BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

PHD THESIS - SUMMARY

Supporting Software Maintainability, Reliability and Security through Static Analysis

Supervisor
Prof. Dr. Simona Motogna

PhD Student
Liviu Marian Berciu

2026

Acknowledgements

At last, after a long path full of challenges and unpredictabilities, this work can be considered complete. And while this is a moment of great happiness, it is also a moment to be grateful to the ones who made this work possible.

I want to express my sincere gratitude to my supervisor, professor Simona Motogna, for her dedication and invaluable contribution to this work. She has been a constant source of guidance and support, especially at critical moments throughout this process. She knew how to comfort me when a paper was rejected and how to put me back on track when I was lost. Her work ethic and discipline have been truly admirable and a lasting example for me, prompting me to be more professional and hard-working, both in my academic and professional life.

I am grateful to my thesis committee (Professor Laura Diosan, Associate Professor Camelia Serban, and Associate Professor Arthur Molnar) for their guidance and feedback, which helped in shaping my research and providing me with new scientific threads to work on. Another important pillar that helped me finish this thesis is my PhD colleagues (Vasilica Moldovan, Patcas Rares, Oskar Picus), who have been a source of inspiration and trust and have shown me what real collaboration feels like. I also thank my external thesis committee for their feedback and input, which helped me add the finishing touches to this thesis.

The people who have guided me on the path of academia are my parents, Liviu and Nicoleta, to whom I am extremely grateful. They taught me discipline, how to keep going until you reach the finish line, and how to never give up on my goals. For this, I am forever grateful. I also want to mention my sister, who has been a supportive pillar on my journey.

To the most important person in my life, my wife Alice, who encouraged me when it was difficult, listened to my complaints in silence, and put "my head on my shoulders" when it was time to man up: I thank you with all my heart for your love and encouragement, your patience, and your trust.

At last, for my baby boy Liam, who has given me the commitmment to finish this thesis before he was born.

Abstract

Static code analysis tools such as SonarQube are widely used to identify maintainability, reliability, and security issues, including code smells, bugs, and vulnerabilities. However, it remains challenging to characterize how these issues evolve over time in open-source systems, how they relate to developer activity, and how they can be detected and prioritized more effectively in practice. This thesis investigates these questions with a primary focus on open-source Python projects.

To enable the empirical studies, a repository-mining tool (PyDs Builder) and an open-source dataset (the Python Software Quality Dataset) are introduced to collect and integrate quality-related artifacts from version control and static analysis tools. Using these resources, a longitudinal analysis of SonarQube issues is conducted, showing that code smells represent the majority of reported issues and persist across releases, while bugs and vulnerabilities are less frequent and more unevenly distributed across projects.

Developer behavior is examined through commit-aware analyzes. Commit messages are classified into Conventional Commit Specification categories using large language models, supporting analyzes of how different types of development activity correlate with observed issues. In addition, commit burstiness is analyzed to assess whether rapid development periods are associated with changes in issue prevalence. Cross-language transferability is explored by comparing Python and JavaScript Abstract Syntax Trees for ten shared SonarQube rules using both NLP-based similarity measures and tree-edit distance; the results indicate that meaningful similarity is largely limited to simple, language-independent rules and that ASTs alone are often not enough to support cross-language rule transfer.

Finally, the thesis synthesizes prior work via a systematic literature review on artificial intelligence in refactoring (and vise versa), investigates refactoring prediction based on technical debt issues, and evaluates machine learning models for code smell detection in Python, where highly imbalanced data plays a central role in model performance.

Overall, the thesis contributes datasets, methods, and empirical evidence for long-term, commit-aware assessment and automation of software quality analysis in open-source projects, that benefits both researchers and developers.

Contents

List of publications

1. **L.-M. Berciu**, *"PyDsBuilder - A Dataset Builder Written in Python Django"*, Studia Universitatis Babeș-Bolyai Informatica, vol. 69, no. 2, pp. 5–22, Feb. 2025. ISSN: 2065-9601, DOI: 10.24193/subbi.2024.2.01. (*categ D, **1** point, MathSciNet indexed*) [Ber25]

2. **L.-M. Berciu**, V.-A. Moldovan, *"Software Maintainability and Refactorings Prediction Based on Technical Debt Issues"*, Studia Universitatis Babeș-Bolyai Informatica, vol. 68, no. 2, pp. 22–40, Oct. 2023. ISSN: 2065-9601, DOI: 10.24193/subbi.2023.2.02. (*categ D, **1** point, MathSciNet indexed*) [BM23]

3. M. Petrescu, S. Motogna, **L.-M. Berciu**, *"Women in Scrum Master Role: Challenges and Opportunities*"*, 2023 IEEE/ACM 4th Workshop on Gender Equity, Diversity, and Inclusion in Software Engineering (GEICSE), Melbourne, Australia, pp. 49–55, 2023. DOI: 10.1109/GEICSE59319.2023.00011. (*categ A, **6** points, Scopus indexed*) [PMB23]

4. S. Motogna, **L.-M. Berciu**, V.-A. Moldovan, *"Artificial Intelligence Methods in Software Refactoring: A Systematic Literature Review"*, 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Paris, France, pp. 309–316, 2024. DOI: 10.1109/SEAA64295.2024.00055. (*categ B, **4** points, Scopus indexed*) [MBM24]

5. V.-A. Moldovan, **L.-M. Berciu**, R.-D. Patcas, *"The Python Software Quality Dataset"*, 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Paris, France, pp. 395–398, 2024. DOI: 10.1109/SEAA64295.2024.00066. (*categ B, **4** points, Scopus indexed*) [MBP24]

6. **L.-M. Berciu**, S. Motogna, A. Molnar, *"A long-term exploratory study of source code quality issues in open-source Python projects"*, Software Quality Journal, vol. 34, no. 9, 2026. DOI: 10.1007/s11219-026-09740-z. (*categ C, **2** points, Scopus indexed*) [BMM26b]

7. **L.-M. Berciu**, S. Motogna, O. Pikus, A. Molnar, *"Exploring the relation between source code commit information and SonarQube issues"*, 29th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2025), vol. 270, pp. 841–850, 2025. ISSN: 1877-0509, DOI: 10.1016/j.procs.2025.09.204. (*categ B, **2** points, Scopus indexed*) [BMPM25]

8. **L.-M. Berciu**, V.-A. Moldovan, S. Motogna, *"Exploring the Impact of Commit Burstiness on Software Quality"*, 18th International Conference on the Quality of Information and Communications Technology (QUATIC 2025), 2025. (*categ C, **2** points, Scopus indexed*) [LMBM]

9. V.-A. Moldovan, S. Motogna, **L.-M. Berciu**, *"A Comparative Evaluation of Intelligent Methods for Code Smell Detection in Python"*, currently in publishing. (*categ C, **0** points, Scopus indexed*) [VAMM]

10. R.-D. Patcas, O. Pikus, **L.-M. Berciu**, S. Motogna, *"AST Similarity of Code Quality Issues in Python and JavaScript"*, currently in publishing. (*categ X, **0** points, Scopus indexed*) [RDPM]

Publications total score: **22** points*

* The ranking of publications was performed according to the journal and conference lists published by CNATDCU (National Council for the Recognition of University Degrees, Diplomas and Certificates) applicable for doctoral students enrolled between the years 2020-2024.

Introduction

Context

Software quality explains how well software performs, fits its purpose, and is maintained. In the context of this thesis, the *software quality dimension* is split into three main threads [fSC21]: maintainability, reliability, and security.

Software quality needs to be measured [Ber16] using metrics. The best way to measure it automatically is through *static analysis tools (SATs)* like SonarQube [Son24a] and PyLint [PyL23]. SonarQube is extensively explored in scientific works ([LLTH19, LST19a, LLHT20]) and used in our open-source software (OSS) studies.

Based on these measurements, programmers modify code through *refactoring*, changing the internal structure without altering behavior to make it easier to understand and cheaper to modify [Fow99].

Developer intent is another important aspect, shared through Git [git] commits. The classification of developer intent evolved from Swanson's three categories [Swa76] to the ten granular categories of the conventional commit specification (CCS) [ZZQL25].

The research path of this thesis goes from building a mining tool and an open-source dataset to using Artificial Intelligence to classify commit messages. Finally, we run ML models to predict Code Smells and identify differences and resemblances between Abstract Syntax Trees (ASTs) in different programming languages.

Motivation

The rise of collaborative software ([KS07]) has led to numerous OSS projects spanning multi-disciplinary domains and using different programming languages. This diversity faces a significant threat: vulnerabilities and bugs. The discipline that can prevent further harm is software quality [SGK+09].

While many experiments exist for Java ([LLTH19], [LST19a], [ALRT19]) and SonarQube's capabilities ([LLHT20]), I observed a gap regarding software quality in Python. In Python, most defects are code smells [CCMX16]. Furthermore, SonarQube rules for Python [Son24a] were almost half the number of Java rules, and available datasets were limited. We decided to create a vast Python dataset to analyze how SonarQube performs. This leads to our first objective, ***Objective 1****: analyze software quality attributes and their long term evolution in software projects.*

The second motivation comes from the advent of LLMs and developer intent. For Github [git], we chose to focus on commits as they are one of the most granular units of intent. Different works categorize commit messages into multiple categories [Swa76], while others use a more granular taxonomy [ZZQL25]. The open-source space allows for analyzing long periods of time and creating longitudinal studies [MM20b]. As described in [ZZQL25], LLMs can classify commit messages with higher throughput than manual classification, achieving considerably high precision (76%). This leads to our second objective, **Objective 2: the use of AI technologies for the longitudinal assess-**

ment of software quality issues over time.

Contributions

The goal of this thesis is to advance the domain of Software Engineering, especially regarding Software Quality, combined with Artificial Intelligence. The contributions are divided into three main parts, as presented in the following enumeration:

- **Part 1: Foundations**
  - **PyDs Builder and dataset development [Ber25]**: We present PyDs Builder, a Python-based tool for mining and analyzing quality-related artifacts. The thesis details its engineering process and deployment to produce the public Python Software Quality Dataset [MBP24], spanning SonarQube issues, code metrics, refactoring histories, and commit pairs for over 90 open-source projects.
- **Part 2: Code Issues**
  - **Longitudinal Analysis of Source Code Issues in Python Projects [BMM26b]:** Analyzing 57 open-source Python projects across a decade reveals that static analysis issues maintain a stable presence over time. Code smells, bugs, and vulnerabilities tend to cluster in specific project phases, underscoring the importance of continuous quality assessment.
  - **Source Code Commit Classification in Python using Large language models [BMPM25]:** Using a fine-tuned large language model, we classified over 90,000 commits with high accuracy. The contribution uncovered that documentation, bug fixes, and feature additions dominate, facilitating precise analysis of how commit types correlate with static analysis issues.
  - **Commit Burstiness and Its Impact on Source Code Quality [LMBM]:** Burst periods of rapid, successive commits are generally associated with a lower incidence of static analysis issues. Focused development during burst periods fosters better code quality, while non-burst phases exhibit a higher rate of issues, underscoring the importance of disciplined commit patterns.
  - **AST Similarity of Code Quality Issues in Python and JavaScript [RDPM]:** We compared the Abstract Syntax Trees (ASTs) of code fragments inducing shared SonarQube rules in Python and JavaScript. Results reveal that only simple, language-independent rules exhibit high cross-language AST similarity. Complexity-driven rules show consistently low similarity, demonstrating that ASTs alone are insufficient for cross-language rule transfer.
- **Part 3: Code Smells**
  - **Systematic Literature Review on AI in refactoring [MBM24]**: We conduct a systematic review of artificial intelligence methods applied to software refactoring, mapping the research landscape to identify dominant trends and underexplored challenges.
  - **Predictive Models for Maintainability and Refactoring: [BM23]** We propose supervised machine learning approaches for forecasting maintainability and refactoring requirements. The ExtraTrees classifier yields high accuracy, providing actionable insights for targeted maintenance and technical debt management.

- **Evaluating Intelligent Models for Code Smell Detection [VAMM]**: This contribution evaluates seven machine learning models and a genetic algorithm for detecting ten major code smells in Python. The study demonstrates that traditional classifiers such as kNN and Naive Bayes frequently outperform deep learning models for specific code smells.

Part I

Foundations

I.1 Theoretical Background

As with any piece of research, it is important to grasp the theoretical aspects that surround it. In this chapter, we will try to underline some of the most important concepts of software quality that will introduce the basic concepts and notions used in this thesis. We will explore a more technical view of the foundations consisting of *static analysis* and its purpose, the *tools* that are built out of the static analysis concept, what *datasets* are and how they help researchers worldwide, how developer intent and development practices can be figured out through *commit classification*, how rushing when writing code may introduce software issues through *commit burstiness* , and what *software refactoring* is and how it is indispensable to the software development process.

I.1.1 Static Analysis

Static analysis is a software engineering method that allows developers to identify potential issues by automatically scanning the codebase for its structure, syntax, and semantics. It offers insight into software quality without the actual need to execute the code. This method is implemented by creating a software tool which utilizes abstract syntax trees (ASTs) and heuristics.

From a historical viewpoint, static analysis emerged in the 1970s. Basic correctness checks were separated into dedicated tools with the creation of Stephen C Johnson's *lint* [Tho21], one of the first widely used tools to detect patterns in C code. In parallel, theoretical computer science developed general frameworks such as abstract interpretation [CC77], which provided a rigorous foundation for approximating program behavior.

Static analysis has evolved from simple syntax-level checks to a diverse ecosystem of tools analyzing security, reliability, and maintainability. Different kinds of problems, which we will call *issues*, have started to appear. For security, the issues are called *vulnerabilities*, which a malicious actor can use to damage the owner. For reliability, the issues are called *bugs*, representing an abnormal way for a system to behave. On the side of maintainability, the issues are called *code smells*, which correspond to source code that works but is hard to understand and change. It is important to clarify that tool-defined code smells are not the same as those defined by Fowler [Fow99]. Fowler's code smells indicate a deeper design problem, while static analysis tools define finer-grained checks that harm understandability but are not necessarily bugs or vulnerabilities.

The counterpart is dynamic analysis, which involves scanning the software while it is running. Dynamic analysis does not constitute the subject of our thesis, as we solely focus on static analysis. We will continue by examining some tools that have greatly contributed to the software quality ecosystem.

I.1.2 Static Analysis Tools

A static analysis tool examines the source code of another program to detect issues. We will mainly focus on the three categories of maintainability, reliability, and security.

The tool parses the code into an abstract syntax tree (AST) and applies heuristics for control-flow, data-flow, and semantic checks [Dat25]. The results are reported as issues and displayed in different forms, such as IDE widgets, desktop platforms [ST24], and web platforms [Son24a].

An important component of code issues is code metrics. Metrics denote quantitative measures computed using static formulas on the parsed code. Based on established thresholds, a combination of metric values constitutes a violation. Let's take the example of cognitive complexity [Sonnd]:

$$(V(G) = e - n + 2p)$$

If the result is > 15, it will trigger a code smell rule.
There are multiple tools that fit the category of SATs:

- Sonarqube: a popular tool with multi-language support for detecting code issues and computing metrics [Son25a].
- Understand: a commercial tool supporting dependency graphs and OOP metrics [ST24].
- PyLint: a Python-specific linter following PEP8 conventions and detecting code smells [PyL23].
- PySZZ: an open-source Python tool for identifying bug-inducing commits.
- Radon: an open-source Python tool that computes metrics such as cyclomatic complexity [Rub24].

SonarQube enhances software quality standards [Son24a] by detecting issues through a rule-based pipeline and shifting towards "Clean Code" principles. Understand [ST24] provides a detailed view of code structure and extracts object-oriented metrics to offer insights into potential design flaws [CK94]. PyLint checks Python source code against coding standards and generates categorized messages with severity levels to help the developer. PySZZ [RPS+23] produces defect-tracing metrics to determine which commit most likely introduced a bug. Radon [Rub24] uses custom checkers to extract metrics for detecting code smells.

These tools comprise the technical base that allows us to write our studies. Throughout this thesis we will particularly use Python tools, with some studies also focusing on Java [BM23] and JavaScript [RDPM].

Programmers need to pay special attention to false positives. A *false positive* is a reported issue that matches a rule syntactically, but is actually correct in the real context of the code and does not require a change [Son25b].

I.1.3 Commits Study

A commit represents a logical, atomic unit of change to the source of a codebase. A developer modifies files to achieve a specific purpose, such as implementing a feature or fixing a bug. Git commits act as a snapshot of the codebase at a specific moment in time, allowing for easy version switching and tracking.

The component parts of a commit are as follows:

- Hash ID: the unique identifier of the commit.
- Object Tree: represents the content of the commit, containing file blobs and the directory tree structure. Git stores the entire file system.
- Metadata: the context of the commit, including the author, the committer, and the timestamp.
- Parent Pointer: the reference to the preceding commit.
- Message: the human readable description of the changes. It is often overlooked, leading to unclear descriptions.

Commit classification represents the action of assigning a specific taxonomy to a *commit message* to interpret development activities and enhance code quality. This helps with software maintenance, which modifies a program after delivery to ensure it works correctly.

Swanson's work [Swa76] represents the traditional categorization of software maintenance: *corrective* (addressing defects), *adaptive* (responding to changing environments), and *perfective* (improving functionality).

While investigated by many studies [Hs23, HBS22, MV00], these traditional categories lack the granularity needed for deeper analysis [CHK+01]. Recent research [ZZQL25] emphasizes a finer granularity in commit classifications.

The Conventional Commit Specification (CCS) [Con20] offers a more structured framework to classify commits. It facilitates a deeper understanding of a commit's purpose and helps teams associate commit types with software quality metrics.

Commit bursts are periods of rapid, successive commits focused on particular components. They are indicators of potential defects and are associated with a heightened probability of defects ([NMB10, XF14, WNLB22]). Erratic bursts prompt hurried code changes, resulting in reduced quality or the accumulation of technical debt.

Commit classification and commit burstiness complement each other to form a closer-to-reality portrait of how software quality evolves. The developer intent and time windows of actual code uploads offer an overview of how writing code in a hurry influences the appearance of defects in different commit categories.

I.1.4 Software Refactoring

Software refactoring restructures a software system's internal structure without altering its external behavior [Fow99]. It focuses on reducing technical debt, improving maintainability, and optimizing efficiency.

Early programming lacked deliberate code improvement, risking tangled logic. In the 1970s, *Lehman's Laws of Software Evolution* [Leh80] showed that systems must continuously adapt (*Continuing Change*) and that internal complexity grows unless developers deliberately reduce it (*Increasing Complexity*).

The term *refactoring* first appeared in 1990 [OJ90] and in Opdyke's 1992 thesis [Opd92], and was independently addressed by Griswold [Gri91]. It gained widespread recognition in 1999 through Martin Fowler [Fow99] and Kent Beck [Bec99].

Automated support began with the *Smalltalk Refactoring Browser* in 1997 [RBJ97]. These tools operate on Abstract Syntax Trees (ASTs) to perform safe transformations. Modern IDEs and tools now seamlessly combine static analysis and artificial intelligence. PyRef [ALT+21] is an automated tool used in this thesis to mine refactoring data from Python projects. It analyzes git history and transforms modified files into ASTs to identify operations. Other tools that identify refactoring include RefactoringMiner [1], RefactoringCrawler [2], and RefDiff [3]. Analyzing refactoring in OSS projects al-

[1] https://github.com/tsantalis/RefactoringMiner
[2] http://dig.cs.illinois.edu/tools/RefactoringCrawler/
[3] https://github.com/aserg-ufmg/RefDiff

lows researchers to validate whether these changes actually improve code quality.

I.2 PyDs Builder: A dataset builder and mining tool

Repository mining is a foundational practice in empirical software engineering, but the available tools often impose constraints on the kinds of data that can be extracted, the static analysis tools that can be integrated, or the scale at which they can operate. In this chapter, we present PyDs Builder (acronym for Python Dataset Builder), a web- and API-based repository-mining tool built in Python with the Django framework, designed to address these limitations and help us in building our datasets. The results of this chapter are published in [Ber25], and the main scientific contributions is the creation of the application artifact [Ber24].

I.2.1 Motivation

Open-source software (OSS) projects have seen a constant increase in popularity [MP12]. Made freely accessible on platforms like GitHub, they invite a broad audience to utilize the software. The integration of versioning systems facilitates collaborative development and preserves a comprehensive history of the project's evolution, providing insight into development practices and enriching the research landscape [KS07].

Researchers have focused on the assessment of software quality, which refers to the degree to which a software product meets requirements in a reliable, efficient, and maintainable manner. The large number of OSS projects has allowed researchers to review software quality transparently [SGK+09].

Evaluating software quality requires high volumes of information. Here, data mining comes into play [CSS13], implying tools that extract experiment data. Examples include analyzing technical debt [LST19b] and studying maintainability [MM21]. Applying artificial intelligence also requires large amounts of mined data [Wan18], such as mining Github commit messages [KCA21] and automating bug detection [ESK18]. Repository mining involves custom software solutions. Some extract issue tracker data [JSM21] and [FMNL21], while others offer a wider range of extraction methods [SAB18].

This subchapter outlines an experience report for designing a repository-mining tool created for extracting datasets from systems like GitHub. The primary objective is to present a design proposal and provide an open source artifact [Ber24] called PyDs Builder. We provide an experiment scenario to demonstrate its effectiveness and suggestions for empirical research studies. The insights gained offer valuable lessons on designing scalable data mining tools.

The tool consists of two main parts. The first implies extracting issue tracking and overall project data from Github. The second implies running software quality tools such as SonarQube [Son24a] and SZZ [RPS+23] to refine the dataset and offer insights into software quality attributes. Both parts are essential for successfully building a dataset.

I.2.2 Tool design

Extracting data consistently for experiments often leads to the need to create a software solution. Structuring and normalizing large amounts of data involves employing different algorithms to scan, extract, and aggregate information. Instead of using different software tools that necessitate more effort, we report the experience of creating a solution that aggregates data from different streams into one uniform format that can be extended as researchers see fit.

An important aspect is **to decide what data formats are supported**. Data extracted via API requests or files (JSON, CSV, YAML) are processed into a single, uniform standard. We see the need to support data conversion into a standardized format, such as SQL, that enables complex queries and can scale as the amount of data increases.

Another important issue is the **automation of data extraction**. Implementing automation through task queues allows continuous data processing without constant supervision. Furthermore, the solution should be implemented in a popular programming language to ensure quick adoption and extension.

Obtaining the data in the desired format should allow further processing by feeding them to a data pipe of custom tools. The data obtained from the execution of the tools will then be saved in the same uniform structure. In conclusion, **the decision about flow is important**.

I.2.3 PyDs Builder Solution

We introduce a data mining tool focusing on the guidelines underlined above: **automation of data, format specification** and **flow decision**.

PyDs Builder is a web, API-based solution built using Python and Django Framework that aims to offer a way for researchers to create experimental datasets.

Its main focus is extracting repository data from versioning systems, initially offering support for Github. Data is processed into the desired form and inserted into a custom database, following a **pre-established SQL schema**. Data can be processed further by custom tools to complete an experiment's data acquisition goals, and then used for publishing a new dataset or feeding the data into an artificial intelligence solution.

An overall overview of the solution features is enumerated as follows:

- Allow tool interaction through an API interface
- Extract Github repository data such as issues, issue timelines, commit details, and source code
- Execute software quality tools such as SonarQube [Son24a], SZZ [RPS+23] and PyRef [ALT+21] and ensure data persistence
- Provide automation for fast data processing
- Allow contribution and code extension through project modularity and intuitive programming interface.

I.2.4 Conclusions

Data mining and open source project analysis have been important subjects in academia due to data availability. This subchapter started with the motivation of creating PyDs, a Python with Django Framework solution that enables researchers to generate datasets primarily focusing on software quality attributes experiments. We have covered the conceptual design, implementation steps, and provided detailed instructions for setting up and using the application. Finally, we have explored potential avenues for extending the application and shared our experience.

The subsequent stages involve its practical implementation in research projects. PyDs allows researchers to utilize it for data extraction in popular software quality tools such as SonarQube and SZZ. Currently, PyDs only supports Github, but there are future plans to integrate with Jira and extract CI/CD data from Jenkins. In the end, PyDs Builder has been successfully utilized to create

a dataset focused on open-source Python projects [MBP24], highlighting its flexibility and practical utility.

I.3 The Python Software Quality Dataset

Datasets are one of the most important pillars of a successful empirical study. As we wanted to focus our efforts on studying how software quality evolves in Python OSS projects, given that we built the tool from chapter I.2, and that Python datasets were sufficiently scarce in the literature ecosystem, we considered it necessary to build our own dataset, which will be used by us and by other researchers who might deem it useful for their purposes. This chapter presents the study [MBP24] that was done in collaboration with PhD students Vasilica-Andreea Moldovan and Rares Patcas. My contributions were setting up the criteria for project selection, running the PyDs Builder from chapter I.2 in order to extract data from the different tools, setting up the database architecture and the dataset creation overall.

I.3.1 Motivation

Software engineering has experienced significant growth, making Software Quality a domain of particular interest. It elevates overall software quality by enhancing functionality and correctness, helping to reduce risks associated with bugs.

To gain a deeper understanding of the code, static analysis identifies potential issues without execution. A consequential outcome is the identification of technical debt, which refers to the accumulated cost of suboptimal development decisions that lead to defects and complexity. Furthermore, software refactoring serves as a methodical approach to code cleanup, altering the internal structure without affecting external behavior to mitigate errors.

Numerous studies and datasets, such as [LST19b], [SK21], and [SSM05], have been compiled, but they primarily focus on Java. As Python has emerged as a strong competitor [Sri17], it prompts the consideration of studying Python projects in a similar manner.

Creating a comprehensive Python dataset is important to provide insight into software development practices and facilitate new studies on predicting technical debt, identifying refactoring opportunities, and evaluating software quality.

The objective of this study is to use the repository-mining tool from Chapter I.2 to create a comprehensive dataset [LB24] for Python projects. The dataset contains information for 92 Python projects, comprising 51.805.677 SonarQube issues, 324.969 SonarQube code quality metrics, 12.301 software refactoring records, 16.177 pairs of bug-inducing and bug-fixing commits, and 863.931 entries from the GitHub issue tracker.

I.3.2 Methodology

The methodology employed for constructing the dataset presented in this chapter draws upon various approaches documented in the field of Software Engineering, as discussed in prior works such as [RTLR18] and [LS08]. Furthermore, similar methodologies have been applied in several studies documented in the literature, including those referenced in [ODA+15] and [VSM13].

I.3.3 Inclusion/exclusion criteria

To identify projects for analysis and storage of results in the database, several criteria were systematically evaluated. Only open-source projects were scrutinized to ensure accessibility for validation. We have chosen projects that have Python as the dominant codebase, as we focus only on Python projects. We considered that at least three releases ensure structural stability and comparison points. The projects needed to have at least 50 issues in order to facilitate the SZZ algorithm and capture project complexity. The threshold of commits was set to 500, to ensure sufficient development activity. In the end, a minimum maturity of at least 3 years was required, in order to exclude unstable, early-stage projects. The considered criteria are concisely presented in Table I.3.1.

Criterion	Threshold
Python Code Usage	$\geq 50\%$
Project Releases	≥ 3
Number of Issues	≥ 50
Number of Commits	≥ 500
Project Age	≥ 3 years

Table I.3.1: Selection Criteria for Python Projects

I.3.4 Tools and Dataset Overview

The main tool used to create the dataset is PyDs Builder I.2, utilizing separate application modules and PostgreSQL [Pos23] for its high performance and parallel connection handling. We extracted data using SonarQube for its industry popularity, PyRef for its notable refactoring extraction performance [ALT+21], and PySZZ to find bug-inducing commits. This dataset encompasses an analysis of 92 open-source Python projects, where SonarQube unveiled 51,805,677 technical issues and 324,969 software quality metrics. Additionally, PySZZ yielded 16,177 pairs of bug-inducing and bug-fixing commits across 50 projects, while PyRef identified 12,301 instances of software refactoring across 52 projects. The final dataset is publicly available at [MBP24].

I.3.5 Conclusions

The objective of this chapter was to establish a comprehensive dataset encompassing factors that influence code maintainability. We focused on Python projects due to the language's prominence. The dataset includes technical debt issues, software quality metrics, pairs of bug-inducing and bug-fixing commits, and executed software refactoring processes.

Specialized tools extracted the data: SonarQube for static analysis, PyRef for executed refactorings, and PySZZ to gather information about pairs of bug-inducing and bug-fixing commits. These tools were integrated into a framework with a PostgreSQL database.

Projects with ambiguous errors during analysis were omitted. Our primary objective was to compile diverse data rather than attempting to rectify analysis issues.

The outcomes will empower researchers to conduct diverse investigations utilizing a shared dataset. Future research can explore various quality dimensions, such as technical debt and code smells, alongside potential causal relationships between software refactorings, bug commits, and technical debt issues.

Part II

Code Issues

II.1 A long-term exploratory study of source code quality issues in open-source Python projects

Empirical research targeting software quality has resulted in a consistent body of work, especially with the help of automated tools that allow researchers to mine large amounts of data. However, we find that many of these efforts provide cross-sectional or only short-term longitudinal analyzes. Furthermore, they are often focused on a single, and in most cases, statically typed language such as Java. In this subchapter, we aim to broaden the horizon of existing efforts by exploring the composition, distribution, and evolution of source code quality issues in complex, open-source Python projects. We employ the SonarQube static analysis tool on a dataset comprising 3,656 individual releases of 57 Python projects. We explore the impact of these issues on software maintainability, reliability, and security, investigate the evolution of these issues over the long term, and compare our results with those in the literature. Our findings are thus compared with existing research targeting both Python and Java. The resulting data is published and open source to help replicate the investigation and contribute to building open and large-scale datasets for research.

This chapter presents the study [BMM26b] that was conducted in collaboration with my supervisor, professor Simona Motogna, and associate professor, Arthur Molnar. My contributions were data curation, namely data extraction and processing, focusing on data analysis and answering the **RQ3** of the study by running the Spearman correlation and the creation of the dataset artifact [BMM26a] for the scientific community.

II.1.1 Motivation

Collaborative platforms like GitHub have been widely adopted for software development. In this process, our focus is represented by the quality of software. Source code analysis uses traditional and object-oriented metrics [AAM16, GFS05], as well as static analysis [dPB+23, Mol20, DACA22, MM22]. SonarQube is the most widely used tool [Ao21] due to its comprehensive analyzes. Applying it to successive releases provides historical data on different quality indicators that can serve as an identification of various bad practices.

The motivation of this subchapter lies in covering an empirical research gap. While literature extensively covers Java projects [LST19a, MM20a, DACA22, NCK+19], few studies address Python [TFAL21, TFA20]. We correlate source code modifications and SonarQube issues, focusing on the overall distribution of software quality categories. Java is statically typed and object-oriented, while Python is dynamically typed. This directly impacts SonarQube's rule applicability, meaning that rules common to both languages manifest differently. This motivates the need of a dedicated longitudinal study of Python codebases to better understand the evolution of their code quality.

In order to achieve our goal, we used an extensive dataset of 57 Python projects to extract source code metrics and code quality information using SonarQube.

Our original contributions can be summarized as:

- **Large longitudinal study of Python quality issues** comprising 57 projects and 3656 releases over 10 years;

- **Commit-level correlation analysis** allowing tracking of the long-term evolution of quality issues and relation with basic modifications;
- **Contribute to the understanding of the SonarQube tool itself** by analyzing how recent changes in its taxonomy impact issue detection.

II.1.2 Study Design

Objective and Research Questions

We define this chapter as a quantitative longitudinal case study [Ral21, KK15, MMD11] to observe, describe and explain SonarQube issues. It encompasses 57 Python projects, each having at least 6 investigated releases [KK15], and over 70% spanning at least 3 years.

Using the Goal-Question-Metric approach [BCR94], our objective is to investigate the composition, distribution and evolution of static analysis issues in open-source Python projects, and compare them with Java projects. We operationalize it via:

- **RQ1:** *What is the composition of static analysis issues?*
- **RQ2:** *How are static analysis issues distributed in the application source code?*
- **RQ3:** *What is the evolution of source code over the long term with regards to static analysis issues?*
- **RQ4:** *How does the composition of static analysis issues in the studied projects compare with existing results from the literature?*

Through **RQ1**, we targeted understanding how code smells, bugs, and vulnerabilities were distributed, and their impact on maintainability, reliability, and security [Son24a]. With **RQ2**, we analyzed the distribution of SonarQube issues to provide prioritization recommendations by explaining which rules produced the highest number of issues. Through **RQ3**, we intended to examine issue evolution over the long term alongside project metrics. Finally, **RQ4** compares the composition of uncovered issues with related work, examining frequent findings and discussing threats to validity.

II.1.3 Results and Discussion

RQ1: *What is the composition of static analysis issues?*

We started by analyzing the types of issues returned by SonarQube. Code smells represented, on average, 96.67% of all reported issues, with bugs at 3.08% and vulnerabilities at 0.25%. The majority of projects have a moderate to small number of code smells, indicating good code quality. There is a considerable variance in the average number of bugs per project. Vulnerabilities were rare, except in a few projects: *ansible*, *sqlalchemy*, *scrapy* and *plex-trakt-scrobbler*.

Finding 1: *Code smells represent a wide majority of code issues detected in Python projects. Higher occurrences of bugs or vulnerabilities represent isolated cases of low quality, not a trend.*

Starting with version 8.6, SonarQube introduced clean code attributes. Maintainability issues dominate (89.80%), whereas reliability (7.42%) and security (2.76%) represent smaller proportions. Differences in security distribution appeared in outlier cases like *matplotlib* and *keras* due to rules classified as code smells that have an additional low impact on security.

We explored whether quality issues depended on project size using lines of code (LOC). While visual inspection suggested project quality is not directly related to size, a per-project Spearman rank

correlation ($p < 0.05$) showed that as size increases, issue count also increases for maintainability, reliability, and security counts.

Finding 2: *SonarQube's count metrics provide precise insights into maintainability, reliability, and security. Per-project Spearman correlations show that LOC is generally positively associated with issue counts, and projects with lower quality often exhibit multiple quality concerns concurrently.*

Key Action Point 1: Code smells dominate quality issues, but bugs and vulnerabilities should be prioritized first. Recurring code smells in certain projects suggest quality issues might be inherited from third-party components.

RQ2: *How are static analysis issues distributed in the application source code?*

We aim to investigate how SonarQube issues are spread in the projects. Previous studies suggested [LST19a, MM22, WM18] that a small subset of rules were responsible for most issues. For each application, we considered the top 10 rules that generated issues, leading to a list of 73 rules.

Three rules generating code smell issues appeared in the large majority of the 57 analyzed projects: *S3776: Cognitive Complexity of functions should not be too high* (critical severity), *S1192: String literals should not be duplicated* (critical severity) and *S1481: Unused local variables should be removed* (minor severity). Furthermore, rules *S117* and *S1192* generated a considerable number of total issues.

Combining the distribution and severity of these rules, a small set requires special attention: *S1192*, which hides bad design decisions, and *S3776*, which makes impacted code difficult to understand.

Finding 3: *There is a small number of SonarQube rules that tend to appear in the majority of Python projects and that are responsible for the majority of code issues, with a high impact on maintainability.*

Key Action Point 2: The majority of issues arise from a small set of rules, pointing out cognitive complexity and bad design. These results can be used to prioritize issue resolution.

RQ3: *What is the evolution of source code over the long term with regards to static analysis issues?*

In this subsection, we explore the relationship between software quality (SQ) and commit changes over time. All selected projects were analyzed over their entire lifespan, with 90% spanning between 3 to 10 years. The primary goal was to investigate the correlation between the changes brought by commits and the evolution of software quality issues, informing whether frequent modifications lead to improvement or degradation [BASB17, KRS13, TEHG23].

We calculated the Spearman coefficient between commit metrics (code changes and file changes) and software quality metrics for each repository. Out of **342** total computed coefficients, we focused on statistically significant results ($p < 0.05$). After filtering, we retained **77** coefficients from **23** repositories, drawing from a solid sample size of 3202 entries [Buj24].

Finding 4: *Over 70% of projects show a correlation between file count or code changes and SonarQube issues associated with software quality attributes. Quality attributes associated with maintainability, reliability, and security exhibit medium variation during the development period of the projects.*

Key Action Point 3: Regularly iterating over source code design and implementation is correlated with improvements to source code quality, particularly for aspects related to maintainability and security.

RQ4: ***How does the composition of static analysis issues in the studied projects compare with existing results from the literature?***

To address this research question, we compared the SonarQube issues in our dataset with existing empirical studies. A rigorous comparison is constrained by differences in methodology, SonarQube versions, rule configurations, and selected projects. These factors, alongside the evolution of SonarQube's rules and language-specific differences, limit cross-study comparability. However, our analysis confirms that maintainability-related rules dominate issue counts across all studies. Most notably, *S3776* (high cognitive complexity) and *S125* (commented-out code) repeatedly appear among the most frequently violated rules in both Java and Python systems.

Finding 5: *There exists significant overlap between the rules generating the most issues across existing studies. Excessive cognitive complexity (S3776) and commented code sections (S125) appear prevalent across both Java and Python projects.*

Key Action Point 4: Some issues detectable using static analysis tools are common across projects and languages. Developers can improve code quality by configuring the applied rules and prioritizing resolution efforts towards issues having high importance.

II.1.4 Conclusions

The findings provide insight into the characterization and evolution of source code quality issues in open source Python projects. The large-scale longitudinal exploration and the comparative analysis with Java contribute to understanding Python specific dynamics of code quality.

One key observation is the dominance of code smells, with bugs and vulnerabilities being far less frequent. As code smells are a major cause of technical debt accumulation [DACA22, MM20a], this study underscores the most common rules violated and can be used as a guideline when prioritizing issue resolution during software development and code refactoring.

Lastly, we found that frequent code changes were associated with improved quality metrics in a subset of projects (≈40%), while most showed no significant relationship. This highlights that the effect of commit activity is project-specific, meaning continuous monitoring of quality indicators remains important.

For researchers, the results provide empirical evidence on the composition and long-term evolution of SonarQube issues for Python, alongside a comparison against Java. For practitioners, the study raises awareness on the most frequent code smells, bugs, and vulnerabilities, enabling proactive action to prioritize the most important issues based on their severity and impact.

II.2 Exploring the relation between source code commit information and SonarQube issues

This chapter represents the second part of code issues analysis, in which we explore how conventional commit specification categories are distributed between software releases. We analyzed 2,600 software release pairs of 57 Python open-source projects and employed large language models to label 90,318 commits. We identified which commit types are predominantly responsible for introducing SonarQube issues and explored the relationship between commit types and clean-code attributes affected by quality issues. We also published the resulting dataset so it can be used by other

researchers.

This study was done in collaboration with my supervisor prof. Simona Motogna, PhD student Oskar Picus and associate professor Arthur Molnar. My contributions were on the methodology, data collection and analysis, investigating research questions one and two, conclusions and the creation of the dataset artifact [BMMP25]. The article was accepted for publication at KES 2025 [BMPM25].

II.2.1 Motivation

Collaborative software development relies on the efficient management of code contributions through structured commits. Commit classification provides a systematic way to interpret development activities, measure their impact, and enhance overall software quality.

Traditionally, software maintenance activities have been categorized into three primary types: *corrective*, *adaptive*, and *perfective* [Swa76]. While investigated by multiple studies [Hs23, HBS22, MV00], these categories lack the granularity needed for deeper analysis [CHK+01]. Consequently, recent research [ZZQL25] emphasized a finer granularity in commit classifications to better understand the development process.

The Conventional Commit Specification (CCS) [Con20] was introduced, offering a structured framework to classify commits beyond traditional maintenance categories. CCS facilitates a deeper understanding of development activities, enabling teams to associate specific commit types with software quality metrics.

In this chapter, we explore how commit classifications correlate with software quality issues identified by the SonarQube [Son24a] static analysis tool, chosen because of its prevalence [TFA23, DLA+18, TFAL21]. We aimed to investigate commit classifications and their impact on software quality by examining the distribution of CCS categories. We identified which commit types were predominantly responsible for introducing SonarQube issues and explored the relationship between commit types and clean code categories [Son24b]. We hypothesize that understanding the types of changes associated with quality issues may help developers better manage technical debt.

II.2.2 Study Design

We refine our objective using the following research questions:

- **RQ5:** *What is the distribution of commit types among commits between releases?* As the CCS classification gained increased attention [ZZQL25], we focus on what types of commits are most prevalent between consecutive releases.
- **RQ6:** *What type of commits introduce most SonarQube issues?* We add the dimension of software quality concerns, as expressed through SonarQube issues to the commit categories.
- **RQ7:** *Is there an association between SonarQube's clean code categories and the CCS commit types?* Each software quality issue affects various clean code attributes, with SonarQube defining 4 main categories [Son24b]. Clean code serves as the foundation of maintainable and high-quality software development; therefore, this enables the assessment of quality aspects from a different perspective. Within this research question, we aim to investigate how this new dimension is associated to the commits.

II.2.3 Methodology

The present study analyzes commit data from open source Python projects, focusing on commits between consecutive GitHub releases [Git24]. We chose releases as they mark a milestone in code development. We targeted Python due to its popularity and the relative scarcity of exploratory work compared to Java. SonarQube (version 10.3) is our static analyzer of choice due to its industry popularity.

We used the dataset from [MBP24], spanning from 2013 to 2023, considering direct commit paths (**R1** $\rightarrow$ **R2**) and discarding non-linear progressions to include **2,648** release pairs and **90,318** commits, publishing the data [BMMP25]. We classified these commits into 10 CCS [Con20] categories using an LLM-driven CodeLLama model [ZZQL25] with an acceptable 76.41% F1-score, eliminating non-English commits using the langdetect library alongside incorrectly classified ones. To identify the commit that introduced a SonarQube issue, we designed a specific algorithm[1] that combines commit diff data, AST context data, and SonarQube uniqueness handling [Son25a] to link an issue in **R1** or **R2** to a specific commit. The algorithm matches Git diffs with issues, validates that the issue does not exist in adjacent releases, and removes duplicates. We validated it using a random sample of 100 issues, ignoring commits with parsing errors.

II.2.4 Analysis Results

RQ5: *What is the distribution of commit types among commits between releases?*

Conventional commit categories include 10 types [Con20]. The distribution shows a strong focus on *docs*, followed closely by *fix*. A percentage of 9% of commits are tagged as *feat*. Testing, refactoring and style have a decent overall distribution, while *CI*, *build* and *perf* appear more infrequently. We explored the distribution between considered releases and throughout the project's life. In most cases, there were one or two releases in which the number of commits significantly exceeded the rest.

Finding 6: *While we detected significant variation in the use of commit classification between releases, there exists a clear balance between different commit types, with documenting and feature development prevailing.*

RQ6: *What type of commits introduce most SonarQube issues?*

To address the second research question, we adopted a three-step method: we present the overall distribution of issues across different CCS categories, examine the top 10 tags, and highlight the top 10 breached rules.

A high number of issues were introduced by the *feat* category. In the case of the *refactor*, we observed roughly equal numbers of added and removed issues.

The data reveal that *convention*, *design*, and *numpy* tags consistently occupied the top position. This indicates that most issues are not critical, highlighting the prevalence of code smells. We highlight that the most frequently introduced and fixed issues are related to Python naming conventions (*python:S117*[2]), duplicated string literals (*python:S1192*), functions having high cognitive complexity (*python:S3776*) and unused parameters (*python:S1172*).

[1] A comprehensive view of the algorithm and its steps can be found in the replication package [BMMP25], in the **matching_algorithm_explained.MD** file.

[2] All SonarQube rules are detailed at `[https://next.sonarqube.com/sonarqube/coding_rules?languages=py&selected=python](https://next.sonarqube.com/sonarqube/coding_rules?languages=py&selected=python)%3AInequalityUsage`

Finding 7: *Regardless of CCS category, the majority of issues are code smells and primarily affect source code maintainability and readability. They can accumulate over time, making the codebase harder to manage.*

RQ7: *Is there an association between SonarQube's clean code categories and the CCS commit types?*

To study this association, we employed the χ^2 test of independence [dTF+19] with a 0.10 significance level [TAA+18]. We conducted pairwise tests grouped by commit type and applied Benjamini-Hochberg corrections. For significant associations, we computed Cramér's V to assess effect strength [Lee16]. Of the 40 tests performed, only 7 associations proved statistically significant, all exhibiting negligible effect sizes. This indicates no meaningful relationship between commit type and the clean code properties affected by associated quality issues.

Finding 8: *We could not identify a statistically significant association between clean code attributes as defined by SonarQube and the CCS commit types.*

II.2.5 Conclusions

This exploratory study examined how different types of commits relate to software quality issues detected by SonarQube in Python projects. We analyzed over 90,000 commits across software releases, using a large language model to classify commit messages and a custom algorithm to trace issues back to individual commits.

The results showed that *docs*, *feat* and *fix* were the most used commit categories. Most introduced issues were related to code smells, which impact code maintainability and readability rather than causing functional defects. These issues frequently appeared in feature additions and refactoring, suggesting the need for stricter code reviews and better refactoring practices.

We also found no meaningful statistical association between commit types and clean code attributes. This indicated that commit type alone is not a reliable predictor of quality concerns, as other elements may play a more significant role.

II.3 Exploring the Impact of Commit Burstiness on Software Quality

Commit burstiness represents periods of rapid development where successive commits are pushed to the repository and can be an indicator of developer behavior and its impact on software quality. This chapter represents the end of the "Code Issues" part, where we went from longitudinally analyzing open-source projects in Python on how software quality evolves over time, to applying CCS categories and see what exactly is the relative purpose of those commits and, in this chapter, we will see if the commit burstiness phenomenon is prevalent in our dataset and what conclusions we can extract from it.

This study [LMBM] was done in collaboration with PhD student Vasilica Moldovan and supervisor prof. Simona Motogna, and my contributions were on the methodology, experiment execution, results discussion, analysis and the dataset artifact creation [BMM25].

II.3.1 Motivation

Effective management of code contributions is crucial, especially in open-source projects. Maintaining quality becomes increasingly complex as systems evolve, highlighting the need for early identification of quality concerns.

Commit burstiness, characterized by rapid and successive commits, has emerged as a factor influencing software quality. Earlier studies ([NMB10, XF14, WNLB22]) introduced commit bursts as predictors of defects and highlighted their link to increased defect likelihood. While steady commit practices reflect healthy development, irregular bursts might trigger rushed code changes, leading to quality degradation or technical debt.

Semantic commit classification complements burst analysis by offering insights into the intentions behind individual changes. The Conventional Commit Specification (CCS) provides standardized labels like *feat*, *fix*, and *refactor*, addressing the limitations of traditional classifications ([Swa76, Con20, ZZQL25]).

In this chapter, we aim to bridge the gap by combining insights from commit burst analysis, structured commit labeling, and static quality metrics. We investigate whether the timing and intent of commits can serve as predictors for quality outcomes reported by SonarQube. Our objective is threefold: to validate whether commit bursts are predictors of increased issues, to explore whether different commit categories are more prone to bursts, and to verify if specific categories are more prone to introducing issues during burst versus non-burst periods.

II.3.2 Study Design

Our work is a quantitative exploratory study, following Software Engineering practices from [Ral21, KK15]. We want to check if there is a relation between commit bursts, SonarQube issues, and CCS categories. Unlike an explanatory study, which explains what is already known, an exploratory study checks if a new research path is worth pursuing.

The study is quantitative because we run the analysis on a large enough dataset, and a case study because we target open-source repositories together with SonarQube data. This approach has been used before in Software Engineering [AdM+23, Jm12].

We split our objective into three research questions:

- **RQ8:** *Does commit burstiness correlate with a higher chance of introducing software quality issues?*
- **RQ9:** *Are certain Conventional Commit Specification (CCS) types more prevalent during burst periods?*
- **RQ10:** *Is there a correlation between particular CCS commit types introducing more SonarQube issues in Burst versus Non-Burst commits?*

Data Selection and Processing

We use the Python Software Quality dataset[1], which contains 90,318 CCS-classified commits from 57 Python open-source projects. We kept only the commits that induce SonarQube issues, ending up with **4,784**. We picked Python because it is widely used in the industry, especially in AI. The commits span from 2013 to 2023. Wen et al. [WNLB22] sampled 500 commits in their work, so we consider 4,784 commits enough for exploratory statistical analysis. The dataset contains commit data (message, date, CCS classification) and SonarQube issues.

[1] Available at `https://doi.org/10.6084/m9.figshare.25008653.v4`

Burst window detection

To define a burst window (the period between two commits), we looked at prior work. Xuan et al. [XF14] use a window between 1 and 10 days. Nagappan et al. [NMB10] define it in *build days*, from one to three days, with 1 to 4 commits per burst. Wen et al. [WNLB22] use 5-minute increments. We also tried wider windows (2, 3, 6, 24, 48, and 72 hours) and commit counts (2 and 3 commits). In the end, we use at least **three commits** within a **two-hour** window, since it gave a clearer split between burst and non-burst activity and more consistent statistical results. Different thresholds may lead to different findings.

We implemented the burst detection ourselves. It merges overlapping activity periods into single events, so overlapping bursts are combined to avoid double-counting and reduce noise. For example, a burst from 3:00–5:00 PM is marked as one event, even if it slightly overlaps another interval. This works well for metrics and alerts, since it prevents multiple notifications for the same peak. Once a burst is identified, the next one starts after the fixed window (e.g., 2 hours), even if activity keeps going, which can split a long period into multiple bursts.

II.3.3 Data Analysis and Results

RQ8: *Does commit burstiness correlate with a higher chance of inducing SonarQube-reported issues?*

We used the Mann-Whitney U test to compare burst and non-burst commits, since it is non-parametric and handles unequal group sizes ($n_{\text{burst}} = 616$; $n_{\text{non-burst}} = 4167$). Following [TEHG23], we first ran the Shapiro-Wilk test [SW65] and confirmed the data was not normal, which justified the non-parametric choice.

We ran a two-sided test. The null hypothesis (H_0) was that there is no difference in induced issues between burst and non-burst periods. We got $U = 1\,184\,240.50$ and $p = 0.0014$, so we reject H_0. Non-burst commits had a higher average number of issues (mean $= 4.54$, median $= 1$) than burst commits (mean $= 2.88$, median $= 1$).

Finding 9: *Non-burst commits may be linked to more SonarQube issues than burst commits, hinting at more detectable quality problems when work is spread out rather than clustered. More analysis is needed to confirm this.*

RQ9: *Are certain Conventional Commit Specification (CCS) types more prevalent during burst periods?*

We ran a Chi-square test of independence to check if CCS categories are distributed differently in burst versus non-burst periods. We checked that every expected cell count was at least 5 [McH13], and applied the Benjamini-Hochberg correction on the p-values [dTF+19].

`refactor` stands out with $\chi^2 = 67.0005$, adjusted p well below 10^{-14}, and Cramér's $V = 0.1184$ (a weak but solid link). We saw 227 `refactor` commits in bursts and 909 outside. Only 20% of `refactor` commits happen in bursts, but `refactor` makes up 36.9% of all burst commits versus 21.8% of non-burst ones. So refactoring takes up a much bigger slice of activity during bursts. `fix`, `build`, `docs`, `style` and `ci` also lean toward non-burst periods, but their effect sizes ($V < 0.08$) are negligible. `perf`, `feat`, `chore` and `test` are spread evenly.

Finding 10: *Refactoring commits tend to be more common during bursty sessions, while most other commit types show little or no preference.*

RQ10: *Is there a correlation between particular CCS commit types introducing more SonarQube issues in Burst versus Non-Burst commits?*

We ran a two-sided Mann-Whitney U test for each CCS commit type to see if SonarQube issue counts differ between burst and non-burst commits. We applied the Holm [dTF+19] correction across all ten tests and noted the median issue count in each group to check directionality.

None of the adjusted p-values fell below 0.05. `perf` had a raw p of 0.036 with a median reduction in burst, but its Holm-adjusted p rose to 0.364. `docs` leaned the other way (more issues in burst), with adjusted $p = 0.869$. All other categories (`build`, `chore`, `fix`, `refactor`, `feat`, `style`, `test`, `ci`) had adjusted p-values at or above 0.868.

Even after multiple comparisons, there is no real evidence that any CCS type systematically introduces more SonarQube issues in burst versus non-burst commits.

Finding 11: *No clear evidence that any specific commit type consistently produces more software quality issues in burst versus non-burst sessions.*

II.3.4 Conclusions

This chapter looked at how commit burstiness interacts with commit semantics and their effect on software quality.

For **RQ8**, burst commits did not introduce more issues than non-burst periods, against our initial expectations. Quick sequences of commits may reflect focused, coordinated work rather than rushed changes. Teams could support these periods with better review processes or automation.

For **RQ9**, refactoring commits were especially frequent during bursts, pointing to a pattern where refactoring clusters together. Teams could use this to manage refactoring efforts more strategically.

For **RQ10**, we found no significant differences between commit types and SonarQube issues in burst versus non-burst intervals. Future work should look into issue severity and specific code smells to better inform QA practices.

Our findings partially align with [NMB10] and [NGM+19], which flagged bursts as defect predictors, but differ in suggesting bursts reflect refactoring rather than problematic activity. The results match [XF14] on team coordination benefits, but contrast with [WNLB22], since we link bursts mainly to systematic refactoring rather than disruptive quick fixes.

Overall, this study suggests rethinking assumptions about rapid code changes and points to potential benefits in productivity and quality. Future work can include project-level analysis, developer interviews, and a closer look at specific issue types.

II.4 AST Similarity of Code Quality Issues in Python and JavaScript

Transitioning from the longitudinal investigation of how code issues evolve, how they correlate with developer intent through CCS categories, and the impact of commit bursts on overall software quality, this chapter focuses on the underlying structure of these issues using Abstract Syntax Trees (ASTs). Specifically, this chapter presents a comparison of code issue similarity at the AST level across two programming languages: Python and JavaScript. The extracted ASTs are compared using NLP-based techniques and graph-based tree-edit distance. Furthermore, a manual inspection of the

code smells was conducted to better analyze the qualitative differences in issue patterns.

The study this chapter is based on was conducted together with PhD students Rares Patcas and Oskar Picus, and my supervisor prof. Simona Motogna. My contributions were data extraction, processing and analysis, results discussion on the AST comparison. The results are submitted for publication in [RDPM].

II.4.1 Motivation

Programming languages keep evolving, with new ones appearing and existing ones releasing often. Python and JavaScript follow an annual release schedule[1][2], while Java ships updates every six months[3]. This pace puts pressure on developers and QA workflows.

Software quality stays a constant concern, and may need even more attention with AI-generated code [CFI+24, LLCW+24]. SATs are widely used to catch quality issues early [MM22, LPS+23], with tools like SonarQube, Codacy, and PMD being popular among practitioners [LAL15, MVS23]. They parse source code into Abstract Syntax Trees and apply language-specific rules based on coding standards, best practices, and anti-patterns, flagging issues like code smells, excessive complexity, and convention violations.

This chapter focuses on the similarity of ASTs tied to SonarQube rules when those rules target the same conceptual issue in Python and JavaScript. High similarity would mean there is value in transferring rules from one language to another, which could close the gap in language support (750 rules for Java versus 86 for Ruby).

We picked Python and JavaScript because they are popular, share similar constructs (functions, classes, control-flow statements) that allow structural comparisons despite syntax differences, and have comparable rule coverage (414 issue-related rules in Python, 424 in JavaScript).

Our contributions are:

- the analysis of a subset of shared SonarQube rules between Python and JavaScript and their comparison through AST representation
- the application of structural and semantic similarity measures to characterize cross-language patterns and challenges of rule similarity
- assess whether AST-based similarity can support cross-language rule transfer.

II.4.2 Approach

We adopt the GQM methodology [BCR94] and define our objective as follows: **Analyze** the ASTs linked to language-specific SonarQube rules for Python and JavaScript **for the purpose of** assessing structural and semantic similarity across languages **with respect to** coding issues that address the same conceptual defects **from the viewpoint of** users of static analysis tools **in the context of** open-source projects written in Python and JavaScript.

The research question is the following: **How similar are the ASTs associated with Python and JavaScript SonarQube rules addressing the same coding issue?** Several similarity metrics are then defined in order to provide the response to this research question.

[1] https://peps.python.org/pep-0602/
[2] https://github.com/tc39/proposals
[3] https://openjdk.org/jeps/3

Methodology

The first step of the methodology is to extract the agreed-upon SonarQube rules from opensource projects for both languages. We then convert the issue code using Tree-sitter. The next and final step is the similarity comparison, both by manual review and by automation, using NLP and Graph-Based techniques.

Dataset

We built a dataset of open-source Python and JavaScript projects with SonarQube metadata and code fragments for AST analysis. For JavaScript, we ran SonarQube 10.3.0 on mature repositories from prior studies [CBZK19, SDC24], getting 64,591 issues. For Python, we used the Python Software Quality Dataset [MBP24], based on the same SonarQube version, and subsampled it to 457,935 valid *OPEN* issues to reduce the size imbalance.

For each issue, we extracted the affected lines and parsed them with Tree-sitter [LAAZ23]. We then ranked the shared SonarQube rules by their minimum occurrence across both languages and kept the top 10, avoiding rarely triggered rules [LML22, BLRS20, MM20a]. These cover a range of code quality categories.

NLP-Based Similarity Analysis

We normalized the ASTs into linearized Tree-sitter representations and reduced cross-language syntax differences with deterministic token replacement on AST labels (e.g. **module** to **program**, **function_definition** to **function_declaration**). The full mapping is in the replication package.

We used lexical set-based measures (Jaccard, Dice, Overlap), TF-IDF with cosine similarity, and Sentence-BERT [RG19] for the semantic side.

AST-Based Similarity Analysis

We parse code fragments with Tree-sitter and normalize ASTs the same way as in the NLP analysis. The trees are wrapped in `LabelTree` objects and passed to AP-TED, configured with unit costs for insertions and deletions, while rename operations cost proportionally to node label similarity via Python's `difflib.SequenceMatcher`.

We ran pairwise distance calculations across twenty parallel workers with periodic checkpointing, then converted the distances to similarity scores on $[0, 1]$, same scale as the NLP measures.

A simple baseline is:

$$\text{sim}_{ij} = \frac{1}{1 + d_{ij}},$$

but it ignores tree size. So we use **TSED**:

$$\text{TSED}_{ij} = \max\left(1 - \frac{d_{ij}}{\max(|\text{AST}_i|, |\text{AST}_j|)}, 0\right)$$

TSED normalizes to the larger tree, giving fairer comparisons across AST sizes.

Since AP-TED is heavy, for each rule we took the smaller count between the two languages and randomly sampled the larger set down to match. For example, rule S1481 has 748 JavaScript instances and 34,792 Python ones, so we picked 748 Python ASTs. This cut execution from over two days to a manageable level while keeping representativeness.

II.4.3 Results and Discussion

NLP-Based Similarity Results

Scores were normalized to $[0, 1]$, with 1 meaning perfect similarity. Full tables are in the replication package.

Jaccard fell between 0.39 and 0.45, implying some structural similarity for most rules. S1135 and S125 scored highest, both about comments, since Tree-sitter only kept the structural markers, not the content.

Overlap similarity was high across all rules, showing that AST nodes are largely included in one another. Code smells are linked with repeated use of similar syntactic structures in both languages.

Dice, between 0.52 and 0.8, shows moderate similarity: shared elements but also real differences from varied coding styles. This reinforces that code smells are not random and show structural uniformity across languages.

TF-IDF confirms this, especially for *S125* and *S1135*. Other rules got low scores, meaning code fragments introducing smells differ in token frequencies and distributions, with developers using similar structural constructs in different ways to introduce the same smell.

Sentence-BERT showed limited cross-language semantic alignment, with means between 0.56 and 0.66.

Overall, the NLP analysis found consistent patterns for commented-out code rules (S125, S1135) and for S1186 (empty function headers). For the other rules, structural differences between Python and JavaScript point to differences in coding practices or inconsistencies in how these smells are defined, which opens room for further investigation.

Graph-Based Similarity Analysis

The TSED results report descriptive statistics for each rule, in the same format as the NLP analysis. The highest means are S125 at 0.70 and S1135 at 0.77. For both, median and upper quartile reach 1, meaning many AST pairs are structurally identical. But their large standard deviations (0.34 for S125, 0.36 for S1135) show that some instances diverge a lot, producing a wider spread. So these rules often show strong cross-language alignment, but the consistency is not uniform across all examples.

The graph-based analysis shows that only some rule categories, mainly those relying on simple syntactic patterns, have higher cross-language similarity and are strong candidates for automatic transposition. Rules with lower TSED values reflect language-specific constructs and need rule-by-rule adaptation. Some rules can be generalized, others need hybrid strategies that combine structural and semantic normalization.

Besides S125 and S1135, most rules have low mean TSED values, like S3358 (0.14), S1481 (0.22), and S1763 (0.26), which reveal deeper language-specific complexities. S3358 involves nested ternary operators that look very different at AST level, S1481 flags unused local variables across varying declaration syntaxes, and S1763 detects unreachable code in language-specific constructs. Intermediate rules like S1186 (0.43), S1854 (0.27), and S905 (0.29) fall between the extremes, showing partial structural overlap with language-dependent variations.

Overall, the graph-based analysis, like the NLP one, shows two distinct groups of rules. Rules with simple, language-agnostic structures (like the ones about comments) show high similarity and can be transferred with minimal adjustment. Rules that capture language-specific constructs (nested ternaries, unused variables, unreachable code) show lower similarity and need careful adaptation. This distinction shows where automated rule transposition is feasible and where hybrid approaches

are needed, giving a concrete basis for future work on adapting static analysis rules across languages.

II.4.4 Conclusions

This chapter explored the structural differences between coding issues extracted from SonarQube in two different programming languages, Python and Javascript. Using the issues dataset, we constructed the ASTs of the respective issue-generating code and compared their similarity using multiple techniques. Our purpose was to investigate the possibility of translation of quality issues between languages.

We have identified three categories of SonarQube rules in terms of their similarity across languages, with AST-related patterns and highly complexity-driven patterns indicating that ASTs are insufficient for this task. Our findings show that ASTs alone cannot support cross-language rule transfer.

From a research perspective, this contribution offers valuable lessons and suggests that translation of quality-related issues between languages is a non-trivial problem requiring other research approaches. For practitioners, it points out that SATs may have limited applicability in detecting quality issues across languages due to structural and semantic differences.

Part III

Code Smells

III.1 Artificial Intelligence Methods in Software Refactoring: A Systematic Literature Review

This chapter represents the first step in exploring refactoring, as it investigates the integration of Artificial Intelligence and Machine Learning within this software process. Throughout it, we analyze 156 articles published between 2010 and 2024 and we map the state-of-the-art applications of AI methods across various refactoring lifecycle stages, while offering insights into the trend of publications and where the focus is in this sub-domain of software.

This research [MBM24] was done in collaboration with my supervisor prof. Simona Motogna and PhD student Vasilica Moldovan. My contributions were data analysis, automating the pipelines for data extraction, methodology and search strategy, and threats to validity.

III.1.1 Motivation

Refactoring is an important process in software development, aiming to improve code structure, maintainability, and reusability without changing functionality [Fow99]. It has stayed a continuous research interest due to its impact on code quality, performance, and energy consumption [CBMP14, TDPT+21, VSPL18].

The last years have seen growing interest in applying Artificial Intelligence (AI) and especially Machine Learning (ML) to software engineering problems. This is driven by fast progress in AI/ML algorithms and the availability of large software datasets. Examples include mining software repositories, code smell detection, test case generation, and development effort [BCRP20]. Tools like WEKA [Uni] and Scikit-Learn [PVG+11] have given the field an extra boost.

The same applies to refactoring. With this work, we want to give a broad overview of how refactoring and AI/ML techniques interact. Prior studies have looked at refactoring, but either as a whole [AAK+20] or from the angle of quality attributes [ADA18].

The main goal of this chapter is a systematic literature review (SLR) that maps the state of the art in applying AI/ML methods to refactoring stages. The SLR follows a defined protocol, with primary studies selected from scientific literature sources. We share the resulting dataset for the research community. The quantitative analysis identifies trends and open topics in the field.

III.1.2 Research Methodology

Our approach to literature review is based on [RB22], defining it as a scoping literature review with the purpose of describing the domain of AI/ML in relation with refactoring software systems and mapping the studies from this domain. We investigated existing standards in the domain [Pau21] and similar studies in software engineering [AAK+20, KC07, CD10, ADA18]. We also followed the guidelines described in [KC07] and documented each step as summarized in Table III.1.1. Steps 1 and 2 belong to planning, steps 3 and 4 for conducting the review, and step 5 refers to reporting. This section describes details about each step.

Table III.1.1: Protocol followed for SLR

No.	Step	Description
1	Goal	Impact of AI/ML on refactoring
2	Research Questions	Which refactoring phases? Which AI/ML methods?
3	Search Strategy	Databases for primary studies What keywords will be used? What information is retrieved?
4	Selection criteria	Inclusion criteria Exclusion criteria
5	Synthesis of data	Information to be synthesized Data Analysis

Research Questions

The research objective of discussing the state-of-the-art in the field of applying AI/ML methods to refactoring lifecycle stages is structured into the following research questions:

- **RQ11:** *What life-cycle phases of refactoring are addressed?* Refactoring is a complex process that incorporates several phases, starting from detection, prioritization, and recommendation, processes that should be accompanied by documentation and testing, and might be predicted. There are several classifications/namings of these phases (such as in [BCH+05] or [AAK+20]). We target to investigate how the selected studies target these phases.
- **RQ12:** *Which AI/ML methods are used?* Our exploration will inquire what type of AI methods are used and which are the most frequent AI/ML applied for refactoring.

III.1.3 Results and Discussion

The results for **RQ11** show that the majority of research effort is directed toward the detection and recommendation stages of the refactoring lifecycle. Detection remains the most popular area, accounting for over 37% of the papers, while recommendation follows closely at nearly 33%. Conversely, stages such as documentation, prioritization, and testing are notably under-explored. To answer **RQ12**, we found out that, from a methodological standpoint, supervised learning is the most frequent approach, used in over 56% of the studies. The most prevalent AI methods identified include Random Forests, Genetic Algorithms, Support Vector Machines, Convolutional Neural Networks, and Decision Trees. Specifically, CNNs are favored for detection tasks, while Genetic Algorithms are predominantly used for refactoring recommendations.

The implications of these findings are the following:

- Over 72% of analyzed research focuses on proposing new solutions, while evaluation, validation, and experience-based reserach is underrepresented. More empirical validation is necessary for these AI solutions to be adopted in real-world software practices.
- More focus on Prioritization and Prediction could significantly reduce development costs and improve performance.
- The rising interest in hybrid AI methods suggest that researchers are combining multiple learning types to improve model reliability and adaptability.

III.1.4 Conclusions

This chapter presents a systematic literature review (SLR) examining the use of AI in software refactoring, following established guidelines [KC07]. From an initial pool of 1,986 papers, 156 studies were retained, outlining the state of research and AI/ML adoption in refactoring. Trends from 2010–2024 show steady growth, with notable acceleration after 2016 and peaks in 2022–2023, reflecting sustained community interest in applying AI/ML to refactoring.

We addressed core research questions on which lifecycle stages are tackled by AI methods and which techniques dominate. Most contributions focus on *detection*, followed by *recommendation* and *prediction*. *Prioritization* and *prediction* appear especially promising for ML-driven improvements, potentially reducing development costs [Fow99]. Hybrid approaches are also increasingly explored, aiming to boost accuracy and robustness by combining methods.

A key gap lies in the scarcity of evaluation, validation, and experience studies, which are critical for real-world adoption. The most common AI methods are Random Forests, Genetic Algorithms, SVMs, CNNs, and Decision Trees, applied differently across lifecycle stages depending on task suitability. Most models (56.41%) use supervised learning, followed by unsupervised and semi-supervised approaches, indicating the benefits of labeled data for precise predictions.

III.2 Software Maintainability and Refactorings Prediction Based on Technical Debt Issues

Software maintainability remains a decisive factor in the software development lifecycle, directly impacting the cost, time, and resource allocation. Hence, as software evolves, software refactoring becomes essential to enhance quality, readability and extensibility. In this subchapter, we aim to explore two main things: first, the prediction of software maintainability and the number of required code refactorings, by leveraging technical debt issues and Machine Learning. Second, an extension of the previous goal, but by directly comparing the SonarQube issues of three projects between versions: TuxGuitar, JEdit and FreeMind, where we aim to observe how a project evolves and what are the issue categories that change between versions. As such, we will concentrate on the second goal and fleetingly mention the first one.

This study [BM23] was conducted in collaboration with my colleague Vasilica Moldovan, and my contributions were related to the in-depth analysis of SonarQube issues in the three analyzed projects.

III.2.1 Motivation

It has come to light that there is a strong connection between software systems' maintainability and technical debt issues. Those issues can be identified using SATs, such as SonarQube. At a first look, a small number of issues is responsible for a higher percentage of maintainability and vice versa. Hence, one of the code's problems found in the static analysis process is represented by the technical debt.

Research confirms a strong correlation between detected software issues, maintainability, and refactoring [ABJ10, Mar04, WDS16]. Refactoring is important for reducing technical debt and enhancing scalability; when excessive issues degrade maintainability, it serves to restore code quality.

To support this, the industry utilizes static analysis tools, such as SonarQube, PyLint, and FindBugs[1], to identify debt , alongside utilities like [2] , that detect refactorings between versions. Some of those tools are also described in I.1.

The goal of the study conducted is twofold: to predict the number of refactorings that need to be performed and, based on this number, to classify the maintainability of each software component that was refactored by using ML algorithms. The second goal is to compare the SonarQube issues for three Java projects in depth by executing SonarQube on their versions, extracting issues, and running a comprehensive statistical analysis on them.

The focus of this subchapter, however, is on the second goal, as it represents my significant contribution, and presents the results from the first one.

III.2.2 Refactoring Prediction: Experiment and Results

The experiment focused on gathering and preparing data to predict software maintainability and the number of refactorings needed. We ran a comprehensive analysis on three open-source Java projects: jEdit[3], FreeMind[4], and TuxGuitar[5]. We looked at technical debt in jEdit 5.5, FreeMind 1.0.1, and TuxGuitar 1.5.2, and at the refactorings done in the next versions (jEdit 5.6, FreeMind 1.1.0, TuxGuitar 1.5.3). The technical debt dataset was already established in prior work [Mol20, MM20a].

To build the dataset, we ran SonarQube on the projects to extract technical debt attributes (severity, debt, issue type), and RefactoringMiner to count the refactorings done per component between versions, which served as *groundtruth*. The final dataset merged these sources, linking technical debt metrics to refactoring count per component. We split the data 70% training, 30% testing.

We framed the prediction problem as both a classification and a regression task. For classification, components fell into three maintainability classes based on refactoring counts: "Great" (< 5 refactorings), "Good" (5–20), and "Poor" (> 20). We tried two approaches for data representation.

The first approach, with compressed component-level data, gave better predictive performance. The ExtraTrees Classifier was the best model, with accuracy, recall, precision, and F1-Score all at 0.92. For regression, the ExtraTrees Regressor also came out on top with R-Squared of 0.92 and RMSE of 2.75. The RNN in the second approach was less effective, reaching only 0.57 classification accuracy and R-Squared of 0.50.

Looking at classification by category, the "Good" class had the highest metrics, while "Poor" had the lowest, reflecting the imbalance in the initial dataset. We compared predictions against the Maintainability Index (MI) [OH92]. There were some mismatches: for example, MI classified FreeMind as "Bad" while our results suggested otherwise, which supports the idea that the traditional Maintainability Index may not accurately capture the complexities of object-oriented systems [MM21].

III.2.3 A deeper dive into SonarQube issues

To present the specifics of SonarQube issues and compare them between versions, we observed that all three datasets showed improvements in total issue counts, suggesting continuous refactoring. However, the quality of changes differed significantly. **JEdit** had the highest issue count, and while total issues dropped from 139,905 in version 5.5 to 63,899 in version 5.6, BUG rule violations rose from 2.86% to 9.59%, meaning overall code quality decreased as severity went up. **Freemind** showed

[1], https://spotbugs.github.io/
[2]RefactoringMiner, https://github.com/tsantalis/RefactoringMiner
[3]jEdit, http://www.jedit.org/
[4]FreeMind, https://freemind.sourceforge.net/
[5]TuxGuitar, https://sourceforge.net/projects/tuxguitar/

steady progress with a 20% improvement in code quality as raw numbers dropped, maintaining a similar ratio of 97% code smells and 2% bugs, although its maintainability index remained "Bad", showing the metric may not apply well to empirical studies. **TuxGuitar** had the best result, with every issue category decreasing from version 1.5.2 to 1.5.3 and a drop in bug distribution, reflecting a "Satisfactory" maintainability index. Tying this back, overall quality can decrease despite fewer total issues if severe bugs rise, as seen in JEdit, whereas projects like TuxGuitar show that refactoring work pays off with fewer bugs, fewer vulnerabilities, and better maintainability.

III.2.4 Conclusions

This subchapter looked at the predictive link between technical debt and software maintainability, using a dataset from three open-source Java projects: jEdit, FreeMind, and TuxGuitar. We combined SonarQube static analysis with RefactoringMiner data to test two approaches for predicting refactoring counts and classifying components.

The first goal showed that compressing technical debt issues into component-level mean values beats analyzing granular issue sequences. The ExtraTrees Classifier reached 0.92 accuracy, well above the RNN (0.57). The ExtraTrees Regressor also led in the regression task with R-Squared of 0.92. Aggregated metrics of severity and issue type seem solid for anticipating maintenance needs, while sequential complexity of individual issues may add noise.

The second goal showed that all three projects had fewer total issues between versions, but the *quality* of the changes varied. jEdit had a concerning trend: fewer code smells but bugs rose from 2% to 9%, pointing to a drop in stability. TuxGuitar had the best trajectory, with reductions across all categories and a better maintainability index.

III.3 A Comparative Evaluation of Intelligent Methods for Code Smell Detection in Python

While the goal of the previous chapter was to investigate the prediction of the number of required refactorings, in order to improve the maintainability of software components using technical debt data, we go further to achieve the goal of this chapter: namely, to evaluate seven of the most commonly used machine learning models and a genetic algorithm for code smell detection. Throughout the study presented next, we will examine and compare the performance of each model across ten distinct code smells, identify the most suitable model for each analyzed code smell, highlight strengths and limitations, and, in the end, augment the dataset from [MBP24] by adding new metric values and code smell classifications.

This study [VAMM] was done in collaboration with my supervisor prof. Simona Motogna and PhD student Vasilica Moldovan. My contributions were dataset creation, methodology, and experiment preparation.

III.3.1 Motivation

A "Code smell" is any code base issue that can lead to more critical problems [Fow99]. Catching smells early has a positive impact on the final system [Fow99, TPB+17]. Since smells are described rather than formally defined, different tool implementations give different results [LPS+23].

Static analysis tools (SAT) detect code smells efficiently [Ao21]. We use "code smell warnings" for what SATs report.

Code smells also depend on the language [BHM+19, TPB+17, SMKA17]. Python's growth raises the question of how well existing solutions fit its specifics. Python code smells can take forms that are hard to describe and detect [CCMX16, CCM+18].

AI has deeply influenced Software Engineering, and code smell detection has benefited from ML models [MBM24]. A systematic literature review [MBM24] flagged a clear need for evaluation. Performance varies a lot across models, datasets, languages, and smell types [DNPT+18]. Python results can differ from Java due to dynamic typing and the Python 2 to 3 transition [TFAL21].

The main goal of this study is to evaluate how suitable existing ML detection models are for code smells in large Python projects.

Our contributions are:

- A critical evaluation of 7 ML models and a GA algorithm for code smell detection;
- A comparison of each model's performance across ten code smells;
- The most suitable model for each smell;
- An enhanced, publicly available version of the dataset from [MBP24].

III.3.2 Study design

This chapter presents an evaluation research, following the definition from [WMMR06], assessing different intelligent methods to detect code smells in large size Python projects. In case of evaluation research, the focus is not on the novelty of the proposed technique, but on the relevance of the knowledge claimed by the evaluation. The evaluation type of research is recommended [WMMR06] when existing techniques in practice (in our case, ML/AI models) addressing the same problem (in our case, code smell detection) are investigated. The results of this type of research provides critical comparison and evidence related to these techniques using independent data and indicators and providing an independent opinion for several existing solutions.

We formulate our research questions and conduct our study to express observations regarding how the different AI models under consideration behave for the specific problem of code smells detection for Python projects.

We aim to study the suitability of the most used AI methods in detecting code smells. As methodology, our explanatory experiment, as described in [JCP08], follows the Goal-Question-Metric (GQM) approach [BCR94]:

Purpose: *Evaluate*
Issue: *the suitability*
Objective: *of ML/AI methods for code smell detection*
Context: *in Python projects*
Viewpoint: *from the perspective of performance evaluation metrics*

Our goal is further devised in the following research questions:

- **RQ13**: *Which is the performance of the ML methods for solving the problem of code smell detection?*
- **RQ14**: *Which is the suitability of ML methods for each code smell?*

The experiment is designed as follows:

- define the set of code smells to be detected using the rules from replication package
- create a dataset with the help of three tools: Radon, PySmell, Understand
- use the extracted metrics in order to meet thresholds for selected code smells
- train and evaluate the chosen AI/ML models (DT, SVM, NB, LSTM, RF, CNN, kNN, GA) and produce the results
- compare results in terms of their performance and suitability.

III.3.3 Analysis, Evaluation and Results

For all employed models, we utilized two dataset configurations: 80% for training and 20% for testing. The sole exception is the genetic algorithm, which employed a dataset distribution of 60% for training, 20% for validation, and 20% for testing.

As previously stated, for each model we computed the values for accuracy, recall, precision and F1-Score, and compared them.

RQ13: *Which is the performance of the AI/ML methods for solving the problem of code smell detection?*

The answer to this question came from the individual evaluation of each model. A short description of the results will be presented for each experiment.

For *Decision Trees*, omitting balancing techniques resulted in higher accuracy and precision, whereas applying balancing improved recall for minority classes. For Random Forests, shallower trees consistently outperformed deeper trees in precision and F1-score, suggesting that limiting depth prevents the model from capturing noise.

The *kNN model* with $k = 5$ proved superior to $k = 10$ in precision and F1-score. In SVM experiments, the linear kernel consistently outperformed the RBF kernel across all metrics.

Gaussian Naive Bayes emerged as a strong performer for imbalanced data, providing higher precision and recall than other variants. Its probabilistic nature allowed it to handle minority classes where other models struggled.

The *LSTM* and *CNN* models generally performed better when utilizing a Sigmoid loss function. However, these complex models frequently underperformed compared to simpler instance-based or probabilistic models due to the limited size and high imbalance of the dataset.

For the *Genetic Algorithm*, increasing its iterations from 50 to 100 led to improved optimization across nearly all metrics.

Finding 12: *The analysis indicates significant variability in model performance, influenced by dataset imbalance and complexity. Simpler models generally outperform complex neural models, highlighting the effectiveness of instance-based and probabilistic approaches on limited and imbalanced datasets.*

RQ14: *Which is the suitability of ML methods for each code smell?*

In regard to this question, the findings indicate that no single model is universally optimal. *kNN* achieved the highest F1-scores for six out of ten categories, including **LongMethod** and **LongMessageChain**. On the other hand, *Naive Bayes* was the most effective for underrepresented smells like *LongScopeChaining* and *LongLambdaFunction*. Statistical validation via the Nemenyi test confirmed that while *NB* and *kNN* are highly competitive, the performance gap between them is often not

statistically significant. However, both significantly outperform deep learning approaches in this specific context.

Finding 13: *While specific AI models demonstrate strengths in detecting particular code smells, overall performance remains inadequate for reliable practical deployment due to moderate F1-scores. Issues such as dataset imbalance and limited data availability substantially impact detection accuracy, emphasizing the necessity for balanced datasets and carefully tailored model selection and configuration.*

III.3.4 Conclusions

This chapter evaluates the suitability of eight AI/ML models for code smell detection in Python projects. The purpose is twofold: to figure out the performance of the AI/ML methods and to determine their suitability for each code smell.

We created a dataset of 23 Python projects from [MBP24]. We extracted code metrics using Radon [Rub24] and Understand [ST24] and detected code smells using PySmell [Che24] to build the training set. We scanned 1359 releases, yielding a final dataset of 3,191,908 entries.

We conducted an evaluation comparing model performance and assessing how suitable each model is for detecting specific code smells. Findings indicate there is no universal solution for code smell detection. Variability in performance is affected by model configuration, dataset properties, and evaluation metrics. Genetic algorithms yielded encouraging outcomes when tuned with a larger number of iterations.

Regarding effectiveness across code smell types, simpler models, such as kNN and NB, frequently surpass more complex approaches like LSTM and CNN. However, outcomes are strongly affected by uneven dataset distribution and specific model configurations.

The results are valuable for both researchers and practitioners, as focusing on the code quality of Python projects is a relevant topic. For researchers, the assessment supports the identification of effective detection approaches and opens up avenues for future investigations. For practitioners, it offers practical guidance on choosing and configuring models.

Part IV

Conclusions and future work

IV.1 Final Conclusions

All the chapters up until this point were meant to emphasize the importance of software quality in the software engineering domain. Throughout the course of this thesis, I went from building tools for data extraction to building datasets and running empirical experiments on commit classification, developer behavior, issue detection, ASTs, and defect prediction in order to generate more knowledge in the literature. All this knowledge is meant to help developers write better code, pay attention to software defects, and follow overall recommendations for their codebases. It is also intended to assist researchers in better analyzing this domain, extrapolating from our findings, building upon our datasets, and reaching new heights when it comes to software quality.

IV.1.1 Main Contributions and Impact

In the next paragraphs, I will underline my contributions and explain how each piece fits with each other in order to build the complete research we stand upon.

- PART 1: Foundations - In this part, we focused on the theoretical background upon which we built our studies, the application that allowed us to build the dataset, and the dataset itself. This paves the way for the studies in the later chapters.
  - In the **PyDs Builder**[Ber25] chapter, my contribution was **building the mechanism and the tool itself, with all that it implies.** The application artifact is publicly available in [Ber24].
  - In the **Python Software Quality Dataset**[MBP24] chapter, my contribution was the creation of the dataset artifact [BMP24].

  All datasets (**??**) are publicly available, under CC by 4.0 License, respecting good practices recommended by the research community, encouraging further investigation of our data. The tool that we developed allowed us to construct datasets in a smoother way and also facilitate the creation of new datasets by other researchers, specific to their needs.

- PART 2: Code Issues - In this part, we utilized the dataset previously created in order to run empirical experiments, such as longitudinal analysis of the evolution of SonarQube issues, CCS commit classification and developer intent, the analysis of commit burstiness and how it relates to commit classification, and the attempt to translate SonarQube rule violations between Python and JavaScript. The first three studies are chained, especially by the datasets generated by each study that are used successively, as can be observed in the diagram, while the last one is standalone.
  - In the **Longitudinal Study** chapter [BMM26b], my contribution was **the creation of the dataset artifact [BMM26a] and the inferential statistics analysis for the third research question**
  - In the **Commit Classification** chapter [BMPM25], my contribution was the **creation of the dataset artifact [BMMP25], as well as the statistical analysis and results for the first two research questions**. I have also evaluated and run the LLM model for CCS detection.

  - In the **Commit Burstiness** chapter, my contribution was the **creation of the dataset artifact [BMM25] and the analysis and results for the three research questions**. One of the important steps in obtaining the results was to design and implement the algorithm that allowed us to link SonarQube issues to the commit that induced them, between two consecutive releases.
  - The last piece of this part, namely the AST article, used a different approach in order to validate whether code smells can be transferred inter-language. My contribution here was a part of the dataset created, the AST analysis interpretation, and the manual investigation of three rules (S3776, S3558, S1763).

  Our work brings empirical evidence related to the composition and distribution of SonarQube issues, the long term evolution of issues and investigation of developer intent and commit bursts, together with AST based analysis between different programming languages. For the practitioners community, the in-depth analysis of the projects in a multi-dimensional manner linking quality issues to commits. Our work can help raise awareness regarding the prevalence of code smells, bugs, or vulnerabilities that developers should prioritize fixing.

- PART 3: Code Smells - This part was more focused on predicting code smells and refactoring through ML models. My role here was more about mining and preparing the data to be used for training, as my co-authors ran the ML experiments.
  - In the SLR [MBM24] article, my contribution was the creation of the dataset and designing the methodology.
  - In the Maintainability Prediction article [BM23], the next tile in the diagram, **my contribution was the SonarQube processing, analysis, and interpretation of the three applications**.
  - In the third and last study of part 3, the focus changes to the evaluation of predictive ML methods for code smell detection. My contribution here was **methodology design and dataset creation**, feeding the data into the ML models.

  The contribution of this part represent evalution type of research for the use of ML methods for software engineering problems, such as code smell detection and refactoring prediction. From a practitioner perspective, it opens opportunity to build specialized tools based on our results.

All parts of this thesis can be aggregated into a single finding: **software quality in open-source projects is predominantly a maintainability problem, driven by code smells, shaped by developer behavior, and addressable through a combination of static analysis and ML/LLM models.** It was presented that code smells dominate the issue landscape across all temporal dimensions: from individual commits to years-long evolutions. Developer intent, as captured through commit classification, reveals that feature additions and refactoring are the primary carriers of new issues; yet, burst periods of development do not worsen this effect. These findings, grounded in over 90 projects and approximately 90,000 classified commits, form a strong argument for integrating commit-aware, static-analysis driven quality feedback into everyday development workflows. As we have shown with the LLM-based commit classification (76% F1-score), Large Language Models can already augment the software quality pipeline by automating tasks that were previously manual, such as categorizing developer intent at scale. As these models continue to improve, their role in issue detection, prioritization, and even automated refactoring recommendations is likely to expand further.

From a methodological perspective, the thesis encompasses different empirical methods in Software Engineering [Pau21]: systematic literature review [MBM24], case studies [BM23], experiments[BMPM25, LMBM, VAMM], and longitudinal evaluations [BMM26b].

From a research type perspective [PFMM08], we have performed: solution [Ber25, MBP24], validation [RDPM], evaluation [BMM26b, VAMM, BM23], and exploratory [BMPM25, LMBM].

IV.1.2 Future Work

Based on the results that we obtained so far, and the expertize that we accumulated, we foresee two important directions that can guide the future work:

The use of Generative AI tools in solving specific software engineering problems, based on the tools and datasets that we own.

- The PyDs tool can be expanded to support data extraction from different projects, to execute different SATs and extract data from them. Another thread here is the implementation of benchmarking systems for LLMs, in order to assess how some models work on different detections, predictions or recommendations when it comes to software quality.
- The dataset can be augmented with issues in different languages, such as Ruby, Rust and Go, some that are not so covered in the literature, such as Java. It can also target projects in different software domains. We can utilize the LLMs in order to generate synthesize data for future experiments.
- The CCS commit classification analysis can leverage newer LLMs to improve upon the 76% F1-score achieved with CodeLlama, enabling finer-grained and more reliable analysis of developer intent.
- The AST study can be augmented by exploring richer code representations, such as program dependence graphs or code embeddings, that capture semantic similarities where pure ASTs failed. In this way, we can introduce LLMs in order to try to capture the similarities, to explore whether this venue could bring positive results.

The analysis of developer behavior and commit patterns across open-source systems.

- The longitudinal study can be replicated in other languages, to determine whether the dominance of code smells and their temporal patterns are language-specific or universal.
- The Commit Burstiness analysis should examine specific issue types and integrate qualitative insights to better understand the impact of commit patterns on software quality.
- Augment existing results with qualitative analysis of developer behavior (surveys and interviews).

The general direction should be to package the classification and analysis pipelines into CI/CD-integrated tools that give developers real-time, commit-level quality feedback.

Bibliography

[AAK+20] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: A systematic literature review. *CoRR*, abs/2007.02194, 2020.

[AAM16] Saleh Almugrin, Waleed Albattah, and Austin Melton. Using indirect coupling metrics to predict package maintainability and testability. *Journal of Systems and Software*, 121:298–310, 2016.

[ABJ10] Erik Arisholm, Lionel C. Briand, and Elvira B. Johannessen. An empirical study on the relationship between software maintainability and bug-proneness. In *2010 IEEE International Symposium on Software Metrics (METRICS)*. IEEE, 2010.

[ADA18] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[AdM+23] Andrei Ahonen, Marea de Koning, Tyrone Machado, Reza Ghabcheloo, and Outi Sievi-Korte. An exploratory study of software engineering in heavy-duty mobile machine automation. *Robotics and Autonomous Systems*, 165:104424, 2023.

[ALRT19] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. Are architectural smells independent from code smells? an empirical study. *Journal of Systems and Software*, 154:139–156, 2019.

[ALT+21] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. Pyref: Refactoring detection in python projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141, 2021.

[Ao21] Paris Avgeriou and o. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software*, PP, 05 2021.

[BASB17] Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 251–262, 2017.

[BCH+05] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 27–36, 2005.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. 1994.

[BCRP20] Olimar Borges, Julia Couto, Duncan Ruiz, and Rafael Prikladnicki. How machine learning has been applied in software engineering? In *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, pages 306–313. INSTICC, SciTePress, 2020.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Ber16] Robert A. Berenson. If you can't measure performance, can you improve it? *JAMA Forum*, 2016. Discusses the oft-quoted management adage and its attribution issues.

[Ber24] Liviu Marian Berciu. Pyds builder, May 2024.

[Ber25] L.-M. Berciu. Pydsbuilder - a dataset builder written in python django. *Studia Universitatis Babeș-Bolyai Informatica*, 69(2):5–22, 2025.

[BHM+19] Emery Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems*, 41:1–24, 10 2019.

[BLRS20] Maria Teresa Baldassarre, Valentina Lenarduzzi, Simone Romano, and Nyyti SaarimÃ€ki. On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. *Information and Software Technology*, 128:106377, 2020.

[BM23] L. Berciu and V. Moldovan. Software maintainability and refactorings prediction based on technical debt issues. *Studia Universitatis Babeș-Bolyai Informatica*, 68(2):22–40, 2023.

[BMM25] Liviu Marian Berciu, Vasilica Moldovan, and Simona Motogna. Dataset for "exploring the impact of commit burstiness on software quality", Jul 2025.

[BMM26a] Liviu Marian Berciu, Simona Motogna, and Arthur-Jozsef Molnar. Dataset for "a long-term exploratory study of source code quality issues in open-source python projects" article, Jan 2026.

[BMM26b] Liviu-Marian Berciu, Simona Claudia Motogna, and Arthur-Jozsef Molnar. A long-term exploratory study of source code quality issues in open-source python projects. *Software Quality*

Journal, 34:9, 2026.

[BMMP25] Liviu Marian Berciu, Arthur-Jozsef Molnar, Simona Motogna, and Oskar Picus. Replication package for exploring the relation between source code commit information and sonarqube issues, May 2025.

[BMP24] Liviu Marian Berciu, Vasilica Moldovan, and Rares Patcas. The python software quality dataset, May 2024.

[BMPM25] Liviu-Marian Berciu, Simona Motogna, Oskar Picus, and Arthur-Jozsef Molnar. Exploring the relation between source code commit information and sonarqube issues. *Procedia Computer Science*, 270:841–850, 2025. 29th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2025).

[Buj24] M.A. Bujang. An elaboration on sample size determination for correlations based on effect sizes and confidence interval width: a guide for researchers. In *Restorative Dentistry and Endodontics*, volume 49, page e21, 2024. Published 2024 May 2.

[CBMP14] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. On the impact of refactoring operations on code quality metrics. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, page 456–460, USA, 2014. IEEE Computer Society.

[CBZK19] Angelos Chatzimparmpas, Stamatia Bibi, Ioannis Zozas, and Andreas Kerren. Analyzing the evolution of Javascript applications. In *ENASE*, pages 359–366, 2019.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[CCM+18] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. Understanding metric-based detectable smells in python software: A comparative study. *Information and Software Technology*, 94:14–29, 2018.

[CCMX16] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. Detecting code smells in python programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 18–23, 2016.

[CD10] Daniela S. Cruzes and Tore Dybå. Synthesizing evidence in software engineering research. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10. Association for Computing Machinery, 2010.

[CFI+24] Domenico Cotroneo, Alessio Foggia, Cristina Improta, Pietro Liguori, and Roberto Natella. Automating the correctness assessment of AI-generated code for security contexts. *Journal of Systems and Software*, 216:112–113, 2024.

[Che24] Zhifei Chen. Pysmell: Python code smells detection tool. `https://github.com/chenzhifei731/Pysmell/tree/master`, 2024.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Con20] Conventional Commits Specification v1.0.0. `https://www.conventionalcommits.org/en/v1.0.0/`, 2020.

[CSS13] K.K. Chaturvedi, V.B. Sing, and Prashast Singh. Tools in mining software repositories. In *2013 13th International Conference on Computational Science and Its Applications*, pages 89–98, 2013.

[DACA22] George Digkas, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The temporality of technical debt introduction on new code and confounding factors. *Software Quality Journal*, 30(2):283–305, jun 2022.

[Dat25] Datadog. What is static analysis & how does it work?, 2025.

[DLA+18] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Am-

patzoglou. How do developers fix issues and pay back technical debt in the apache ecosystem? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 153–163, 2018.

[DNPT+18] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[dPB+23] Dario Amoroso d'Aragona, Fabiano Pecorelli, Maria Teresa Baldassarre, Davide Taibi, and Valentina Lenarduzzi. Technical debt diffuseness in the apache ecosystem: A differentiated replication. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 825–833, 2023.

[dTF+19] Francisco Gomes de Oliveira Neto, Richard Torkar, Robert Feldt, Lucas Gren, Carlo A. Furia, and Ziwei Huang. Evolution of statistical analysis in empirical se research: Current state and steps forward. *JSS*, 156:246–267, 2019.

[ESK18] Amir Elmishali, Roni Stern, and Meir Kalech. An artificial intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence*, 69:147–156, 2018.

[FMNL21] Aron Fiechter, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing github issues. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 155–159, 2021.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[fSC21] International Organization for Standardization and International Electrotechnical Commission. Information technology, software measurement, software quality measurement, automated source code quality measures. https://www.iso.org/standard/80623.html, 2021.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[git] Github api. `https://docs.github.com/en/rest/`.

[Git24] GitHub, Inc. *About Releases on GitHub*, 2024.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[HBS22] Tjasa Hericko, Saša Brdnik, and Bostjan Sumak. Commit classification into maintenance activities using aggregated semantic word embeddings of software change messages. In *SQAMIA*, 2022.

[Hs23] Tjasa Hericko and Bostjan sumak. Commit classification into software maintenance activities: A systematic literature review. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1646–1651, 2023.

[JCP08] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer London, London, 2008.

[Jm12] Ronald Jabangwe and Darja mite. An exploratory study of software evolution and quality: Before, during and after a transfer. In *2012 IEEE Seventh International Conference on Global Software Engineering*, pages 41–50, 2012.

[JSM21] Joselito Mota Jr., Railana Santana, and Ivan Machado. Grumpy: an automated approach to simplify issue data analysis for newcomers. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, SBES '21, page 33–38, New York, NY, USA, 2021. Association for Computing Machinery.

[KC07] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[KCA21] Stratos Kourtzanidis, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. Reposkillminer: identifying software expertise from github repositories using natural language processing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1353–1357, New York, NY, USA, 2021. Association for Computing Machinery.

[KK15] Flavius Kehr and Tobias Kowatsch. Quantitative longitudinal research: A review of is literature,

and a set of methodological guidelines. 09 2015.

[KRS13] Carsten Kolassa, Dirk Riehle, and Michel A. Salim. The empirical commit frequency distribution of open source projects. In *Proceedings of the 9th International Symposium on Open Collaboration*, WikiSym '13, page 1–8. ACM, August 2013.

[KS07] Georg von Krogh and Sebastian Spaeth. The open source software phenomenon: Characteristics that promote research. *The Journal of Strategic Information Systems*, 16(3):236–253, 2007.

[LAAZ23] Afshan Latif, Farooque Azam, Muhammad Waseem Anwar, and Amina Zafar. Comparison of leading language parsers–antlr, javacc, sablecc, tree-sitter, yacc, bison. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*, pages 7–13. IEEE, 2023.

[LAL15] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[LB24] R. Patcas L. Berciu, V. Moldovan. The python software quality dataset. `https://figshare.com/s/ca638816c59abf2195d9`, 2024.

[Lee16] Dong Kyu Lee. Alternatives to p value: confidence interval and effect size. *Korean journal of anesthesiology*, 69(6):555–562, 2016.

[Leh80] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LLCW+24] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. 33(5), 2024.

[LLHT20] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. Are sonarqube rules inducing bugs? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–511, 2020.

[LLTH19] Valentina Lenarduzzi, Francesco Lomio, Davide Taibi, and Heikki Huttunen. On the fault proneness of sonarqube technical debt violations: A comparison of eight machine learning techniques. *CoRR*, abs/1907.00376, 2019.

[LMBM] Vasilica Andreea Moldovan Liviu Marian Berciu and Simona Motogna. Exploring the impact of commit burstiness on software quality. In *QUATIC 2025*.

[LML22] Francesco Lomio, Sergio Moreschini, and Valentina Lenarduzzi. A machine and deep learning analysis among sonarqube rules, product, and process metrics for fault prediction. *Empirical Software Engineering*, 27(7):189, 2022.

[LPS+23] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, 198, 2023.

[LS08] G. A Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 39–44, 2008.

[LST19a] Valentina Lenarduzzi, Nyyti Saarimaki, and Davide Taibi. On the diffuseness of code technical debt in java projects of the apache ecosystem. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 98–107, 2019.

[LST19b] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'19. ACM, September 2019.

[Mar04] Radu Marinescu. An empirical study of the relationship between code smells and refactoring. *Empirical Software Engineering*, 9(4):429–462, 2004.

[MBM24] Simona Motogna, Liviu-Marian Berciu, and Vasilica-Andreea Moldovan. Artificial intelligence methods in software refactoring: A systematic literature review. In *50th Euromicro Conference Series on Software Engineering and Advanced Applications*, 2024.

[MBP24] Vasilica-Andreea Moldovan, Liviu-Marian Berciu, and Rares-Danut Patcas. The python software quality dataset. In *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 395–398, 2024.

[McH13] Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

[MM20a] Arthur-Jozsef Molnar and Simona Motogna. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[MM20b] Arthur-Jozsef Molnar. and Simona Motogna. Longitudinal evaluation of open-source software maintainability. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 120–131. INSTICC, SciTePress, 2020.

[MM21] Arthur-Jozsef Molnar and Simona Motogna. A study of maintainability in evolving open-source software. In Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 261–282, Cham, 2021. Springer International Publishing.

[MM22] Arthur-Jozsef Molnar. and Simona Motogna. Characterizing technical debt in evolving open-source software. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 174–185. INSTICC, SciTePress, 2022.

[MMD11] Laurie McLeod, Stephen MacDonell, and Bill Doolin. Qualitative research on software development: A longitudinal case study methodology. *Empirical Software Engineering*, 16:430–459, 08 2011.

[Mol20] Arthur-Jozsef Molnar. Collection of technical debt issues in freemind, jedit and tuxguitar open source software. 7 2020.

[MP12] Vishal Midha and Prashant Palvia. Factors affecting the success of open source software. *Journal of Systems and Software*, 85(4):895–905, 2012.

[MV00] Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000.

[MVS23] Simona Motogna, Andreea Vescan, and Camelia Serban. Empirical investigation in embedded systems: Quality attributes in general, maintainability in particular. *J. Syst. Softw.*, 201, 2023.

[NCK+19] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. A longitudinal study of identifying and paying down architecture debt. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 171–180, 2019.

[NGM+19] Malanga Ndenga, Ivaylo Ganchev, Jean Méhat, Franklin Wabwoba, and Herman Akdag. Performance and cost-effectiveness of change burst metrics in predicting software faults. *KIS*, 60, 07 2019.

[NMB10] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. Change bursts as defect predictors. In *ISSRE*, pages 309–318, 2010.

[ODA+15] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi, and R. Tonelli. The jira repository dataset: Understanding social aspects of software development. In *Proceedings of the 11th international conference on predictive models and data analytics in software engineering*, pages 1–4, 2015.

[OH92] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, Nov 1992.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pau21] Paul Ralph (ed.). Acm sigsoft empirical standards for software engineering research, version 0.2.0, 2021.

[PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, page 68–77. BCS Learning & Development Ltd., 2008.

[PMB23] Manuela Petrescu, Simona Motogna, and Liviu Berciu. Women in scrum master role: Challenges and opportunities*. pages 49–55, 05 2023.

[Pos23] PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. `https://www.postgresql.org/`, 2023.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Pret-

tenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PyL23] PyLint Contributors. *PyLint*. PyLint, 2023.

[Ral21] Paul Ralph. Acm sigsoft empirical standards released. *SIGSOFT Softw. Eng. Notes*, 46(1):19, February 2021.

[RB22] Paul Ralph and Sebastian Baltes. Paving the way for mature secondary research: the seven types of literature review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1632–1636. Association for Computing Machinery, 2022.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[RDPM] Liviu Marian Berciu Rares Danut Patcas, Oskar Pikus and Simona Motogna. Ast similarity of code quality issues in python and javascript. submitted to EASE2026.

[RG19] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[RPS+23] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. A comprehensive evaluation of szz variants through a developer-informed oracle. *Journal of Systems and Software*, 202:111729, 2023.

[RTLR18] M. M. Rosli, E. Tempero, and A. Luxton-Reilly. Evaluating the quality of datasets in software engineering. *Advanced Science Letters*, 24(10):7232–7239, 2018.

[Rub24] Micheal Rubik. Radon: Measure cyclomatic complexity, maintainability index, and halstead metrics in python code. `https://radon.readthedocs.io/`, 2024.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery.

[SDC24] Diego S Sarafim, Karina V Delgado, and Daniel Cordeiro. Detecting code smells in JavaScript: An annotated dataset for software quality analysis. In *Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 313–322. SBC, 2024.

[SGK+09] Diomidis Spinellis, Georgios Gousios, Vassilios Karakoidas, Panagiotis Louridas, Paul J. Adams, Ioannis Samoladas, and Ioannis Stamelos. Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science*, 233:5–28, 2009.

[SK21] T. Sharma and M. Kessentini. Qscored: A large dataset of code smells and quality metrics. In *2021 IEEE/ACM 18th MSR*, pages 590–594, 2021.

[SMKA17] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. An empirical study of code smells in javascript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305, 2017.

[Son24a] SonarQube. Sonarqube documentation, 2024.

[Son24b] SonarQube. What is clean code, 2024.

[Son25a] SonarSource. How new issues are identified, 2025. SonarQube Server 10.3 User Guide.

[Son25b] SonarSource. Managing issues, 2025.

[Sonnd] SonarSource. Cyclomatic complexity: developer's guide, n.d.

[Sri17] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017.

[SSM05] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[ST24] Inc. Scientific Toolworks. Understand: Static analysis and code visualization tool. `https://www.scitools.com/`, 2024.

[SW65] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples).

Biometrika, 52(3/4):591–611, 1965.

[Swa76] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.

[TAA+18] David Trafimow, Valentin Amrhein, Corson N Areshenkoff, Carlos J Barrera-Causil, Eric J Beh, Yusuf K Bilgiç, Roser Bono, Michael T Bradley, William M Briggs, Héctor A Cepeda-Freyre, et al. Manipulating the alpha level cannot cure significance testing. *Front. Psychol.*, 9:699, May 2018.

[TDPT+21] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. How software refactoring impacts execution time. *ACM Trans. Softw. Eng. Methodol.*, 31(2), 2021.

[TEHG23] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes. *Empirical Software Engineering*, 28(2):30, 2023.

[TFA20] Jie Tan, Daniel Feitosa, and Paris Avgeriou. An empirical study on self-fixed technical debt. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery.

[TFA23] Jie Tan, Daniel Feitosa, and Paris Avgeriou. The lifecycle of technical debt that manifests in both source code and issue trackers. *Information and Software Technology*, 159:107216, 2023.

[TFAL21] Jie Tan, Daniel Feitosa, Paris Avgeriou, and Mircea Lungu. Evolution of technical debt remediation in python: A case study on the apache software ecosystem. *Journal of Software: Evolution and Process*, 33(4):e2319, 2021. e2319 smr.2319.

[Tho21] Patrick Thomson. Static analysis: An introduction. *ACM Queue*, 19(4):29–41, 2021.

[TPB+17] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Softw. Eng.*, 43(11):1063–1088, nov 2017.

[Uni] University of Waikato. Weka - data mining software in java. `https://sourceforge.net/projects/weka/`.

[VAMM] Liviu Marian Berciu Vasilica Andreea Moldovan and Simona Motogna. A comparative evaluation of intelligent methods for code smell detection in python. submitted to Scientific Annals of Computer Science.

[VSM13] B. Vasilescu, A. Serebrenik, and T. Mens. A historical dataset of software engineering conferences. In *2013 10th MSR*, pages 373–376. IEEE, 2013.

[VSPL18] Roberto Verdecchia, Ren\'e Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. In Birgit Penzenstadler, Steve Easterbrook, Colin Venters, and Syed Ishtiaque Ahmed, editors, *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability*, volume 52 of *EPiC Series in Computing*, pages 365–383. EasyChair, 2018.

[Wan18] Divanshi Priyadarshni Wangoo. Artificial intelligence techniques in software engineering for automated software reuse and design. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)*, pages 1–4, 2018.

[WDS16] Michael Wahler, Uwe Drofenik, and Will Snipes. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 493–501, 2016.

[WM18] Neil Walkinshaw and Leandro Minku. Are 20% of files responsible for 80% of defects? In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 1–10. Association for Computing Machinery, 2018.

[WMMR06] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. *Requir. Eng.*, 11:102–107, 03 2006.

[WNLB22] Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Quick remedy commits and their impact on mining software repositories. *ESE*, 27(1), January 2022.

[XF14] Qi Xuan and Vladimir Filkov. Building it together: synchronous development in oss. In *Proceedings 36th ICSE*, page 222–233, 2014.

[ZZQL25] Qunhong Zeng, Yuxia Zhang, Zhiqing Qiu, and Hui Liu. A First Look at Conventional Commits Classification . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 127–139. IEEE Computer Society, 2025.