Received: 24 March 2020 | Revised: 2 September 2020 | Accepted: 8 September 2020
DOI: 10.1002/smr.2319

RESEARCH ARTICLE - EMPIRICAL

Journal of Software: Evolution and Process

WILEY

Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem

Jie Tan[1] | Daniel Feitosa[1,2] | Paris Avgeriou[1] | Mircea Lungu[3]

[1]Faculty of Science and Engineering, University of Groningen, Groningen, The Netherlands

[2]Data Research Centre, University of Groningen, Groningen, The Netherlands

[3]Computer Science Department, IT University of Copenhagen, Copenhagen, Denmark

Correspondence
Jie Tan, Faculty of Science and Engineering, University of Groningen, Groningen, The Netherlands.
Email: jessie_tanjie@hotmail.com

Funding information
ITEA3; RVO, Grant/Award Number: 17038 VISDOM

Abstract

In recent years, the evolution of software ecosystems and the detection of technical debt received significant attention by researchers from both industry and academia. While a few studies that analyze various aspects of technical debt evolution already exist, to the best of our knowledge, there is no large-scale study that focuses on the remediation of technical debt over time in Python projects—that is, one of the most popular programming languages at the moment. In this paper, we analyze the evolution of technical debt in 44 Python open-source software projects belonging to the Apache Software Foundation. We focus on the type and amount of technical debt that is paid back. The study required the mining of over 60K commits, detailed code analysis on 3.7K system versions, and the analysis of almost 43K fixed issues. The findings show that most of the repayment effort goes into testing, documentation, complexity, and duplication removal. Moreover, more than half of the Python technical debt is short term being repaid in less than 2 months. In particular, the observations that a minority of rules account for the majority of issues fixed and spent effort suggest that addressing those kinds of debt in the future is important for research and practice.

KEYWORDS

Apache Software Foundation, software ecosystems, software evolution, technical debt repayment

1 | INTRODUCTION

Technical debt (TD) is a metaphor used to describe a trade-off between the short-term benefits of 'cutting corners' in software development and the long-term sustainability of a software system.[1] Incurring technical debt through these shortcuts can bring benefits in terms of time and/or resources (e.g., shorter time to market and less effort required to implement a feature). However, this expediency has a detrimental effect on the future maintainability and evolvability of the system, as it becomes increasingly harder to make changes.[2,3] This hardening translates into an additional effort for developers and, in turn, costs that can even become prohibitive to organizations. If not repaid, TD can lead to severe quality problems, unexpectedly cost overruns and substantial financial loss on software maintenance.[4] It can even lead to crisis points when entire components or the system need to be replaced.[5]

Although the technical debt community has reached a basic consensus on the concepts around TD,[6] we still need to understand better the realization and impact of TD in practice. One important aspect of that is how TD evolves over time and how it is paid back. Obtaining such knowledge can be useful, among others, for urging refactoring activities when the amount of TD increases fast or for preventing the accumulation of new TD if repayment strategies are too costly.

Several studies that analyzed technical debt evolution have primarily focused on software written in Java.[3,7,8] Few other languages besides Java have been studied in the context of TD, including C#[9] and C/C++.[10] The Tiobe Index[*] ranks Java as the most popular programming language, with C coming in second place and Python in third. At the time of writing this paper—the beginning of 2020—GitHub contains more than 1M open source Python projects and almost 1M involved developers[†].

Despite the huge importance of Python in current software engineering practice, reflected also in several recent empirical studies,[11,12] we are not aware of any studies that focus on the evolution of technical debt in Python. This raises the question of whether and to what extent the results of the existing studies on TD evolution are applicable to Python. If we compare, for example, Python and Java, although they are both interpreted languages, they represent two different schools of thought: Java is statically typed while Python is served by a dynamic type system. Generally, developers have to spend extra effort on software maintenance[13] and software quality improvement[14] in Python, because Python code is more change-prone due to having a higher number of dynamic features.[13] In addition, Python code containing dynamic features is inserted or updated more frequently when fixing bugs.[15]

Furthermore, Python code is characterized by a unique kind of technical debt: Python has two major versions in use (i.e., Python 2 and Python 3), and developers often add workarounds and mix code from the two versions. Even though there is consensus that the latest version should be the de facto standard, it is common to still use Python 2 libraries because old code may rely on them. This kind of backward compatibility results in the usage of redundant and complicated features of the language. To make matters worse, as of 1 January 2020, Python 2 is no longer maintained[‡]; this will cause security vulnerabilities.

Given these unique characteristics of Python, studying the TD evolution of Python programs can guide developers specifically in managing TD for this language. It can also highlight the differences with other languages, for example, Java; this can be used to further help developers, team leaders, or teams that transition from Java to Python projects or work in both languages.

In this study, we investigate the evolution of technical debt remediation in Python; in other words, how technical debt is paid back along time. Our scope is TD at the source code level: We use static source code analysis to examine how TD issues are fixed. Specifically, we (1) provide an overview of the fixing rates for a number of projects and investigate whether the fixing rates of issues differ among projects (as each project has a different size and number of issues); (2) examine the prevalence of the various types of issues; (3) analyze the fixing rate of the most frequent issues; (4) investigate the remediation effort for the various issues; and (5) look at the remediation time for various types of debt.

Furthermore, to understand how the aforementioned unique characteristics of Python affect TD remediation, we compare our findings with a similar study in Java.[3] We expect to observe several differences between Python and Java due to the aforementioned aspects: static versus dynamic type system and transition from Python 2 to Python 3. In addition, we highlight technical debt that is language-dependent and further discuss the possible reasons for the differences, both in terms of Python features and the transition from Python 2 to Python 3.

Our findings show some similarities between Python and Java; 20–30% of TD issues have a great chance to be fixed in large projects, while smaller projects have lower fixing rates. Furthermore, issues related to duplicated code require the most remediation effort, and the majority of issues with the highest fixing rate are language-specific (i.e., either Python-specific or Java-specific). In addition, the majority of issues are fixed relatively quickly while a minority of issues can live for a long time. Finally, issues that pertain to the same type of TD (e.g., design debt or defect debt) tend to have similar survival times.

On the other hand, we found a considerable amount of debt repayment that is specific to Python. For example, test and documentation debt account for almost half of all the fixes and the majority of the remediation effort, while significant debt repayment is related to changes between major versions of the Python interpreter.

These findings can serve both researchers and practitioners: Researchers can benefit from the knowledge in future approaches for automated and semi-automated debt remediation; practitioners can consider the highlighted findings and suggestions as a starting point for discussions about establishing their own practices and guidelines.

The remainder of this paper is structured as follows. Section 2 presents our study design, elaborating on the research questions, as well as data collection and analysis. Section 3 reports the results of our study, and Section 4 offers a discussion of the results. Section 5 discusses the threats to the validity of our study, and Section 6 elaborates on the dataset and the replication package. After the discussion of related work in Section 7, Section 8 concludes the paper and outlines directions for future work.

2 | STUDY DESIGN

In this section, we describe the design of our study. We first state our goal and elaborate on the derived research questions. Then, we present and justify the population of the study (i.e., the selected cases). Next, we discuss the variables and procedures for the data collection, that is, technical debt identification. Finally, we describe the analysis method for each question and subquestion.

[*]One of the best known indexes of programming languages popularity, https://www.tiobe.com/tiobe-index/, visited January 2020.
[†]https://github.com/search?l=Python&q=language%3APython&type=Repositories
[‡]https://www.python.org/doc/sunset-python-2

2.1 | Objective and research questions

In this section, we follow the Goal-Question-Metric (GQM) approach[16] to describe the goal of our study. The GQM is a measurement paradigm based on three levels: conceptual, operational, and quantitative. The conceptual level (i.e., the Goal) is defined with respect to the object of study, the purpose, the focus, the stakeholders, and the context. The operational level (i.e., the Question) regards a set of questions to describe the assessment or achievement of the goal that is defined at the conceptual level. Finally, the quantitative level (i.e., the Metric) regards a set of metrics that can be measured to address each question of the operational level in a measurable way. The goal of our study is to 'analyze software systems written in Python for the purpose of understanding technical debt remediation with respect to its evolution and comparison with Java, from the point of view of software developers in the context of object-oriented open source software.' This objective is further refined in terms of the following research questions:

RQ_1 How does TD remediation evolve in Python projects?

- **$RQ_{1.1}$** How does the issue fixing rate vary for different projects?
- **$RQ_{1.2}$** What is the fixing prevalence for various issues?
- **$RQ_{1.3}$** How does the fixing rate vary for different issues?
- **$RQ_{1.4}$** How is the effort of debt repayment distributed over different issues?
- **$RQ_{1.5}$** After how much time is technical debt paid back?

RQ_2 How does the evolution of TD remediation in Python projects compare with that of Java projects?

RQ_3 How are the differences between Python and Java explained?

- **$RQ_{3.1}$** How are Python features associated with the explanations?
- **$RQ_{3.2}$** How is the transition from Python 2 to Python 3 associated with the explanations?

This is one of the first studies to examine TD in Python, and thus, the answers to RQ_1 have great value to both researchers and practitioners. An overview of TD remediation in practice can inform researchers about the generalization of existing findings in scientific literature. For example, it can provide evidence about to what extent remediation strategies at source code level are common among open-source projects or even among object-oriented languages; this may, in turn, drive the proposal of guidelines or development of tools. Furthermore, practitioners can use the results to understand the TD repayment in their own projects. For example, a development team can use the findings to discuss how much TD related to source code issues they expect to incur and how much they plan to pay back in the near future.

We note that research subquestions $RQ_{1.1}$–$RQ_{1.5}$ are the same as those posed by Digkas et al[3] in a study to examine TD remediation in Java projects. We used the same questions in order to be able to address RQ_2, that is, to compare the evolution of TD remediation between Python and Java. Answering RQ_2 thus allows to discuss and highlight the differences and similarities with Java. The answer to RQ_2 is particularly relevant to researchers and practitioners that are experienced with Java development. Practitioners that work on both languages can interpret the differences in TD remediation and better understand TD analysis output. Researchers that aim at extending or fork their tools to work on Python can use our findings to inform their decisions and consider potential changes on how to interpret the output.

Identifying the differences and similarities between the two languages raises the question of how they can be explained. We particularly focus on differences as these can provide important lessons. The answer to RQ_3 can help researchers understand the impact of Python features on technical debt and further consider which features could lead to more debt in Python projects. Furthermore, practitioners can use the results to improve the cost estimation of their debt, for example, when planning to migrate a project from Python 2 to Python 3 or re-implement a component in Python 3.

2.2 | Cases selection

The projects that we use for studying debt evolution and, in particular, TD repayment strategies are all the Python projects of the Apache Software Foundation ecosystem[§] (hereafter referred to as the Apache ecosystem or the ASF ecosystem). We made this choice for several reasons. (1) The Apache ecosystem contains a diverse set of Python projects of various sizes (up to 1000 KLOC), activity (up to 10K commits to master), and domains. (2) Using the ASF ecosystem results in an implicit quality filter for the analyzed projects: Every project is managed by a self-selected team of technical experts and is accepted in the foundation only after an initial incubation process[¶]. (3) The ecosystem has been used before in

[§]As not all projects are tightly integrated, opinions regarding the terminology may differ. We use the term ecosystems to maintain consistency with Java study[3] and Bavota et al.[17]
[¶]https://incubator.apache.org/policy/process.html

the study of Digkas et al. for technical debt repayment in Java,[3] the use of Python projects that are part of the same organization increases the comparability of the results in the two papers.

The GitHub organization of the ASF contains projects in more than 30 programming languages. For the purpose of our study, we used the GitHub programming language filter and selected all the 44 Python projects without filtering them by any further criterion. The majority of the projects have a long history of commits, which ensures that the evolution of technical debt symptoms can be followed over an extensive period.[8] In order to focus on the complete ecosystem, we decided to include in our study also the minority of projects with a small number of commits and thus a short history.

Table A1 shows that these projects present considerable variation in their characteristics, that is, their number of commits, their size as measured by counting the source lines of code (SLOC) and the number of classes, their age (in the number of days the project has existed), and the contributors. As shown in the Table A1, these projects cover a wide range of different sizes, domains, survival times, and the number of contributors, which could strengthen the external validity.[18]

2.3 | Technical debt identification

One important step for efficiently monitoring the evolution of TD is the selection of a tool that will be able to measure various aspects of technical debt, as accurately as possible. In the state-of-research and -practice, one can identify several tools and approaches for measuring technical debt. To detect the evolution and remediation of technical debt in the subject systems, we rely on a third-party detection tool: SonarQube, an open-source code-quality measuring and management tool.[19] The tool was selected for three main reasons: (1) its broad usage for estimating technical debt, both in academic research studies[3,7,20,21] and in industry (being used by more than 1000 companies[#]); (2) its capacity to perform multiversion analysis and thus track the evolution and repayment of technical debt over time; and (3) the fact that the tool is based on the SQALE method,[22,23] which has been published and evaluated academically.[24-27] Of course it is not a perfect solution for measuring TD, as a perfect solution does not exist; we expand on the limitations of using SonarQube in the Threats to Validity Section.

This section briefly introduces the main terminology and concepts that are specific to SonarQube and are critical for understanding the remainder of the paper. The tool analyzes source code to detect code smells, vulnerabilities, and bugs and assesses/calculates technical debt as the time estimated to fix these issues.

SonarQube uses a set of **rules**, representing desirable code related practices, which, when absent, introduce technical debt. During the analysis of a project, SonarQube creates a new **issue** every time a piece of code breaks one of the rules. The rules are based on well-known sources of documented bugs and vulnerabilities such as CERT and CWE[‖], as well as the Python Enhancements Proposal[**] (PEP), which are standards largely adopted and enforced by the Python community.

2.3.1 | Multiversion analysis

One limitation of SonarQube (and all the other equivalent tools that we have investigated) is that the evolutionary analysis is not incremental.[28] This means that a new version (e.g., originated from a new commit) is fully parsed and analyzed, even if only a single line in a single file is changed. This imposes constraints on evolution analysis because it becomes prohibitive to analyze all the commits of a system. It is only recently that researchers have started proposing models of source code that are effective at incremental parsing and modeling of software, but no off-the-shelf tools make use of them yet.[29]

Considering that the majority of the analyzed Python projects have at least 187 commits (with a maximum of 10 942 commits) at the time of starting this work, one must make a trade-off in analyzing a limited but sufficient number of revisions of every system. In this study, we choose a frequency of one week;that is, we analyze weekly versions of each project, just as it was done in the Java study we use for our comparison. Furthermore, we only analyze the versions on the master branch of the studied systems.

2.3.2 | Issue classification and effort estimation

None of the project repositories of the 44 selected systems consists exclusively of Python code. By default, SonarQube detects issues for all the source code files found in a system, and when analyzing the subject systems, it produces issues in code files written in different languages (including JavaScript, Java, and XML). However, due to the focus of this study, we only analyze those issues that are found in Python files.

[#]https://www.sonarsource.com/customers/
[‖]https://www.sonarsource.com/products/codeanalyzers/sonarpython.html
[**]https://www.python.org/dev/peps/

In terms of severity of the rules, SonarQube defines four levels (in decreasing order): blocker, critical, major, and minor. However, we limit the severity level to blocker, critical, and major, because many of the minor issues are trivial and are not what developers normally think about when they talk about technical debt (e.g., 'Lines should not end with trailing whitespaces'). Moreover, the minor issues have low impact and likelihood[††] and could, therefore, bias the results. The same decision was made in the Java study,[3] and consequently, we are in a better position to compare the Python results with the Java ones.

From the total set of rules for Python in SonarQube, we detected violations of 127 rules in this study. If we exclude the minor rules among these 127, we are left with 56 blocker, critical, and major rules[‡‡]; Table A2 shows the types and their severity. Among them, 15 rules appear both in Python and Java (marked as ☕ symbol). Furthermore, we grouped the rules into five higher-level technical debt categories defined by Alves et al[30] and Li et al[1]: Code Debt, Defect Debt, Design Debt, Documentation Debt, and Test Debt. Both Alves et al[30] and Li et al[1] contain these five categories; yet, they were derived independently, and they have both been widely used in other studies.[31,32] To perform this grouping, the first and second authors classified the rules independently into the five TD categories, using the description of the rules and the definition of the categories. There were disagreements in the classification of eight rules. To assess the disagreements numerically, we estimated the inter-rater agreement using Krippendorff's alpha[33] ($\alpha = .74$).[§§] In the conflicting cases, the first and second authors discussed with the third author until they achieved consensus.

To identify and analyze issues related to version migration, the first and second authors examined the description of all 56 rules together and identified those that can be associated with the update of the Python interpreter. We also checked the source code of the issues to further confirm if the rule is migration related. Table 1 presents the ID numbers, descriptions, and explanations of these rules.

For every issue it detects, SonarQube assigns an estimate of how much time is required to resolve it based on the SQALE[¶¶] method.[22] SonarQube analyzes the source code and uses remediation functions to work out remediation costs for each issue type.[24] The tool uses two strategies to estimate technical debt for different types of issues: Some types of issues are assigned a **constant fixing time** (e.g., the issue 'Docstrings should be defined' is assigned five minutes); some other types of issues are assigned a custom fixing time according to their particular characteristics (e.g., 'Source files should not have any duplicated blocks').

2.3.3 | Fixed issue detection

When analyzing multiple revisions of a system, SonarQube tracks the issues that are fixed, and therefore, the debt that is repaid. According to the documentation of SonarQube[##], issues flow through a lifecycle, and being assigned the 'fixed' status could be due to two cases: (1) Issues have been corrected (i.e., the issues are actually fixed); (2) or the file is no longer available (removed from the project or renamed). The second case is problematic for our purpose, as it is doubtful whether the developers actually aimed at fixing the technical debt by deleting or renaming a file. Thus, we filter out the issues that are marked as fixed due to the disappearance of the file name (e.g., *deletions* and *renames*) and remove them by using the SonarQube API. The same method was used in the Java study of Digkas et al.[3]

Table 2 shows the number of issues in the analyzed systems once the Minor issues are filtered out. In total, 54 Blocker, 5327 Critical, and 89 772 Major issues have been *actually detected* in the whole history of 44 systems under study. This aligns with the Java findings, where the Major issues had the highest percentage of fixes.

2.4 | Data analysis

This study detected more than 288K issues by analyzing over 3.7K weekly commits. If we exclude the minor issues, we are left with a total of 95K issues. Next, these data are analyzed to answer the research questions as follows.

To answer $\mathbf{RQ}_{1.1}$, we calculate the issue fixing rate for each project, that is, the percentage of issues that are fixed in each project, and discuss their distribution. Because different projects have a different number of fixed issues and lines of code, we investigate the relationship between SLOC and the number of fixed issues during the evolution as well as the issue-fixing rate.

For $\mathbf{RQ}_{1.2}$, we sum up all the fixed issues grouped by rules without differentiating among projects and calculate the prevalence, that is, the percentage of issues fixed for each rule from the total number of fixes. Then we sort the prevalence rates of all the rules to get the most frequently fixed issues and classify them according to the aforementioned high-level technical debt categories.

[††]https://docs.sonarqube.org/latest/user-guide/rules/, visited in January 2020

[‡‡]Compared with 160 rules detected in the Java study

[§§]Krippendorff's inspection of the trade-offs between statistical techniques establishes that it is customary to require $\alpha \geq .80$. However, where tentative conclusions are still acceptable, $\alpha \geq .67$ is the lowest conceivable limit.[34]

[¶¶]https://blog.sonarsource.com/sqale-the-ultimate-quality-model-to-assess-technical-debt

[##]https://docs.sonarqube.org/latest/user-guide/issues/

TABLE 1 Migration-related issues

ID	Description and explanation
12	'<>' should not be used to test inequality
	The forms '<>' and '! = ' are equivalent. But in Python 2.7.3, the '<>' form is considered obsolete.
21	The 'print' statement should not be used
	The 'print' statement was removed in Python 3. The built-in function should be used instead.
31	The 'exec' statement should not be used
	The 'exec' statement was removed in Python 3. Instead, the built-in exec() function can be used.
52	Backticks should not be used
	Backticks are a deprecated alias for repr(). The syntax was removed in Python 3.0.
276	Access of nonexistent member
	Some issues are related to references to '`socket`' instead of '`SocketIO`.'
281	Syntax error
	Some issues are related to references to invalid syntax in `except` handler with a comma.
360	Undefined name
	Some issues are related to references to refactored parts of the module `urllib`.
394	Undefined variable
	Some issues are related to references to Python 2 identifiers, e.g., '`xrange`,' '`unicode`,' and '`basestring`.'
432	Mixed tabs/spaces indentation
	Indentation is rejected as inconsistent if tabs and spaces are mixed and a TabError is raised in Python 3.

TABLE 2 Number of fixed and open issues for priority type

	Fixed issues	Open issues	Fixing rate
Blocker	15	39	27.78%
Critical	2,189	3,138	41.09%
Major	40,461	49,311	45.07%

In **RQ**$_{1.3}$, we compute the issue-fixing rate for each rule to investigate whether some rules are fixed more often than others. However, because some rules only appear in a few projects, we introduce the number of projects in which violations of the rule appear as a reference value.

RQ$_{1.4}$ aims at investigating the effort required to fix the technical debt issues. To answer this research question, we sum up all the effort required to fix the issues according to the estimates provided by SonarQube. To also verify the relationship between remediation effort and frequency of fixes, we compare the rank of the rules based on effort against the ranks from **RQ**$_{1.2}$ and **RQ**$_{1.3}$.

To investigate **RQ**$_{1.5}$, we calculate and analyze the survival time of each issue. This variable is measured as the number of days between the introduction of an issue and the moment when it is fixed in the source code.

To answer **RQ**$_2$, we compare the findings of the aforementioned subquestions with those reported in the Java study. In particular, we describe the similarities and differences regarding each subquestion.

Finally, to answer **RQ**$_3$, that is, to explain the observed differences between Python and Java, we focus on the characteristics of the Python programming language (**RQ**$_{3.1}$) and the transition from Python 2 to Python 3 (**RQ**$_{3.2}$). For **RQ**$_{3.1}$, we mainly investigate the Python-specific TD and discuss the differences in the TD categories that are caused by Python features. For **RQ**$_{3.2}$, we focus on the issues that are caused by the backward incompatibility between Python 2 and Python 3 and further discuss the possible influence of updating Python versions for TD evolution.

3 | RESULTS

In this section, we present our findings and answer each research question. We note that RQ$_1$ investigates the evolution of TD remediation in Python through five subquestions, whereas RQ$_2$ compares the results of RQ$_1$ with those observed in Java by Digkas et al.[3] Therefore, in Section 3.1, we address both research questions together, and for each subquestion, we first present the findings regarding Python and then compare with those from the study in Java. Finally, in Section 3.2, we elaborate on the results for RQ$_3$.

3.1 | Evolution of technical debt remediation in Python and comparison with Java

3.1.1 | Fixing rate variation among projects

Figure 1 presents the issue fixing rate (dark color), that is, the number of closed issues divided by the total number of issues, for every Python project. Moreover, the projects are sorted in decreasing order of their fixing rate. More than half of the issues were fixed in the first six projects; however, the number of issues are quite different among those projects. For example, INCUBATOR-MXNET has the largest number of total issues (almost 30K issues), while less than 200 issues appear in INCUBATOR-SENSSOFT-USERALE-PYQT5. At the opposite spectrum, the fixing rates of eight projects are less than 5%. These projects have small SLOC (less than 1.4K) and number of total issues (less than 161). The only exception is INCUBATOR-SDAP-EDGE with 4.9K SLOC and 940 issues and only has two commits during its evolution, which is also the minimum value in our dataset.

From Table A1, we noticed that the projects have a median value of 187 for the number of commits. This seems to indicate that a certain number of projects are small. To further investigate the relationship between fixing rates and the number of commits and to better compare with the fixing rates of Java projects, we conducted a separate analysis for projects with small and large number of commits. Figure 2 illustrates box plots depicting the distribution of fixing rates for small (less than 187) and large (more than 187) number of total commits for all the projects. It is obvious that projects with small number of commits tend to have more extreme values of fixing rates, and the majority of them have a fixing rate of less than 5%. However, the fixing rates of larger projects are much higher, and the median value is 27.8%.

Furthermore, we analyzed the SLOC and the absolute number of fixed issues in each project. We observed a similar trend on SLOC and the number of fixed issues; that is, **technical debt is more likely to be repaid in larger projects during their evolution**. To verify this observation, we sought to calculate the correlation coefficient between all four variables. For that, we first used the Kolmogorov–Smirnov test[35] to check if the data of each variable fail to follow the normal distribution. The results showed that only the fixing rate is normally distributed (p value $= .33$[||||]). Following these results, we decided to verify the observation by calculating the Spearman correlation.[36] The result between SLOC and the number of fixed issues showed to be positively strong[***] ($\rho = 0.79$). Moreover, we found that the fixing rate has a strong positive correlation with the number of fixed issues ($\rho = 0.76$) and a medium positive correlation with the number of total issues ($\rho = 0.42$) and SLOC ($\rho = 0.33$).

Comparison. Similarly to the Java study, we observe a near-exponential decrease trend of the fixing rate (see Figure 1), where a small percentage of projects (about 10–15%) from both Python and Java samples show rates above 50%. Moreover, both Python and Java samples contain only two projects that each display fixing rates more than 70%. One possible explanation for these two Python projects is that the majority of their issues are related to the transition from Python 2 to Python 3 (we elaborate further on this kind of technical debt in Section 3.2.2); whereas for the two Java projects with the high fixing rates, Digkas et al. suggest that it may be related to a systematic use of SonarQube by their developers to identify and fixes issues in maintenance activities.

Furthermore, we noticed that the fixing rates for larger Python projects are similar to those in larger Java projects: The median fixing rate for projects that contain more than 1000 commits is between 20% and 30%.

Summary. In $RQ_{1.1}$, we asked how the issue fixing rate varies for different projects. We showed that the fixing rates for different projects vary greatly, indicating a wide variation in the repayment practices. To partially answer RQ_2, we compared these results with those from the Java study and concluded that despite dissimilar rules, the fixing rate in the studied sample of Python projects shows a similar distribution with the fixing rate observed in Java projects. Moreover, larger projects have a high probability that around 20–30% TD issues will be fixed during their evolution regardless of the programming language.

3.1.2 | Fixing prevalence among different kinds of debt

The chart in Figure 3 presents the distribution of the number of fixed issues for each of the 56 Python rules detected in our project set by SonarQube. The figure shows a strongly skewed distribution where issues corresponding to about a dozen rules are fixed overwhelmingly more often. Calculating the Gini index, estimated to be the most appropriate single measure of inequality,[38] we obtain a very high value of 0.814, which indicates a very unequal distribution across the rules. We thus conclude that **most of the issues fixed during repayment concern only a small number of rules**.

However, this conclusion depends on the accuracy of SonarQube to detect TD issues that are fixed. To assess SonarQube's limitations, the first and second authors manually analyzed 1% of the number of total fixed issues (i.e., ≈430) to investigate whether those issues represent technical debt and whether their evolution is accurately captured. According to the results, the number of fixed issues is unequally distributed across the rules. Thus, we randomly selected issues by using *stratified random sampling*,[39] which is used to estimate population parameters efficiently when subpopulations have substantial variability.[40] For each rule, we randomly selected a number of issues based on its fixing prevalence and

[||||]The Kolmogorov–Smirnov test verifies if values deviate from the normal distribution and thus, a statically significant results means that values are not normally distributed.
[***]We interpret the correlation coefficient according to Cohen,[37] that is, no correlation when $0 \le |\rho| < 0.1$, small correlation when $0.1 \le |\rho| < 0.3$, medium correlation when $0.3 \le |\rho| < 0.5$, and strong correlation when $0.5 \le |\rho| \le 1$.

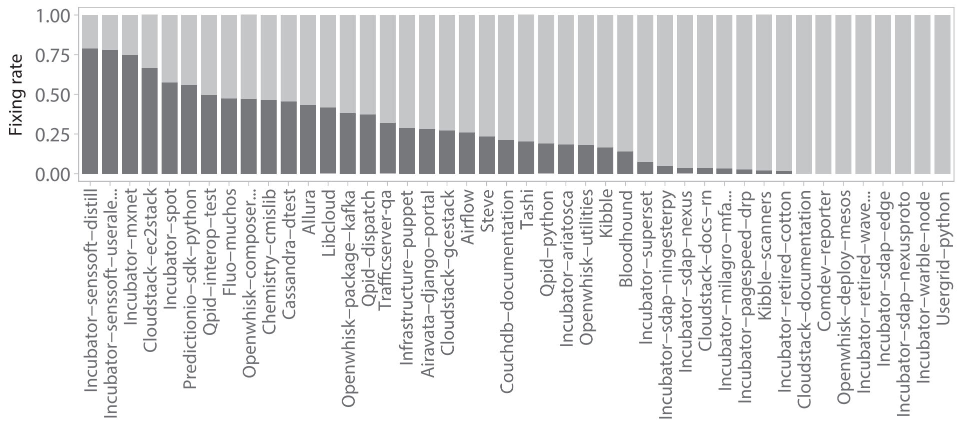

FIGURE 1 Percentage of fixed issues (black) in each project

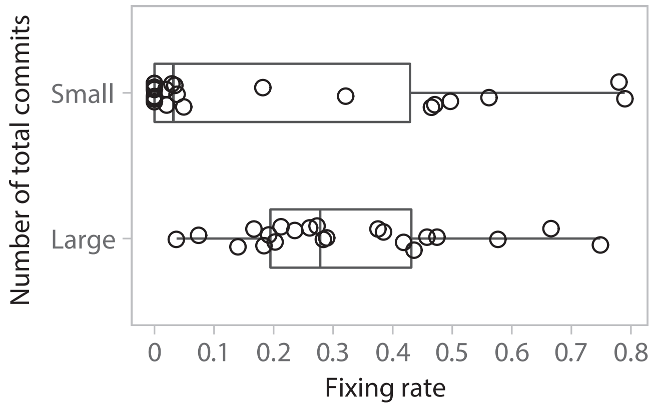

FIGURE 2 Distribution of fixing rate for small and large number of total commits

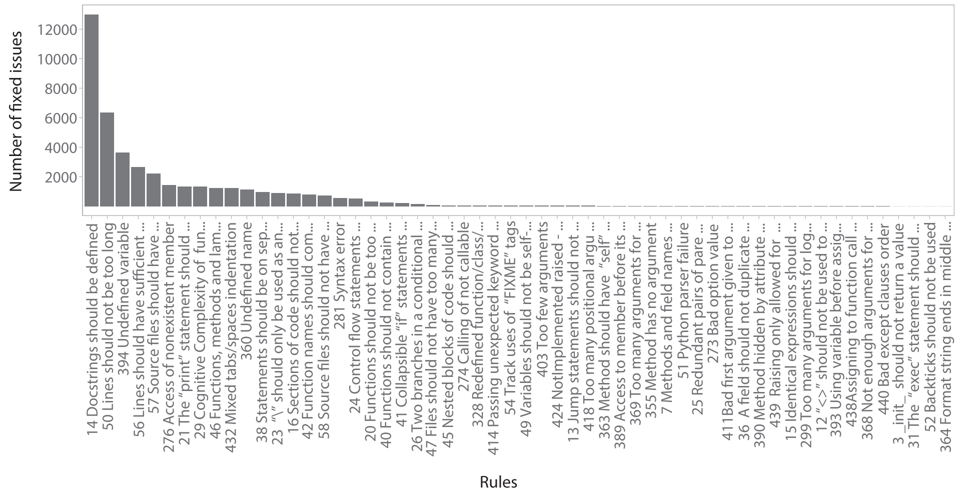

FIGURE 3 Distribution of fixed issues for the 56 rules detected in this study. Every bar represents a rule, height proportional to the number of fixed issues

checked their source code to verify whether they may represent TD according to the definitions of Alves et al[30] and Li et al.[1] In addition, we checked if the issue was actually fixed.

The results of our analysis are presented in Table 3. We clarify that the issues of 32 rules were not checked, as their prevalence, stratified to 1%, was near zero (i.e., no issues to select). However, the fixing prevalences of the 24 considered rules account for 98.54% of all fixes in the analyzed projects. A total of 426 issues have been randomly selected, which were all found to be potential TD issues (i.e., suit the used definitions). Moreover, only nine issues were not actually fixed.

To further investigate the research question, we examine the fifteen rules that account for over 90% of all fixes in the analyzed ecosystem (43K). Table 4 presents these 15 rules and the percentage of fixed issues for each, from the total number of fixed issues. As mentioned in Section 2, we grouped the rules into five higher-level technical debt categories, that is, Code Debt, Defect Debt, Design Debt, Documentation Debt, and Test Debt. In the following, we discuss the categories of the most fixed types of technical debt at the ecosystem level.

TABLE 3 Stratified random sampling for 426 issues

Rule	Prevalence	Issues[a]	TD[b]	Fixed[c]
Docstrings should be defined	30.49%	132	100%	100%
Lines should not be too long	14.88%	64	100%	100%
Undefined variable	8.50%	37	100%	100%
Lines should have sufficient coverage by tests	6.26%	27	100%	81.48%
Source files should have a sufficient density of comment lines	5.19%	22	100%	81.82%
Access of nonexistent member	3.37%	15	100%	100%
The 'print' statement should not be used	3.16%	14	100%	100%
Cognitive Complexity of functions should not be too high	3.12%	13	100%	100%
Functions, methods, and lambdas should not have too many parameters	2.90%	13	100%	100%
Mixed tabs/spaces indentation	2.86%	12	100%	100%
Undefined name	2.66%	11	100%	100%
Statements should be on separate lines	2.29%	10	100%	100%
'\' should only be used as an escape character outside of raw strings	2.11%	9	100%	100%
Sections of code should not be 'commented out'	2.03%	9	100%	100%
Function names should comply with a naming convention	1.90%	8	100%	100%
Source files should not have any duplicated blocks	1.72%	7	100%	100%
Syntax error	1.30%	6	100%	100%
Control flow statements should not be nested too deeply	1.24%	5	100%	100%
Functions should not be too complex	0.77%	3	100%	100%
Functions should not contain too many return statements	0.60%	3	100%	100%
Collapsible 'if' statements should be merged	0.48%	2	100%	100%
Two branches in a conditional structure should not have exactly the same implementation	0.37%	2	100%	100%
Files should not have too many lines of code	0.19%	1	100%	100%
Nested blocks of code should not be left empty	0.16%	1	100%	100%
Total	98.54%	426		

Abbreviation: TD, technical debt.
[a]The number of sampled fixed issues.
[b]The percentage of sampled issues that may present technical debt.
[c]The percentage of TD issues that were actually fixed.

TABLE 4 Fixing prevalence for Top 15 rules

#[a]	Category	Severity	Rule	Prevalence
1	Documentation	Major	Docstrings should be defined	30.49%
2	Code	Major	Lines should not be too long	14.88%
3	Defect	Major	Undefined variable	8.50%
4	Test	Major	Lines should have sufficient coverage by tests	6.26%
5	Documentation	Major	Source files should have a sufficient density of comment lines	5.19%
6	Defect	Major	Access of nonexistent member	3.37%
7	Defect	Major	The 'print' statement should not be used	3.16%
8	Design	Critical	Cognitive Complexity of functions should not be too high☕	3.12%
9	Code	Major	Functions, methods, and lambdas should not have too many parameters	2.90%
10	Defect	Major	Mixed tabs/spaces indentation	2.86%
11	Defect	Major	Undefined name	2.66%
12	Code	Major	Statements should be on separate lines☕	2.29%
13	Code	Major	'\' should only be used as an escape character outside of raw strings	2.11%
14	Code	Major	Sections of code should not be 'commented out'☕	2.03%
15	Code	Major	Function names should comply with a naming convention	1.90%

[a]Overall ranking across all the issue rules.
☕Also among the most fixed issues in the Java study.[3]

Rules #1 and #5 refer to Documentation Debt and account for more than one third (35.68%) of all fixed issues. 'Docstrings should be defined' is by far the most fixed problem. These issues are caused by incomplete design specifications and insufficient comments in code. This paints a positive picture in the sense that the most prevalent repaid debt issues tend to be the code that is initially committed without comments and eventually fixed by adding comments. In addition, developers invest great effort in fundamental activities that can impact reuse and maintenance.

Code Debt is also of concern as it accounts for more than 26% of all fixes. Rules #2, #9, #12, #13, #14, and #15 refer to the issues found in the source code that can affect its maintainability. Moreover, we notice that these rules impact the complexity and readability of the source code. Among these rule violations, the majority (i.e., 57%) of them are related to long lines, while long parameter lists rank the second highest. The remaining four rules are all related to code conventions and rank at the bottom of Table 4.

Test Debt receives also some attention as rule #4 accounts for more than 6% of all the fixed issues. Together, Code Debt and Test Debt fixes suggest that **reducing code complexity and boosting maintainability is paramount**.

Defect Debt includes known defects that, due to competing priorities and limited resources, should be fixed later, as well as unknown defects.[30,32] Table 4 shows that rules #3, #6, #7, #10, and #11 account for more than one fifth (20.56%) of all the fixed issues. Although it may seem that the amount of such debt is alarmingly high, in the majority of the cases, this debt concerns a kind of debt particular to Python, for example, the transition from Python 2 to Python 3. We further discuss this matter in Section 3.2.2.

There is only one rule (#8—'Cognitive Complexity of functions should not be too high') appearing in Table 4, which belongs to Design Debt, and is of critical severity. This indicates that only about 3% of fixed issues are related to Design Debt.

Comparison. First, we notice that the distribution of fixed issues (Figure 3) is similar to that observed in the case of Java. Moreover, the Gini index value is also close, that is, within 3% of the corresponding Java value.[3] In Table 4, the ☕ symbol marks the rules that are also found in the most frequently fixed rules for Java.[3] Among the common rules, rule #8 also appears among the 10 most fixed rule violations in the Java study, where they account for 5.4% of total fixed issues.

However, the similarities end there. The rules related to code convention (i.e., #12 and #14) are fixed at a relatively low frequency in Python, whereas they are the most frequently fixed for Java projects. Furthermore, in the Java study, four out of the Top 10 most addressed rules are related to Design Debt, accounting together for more than 21.5% of the total fixed issues. The difference might indicate that Design Debt violations are more urgent for Java developers. Finally, although rules related to testing and documentation account for approx. 42% of all the issue fixes in Python, they do not even make the Top 10 in Java. This finding may be justified by the different needs of the Python language, which are discussed in Section 3.2.1.

Summary. In $RQ_{1.2}$, we asked what is the fixing prevalence among different kinds of debt, which we answer as follows: A small number of issues are responsible for most of the fixes; that is, the number of issue fixes is unequally distributed across different rules. To partially answer RQ_2, we compared these results with those from the Java study and concluded that issues related to testing and documentation are fixed much more often in Python than in Java; in contrast, Design Debt is fixed much more often in Java.

3.1.3 | Fixing rate variation among different kinds of debt

To compute the fixing rate for each rule, we determine the percentage of issues that are fixed from the total amount of issues corresponding to that rule. To focus on issues that are the most likely to be relevant for software developers, in the remaining analysis for this research question, we limit our discussion to rules with more than 500 issues in the ecosystem. This results in 20 rules that are presented in Table 5. For each rule, the table also presents the category of the rule (as already discussed in **$RQ_{1.2}$**), the number of issues, the fixing rate, and the number of projects in which violations of the rule appear.

By analyzing Table 5, we observe that the rule '"\" should only be used as an escape character outside of raw strings,' has the highest fixing rate (68.52%), and appears in 20 systems. However, the median fixing rate in these 20 systems is about 18%; this means that if we exclude few outlier systems, violations of this rule are not fixed very frequently. Except for this rule, the other five rules related to Code Debt all rank in the Top 11 and have similar fixing rates, that is, between 45% and 56%.

The **fixing rate of the rules related to Defect Debt are found at both ends of the spectrum: Some are at the high end, others at the low end**. Rules #2, #3, and #4 have some of the highest individual fixing rates (more than 56%, and the latter two also appear in the table with the highest number of fixes overall from **$RQ_{1.2}$**). Rules #14, #17, and #20 have some of the lowest rates. Although rules #4 and #20 are commonly violated by delaying the update to Python 3, the fixing rates are very high and very low, respectively, different. We further discuss the reasons in Section 3.2.2.

In Table 5, a quarter of the rules refer to Design Debt. All of these rules influence the quality of the source code, mostly by incurring unnecessary complexity (marked with bold keywords in the table). Arguably, complexity is a type of debt that is harder to address. Rule #6, that is, 'Source files should not have any duplicated blocks,' has the highest fixing rate among them. One possible reason is the perception that duplicated code severely complicates the maintenance and evolution of large software systems,[41,42] even if emerging empirical evidence does not strongly support this assumption.[43] However, the other four rules are in the bottom half of the table with the lower fixing rates.

TABLE 5 Fixing rate for rules with at least 500 occurrences

#[a]	Category	Rule	Projects	Issues	Rate %[b]
1	Code	'\' should only be used as an escape character outside of raw strings	20	1312	68.52
2	Defect	Syntax error	31	972	56.89
3	Defect	Undefined variable	32	6385	56.82
4	Defect	The 'print' statement should not be used	31	2396	56.34
5	Code	Sections of code should not be 'commented out'	31	1558	55.58
6	Design	Source files should not have any **duplicated** blocks	31	1346	54.38
7	Code	Functions, methods, and lambdas should not have too many parameters	26	2278	54.30
8	Code	Statements should be on separate lines	14	1806	53.99
9	Code	Lines should not be too long	35	12 747	49.80
10	Documentation	Source files should have a sufficient density of comment lines	40	4567	48.48
11	Code	Function names should comply with a naming convention	28	1770	45.71
12	Design	**Control flow** statements should not be **nested too deeply**	27	1265	41.98
13	Documentation	Docstrings should be defined	44	31 364	41.48
14	Defect	Access of nonexistent member	34	3473	41.41
15	Design	Cognitive **Complexity** of functions should not be too high	38	3250	40.89
16	Design	Functions should not be **too complex**	29	812	40.52
17	Defect	Undefined name	26	2975	38.15
18	Design	Functions should not contain **too many return statements**	31	706	36.40
19	Test	Lines should have sufficient coverage by tests	44	7595	35.17
20	Defect	Mixed tabs/spaces indentation	7	3544	34.45

[a]Overall ranking across all the rules.
[b]Fixing rate.

Almost all projects suffer from issues regarding *Documentation Debt* (40 projects for rule #10 and 44 projects for rule #13) and *Test Debt* (44 projects for rule #19). However, *Test Debt*has a lower fixing rate than *Documentation Debt*. In fact, Test Debt has the second lowest fixing rate: There is only one rule for Test Debt (sufficient code coverage), and this seems rather difficult to fix. A possible explanation is offered in the next research question where achieving sufficient test coverage shows to require the highest (repayment) effort. Finally, all 15 issues with the most fixes (see Table 4) also appear in Table 5, which means that **the issues with the highest fixing rate are also fixed more frequently in the ecosystem as a whole**.

Comparison. The threshold of the number of rule violations, that is, more than 500 issues, is the same one that was used in the Java study.[3] Similarly to Python, some of the rules related to Defect Debt (e.g., exception handling) also have the lowest fixing rate for Java. Furthermore, rules related to Design Debt have a low fixing rate in Python, while the Java study similarly reports that a significant part of the rules with the lowest fixing rate are related to design problems, for example, complexity and duplication. However, unlike Python, none of the rule violations with the highest fixing rate in Java appears among the most frequently fixed ones.

Summary. In $RQ_{1.3}$, we asked how the fixing rate varies for different issues, which we answer as follows: Different issues have a wide variation of fixing rates, even if they belong to the same debt type. However, the issues with the highest fixing rate are fixed more frequently in the whole ecosystem. To partially answer RQ_2, we compared these results with those from the Java study and concluded that the majority of the rules with the highest fixing rates are language-specific, either for Java or for Python and that many of the Python issues are caused by the backward incompatibility between versions 2 and 3.

3.1.4 | Repayment effort among different issues

To answer this question, we sum up all the effort required to fix the issues, according to the effort estimates provided for each issue by SonarQube. The total amount of repayment effort due to all the Blocker, Critical, and Major issue fixes is estimated to 11 788 h.

Table 6 shows the 10 rules for which their fixed issues required the most effort for TD repayment. The **Effort** column represents the percentage of remediation effort for each rule, compared with the total effort. Summing up the estimated effort for the 10 rules in the table adds up to more than 92.56% of the estimated effort spent in debt repayment in the ecosystem. Thus, **a minority of rules are responsible for the majority of repayment effort**. Besides effort, the table highlights two other measures: The **RT3** column shows the rank of the same issue in Table 5(**$RQ_{1.3}$**),

TABLE 6 Remediation effort distribution: Ten issues are responsible for more than 90% of the remediation effort

#[a]	Category	Rule	RT3[b]	Effort[c]	CT2[d]
1	Test	Lines should have sufficient coverage by tests	19	46.05%	↑3
2	Documentation	Source files should have a sufficient density of comment lines	10	16.11%	↑3
3	Documentation	Docstrings should be defined	13	9.20%	↓2
4	Defect	Undefined variable	3	5.13%	↓1
5	Design	Source files should not have any duplicated blocks☕	6	3.98%	↑11
6	Design	Cognitive Complexity of functions should not be too high☕	15	3.82%	↑2
7	Code	Functions, methods, and lambdas should not have too many parameters	7	3.50%	↑2
8	Defect	Access of nonexistent member	14	2.03%	↓2
9	Defect	Undefined name	17	1.60%	↑2
10	Code	Function names should comply with a naming convention	11	1.14%	↑5

[a] Overall ranking across all the rules.
[b] Rank of the same issue in Table 5 (fixing rate per individual rule).
[c] Remediation effort percentage
[d] Change relative to the ranking of the same issue in Table 4 (fixing prevalence for rules).
☕ Also found in the Java study of Digkas et al.[3]

and **CT2** column shows the change in position with respect to Table 4(**RQ**$_{1.2}$). These extra columns allow us to correlate the effort with the prevalence and fixing rate for a given rule. The most effort-intensive rule (to fix) corresponds to Testing Debt, which ranks the second lowest in terms of fixing rate (19) of all the rules with more than 500 occurrences. Also the second (code comments) and the third rules (docstrings) related to *Documentation Debt* have high effort in Table 6 (account for 25.32% of the remediation effort) but have relatively low fixing rates, hinting at **a possible inverse relationship between required effort and fixing rate**. When we compare the rank of remediation effort of the rules with their fixed issues count, we find that the majority of rules that required the most effort for technical debt repayment are not among those with the highest number of fixes. This indicates that developers prefer to fix issues that are easier first.

From Table 6, we notice that **test and documentation are the most costly activities in terms of remediation effort**. *Test Debt* (rule #1) requires the largest effort during evolution: Almost half (46.05%) of all the effort is devoted to it. This issue is introduced in a file when its test coverage per line is less than a required threshold. The effort that is required to fix such an issue varies significantly, depending on how many lines of code in a file are inadequately covered by testing.

Moreover, we compared the remediation effort of the top 10 rules with the rank of their fixed issues count (CT2 column) and fixing rates across all the rules (RT3 column). Among those rules, rule #5 ('Source files should not have any duplicated blocks') ranks as the sixth highest fixing rate in Table 5 and has moved up 11 positions with respect to Table 4 showing the number of fixes. This rule has the largest remediation effort among all the rules related to Design Debt, and as mentioned before, it also has the highest fixing rate of all the complexity rules which have more than 500 occurrences.

The only two Design Debt rules among the Top 10 rules with the most remediation effort are #5 and #6; each one of them only accounts for almost 4% of effort in Python. Furthermore, the percentage of remediation effort for Design Debt is almost twice as much as Code Debt. This indicates that **problems associated with Design Debt, for example, duplication and complexity, require considerable repayment effort and pose critical concerns**.

Code Debt (rules #7 and #10) has a relatively low remediation effort, accounting for only 4.6% of the total remediation effort. Finally, the table shows that the effort required to resolve the problems associated with Defect Debt (rules #4, #8, and #9) follows a similar order to their fixing rates (RT3 column).

Comparison. Similarly to our study, Digkas et al[3] were also not able to identify a relationship between repayment effort and fixing rate. However, it is worth mentioning that rule #5 also has the biggest jump when we compare with the ranking of frequently fixed Java issues. Moreover, in both languages, the percentage of remediation effort for Design Debt tends to be higher than Code Debt. The results regarding Code Debt show another point of similarity, as rules of this type have low remediation efforts in both languages.

Unlike the findings in Python, the two Design Debt rules cost the highest payback effort in the Java study; that is, they are responsible for over 11% of total remediation effort, respectively. Moreover, five Design Debt rules appear in the Top 10 Java rules with the highest amount of effort, accounting for 22.9% of the total remediation effort. In contrast, the effort for Design Debt rules in Python is around 7.8%. The result indicates that developers spend more effort to address Design Debt issues in Java projects.

Furthermore, Test Debt and Documentation Debt account for almost half of all the effort in Python projects. However, none of the rules related to Test Debt and Documentation Debt appear in the corresponding list for Java projects.

Summary. In $RQ_{1.4}$, we asked how the repayment effort is distributed among different issues. To that end, we showed that a minority of rules are responsible for the majority of the repayment effort and that the variation in the remediation effort seems to be influenced by characteristics of the programming languages, as we observed different trends for some kinds of debt. To partially answer RQ_2, we compared these results with those from the Java study and found that Design Debt requires more remediation effort than Code Debt in both languages.

3.1.5 | Survival time of issues

To answer this question, we analyzed the survival time of the 43K fixed issues found in this study. Figure 4 shows the survival time organized in bins of 30 days; it highlights that **from the technical debt that does get fixed, a large amount is fixed in a short time**. In particular, a third (34%, ≈15K/43K) of the issues are fixed within 1 month (30 days). Moreover, a majority (60%, ≈26K/43K) of the issues are fixed within 2 months (60 days), and an additional 27% (totaling 87%) are fixed within the first year; only a minority of issues (approx. 13%) last for a longer than a year. We also observe that the longest time it took to fix an issue in the current dataset was almost 8 years.

Further, we focus on the survival time of the 20 rules with at least 500 occurrences reported (see Table 5 in **RQ_1**.3), including 15 rules with the most fixed issues (see Table 4 in **RQ_1**.2), which are responsible for more than 98.3% of the effort required for payback. Because different projects have been developed at different paces, some of them have lived for a long time, and TD issues are removed at any time during this evolution. Therefore, the different paces may also affect the survival time of fixed issues in different projects; to mitigate this potential bias, we calculated the median survival time of each rule in each project.

Figure 5 shows box plots depicting the distribution of the survival median values in years (*x*-axis) for these 20 rules in different projects. The box plots are sorted according to Table 5, that is, from high to low fixing rate (top to bottom), and are color coded according to their TD category. To simplify the figure, we hide the outliers of these box plots.

We observe that the majority of rules (13 out of 20) have a median survival time of 50 days or even less: Their survival time is shorter than the median survival time of all issues. Only one rule (i.e., 'Statements should be on separate lines') tends to survive around 1 year (median value). Moreover, eight out of the Top 10 rules with the largest remediation effort seem to have a shorter survival time (≈30 days); however, there is no obvious relationship between fixing rate and survival time of each rule.

Furthermore, rules from the same TD category tend to have a similar survivability distribution; for example, the median survival time of Documentation Debt rules is about 30 days, while Design Debt has a median survival time of 2 months. Also, Test Debt has a relatively short and concentrated survival time compared with other debt types, whereas the majority of rules related to Defect Debt have a longer lifetime. However, violations to the six Code Debt rules have rather diverse median survival times. Issues related to long lines ('Statements should be on separate lines' and 'Lines should not be too long') are usually the ones with the longest survivability.

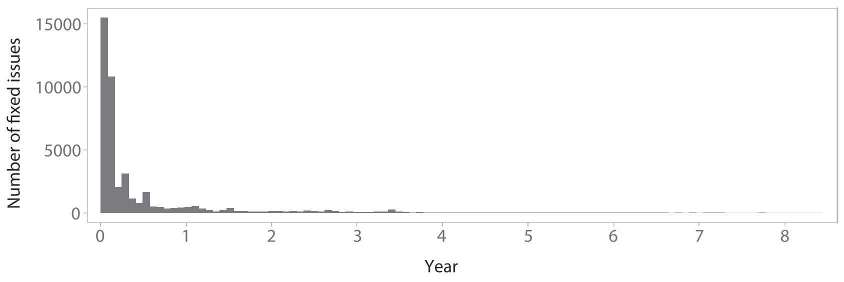

FIGURE 4 Survival time histogram of 43K fixed issues with bins of 30 days

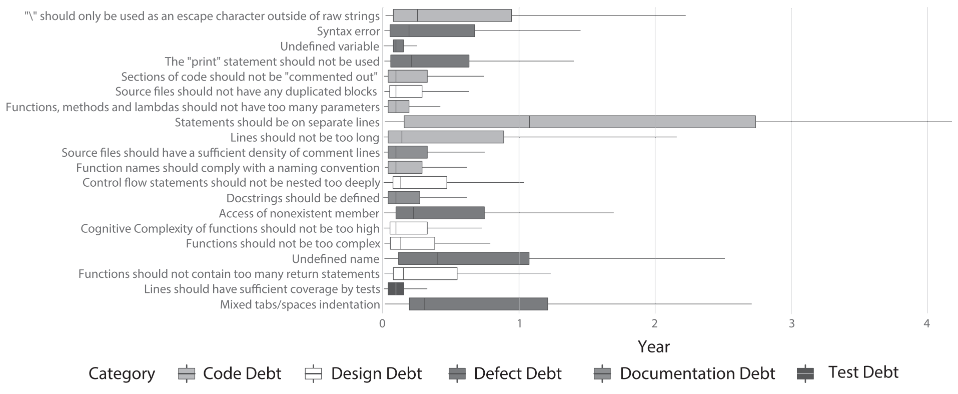

FIGURE 5 Distribution of median survival time for 20 rules with the highest fixing rates (according to $RQ_{1.3}$)

Comparison. In both languages, the data indicate that the majority of the debt that is fixed is fixed relatively soon after its introduction. Moreover, a minority of issues can live in the system for a very long time (the longest lived issues for Python survive for 8 years, while for Java, it is 10 years).

However, technical debt seems to be repaid faster in Python than in Java. In Python, approx. 34% of the issues are fixed within 30 days and 87% within 1 year, whereas for Java, the numbers are 20% and 50%, respectively. The longest survival time of issues for Python (almost 8 years) is less than Java (10 years). We further discuss that difference in Section 3.2.1.

Summary. In $RQ_{1.5}$, we asked after how much time technical debt is paid back. To that end, we showed that a majority of the debt that is fixed is fixed relatively soon after its introduction. To partially answer RQ_2, we compared these results with those from the Java study and learned that (a) only a minority of issues can live in the system for a very long time (8 years the longest-lived issues for Python and 10 years for Java), (b) the rules from the same TD category have a similar survivability distribution, and (c) technical debt seems to be repaid faster in Python than in Java.

3.2 | Learning from the differences between Python and Java

3.2.1 | Python features

Python is known for its dynamic typing system, making it popular for flexibility, expressiveness, and succinctness and less maintainable and secure.[44] Altogether, the misuse of dynamic features could lead to coding issues that are often fixed by adding tests (Test Debt remediation) or exception handling (Defect Debt remediation).[15] Thus, remediation of Test Debt (i.e., 'Lines should have sufficient coverage by tests') and Defect Debt tend to appear much more frequently in Python.

Examining Test Debt closer, it is a widely spread phenomenon: Test Debt appears in all the 44 Python projects. The results of $\mathbf{RQ}_{1.3}$ (see Section 3.1.3) and $\mathbf{RQ}_{1.4}$ (see Section 3.1.4) reveal that although Test Debt has the second lowest fixing rate; it requires the largest effort. Moreover, it only has been fixed in one third (15 out of 44) of the Python projects during their evolution. To further investigate Test Debt, we focused on these 15 projects and found that they have larger sizes (45K SLOC in average) and higher longevity (around 4 years, with 3.5K commits in average). This confirms that Python code that is large and long-lived forces developers to increase test coverage over time.[45]

Furthermore, it is noticeable that issues related to Test Debt are not only being regularly fixed in projects with long survival time, but they are also fixed relatively quickly. This indicates that in order to improve software maintainability, developers should be conscious of fixing issues related to Test Debt earlier in larger and long-term maintenance projects. This confirms literature findings that Test Debt has a strong negative influence on software maintenance[46-48] and high test coverage may increase software quality.[49,50] Considering the Python features, we showed in Section 3.1that projects with deeply nested control flow statements (i.e., Design Debt) and exception handling statements (i.e., Defect Debt) tend to be significantly less likely to be covered by tests.[45]

Finally, our results also showed that technical debt tends to be repaid faster in Python, even looking across the same rules. In addition, the longest survival time of issues among the Python projects (8 years) is shorter than among Java projects (10 years). One possible reason may be that the first Python project from the ASF was developed in 2006. Java appeared earlier than Python, and thus, the development time of Java projects is generally much longer than that of Python projects in the Apache ecosystem. Thus, long-lived projects are more likely to have issues with longer survival time. Another potential reason could be that, although Python is harder to maintain, its dynamic typing system also allows for smaller changes, which may facilitate maintenance activities.

3.2.2 | Transition from Python 2 to Python 3

According to the results of $\mathbf{RQ}_{1.1}$, there is a wide variation in the fixing rate for the projects in the dataset. We observe that both Java and Python projects contain two projects with very high fixing rates, but the reasons are different. In Python, the two projects that display the highest fixing rates (more than 78% of the issues were fixed during their evolution) are as follows: INCUBATOR-SENSSOFT-USERALE-PYQT5 provides comprehensive user event tracking for web pages and INCUBATOR-SENSSOFT-DISTILL is an analytics framework for handling and analyzing user data[†††].

In these projects, the majority of the issues (60% and 52%, respectively) are violations of the rule 'Mixed tabs/spaces indentation,' and these issues are all fixed during their evolution. That rule is related to the transition from Python 2 to Python 3 because tabs and spaces for indentation are not allowed to be mixed in the latter. In the case of Java, Digkas et al[3] found that development teams use SonarQube to manage and maintain the two projects with the highest fixing rates in the Java study. This is not the case for Python: We checked the repository and commit messages for all the Python projects and could not find any evidence of using SonarQube to manage TD.

[†††]https://senssoft.incubator.apache.org/system/

As shown in **RQ**$_{1.2}$, five rules related to Defect Debt account for more than one fifth of all the fixed issues. Among them, rules #7 and #10 occur with the update of the Python interpreter (from Python 2 to Python 3). In Python 3, the 'print' statement requires parentheses, and mixing the use of tabs and spaces for indentation is no longer allowed. Furthermore, some issues that are mapped in rules #3, #6, and #11 are also caused by the update of the Python interpreter. For these rules, we randomly selected 300 fixed issues for each rule and checked them manually. Regarding rule #3 ('Undefined variable'), over 70% of the fixed issues are related to references to Python 2 identifiers (e.g., `'xrange,'` `'unicode,'` and `'basestring'`). For rules #6 ('Access of nonexistent member') and #11 ('Undefined name'), only about 6% of the issues were related to naming or syntax affected by the different versions of the Python interpreter, for example, references to `'socket'` instead of `'SocketIO'` and to refactored parts of the module `urllib`. Among them, the first two authors had disagreements only on the issues related to rule #3. However, there was a high level of inter-rater agreement between two authors when we calculated Krippendorff's alpha ($\alpha = .82$).[34] Altogether, we estimate that **a majority of the fixed Defect Debt(≈60%) is related to incompatibilities between Python 2 and Python 3**. That is a plausible explanation of why the fixed issues related to Defect Debt account for a higher percentage of the total fixed issues in Python compared to Java (≈15%). Moreover, unlike the effect of updating the Python interpreter, the majority of Defect Debt issues in Java projects are related to exception handling.

Although multiple Python-specific issues are related to postponing the ion to Python 3, the rule violations have a wide distribution: Some rules have a high fixing rate while others a low fixing rate (discussed in **RQ**$_{1.3}$). This is partially in contrast to the Java study, where the majority of Java-specific rule violations tend to have a quite high fixing rate. To study this further, for rules #2 and #3 ('Syntax error' and 'Undefined Variable') in Table 5, we randomly sampled 300 fixed issues and manually inspected them. We found out that most of the issues are associated with the use of older versions of the Python interpreter (e.g., invalid syntax in `except` handler with a comma or the use of Python 2 specific identifiers such as `basestring`). As we did before, we also estimated the inter-rater agreement for rules #2 and #3 by calculating Krippendorff's alpha,[33] and the values are .78 and .85, respectively. We thus estimate that almost half of the Defect Debt (47%) is related to postponing the update to the latest Python interpreter and has a high fixing rate (over 56%). In contrast, rule #20 is also commonly violated by delaying the update to Python 3; however, it has the lowest fixing rate in Table 5. Issues pertaining to this rule appear 3544 times in only seven projects. Although six projects have higher fixing rates (over 50%) and three of those have totally fixed all these issues, one project (TASHI) has a rate of 16% and contains more than 69% of all issues (2463 out of 3544). Such observations suggest that **some applications within the Apache ecosystem have still not migrated from Python 2**.

It is worth noting that Test Debt (i.e., insufficient test coverage) accounts for almost half of the remediation effort (approx. 46%) for Python. Because there is a great difference between Python 2 and Python 3, developers should make sure that the test suite is thorough before migrating to Python 3. Furthermore, migrating the tests before the rest of the code facilitates the identification of defects and contributes to a smoother transition. Moreover, migrating the test suites tend to be simpler than migrating code; thus, it could also provide an idea of how easy it can be to migrate projects to Python 3.[51] Thus, issues related to insufficient test coverage tend to be more important in Python projects, especially for projects with a long history, because they are more likely to have migration problems with different Python versions.

4 | DISCUSSION

In the previous section, we reported on the evolution of TD remediation in Python and pointed out the similarities and differences compared with Java. Although the sets of SonarQube rules for the two languages do not completely overlap, we identified a number of similarities. In both languages, projects are likely to have around 20–30% of their TD issues fixed, with a small number of issues accounting for most of the fixes. Moreover, the majority of the rules with the highest fixing rates are related to language-specific rules. In addition, a minority of rules account for the majority of the remediation effort, and Design Debt requires more effort than Code Debt. Furthermore, for the technical debt issues that do get repaid, the majority is fixed relatively soon after they are introduced. However, a minority of the issues that get fixed stay in the system for a very long time in both languages (until they are eventually fixed).

We also found some differences, that is, results that are unique to either Python or Java: The potential reasons for the high fixing rates tend to be different between Python and Java projects; Test Debt and Documentation Debt are fixed much more often and also require more effort in Python, while Design Debt is fixed much more often in Java; the high fixing rates for some Python rules are caused by the changes between major versions of the Python interpreter. Furthermore, we presented possible explanations for the observed differences based on particular features of the Python language and the migration from Python 2 to Python 3. This discussion was at the level of TD categories (e.g., Code Debt and Design Debt). However, the differences in the number of rules among the categories may influence the comparison; for example, Test Debt only has one rule in the Python results.

One possible way to compare at the level of rules is to look at the rules that are common between the two languages. To that end, we analyzed the 15 rules that are common to both Python and Java. Thus, in Section 4.1, we focus on these 15 common rules, revisit the research questions **RQ**$_{1.2}$, **RQ**$_{1.3}$, and **RQ**$_{1.4}$, and discuss them. We do not revisit **RQ**$_{1.5}$ as we could not derive the related information from the Java study. Finally, in Section 4.2, we focus on the implications of this work for both developers and researchers.

[$]https://senssoft.incubator.apache.org/system/

4.1 | Common rules in both languages

Table 7 shows the comparison between the findings for Python and Java for the 15 common rules, regarding the prevalence of fixed issues (**RQ$_{1.2}$**), the fixing rate (**RQ$_{1.3}$**), the effort required to fix the issues (**RQ$_{1.4}$**), and the percentage of projects in which violations of the rule appear. Fixing rates of the common rules have been calculated in RQ$_{1.3}$; thus, we used those values here. However, for the comparison in RQ$_{1.2}$ and RQ$_{1.4}$, the results we had concerned the fixing prevalence and remediation effort for all the rules; thus, we calculated the distribution of the number of fixed issues and remediation effort among the 15 common rules. Moreover, the 'Aggregated' row on Table 7 shows the results from combining the 15 rules; that is, those 15 common rules account for 11.89% of all fixed issues in Python and 34.53% in Java.

For **RQ$_{1.2}$**, it is interesting that the five most prevalent fixed issues are the same in both Python and Java: the Top 5 rules in Table 7. Moreover, the number of issue fixes related to these five rules accounts for almost 90% of common issue fixes in both Python (87.35%) and Java (89.18%). Considering all of the most prevalent issues in both languages, we note that those five rules appear in the Top 10 Java rules with the most fixing prevalence, and three of them appear in Python. Furthermore, among the common rules, issues related to 'Cognitive Complexity of functions should not be too high' appear in the largest number of Java and Python projects (esp. in the case of Java this rule is found in all projects as shown in the last column of Table 7). This highlights that complexity is a major source of concern in both languages.

Regarding **RQ$_{1.3}$**, there is a wide variation of fixing rates in Table 7. The rule 'Variables should not be self-assigned' seems to be a special case with a fixing rate of 100% in Java projects. However, there are only 30 issues related to that rule, so it appears to be an outlier. Excluding that extreme value, fixing rates of the rest of the common rules have similar variation intervals in Python and Java, that is, from 19% to 56%.

For **RQ$_{1.4}$**, we note that the three most costly activities (i.e., rules #1, #2, and #5) are the same in both languages and even appear in the same order. These three rules require the majority of remediation effort, that is, more than 80% of the total effort estimates for the common rules. Among them, the violations of the rule 'Source files should not have any duplicated blocks' require the most estimated effort in both Java and

TABLE 7 Comparison of 15 common rules for Python and Java

			RQ$_{1.2}$: P %[c]		RQ$_{1.3}$: F %[d]		RQ$_{1.4}$: R %[e]		Projects%[f]	
#[a]	C[b]	Rule	Python	Java	Python	Java	Python	Java	Python	Java
1	S	Cognitive Complexity of functions should not be too high	26.19	15.76	40.89	42.34	36.46	32.30	86.36	100
2	C	Statements should be on separate lines	19.21	22.77	53.99	40.59	1.32	1.96	31.82	91.23
3	C	Sections of code should not be 'commented out'	17.06	22.56	55.58	41.44	5.85	9.69	70.45	100
4	S	Source files should not have any duplicated blocks	14.42	11.05	54.38	19.47	38.01	34.00	70.45	100
5	S	Control flow statements should not be nested too deeply	10.46	17.04	41.98	31.01	7.17	14.63	61.36	100
6	S	Collapsible 'if' statements should be merged	4.04	2.56	41.75	36.69	1.38	1.10	68.18	100
7	S	Two branches in a conditional structure should not have exactly the same implementation	3.07	0.27	48.75	48.00	2.11	0.23	47.73	70.18
8	S	Files should not have too many lines of code	1.64	0.61	31.44	25.53	6.73	3.14	36.36	98.25
9	C	Nested blocks of code should not be left empty	1.32	4.50	37.22	38.24	0.45	1.93	36.36	96.49
10	C	Methods and field names should not differ only by capitalization	0.30	0.89	34.88	40.42	0.20	0.76	20.45	87.72
11	C	Redundant pairs of parentheses should be removed	0.26	0.21	44.83	32.49	0.02	0.02	15.91	56.14
12	C	A field should not duplicate the name of its containing class	0.16	0.20	28.57	23.19	0.11	0.17	25.00	89.47
13	D	Track uses of 'FIXME' tags	0.91	1.16	26.59	44.92	0.00	0.00	22.72	68.42
14	C	Identical expressions should not be used on both sides of a binary operator	0.12	0.28	19.35	55.51	0.02	0.05	20.45	56.14
15	F	Variables should not be self-assigned	0.85	0.02	55.13	100	0.17	0.00	22.72	7.02
		Aggregated	11.89	34.53			10.47	34.64		

[a]Ranking across the common rules.
[b]Category (C, Code Debt; S, Design Debt; D, Documentation Debt; F, Defect Debt).
[c]Prevalence percentage.
[d]Fixing rate.
[e]Remediation effort.
[f]The percentage of projects in which the rule appears.

Python, accounting for 34% and 38%, respectively, of the total remediation effort. Furthermore, 'Cognitive Complexity of functions should not be too high' accounts for the second largest amount of repayment effort (about 32% and 36%, respectively) for projects of both languages. In addition, the fixing rates for this rule are also very similar (approx. 40%).

The high fixing rates and remediation effort of common issues related to complexity (i.e., rules #1–#5) indicate that those issues are given priority despite requiring significant effort to fix; this is likely because developers understand their impact on software maintenance and try to fix them to avoid paying high technical debt interest. Moreover, three of these rules (i.e., rules #1, #4, and #5) are related to Design Debt. In fact, except for two rules (#2 and#3) that belong to Code Debt, six of the top eight rules with the highest percentage of fixed issues are related to Design Debt. The majority of them also have the highest percentage of remediation effort and appear in most projects. Again, we conjecture that issues related to Design Debt are more commonly addressed by software developers because they tend to cause a lot of extra maintenance effort; despite their high remediation effort, fixing them saves effort in the long term. Consequently, we advise practitioners to prioritize the remediation of these Design Debt issues in both Python and Java. The longer these issues stay in the system, the higher the technical debt interest to be paid.

In addition, almost all of the common rules (i.e., 13 out of 15) are related to Design Debt and Code Debt, which indicates that developers might concentrate on similar issues related to code comprehension, maintenance, and complexity regardless of the programming language. Although half of the rules in the Python study (28 out of 56) belong to Defect Debt, a large number of them constitute Python-specific technical debt; the two studies have only one Defect Debt rule in common. This indicates that, among all technical debt types, Code Debt and Design Debt are more independent of programming languages and have a deeper impact. We see this as an important point for technical debt tool vendors that offer support for multiple languages: Code Debt and Design Debt rules should be assigned higher weights, and the commonalities in these rules across languages should be exploited. Furthermore, development teams that work with two or more languages can have common thresholds or quality gates for Design and Code Debt issues across all languages.

Furthermore, issues with a similar remediation effort might be treated very differently, depending on the programming language. For example, although the remediation effort for the rule violation 'Source files should not have any duplicated blocks' is quite similar for both languages, the fixing rates of that rule in Python and Java vary greatly, approx. 54% and 19%, respectively. Moreover, the average number of issues of this type per file is similar in both languages (approx. 0.24). We investigated this rule further and found that, for the Python projects, the average effort per issue is doubled when compared with the Java projects. SonarQube estimates the effort to fix this issue based on the size of the duplicated block, which means that duplicated blocks in Python are likely to be approx. twice the size of duplicated locks in Java and, therefore, may receive more attention from developers.

Another interesting rule is the one with the second most remediation effort, that is, 'Cognitive Complexity of functions should not be too high.' We analyzed the issues of this rule further and noticed that despite the similarities in repayment effort and fixing rates of that rule in two languages: (a) Python has considerably more issues per files than Java (approx. 0.59 and 0.24, respectively); whereas (b) Java has marginally more complex methods (based on the repayment effort per issue). This sheds light on the programming style associated with the two languages. For example, methods in Java classes can be inherently more complex, while Python offers a variety of idioms (e.g., list comprehension) that alleviates cognitive complexity. Moreover, Python modules are often implemented in a more imperative style (as opposed to object-oriented), which may, for example, result in fewer accessor methods.

4.2 | Implications

The previous sections elaborated on both generalized and language-dependent results. Because both of them have implications for developers and researchers, we discuss these implications first for language-dependent and subsequently for generalized findings.

4.2.1 | Lessons learnt from the differences

On the basis of the findings that are different between Python and Java projects, we highlight that most of the rules with the highest fixing rates and highest spent remediation effort are language-dependent. This limits the external validity of empirical studies (like this one) to the language studied. Researchers should consider this threat to validity and if possible replicate studies among languages and compare results. In fact, finding differences among languages can guide the calibration of technical debt tools towards the individual features of each language, instead of simply having different rules for different languages.

Test Debt and Documentation Debt are fixed much more often and repaid very fast in Python projects. Practitioners can consider this information to prioritize their maintenance activities: They can check whether these issues are also fixed quickly in their project; if they are not, practitioners can reflect whether they have a good reason not to fix them. Moreover, researchers can focus on improving the detection of issues related to other debt types, for example, Defect Debt and Code Debt, and try to understand why these are not prioritized by practitioners.

Although Test Debt is highly spread in Python projects, it is also likely to be resolved within the first year. This information can be used as an early warning of long-lived debt, as any Test Debt surviving for longer than a year will potentially survive for a much longer period.

A large proportion of debt (esp. Defect Debt) is associated with the migration between Python 2 and Python 3. This should urge researchers to pay close attention to debt emerging from programming language updates (e.g., from changes in syntax or added idioms) and improve related TD detectors and analyzers in time. For example, these tools can warn developers of such migration issues and promote improved best practices early enough to mitigate further risky debt such as from the discontinuation of an entire major version.

Although the dynamic features of Python may be associated with greater maintenance efforts, they can also lead to a more cautious development style, for example, with a stronger debt repayment culture. On the one hand, this indicates a potential maturity of the Python community in adopting the observed debt repayment practices. On the other hand, it should draw the attention of researchers to further investigate the tight relationship between the features (and associated culture) of a programming language and TD repayment practices.

4.2.2 | Lessons learnt from the similarities

On the basis of the general findings that are the same or similar for both languages, we highlight that fixing rates for the majority of projects are not high, but that is not necessarily alarming; projects may thrive despite the presence of technical debt. Practitioners should look more at the trends (is the debt increasing or decreasing in the long term?) rather than absolute numbers (not all debt has to be repaid) and only set realistic goals for repayment, for exmple, targeting issues that incur high technical debt interest.

Issues that are easier to fix also have a higher fixing rate—developers seem to be dealing with the 'low-hanging fruits.' This unfortunately corresponds to a minority of all rules. Researchers should focus on tool support for the more complex types of debt (with low fixing rates), as they tend to survive much longer incurring developers to pay significant amounts of interest. Any approach that can support developers in fixing these 'expensive' issues more cost-effectively would be of added value.

Because issue fixes related to complexity account for almost 90% of common issues and cost around 90% of the remediation effort, developers should pay more attention to fixing them by giving them high priority among all issues. In addition, TD management tools should emphasize these issues and to some extent prompt developers to deal with them with high priority.

There seems to be an inverse relation between repayment effort and fixing rates, but that was not clearly established. Researchers could look into confirming this relation or even better establishing causality between the two concepts. This would have wide implications for technical debt tools, as they would need to strongly encourage the remediation of debt with high repayment effort. Development organizations can also take this into account by encouraging debt repayment of costly issues as part of the organizational culture.

The majority of issues that do get fixed are fixed rather quickly after being introduced. In contrast, a minority of issues can live for a long time. This has two important implications. First, practitioners can bear this in mind: If they postpone fixing an issue, it will likely survive long enough to incur significant technical debt interest. Second, researchers should explore the reasons behind the long survival of certain issues. Do developers assign them a low priority because they do not consider them important enough or because the remediation effort is too high? Furthermore, researchers can build on this information to calibrate the priority weights and early warnings for such issues in tools. For example, issues that are not incurring high technical debt interest may be assigned lower priorities.

5 | THREATS TO VALIDITY

We discuss the threats to construct and external validity, reliability, and confounding factors. We note that internal validity is not relevant to our study, because we did not seek to establish causal relations.

5.1 | Construct validity

This type of threat pertains to the connection between the research questions and the objects of study (i.e., do we measure what we intend to measure?). In this regard, the result of this study relies on SonarQube to detect TD issues that are fixed and the amount of TD that is paid back. Although the tool is widespread in both industry and academia, our interpretation of TD is limited to the tool's capabilities. Different strategies could be used in different tools to detect TD, which might lead to other possible definitions of technical debt rules and, consequently, potential TD issues that are fixed during the evolution. To assess SonarQube's limitations, the first and second authors randomly selected 426 issues (i.e., 1% of the number of total fixed issues) by using *stratified random sampling*. Then, we manually checked whether those issues represent technical debt and whether their evolution is accurately captured. The results show that all of them represent technical debt, and almost 98% of them were actually fixed.

Furthermore, when we analyze the evolution of projects, we choose the change history on a weekly basis, and thus, the introduction and removal time might have a maximum error of 1 week. Therefore, this could lead to a range of errors in calculating the survival time of fixed issues.

Moreover, projects may be developed at different paces, which may also affect the survival time of fixed issues in different projects. Overall, these two threats only affect the result of $RQ_{1.5}$, that is, the survival time of different fixed issues. To mitigate these threats, at least to some extent, we compare the median survival time of the issues between different rules.

5.2 | External validity

This type of threat concerns the generalizability of our findings. Although we analyzed all the Python systems in the Apache ecosystem, which represent a considerable corpus of evolving Python systems, we cannot claim that these results can fully represent the entire population of non-trivial Python projects.

Furthermore, the set of rules considered in this study is not exhaustive and does not portray the complete set of TD related issues that may affect Python source code. Some of them are language-specific, and some might even have different implementations for different versions of the tool itself. From this point of view, the comparison with Java is limited. To partially address this, we investigate the results of common rules that appear in both Python and Java projects in Section 4.1independently.

Because the version of SonarQube used for this study (SonarQube 7.0) is newer than the one used in the Java study (SonarQube 6.4), we compared the sets of rules to better understand the impact of different versions of the tool in analyzing TD evolution. As a result, we only found two new common rules in the last version; that is, 'Lines should have sufficient coverage by tests,' and 'Identical expressions should not be used on both sides of a binary operator.' The former has a great impact on our result as we discussed in Section 3.2.1. However, the latter has a negligible effect because it only has six fixed issues, accounting for 0.01% of the total fixed issues that have been detected in our study. Finally, we note that many of the technical debt issues are language independent (e.g., method complexity and code duplication).

5.3 | Reliability

To address reliability threats, at least three researchers were involved in both data collection and analysis. Moreover, samples of analysis output at the different steps were manually inspected for irregularities and alignment with the proposed study design. Finally, most steps were automated by scripts, which are publicly available together with the collected dataset.

Moreover, the mapping of the 56 SonarQube rules into the five TD categories was performed independently by two researchers, with a third researcher helping to resolve conflicts (see Section 2.3.2). However, it is possible that different researchers might map these rules to different categories.

5.4 | Confounding factors

Confounding factors are variables that may affect the dependent variables without the knowledge of the researchers. In our study, the main limitations that we expect in this regard pertain to the differences between Java and Python. Despite our attention to details while replicating the study of Digkas et al,[3] some aspects that could not be controlled may affect the comparability of the results.

The most prominent factors are related to the characteristics of the projects. Although all (Java and Python) projects are part of the Apache foundation, the domain of the projects is not uniform among the languages. For example, NUTCH is a Web crawler relying on Apache Hadoop data structures, which has the highest fixing rate among Java projects. However, there is no Web crawler in the selected Python projects. Furthermore, although language features can affect the complexity of the project, and thus comprise valid comparisons, the complexity of the source code can also be influenced by the type of project and developers' experience, which may also not be uniform among Java and Python projects.

6 | ASSOCIATED DATASET AND REPLICATION

Due to the limitations of multiversion analysis with SonarQube, collecting the data required for this analysis can take a long time. In our case, importing the 3643 analyzed versions of the 44 systems data took more than 2 months of work on an Intel Core i7-5500U personal computer with 8GB of RAM.

To support the replication of this study, we created an online repository[‡‡‡] with instructions and scripts to collect the same data that we used in the study. The repository helps with setting up the necessary environment, that is, tools and configuration: It provides a Vagrant script to bootstrap a virtual machine and automate most of the environment setup. The environment includes SonarQube, with a PostgreSQL database, and

[‡‡‡]https://github.com/jieshanshan/TD-Apache-Python

Jupyter[§§§], which is used to support the data collection. The provided Jupyter notebook guides the procedure all the way from acquisition of the Git repositories, through extracting the weekly snapshots, to submitting them to SonarQube for analysis.

For researchers that prefer to avoid the effort of replicating our data collection but might want to perform other kinds of analysis on the data, we publish besides the scripts for data collection also their result: a data dump of the detailed information for the 95 K TD issues analyzed in this paper.[¶¶¶] The detailed information includes ID number, description, severity, status and effort of each issue, together with the project that contains it, the hash values, and dates of the commits that the issue is introduced and removed.

7 | RELATED WORK

The design and results of our work are more closely comparable with two studies by Digkas et al.[3,7] The first study[7] examines the number of TD issues in 66 Java projects of the Apache Software Foundation over a period of 5 years and also use SonarQube to investigate how TD evolved and what types of issues are involved. The results show that on the one hand, there is a significant increase trend on size, number of issues, and on the complexity metrics of the project, while on the other hand, the normalized TD decreases as project evolves. The follow-up study[3] investigates the amount of TD that is paid back and the issues that are fixed.[3] An in-depth comparison with this work is presented in the results of RQ_2 and further discussed in Section 4.1.

Although we used SonarQube like Digkas et al,[3,7] there are other means to detect TD that have been used in other studies. For example, Marinescu[8] proposed and evaluated a framework to detect TD via metrics-based detection rules for seven object-oriented design flaws. Their case study shows how the framework can detect debt symptoms and past refactoring actions. However, they only focused on Design Debt, and their findings are based on two Eclipse projects. Unlike them, we investigate five debt types (including Design Debt) in 44 Apache Python projects. Aligning to our results, the authors also noticed that issues related to code complexity have a significant impact on the software maintenance.

More recently, Bavota et al[32] detected technical debt by analyzing code comments for self-admitted technical debt instead of analyzing the source code itself. In this context, they found that self-admitted technical debt is diffused in the mined open-source projects. Although our study analyzed TD in source code, we also found that TD issues are diffused during the evolution of Python projects.

We detected 56 different TD issues in our study, which belong to five TD categories. However, to the best of our knowledge, most of the previous studies involved only a few types of TD. For example, Olbrich et al[52] analyzed historical data over several years of two projects to investigate two code smells (i.e., Code Debt), namely, God Class and Shotgun Surgery. The results show that they can identify different phases in the evolution of code smells during system development. While we considered 14 rules related to Code Debt in our study, these two code smells are not included. However, similar to our results, their findings also indicate files affected by Code Debt (in their study, by smells) as noticeable maintenance challenges.

Another large scale empirical study was presented by Palomba et al,[53] which focused on investigating the diffuseness of 13 code smells and their impact on maintenance properties. Their findings show that the most diffused smells are related to size and complexity. This observation is in line with our finding that the majority of fixed issues and effort estimates pertain to issues related to complexity. However, unlike them, we detected additional TD categories other than Code Debt and focused on the remediation.

Our investigation of Python was partially motivated by the goal of investigating another prominent programming language besides the highly analyzed Java. A handful of other studies also aimed to do the same. For example, Sharma et al[9] mined 19 design smells and 11 implementation smells in 1988 C# repositories to investigate fundamental characteristics of code smells. Some of those smells were also found in our study, for example, *duplicate code* and *long method*. They found that open-source C# programs have a high average smell density and that some smells occur more frequently. Comparing with our results, we found that TD issues are also widespread in Python projects, as well as that some issues are more prevalent.

8 | CONCLUSIONS AND FUTURE WORK

In this paper, we presented the results of a case study of the multiyear evolution of technical debt remediation in 44 Python projects from the Apache Ecosystem, focusing on the effort required for technical debt repayment and the issues that are fixed. The analysis we performed shows that most of the repayment effort goes into improving testing, adding documentation, reducing complexity, and removing duplication. We also made a comparison between the results of this study and the one on Java reported by Digkas et al,[3] presenting similarities and differences between the two languages.

[§§§]https://jupyter.org/

[¶¶¶]https://github.com/jieshanshan/TD-Apache-Python/blob/master/Original_Data.csv

Several of the findings are aligned with the study on the Java part of the ecosystem. First, a minority of rules account for the majority of issues fixed and the spent effort, and this suggests that addressing those kinds of debt in the future might be important for research and practice. In addition, almost 90% of fixed issues and 80% of effort estimated to be invested in fixing pertain to rules related to Design Debt, for example, complexity; those issues are of common concern to all software developers and maintainers. Furthermore, the majority of the rules with the highest fixing rate (up to 60%) are language-specific issues. Finally, a minority of issues can live for a long time, and the issues from the same TD category have a similar survival time.

Some aspects of TD repayment in the Apache ecosystem are particular to Python projects: (a) More than half of the Python technical debt that gets fixed is short term, that is, being repaid in less than 2 months, which is much faster than the previous study on Java has indicated; (b) documentation-related debt seems more prominent in Python than in Java; and (c) despite Test Debt appearing in all projects, it has only been fixed in one third of Python projects, and those have a long history. Despite these indications, we also highlight that biases such as project-dependent variables (complexity, domain, etc.) may affect the comparison between program languages.

The results of our study have several implications for practitioners, particularly developers. First, our study highlights the importance of strategically managing debt rather than repaying it all. We have shown that in one of the most successful collections of open-source software projects, not all debt is repaid, and there seems to be a preference in the way some types are prioritized over others. Indeed, TD issues related to code complexity seem to receive high fixing priority in the Apache ecosystem. Furthermore, test coverage may require more attention during software maintenance because it tends to require more effort to be fixed; however, on the basis of our observations, developers seem more conscious of testing issues in long-lived projects.

Finally, our results indicate that the majority of the issues are fixed in a relatively short time. Therefore, we believe that quality-minded developers reading this study will verify whether the issues in their Python projects are also as quickly addressed; noticeable delays in this regard may indicate the accumulation of substantial TD.

The findings reported in this paper shed light on several aspects of technical debt in Python projects. However, because this has been an exploratory study, it naturally leads to many questions that are still not answered. In particular, we believe that the following observations and hypotheses stemming from this study are worthy of further investigation.

The differences between the two languages presented and discussed in Section 3 hint at the possibility that the language properties and/or the culture that emerges in the ecosystem might have an impact on the debt repayment practices. However, more research must be done to better understand the impact of the studied types of debt and whether delaying remediation is optimal or there could be better strategies.

The magnitude of the study limited its scope, which only considered the source code. Thus, we did not investigate the motivations of developers. A follow-up study with the ecosystem developers would be imperative to complement the observations in this paper and shed light on the actual intentions and strategies of the various communities. For example, it would be useful to survey developers to see whether they recognize the fast repayment of test and documentation debt that we observed as intentional strategies. Moreover, it would be enlightening to find out whether the rationale for prioritization is related to the programming language.

Furthermore, we plan to also analyze industrial ecosystems because no work has been done in that direction. We also plan to compare the way complementary operationalizations of technical debt fare against the SonarQube one. In addition, we plan to provide an online application that allows any Python project to compare their own technical debt with the projects in the Apache ecosystem.

Finally, it still remains to be seen whether this article is the best way of presenting the data in this study to practitioners or a better way exists that would help them to compare, evaluate, and improve their TD management strategy. We would like to investigate whether a web application that would allow developers to compare their debt remediation statistics with the Apache systems would be a good starting point for discussions and actionable insight.

ACKNOWLEDGMENTS

This work was supported by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website/).

ORCID

Jie Tan https://orcid.org/0000-0003-1868-0123
Daniel Feitosa https://orcid.org/0000-0001-9371-232X

REFERENCES

1. Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. *J Syst Softw*. 2015;101(C):193-220.
2. Kruchten P, Nord RL, Ozkaya I. Technical debt: from metaphor to theory and practice. *IEEE Softw*. 2012;29(6):18-21.
3. Digkas G, Lungu M, Avgeriou P, Chatzigeorgiou A, Ampatzoglou A. How do developers fix issues and pay back technical debt in the apache ecosystem? In: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18). IEEE; 2018; Campobasso, Italy:153-163.
4. Tom E, Aurum A, Vidgen R. An exploration of technical debt. *J Syst Softw*. 2013;86(6):1498-1516.

5. Besker T, Martini A, Bosch J. The pricey bill of technical debt: when and by whom will it be paid?In: Proceedings of the IEEE 33th International Conference on Software Maintenance and Evolution (CSME '17). IEEE; 2017; Shanghai, China:13-23.
6. Avgeriou P, Kruchten P, Ozkaya I, Seaman C. Managing technical debt in software engineering (Dagstuhl Seminar 16162). *Dagstuhl Rep*. 2016;6(4): 110-138.
7. Digkas G, Lungu M, Chatzigeorgiou A, Avgeriou P. The evolution of technical debt in the apache ecosystem. In: Proceedings of the 11th European Conference on Software Architecture (ECSA '17). Springer; 2017; Canterbury, UK:51-66.
8. Marinescu R. Assessing technical debt by identifying design flaws in software systems. *IBM J Res Dev*. 2012;56(5):9:1-9:13.
9. Sharma T, Fragkoulis M, Spinellis D. House of cards: code smells in open-source c# repositories. In: Proceedings of the ACM/IEEE 11th International Symposium on Empirical Software Engineering and Measurement (ESEM '17). IEEE; 2017; Toronto, ON, Canada:424-429.
10. Biaggi A, Arcelli Fontana F, Roveda R. An architectural smells detection tool for c and c++ projects. In: Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '18). IEEE; 2018; Prague, Czech Republic:417-420.
11. Ma W, Chen L, Zhang X, Zhou Y, Xu B. How do developers fix cross-project correlated bugs?: a case study on the github scientific python ecosystem. In: Proceedings of the 39th International Conference on Software Engineering (ICSE '17). IEEE; 2017; Piscataway, NJ, USA:381-392.
12. Alexandru CV, Merchante JJ, Panichella S, Proksch S, Gall HC, Robles G. On the usage of pythonic idioms. In: Proceedings of the acm sigplan international symposium on new ideas, new paradigms, and reflections on programming and software. ACM; 2018; Boston, MA, USA:1-11.
13. Wang B, Chen L, Ma W, Chen Z, Xu B. An empirical study on the impact of python dynamic features on change-proneness. In: Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE '15). KSI Research Inc.; 2015; Pittsburgh, USA:134-139.
14. Ray B, Posnett D, Filkov V, Devanbu P. A large scale study of programming languages and code quality in github. In: Proceedings of the ACM SIGSOFT 22nd international symposium on foundations of software engineering (FSE '14); 2014; Hong Kong, China:155-165.
15. Chen Z, Ma W, Lin W, Chen L, Li Y, Xu B. A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of python dynamic features. *Sci China Inf Sci*. 2018;61(012107):1-18.
16. Van Solingen R, Basili V, Caldiera G, Rombach HD. Goal Question Metric (GQM) approach. *Encyclopedia of Software Engineering*; 2002:528-532.
17. Bavota G, Canfora G, Penta MD, Oliveto R, Panichella S. The evolution of project inter-dependencies in a software ecosystem: the case of apache. In: Proceedings of the IEEE 29th International Conference on Software Maintenance (ICSM '13) IEEE; 2013; Eindhoven, the Netherlands:280-289.
18. Easterbrook S, Singer J, Anne Storey M, Damian D. Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*; 2008:285-311.
19. Campbell GA, Papapetrou PP. *Sonarqube in action*. 1st ed. Greenwich, CT, USA: Manning Publications Co.; 2013.
20. Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G. Are static analysis violations really fixed?: a closer look at realistic usage of sonarqube. In: Proceedings of the 27th International Conference on Program Comprehension (ICPC '19). IEEE; 2019; Montreal, Quebec, Canada:209-219.
21. Lenarduzzi V, Saarimaki N, Taibi D. On the diffuseness of code technical debt in java projects of the apache ecosystem. In: Proceedings of the IEEE/-ACM 2rd International Conference on Technical debt (TECTDEBT '19). IEEE; 2019; Montreal, QC, Canada:98-107.
22. Letouzey J. The sqale method for evaluating technical debt. In: Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD '12). IEEE; 2012; Zurich, Switzerland:31-36.
23. Letouzey J, Coq T. The sqale analysis model: an analysis model compliant with the representation condition for assessing the quality of software source code. In: Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID '10).; 2010; Nice, France:43-48.
24. Letouzey J, Ilkiewicz M. Managing technical debt with the sqale method. *IEEE Software*. 2012;29(6):44-51.
25. Griffith I, Izurieta C, Huvaere C. An industry perspective to comparing the sqale and quamoco software quality models. In: Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '17). IEEE; 2017; Markham, Ontario, Canada: 287-296.
26. Fontana FA, Roveda R, Zanoni M. Tool support for evaluating architectural debt of an existing system: an experience report. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16). ACM; 2016; Pisa, Italy:1347-1349.
27. Dale MR, Izurieta C. Impacts of design pattern decay on system quality. In: Proceedings of the ACM/IEEE 8th International Symposium on Empirical Software Engineering and Measurement (ESEM '14). ACM; 2014; Torino, Italy:37:1-37:4.
28. Ghezzi G, Würsch M, Giger E, Gall HC. An architectural blueprint for a pluggable version control system for software (evolution) analysis. In: Proceedings of the 2nd international workshop on developing tools as plug-ins (TOPI '12). IEEE; 2012; Piscataway, NJ, USA:13-18.
29. Alexandru CV, Panichella S, Gall HC. Reducing redundancies in multi-revision code analysis. In: Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17). IEEE; 2017; Klagenfurt, Austria:148-159.
30. Alves N, Ribeiro LF, Caires V, Mendes T, Spínola R. Towards an ontology of terms on technical debt. In: Proceedings of the 6th International Workshop on Managing Technical Debt (MTD '14). IEEE; 2014; Victoria, BC, Canada:1-7.
31. Maldonado EDS, Shihab E. Detecting and quantifying different types of self-admitted technical debtIEEE; 2015; Bremen, Germany:9-15.
32. Bavota G, Russo B. A large-scale empirical study on self-admitted technical debt. In: Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR '16). IEEE; 2016; Austin, TX, USA:315-326.
33. Krippendorff K. Computing krippendorff's alpha-reliability. *Dep Pap (ASC)*. 2011:1-12.
34. Krippendorff K. *Content Analysis: An Introduction to Its Methodology*. Thousand Oaks, California, United States: Sage Publications; 2018.
35. Field A. *Discovering statistics using ibm spss statistics*. 5th. London: SAGE Publications; 2017.
36. Zar JH. Significance testing of the spearman rank correlation coefficient. *J Amer Stat Ass*. 1972;67(339):578-580.
37. Cohen J. *Statistical Power Analysis for the Behavioral Sciences*. New Jersey, United States: Lawrence Earlbaum Associates; 1988.
38. Gastwirth JL. The estimation of the lorenz curve and gini index. *Rev Econ Stat*. 1972;54(3):306-16.
39. Shull F, Singer J, Sjøberg DI. *Guide to Advanced Empirical Software Engineering*. Berlin, Germany: Springer; 2007.
40. Cochran WG. *Sampling Techniques*. New Jersey, United States: John Wiley & Sons; 2007.
41. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM '99); 1999; Oxford, UK:109-118.
42. Lozano A, Wermelinger M. Assessing the effect of clones on changeability. In: Proceedings of the IEEE 24th International Conference on Software Maintenance (ICSM '08). IEEE; 2008; Beijing, China:227-236.

43. Rahman F, Bird C, Devanbu P. Clones: What is that smell? *Emp Softw Eng*. 2012;17(4):503-530.
44. Tratt L. Dynamically typed languages. *Adv Comput*. 2009;77:149-184.
45. Zhai H, Casalnuovo C, Devanbu P. Test coverage in python programs. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR '19). ACM; 2019; Montreal, QC, Canada:116-120.
46. Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. An empirical investigation into the nature of test smells ACM; 2016; Singapore, Singapore:4-15.
47. Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: Proceedings of the IEEE 28th International Conference on Software Maintenance (ICSM '12) IEEE; 2012; Riva del Garda, Italy:56-65.
48. Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. Are test smells really harmful? An empirical study. *Emp Soft Eng*. 2015;20(4):1052-1094.
49. Shamshiri S, Rojas JM, Galeotti JP, Walkinshaw N, Fraser G. How do automatically generated unit tests influence software maintenance? In: Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST '18) IEEE; 2018; Västerås, Sweden:250-261.
50. Horgan JR, London S, Lyu MR. Achieving software quality with testing coverage measures. *Computer*. 1994;27(9):60-69.
51. Cannon B. Porting python 2 code to python 3. https://docs.python.org/3/howto/pyporting.html, Accessed: 2020-07-14; 2017.
52. Olbrich S, Cruzes DS, Basili V, Zazworka N. The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of the ACM/IEEE 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09). IEEE; 2009; Lake Buena Vista, FL, USA: 390-400.
53. Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Emp Softw Eng*. 2018;23(3):1188-1221.

How to cite this article: Tan J, Feitosa D, Avgeriou P, Lungu M. Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem. *J Softw Evol Proc*. 2020;e2319. https://doi.org/10.1002/smr.2319

APPENDIX A

TABLE A1 Project statistics

Project name	Commits	SLOC	Classes	Age (days)	Contributors	Fixed issues	Open issues	Fixing rate
Airavata-django-portal	743	41 192	614	663	5	1277	3225	28.37%
Airflow	6074	89 773	1044	1528	742	1387	3935	26.06%
Allura	9271	77 042	1194	3345	45	3730	4833	43.56%
Bloodhound	1238	74 658	1068	2230	9	855	5244	14.02%
Cassandra-dtest	5148	61 333	487	2619	68	3337	3957	45.75%
Chemistry-cmislib	117	5020	86	2601	3	187	215	46.52%
Cloudstack-docs-rn	197	111	0	1653	15	1	26	3.70%
Cloudstack-documentation	59	45	1	119	16	0	2	0
Cloudstack-ec2stack	341	4456	21	575	5	253	127	66.58%
Cloudstack-gcestack	332	3102	37	1789	4	116	309	27.29%
Comdev-reporter	34	1341	3	0	1	0	161	0
Couchdb-documentation	964	944	22	2179	105	17	63	21.25%
Fluo-muchos	261	983	4	1416	7	37	41	47.44%
Incubator-ariatosca	190	35 603	557	634	11	557	2467	18.42%
Incubator-milagro-mfa-server	25	4598	84	532	7	14	401	3.37%
Incubator-mxnet	9381	95 883	743	1302	677	22 142	7434	74.87%
Incubator-pagespeed-drp	40	611	10	2597	4	1	33	2.94%
Incubator-retired-cotton	56	4125	74	223	2	5	251	1.95%
Incubator-retired-wave-docs	10	183	0	994	3	0	10	0
Incubator-sdap-edge	2	4900	94	19	1	0	940	0
Incubator-sdap-nexus	73	17 056	173	388	5	89	2281	3.76%
Incubator-sdap-nexusprotp	13	48	0	350	1	0	10	0
Incubator-sdap-ningesterpy	28	1623	45	300	3	8	155	4.91%

(Continues)

TABLE A1 (Continued)

Project name	Commits	SLOC	Classes	Age (days)	Contributors	Fixed issues	Open issues	Fixing rate
Incubator-senssoft-distill	98	913	12	622	5	274	73	78.96%
Incubator-senssoft-userale-pyqt5	40	612	10	525	1	142	40	78.02%
Incubator-spot	654	7602	38	1009	20	993	729	57.67%
Incubator-superset	3922	16 279	90	1205	344	39	487	7.41%
Incubator-warble-node	29	753	7	18	1	0	75	0
Infrastructure-puppet	10 942	12 112	122	1624	60	524	1289	28.90%
Kibble	586	8103	8	386	4	71	354	16.71%
Kibble-scanners	152	3940	12	362	1	8	386	2.03%
Libcloud	6287	126 083	1394	1156	264	3692	5132	41.84%
Openwhisk-composer-python	83	1655	50	219	5	217	244	47.07%
Openwhisk-deploy-mesos	2	1373	4	249	2	0	93	0
Openwhisk-package-kafka	202	1157	9	786	11	77	123	38.50%
Openwhisk-utilities	48	364	0	515	7	2	9	18.18%
Predictionio-sdk-python	122	1487	22	1604	9	119	93	56.13%
Qpid-dispatch	2160	29 628	447	1874	17	863	1,437	37.52%
Qpid-interop-test	184	4660	59	1156	2	217	220	49.66%
Qpid-python	1004	19 285	385	4232	23	352	1482	19.19%
Steve	384	2662	6	377	4	85	276	23.55%
Tashi	449	12 311	90	1582	3	926	3639	20.29%
Trafficserver-qa	109	1083	36	724	3	51	108	32.08%
Usergrid-python	2	600	15	0	1	0	79	0
Median	187	4032	38	1092	5			
Average	1410	17 667	209	1381	57			

TABLE A2 Issue rules

ID	Description	Type	Severity	Category	Issues	Rate %
3	__init__ should not return a value	Bug	BL	Code	1	0
7	Methods and field names should not differ only by capitalization☕	CS	BL	Code	43	34.88
12	'<>' should not be used to test inequality	CS	MA	Defect	2	100
13	Jump statements should not be followed by other statements	Bug	MA	Code	47	76.60
14	Docstrings should be defined	CS	MA	Document	31 364	41.48
15	Identical expressions should not be used on both sides of a binary operator☕	Bug	MA	Code	31	19.35
16	Sections of code should not be "commented out"☕	CS	MA	Code	1558	55.58
20	Functions should not be too complex	CS	CR	Design	812	40.51
21	The 'print' statement should not be used	CS	MA	Defect	2396	56.34
23	'\\' should only be used as an escape character outside of raw strings	Bug	MA	Code	1312	68.52
24	Control flow statements should not be nested too deeply☕	CS	CR	Design	1265	41.98
25	Redundant pairs of parentheses should be removed☕	CS	MA	Code	29	44.82
26	Two branches in a conditional structure should not have exactly the same implementation☕	CS	MA	Design	320	48.75
29	Cognitive Complexity of functions should not be too high☕	CS	CR	Design	3250	40.89
31	The 'exec' statement should not be used	Bug	BL	Defect	5	0
36	A field should not duplicate the name of its containing class☕	CS	MA	Code	28	28.57
38	Statements should be on separate lines☕	CS	MA	Code	1806	53.99

(Continues)

TABLE A2 (Continued)

ID	Description	Type	Severity	Category	Issues	Rate %
40	Functions should not contain too many return statements	CS	MA	Design	706	36.40
41	Collapsible 'if' statements should be merged☕	CS	MA	Design	491	41.75
42	Function names should comply with a naming convention	CS	MA	Code	1770	45.71
45	Nested blocks of code should not be left empty☕	CS	MA	Code	180	37.22
46	Functions, methods, and lambdas should not have too many parameters	CS	MA	Code	2278	54.30
47	Files should not have too many lines of code☕	CS	MA	Design	264	31.44
49	Variables should not be self-assigned☕	Bug	MA	Defect	78	55.13
50	Lines should not be too long	CS	MA	Code	12 747	49.80
51	Python parser failure	CS	MA	Defect	19	78.95
52	Backticks should not be used	Bug	BL	Defect	5	0
54	Track uses of 'FIXME' tags☕	CS	MA	Document	173	26.59
56	Lines should have sufficient coverage by tests	CS	MA	Test	7595	35.17
57	Source files should have a sufficient density of comment lines	CS	MA	Document	4567	48.48
58	Source files should not have any duplicated blocks☕	CS	MA	Design	1346	54.38
273	Bad option value	CS	MA	Defect	22	59.09
274	Calling of not callable	CS	MA	Defect	82	79.27
276	Access of nonexistent member	CS	MA	Defect	3473	41.41
281	Syntax error	CS	MA	Defect	972	56.89
299	Too many arguments for logging format string	CS	MA	Defect	12	25.00
328	Redefined function/class/method	CS	MA	Defect	106	57.55
355	Method has no argument	CS	MA	Defect	39	43.59
360	Undefined name	CS	MA	Defect	2975	38.15
363	Method should have 'self' as first argument	CS	MA	Defect	105	30.48
364	Format string ends in middle of conversion specifier	CS	MA	Defect	2	0
368	Not enough arguments for format string	CS	MA	Defect	8	12.50
369	Too many arguments for format string	CS	MA	Defect	38	57.89
389	Access to member before its definition	CS	MA	Defect	49	61.22
390	Method hidden by attribute of super class	CS	MA	Design	23	34.78
393	Using variable before assignment	CS	MA	Defect	5	40.00
394	Undefined variable	CS	MA	Defect	6385	56.82
403	Too few arguments	CS	MA	Defect	198	19.70
411	Bad first argument given to super	CS	MA	Defect	31	38.71
414	Passing unexpected keyword argument in function call	CS	MA	Defect	91	56.04
418	Too many positional arguments for function call	CS	MA	Defect	128	28.12
424	NotImplemented raised—should raise NotImplementedError	CS	MA	Defect	45	82.22
432	Mixed tabs/spaces indentation	CS	MA	Defect	3544	34.45
438	Assigning to function call which does not return	CS	MA	Code	3	66.67
439	Raising only allowed for classes, instances, or strings	CS	MA	Defect	328	2.13
440	Bad except clauses order	CS	MA	Design	1	100

[a]ID number of the rules.
[b]Category (Code = Code Debt; Design = Design Debt; Defect = Defect Debt; Document = Documentation Debt; Test = Test Debt).
☕Also appeared in the Java study.[3]