-RQ3** denote the three dissertation-level research questions defined in this section.
- **S1-RQ1**–**S1-RQ3** denote the study-specific sub-questions used **only in Study 1** to operationalize **D-RQ1** for measurement and analysis.
- **Study 2 and Study 3** address **D-RQ2** and **D-RQ3** directly and therefore **do not define additional study-specific research questions** (i.e., no S2-RQx or S3-RQx)

### Summary

In summary, this dissertation is driven by three overarching motivations: (1) to understand the extent and impact of version compatibility issues in real-world code reuse (D-RQ1); (2) to design and evaluate automated tooling for real-time compatibility detection and support (D-RQ2); and (3) to investigate functional equivalence and similarity to understand redundancy and diversity in large repositories (D-RQ3). The next section describes the methodological approach used to address these questions and outlines how each study contributes empirical evidence, tooling, and datasets toward sustainable software evolution.

## 1.5 Research Approach and Methodology

To address the dissertation-level research questions (D-RQ1–D-RQ3) defined in Section 1.4, this dissertation adopts a mixed empirical and tool-based methodology. The approach integrates large-scale data analysis, automated program analysis, and artifact construction (tools and datasets) to investigate software evolution from complementary perspectives. The research design follows three studies/phases that align one-to-one with the three dissertation-level RQs, ensuring structural consistency across the manuscript.

### Overview of the Research Design

The research follows a three-study structure. It begins with an empirical analysis of real-world reuse data to quantify the scale and characteristics of version compatibility problems (Study 1 / D-RQ1). These insights motivate the design of a browser-based tool for real-time compatibility detection and visualization, which is then evaluated for correctness and practical implications (Study 2 / D-RQ2). Building on the compatibility perspective, the research further investigates software evolution through functional equivalence and structural similarity by constructing and analyzing a dataset of method pairs from open-source repositories (Study 3 / D-RQ3).

This staged design ensures both depth and continuity. Each study addresses its own RQ while informing the next: empirical evidence motivates tool requirements, and compatibility analysis experience informs later equivalence validation and dataset construction. Through this progression, the dissertation contributes both empirical understanding and reusable artifacts that support sustainable software evolution.

### Study 1: Empirical Study of Code Compatibility in Reuse (D-RQ1)

Study 1 investigates the prevalence and impact of version compatibility issues in reused Python snippets.

**Data Collection.** Python code snippets are collected from Stack Overflow using *SOTorrent* [10]. The dataset focuses on *top-scored (i.e., highest-voted)* answers to questions tagged with "Python" over multiple years. Each snippet is retrieved together with contextual metadata (e.g., tags, timestamps, and answer scores).

**Analysis Method.** Following the compatibility scope defined in Section 1.3, we primarily assess *syntax-level version compatibility* (interpretability). Each snippet is analyzed against multiple Python versions to determine whether it can be parsed/compiled by a given version. Based on these results, snippets are categorized by their compatibility profiles (e.g., compatible across all considered versions vs. only specific versions). Additional analysis examines correlations between snippet metadata (e.g., tags or time) and observed compatibility.

**Outcome.** Study 1 provides large-scale empirical evidence on the extent of version-related breakage in widely reused code. It quantifies how often online snippets exhibit version incompatibilities and identifies common incompatibility patterns. These findings establish the motivation and requirements for automated support in Study 2.

### Study 2: Tool Development and Evaluation for Automated Compatibility Support (D-RQ2)

Study 2 addresses D-RQ2 by designing and implementing **PyVerDetector**, a browser-based tool that detects and visualizes Python version compatibility of Stack Overflow snippets in real time.

**Tool Design: PyVerDetector.** PyVerDetector is implemented as a Chrome extension integrated into the browsing environment. Its design consists of:

- a **frontend module** that extracts code blocks from Stack Overflow pages and displays version-related feedback in context; and
- a **backend module** that analyzes each snippet using version-specific grammars (and supporting infrastructure compiled to WebAssembly), enabling on-page compatibility assessment across Python versions.

**Functionality.** Upon page load, PyVerDetector detects Python code blocks, determines their compatibility across supported versions, and annotates the page with visual indicators. When a snippet is incompatible with a selected version, the tool reports the corresponding parsing/compilation failure information to help users understand the incompatibility source.

**Evaluation aligned with the two aspects of D-RQ2.** To align evaluation with the interpretation of D-RQ2 (Section 1.4), we structure the assessment as follows:

- **Detection/visualization feasibility and correctness:** We evaluate the accuracy of PyVerDetector's version-compatibility decisions by comparing its parsing outcomes with interpreter-based ground truth on a large sample of Stack Overflow snippets (and, where applicable, against an existing tool such as PyComply). Feasibility is demonstrated by an end-to-end browser-integrated implementation that performs version-specific parsing and in-context visualization on Stack Overflow pages.

- **Usefulness in reuse settings:** Rather than claiming a full-scale user study, we complement the quantitative accuracy evaluation with representative usage scenarios and a discussion of practical implications. These scenarios illustrate how in-context compatibility signals can help developers anticipate version-related reuse risks *before* integration.

**Outcome.** Study 2 delivers a publicly available, browser-integrated tool that operationalizes automated compatibility support in the reuse context. The evaluation provides evidence that real-time compatibility detection and visualization are feasible and that the resulting signals are practically meaningful for avoiding version-related reuse faults.

### Study 3: Dataset Construction and Analysis of Code Equivalence and Similarity (D-RQ3)

Study 3 addresses D-RQ3 by constructing and analyzing a dataset of Python methods to study functional equivalence and structural similarity at scale.

**Data Source and Extraction.** We use *ManyTypes4Py* [49], a benchmark dataset of type-annotated Python repositories collected from GitHub. Methods are extracted, normalized, and grouped by compatible signatures (inferred via type information and/or type inference models where applicable).

**Equivalence Identification via Testing.** Candidate functionally equivalent pairs are identified through automated test generation (e.g., using *Pynguin* [44]) and mutual execution. Under the observation model defined in Section 1.3, equivalence is operationalized as *observational equivalence w.r.t. a test suite*: two methods are treated as equivalent if they exhibit the same observable outcomes on generated tests, followed by manual validation when necessary.

**Dataset Construction and Analysis.** Validated pairs are compiled into a dataset (e.g., **PyFuncEquivDataset**) consisting of equivalent and non-equivalent method pairs, along with relevant metadata (code, tests, and labels). The dataset is analyzed to understand implementation diversity, structural variation, and performance/readability trade-offs among equivalent methods. We further evaluate LLMs on their ability to recognize equivalence and similarity, highlighting both capabilities and limitations.

### Integrated Framework and Methodological Rationale

Across the three studies, the methodology follows a consistent philosophy: grounding in empirical evidence (Study 1), translating insights into actionable automation (Study 2), and enabling generalization and reproducibility through dataset construction (Study 3). The studies are mutually reinforcing: incompatibility patterns found in Study 1 inform tool design in Study 2, while the testing-based validation mindset in Study 2 informs equivalence validation in Study 3.

### Summary

In summary, this dissertation employs a three-study research design aligned with D-RQ1–D-RQ3:

- Study 1 provides large-scale empirical evidence of version compatibility issues in reused Python snippets (D-RQ1).
- Study 2 delivers and evaluates PyVerDetector as a concrete instantiation of automated compatibility support in the reuse context (D-RQ2).
- Study 3 constructs and analyzes a dataset for functional equivalence and similarity research and evaluates LLMs on these tasks (D-RQ3).

Together, these methodological components form a coherent framework for understanding and supporting sustainable software evolution. The next section summarizes the key contributions of this dissertation and outlines its overall structure.

## 1.6 Contributions and Significance

This dissertation contributes empirical evidence, practical tooling, and reusable datasets to the study of software maintenance and evolution. Grounded in the three dissertation-level research questions (D-RQ1–D-RQ3) and the corresponding three studies (Study 1–Study 3) introduced in Sections 1.4–1.5, the dissertation advances a unified perspective on sustainable software evolution (Definition 1.1) through the joint lenses of compatibility, functional equivalence, and structural similarity.

### Academic Contributions

**1. Empirical characterization of version compatibility risks in online code reuse (Study 1 / D-RQ1).** This dissertation provides a large-scale empirical analysis of Python code snippets reused from Stack Overflow to quantify how often version incompatibilities arise and how they affect reuse reliability.

- It measures version compatibility primarily at the syntax level (interpretability), a necessary precondition for execution and a direct indicator of version-related breakage in reused snippets.
- It characterizes compatibility profiles across Python versions and analyzes how snippet metadata (e.g., time and tags) relates to observed compatibility.
- It provides data and methodology that can support further research on language evolution and the reliability of community-shared code.

**2. A browser-integrated tool as an existence proof of automated compatibility support (Study 2 / D-RQ2).** This dissertation proposes **PyVerDetector**, a browser-based tool that detects and visualizes Python version compatibility of Stack Overflow snippets in real time, and evaluates it as a concrete instantiation of automated compatibility tooling in the reuse setting.

- **Detection/visualization feasibility and correctness:** the dissertation presents a realizable browser-integrated design for accurate detection and clear visualization of *syntax-level* version-compatibility signals for online snippets.
- **Usefulness in reuse settings:** the dissertation complements quantitative accuracy evaluation with representative usage scenarios and a discussion of practical implications, illustrating how in-context signals can help developers anticipate version-related reuse risks prior to integration.

**3. Construction of a test-based functional-equivalence dataset for dynamic-language research (Study 3 / D-RQ3).** This dissertation constructs a curated dataset of Python method pairs to support research on functional equivalence and similarity in dynamic languages.

- The dataset is built via automated test generation, mutual execution, and manual validation, under a test-suite-based observational notion of functional equivalence (Section 1.3).
- It enables empirical analysis of implementation diversity and structural variation among functionally equivalent methods, and supports benchmarking of program-analysis and learning-based approaches (including LLMs).
- It contributes reproducible artifacts (code, tests, and labels) that facilitate future work on semantic code understanding.

**4. An integrated perspective linking compatibility, equivalence, and similarity for sustainable software evolution.** Synthesizing the three studies, this dissertation articulates a unified view of software evolution as a multidimensional process:

- **Compatibility** captures environment-dependent constraints and version-related breakage risks in reuse.
- **Equivalence** captures behavior-preservation and redundancy across alternative implementations.
- **Similarity** captures structural relationships that inform reuse, refactoring, and maintainability.

Together, these dimensions support systematic reasoning about sustainable software evolution under continual ecosystem change.

## Practical Contributions

**1. Open artifacts for reuse and replication.** The dissertation releases the developed tool, scripts, and datasets to facilitate replication and future research.

**2. Developer-centered automation in the reuse context.** By embedding compatibility checking directly into the browsing environment, PyVerDetector demonstrates how empirical insights can be translated into lightweight, workflow-aligned automation for reducing version-related reuse risks.

**3. Reusable methodological pipeline.** Beyond the artifacts themselves, the end-to-end methodology—data extraction, version-aware parsing/analysis, test-based validation, and manual confirmation—provides a reusable template for future studies on software evolution and semantic code analysis.

### Summary

In summary, the contributions of this dissertation are fourfold:

1. Large-scale empirical evidence on version compatibility risks in reused Python snippets (Study 1 / D-RQ1).
2. A browser-integrated tool and aligned evaluation for automated compatibility detection and support as an existence proof (Study 2 / D-RQ2).
3. A curated dataset and analyses for test-based functional equivalence and similarity research in Python (Study 3 / D-RQ3).
4. An integrated perspective that connects compatibility, equivalence, and similarity to support sustainable software evolution (Definition 1.1).

## 1.7 Structure of the Dissertation

This dissertation is organized into three main studies that align one-to-one with the three dissertation-level research questions (D-RQ1–D-RQ3) defined in Section 1.4. Chapters 2–4 present the three studies, followed by an integrative discussion and conclusion.

Chapter 2 (**Study 1: Empirical Analysis of Version Compatibility in Reused Python Code**) addresses **D-RQ1** by conducting a large-scale empirical investigation of Python code snippets collected from Stack Overflow and assessing their version compatibility primarily at the syntax level (interpretability).

Chapter 3 (**Study 2: PyVerDetector—A Browser-Based Tool for Real-Time Compatibility Support**) addresses **D-RQ2** by presenting the design and implementation of **PyVerDetector** and evaluating it in alignment with the two aspects of D-RQ2: (i) detection/visualization feasibility and correctness for *syntax-level* version-compatibility signals, and (ii) the practical usefulness of such signals in snippet reuse settings.

Chapter 4 (**Study 3: Functional Equivalence and Similarity in Python Repositories**) addresses **D-RQ3** by constructing and analyzing a dataset of Python method pairs to study test-based functional equivalence and structural similarity, and by evaluating the capability of large language models to recognize such relationships.

Chapter 5 synthesizes findings from Studies 1–3 to discuss how compatibility, equivalence, and similarity interact in practice, and to derive implications for sustainable software evolution under continual ecosystem change.

Finally, Chapter 6 concludes the dissertation by summarizing the main results and contributions, explicitly reflecting on D-RQ1–D-RQ3, and outlining directions for future work.

# Chapter 2

# Empirical Analysis of Code Compatibility on Stack Overflow

## 2.1 Introduction

This chapter reports Study 1 of this dissertation. It addresses D-RQ1 by providing large-scale empirical evidence on the prevalence and characteristics of Python version incompatibilities in reused code snippets on Stack Overflow. The results quantify how often widely reused snippets are syntactically incompatible across Python 2 and Python 3, how frequently such incompatibilities are explicitly indicated, and how quickly the community responds to new Python releases. These findings motivate the need for in-context, automated support, which is developed and evaluated in Chapter 3 (Study 2).

Stack Overflow is a popular developer Q&A platform that provides a venue for disseminating knowledge and exchanging ideas, allowing users to post questions, provide answers, and search for content that interests them. Code snippets represent the core knowledge shared on Stack Overflow, as users regularly integrate them into their questions or answers for heightened clarity and understanding. As of August 2023, Stack Overflow has over 24 million questions, and 35 million answers [19]—a testament to its active user base and a number that continues to grow daily.

Within these questions and answers lies a treasure trove of code snippets. This abundance of available code offers developers an easy path to finding solutions to everyday programming challenges. It is commonplace for developers to copy code examples from Stack Overflow, highlighting its integral role in the development process [9].

Despite the convenience Stack Overflow offers in finding the needed code snippets, recent research has highlighted that these code snippets can

be toxic [61], outdated [87, 89], or of low quality [83]. These issues can further lead to compromised software quality [21, 87], license violations [6], or migration of security vulnerabilities [75].

Numerous factors contribute to the problems associated with code snippets on Stack Overflow, with the use of outdated programming language features in these code snippets being a major concern. Programming languages are inherently fluid, constantly evolving to meet new needs and sustain longevity. To signify this evolution, popular programming languages often employ versioning, wherein more recent versions generally denote more mature forms of the language.

The issue of backward compatibility is a key consideration for many programming languages as they evolve. This principle would allow programs written in an earlier language version to be compiled and run using a later version while exhibiting the same behavior as in the previous version. This principle, however, is violated by Python. With the release of Python 3.0, the language intentionally broke backward compatibility with its predecessor, Python 2.

This lack of backward compatibility poses significant challenges to the users of Stack Overflow. For instance, a user might find a useful Python 2 code snippet on the platform that may not be directly compatible with their Python 3 project. Consequently, the usability of Python snippets on Stack Overflow could be significantly hampered due to these compatibility issues.

To better understand the extent and nature of these challenges, we conduct Study 1 in this chapter, addressing D-RQ1 (Section 1.4) through an empirical quantification of syntax-level version-compatibility risks in Python code snippets reused from Stack Overflow. To operationalize D-RQ1 in a measurable way, we formulate the following study-specific research questions:

- **S1-RQ1: How many Python code snippets have version compatibility issues in the good answers to Stack Overflow questions?**

  **Answer to S1-RQ1:** About 13% of code snippets have version compatibility issues in the good answers to questions.

- **S1-RQ2: How many of the code snippets interpretable only by Python 2 or only by Python 3 have Python version-specific identification?**

  **Answer to S1-RQ2:** Only about 20% of code snippets interpretable only by Python 2 or only by Python 3 have accurate Python version-specific identification.

- **S1-RQ3: How do users on Stack Overflow react and adapt to the introduction of new Python releases?**

  **Answer to S1-RQ3:** After the release of a new Python version, responses can generally be received on Stack Overflow on the same day.

## 2.2 Background

### 2.2.1 Stack Overflow

Stack Overflow, a widely used online community platform, serves as an essential resource for developers looking to share and acquire programming knowledge through a question-and-answer (Q&A) format. Questions posted on the platform invite solutions from the community, which are formatted as answers. Crucial to Stack Overflow's efficacy is the code snippets embedded within these questions and answers. These provide specific contexts or demonstrate direct solutions, acting as ready-to-use references. This characteristic significantly bolsters Stack Overflow's appeal among developers.

For instance, consider a post titled "How do I check if a variable exists?[1]", as shown in Figure 2.1. In this post, the questioner employs a blend of textual descriptions and code snippets to articulate the question effectively, using three tags to encapsulate the question's topic succinctly. The answer presents a concise solution exemplified via a code snippet. This instance not only underscores the issue and its precise solution but also illuminates the collaborative and community-driven nature of knowledge exchange that defines Stack Overflow.

Beyond the code snippets, Stack Overflow has other distinguishing features that enhance its utility. One such feature is its community voting system, which allows users to upvote or downvote questions and answers based on their usefulness or relevance. The platform also employs a tagging system to categorize questions, streamlining the search and discovery process.

### 2.2.2 The Evolution of Python and Version Compatibility

Backward compatibility refers to the ability of software to function seamlessly when a newer version of the language is used without modifying the existing code. Ensuring this continuity gives developers confidence that their previously written code will not become obsolete with each new language update.

[1] https://stackoverflow.com/questions/843277

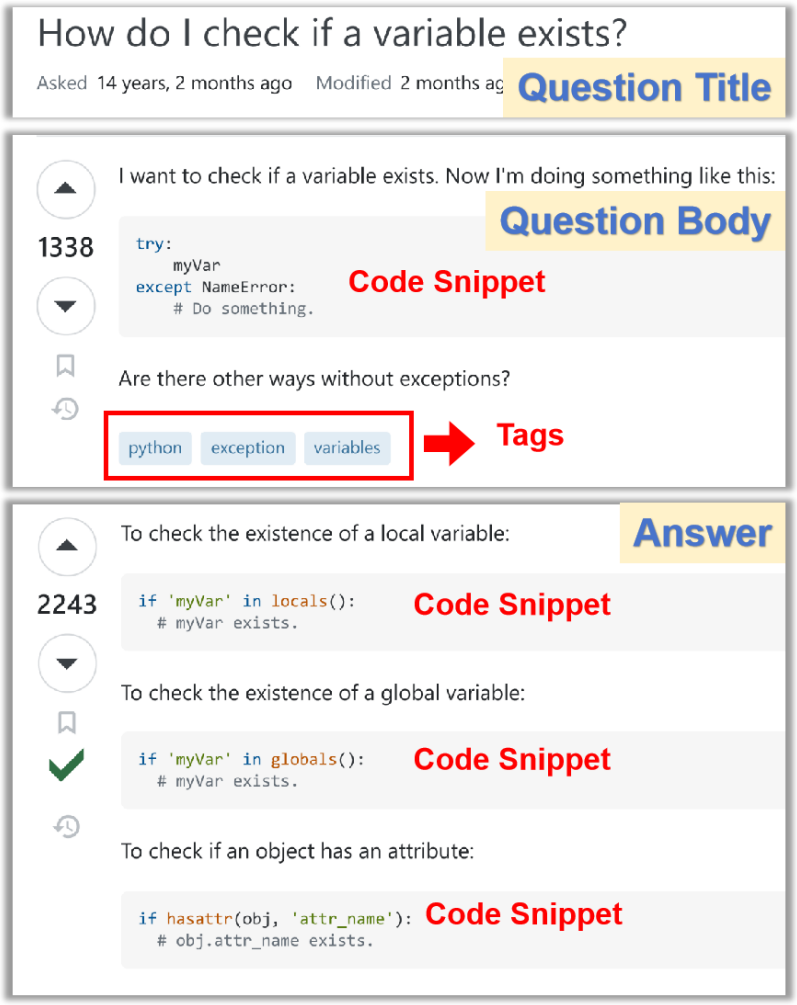



Figure 2.1: An example of a Q&A post on Stack Overflow

### Development history of Python

Python 2.0 was released in October 2000, bringing various features that appealed to developers. Known for its readability and uncluttered syntax, the language comes with an array of built-in data types, such as tuples, lists, sets, and dictionaries. Beyond these core capabilities, Python also boasts a comprehensive standard library and a rich repository of user-contributed packages, facilitating rapid prototyping and effective system integration. Its powerful scripting capabilities also make it versatile for various tasks

[70]. These advantages have collectively propelled Python to become one of today's most popular and widely used programming languages.

The trajectory of Python experienced a monumental shift in 2008. The launch of Python 2.6 aimed to prolong the success of the popular 2.x series, guaranteeing a continuation of support and development. Concurrently, Python 3.0 emerged as a transformative iteration focusing on modernizing and refining the language, even at the expense of forfeiting backward compatibility.

Although Python 2 offered a rich feature set, it eventually hit a technological ceiling that hindered further advancements. This prompted the Python community to make a calculated shift, strategically phasing out Python 2 in favor of the more progressive Python 3. The community's decision reached its apex when support for Python 2.7 was officially terminated on January 1, 2020, marking the series' end-of-life [68].

This strategic bifurcation led to both gains and drawbacks. On the positive side, it enabled Python to innovate and push the boundaries, solidifying its rapid ascendance in the tech world. However, this bold move also led to a bifurcation within the Python community and posed challenges for projects transitioning from Python 2 to Python 3 [45].

#### Backward Compatibility in Python Versions

In its evolution, Python has been known to break backward compatibility almost with each new release. This policy, including rules that govern the instances when compatibility can be broken, is meticulously documented in PEP 387 [54]. Python 3.0, for instance, markedly broke backward compatibility with Python 2. Moreover, each subsequent release since Python 3.5 has seen the removal of deprecated features essential for running older Python programs, with most releases introducing minor changes that impact only a small proportion of language features.

#### Compatibility Issues of Python Snippets on Stack Overflow

The parallel paths of Python 2 and 3 heralded not only unique challenges in language enhancement but also implications for the vast reservoir of Python snippets on platforms like Stack Overflow. Historically, Python 2 was rich in capabilities but eventually encountered a juncture where it couldn't encompass newer advancements. This prompted Python's custodians to strategically sunset Python 2 in favor of Python 3, culminating in the community's decision to officially terminate support for Python 2.7 as of January 1, 2020, marking the end-of-life for this Python iteration [68].

As an integral knowledge repository, Stack Overflow is brimming with Python code snippets spanning various versions. Yet, the divergence between Python 2.x and 3.x has cast a shadow over the applicability of these

snippets. Many code snippets authored during the Python 2.x epoch may not operate smoothly within the Python 3.x framework, sparking concerns about their lasting utility. This landscape accentuates the pressing need to assess the Python snippets on Stack Overflow critically. It's essential to ascertain their compatibility with contemporary Python standards, ensuring they remain valuable resources for the vast and evolving community of Python developers.

## 2.3 Study Approach

### 2.3.1 Data Collection

To unveil the challenges of version compatibility within Python code snippets on Stack Overflow, we decided to use the collection of Python code snippets from the "good answer" on the platform. As introduced in Section 2.2, Stack Overflow features a voting system for questions and answers. This chapter defines a "good answer" as the one with the highest positive score. Note that not every question has a good answer; those without are omitted according to the data selection criteria specified in this section. Moreover, in instances where several answers to a question have the same score, all such answers are regarded as good answers. The reason for choosing good answers is that they are expected to be correct answers and most likely to be used by other users. This section diligently delineates the systematic approach we have adopted to extract Python code snippets from good answers on Stack Overflow.

**Data Selection Criteria:** To conduct our empirical study, we obtained Python code snippets from the good answers on Stack Overflow.

When an original questioner posts a question on Stack Overflow, tags are added to describe the topic of the question, we use these tags to distinguish questions related to Python. Additionally, the content of the posted code snippet can differ from the initial posting due to edits over time. In light of these considerations, we focused on the most up-to-date data available to users in exploring the compilability of the current code snippets. Therefore, we employed the following five criteria to pinpoint the code snippets needed for our study:

- Code snippets extracted from good answers to questions whose tags include "python", or whose corresponding answers contain textual mentions of "python".
- Answer scores must be positive.
- Answers that share the same positive score are all recognized as good answers.

- Questions lacking answers are omitted.
- Posting data is the latest version.

**Data Source:** We use SOTorrent [10] to extract Python code snippets on Stack Overflow. SOTorrent is an open dataset based on the official Stack Overflow data dump, which provides access to the version history of Stack Overflow content at the level of whole question posts and individual text or code blocks. SOTorrent has been continuously updated with many versions since its creation. When we started our study, the latest version of SOTorrent was SOTorrent20_03 as of March 15, 2020[2].
**Data Extraction:** Applying the established criteria to the SOTorrent dataset, we successfully extracted 1,498,133 questions. These questions led to 2,161,905 answers, among which 976,807 were good answers. Acknowledging that an answer could contain multiple code snippets, we had a considerable corpus of 1,376,571 code snippets.
**Data Preprocessing:** During the data extraction process, we noticed that code snippets stored in the SOTorrent database could introduce formatting issues that were not originally present, such as redundant or inconsistent indentation. These formatting issues might hinder subsequent analysis efforts, prompting us to preprocess the data. This step primarily involved rectifying redundant or inconsistent indentations within the code snippets, ensuring their readiness for future parsing and analysis. Moreover, we tackled instances where the code was framed in a read-eval-print loop (REPL) mode, further enhancing the quality and uniformity of our dataset.

### 2.3.2 Code Snippet Analysis

To investigate version-related incompatibilities in Python code snippets on Stack Overflow, we analyze each snippet under multiple Python interpreter versions (Python 2.7 and Python 3.5–3.8). The selection of these versions was driven by the availability of the Python interpreters and the timeline of the SOTorrent release we employed, which incorporates Python 3.8 as the latest version. We excluded older versions due to the challenges in preparing an execution environment and newer versions released after the dataset. Following the compatibility scope defined in Section 1.3, we operationalize compatibility at the syntax level, i.e., syntax-level compatibility (interpretability). Concretely, a snippet is considered syntax-level compatible with a Python version if that version's interpreter can successfully parse and compile the snippet (without executing it).

We implement this check using Python's native py_compile module under each interpreter version. For a given version, we attempt to compile

[2]https://zenodo.org/record/3746061

the snippet; if compilation succeeds, we record the outcome as pass (interpretable) for that version, otherwise we record fail (uninterpretable). The resulting pass/fail profile across versions provides a large-scale, version-aware view of syntax-level compatibility for the extracted snippet corpus.

We intentionally focus on parse/compile-time outcomes rather than runtime execution. This choice enables scalable analysis on a corpus of over one million snippets and avoids the need to construct inputs or reproduce execution environments for each snippet. Moreover, in the copy–paste reuse scenario, syntax-level failures (e.g., SyntaxError and other parse/compile-time errors) often constitute the first and most direct form of breakage that prevents reuse.

Because our procedure does not execute snippets, it does not resolve or validate runtime dependencies (e.g., third-party libraries). Import statements are not executed during compilation, and therefore missing third-party packages typically do not surface as compile-time failures in our pipeline. As a result, our measurements isolate version-induced, syntax-level incompatibilities, while runtime errors (including missing dependencies) and behavioral differences remain outside the scope of this chapter and are discussed as threats to validity.

## 2.4 Results

Based on the data collection and analysis procedure described in Section 2.3, we now report the results organized around S1-RQ1 to S1-RQ3 (introduced in Section 2.1). In this section, we explore the underlying motivation, the approach taken, and the results of our three research questions concerning the challenges of version compatibility within Python code snippets on Stack Overflow.

### 2.4.1 S1-RQ1: How many Python code snippets have version compatibility issues in the good answers to Stack Overflow questions?

**Motivation:** This research question aims to measure the prevalence of version compatibility issues in Python code snippets from good answers on Stack Overflow. Given that the platform is a widely used resource for developers, understanding the scope of such issues is crucial. Code snippets with compatibility issues may prevent users from reusing the code. By quantifying the extent of these version compatibility issues, this research question offers actionable insights into the prevalence and nature of version-specific problems in Python code snippets. Such insights can guide Python programmers in adopting best practices for version compatibility, inform educators in structuring their coding curricula, and provide tool developers

with a clearer understanding of common issues to address in future updates, aiding them in their respective endeavors.

**Approach:** In addressing this research question, we parsed these Python code snippets from the good answers on Stack Overflow using Python 2.7, Python 3.5, Python 3.6, Python 3.7, and Python 3.8. And we marked whether it is interpretable in each version as described in Section 2.3.2. In this way, we compiled a comprehensive catalog of Python versions compatible with each snippet.

**Results:** Analyzing the parsing results, we found that 440,456 (32.0%) code snippets failed to be parsed by all Python versions, i.e., these code snippets are uninterpretable for all Python versions. There are several possible reasons: 1) our study does not cover very old Python versions (e.g., 2.0 or 3.0), 2) some answers associated with Python-related questions contain non-Python content (e.g., shell commands, outputs, or other languages), 3) snippets may contain genuine syntax errors, (4) many snippets are incomplete fragments (e.g., missing surrounding lines, headers, or matching indentation), and 5) some snippets include other formatting artifacts that remain difficult to normalize perfectly. Therefore, "fail for all analyzed versions" primarily captures snippet-quality/context issues rather than version-upgrade-induced incompatibilities.

In other words, they are not caused by the Python version upgrade we would like to investigate. In addition, we found that 755,699 (54.9%) code snippets could pass the parsing for all Python versions, i.e., these code snippets were interpretable for all Python versions. This may be the case because they are not using some features that would affect the compilability of code snippets vary between Python versions.

Finally, the remaining 180,416 (13.1%) code snippets are the parts that face compatibility issues across Python 2.7 and Python 3.x. This percentage indicates a significant challenge within the Python community, suggesting that a notable fraction of shared code on Stack Overflow may be incompatible with either Python 2 or 3. For these code snippets that face compatibility issues, we found that they can be divided into the following three categories:

**Pass Python 2.7, Fail for some Python 3:**
The code snippet passes Python interpreter parsing for Python 2.7 and at least one Python interpreter parsing for Python 3, but not all Python 3 versions.

**Pass Python 2.7, Fail for all Python 3:**
The code snippet fails for any Python 3 Python interpreter parsing but passes Python 2.7 Python interpreter parsing.

Table 2.1: Python code snippets facing compatibility issues in Stack Overflow.

| **Categories** | **#Snippets** | **Overall percentage** |
|---|---|---|
| Pass Python 2.7, Fail for some Python 3 | 59,772 | 4.3% |
| Pass Python 2.7, Fail for all Python 3 | 72,932 | 5.3% |
| Fail for Python 2, Pass Python 3 (all or some) | 47,712 | 3.5% |
| **Total** | **180,416** | **13.1%** |

**Fail for Python 2.7, Pass Python 3 (all or some):**
The code snippet fails for Python 2.7 Python interpreter parsing but passes all or some Python 3 Python interpreter parsing.

Of the above categories, the results are shown in Table 2.1, which provides a comprehensive overview of the situation of the parts of Python code snippets that face compatibility issues extracted from good answers on Stack Overflow. The "Overall percentage" column in the table indicates the proportion of the three categories of code snippets relative to all snippets extracted from the good answers on Stack Overflow. Notably, as indicated in the table, a considerable portion of these code snippets is exclusively interpretable by Python 2. This demonstrates the persistence of legacy Python 2 code within the developer ecosystem, underscoring the challenges in migrating to Python 3 despite its growing adoption and the cessation of official support for Python 2.

These results underline the extent of version compatibility issues within Python code snippets on Stack Overflow, pointing to a pressing need for attention and potential improvements in this area.

**Answer to S1-RQ1:** About 13% of code snippets have version compatibility issues in the good answers to questions.

### 2.4.2 S1-RQ2: How many of the code snippets interpretable only by Python 2 or only by Python 3 have Python version-specific identification?

**Motivation:** Within Stack Overflow, question tags and the answer texts are crucial navigational tools, guiding users toward content that aligns with their specific programming needs. Stack Overflow's tagging system includes "Python 2.x" and "Python 3.x" tags, allowing users to designate their questions as specifically pertaining to Python 2 or Python 3. Similarly, mentioning the Python version in the answer texts can guide users in understanding the applicability of the given code snippet. Correctly identifying

Table 2.2: The results of version annotation for code snippets in the good answers on Stack Overflow, which are only compatible with either Python 2 or Python 3.

| **Categories** | **#Snippets** | **#Snippets (tagged)** | **#Snippets (text, untagged)** | **Percentage (version-specific)** |
|---|---|---|---|---|
| Fail for Python 2, Pass Python 3 (all or some) | 47,712 | 9,981 | 1,944 (2,356) | 25% |
| Fail for all Python 3, Pass Python 2 | 72,932 | 9,202 | 2,418 (2,418) | 16% |

a code snippet as exclusively interpretable by either Python 2 or Python 3, through tags or textual mentions (i.e., version-specific identification) is crucial. For instance, if a code snippet is solely compatible with Python 2 and is correctly identified as Python 2, it helps users avoid inadvertently using this snippet in a Python 3 context, and vice versa. Nevertheless, the effectiveness of this approach is contingent on the accurate application of these tags and textual mentions. In our research, we aim to determine how many Python code snippets can be interpreted exclusively by Python 2 or Python 3 are accurately identified with their respective version, either through tags or textual mentions in the corresponding answers. This analysis will assess the precision of Python version-specific identification on Stack Overflow and identify potential areas for enhancement to improve the user experience on the platform.

**Approach:** To address this research question, we turned to the parsing results of the code snippets obtained in S1-RQ1. We focused on the categories "Fail for Python 2.7, Pass Python 3 (all or some)" and "Fail for all Python 3, Pass Python 2.7" as they represent code snippets exclusively compatible with Python 3 and Python 2, respectively. These identified snippets were then scrutinized for the presence of corresponding "Python 2.x" or "Python 3.x" tags, or textual mentions in their associated answers, to determine whether they were for Python 2 or Python 3. This enabled us to gauge the accuracy of version-specific identification in relation to the actual version compatibility of the code snippets.

**Results:** The findings from our study are presented in Table 2.2. The first row of the table shows that out of 47,712 code snippets that are solely compatible with Python 3, only 9,981 have been correctly tagged with "Python 3.x". Furthermore, 2,356 snippets mention the Python version in the text, of which 1,944 snippets mention the correct version in the text without the "Python 3.x" tag. This corresponds to only about 25% of Python 3-specific code snippets being properly labeled. This indicates a sizable gap in version-specific identification, highlighting that nearly 75% of these snippets lack the crucial information to inform users of their compatibility solely with Python 3.

Similarly, the second row reveals an even more glaring issue: among the 72,932 code snippets that are only interpretable by Python 2, merely 9,202

have been correctly tagged as "Python 2.x". Additionally, 2,418 snippets include a text mention of the Python version, and all of these are without a corresponding version tag. This amounts to a mere 16% of Python 2-specific snippets being appropriately labeled, leaving a staggering 84% of these snippets without clear version-specific indications.

The discrepancy between the 25% accurate identifying rate for Python 3 and the 16% for Python 2 demonstrates that Python 3-specific snippets are, on average, more likely to be correctly identified than their Python 2 counterparts on Stack Overflow.

These numbers reveal a critical issue with the current state of version-specific identification on Stack Overflow. Despite the platform's wide use and the community's dependency on it for reliable coding solutions, the lack of accurate version labeling potentially misleads users and hampers effective code reuse. In summary, the data call for immediate improvements in version-specific identification practices on the platform to better guide its vast user base, especially those working in mixed-version Python environments.

**Answer to S1-RQ2:** Only about 20% of code snippets interpretable only by Python 2 or only by Python 3 have accurate Python version-specific identification.

### 2.4.3 S1-RQ3: How do users on Stack Overflow react and adapt to the introduction of new Python releases?

**Motivation:** The evolution of Python, with its frequent version updates, greatly influences the coding practices among its community, particularly regarding version compatibility. Given the importance of Stack Overflow as a knowledge base for Python users, understanding how users react and adapt to new Python releases can provide critical insights into the challenges and solutions associated with Python version compatibility.

**Approach:** To investigate the response of Stack Overflow users to each Python release, we sourced the release dates of each Python version directly from Python's official website. Our approach tracks the evolution of Python code snippets corresponding to each version since its release. This allows us to gauge the platform-wide response in coding practices to each Python version update. We defined a Python code snippet "responding" to a certain Python version based on two criteria:

- The posting or the most recent modification date of the code snippet must be after the release of that Python version. While this does not guarantee that the snippet will include newly introduced language features or syntax, it increases the likelihood that the snippet is influenced by or compatible with that version.

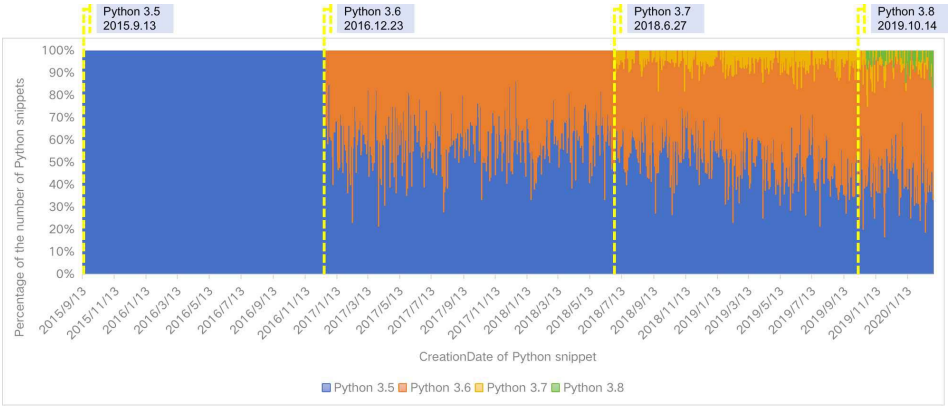



Figure 2.2: Responses by Python version on Stack Overflow.

- The code snippet should be syntax-level compatible with the target version (i.e., it passes interpreter-based compilation for that version) and fails interpreter-based compilation for all preceding analyzed versions, suggesting that it uses syntax introduced in the newer version.

Our study primarily focuses on Python versions 3.5 through 3.8 for this research question. Additionally, we incorporate Python 2.7 from the Python 2 series for comparative insight.

**Results:** To precisely assess the response of Stack Overflow users to each Python version release, we categorized code snippets corresponding to each version on a daily basis. Figure 2.2 presents a stacked area percentage chart for various Python versions, where the release date for each Python version is marked with a yellow dashed line. The horizontal axis represents the code snippets' post or the latest modification date, while the vertical axis indicates the percentage of each Python version in all response code snippets.

Figure 2.2 shows that, except for Python 3.5 (which was not assessed against previous versions), the three other Python versions (3.6, 3.7, and 3.8) elicited a rapid and substantial response within days of their release. This result not only underscores the enthusiastic attitude of the Python developer community towards new versions but also manifests the swift acceptance and incorporation of unique features offered by new versions. Particularly noteworthy is the prominence of Python 3.6 among the community. Several factors may contribute to this phenomenon. Python 3.6 introduced compelling new features like f-strings, which made string formatting more intuitive. It also received performance optimizations and benefitted from broader third-party library support, making it a particularly attractive choice. Its longer-term support has also rendered it a stable option for ongoing projects. These factors likely fueled rapid community engagement with Python 3.6, emphasizing the Python ecosystem's

dynamism and willingness to embrace new advancements. Furthermore, when investigating the response time of Python 3.8, we noticed a certain delay compared to other Python versions. This delay may have various reasons, including the complexity of new features, community response time, code example availability, and version migration speed. However, since this study focused on the analysis of code snippets and did not conduct a detailed study of the full-text content of posts, there is not enough data to comprehensive explanation for this delay. An extensive analysis of the full textual content of the posts is needed to fully understand the community's reaction to the release of the new Python version. Specifically, developers are eager to experiment with and integrate new capabilities into their code, emphasizing the language's adaptability and the community's openness to change.

**Answer to S1-RQ3:** After the release of a new Python version, responses can generally be received on Stack Overflow on the same day.

## 2.5 Threats to Validity

Our research faces inherent limitations and threats to validity that are worth considering.

**Data and scope limitations:** A substantial portion of the extracted snippets are not parsable/compilable under any of the analyzed Python versions. This subset may include (i) non-Python content (e.g., command outputs or mixed-language blocks), (ii) incomplete fragments intended only as illustrative pseudo-code, and (iii) formatting artifacts introduced during extraction. Because our goal is to quantify version-induced breakage rather than general snippet quality issues, we do not attempt to recover or "repair" these snippets, and we exclude them from cross-version compatibility comparisons. This decision may bias the reported prevalence upward or downward depending on how many of these snippets would become parsable if additional context or corrections were available.

**Construct validity:** We operationalize "version compatibility" as syntax-level interpretability: a snippet is considered compatible with a Python version if it can be successfully parsed/compiled by that version's interpreter. This operationalization is intentional and consistent with the dissertation's scope: syntax-level compatibility is a necessary precondition for execution and a direct indicator of copy–paste breakage caused by language-version differences. Hence, the reported incompatibility rates should be interpreted as syntactic (parse/compile-time) incompatibilities rather than comprehensive runtime or behavioral incompatibilities.

This operationalization does not capture all incompatibilities that matter in practice. A snippet that compiles may still fail at runtime due to missing dependencies, required inputs, environmental assumptions, or behavioral changes across versions. Conversely, a snippet may fail to compile in isolation because it is incomplete or depends on surrounding context not included in the snippet. We mitigate tool-induced threats by using interpreter-based compilation (rather than a custom parser) and by applying preprocessing (e.g., indentation/REPL formatting corrections) to reduce formatting noise, but the measured construct remains syntax-level interpretability.

**External validity:** Our empirical results are derived from Python code snippets embedded in good answers on Stack Overflow and analyzed under a specific notion of version compatibility (syntax-level interpretability). Consequently, the observed prevalence of incompatibilities (e.g., the fraction of snippets that are interpretable in Python 2.7 but not in Python 3.x, or vice versa) should not be assumed to transfer unchanged to other settings. First, the platform may matter. Stack Overflow has a distinctive Q&A workflow (community voting, accepted/high-score answers, post edits, and tagging practices), and these mechanisms can influence both the visibility and the maintenance of code examples. Other platforms (e.g., alternative Q&A forums, issue trackers, blog posts, or repository-hosted snippets) differ in moderation, incentives for updating old answers, and the typical amount of surrounding context provided with code. As a result, the proportion and the types of version-related failures may differ. For example, platforms with weaker editing culture may preserve more legacy snippets, whereas platforms with stricter moderation or stronger conventions for indicating environments may exhibit fewer accidental mismatches. Second, the language and ecosystem transition may matter. Python 2→3 is a particularly well-known and intentionally backward-incompatible transition, which makes it a data-rich and representative case for studying version-induced breakage in reused snippets. However, other ecosystems have also experienced disruptive transitions (e.g., Swift 2→3, major PHP upgrades, or major Ruby-version shifts). The prevalence of snippet incompatibility in such cases could be smaller than in Python, depending on the language's compatibility policy, migration tooling, and community adoption patterns. Importantly, a smaller proportion does not necessarily imply limited practical usefulness: even a single-digit incompatibility rate can still affect a large absolute number of widely reused snippets on large platforms, and failures at the "copy–paste–run" stage can impose disproportionate costs on developers (especially newcomers) because they interrupt learning and debugging workflows. From this perspective, our results should be interpreted as evidence that version-induced breakage is non-negligible even in highly visible ("good answer") content, and that lightweight compatibility

signals can be valuable even when the failure rate is not dominant. To mitigate this threat in future work, the same measurement pipeline can be replicated across (i) other knowledge-sharing platforms and (ii) other language transitions with documented breaking changes, while adapting the data selection criteria to each platform's notion of "high-quality" or "high-visibility" content. Such replications would clarify how the prevalence and characteristics of incompatibilities vary across ecosystems and would help delimit the boundary conditions under which the findings generalize.

Finally, we note that this study focuses on good answers, which are expected to be more curated than arbitrary snippets. This choice improves the relevance to real reuse, but it may also underestimate incompatibility rates compared to less-vetted content (e.g., low-score answers or comments). Therefore, the reported prevalence should be viewed as a conservative estimate for highly visible snippets rather than an upper bound for all online Python code examples.

## 2.6 Related Work

**Quality of Code Snippets on Q&A Platforms** Within the scholarly discourse on Q&A platforms, the quality and reliability of shared code snippets have received widespread attention. The study by Wu et al. [83] underscored the critical need for improved mechanisms to evaluate the applicability of reused code, highlighting the role of such platforms in software development practices. Another investigation revealed a concerning trend of obsolete or license-violating code being circulated, underscoring a deficiency in the upkeep of these code fragments [61]. The ExampleCheck framework's exploration into API misuse within accepted answers on these forums further suggests a pressing need for more stringent solution validation processes [88]. The reliability of Java API-related snippets was specifically questioned. In the work of Zerouali et al. [85], the impermanent reliability of these snippets is brought to the fore, underscoring how library updates can swiftly invalidate previously dependable solutions. This reality underscores the ephemeral nature of code snippet accuracy. To counteract the widespread issue of reliance on outdated libraries in Java snippets on Stack Overflow, Zerouali et al. developed an automated approach for pinpointing Maven library version ranges. Their methodology illuminates the tendency within the developer community to lean on antiquated libraries and signals the need for a systematic reassessment of code snippet maintenance and the protocols for updating them. To address the challenge of outdated library usage in Java snippets on Stack Overflow, Zerouali et al. [85] developed a methodology for automatic identification of Maven library version ranges, highlighting the commonality of reliance on older libraries.

**Compatibility Issues and Tools** The evolution of programming languages and their corresponding updates are pivotal in influencing the longevity and relevance of code snippets shared on platforms like Stack Overflow. In this context, the work by Malloy et al. [45, 46] merits attention, as they delved into the challenges posed by Python version upgrades on code compatibility. Their contribution, a tool named *PyComply*, has been developed specifically to detect the compatibility of Python code snippets with targeted versions of the Python language.

## 2.7 Conclusion

In this chapter, we conducted an empirical study investigating Python code snippet compatibility issues on Stack Overflow. The results are quite telling: Firstly, approximately 13% of code snippets in the good answers to questions exhibit version compatibility issues. This significant percentage underscores the need for more robust mechanisms or tools to assist users in accurately identifying the appropriate Python version for a given code snippet. Secondly, our findings reveal that only about 20% of code snippets interpretable exclusively by Python 2 or Python 3 have accurate Python version-specific identification. Finally, our study observes that new Python versions are quickly adopted and discussed on Stack Overflow, indicating an engaged community of users who readily respond and adapt to these changes.

These findings clearly show the Python version compatibility landscape on Stack Overflow and highlight key areas for potential improvement. We hope this research brings awareness to these issues and spurs further research and development in creating solutions that could substantially enhance the user experience on Q&A platforms like Stack Overflow. For further insights and a detailed look at the code snippets and other valuable information utilized in this chapter, interested readers and researchers can access our dataset on zenodo[3].

[3]https://zenodo.org/records/10790233

# Chapter 3

# Tool Support for Python Version Compatibility Detection

## 3.1 Introduction

Developers frequently reuse code snippets from online Q&A platforms such as Stack Overflow. However, such reuse is risky when a snippet was written for a different execution environment. This risk is particularly salient in Python, which has introduced backward-incompatible changes across major releases, most notably the transition from Python 2 to Python 3.

Chapter 2 quantified this problem on Stack Overflow under the dissertation's syntax-level notion of version compatibility (i.e., interpretability via parse/compile): 13.1% of snippets in high-quality answers are not syntactically compatible across Python 2.7 and Python 3.x, and only about 20% of version-specific snippets are explicitly identified with their target version. These findings motivate the need for in-context, lightweight support that allows developers to check version compatibility before copy-and-paste reuse.

This chapter addresses the dissertation-level research question D-RQ2 (Section 1.4) by presenting PyVerDetector, a browser-based tool that detects and visualizes Python version compatibility for code snippets on Stack Overflow in real time. PyVerDetector provides immediate feedback directly in the reuse context, so users do not need to locally install multiple interpreters or run offline analysis tools to assess compatibility.

PyVerDetector consists of (i) a frontend component that extracts code blocks from web pages and renders compatibility feedback, and (ii) a backend analysis component that checks syntax-level compatibility using version-specific parsers. We evaluate the correctness of PyVerDetector's compatibility decisions against interpreter-based ground truth and compare it with

an existing analyzer (PyComply). We also illustrate practical usefulness through representative usage scenarios.

The remainder of this chapter is organized as follows. Section 3.2 describes the design and implementation of PyVerDetector. Section 3.3 presents usage scenarios. Section 3.4 evaluates the tool's accuracy. Section 3.5 concludes the chapter by summarizing its contributions in the context of the dissertation.

## 3.2 PyVerDetector

We developed a Chrome extension, PyVerDetector to help SO users address the issue of version compatibility of Python code snippets. PyVerDetector has two main features:

1. For each code snippet inside a code block, the tool determines if the code snippet is compiled without errors using the user-selected Python version.

2. If not, the tool provides the error message and the location of the error.

PyVerDetector consists of two components: a frontend part (running on the user's browser) responsible for fetching the code snippet from SO and displaying the results, and a backend part (running on a server) responsible for statically analyzing the given code snippet across multiple Python versions. The overview of PyVerDetector is shown in Figure 3.1. Upon loading a SO page, for each Python snippet, the frontend calls the backend and retrieves the parse result containing all the supported Python versions for that snippet. Finally, the frontend alters the page to present the result to the user. PyVerDetector supports versions (2.0 to 2.7, 3.0 to 3.8).

### 3.2.1 Frontend

The frontend has two main features:

#### Format code snippet

The frontend fetches Python code snippets from the page, formats the code snippets of the REPL mode, and sends them to the backend for parsing. The frontend also copies the formatted code snippet to the clipboard for the user to use.

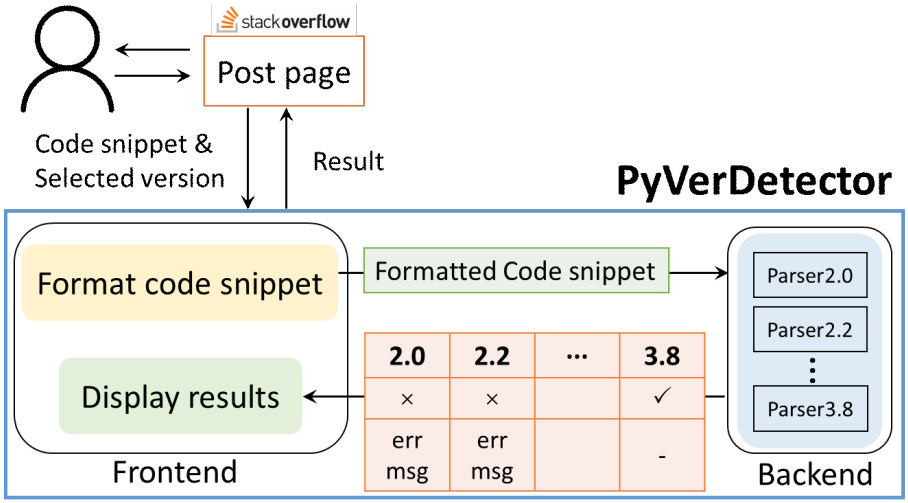



Figure 3.1: Overview of PyVerDetector.

#### Display result

The result from the backend contains the parsing results of the code snippet for all Python versions. The frontend presents them to the user according to the Python version selected by the user in the drop-down menu inserted below the code snippet. The frontend shows the following two types of messages on the page:

- Parsing result for the user-selected Python version (default: Python 3.8). If the code snippet parses successfully under that version, the message `No error for Python X.X` is displayed. Otherwise, the error message and the line number at which the error occurred are displayed.

- If the snippet also parses successfully under other Python versions besides the selected one, list them (e.g., `Also works for: Python Y.Y, Python Z.Z`).

### 3.2.2 Backend

The backend part is inspired by CPython and PyComply [46]. Since it is not practical to wrap multiple execution environments of the old Python versions inside a Chrome extension, we decided to use an Abstract Syntax Tree (AST) parser following the grammar of Python. If the Python snippet can be parsed, with a grammar for a specific Python version, we assume the snippet is compliant for the particular version. We used a combination of

*Flex*[1] and *Bison*[2] to generate the code snippet's AST, reporting compliance only in case this AST is generated successfully.

Naively wrapping the output of *Flex* and *Bison*, however, is not sufficient. Unlike PyComply, the web-based nature of our tool required us to perform heavy modifications to the generated parsers in order to support asynchronous invocations, and join multiple parsers together in a single executable. In our approach, a code snippet is tested against each grammar sequentially, and the various error message collected in a JSON message to be sent to the frontend.

Finally, in order to run the backend inside a web extension, we used the *Emscripten* toolchain[3] to compile the original C code into cross-platform WebAssembly to be bundled inside the extension.

### 3.2.3 Python Grammars and Extensibility

Being our tool based on the *Flex* and *Bison* parser generators, it implies the necessity of having an input grammar representing the Python Language. While writing a different grammar for each Python version is certainly not impossible, being up-to-date with the annual Python release schedule by manually writing a new grammar for each release would require considerable effort nonetheless. Fortunately, the Python website provides the full changelog[4] and grammar of each released version since Python 2.2[5].

These grammars, however, are written for a LL(1) parser, while *Bison* is an LALR(1) parser. Despite every LL(1) grammar being LR(1), but not necessarily LALR(1) [3], these two in practice have a great intersection. For this reason, we managed to write a tool to convert the provided grammars from LL(1) EBNF syntax found in the Python archives to LALR(1) Bison syntax expected by our parser generator.

Unfortunately, since version 3.10, CPython switched from a LL(1) parser to a PEG parser [74], with the grammars being provided only in PEG syntax since Python 3.9. Converting from a PEG grammar to LALR(1) is not as easy as converting from a LL(1) to LALR(1). In fact, the equivalence between PEG and Context-Free Grammars such as EBNF has been proven undecidable [22]. For this reason, our extension works with Python versions up to 3.8, but to extend it to future versions of Python, an additional parser for PEGs should be wrapped alongside the current Context-Free Grammar parser.

[1] `https://github.com/westes/flex`
[2] `https://www.gnu.org/software/bison`
[3] `https://emscripten.org/`
[4] `https://docs.python.org/3/whatsnew/changelog.html`
[5] `https://docs.python.org/release/2.2/ref/grammar.txt`

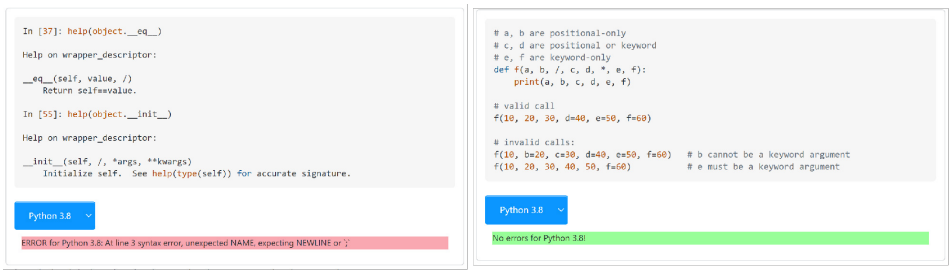



Figure 3.2: Example of how the extension works in the default version. (Left): Parse error, (Right): Pass.

## 3.3 Usage Scenarios

### 3.3.1 Displaying the latest version as default

The default value displayed by PyVerDetector is the parsing result for the latest available version. As shown in Figure 3.2, when the user opens a SO page[6] with some Python code snippets, PyVerDetector will immediately display its parsing results for 3.8 for all Python code snippets on the page. We use green to show pass messages and red to show error messages. This allows the user to quickly get an at-a-glance view of the compatibility of the code snippets for a recent Python version.

### 3.3.2 Accurate display of results for selected versions

When the user wants to know the compatibility of a code snippet for a particular Python version, PyVerDetector can show the user exactly the relevant information. As shown in Figure 3.3, the same question in Figure 3.2, the user has selected 3.7, and PyVerDetector returns an error message that the code snippet cannot be interpreted by 3.7 because the "Positional-only parameters" is used in the fourth line of the snippet. This feature, in fact, has only been supported since 3.8.

## 3.4 Evaluation of Accuracy

In this section, we evaluate the accuracy of PyVerDetector and compare it with an existing tool: PyComply [46].

We evaluate PyVerDetector and PyComply on the Stack Overflow Python snippet dataset, using the interpreter-based ground truth generation procedure described in Chapter 2.

The accuracy of PyVerDetector and PyComply was measured by comparing their respective parsing results with the ground truth in terms of

[6] https://stackoverflow.com/questions/28243832

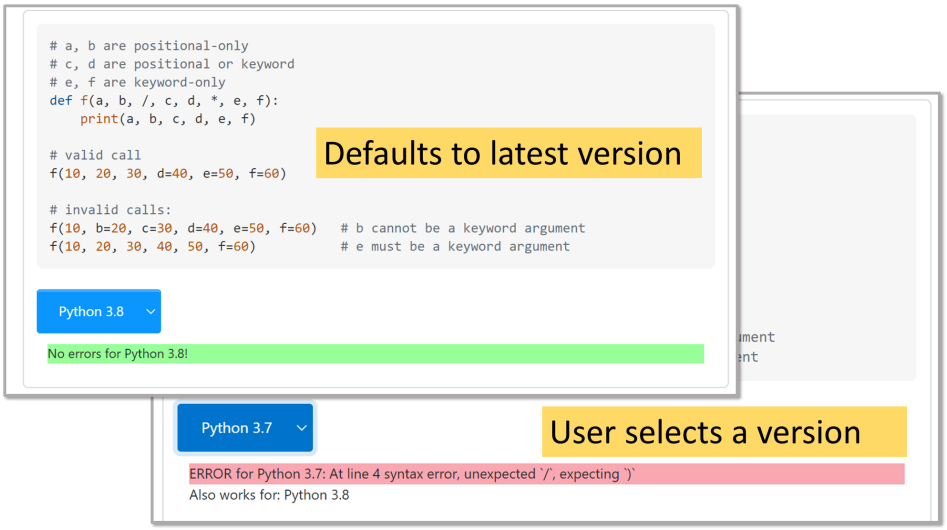



Figure 3.3: User manually selects a version of Python, incompatible with the given snippet.

code snippets. We evaluate PyVerDetector using the well-known metrics for binary classification: precision, recall, and accuracy.

The results are shown in Table 3.1. We can see that PyVerDetector (PyVer) and PyComply (PyC) have the same accuracy for the three Python versions 2.7, 3.5, and 3.6. However, PyVerDetector can provide code detection for the two newer Python versions, 3.7 and 3.8, and has shown high accuracy in both versions.

Table 3.1: Comparison results of PyVerDetector (PyVer) and PyComply (PyC).

| **Python Version** | **Precision** | | **Recall** | | **Accuracy** | |
|---|---|---|---|---|---|---|
| | **PyVer** | **PyC** | **PyVer** | **PyC** | **PyVer** | **PyC** |
| Ver2.7 | 98.28% | 98.28% | 99.95% | 99.95% | 98.82% | 98.82% |
| Ver3.5 | 98.02% | 98.02% | 99.98% | 99.98% | 98.72% | 98.72% |
| Ver3.6 | 97.98% | 97.98% | 99.98% | 99.98% | 98.67% | 98.67% |
| Ver3.7 | 97.98% | – | 100.00% | – | 98.68% | – |
| Ver3.8 | 97.98% | – | 100.00% | – | 98.67% | – |

## 3.5 Related Work

This chapter focuses on tool support for checking Python version compatibility at the moment of online code reuse, complementing Chapter 2 which reviewed prior work mainly from the perspective of platform studies and

empirical analyses of snippet quality and compatibility. In contrast, we summarize related work here along three tool-oriented directions: (1) compatibility checking and migration tools, (2) in-context developer assistance, and (3) version inference and metadata-based approaches.

**Compatibility checking and migration support:** A large body of tooling aims to help developers cope with version changes in evolving language ecosystems. For Python, prior research proposed offline analyzers such as PyComply [45, 46] to determine whether a snippet can be parsed under specific Python versions, and to characterize cross-version incompatibilities.

In practice, migration-oriented tools (e.g., source-to-source translators and upgrade assistants such as 2to3 [60] and other modernization utilities[7]) can help update legacy Python code, but they typically operate on a local codebase and target migration rather than providing lightweight, immediate feedback during web-based snippet reuse.

In comparison, PyVerDetector is designed for an in-browser setting where developers read and reuse snippets on Stack Overflow; it provides real-time, in-context feedback without requiring users to set up local analyzers or run migration pipelines.

**Linting and static analysis in development environments:** Linters and static analyzers (e.g., Python linters and code-quality checkers) can detect certain syntax errors and problematic constructs, and some can be configured to target a particular interpreter version.[8]

However, such tools are usually integrated into IDEs or CI pipelines and assume a project context (dependencies, configuration, and file structure). They are not intended to offer multi-version interpretability signals for small, context-free snippets encountered on web pages. Even tools that explicitly target version compatibility at the project level (e.g., vermin[9], which infers the minimum required Python version) primarily operate on local codebases and are not designed for in-browser snippet reuse scenarios. Moreover, many development-time tools are optimized for code quality or best practices rather than explicitly answering "which Python versions can parse this snippet?". This motivates a browser-integrated design that focuses on a clear and scalable compatibility criterion aligned with the reuse scenario.

**In-context assistance for code reuse:** Recent developer workflows increasingly benefit from in-context assistance, such as IDE feedback (syntax highlighting, quick fixes) and web-based augmentation [13]. There is also a line of research exploring tighter integration of crowd knowledge and

[7] https://github.com/asottile/pyupgrade

[8] `https://github.com/python/mypy`; `https://github.com/pylint-dev/pylint`; `https://github.com/astral-sh/ruff`

[9] https://github.com/netromdk/vermin

Q&A resources into development environments, aiming to reduce context switches between the IDE and the browser [27, 57, 58]. The key difference is that IDE assistance typically applies after code has been copied into a local project, whereas snippet reuse often happens earlier: developers decide whether and how to reuse code while reading answers on Q&A sites. PyVerDetector explicitly targets this earlier decision point by surfacing compatibility signals directly on the Stack Overflow page, reducing the chance that developers copy code that is syntactically incompatible with their intended Python version.

**Version inference and metadata-based approaches:** Another line of work attempts to infer version information from surrounding metadata, such as tags (e.g., python-2.x or python-3.x) and textual mentions in answers. While such cues can be helpful, they can be missing, inconsistent, or not specific enough to reliably characterize the intended runtime environment for a given snippet [28]. Relatedly, prior work has proposed metadata- or dependency-oriented inference methods, such as identifying version ranges of referenced libraries for code snippets [86]. In contrast, parser-based checking provides a deterministic and content-driven signal: given a specific grammar/interpreter definition, the tool can directly determine whether a snippet is syntactically interpretable by a target version. PyVerDetector adopts this principle and operationalizes it in a lightweight, browser-based workflow.

In summary, existing approaches either (i) operate offline and require local setup, (ii) focus on migration and refactoring rather than on-the-spot reuse decisions, or (iii) rely on imperfect metadata. PyVerDetector complements these approaches by providing real-time, in-context version compatibility detection for online code snippets, thereby bridging empirical evidence (Chapter 2) and actionable developer support (this chapter).

## 3.6 Conclusion

In this chapter, we presented PyVerDetector, a browser-based tool that provides in-context, syntax-level version-compatibility signals for Python snippets embedded in Stack Overflow pages. Building on the empirical evidence from Chapter 2 that version incompatibilities are non-negligible and that explicit version cues are often missing, PyVerDetector operationalizes these findings into actionable feedback at the point of code reuse.

Technically, PyVerDetector extracts Python code blocks from a page, checks their syntax-level compatibility (interpretability) against multiple Python versions, and visualizes pass/fail outcomes together with informative error messages. Empirically, our evaluation shows that PyVerDetector's predictions closely agree with interpreter-based ground truth and that

it supports a broader range of versions than prior tools such as PyComply in our setting.

Overall, this chapter demonstrates that lightweight, non-executing compatibility checks can meaningfully reduce the risk of copying incompatible snippets during browsing. At the same time, the tool is intentionally scoped to syntax-level compatibility and does not guarantee runtime correctness or behavioral equivalence—limitations that motivate the shift in Chapter 4 toward semantic-level analysis via test-based functional equivalence.

# Chapter 4

# Detecting Functionally Equivalent or Similar Code for Software Evolution

## 4.1 Introduction

Chapters 2 and 3 investigated syntax-level version compatibility in online code reuse and introduced an in-context tool to help developers avoid interpreter-level incompatibilities. However, even when code is syntactically compatible, maintenance and evolution tasks often require reasoning at a deeper semantic level: different implementations may exhibit the same observable behavior while differing substantially in efficiency and other non-functional properties. To address this behavioral perspective of the dissertation (D-RQ3), this chapter presents Study 3, which constructs a test-based dataset of functionally equivalent Python methods mined from open-source projects and uses it to analyze performance trade-offs and to examine the feasibility of LLM-based equivalence judgment.

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 valu-

able 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 [65] and DECKARD [35], 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. In this chapter, we operationalize functional equivalence in an observational, test-based sense: two methods are treated as functionally equivalent if they exhibit the same observable outcomes for the same inputs under a shared test suite (including manually augmented tests when necessary). This definition provides high-confidence labels for empirical analysis, while not constituting a formal proof over all possible inputs. Our approach uses Pynguin [44] 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 [49] 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.

Beyond dataset construction, we conduct two downstream analyses to demonstrate why functional equivalence matters for software evolution and to illustrate what this dataset enables. First, we study execution-time performance among FE pairs. Although FE methods are behaviorally indistinguishable under our test-suite-based notion of equivalence, they are not

necessarily equivalent in efficiency: different algorithms, data structures, and Python idioms can lead to substantial runtime gaps. By comparing FE implementations (i.e., "same functionality, different code"), we obtain an apples-to-apples setting for characterizing performance trade-offs and for identifying recurring factors that explain speedups or slowdowns.

Second, we evaluate whether large language models (LLMs) can recognize functional equivalence from source code alone. LLM-based assistants are increasingly used to support code reuse and transformation, where deciding whether two implementations preserve behavior is a recurring need. Our manually verified FE and non-FE pairs provide ground truth for this capability, allowing us to establish an initial baseline (using GPT-4o) and to analyze typical failure modes when the implementations differ substantially in structure.

Overall, these two analyses complement the dataset contribution: performance evaluation highlights practical implications of choosing among equivalent implementations, and LLM evaluation assesses the feasibility and limitations of learning-based equivalence recognition grounded in real-world Python methods.

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.

## 4.2 Background

### 4.2.1 Definition of FE Methods

Functional equivalence (FE) refers to the situation where two methods differ in implementation but exhibit the same observable behavior. In this study, we adopt a practical operationalization aligned with Section 1.3: a pair is labeled as FE if both methods pass a common set of automatically generated and (when needed) manually augmented tests, indicating observational equivalence under the tested input domain. This test-based notion enables scalable mining and reproducible evaluation, while acknowledging that untested corner cases may still exist.

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.

### 4.2.2 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 [31] utilizes mutually executed, automatically generated test cases to identify sets of FE methods. Additionally, by drawing on Borge's dataset [12], a dataset containing 276 FE method pairs was constructed. Building on similar methodologies, FEMPDataset [29] 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 [18] 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.

### 4.2.3 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 [66]. 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 [33] 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 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.

### 4.2.4 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 [50], CodeT5 [77, 78], and TypeT5 [81], which all utilize deep learning techniques and supplement them with formal methods and static analysis [52, 53]. 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.

### 4.2.5 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.

## 4.3 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.

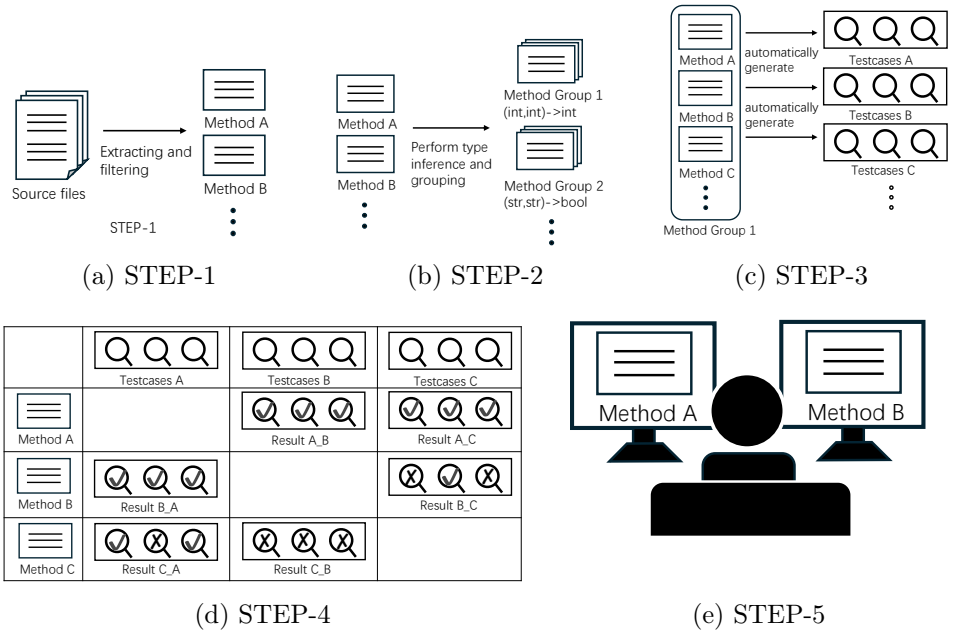
**STEP-2 Type Inference and Grouping.**
Perform type inference on each method and group them based on the inferred types.

**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 4.1 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.




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

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

Figure 4.1: Steps to obtain pairs of functionally equivalent Python methods.

### 4.3.1 STEP-1

To construct the dataset of functionally equivalent (FE) method pairs, we used the ManyTypes4Py dataset [49] 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.

**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,

```
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 4.2: Example of normalization.

- number of statements and conditional predicates,
- file path, start line, and end line.

**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 4.2 illustrates this process. Subfigure 4.2a shows the original method, whereas subfigure 4.2b 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.

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.

**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.

**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.

### 4.3.2 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 [81] 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.

```
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 4.3: Example of type inference.

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. Those 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 4.3 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.

### 4.3.3 STEP-3

In this step, we aimed to generate test cases for all the target methods. To accomplish this, we selected Pynguin [44], 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 4.4 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.

```
@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 4.4: Examples of removed test cases.

### 4.3.4 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.

### 4.3.5 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.

## 4.4 Dataset

In this section, we present the dataset constructed in this study. The dataset is built from source code in ManyTypes4Py [49], 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.

Although the final number of confirmed equivalent pairs (68) is modest, we emphasize that PyFuncEquivDataset is an initial, curated release (v1) intended as a high-confidence ground-truth benchmark, rather than a large weakly labeled corpus. This design choice is driven by strict con-

trollability and reproducibility requirements: each candidate pair must be (i) executable in isolation, (ii) testable under a common harness, and (iii) supported by automatically generated tests that satisfy our coverage constraints (100% branch coverage), with manual validation when automated tests are insufficient to rule out subtle behavioral differences. These requirements make confirmation expensive and naturally limit the yield, but they substantially improve label reliability.

Importantly, the dataset's value is not captured by the FE count alone. In addition to the 68 confirmed FE pairs, we provide 683 manually validated non-equivalent pairs as hard negatives with concrete counterexamples, enabling controlled evaluation of equivalence recognition methods (including LLM-based judgment) and supporting robustness analysis under challenging near-miss cases. Moreover, the pipeline starts from a large corpus (about 1.5 million methods) and surfaces 7,415 executable candidates, indicating that the bottleneck lies in validation under strict constraints rather than in candidate availability.

Finally, we explicitly position this dataset as a seed benchmark that will be expanded. As automated test generation, dependency handling, and environment isolation improve, the same mining and verification pipeline can be scaled to validate more candidates and to broaden coverage across projects, types, and difficulty levels. We therefore view the current release as a reliable foundation for semantics-aware evaluation, with clear and actionable paths for future growth.

**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.

**Dataset Organization.** Finally, we published our dataset on GitHub[1]. 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.

[1]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`.

Figure 4.5: Examples of FE method pairs.

**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.

**Examples of FE and Non-FE Method Pairs.** Figure 4.5 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 4.6 shows an example of two methods deemed 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`

```
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 4.6: Example of functionally non-equivalent method pairs.

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.

## 4.5 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[84]. 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[72]. For example, eliminating 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[14]. 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.

### 4.5.1 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.

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

```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 4.7: Examples of differences in generator expressions.

reliable basis for subsequent performance comparisons, optimization analysis, and result interpretation.

### 4.5.2 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.

**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:

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

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.

Table 4.1: 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%** |

**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 4.1 illustrates the distribution of time ratios across all evaluated pairs. Most pairs (72.06%) exhibit ratios ranging from 1 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.

### 4.5.3 Analysis of Performance Differences

Building on the execution-time findings, we utilized Python's performance profiling tool, *cProfile* [59], 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 4.7, 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 4.8, 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 4.9, both methods perform modular exponentiation, which computes the result of $b^e \ mod \ m$ (where $b$ is the

```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 4.8: Examples of differences in built-in function calls.

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.

## 4.6 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-4o, have exhibited remarkable performance across diverse programming tasks, including code generation [32], bug fixing [34], code summarization [67], and code explanation [15]. 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.

```
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 4.9: Examples of differences in computational details.

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.

### 4.6.1 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.

### 4.6.2 Prompt Design

This study evaluates GPT-4o's capability to recognize FE method pairs using a zero-shot prompt[37]. 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.**
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`.

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 4.10: Template of prompt.

Figure 4.10 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.

### 4.6.3 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 [79], 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 4.2: 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** |

The test results for all method pairs are shown in Table 4.2. Using the above results, the metrics were computed as follows.

$$\begin{aligned} \text{Precision} &= \frac{64}{64+107} = 0.374\,(37.4\%) \\ \text{Recall} &= \frac{64}{64+4} = 0.941\,(94.1\%) \\ \text{Accuracy} &= \frac{64+576}{68+683} = \frac{640}{751} = 0.852\,(85.2\%) \end{aligned}$$

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 4.3 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.

Table 4.3: 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** |

## 4.7 Related Work

### 4.7.1 FEMPDataset

This research draws on the work of FEMPDataset [29], 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.

### 4.7.2 SeqCoBench

SeqCoBench [47] 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 differences among implementations, making our dataset complementary to SeqCoBench's controlled, transformation-based setting.

### 4.7.3 EqBench

EqBench [8] provides a dataset of equivalent and non-equivalent program pairs designed to support and benchmark research on program equivalence. Compared with EqBench, our PyFuncEquivDataset focuses on Python method-level code mined from real GitHub projects and emphasizes a test-backed construction pipeline (automated test generation, mutual execution, and manual validation) that yields executable artifacts for reproducible evaluation. The two datasets are complementary: EqBench offers a general-purpose benchmark for equivalence tasks, while our dataset targets dynamic-language settings and additionally enables empirical analyses that require runnable tests, such as controlled runtime performance comparisons and model-based equivalence judgment.

### 4.7.4 EquiBench

EquiBench [80] frames *equivalence checking* as a benchmark to probe LLMs' semantic code reasoning capabilities. It provides a dataset of 2,400 program pairs spanning four languages (Python, C, CUDA, and x86-64), aiming for high-confidence labels via automated generation procedures grounded in program analysis and semantics-preserving transformations. The benchmark is organized into six categories, each containing balanced equivalent and non-equivalent pairs. In contrast to EquiBench's transformation-driven construction across multiple languages and settings, our dataset targets Python method-level code mined from real GitHub repositories and emphasizes test-backed validation artifacts, which further support analyses

beyond equivalence recognition (e.g., controlled performance comparison among functionally equivalent implementations).

### 4.7.5 S4Eq (proof-oriented equivalence)

Beyond benchmarks, S4Eq [39] proposes a machine-learning framework for *proving* semantic equivalence between programs using a system of semantics-preserving rewrite rules. Given two programs, S4Eq generates a sequence of rewrite-rule applications that can be checked automatically; therefore, only valid rewrite sequences are accepted and the approach is designed to yield no false positives. The framework targets a restricted setting of straight-line programs represented as ASTs and assumes a pure, single-entry single-exit form, and it also evaluates applicability by mining C functions from GitHub. This line of work is complementary to our test-based validation pipeline: while proof-oriented approaches can provide stronger guarantees under restrictive assumptions, our dataset construction is tailored to dynamic-language (Python) methods in the wild, where executable tests and manual validation provide a practical path to curating FE pairs for empirical analyses and LLM evaluation.

### 4.7.6 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 [76]. 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.

## 4.8 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.

## 4.9 Conclusion

In this chapter, 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-4o 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.

Overall, this chapter advances D-RQ3 by providing a test-backed benchmark and evidence for semantic-level analyses, which Chapter 5 integrates with Studies 1–2.

# Chapter 5

# Integrated Discussion and Implications

## 5.1 Overview

This chapter integrates the key findings from the three studies presented in this dissertation—the empirical analysis of Python version compatibility issues on Stack Overflow (**Study 1**), the development of the *PyVerDetector* tool for detecting version incompatibility (**Study 2**), and the construction of a dataset of functionally equivalent Python methods using automated test generation (**Study 3**). While each study focuses on a distinct aspect of code analysis, together they form a coherent framework for understanding and supporting software maintenance and evolution through the lenses of **code compatibility**, **code equivalence**, and **code similarity**.

The following sections discuss three major themes that emerge from these studies:

1. the common challenges that span from version compatibility to functional equivalence;
2. the scalability and extensibility of the datasets and tools developed in this dissertation;
3. and the broader implications of these findings for software maintenance and evolution.

## 5.2 Common Challenges from Version Compatibility to Functional Equivalence

The three studies conducted in this dissertation reveal that software evolution challenges manifest at multiple abstraction levels but often share common underlying causes.

At the **syntactic level**, version compatibility problems, such as the well-known incompatibilities between Python 2 and Python 3, demonstrate that even seemingly minor language-level changes (e.g., print statements vs. functions, or differences in integer division semantics) can disrupt code reuse and lead to widespread maintenance costs. The empirical analysis in Study 1 (Chapter 2) showed that more than 10% of frequently used Stack Overflow code snippets contain such version-specific issues.

At the **semantic level**, the problem of identifying *functionally equivalent* methods in Study 3 (Chapter 4) demonstrates a similar tension between *surface differences* and *behavioral consistency*. Even when syntax diverges across implementations, the underlying semantics may remain equivalent. This parallel between syntactic incompatibility and semantic equivalence underscores a continuum of challenges ranging from mechanical code breakage to deeper behavioral divergence.

Both perspectives converge on a shared insight: effective maintenance requires recognizing equivalence and difference not only at the level of syntax or API usage, but also at the behavioral level. Thus, the methods and tools presented in this dissertation collectively advance toward a unified goal—*understanding when two code fragments, versions, or functions can be considered the same for practical purposes of reuse and evolution.*

## 5.3 Scalability and Extensibility of Datasets and Tools

The second cross-cutting theme concerns the scalability and extensibility of the empirical frameworks, datasets, and tools introduced in this dissertation.

The large-scale empirical study in Study 1 required processing over one million Python code snippets, demonstrating that compatibility analysis can be conducted at web scale using structured datasets such as SOTorrent. The approach adopted in *PyVerDetector* (Study 2) further extended this capability to interactive, real-time analysis within developers' browsing environments, demonstrating how empirical insights can be operationalized into practical tooling.

Meanwhile, the dataset of functionally equivalent Python methods introduced in Study 3 provides a reusable resource for future research in code similarity, semantic clone detection, and automated repair. The modular design of the test generation pipeline allows adaptation to other programming languages, enabling broader studies on equivalence detection across ecosystems.

Collectively, these contributions establish a **scalable research-to-tool pipeline**: empirical observation → interactive tooling → dataset construction *for semantic analyses*. This pipeline not only scales with increas-

ing data volume but also generalizes across contexts, enabling future researchers to reuse both the datasets and the underlying methodology when studying software evolution phenomena beyond Python.

## 5.4 Implications for Software Maintenance and Evolution

The integrated findings of this dissertation have several important implications for the practice and automation of software maintenance and evolution.

### 5.4.1 Bridging Online Knowledge and Real-World Codebases

The version compatibility study (Study 1) underscores the gap between community knowledge (e.g., Stack Overflow) and real-world software projects. Tools such as **PyVerDetector** (Study 2) can bridge this gap by automatically verifying the compatibility of online code before reuse. This line of work contributes to improving the reliability of code examples and reducing technical debt introduced through copy-and-paste reuse.

### 5.4.2 Toward Behaviorally Aware Maintenance Tools

The dataset of functionally equivalent methods constructed in Study 3 demonstrates the feasibility of behavior-level comparison and automated test-based validation. Incorporating equivalence detection into maintenance workflows could enhance refactoring tools, duplication detection systems, and automated program repair, enabling them to reason about *semantics* rather than syntax alone.

### 5.4.3 Integrating Multi-Level Analyses for Evolutionary Understanding

The combination of version compatibility detection (Studies 1 and 2) and functional equivalence analysis (Study 3) forms a comprehensive multi-level perspective on software evolution. Future evolution-support systems could integrate these capabilities to automatically detect not only when code becomes syntactically incompatible, but also when its behavior diverges or converges across versions. Such integration moves toward **language-agnostic evolution analytics**, facilitating longitudinal studies and automated refactoring assistance.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

This dissertation set out to advance the understanding and automation of software maintenance and evolution by analyzing software artifacts through three complementary perspectives: **code compatibility**, **code equivalence**, and **code similarity**. To achieve this goal, the dissertation conducted three main studies (Study 1–3) that collectively address challenges developers face when reusing, maintaining, and evolving software systems.

- **Chapter 2 (Study 1)** presented a large-scale empirical study of Python version compatibility issues in Stack Overflow code snippets, revealing that version-specific incompatibilities remain a practical risk during online code reuse.
- **Chapter 3 (Study 2)** introduced **PyVerDetector**, a Chrome extension that detects Python version compatibility issues directly on Stack Overflow pages and provides real-time, in-context feedback to developers.
- **Chapter 4 (Study 3)** described the construction of the **PyFuncEquivDataset**, a test-backed dataset of functionally equivalent Python methods, and demonstrated how it enables controlled analyses of equivalence, performance differences, and LLM-based equivalence recognition.

#### Answers to D-RQ1–D-RQ3

To explicitly close the loop on the dissertation-level research questions, we summarize the answers to D-RQ1–D-RQ3 below.
**D-RQ1.** Study 1 shows that Python snippet reuse on Stack Overflow faces non-trivial cross-version risk: about 13% of snippets in good answers are

incompatible across Python 2 and Python 3, only about 20% are correctly identified as version-specific by tags/text, and relevant answers often appear within days (and sometimes on the release day) of new Python versions.

**D-RQ2.** Study 2 demonstrates that version-compatibility issues can be detected and communicated at reuse time: on 500 randomly sampled Stack Overflow snippets, PyVerDetector achieves consistently high accuracy (98.67%–98.82%) across Python 2.7 and Python 3.5–3.8 while providing lightweight, in-browser feedback.

**D-RQ3.** Study 3 shows that test-backed equivalence mining is feasible and useful: PyFuncEquivDataset includes 68 confirmed functionally equivalent pairs and 683 hard negatives, and GPT-4o achieves 85.2% accuracy (94.1% recall, 37.4% precision) on equivalence recognition over these manually validated pairs.

These answers provide a concise end-to-end closure of the dissertation's research questions and motivate the key contributions summarized in Section 6.2.

## 6.2 Key Contributions

The main contributions of this dissertation are summarized as follows:

1. **Empirical Characterization of Version Compatibility Issues.** A comprehensive measurement of Python 2 and Python 3 incompatibilities in real-world Stack Overflow snippets revealed the prevalence, categories, and developer responses to version-specific issues.

2. **Development of a Practical Compatibility Detection Tool.** The design and implementation of **PyVerDetector** demonstrated how empirical results can be transformed into a practical browser-based tool that aids developers in avoiding incompatible code reuse.

3. **Construction of a Functionally Equivalent Dataset.** The dissertation presented a scalable pipeline for collecting, validating, and labeling functionally equivalent Python methods through automated test generation, resulting in the **PyFuncEquivDataset** as a reusable benchmark for future research.

4. **Integration Across Abstraction Levels.** By linking syntactic compatibility analysis, automated detection, and semantic equivalence evaluation, this research establishes a multi-level analytical perspective for studying software evolution phenomena.

## 6.3 Limitations

Despite its contributions, this dissertation also has several limitations that open opportunities for further improvement:

- The empirical analysis focused solely on Python. Generalizing the findings to other languages requires additional investigation.
- The scope of **PyVerDetector** was limited to syntactic compatibility; behavioral compatibility and runtime environment differences remain unexplored.
- The dataset of functionally equivalent methods primarily targeted small- to medium-sized functions. Scaling the approach to complex modules or multi-file projects poses technical challenges.

## 6.4 Future Directions

### 6.4.1 Cross-Language and Cross-Ecosystem Compatibility Analysis

Future work could extend the methodology beyond Python to other programming languages (e.g., JavaScript, Java, or C++). Comparative studies across ecosystems may reveal whether the compatibility issues observed in Python are language-specific or indicative of more general patterns in language evolution. Moreover, integrating compatibility analysis into continuous integration pipelines could help automatically detect version mismatches across dependencies and environments.

### 6.4.2 Semantic-Aware Refactoring and Maintenance Support

The findings on functional equivalence open new possibilities for semantic-aware tools that go beyond syntax matching. Future research could develop refactoring or repair systems that leverage equivalence detection to automatically identify redundant implementations, suggest consolidation, or validate behavioral preservation after code transformations.

### 6.4.3 Automated Dataset Expansion and Benchmarking

The dataset of functionally equivalent methods can be expanded automatically by combining code mining, mutation, and dynamic validation. Establishing public benchmarks and leaderboards based on this dataset would facilitate reproducibility and fair comparison among emerging techniques in code similarity, clone detection, and automated program repair.

### 6.4.4 Integrating Multi-Level Analyses into Evolution Analytics

An important direction is to combine compatibility, equivalence, and similarity analyses into a unified framework for *software evolution analytics*. Such integration would enable automated reasoning about when code changes are purely syntactic, behaviorally equivalent, or functionally divergent, supporting intelligent recommendations for maintenance, migration, and refactoring.

# Bibliography


[1] (1998). Ieee standard for software maintenance. *IEEE Std 1219-1998*, pages 1–56.

[2] (2022). Iso/iec/ieee international standard - software engineering - software life cycle processes - maintenance. *ISO/IEC/IEEE 14764:2022(E)*, pages 1–46.

[3] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: principles, techniques, & tools*, page 242. Pearson, 2 edition.

[4] Alam, A. I., Roy, P. R., Al-Omari, F., Roy, C. K., Roy, B., and Schneider, K. A. (2023). Gptclonebench: A comprehensive benchmark of semantic clones and cross-language clones using gpt-3 model and semanticclonebench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–13.

[5] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, **3**(POPL).

[6] An, L., Mlouki, O., Khomh, F., and Antoniol, G. (2017). Stack overflow: A code laundering platform? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293.

[7] Assi, M., Hassan, S., and Zou, Y. (2025). Unraveling code clone dynamics in deep learning frameworks. *ACM Trans. Softw. Eng. Methodol.*, **34**(8).

[8] Badihi, S., Li, Y., and Rubin, J. (2021). Eqbench: A dataset of equivalent and non-equivalent program pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 610–614.

[9] Baltes, S. and Diehl, S. (2019). Usage and attribution of stack overflow code snippets in github projects. *Empirical Software Engineering*, **24**(3), 1259–1295.

[10] Baltes, S., Treude, C., and Diehl, S. (2019). Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 191–194.

[11] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377.

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

[13] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. CHI '09, page 1589–1598, New York, NY, USA. Association for Computing Machinery.

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

[15] Dvivedi, S. S., Vijay, V., Pujari, S. L. R., Lodh, S., and Kumar, D. (2024). A comparative analysis of large language models for code documentation generation.

[16] Eick, S., Graves, T., Karr, A., Marron, J., and Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, **27**(1), 1–12.

[17] Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, **2**(3), 17–23.

[18] EvoSuite Team (2021). Automatic Test Suite Generation for Java. `https://www.evosuite.org/`.

[19] Exchange, S. (2023). Stack exchange data. Accessed on 2023-08.

[20] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

[21] Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. (2017). Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136.

[22] Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122.

[23] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[24] Fraser, G. and Arcuri, A. (2013). Whole test suite generation. *IEEE Transactions on Software Engineering*, **39**(2), 276–291.

[25] Gabel, M., Jiang, L., and Su, Z. (2008). Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 321–330, New York, NY, USA. Association for Computing Machinery.

[26] Galindo-Gutierrez, G., Sandoval Alcocer, J. P., Jimenez-Fuentes, N., Bergel, A., and Fraser, G. (2025). Increasing the effectiveness of automatically generated tests by improving class observability. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1553–1565.

[27] Greco, C., Haden, T., and Damevski, K. (2018). Stackintheflow: behavior-driven recommendation system for stack overflow posts. ICSE '18, page 5–8, New York, NY, USA. Association for Computing Machinery.

[28] He, J., Xu, B., Yang, Z., Han, D., Yang, C., and Lo, D. (2022). Ptm4tag: sharpening tag recommendation of stack overflow posts with pre-trained models. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 1–11. ACM.

[29] HIGO, Y. (2024). 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.

[30] Higo, Y., Kamiya, T., Kusumoto, S., and Inoue, K. (2007). Method and Implementation for Investigating Code Clones in a Software System. *Information and Software Technology, Vol.49, issues 9-10, pp.985-998, September, 2007*.

[31] Higo, Y., Matsumoto, S., Kusumoto, S., and Yasuda, K. (2022). 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.

[32] Huang, T., Sun, Z., Jin, Z., Li, G., and Lyu, C. (2024). Knowledge-aware code generation with large language models.

[33] Index, T. (2025). Tiobe index for january 2025. [Online; accessed 2025-01-28].

[34] Islam, N. T., Karkevandi, M. B., and Najafirad, P. (2024). Code security vulnerability repair using reinforcement learning with large language models.

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

[36] Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, **28**(7), 654–670.

[37] Khajezade, M., Wu, J. J., Fard, F. H., Rodríguez-Pérez, G., and Shehata, M. S. (2024). Investigating the efficacy of large language models for code clone detection.

[38] Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112.

[39] Kommrusch, S., Monperrus, M., and Pouchet, L.-N. (2023). Self-supervised learning to prove equivalence between straight-line programs via rewrite rules. *IEEE Trans. Softw. Eng.*, **49**(7), 3771–3792.

[40] Komondoor, R. and Horwitz, S. (2001). Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, page 40–56, Berlin, Heidelberg. Springer-Verlag.

[41] Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, **68**(9), 1060–1076.

[42] Lehman, M. M. (1996). Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, page 108–124, Berlin, Heidelberg. Springer-Verlag.

[43] Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley. Accessed: 21 November 2025.

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

[45] Malloy, B. and Power, J. (2019). An empirical analysis of the transition from python 2 to python 3. *Empirical Software Engineering*, **24**.

[46] Malloy, B. A. and Power, J. F. (2017). Quantifying the transition from python 2 to 3: An empirical study of python applications. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 314–323.

[47] Maveli, N., Vergari, A., and Cohen, S. B. (2025). What can large language models capture about code functional equivalence? In L. Chiruzzo, A. Ritter, and L. Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6865–6903, Albuquerque, New Mexico. Association for Computational Linguistics.

[48] Mens, T. and Demeyer, S. (2008). *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition.

[49] Mir, A. M., Latoskinas, E., and Gousios, G. (2021). 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.

[50] Mir, A. M., Latoškinas, E., Proksch, S., and Gousios, G. (2022). Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252.

[51] Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, page 279–287, Washington, DC, USA. IEEE Computer Society Press.

[52] Peng, Y., Gao, C., Li, Z., Gao, B., Lo, D., Zhang, Q., and Lyu, M. (2022). 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.

[53] Peng, Y., Gao, S., Gao, C., Huo, Y., and Lyu, M. R. (2023). Domain knowledge matters: Improving prompts with fix templates for repairing python type errors.

[54] Peterson, B. and Cannon, B. (2009). Backwards compatibility policy. PEP 387, Python Software Foundation.

[55] PHP Group (2015). Backward incompatible changes in php 7. https://www.php.net/manual/en/migration70.incompatible.php. Accessed: 2025-12-26.

[56] PHP Group (2020). Backward incompatible changes in php 8. https://www.php.net/manual/en/migration80.incompatible.php. Accessed: 2025-12-26.

[57] Ponzanelli, L., Bacchelli, A., and Lanza, M. (2013). Seahawk: stack overflow in the ide. ICSE '13, page 1295–1298. IEEE Press.

[58] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and Lanza, M. (2014). Mining stackoverflow to turn the ide into a self-confident programming prompter. MSR 2014, page 102–111, New York, NY, USA. Association for Computing Machinery.

[59] Python Software Foundation (2024). cProfile: Profile execution time of a program. https://docs.python.org/3/library/profile.html.

[60] Python Software Foundation (2025). 2to3 — automated python 2 to 3 code translation. https://docs.python.org/uk/3.11/library/2to3.html. Python 3.11 Documentation.

[61] Ragkhitwetsagul, C., Krinke, J., Paixao, M., Bianco, G., and Oliveto, R. (2021). Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, **47**(3), 560–581.

[62] Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, **74**(7), 470–495.

[63] Ruby Core Team (2009). News for ruby 1.9.1. https://docs.ruby-lang.org/en/2.4.0/NEWS-1_9_1.html. Accessed: 2025-12-26.

[64] Ruby Core Team (2020). Ruby 3.0.0 released. https://www.ruby-lang.org/en/news/2020/12/25/ruby-3-0-0-released/. Accessed: 2025-12-26.

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

[66] Srinath, K. R. (2017). Python – the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, **4**(12), 354–357.

[67] Su, C.-Y. and McMillan, C. (2024). Distilled gpt for source code summarization.

[68] Sunsetting Python 2 (2022). Sunsetting python 2. `https://www.python.org/doc/sunset-python-2/`. Accessed: 2022-06-27.

[69] Swift.org (2016). Migrating to swift 3. `https://www.swift.org/migration-guide-swift3/`. Accessed: 2025-12-26.

[70] Toal, R., Rivera, R., Schneider, A., and Choe, E. (2016). *Programming Language Explorations*. Chapman and Hall/CRC.

[71] Valiev, M., Vasilescu, B., and Herbsleb, J. (2018a). 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. Association for Computing Machinery.

[72] Valiev, M., Vasilescu, B., and Herbsleb, J. (2018b). 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. Association for Computing Machinery.

[73] van Rossum, G. (2006). Python 3000. PEP 3000, Python Software Foundation.

[74] van Rossum, G., Galindo, P., and Nikolaou, L. (2020). New peg parser for cpython. PEP 617, Python Software Foundation.

[75] Verdi, M., Sami, A., Akhondali, J., Khomh, F., Uddin, G., and Motlagh, A. K. (2022). An empirical study of c++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering*, **48**(5), 1497–1514.

[76] Wagner, S., Abdulkhaleq, A., Bogicevic, I., Ostberg, J., and Ramadani, J. (2016). How are functionally similar code clones syntactically different? an empirical study and a benchmark. *PeerJ Computer Science*, **2**, e49.

[77] Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*.

[78] Wang, Y., Le, H., Gotmare, A. D., Bui, N. D., Li, J., and Hoi, S. C. H. (2023a). Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*.

[79] Wang, Y., Ye, Y., Wu, Y., Zhang, W., Xue, Y., and Liu, Y. (2023b). 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.

[80] Wei, A., Cao, J., Li, R., Chen, H., Zhang, Y., Wang, Z., Liu, Y., Teixeira, T. S. F. X., Yang, D., Wang, K., and Aiken, A. (2025). EquiBench: Benchmarking large language models' reasoning about program semantics via equivalence checking. In C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33856–33869, Suzhou, China. Association for Computational Linguistics.

[81] Wei, J., Durrett, G., and Dillig, I. (2023). Typet5: Seq2seq type inference using static analysis. In *The Eleventh International Conference on Learning Representations*.

[82] White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98.

[83] Wu, Y., Wang, S., Bezemer, C.-P., and Inoue, K. (2019). How do developers utilize source code from stack overflow? *Empirical Software Engineering*, **24**(2), 637–673.

[84] Zehra, F., Javed, M., Khan, D., and Pasha, M. (2020). Comparative analysis of c++ and python in terms of memory and time. *Preprints*.

[85] Zerouali, A., Velázquez-Rodríguez, C., and De Roover, C. (2021a). Identifying versions of libraries used in stack overflow code snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 341–345.

[86] Zerouali, A., Velázquez-Rodríguez, C., and De Roover, C. (2021b). Identifying versions of libraries used in stack overflow code snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 341–345.

[87] Zhang, H., Wang, S., Chen, T.-H., Zou, Y., and Hassan, A. E. (2021). An empirical study of obsolete answers on stack overflow. *IEEE Transactions on Software Engineering*, **47**(4), 850–862.

[88] Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., and Kim, M. (2018). Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, page 886–896.

[89] Zhou, J. and Walker, R. J. (2016). Api deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 266–277.

# Appendix A

# Supplementary Dataset Details for Functionally Equivalent Python Methods

## A.1 Examples of Basic Dataset Operations

The dataset (*PyFuncEquivDataset*) is provided as an SQLite database file (`PyFuncEquivDataset.db`), available on GitHub[1]. 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.1.1 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.1.2 Retrieving Detailed Information

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

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

```
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;
```

## A.2 False Positive Categorization Schema

### Functional Domain Categories

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