Automated Detection and Correction of Python Code Style Violations

An Empirical Study in Open Source Projects

Ratish Thakoersingh

Automated Detection and Correction of Python Code Style Violations

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ratish Thakoersingh
born in Amsterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Automated Detection and Correction of Python Code Style Violations

Author: Ratish Thakoersingh
Student id: 4940474

Abstract

This thesis investigates the prevalence of Pylint warnings in open-source Python projects and evaluates the effectiveness of an AI-driven tool for automatically fixing these warnings. The study also explores how developers perceive automated code suggestions and seeks to streamline consent mechanisms for research-related code changes. The primary research questions addressed are: (1) What is the prevalence of Pylint warnings across open-source Python projects? (2) How effective is the AI tool developed to fix Pylint warnings? (3) How do developers perceive the automated suggestions and (4) how can the process of proposing research-related code changes with developer consent be streamlined?

To address these questions, the research draws on literature related to static code analysis, fault detection, and the increasing use of artificial intelligence (AI) in automated code repair. Previous studies highlight the challenges developers face in maintaining consistent code quality and the role of AI in automating such tasks.

The research follows a mixed-method approach. Quantitatively, a dataset of 205 open-source Python projects was analyzed to identify and address common Pylint warnings. An AI-driven tool was employed to attempt fixing these warnings, achieving a success rate of 88%. In 60 projects, pull requests were submitted to open source maintainers to assess the effectiveness and reception of the tool. Qualitative feedback from maintainers was collected and analyzed, leading to a shift in the contribution strategy from pull requests to submitting issues first, as this was perceived as less intrusive and more manageable by developers.

The analysis revealed a high prevalence of Pylint warnings, particularly *missing-function-docstring* and *line-too-long*, across projects of all sizes. The AI-driven tool effectively fixed 88% of the warnings, resulting in 70% of the projects being fully warning-free. However, developer responses to automated pull requests were mixed, prompting the adoption of a more collaborative issue-first approach. These results suggest that AI tools can significantly improve code quality, but challenges remain to foster developer engagement and integrating such tools into established open source workflows.

The study has certain limitations, mainly the focus on Python projects, which may limit the generalizability of the findings to other languages or more complex projects. Furthermore, developer consent and participation were limited, which affected the full implementation of automated changes. Future research should focus on improving the integration of AI tools into developer workflows and expanding the scope of automated code fixes to more diverse and complex projects.

Thesis Committee:

Chair:	Prof.dr.ir. D. Spinellis PhD, Faculty EEMCS, TU Delft
Committee Member:	Dr. P. K. Murukannaiah, Faculty EEMCS, TU Delft

Preface

I would like to express my deepest gratitude to my supervisor, Dr. Diomidis Spinellis, for his unwavering support, guidance, and encouragement throughout this project. His thoughtful suggestions, willingness to engage in meaningful discussions and genuine consideration of my preferences not only helped shape this thesis, but also made the entire process more enjoyable.

I am also grateful to Richard Grimes, data steward at TU Delft, for his invaluable assistance in improving my data management plan, navigating the HREC requirements, and helping me implement a streamlined approach for obtaining developer consent.

I would also like to extend my sincere thanks to Dr. P. K. Murukannaiah for agreeing to be a Committee Member and taking the time to review my work. His participation in this process is greatly appreciated.

I would like to thank my friends and family for always being a listening ear when I needed it, showing interest in my work, and offering their constant support and belief in me throughout this journey.

Finally, I wish to thank all the developers who participated in this research. Their participation and cooperation were crucial to the success of this study.

Ratish Thakoersingh
Delft, the Netherlands
September 9, 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an overview of the background, motivation, and objectives of the research presented in this thesis. It begins by outlining the foundational aspects of Python programming and the significance of coding standards and static analysis tools like Pylint. The discussion extends to the application of artificial intelligence, particularly large language models (LLMs), in automating code generation and repair.

The chapter then transitions to a detailed exploration of the research questions driving this study. We investigate the prevalence of Pylint warnings in open source Python projects, assess the effectiveness of an automated tool designed to address these warnings, and examine developers' perceptions of such tools. This introduction sets the stage for understanding the scope of our research and its relevance to the broader field of software development.

A brief reader's guide is provided to navigate the structure of the thesis and understand the key areas of focus.

1.1 Background

Python stands out as a highly favored programming language due to its rapid learning curve, straightforward syntax, and broad applicability across diverse domains, making it an appealing choice for developers of all skill levels. [31, 29]

In line with its widespread adoption, Python has a set of best-practices and coding conventions, epitomized by Python Enhancement Proposal 8 (PEP 8). This comprehensive style guide encompasses recommendations on formatting, naming conventions, and code organization, among others. These standards serve as pillars for maintaining code quality and readability within the Python community. However, adherence to these guidelines can be challenging to enforce manually. [32]

Adherence to these practices can be facilitated with static analysis tools. One of these tools is Pylint[27], a Python-specific static analysis tool designed to detect and report code inconsistencies, deviations from coding standards, and potential errors. Leveraging a sophisticated set of rules and heuristics, Pylint scrutinizes Python codebases, offering developers actionable insights to enhance code quality and maintainability. [13]

Despite the availability of established coding conventions and tools like Pylint, the open source landscape of Python projects often showcases variations in adherence to these standards. Notable open source Python projects, such as Django, TensorFlow, and Flask, illustrate the diverse array of applications and utilities developed within the Python ecosystem. However, it is not certain whether these projects consistently conform to prescribed coding standards, presenting an opportunity for investigation into the prevalence of code style discrepancies and their impact on project quality. [25]

In recent years, artificial intelligence (AI) has transitioned from a niche field of study to a central force driving innovation across numerous sectors. AI's capabilities have grown exponentially, fueled by advancements in machine learning, data processing, and computational power. [21] Among the most groundbreaking developments in AI is the advent of large language models (LLMs). These models, which are trained on vast amounts of text data, have demonstrated remarkable proficiency in understanding and generating human language. This has positioned LLMs as versatile tools capable of addressing a wide range of tasks that were previously the domain of specialized algorithms. [14, 6, 5, 37]

One of the most impactful applications of LLMs is in the realm of code generation and repair. Modern tools like GitHub Copilot [8, 34, 23] and ChatGPT[26, 10] have revolutionized a variety of code-related tasks, including code completion, translation, and error correction. By interpreting natural language inputs, these tools can generate entire code snippets, suggest improvements, and even identify and fix bugs. This has not only streamlined the development process but also alleviated much of the routine burden on developers, allowing them to focus on more complex and creative aspects of programming. [17, 33]

1.2 Problem Statement

Although there are well-established best practices, coding conventions, and awareness of common bugs in Python development, it remains unclear whether open source developers consistently adhere to these standards. This gap in knowledge leads to our first research question:

Q1: *What is the prevalence of Pylint warnings in open source projects?*

Understanding the prevalence of Pylint warnings is vital for pinpointing common code quality issues and their distribution across various project sizes and popularities. By identifying common issues, developers can better prioritize which aspects of their codebase to improve, ultimately enhancing the overall quality and maintainability of their projects. This analysis could also highlight specific areas where existing tools and practices may need further development or reinforcement.

To ensure that our findings are representative of the broader open source Python community, we employ stratified sampling. This approach allows us to capture a diverse set of projects across different sizes and popularities, ensuring that our conclusions are not biased toward any particular type of project. Additionally it will give us data on whether the code quality differs between these selected strata.

Identifying the frequency of Pylint warnings provides insights into the most common code quality issues in open source projects. This information guides developers in prioritizing code quality improvements and ensures efficient allocation of resources. We hypothesize that larger and more popular open source projects will exhibit fewer violations per lines of code due to higher development standards and more rigorous code reviews. Furthermore, we anticipate that violations of coding conventions will be more common than those related to potential bugs, as developers may prioritize functionality over strict adherence to style guidelines.

Although static analysis tools like Pylint are widely used to identify issues in Python codebases, there is a notable gap in tools that provide automated suggestions and fixes specifically for Pylint warnings. Existing tools often focus on code linting or providing generic suggestions, but few offer automated, context-aware fixes tailored to the specific issues reported by Pylint. This limitation presents a significant opportunity for innovation.

To address this gap, we developed an automated tool designed specifically to target Pylint warnings. Our tool leverages advanced machine learning techniques and natural language processing to analyze Pylint reports and generate precise code fixes. By utilizing a sophisticated large language model, our tool aims to automatically correct identified issues while adhering to best practices and maintaining code functionality. This development is intended to streamline the process of code maintenance and enhance overall code quality, leading us to our second research question:

Q2: *How effective is the developed automated tool at fixing Pylint warnings?*

Answering this question is crucial for several reasons. First, evaluating the effectiveness of our tool provides insights into its ability to address Pylint warnings accurately and efficiently. This assessment will help determine whether the tool can reliably improve code quality and reduce the manual effort required for code maintenance. Additionally, understanding the effectiveness of the tool can guide further refinements and enhancements, making it a more valuable asset for developers.

The potential impact of answering this research question is significant. If the tool demonstrates high effectiveness, it could become a key resource for developers, enhancing productivity and code quality across Python projects. Conversely, if the tool's performance is suboptimal, it will highlight areas for improvement, contributing to the ongoing development of more advanced and reliable automated code repair solutions.

We hypothesize that our automated tool will show a decent level of effectiveness in fixing Pylint warnings, particularly in addressing common issues related to code style and potential errors. This expectation is based on the advanced capabilities of the underlying language model and the targeted nature of the tool's fixes. However, the true measure of effectiveness will be determined through rigorous testing and analysis of the performance of the tool in open source projects in the real world.

The introduction of automated tools in software development, especially those utilizing artificial intelligence, has been met with mixed reactions. Although these tools promise increased efficiency and reduced manual effort, developers often exhibit skepticism regarding their suggestions. This skepticism can stem from concerns about the accuracy, relevance, and overall quality of the fixes provided by AI-driven solutions.

In the realm of Python development, where adherence to coding standards and best practices is crucial, the perception of automated code fixers is particularly significant. Understanding how open source developers view the suggestions made by our tool is essential for evaluating its practical utility and acceptance within the developer community. To address this, we formulated our third research question:

Q3: *How do open source developers perceive the suggestions made by the developed tool?*

Answering this question will shed a light on whether developers find the tool's suggestions useful and relevant, or if they view them with skepticism or outright rejection. This understanding will help us gauge the tool's integration potential and its impact on developer workflows. It will also provide valuable feedback on how to improve the functionality of the tool and ensure that it meets the needs and expectations of its users.

The results from this inquiry could lead to more nuanced and developer-friendly approaches to automated code repair. By identifying the aspects of the tool that resonate with developers and those that do not, we can refine its suggestions, enhance user trust, and increase the likelihood of widespread adoption. Moreover, proactively addressing developer concerns can improve the overall effectiveness and usability of AI-driven tools in the software development process.

We hypothesize that, while our tool will provide valuable suggestions for fixing Pylint warnings, it may face some resistance or negative perception from developers. This skepticism could be attributed to a broader trend where developers often harbor reservations towards automated AI tools, questioning their reliability and relevance. Overcoming these perceptions will require not only demonstrating the effectiveness of the tool but also engaging with developers to address their concerns and incorporate their feedback into ongoing improvements.

As we explore the perceptions of open source developers towards the suggestions made by our automated tool, it's crucial to recognize that the process of integrating automated changes involves more than just technical and perceptual considerations. A key aspect of this process is to ensure that the integration of these changes is done ethically, with the proper consent of the project maintainers and contributors. This consideration often gets overshadowed in automated tools and research initiatives, potentially leading to conflicts and a lack of trust.

To address this concern, we need to understand how to effectively streamline the process of proposing code changes and obtaining developer consent. Proper consent is not only a matter of ethical practice but also essential for fostering a collaborative environment where automated tools are seen as valuable contributors rather than disruptive forces. Ensuring that maintainers are fully informed and able to give their explicit consent helps maintain the integrity of the open source community and enhances the likelihood of positive interactions.

This leads us to our next research question:

Q4: *How can we streamline a way of proposing research-related code changes with developer consent?*

The results of this question can highlight the importance of ethical practices in research, particularly when involving automated contributions to open source projects. A well-structured consent mechanism ensures that changes are proposed transparently and respectfully, preventing any imposition on the project's established goals and standards. This approach not only helps in building trust but also facilitates smoother collaboration between researchers and the open source community.

By developing a streamlined consent process, we aim to enhance the acceptance rate of automated code changes and create a more positive and collaborative relationship with developers. We hypothesize that such a process will reduce resistance and improve the overall effectiveness of integrating automated fixes, setting a precedent for ethical and respectful research practices in the future.

1.3 Thesis Structure

This thesis is organized into five chapters, each contributing to an exploration of the research questions and findings. Chapter 2 reviews the related work, situating this research within the broader context of existing studies and providing a foundation for understanding the significance of the contributions. Chapter 3 details the methodology, outlining the systematic approach employed to address the research questions, including the design of experiments, data collection, and analysis methods. The results of these experiments are presented in Chapter 4, offering insights into the effectiveness and implications of the approaches used. Chapter 5 provides a critical discussion of these results, evaluating the benefits and limitations of the methods and drawing conclusions based on the findings. This chapter also outlines potential directions for future research, emphasizing areas where further exploration could enhance understanding and application. Appendix A supplements the findings by providing a comprehensive list of Pylint warnings referenced in the results, offering additional clarity and context for the data presented.

Chapter 2

Related Work

This chapter reviews existing research relevant to our study, focusing on three main areas: the prevalence of code style violations, advancements in automated code fixes, and ethical considerations in automated contributions. We begin by examining the widespread occurrence of code style violations and their implications for software quality. Next, we explore recent advancements in automated code repair, particularly the role of large language models in fixing code issues. Finally, we address the ethical and practical challenges associated with integrating automated tools into development workflows. This overview sets the context for our research and highlights the significance of our contributions.

2.1 Prevalence of Code Style Violations

Understanding code style violations is essential for assessing the quality and maintainability of software across different programming environments. Research in this domain highlights the widespread occurrence of these violations, their implications for code quality, and the effectiveness of various tools in enforcing coding standards.

In collaborative online environments such as Stack Overflow (SO), where programmers frequently share and exchange code snippets, the adherence to coding standards is often compromised. Studies reveal that code snippets shared on such platforms are highly prone to style violations. For instance, an analysis of 407,097 Python code snippets from Stack Overflow found that a remarkable 93.87% contained style violations, averaging 0.7 violations per statement [2]. Expanding this analysis to multiple programming languages, another study found that over 90% of Python snippets, along with significant portions of C/C++ and JavaScript snippets, exhibited similar issues [22]. These findings suggest that the code shared online often falls short of best practices, potentially spreading suboptimal coding habits within the programming community.

The impact of these style violations extends beyond online code-sharing platforms to large-scale software projects. In a comprehensive study of 729 GitHub projects involving 17 different languages, Ray et al. [28] explored how programming language features influence software quality. Their findings indicate that language design, such as static versus dynamic typing and strong versus weak typing, does have a significant effect on software quality. However, these effects are modest when compared to process factors like project size and team dynamics. This underscores the complexity of maintaining code quality, where language-specific coding conventions interact with broader project management practices.

Moreover, the challenge of enforcing coding standards becomes even more pronounced in multi-language projects. Kochhar et al. [20] analyzed 628 GitHub projects that employed multiple programming languages and found that the use of diverse languages within a single project significantly increases defect proneness. Languages like C++, Objective-C, and Java, when used together, were particularly prone to style violations, reflecting the difficulty of maintaining consistent coding practices across different languages. This challenge is further compounded when developers frequently switch between languages, as observed by Horschig et al. [15], who found that Java and C++ programmers writing Python code often violated Python-specific conventions such as indentation and scoping. This highlights the need for targeted tools and training to help developers maintain coding standards, especially when they work with multiple languages.

To address these challenges, the integration of static analysis tools in Continuous Integration (CI) pipelines has become a common practice. Zampetti et al. [36] studied 20 Java open source projects on GitHub that use Travis CI and found that static analysis tools are primarily employed to enforce coding standards, with build breakages often related to these issues. When violations are detected, they are typically resolved quickly and documented thoroughly, indicating a proactive approach to maintaining code quality through consistent style enforcement.

These studies collectively underscore the widespread prevalence of code style violations and the importance of automated tools in addressing them. They highlight that significant proportions of code snippets, even in high-visibility platforms like Stack Overflow, often exhibit style violations, reflecting a broader challenge in maintaining coding standards across diverse programming environments. Additionally, research into static analysis tools and their integration into continuous integration pipelines reveals that while these tools are effective in identifying issues, their resolution remains a manual and often overlooked process.

Our research responds to these insights by focusing on the frequency of Pylint warnings in open source Python projects, demonstrating a clear need for tools that not only detect but also address these warnings. Given the high incidence of code style violations documented in prior studies, a tool that automatically fixes Pylint warnings offers a practical solution to enhance code quality and maintainability. By automating the correction of these prevalent warnings, our tool aims to significantly reduce the manual effort involved in code repair and improve the overall adherence to coding standards in open source projects.

2.2 Advancements in Automated Code Fixes

The field of automated code repair has seen remarkable progress, driven by innovations in machine learning, natural language processing, and static analysis. These advancements have significantly enhanced the ability to identify and fix various types of code issues, from bugs and type errors to code style violations.

A major leap in automated code repair is represented by the development of large language models (LLMs) tailored for complex semantic bugs. For instance, Berabi et al. introduced techniques that optimize LLMs for addressing security vulnerabilities by focusing their attention mechanisms on relevant parts of the code [3]. This approach has not only improved the efficiency of LLMs in generating accurate fixes but also demonstrated the potential of applying similar methodologies to a broader range of code issues. Similarly, RepairCAT has shown the potential of LLMs to automatically fix bugs in AI-generated programs by understanding and suggesting context-aware repairs[18]. Furthermore, advancements in using LLMs integrated with formal verification techniques have enabled tools to generate more reliable fixes by validating patches for correctness, as demonstrated in recent research[7]. These approaches illustrate the growing role of LLMs in not only detecting but also accurately repairing complex code faults, expanding their utility beyond simple syntactic fixes.

In addition, automated repair tools have improved their ability to handle specific types of API misuses. Studies show that API-related bugs are successfully repaired through advanced APR tools, marking a shift in the focus of these tools from syntactic to semantic repairs. These tools can detect patterns in API misuses and correct them without human intervention[19]. Such capabilities are key to handling the complex interactions between code and external libraries, further advancing the scope of automated program repair.

Another key advancement is the structured process used by modern Automated Program Repair (APR) tools, which includes fault localization, patch generation, and patch validation. These tools automate the identification of fault locations and generate context-aware patches that align with the intended functionality of the code. Moreover, the development of platforms like StandUp4NPR has made it easier to compare different neural program repair systems by standardizing the benchmarking and evaluation process across tools. This allows for a more consistent and empirical comparison of different repair methods[38, 16].

In the realm of Python, specific automated repair tools have emerged to address language-specific challenges. PyTER, developed by Oh et al., is designed to tackle type errors in Python code through a combination of dynamic and static analysis [24]. By identifying and correcting type mismatches, PyTER exemplifies the potential of automated repair tools in managing specific error types. Similarly, PyTy, another tool highlighted by Chow et al., focuses on fixing statically detectable type errors, reinforcing the trend of leveraging AI to streamline code maintenance [9].

Furthermore, recent research has delved into lint-based warnings in Python, revealing their prevalence across open source projects and the general preference among developers for cleaner code [25]. This study underscores the importance of addressing lint warnings to enhance code quality and maintainability. It also demonstrates that even simple refactorings can effectively resolve many of these warnings, highlighting the value of automated tools in this context.

Building on these advancements, our research introduces a tool specifically designed to address Pylint warnings in Python code. While previous works have focused on specific bugs and type errors, our tool aims to fill a critical gap by providing LLM based automated fixes for a broad spectrum of Pylint warnings. Given the high frequency of these warnings and their impact on code quality, our tool represents a significant step forward in automating the resolution of common code issues. By extending the capabilities of automated repair tools to include lint-based warnings, our work contributes to the ongoing evolution of code quality improvement tools and supports the broader goal of maintaining high standards in open source Python projects.

2.3 Ethical and Practical Considerations in Automated Contributions

As automated tools become increasingly integrated into software development workflows, the interaction dynamics between these tools and human maintainers become a critical area of study. Research has shown that pull requests (PRs) generated by bots often face lower acceptance rates and longer response times compared to those created by humans [35]. This discrepancy can be attributed to the lack of social engagement capabilities in bots, which are crucial in motivating maintainers to act promptly on PRs.

Our research acknowledges these challenges and seeks to improve the integration of bot-generated fixes into open source projects. By focusing on Python-specific warnings and enhancing the interaction process through ethical considerations and consent mechanisms, we aim to address the limitations observed in previous studies. The inclusion of these ethical considerations, in accordance with the Human Research Ethics Committee (HREC) guidelines of Delft University of Technology, ensures that our approach respects the autonomy of maintainers and fosters a collaborative environment that is conducive to the acceptance of automated contributions.

Chapter 3

Methodology

This section delineates the methodology employed in our study, which is structured around four primary phases: dataset creation, automated linting analysis, application of automated fixes, and evaluation of these fixes through pull requests. The methodology is designed to provide a systematic, rigorous, and reproducible approach to addressing our research questions, drawing from established best practices in software engineering research while incorporating innovative techniques tailored to our study's objectives.

Our approach integrates robust data collection methods, advanced analysis tools, and ethical considerations to ensure comprehensive, reliable, and replicable results. By detailing each phase of the methodology, we aim to provide transparency and enable future researchers to reproduce our findings. The following subsections outline the specific processes and techniques used in each phase, highlighting their relevance and application in the context of our research.

3.1 Dataset Creation

To construct a representative dataset for our study, we developed a custom script to gather open source Python projects from GitHub. This script employs a methodical approach that combines *random stratified sampling* with *activity checks* to ensure a diverse and relevant selection of projects.

The stratified sampling method involves dividing the project population into strata based on key metrics, specifically stars and forks, using a logarithmic distribution. This approach facilitates the inclusion of a wide range of projects, from small-scale to large-scale repositories, while mitigating bias toward highly engaged projects.

For each stratum, we maintain a corresponding CSV file containing the list of sampled projects. CSV files were chosen for their readability, ease of alteration, and compatibility with Python, allowing for streamlined data processing and future modifications. The use of CSV ensures that the dataset can be easily reviewed and reproduced.

The table below summarizes the number of projects sampled per stratum:

Stratum	Forks	Stars
11–100	20	20
101–1000	20	20
1001–10000	20	20
10001–100000	0	20
100001–1000000	0	9

Table 3.1: Number of Projects Sampled in Each Stratum for Forks and Stars

For both forks and stars, we set a maximum of 20 projects per stratum, which was achieved except for the two cases where fork stratum 4 and 5 yielded no projects, and the stars stratum 5 yielded only 9 projects. This cap was imposed due to GitHub API limits and hardware constraints, ensuring a manageable and balanced dataset for the analysis.

To further ensure the relevance of the dataset, our script performs activity checks to verify that the selected projects are actively maintained. We assess recent activity by confirming that projects have had pull requests merged within the past month. Additionally, we check whether there has been issue activity in the past month. This ensures that the dataset reflects currently maintained, active projects, which are more likely to respond to pull requests and engage with issues during the subsequent phases of our research.

This methodology draws on established techniques for data sampling and collection in software engineering research, with specific adaptations to suit the needs of our study. The random stratified sampling is based on the method used in the paper *Broken Windows: Exploring the Applicability of a Controversial Theory on Code Quality*[30], where they employ stratified sampling to obtain a random representative sample of open source Java and C repositories. By combining stratified sampling with activity checks, we aim to create a balanced and representative dataset that enhances the reliability and applicability of our subsequent analyses.

3.2 Project Setup

In this section, we outline the methodology for preparing and setting up Python projects for Pylint analysis, ensuring that our approach is both efficient and systematic. The setup process is designed to address common issues related to dependency management and environment isolation, which are crucial for accurate and reliable code analysis.

The first step involves creating a virtual environment for each project. This isolation is critical as it prevents conflicts between project-specific dependencies and global packages. By using virtual environments, we ensure that the analysis reflects the project's true dependencies without interference from other Python projects or system-wide packages. This practice follows standard guidelines in software development, where isolated environments are used to maintain consistency and avoid dependency-related issues [12].

Following the establishment of the virtual environment, we focus on dependency management. We begin by scanning the project directory for standard dependency files such as 'pyproject.toml' or 'requirements.txt'. These files specify the libraries and versions that the project relies on, and their presence simplifies the setup process. If these files are found, we use ‘pip‘ to install the listed dependencies, ensuring that the environment mirrors the project’s requirements accurately.

In cases where dependency files are missing, we utilize ‘pipreqs‘, a tool that generates a ‘requirements.txt‘ file by analyzing the import statements in the project code. This approach is inspired by best practices in dynamic dependency resolution [4], allowing us to capture all necessary libraries, including those not explicitly documented. This step ensures that our analysis environment is comprehensive and accurately reflects the project’s needs.

Additionally, the lines of code (LOC) are calculated to serve as a baseline for subsequent metrics such as warning density. Only Python files that are external to the virtual environment are considered in this calculation.

Each step of the process is documented in a CSV file, with fields capturing critical project and setup information. The fields include: *project* (project ID), *cloned* (status on whether the project has been cloned), *setup* (status of virtual environment setup), *pylint* (status of Pylint report generation for subsequent analysis), *tests* (number of tests found and how many are passed), *size* (lines of code), and metadata fields *description* and *date*.

By adhering to these practices, we mitigate common issues related to environment configuration and dependency management, which can significantly impact the reliability of code quality analysis.

3.3 Code Quality Analysis

With the project setup completed, we proceed to the code quality analysis phase using Pylint, a widely adopted static code analysis tool for Python. This phase is essential for generating insight into the quality of the code across various projects, identifying issues related to code style, potential bugs, and overall maintainability.

The process begins by running Pylint within the prepared virtual environment. Pylint’s static analysis capabilities allow us to produce detailed reports that highlight a range of issues, from stylistic inconsistencies to potential errors. This comprehensive analysis provides valuable information on the code quality of each project, aligning with the objectives of our research.

To effectively utilize Pylint’s findings, we converted the generated reports into a format that aligns with our research requirements. This conversion process is crucial for integrating the data with our analytical framework, particularly for answering questions related to the prevalence of code quality issues and the effectiveness of automated fixes. By structuring the reports to fit our analysis needs, we ensure that the subsequent evaluation steps are coherent and focused.

This structured approach to code quality analysis not only enhances the reliability of our findings but also ensures that our research is grounded in a thorough and methodical examination of code quality. By leveraging Pylint’s capabilities in a systematic manner, we aim to provide insightful and actionable results that contribute to the broader understanding of code quality in open source Python projects.

3.4 Warning Fixes with Anthropic's Claude AI

Following the Pylint analysis, we address the identified warnings using Anthropic's Claude AI, specifically the Claude 3 Haiku model. Claude 3 Haiku is a state-of-the-art large language model (LLM) designed for advanced natural language processing (NLP) tasks, with a focus on understanding and generating human-like text, including code snippets and technical fixes. Developed by Anthropic, a company dedicated to building safe and steerable AI, Claude excels in tasks requiring deep contextual understanding and reasoning. Its capabilities make it highly effective for code-related tasks, such as analyzing Pylint warnings and proposing contextually appropriate fixes [1].

Claude 3 Haiku represents the cost-effective tier of Anthropic's AI offerings, optimized for projects that require intelligent problem-solving without the higher computational expenses of more advanced models. Anthropic also provides more powerful, yet costly models that offer enhanced reasoning and language comprehension abilities. While these higher-end models might provide more sophisticated code repair capabilities, the choice of Claude 3 Haiku for this research balances performance with budgetary constraints. Despite being the less expensive option, Claude 3 Haiku offers reliable accuracy in code analysis and correction, making it a suitable choice for this project. Its ability to generate precise, minimal code fixes without introducing unnecessary changes ensures that it adheres to best practices while maintaining the overall integrity of the code.

Our approach involves guiding Claude 3 Haiku with a structured prompt to ensure precise and effective modifications. The prompt is meticulously designed to achieve high-quality code fixes. It reads as follows:

"Your task is to analyze the provided Pylint warning and Python code snippet and provide a corrected version of the code that resolves the warning. If a function or package is getting called, assume that it already exists and is imported. You can also assume that the code is syntactically correct and the only issue is the Pylint warning. The corrected code should be functional in a bigger context, efficient, and adhere to best practices in Python programming. Keep in mind that it is a snippet from a bigger script, therefore keep things such as indentation level and ending commas such that it still works in the broader context and don't add imports, packages, new classes or new functions. Only return the corrected code snippet as markdown python code and no explanation or text. If you are unable to resolve the warning, return the original code. Do not remove comments or docstrings, update them accordingly."

This structured prompt ensures that Claude 3 Haiku focuses on providing fixes that are not only correct but also contextually appropriate. The clear instructions regarding the preservation of code context, indentation, and existing comments are intended to avoid disruptions in the broader script and ensure that the generated fixes are practical and directly applicable. This methodology is inspired by similar practices in automated code repair, where contextual understanding and minimal alterations are emphasized to maintain code integrity.

Each Pylint warning is processed by providing Claude 3 Haiku with a combined input of the warning message and the relevant code snippet, formatted as: `"{the Pylint warning name} \n the code"`. This format allows the model to understand both the specific issue reported and the surrounding context, facilitating more accurate and relevant fixes.

After generating the proposed fixes, we take several steps to ensure that the corrected code meets the required standards. First, we format the code using Black, a widely adopted code formatter for Python, to ensure consistency and adherence to formatting best practices [11]. Following the formatting, we rerun Pylint to verify that the specific warning has been successfully resolved and to check that no new issues have been introduced.

In addition to Pylint verification, we also conduct functional testing by running the project's existing test suite. This step is crucial to confirm that the fixes do not inadvertently introduce new bugs or disrupt any previously functioning code. We specifically check if any of the tests that passed before the fix now fail, ensuring that the code's operational integrity remains intact. This multifaceted verification process, combining code formatting, static analysis, and functional testing, confirms the effectiveness of the automated fixes and ensures that the overall quality of the codebase is maintained.

The results of this process are documented in a CSV file with the fields: *project*, *issue*, *fixed*, *fixes*, *description*, and *date*. Each project and corresponding Pylint warning generates an entry in this file, detailing the number of warnings that were successfully resolved for each project. This allows us to systematically track the tool's effectiveness in addressing the identified issues across various projects.

Additionally, to capture instances where the tool was unable to provide a fix, a separate CSV file is maintained. This file contains the fields: *project*, *issue*, *file_name*, *line_nr*, *stage*, *description*, and *date*. This documentation allows for detailed analysis of failures, isolating the specific warning, its location, and the stage at which the process failed, whether during API interaction, formatting, Pylint validation, or due to test failures introduced by the fix. By documenting these failures comprehensively, we aim to refine the tool's approach and better understand the limitations of automated fixes.

By leveraging Claude 3 Haiku's capabilities and following a structured approach to generating and validating fixes, we aim to enhance the quality of the code in a systematic and reliable manner.

3.5 Automated Pull Requests and Streamlined Consent Mechanism

After generating code fixes using Anthropic's Claude AI, we implement a comprehensive process for validating and integrating these changes into open source repositories. This approach is designed to align with both technical best practices and ethical standards, particularly but not limited to those outlined in the Human Research Ethics Committee (HREC) guidelines of Delft University of Technology. The methodology we employ ensures that the proposed changes are not only technically sound but also introduced in a manner that respects the autonomy and consent of open source project maintainers.

3.5.1 Manual Validation of AI-Generated Fixes

The first step in our process involves manually validating the AI-generated fixes. Although the use of advanced AI models like Claude 3 Haiku often results in high-quality code suggestions, it is crucial to manually review these changes to ensure their appropriateness within the context of the target project. This manual review is essential to avoid introducing suboptimal or incorrect changes that could frustrate project maintainers or reduce the likelihood of pull request (PR) acceptance. Without this step, the risk of submitting poorly considered changes increases, potentially damaging the reputation of the research project and the relationships with the open source community.

This manual validation process also ensures that the proposed changes adhere to the specific coding style and standards of each project, which is critical to maintaining consistency and reducing friction during the PR review process. This step is inspired by best practices in software engineering, where human oversight is often necessary to complement automated processes and ensure high-quality outcomes.

3.5.2 Issue Creation and Informed Consent

Following manual validation, we create an issue in the target repository for each Pylint warning that has been addressed. This step serves as the initial point of contact with the project's maintainers. Each issue includes a detailed description of the specific Pylint warning, an explanation of our research project, and contact information for the researchers involved. A link to the PyWarnFixer project page on GitHub is also provided, where maintainers can find comprehensive information about the study's objectives, methodologies, and ethical considerations.

The creation of an issue before submitting a PR is inspired by best practices in open source collaboration, where maintainers often prefer to discuss proposed changes before they are formally submitted. This approach helps build trust and transparency, ensuring that maintainers are fully aware of the nature of the proposed changes and the research context behind them. Skipping this step could result in misunderstandings, leading to PR rejections or negative reactions from the maintainers, which would undermine the research's goals and ethical standard.

3.5.3 Automated Pull Request Submission

If the maintainers express interest in the proposed changes, we proceed with the automated creation of PRs. For each fix, a new branch is created in a forked version of the repository, and a PR is submitted to the original project. Each PR includes a clear and concise description of the specific contributions made, along with an explicit mention that the PR is part of the "Automated Detection and Correction of Python Code Style Violations" research project conducted by TU Delft.

To further ensure transparency and support the consent process, the PR includes a link to the research project's information page on GitHub. This page provides detailed information on the study, including its objectives, methodologies, and ethical considerations. By including this information directly in the PR, we make it easy for maintainers to understand the broader context of the proposed changes and to make an informed decision about whether to accept the PR.

This approach is informed by guidelines from both open source communities and academic research ethics, emphasizing the importance of clear communication and consent when interacting with contributors. Without such a mechanism, there is a risk that maintainers might feel pressured or misled, which could lead to negative outcomes such as PR rejections, community backlash, or ethical violations.

3.5.4 Monitoring and Feedback Integration

After the PRs are submitted, we actively monitor them for feedback or comments from project maintainers and contributors. This monitoring allows us to respond promptly to any queries or concerns and to make adjustments to the proposed changes if requested. The acceptance rate of these PRs serves as a key metric for evaluating the developers' reception of the automated fixes. By calculating the acceptance rate for each project, we gain insights into how well the proposed changes are received and the overall effectiveness of the automated fixing process.

The results of the pull requests are stored in a CSV file with the fields: *project*, *warning_name*, *created_date*, *status*, *pull_request_link*, and *description*. The *project* field records the identifier of the repository, while the *warning_name* specifies the particular Pylint warning that prompted the fix. The *created_date* logs when the pull request (PR) was submitted, ensuring a clear timeline for tracking purposes. The *status* field indicates whether the PR is open, closed, or merged, allowing us to monitor the outcome of each submission. The *pull_request_link* provides a direct reference to the PR within the repository, and the *description* field stores relevant notes on the fix or any additional context needed to understand the changes made. This systematic documentation allows us to evaluate the success of the PRs and assess maintainer engagement with our automated fixes.

Similarly, the results of the issues raised in open source repositories are stored in a CSV file with the fields: *project*, *warning_name*, *created_date*, *status*, *issue_link*, and *description*. Here, the *project* field again refers to the repository in question, and *warning_name* details the specific Pylint warning. The *created_date* logs when the issue was opened, while the *status* tracks whether the issue remains open or has been closed. The *issue_link* provides a direct URL to the issue for easy reference, and the *description* field allows for additional details about the issue or the interaction with maintainers. By documenting this information, we maintain a clear record of engagement with the open source community, ensuring transparency and accountability throughout the research process.

Regular monitoring and feedback integration are crucial for ensuring the success of the PRs and for maintaining positive relationships with the open source community. Ignoring this step could result in unresolved issues or unanswered questions, leading to frustration among maintainers and potentially lower acceptance rates for the PRs.

Through this streamlined and ethically sound approach, we aim to facilitate a collaborative process for integrating automated fixes into open source projects. This methodology not only enhances the quality of the codebases but also ensures that maintainers are fully informed and respected throughout the process.

3.6 Handling of Personally Identifiable Information

This research carefully adheres to ethical standards regarding the handling and storage of Personally Identifiable Information (PII). All data collected in the course of this research is securely stored on an encrypted disk to ensure that sensitive information is protected from unauthorized access. The primary data being stored consists of the names of the open-source projects analyzed during the study.

Additionally, when developers have provided explicit consent, their GitHub usernames and any data they have agreed to share, such as comments and reviews on pull requests and issues, are also stored. This data is strictly limited to what is relevant to the research, and its purpose is thoroughly communicated to all participants involved. Before any information is collected, participants are made aware of the specific data being stored, how it will be used, and their rights concerning the data.

If any individual wishes to have their data removed from the study, they are provided with the necessary contact information and clear instructions on how to request data deletion. This process ensures that participants maintain control over their personal information, in accordance with ethical guidelines and data protection regulations.

By employing these measures, this research ensures transparency and maintains the privacy of all individuals involved while only storing the minimum amount of PII necessary for the study's objectives.

Chapter 4

Results

This chapter presents the findings from the experiments conducted to address the research questions posed in this study. The data collected from open source Python projects is analyzed to provide insights into the prevalence of Pylint warnings, the effectiveness of the developed automated tool in fixing these warnings, and how open source developers perceive the tool's suggestions. Additionally, the chapter explores how developer consent can be streamlined when proposing research-related code changes. Each section is structured around the individual research questions and is supported by graphical and statistical evidence.

4.1 What is the prevalence of Pylint warnings in open source projects?

This section presents the findings related to the first research question: *What is the prevalence of Pylint warnings in open source Python projects?* The results are analyzed across different project strata based on the number of stars and forks, providing insights into how Pylint warnings vary in frequency and consistency across a diverse range of projects.

For each stratum, the top 15 Pylint warnings are identified, and their warning densities (warnings per 1000 LOC) and presence ratios (how often they appear in projects) are visualized. These metrics help to understand both the pervasiveness and distribution of the warnings in open source projects, highlighting common code quality issues across the dataset. The table A.1 contains a list of all pylint warnings mentioned in this section, along with their warning type and a link to their documentation.

4.1.1 Prevalence of Pylint warnings across star ranges

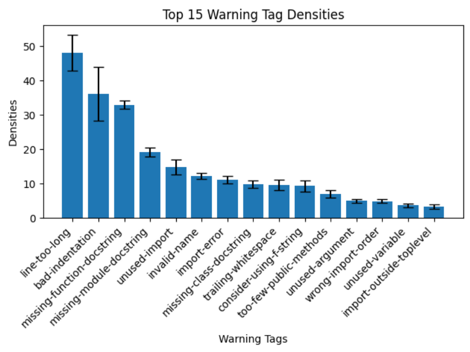
This section explores the warning densities and presence ratios of Pylint warnings across different project star ranges, from 11 stars to over 1 million. By aggregating the data, we can observe trends in how these warnings manifest in projects of varying popularity and complexity.

In the lower star ranges (11–100 and 101–1000 stars), common warnings such as *line-too-long*, *missing-function-docstring*, and *bad-indentation* dominate in terms of both warning density and presence ratios. For example, in the 11–100 star range, *line-too-long* exhibited the highest density, closely followed by *missing-function-docstring* (Figure 4.1a). These warnings appeared in almost all projects, with presence ratios nearing 100% for *missing-function-docstring* and *line-too-long*, indicating that even smaller projects struggle with basic code style adherence (Figure 4.1b). As the star range increases to 101–1000 stars, the density of *line-too-long* grows significantly, reaching up to 35 warnings per 1000 lines of code (LOC), while its presence ratio remains high across nearly all projects (Figure 4.1c and 4.1d). This suggests that larger projects experience more pervasive style issues, but these issues are consistently present across the majority of repositories.

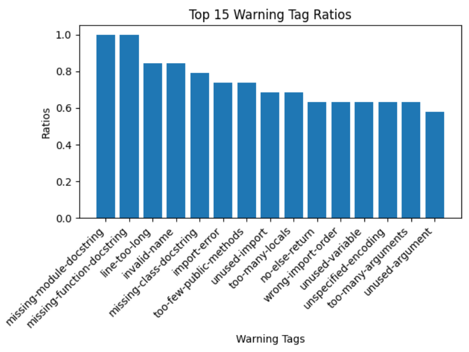
Moving to the 1001–10,000 star range, a shift is observed. While *missing-function-docstring* and *line-too-long* remain prevalent, the density of *bad-indentation* exceeds 70 warnings per 1000 LOC, becoming the most frequent issue in this range (Figure 4.1e). Interestingly, the presence ratios for *bad-indentation*, as well as other common warnings like *missing-function-docstring*, remain high, hovering around 90% (Figure 4.1f). This pattern suggests that as projects become larger and more complex, structural issues such as indentation become more frequent, while the presence of these warnings remains consistently high across most projects.

In the higher star ranges (10,001–100,000 stars and 100,001–1,000,000 stars), a different trend emerges. The density of the top warnings, such as *missing-function-docstring* and *line-too-long*, generally decreases compared to the lower star ranges, indicating that larger and more popular projects may have addressed these basic issues to some extent (Figure 4.1g, 4.1i). However, the presence ratios remain consistently high across these projects, with many warnings appearing in nearly all repositories (Figure 4.1h, 4.1j). For instance, in the 10,001–100,000 star range, the top warnings are present in 100% of the projects, though their densities are reduced to around 22 warnings per 1000 LOC. This suggests that while larger projects may have developed mechanisms to address these frequent warnings, such issues still persist across most codebases. In the highest star range (100,001–1,000,000 stars), the density of *missing-function-docstring* increases again to 35 warnings per 1000 LOC, reflecting the increased complexity and scale of these large projects (Figure 4.1i, 4.1j).

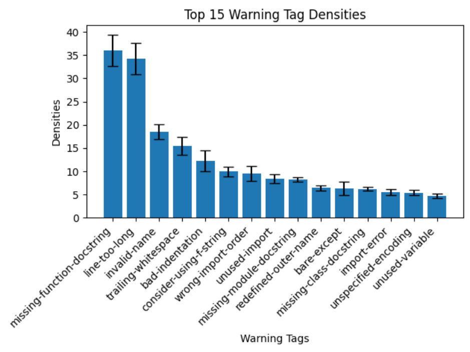
Overall, the data reveal that while warning densities tend to decrease in larger, more popular projects, the presence ratios remain high. This suggests that even as projects scale and become more mature, they continue to encounter frequent issues with basic code quality and style adherence across a wide array of warnings.



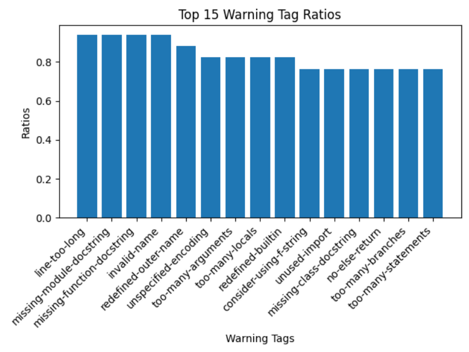
(a) 11–100 stars (Densities)



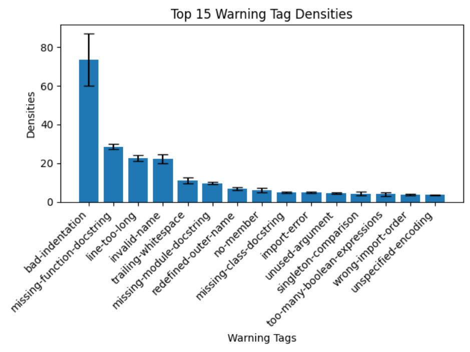
(b) 11–100 stars (Ratios)



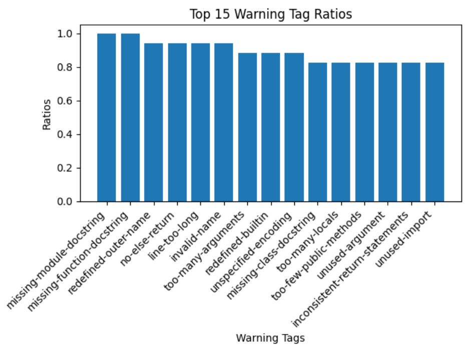
(c) 101–1000 stars (Densities)



(d) 101–1000 stars (Ratios)

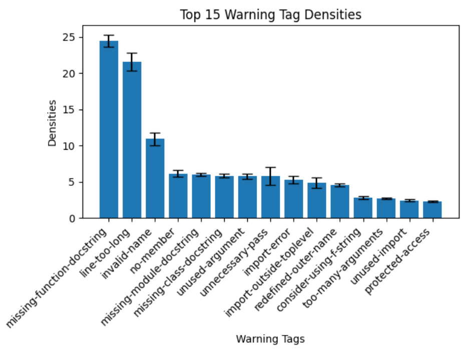


(e) 1001–10000 stars (Densities)

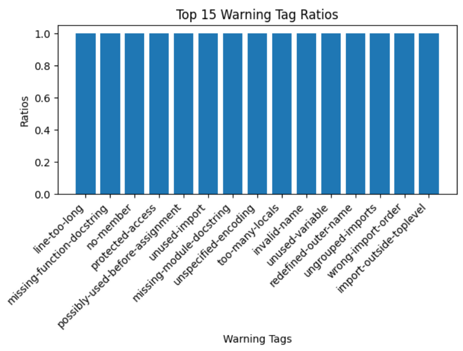


(f) 1001–10000 stars (Ratios)

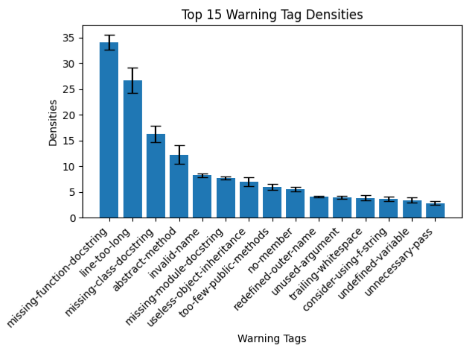
Figure 4.1: Warning Densities and Ratios Across All Star Ranges (1/2)



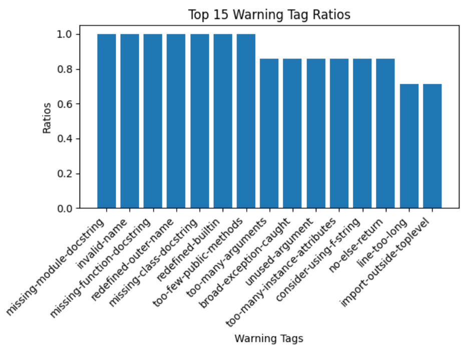
(g) 10001–100000 stars (Densities)



(h) 10001–100000 stars (Ratios)



(i) 100001–1000000 stars (Densities)



(j) 100001–1000000 stars (Ratios)

Figure 4.1: Warning Densities and Ratios Across All Star Ranges (2/2)

4.1.2 Prevalence of Pylint warnings across fork ranges

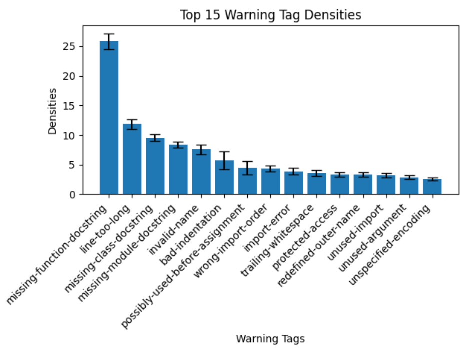
This section explores the warning densities and presence ratios of Pylint warnings across different project fork ranges, from 11 to 10,000 forks. By aggregating the data, we can observe how these warnings manifest in projects of varying complexity and contributions.

In the lower fork range (11–100 forks), the warning *missing-function-docstring* exhibits the highest density by a significant margin, reaching over 25 warnings per 1000 lines of code (LOC), as shown in Figure 4.2a. Other warnings, such as *line-too-long* and *missing-class-docstring*, appear at much lower densities, falling below 10 warnings per 1000 LOC. The presence ratios show that *missing-function-docstring*, *missing-module-docstring*, and *line-too-long* are nearly universal, appearing in almost all projects, with ratios approaching 1.0 (Figure 4.2b).

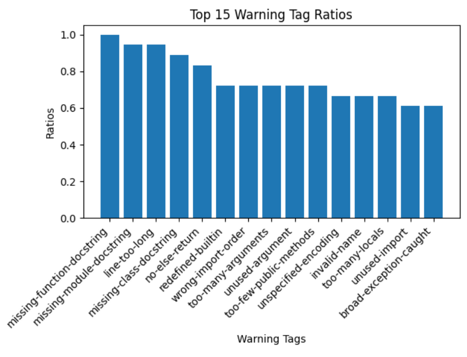
As we move to the 101–1000 forks range, there is a substantial increase in warning density for *line-too-long* and *bad-indentation*, both reaching nearly 60 warnings per 1000 LOC, representing a sharp rise compared to the previous range (Figure 4.2c). Other warnings, such as *consider-using-f-string* and *invalid-name*, also show higher densities in this fork range. The presence ratios remain consistent for warnings such as *missing-function-docstring* and *line-too-long*, but the overall ratios are slightly higher than in the lower range, indicating more frequent occurrences across projects (Figure 4.2d).

In the highest fork range (1001–10,000 forks), the density of *line-too-long* and *bad-indentation* decreases compared to the previous range, although *missing-function-docstring* continues to exhibit the highest density at just under 30 warnings per 1000 LOC (Figure 4.2e). The presence ratios in this range are slightly higher than in the previous ranges, with warnings such as *missing-module-docstring*, *missing-function-docstring*, and *invalid-name* continuing to appear in nearly all projects (Figure 4.2f). However, the ratios for certain warnings, like *too-many-statements* and *import-outside-toplevel*, decrease, indicating that these warnings become less consistent in larger projects.

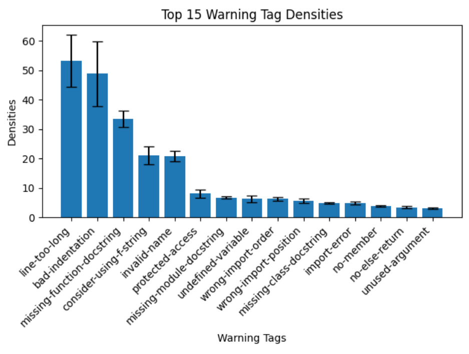
Overall, the data reveal that as the number of forks increases, the density of certain warnings fluctuates, but the presence ratios for common warnings like *missing-function-docstring* and *line-too-long* remain high across all projects, suggesting that these issues are widespread regardless of project size or complexity.



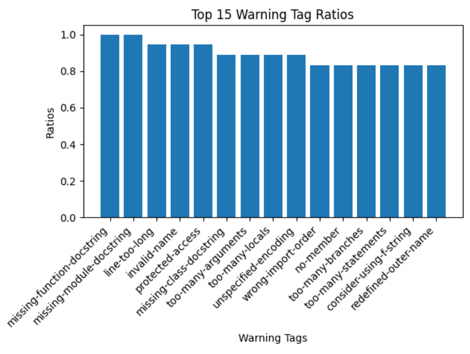
(a) 11–100 forks (Densities)



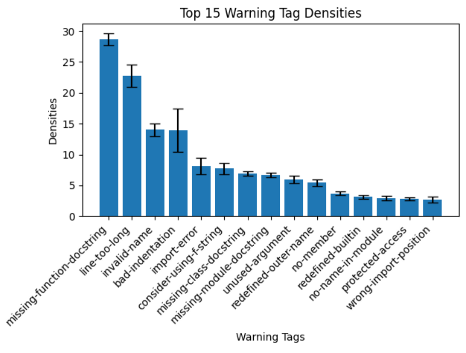
(b) 11–100 forks (Ratios)



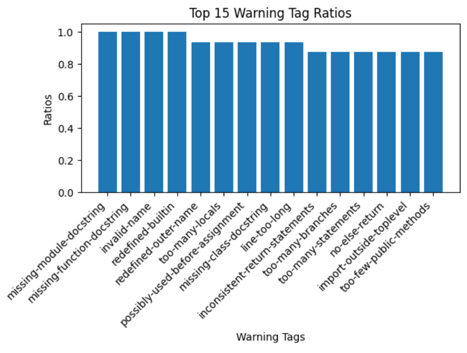
(c) 101–1000 forks (Densities)



(d) 101–1000 forks (Ratios)



(e) 1001–10000 forks (Densities)



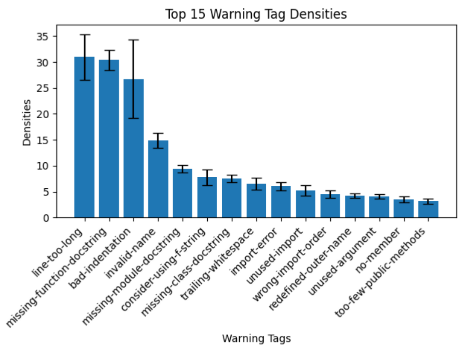
(f) 1001–10000 forks (Ratios)

Figure 4.2: Warning Densities and Ratios Across All Fork Ranges

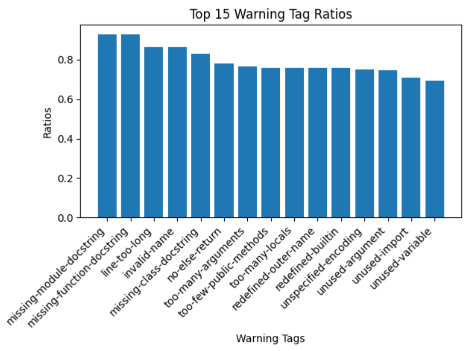
4.1.3 All projects

For all projects, the warnings *line-too-long* and *missing-function-docstring* have the highest densities, with approximately 30 warnings per 1000 LOC, as illustrated in Figure 4.3a. These warnings consistently appear at higher densities than others, such as *bad-indentation* and *invalid-name*, which still exhibit notable densities, but at lower levels.

Figure 4.3b shows that while *line-too-long* and *missing-function-docstring* have the highest densities, they do not always have the highest presence ratios. Warnings such as *missing-module-docstring* and *invalid-name* appear in a larger proportion of projects, despite their lower densities. This indicates that these warnings are more consistently present across the board, whereas higher-density warnings like *line-too-long* are more concentrated within specific projects.



(a) Number of warnings per 1000 LOC



(b) Presence ratio of warnings in projects

Figure 4.3: Warning densities and ratios for all projects

The graph (Figure 4.4) further reinforces this observation. It highlights the disparity between density and ratio for some warnings. For instance, *line-too-long* has one of the highest densities but does not achieve the same level of presence across projects, suggesting that it may be a more localized issue within certain repositories. Conversely, warnings such as *missing-module-docstring* demonstrate a higher ratio but lower density, indicating that while these warnings are found in more projects, they occur less frequently per 1000 lines of code.

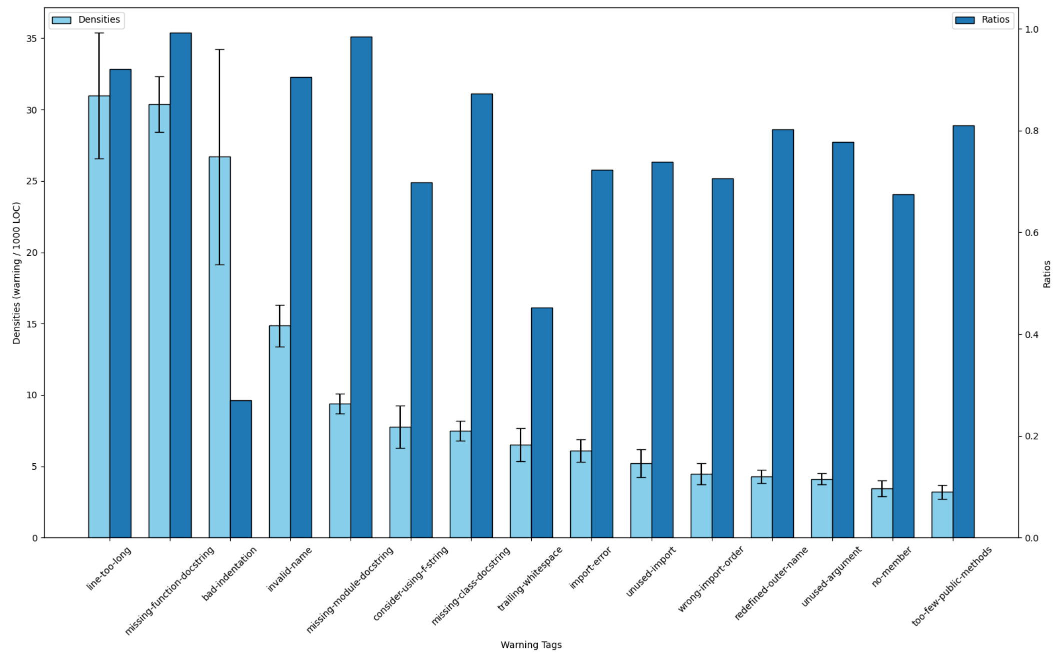


Figure 4.4: Warning densities with their ratios for all projects

4.2 How effective is the developed automated tool at fixing Pylint warnings?

The effectiveness of the automated tool developed to address Pylint warnings was evaluated across a total of 205 open source Python projects. Among these, the tool successfully resolved all detected Pylint warnings in 144 projects, producing a complete project fix rate of 70%.

Among the 205 projects, a total of 5188 Pylint warnings were identified. The automated tool was able to correct 4549 of these warnings, resulting in an overall fix rate of 88% for individual warnings.

However, the remaining 12% of warnings could not be resolved due to various technical challenges encountered during the execution of the tool's workflow. These challenges were predominantly observed in the stages that involved the extraction of relevant code snippets, communication with the Anthropic Claude AI API, and reinsertion of the generated fixes back into the code. Specifically, in some instances, the AI did not return a recognized or complete code block, or the tool was unable to locate the full context of the code associated with the Pylint warning.

4.3 How do open source developers perceive the suggestions made by the developed tool?

A total of 60 pull requests were submitted to open source projects as part of the tool's automated suggestion mechanism. Of these, 3 pull requests were fully approved and merged into the respective projects. An additional three pull requests received positive feedback from the project maintainers, though they were not merged for unspecified reasons.

A total of 7 pull requests were closed by the project maintainers. In 3 of these cases, the reviewers explicitly indicated that they did not appreciate automated pull requests. The remaining pull requests did not receive any interaction from the project maintainers or contributors.

The detailed outcomes of these pull requests, including the links to them, can be found in Appendix B.

4.4 How can we streamline a way of proposing research-related code changes with developer consent?

Initially, the approach taken in this project for proposing research-related code changes involved submitting pull requests directly to the relevant open source repositories. This method was intended to initiate a dialogue with the maintainers, with the pull requests serving as the first point of contact. Using this strategy the response rate was relatively low. A total of four contributors or reviewers from the 60 projects provided their explicit consent for the proposed changes. By the end of the project, two of these contributors had withdrawn their consent, leaving two active consents.

After receiving some negative feedback from maintainers, it became evident that this direct approach could be perceived as intrusive, particularly in cases where maintainers were not expecting automated contributions or had specific workflows in place for managing external input.

One of the key concerns raised was that many projects prefer pull requests to address issues that have already been logged in their issue boards. This allows maintainers to track and prioritize changes systematically. Directly submitting pull requests without prior discussion can be seen as bypassing this established process, making it harder for maintainers to integrate the changes into their workflows. Moreover, pull requests demand immediate attention and review, potentially adding more work for maintainers who may not be prepared to assess the proposed changes, especially if they are generated by an automated tool.

In response to this feedback, the process was adjusted to be more in line with typical open source contribution practices. Instead of submitting pull requests as the initial point of contact, the modified approach involved creating issues on the project's issue board to inquire whether the maintainers were interested in having the warnings addressed by the AI-supported tool. This shift allowed the maintainers to evaluate the proposed changes at a conceptual level before committing to a full code review, thereby reducing the pressure and workload associated with immediate pull request reviews.

The rationale behind this change was twofold. First, submitting issues rather than pull requests aligned the contribution process with the established norms of many open source projects, where maintainers often expect changes to follow a defined issue-resolution pipeline. By logging issues first, maintainers were given the opportunity to provide input on whether they wanted the proposed fixes and were able to review the context of the warning without the burden of reviewing code at the outset. Second, issues were seen as less confronting than pull requests, offering a lower-stakes way to gauge interest in automated fixes. With issues, maintainers could choose whether and when to engage with the proposed changes, without feeling obligated to immediately approve or reject a pull request.

By making this adjustment, the project aimed to respect the autonomy of open source maintainers and streamline the process of obtaining consent for research-related code changes. This revised process also contributed to better alignment with project workflows, increasing the likelihood of positive engagement from maintainers.

The evolution of this approach reflects the challenges inherent in balancing the need for research contributions with the expectations of the open source community. The changes implemented demonstrate an effort to be more collaborative and less disruptive, with the hope that future contributions will benefit from this refined process.

Chapter 5

Conclusions and Future Work

This chapter provides a comprehensive overview of the research findings and their implications. First, we reflect on the key results in the *Discussion* section, analyzing each of the research questions in detail. We then explore possible extensions to the research in the *Future Work* section, discussing areas that would benefit from further investigation. Finally, the *Conclusion* section summarizes the overall contributions of the project and offers closing thoughts on the significance of the work.

5.1 Discussion/Reflection

In this section, we examine the outcomes of each research question, reflecting on the results presented in the previous chapters. Each question is addressed in detail, providing a comprehensive discussion of the findings and their broader implications for Python code quality, tool effectiveness, and developer interaction.

5.1.1 What is the prevalence of Pylint warnings in open source projects?

The first research question sought to determine the prevalence of Pylint warnings in open source Python projects. Our analysis revealed that Pylint warnings are highly prevalent across a wide range of projects, with certain warnings like *line-too-long* and *missing-function-docstring* consistently appearing at the top of the list in both warning density and presence ratio.

The high frequency of warnings such as *missing-function-docstring* and *missing-module-docstring* suggests that documentation practices are often neglected in open source projects. This trend implies that many open source developers prioritize code functionality and contributions over thorough documentation. Writing docstrings can be viewed as an additional, non-essential task, particularly in community-driven projects where contributors may focus on achieving feature parity or resolving critical bugs rather than adhering strictly to style and documentation guidelines. This phenomenon may also reflect the transient nature of open source contributions, where developers who do not expect to engage long-term with a project might deprioritize writing detailed documentation.

Similarly, the persistent presence of the *line-too-long* warning across all strata could indicate that open source developers do not always consider strict line length limits as essential to code quality. Given the collaborative and distributed nature of open source projects, it is possible that line length limitations, while contributing to readability, are not enforced consistently due to differing coding practices, varied editor configurations, and the fact that enforcing this warning may not be seen as critical to a project's success. It also reflects the tension between personal coding habits and adherence to collective style guidelines in open source environments, where contributors bring diverse backgrounds and styles.

Warnings related to naming conventions, such as *invalid-name*, also appeared frequently. This suggests that developers in open source projects may not always adhere to standardized naming practices, possibly due to the organic and evolving nature of community-driven codebases. Names may be chosen for expedience rather than clarity or conformity with PEP 8 guidelines, particularly in fast-paced or collaborative coding environments where consistency is harder to maintain.

The prevalence of these warnings varied depending on the size and popularity of the projects, as categorized by forks and stars. In projects with fewer stars or forks, the warning densities were generally lower, but warnings like *missing-function-docstring* were ubiquitous across all strata. This suggests that smaller projects, which may have fewer contributors or be at an earlier stage of development, may focus less on thorough documentation and code style enforcement.

In larger projects, especially those with more than 100,000 stars or forks, the warning densities remained significant but showed more consistency across different warning types. This consistency indicates that larger projects, which often involve many contributors and more complex codebases, tend to have recurring code quality issues across a variety of areas, such as naming conventions, line length, and documentation. This may reflect the challenges of maintaining consistent coding standards in large, distributed teams where many contributors bring their own coding practices.

Overall, the findings demonstrate that while certain code style violations are frequent across all project sizes, larger and more complex projects exhibit a broader distribution of Pylint warnings. This suggests that code quality challenges are present at all levels of open source development, though they may manifest differently depending on the project's size and maturity. The recurring presence of specific warnings, such as missing docstrings and long lines, highlights broader trends in how open source developers approach code quality and the compromises made when balancing code contributions with best practices.

5.1.2 How effective is the developed automated tool at fixing Pylint warnings?

The second research question focused on the effectiveness of the developed automated tool in fixing Pylint warnings. The tool demonstrated a strong performance, achieving a complete project fix rate of 70%, and successfully addressing 88% of the identified warnings across all projects.

The tool was able to fix many Pylint warnings, including *line-too-long* and *missing-function-docstring*, with high accuracy. However, it encountered challenges with more complex warnings, particularly those requiring a broader context beyond the immediate code snippet provided. The remaining 12% of warnings that could not be fixed were primarily due to issues in extracting relevant code snippets, communicating with the Anthropic Claude API, or reinserting the fixes back into the original code.

These results highlight the effectiveness of AI-driven code repair tools, particularly for straightforward and frequent warnings. However, the technical limitations encountered in this study suggest that further refinement is needed to handle more complex warnings and improve the integration of automated fixes into diverse codebases.

5.1.3 How do open source developers perceive the suggestions made by the developed tool?

The third research question explored how open source developers perceive the suggestions made by the automated tool. Out of the 60 pull requests submitted, only 3 were fully approved and merged, while an additional 3 received positive comments but were not merged for unknown reasons. Seven pull requests were closed, with reviewers explicitly stating that they did not appreciate automated submissions.

These mixed reactions indicate that while some developers are open to automated code fixes, there remains significant resistance within the open source community to bot-generated pull requests. This resistance may stem from concerns about the accuracy or appropriateness of the fixes, or from a broader reluctance to adopt AI-generated changes in collaborative projects. The low interaction rate with the remaining pull requests suggests that many developers either overlooked the submissions or did not prioritize them.

This result highlights the need for more user engagement strategies and improved communication when proposing automated fixes in open source environments. Ensuring that developers feel informed and in control of the process could help alleviate concerns and increase the acceptance of automated pull requests.

5.1.4 How can we streamline a way of proposing research-related code changes with developer consent?

The final research question examined how to streamline the process of proposing research-related code changes with developer consent. Out of the 60 contributors and reviewers involved in the study, 4 provided explicit consent to the proposed changes. However, by the end of the project, two contributors had withdrawn their consent, leaving two active consents.

This result indicates that while some contributors are willing to engage with research-related changes, maintaining ongoing consent throughout the project can be challenging. The withdrawal of consent by two contributors highlights the importance of clear communication and transparency when requesting consent for automated changes. Additionally, the low overall consent rate suggests that more work is needed to refine the consent mechanism and make it more appealing to developers.

5.2 Future Work

There are several areas where this research could be expanded or refined to build upon the findings presented in this thesis.

5.2.1 Scaling the Dataset

One of the key limitations of this study is the relatively small dataset of 205 projects. Future research could expand the dataset to include a larger number of projects across a broader range of stars and forks. This would provide more comprehensive insights into the prevalence of Pylint warnings and the effectiveness of the tool across different types of open source projects. By increasing the number of projects, researchers could also explore whether the trends observed in this study hold true at a larger scale.

5.2.2 Improving AI Model Integration

The Claude 3 Haiku model used in this study was selected for its balance of performance and cost, but more advanced models, such as higher-end versions of Claude or even other models like GPT-4, could potentially offer improved results. Future work could explore the use of more sophisticated AI models to handle more complex warnings and improve the overall accuracy of the tool. Additionally, optimizing the communication between the tool and the AI model to ensure better context understanding could help address the technical challenges observed in this study.

5.2.3 Developer Engagement and Consent Mechanisms

Further research could investigate ways to increase developer engagement with automated pull requests. This could involve developing user interfaces that allow developers to review and approve automated suggestions more easily or creating mechanisms to better communicate the value of the proposed changes. Additionally, improving the consent process, possibly by offering more granular consent options or integrating it more seamlessly into the workflow, could help address the challenges related to developer withdrawal of consent.

5.2.4 Exploring Bias Against Automated Contributions

A notable finding from this research is the bias against automated pull requests, with several developers expressing negative views toward them. Future research could delve deeper into understanding this bias, exploring why some developers are hesitant to accept automated contributions and what can be done to improve the perception of such tools in the open source community. This could involve qualitative studies, surveys, or interviews with developers to gather more detailed insights.

5.3 Conclusion

This thesis explored the prevalence of Pylint warnings in open source Python projects, the effectiveness of an AI-driven tool to address these warnings, developer perceptions of automated code fixes, and the process of streamlining research-related code changes with developer consent. The findings demonstrate that Pylint warnings are highly prevalent across projects of all sizes and levels of popularity. Common warnings such as *line-too-long* and *missing-function-docstring* appear frequently, indicating that code quality challenges related to style and documentation are widespread in the open source Python ecosystem. These warnings were observed consistently across both small and large projects, suggesting that even well-established projects with significant contributor bases struggle with code style enforcement.

The automated tool developed as part of this research proved to be highly effective at addressing many of these common issues. With a fix rate of 88% for individual warnings and a 70% project-level fix rate, the tool demonstrated its capability to improve code quality by resolving frequent Pylint warnings. However, certain technical limitations were observed, particularly with more complex warnings that require a deeper contextual understanding of the code. Additionally, challenges related to developer engagement emerged, as some developers expressed hesitation or resistance toward automated pull requests.

The work presented in this thesis lays a strong foundation for further research into AI-driven code repair tools and their integration into open source development workflows. By continuing to refine the tool and addressing both the technical limitations and developer concerns, there is significant potential for automated tools to play an even larger role in improving code quality across the Python ecosystem. As AI-driven solutions continue to evolve, they could become indispensable for maintaining high coding standards in large and diverse codebases.

Bibliography

[1] Anthropic. Claude 3 haiku: Enhancing large language models for code generation. `https://www.anthropic.com/claude3/haiku`.

[2] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. Python coding style compliance on stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 210–214. IEEE, 2019. doi: 10.1109/MSR.2019.00042.

[3] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. Deepcode ai fix: Fixing security vulnerabilities with large language models. *arXiv preprint arXiv:2402.13291*, 2024. URL `https://arxiv.org/abs/2402.13291`.

[4] bndr. pipreqs: A tool to generate requirements.txt files for python projects. `https://github.com/bndr/pipreqs`.

[5] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023. URL `https://arxiv.org/abs/2303.12712`.

[6] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15 (3):1–45, 2024. URL `https://arxiv.org/abs/2307.03109`.

[7] Yiannis Charalambous, Edoardo Manino, and Lucas C Cordeiro. Automated repair of ai code with large language models and formal verification. *arXiv preprint arXiv:2405.08848*, 2024. URL `https://arxiv.org/abs/2405.08848`.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL `https://arxiv.org/abs/2107.03374`.

[9] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. Pyty: Repairing static type errors in python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[10] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023. URL `https://arxiv.org/abs/2304.07590`.

[11] Python Software Foundation. Black: The uncompromising code formatter. `https://black.readthedocs.io/`, .

[12] Python Software Foundation. Virtual environments. `https://docs.python.org/3/library/venv.html`, .

[13] Hristina Gulabovska and Zoltán Porkoláb. Survey on static analysis tools of python programs. In *SQAMIA*, 2019.

[14] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. A survey on large language models: Applications, challenges, limitations, and practical usage. *Authorea Preprints*, 2023.

[15] Siegfried Horschig, Toni Mattis, and Robert Hirschfeld. Do java programmers write better python? studying off-language code quality on github. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 127–134, 2018. doi: 10.1145/3191697.3214341.

[16] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. A survey on automated program repair techniques. *arXiv preprint arXiv:2303.18184*, 2023. URL `https://arxiv.org/abs/2303.18184`.

[17] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024. URL `https://arxiv.org/abs/2406.00515`.

[18] Nan Jiang and Yi Wu. Repaircat: Applying large language model to fix bugs in ai-generated programs. In *2024 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 58–60. IEEE, 2024. doi: 10.1145/3643788.3648020.

[19] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering*, 48(7):2658–2679, 2021. doi: 10.1109/TSE.2021.3067156.

[20] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573. IEEE, 2016. doi: 10.1109/SANER.2016.112.

[21] Matheus E Leusin, Bjoern Jindra, and Daniel S Hain. An evolutionary view on the emergence of artificial intelligence. *arXiv preprint arXiv:2102.00233*, 2021. URL `https://arxiv.org/abs/2102.00233`.

[22] Qing Mi, Haotian Bai, Xiaozhou Wang, Wenrui Liu, and Xingyue Song. An empirical study of coding style compliance on stack overflow.

[23] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022. doi: 10.1145/3524842.3528470.

[24] Wonseok Oh and Hakjoo Oh. Pyter: Effective program repair for python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 922–934, 2022. doi: 10.1145/3540250.3549130.

[25] Naelson Oliveira, Márcio Ribeiro, Rodrigo Bonifácio, Rohit Gheyi, Igor Wiese, and Baldoino Fonseca. Lint-based warnings in python code: Frequency, awareness and refactoring. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 208–218. IEEE, 2022. doi: 10.1109/SCAM55253.2022.00030.

[26] OpenAI. Chatgpt. `https://chat.openai.com/`, 2023.

[27] PyCQA. *Pylint Documentation*. Python Code Quality Authority, 2024. URL `https://pylint.pycqa.org/en/stable/`.

[28] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014. doi: 10.1145/2635868.2635922.

[29] AS Saabith, MMM Fareez, and T Vinothraj. Python current trend applications-an overview. *International Journal of Advance Engineering and Research Development*, 6(10), 2019.

[30] Diomidis Spinellis, Panos Louridas, Maria Kechagia, and Tushar Sharma. Broken windows: Exploring the applicability of a controversial theory on code quality. In *Proceedings of the 40th International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024. To appear.

[31] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017.

[32] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8–style guide for python code. *Python. org*, 1565:28, 2001. URL `https://peps.python.org/pep-0008/`.

[33] Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE, 2023. doi: MedAI59581.2023.00044.

[34] Michel Wermelinger. Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 172–178, 2023. doi: 10.1145/3545945.3569830.

[35] Marvin Wyrich, Raoul Ghit, Tobias Haller, and Christian Müller. Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests. In *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, pages 6–10. IEEE, 2021. URL `https://arxiv.org/abs/2103.03591`.

[36] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017. doi: 10.1109/MSR.2017.2.

[37] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023. URL `https://arxiv.org/abs/2303.18223`.

[38] Wenkang Zhong, Hongliang Ge, Hongfei Ai, Chuanyi Li, Kui Liu, Jidong Ge, and Bin Luo. Standup4npr: Standardizing setup for empirically comparing neural program repair systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022. doi: 10.1145/3551349.3556943.

Appendix A

List of pylint warnings

In this appendix, we give a list of all the pylint warnings that were in the lists of top 15 warnings. The warnings include what type of warning they are and a link to their documentation.

Warning Name	Warning Code (Link)	Message Type
abstract-method	abstract-method	Warning
bad-indentation	bad-indentation	Warning
bare-except	bare-except	Warning
broad-exception-caught	broad-exception-caught	Warning
consider-using-f-string	consider-using-f-string	Warning
consider-using-with	consider-using-with	Warning
import-error	import-error	Warning
import-outside-toplevel	import-outside-toplevel	Warning
inconsistent-return-statements	inconsistent-return-statements	Warning
invalid-name	invalid-name	Convention
line-too-long	line-too-long	Convention
missing-class-docstring	missing-class-docstring	Convention
missing-function-docstring	missing-function-docstring	Convention
missing-module-docstring	missing-module-docstring	Convention
no-else-return	no-else-return	Refactor
no-member	no-member	Error
no-name-in-module	no-name-in-module	Error
possibly-used-before-assignment	possibly-used-before-assignment	Warning
protected-access	protected-access	Warning
redefined-builtin	redefined-builtin	Warning
redefined-outer-name	redefined-outer-name	Warning
singleton-comparison	singleton-comparison	Refactor
syntax-error	syntax-error	Fatal
too-few-public-methods	too-few-public-methods	Refactor
too-many-arguments	too-many-arguments	Refactor
too-many-boolean-expressions	too-many-boolean-expressions	Refactor
too-many-branches	too-many-branches	Refactor
too-many-instance-attributes	too-many-instance-attributes	Refactor
too-many-locals	too-many-locals	Refactor
too-many-statements	too-many-statements	Refactor
trailing-whitespace	trailing-whitespace	Convention
undefined-variable	undefined-variable	Error
ungrouped-imports	ungrouped-imports	Convention
unnecessary-pass	unnecessary-pass	Refactor
unspecified-encoding	unspecified-encoding	Warning
unused-argument	unused-argument	Warning
unused-import	unused-import	Warning
unused-variable	unused-variable	Warning
useless-object-inheritance	useless-object-inheritance	Refactor
wrong-import-order	wrong-import-order	Convention
wrong-import-position	wrong-import-position	Convention

Table A.1: Pylint Warnings with Links and Message Types

Appendix B

Pull request data

This appendix provides detailed information about the pull requests submitted as part of the empirical study on how open source developers perceive automated suggestions. The table below includes the project name, the specific Python warning that the pull request aimed to resolve, the status of the pull request (open, closed, or merged), and a direct link to the submitted pull request for each project.

- **Project Name**: This column lists the names of the open source projects to which the automated pull requests were submitted. These projects span a wide range of domains and are hosted on GitHub.
- **Warning Name**: This column indicates the type of Python warning that was addressed in each pull request. The tool focused on resolving common Pylint warnings such as ”unspecified-encoding,” ”redefined-outer-name,” and ”no-else-return,” among others.
- **Status**: This column reflects the current status of each pull request. It indicates whether the pull request is still *open*, has been *closed* without merging, or has been *merged* into the project.
- **Pull Request Link**: This column provides a direct URL to the corresponding pull request on GitHub. Readers can follow the link to review the proposed changes, conversations with maintainers, and the final outcomes of the pull requests.

project	warning_name	status	pull_request_link
am-silex/anki_cambridge	unspecified-encoding	open	https://github.com/am-silex/anki_cambridge/pull/31
allenai/mmc4	unspecified-encoding	open	https://github.com/allenai/mmc4/pull/24
lanmaster53/recon-ng	unspecified-encoding	open	https://github.com/lanmaster53/recon-ng/pull/207
3b1b/manim	unspecified-encoding	open	https://github.com/3b1b/manim/pull/2141
Significant-Gravitas/AutoGPT	unspecified-encoding	open	https://github.com/Significant-Gravitas/AutoGPT/pull/7222
subframe7536/maple-font	unspecified-encoding	merged	https://github.com/subframe7536/maple-font/pull/203
magic-wormhole/magic-wormhole	unspecified-encoding	open	https://github.com/magic-wormhole/magic-wormhole/pull/529
digitaljohn/comfyui-propost	redefined-outer-name	open	https://github.com/digitaljohn/comfyui-propost/pull/25
lich0821/WeChatRobot	redefined-outer-name	open	https://github.com/lich0821/WeChatRobot/pull/65
PeterL1n/BackgroundMattingV2	redefined-outer-name	open	https://github.com/PeterL1n/BackgroundMattingV2/pull/210
pallets/flask	redefined-outer-name	closed	https://github.com/pallets/flask/pull/5503
AUTOMATIC1111/stable-diffusion-webui	redefined-outer-name	closed	https://github.com/AUTOMATIC1111/stable-diffusion-webui/pull/16048
bloomberg/memray	redefined-outer-name	closed	https://github.com/bloomberg/memray/pull/632
alexmohr/sonyapilib	unspecified-encoding	open	https://github.com/alexmohr/sonyapilib/pull/74
OpenDevin/OpenDevin	unspecified-encoding	closed	https://github.com/OpenDevin/OpenDevin/pull/2638
jaraco/keyring	unspecified-encoding	merged	https://github.com/jaraco/keyring/pull/685
hummingbot/hummingbot	unspecified-encoding	open	https://github.com/hummingbot/hummingbot/pull/7093
pep8speaks-org/pep8speaks	unspecified-encoding	open	https://github.com/pep8speaks-org/pep8speaks/pull/232
mysociety/mapit	no-else-return	open	https://github.com/mysociety/mapit/pull/431
automl/ConfigSpace	no-else-return	open	https://github.com/automl/ConfigSpace/pull/365
AcademySoftwareFoundation/OpenTimelineIO	no-else-return	open	https://github.com/AcademySoftwareFoundation/OpenTimelineIO/pull/1773
ajenti/ajenti	no-else-return	open	https://github.com/ajenti/ajenti/pull/1481
Unstructured-IO/unstructured-api	no-else-return	open	https://github.com/Unstructured-IO/unstructured-api/pull/435
onnx/onnx	consider-using-f-string	open	https://github.com/onnx/onnx/pull/6199
Lightning-AI/torchmetrics	redefined-outer-name	open	https://github.com/Lightning-AI/torchmetrics/pull/2610
virejdasani/Alexis	redefined-outer-name	open	https://github.com/virejdasani/Alexis/pull/48
yihong0618/bilingual_book_maker	redefined-outer-name	open	https://github.com/yihong0618/bilingual_book_maker/pull/408
motioneye-project/motioneye	redefined-outer-name	open	https://github.com/motioneye-project/motioneye/pull/3019
thatmattlove/hyperglass	raise-missing-from	closed	https://github.com/thatmattlove/hyperglass/pull/266
andrew-cr/jump-diffusion	raise-missing-from	open	https://github.com/andrew-cr/jump-diffusion/pull/4
vmware/vsphere-automation-sdk-python	raise-missing-from	open	https://github.com/vmware/vsphere-automation-sdk-python/pull/422
pydantic/FastUI	raise-missing-from	open	https://github.com/pydantic/FastUI/pull/336
python-visualization/folium	no-else-return	open	https://github.com/python-visualization/folium/pull/1983
getredash/redash	no-else-return	open	https://github.com/getredash/redash/pull/7041
1Panel-dev/MaxKB	no-else-return	merged	https://github.com/1Panel-dev/MaxKB/pull/678
tatsu-lab/alpaca_farm	no-else-return	open	https://github.com/tatsu-lab/alpaca_farm/pull/92
hgrecco/pint	no-else-return	open	https://github.com/hgrecco/pint/pull/2027
graphistry/pygraphistry	no-else-return	closed	https://github.com/graphistry/pygraphistry/pull/571
delitamakanda/elearning	no-else-return	open	https://github.com/delitamakanda/elearning/pull/96
electronstudio/raylib-python-cffi	no-else-return	closed	https://github.com/electronstudio/raylib-python-cffi/pull/133
carson-katri/dream-textures	no-else-return	open	https://github.com/carson-katri/dream-textures/pull/805
xrouting/xroute_env	no-else-return	open	https://github.com/xrouting/xroute_env/pull/5
freedomofpress/securedrop	raise-missing-from	open	https://github.com/freedomofpress/securedrop/pull/7198
FlareSolverr/FlareSolverr	raise-missing-from	open	https://github.com/FlareSolverr/FlareSolverr/pull/1243
EleutherAI/gpt-neox	raise-missing-from	open	https://github.com/EleutherAI/gpt-neox/pull/1249
run-llama/llama_index	raise-missing-from	open	https://github.com/run-llama/llama_index/pull/14521
HewlettPackard/ilo-ansible-collection	raise-missing-from	open	https://github.com/HewlettPackard/ilo-ansible-collection/pull/35
bordaigorl/rmview	raise-missing-from	open	https://github.com/bordaigorl/rmview/pull/166
PaddlePaddle/PaddleGAN	raise-missing-from	open	https://github.com/PaddlePaddle/PaddleGAN/pull/854
Blazemeter/taurus	raise-missing-from	open	https://github.com/Blazemeter/taurus/pull/1851
httpie/cli	raise-missing-from	open	https://github.com/httpie/cli/pull/1585
healthchecks/healthchecks	raise-missing-from	open	https://github.com/healthchecks/healthchecks/pull/1022
outlines-dev/outlines	raise-missing-from	open	https://github.com/outlines-dev/outlines/pull/1016
pyg-team/pytorch_geometric	raise-missing-from	open	https://github.com/pyg-team/pytorch_geometric/pull/9486
recommenders-team/recommenders	raise-missing-from	open	https://github.com/recommenders-team/recommenders/pull/2124
streamlit/streamlit	raise-missing-from	open	https://github.com/streamlit/streamlit/pull/9037
Integration-Automation/AutomationIDE	raise-missing-from	open	https://github.com/Integration-Automation/AutomationIDE/pull/75
elapouya/python-docx-template	raise-missing-from	open	https://github.com/elapouya/python-docx-template/pull/550
ivelum/djangoql	raise-missing-from	open	https://github.com/ivelum/djangoql/pull/119
pylint-dev/pylint	raise-missing-from	open	
abelcheung/types-lxml	raise-missing-from	open	

Table B.1: Pull request data