Journal of Software Engineering Research and Development, 2025, 13:12, doi: 10.5753/jserd.2025.5722

Technical Debt in Pull Requests: Insights from Apache Projects

Felipe E. de O. Calixto [**Universidade Federal de Campina Grande** | ***felipecalixto@copin.ufcg.edu.br***]
Eliane C. Araújo [**Universidade Federal de Campina Grande** | ***eliane@computacao.ufcg.edu.br***]
Everton L. G. Alves [**Universidade Federal de Campina Grande** | ***everton@computacao.ufcg.edu.br***]

Abstract Technical Debt (TD) represents the effort required to address quality issues that affect a software system and progressively hinder code evolution over time. A pull request (PR) is a discrete unit of work that must meet specific quality standards to be integrated into the main codebase. PRs offer a valuable opportunity to assess how developers handle TD and how codebase quality evolves. In this work, we conducted two empirical analyses to understand how developers address TD within PRs and whether TD is effectively managed during PR reviews by both developers and reviewers. We examined 12 Java projects from Apache. The first study employed the SonarQube tool on 2,035 merged PRs to evaluate TD variation, identify the most frequently neglected and resolved types of TD issues, and analyze how TD evolves over time. The second study involved a qualitative analysis of review threads of 250 PRs, focusing on the types of PRs that frequently discuss TD, the characteristics of TD fix suggestions, and the reasons some suggestions are rejected. Our findings reveal that TD issues are prevalent in PRs, following a ratio of 1:2:1 (reduced: unchanged: increased). Among all TD issues, those related to code duplication and cognitive complexity are most frequently overlooked, while code duplication and obsolete code are the most commonly resolved. Regarding PR code review, we found that around 76% of review threads address TD, with code, design, and documentation being the most frequently discussed areas. Additionally, 96% of discussions include a fix suggestion, and over 80% of the discussed issues are resolved. These insights can help practitioners become more aware of TD management and may inspire the development of new tools to facilitate TD handling during PRs.

Keywords: *Technical Debt, Pull Request, Code Review, Empirical Study, Software Evolution, Mining Software Repositories*

1 Introduction

Technical Debt (Cunningham, 1992) refers to the implicit costs associated with technical shortcomings. These issues often arise from various factors, such as rushed, low-quality design decisions, a team's lack of expertise, or pressure to meet tight deadlines. Such challenges can affect both the source code and the overall development process (Lenarduzzi et al., 2019).

Two key concepts related to the cost of technical debt are: (i) the *principal*, which denotes the effort required to address all technical issues and achieve an optimal level of maintainability, either in a specific part of the software or throughout the system; and (ii) the *interest*, which reflects the extra cost incurred when implementing changes due to the presence of technical debt (Ampatzoglou et al., 2020; Avgeriou et al., 2021). In this work, we focus on the evolution of the technical debt principal; specifically, we measure how the principal fluctuates, whether increasing or decreasing. Hereafter, we abbreviate the technical debt principal as TD.

The level of TD can significantly influence the evolution and maintainability of software. High levels of TD can increase the effort required for future changes (Tan et al., 2021) and impact other critical aspects, such as security and performance (Digkas et al., 2017). Understanding how TD evolves during the development process can help identify when and why debt is introduced or eliminated, and how it affects software quality and long-term maintainability.

TD issues can manifest in various forms, including *code* (e.g., code complexity and duplication), *design* (e.g., cohesion and coupling problems), *architectural* (e.g., violations of architectural standards, scalability challenges, and performance issues), *test* (e.g., inadequate test coverage), and *infrastructure* (e.g., slow build times) (Li et al., 2015; Kruchten et al., 2019). Among these, *code* and *design-related* TD are the most frequently studied (Li et al., 2015; Coq and Rosen, 2011; Nugroho et al., 2011; Eisenberg, 2012; Vetrò, 2012; Monteith and McGregor, 2013; Griffith and Izurieta, 2014; Zazworka et al., 2014).

In this scenario, code smells are recognized as quality problems that can affect code readability and maintainability, increasing the likelihood of bugs (Yamashita and Counsell, 2013). These issues can be mitigated through refactoring (Lacerda et al., 2020) and often serve as indicators of TD within the codebase (Giordano et al., 2023).

Automated Static Analysis Tools (ASATs) analyze the source code of a project and identify quality issues that can relate to TD (Vassallo et al., 2018; Panichella et al., 2015; Trautsch et al., 2023). For instance, SonarQube [1] is a well-known tool that aggregates several code quality analyses and applies a cost estimation strategy to measure the effort (in minutes, hours, or days) required to address the detected TD issues: the *technical debt* metric [2]. Several studies have used this metric as a valid proxy to measure TD (Digkas et al., 2017; Molnar and Motogna, 2020; Nikolaidis et al., 2023).

The growth of TD can lead to consequences for both organizations and developers. For organizations, it implies software degradation, leading to delays and increased maintenance costs. For developers, it can negatively impact their confidence and morale (Besker et al., 2020).

Prior studies (Digkas et al., 2017; Molnar and Motogna,

[1] `https://www.sonarsource.com/products/sonarqube/`
[2] `https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/`

2020; Tan et al., 2021) have explored TD evolution at the granularity of software releases and commits in the main branch. These approaches are useful for observing macro-level and long-term TD management, but they obscure the actions and negotiations undertaken by developers that directly influence TD.

Conversely, a Pull Request (PR) represents a discrete, self-contained development cycle comprising a set of proposed changes. The pull request cycle involves an initial proposal, which undergoes collaborative review before being integrated. Thus, analyzing PRs enables more than just measuring TD; it offers the opportunity to understand how technical and social factors impact TD evolution, specifically: (i) the technical decisions developers make in the code (i.e., whether to resolve issues), (ii) the social interactions inherent in the code-review process (i.e., TD-related discussions), and (iii) the socio-technical decisions that emerge from those reviews (i.e., fixes applied for issues raised during the review).

In addition to ASATs, *code review*, which is commonly employed in PR-based development, also plays an important role in identifying issues related to TD. While ASATs are effective at detecting code and design-level issues, they have limitations in identifying more complex problems or those dependent on the project's context.

To address these limitations, code review enables a more comprehensive evaluation. Multiple reviewers can analyze the code for faults and quality issues (Beller et al., 2014), ensuring that the source code meets acceptable standards of readability and maintainability (Han et al., 2022). Previous studies have highlighted that most issues identified during code reviews are related to problems affecting software evolvability (Mäntylä and Lassenius, 2009; Beller et al., 2014). Platforms such as GitHub enable proper code reviews in the context of PRs validation. In that context, reviewers can select specific code snippets and initiate discussions (review threads), in which the author can participate, fostering collaboration and problem resolution.

In this work, we present two empirical studies conducted on 12 Apache Java repositories. The first is a quantitative study using the SonarQube tool that aims to investigate how TD evolves within PRs—whether it increases or decreases—and to identify which types of issues are most frequently neglected and resolved. In the second study, we delved into the review threads of the PRs, aiming to assess whether TD was effectively managed during the code-review process. The qualitative study considers a sample of PRs and investigates the frequency of TD associated with typical maintenance activities. Moreover, we identify common characteristics of the code review discussions and the outcomes they produce.

This work extends our previous study (Calixto et al., 2024). The contributions of this work are fourfold:

- A quantitative study that analyzed 2,035 merged PRs from 12 Apache Java projects. We established conclusions on the presence of TD issues, how they evolve, which issues are most common to happen in the scope of a PR, and whether time influences both TD variation and issue severity.
- A qualitative study that involved 250 PRs and 929 review threads. Our findings include identifying which maintenance activities most frequently discuss TD, the key characteristics of suggestions for addressing TD issues, and the motivations behind the rejection of such suggestions.
- We assembled a dataset of 250 PRs and another containing 929 review threads, both manually labeled. These datasets include features related to the primary objective of the PR, whether the review threads discuss technical debt, whether suggestions are provided, among other aspects.
- A reproduction package with all artifacts of our study (Calixto et al., 2025). This package can help other researchers in similar fields.

The rest of the paper is organized as follows. Section 2 provides background on some important concepts related to the SonarQube tool and PR-based development. Section 3 introduces the design of the studies, results, and conclusions. Section 4 addresses potential threats to validity. Section 5 discusses the related work, while Section 6 provides some concluding remarks and possible future work.

2 Background

This section discusses concepts that are fundamental to this paper. In Section 2.2, we delve into PR-based development and present the methodology used for investigating TD within this framework. Sections 2.1 and 2.1.1 provide an overview of the SonarQube tool, elucidating its method for measuring TD and introducing the SQALE model, used by SonarQube.

2.1 SonarQube

SonarQube is an open-source ASAT that can identify around 5,000 code quality issues in projects of more than 30 programming languages, including Java. Widely adopted by over 400,000 organizations globally (SonarSouce, 2022), it has been extensively used in empirical software engineering research (Avgeriou et al., 2021; Saarimäki et al., 2019; Marcilio et al., 2019; Lenarduzzi et al., 2020; Yu et al., 2023). In this study, SonarQube is employed to measure code and design-related TD issues (Nikolaidis et al., 2023) within PRs.

SonarQube has a set of best-practice rules, referred to as *coding rules*[3], which generate warnings (issues) when violated. These issues are categorized into three types: *BUG*, representing programming faults that may cause errors or unexpected behavior during execution; *CODE SMELL*, indicating sub-optimal code quality that impacts maintainability; and *VULNERABILITY*, denoting security-related flaws. The *technical debt* metric primarily focuses on *CODE SMELL* issues. Additionally, issues are classified by severity levels: *INFO*, *MINOR*, *MAJOR*, *CRITICAL*, and *BLOCKER*, listed in ascending order of severity.

When an issue is first detected, it is labeled as *OPEN* (unfixed). In subsequent analyses, if an issue previously marked

[3] `https://rules.sonarsource.com/`

as *OPEN* is no longer detected or has been properly addressed, SonarQube marks it as *CLOSED*[4].

To estimate the effort required to resolve issues, SonarQube employs a heuristic () that ranges from *TRIVIAL* (e.g., removing unused imports without affecting logic) to *COMPLEX* (e.g., eliminating cyclic dependencies between packages, which may require application redesign). For Java, the estimated effort ranges from 5 minutes for *TRIVIAL* issues to 1 day for *COMPLEX* ones.

2.1.1 The SQALE Method

SonarQube employs the SQALE quality model, which posits that assessing quality is equivalent to measuring TD (Letouzey, 2012). This model defines quality as the degree to which a system adheres to specified requirements, using remediation costs to derive quality indicators. The SQALE method further organizes requirements into categories and subcategories related to code quality, enabling the identification of which aspects are most affected by issues. For each requirement, a remediation function is applied to estimate the effort, quantified in time, required to fulfill the requirement within a specific system component, such as a module, file, or class. This approach provides a structured framework for evaluating and addressing code quality issues.

```
public HashMap<String,String> tableConfig;
```

Make tableConfig a static final constant or non-public and provide accessors if needed.

Figure 1. This example illustrates a violation of SonarQube's rule S1104.

Figure 1 illustrates a SonarQube violation associated with rule S1104, which states that class attributes must not be public. To address this issue, one may either convert the attribute into a constant using the `static final` modifiers or change its visibility to `private` (refer to remediation details). In this scenario, the remediation function estimates a resolution time of 10 minutes per occurrence.

2.2 PR-Based Development and Code Review

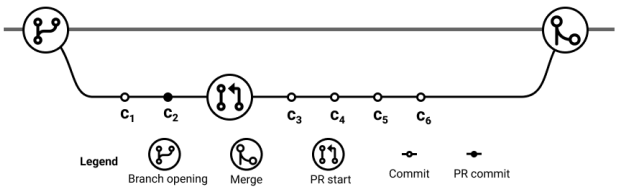

Figure 2. Example of a PR.

PR-based development has been widely adopted by both commercial organizations and open-source projects, enabling developers in distributed teams to implement changes autonomously. Figure 2 provides an overview of this model. Developers can fork a development branch and implement code changes through a series of commits (c_1 to c_2). The implemented changes are typically associated with a primary maintenance activity, which we refer to as the PR's primary goal. This activity can encompass one of three types of maintenance tasks: requirement changes (implementing new features or modifying existing ones), bug fixes, or code improvements.

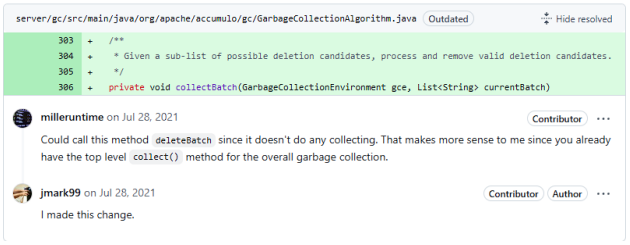

Figure 3. Example of a review thread.

Subsequently, they submit a PR to be evaluated by other developers (reviewers) through code reviews. The review process aims to ensure a desirable level of quality in various factors, such as readability and maintainability (Han et al., 2022), by identifying defects and code quality issues (Beller et al., 2014). On the GitHub platform, reviewers can select specific code snippets and initiate a review thread (Figure 3) about them or leave general comments without referencing specific code snippets. In our qualitative study, we chose to evaluate only review threads, as they have a well-defined scope (with clear start and end points) and are associated with specific code snippets, which allowed us, among other things, to later determine whether an issue was resolved or not.

After inspecting the code, reviewers can initiate a review thread to request additional information if they have questions about the code or to suggest modifications. If additional work is required, the developer implements the changes and adds new commits (c_3 to c_6), leading to a potential code merge upon acceptance of the changes.

This study aims to explore the evolution of TD within the context of PRs. For this analysis, we use the SonarQube tool. We start by examining the commit that initiates the PR, known as the *PR commit*. This commit is typically the latest one made before the PR is submitted. In Figure 2, this is represented by commit c_2. Our analysis covers all subsequent commits from the *PR commit* up to the final commit in the PR branch (c_2 to c_6 in the example). We hypothesize that developers may enhance code quality during these commits, either independently or in response to feedback from code reviews, as these changes are part of the review process. In addition, in the qualitative study, we evaluate only review threads, which are blocks of code reviews that include the reviewed code snippet.

3 The Empirical Studies

The objective of this study is to investigate the evolution of TD within the context of a single PR and to understand how developers' behaviors and review dynamics influence TD management both in the code and during collaborative code reviews. By evolution of TD, we refer to the variations that occur within a PR—whether TD increases, decreases, or remains unchanged—and similarly for code reviews, whether TD-related issues raised are discussed, resolved, or ignored throughout the PR lifecycle. Ultimately,

[4] `https://docs.sonarsource.com/sonarqube/10.0/user-guide/issues/`

we seek to uncover patterns that reveal whether TD is being effectively addressed or neglected during the PR process and how these patterns correlate with code quality and architectural implications.

To achieve this goal, we conducted two empirical studies. In the first study, we employed the SonarQube tool to analyze merged PRs, aiming to assess the extent to which developers manage TD in the code. In the second study, we sought to explore the same idea but from the perspective of code reviews. In both studies, we adopted a two-stage purposive sampling approach, following the guidelines for sampling in software engineering established by Baltes and Ralph (2022):

- **Project selection**. We focused on twelve Java repositories from the Apache Software Foundation for three reasons: (i) they have been extensively studied in prior literature (Lenarduzzi et al., 2020, 2021; Tan et al., 2021; Digkas et al., 2022; Zabardast et al., 2022; Coelho et al., 2021); (ii) they represent a mature software development environment with active, high-quality codebases; and (iii) they compiled successfully on our machines, a requirement for automated SonarQube analysis.
- **Pull-request selection**. From each of these twelve projects, we first extracted all PRs that (i) were merged into the main branch and (ii) fell within our specified time window, ending on February 29, 2024, at 23:59:59. We then applied additional filters, retaining only those PRs whose commits were fully analyzed by SonarQube, yielding 2,035 merged PRs for the first study. For the second study, we first selected all PRs from the 2,035 that contained at least one review thread, resulting in a total of 712 PRs. From this set, we then drew a stratified random sample of 250 PRs by project.

Details of each sampling step are provided in the following subsections.

3.1 Quantitative Analysis - The SonarQube Study

In our first empirical study, we ran a quantitative analysis to explore TD within PRs using the SonarQube tool. To guide the study, we defined the following research questions:

RQ1: How does TD evolve within PRs? Our focus is to evaluate how TD varies within PRs, specifically examining whether it decreases, increases, or remains stable. To conduct this analysis, we propose a TD Variation metric designed to systematically quantify and evaluate these changes.

RQ2: Which TD issues are most commonly resolved and neglected within PRs? Given the comprehensive catalog of coding rules in SonarQube, we hypothesize that developers are more likely to address issues related to specific rules while overlooking others. Our goal is to identify these patterns and to gain a better understanding of the factors that influence developers' decisions when resolving issues.

RQ3: How does TD within PRs evolve over time? We aim to understand whether time influences TD, exploring this relationship by considering the variation of TD and the evolution of issue severity.

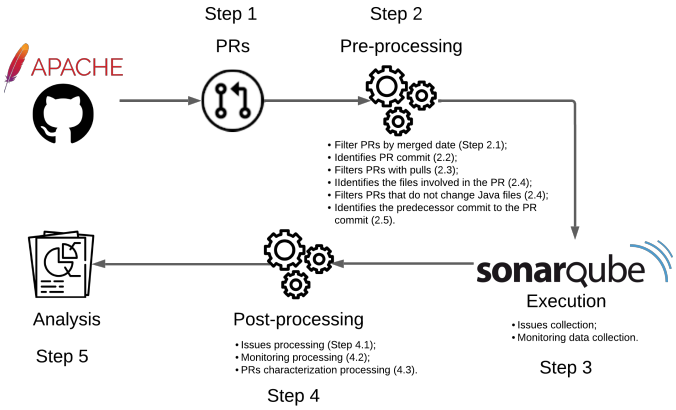

Figure 4. Data collection and processing.

Figure 4 provides an overview of the methodology used to collect and process data in this study. The process begins with the collection of all merged PRs from each project (Step 1), followed by filtering and processing through multiple stages (Step 2). For each PR, we executed SonarQube on the commits ranging from the PR commit to the final commit of the PR branch (Step 3). Subsequently, we performed post-processing (Step 4) to prepare the data for analysis (Step 5). The following sections elaborate on steps 1-4. A summary of our dataset, including the selected projects and PRs, is presented in Table 1.

3.1.1 Dataset

Since Apache projects are hosted on GitHub, we leveraged both the GitHub REST[5] and GraphQL[6] APIs to retrieve PRs from each repository. The extraction was carried out between March 13, 2024, and March 14, 2024, using a set of specific filters to standardize our dataset:

- **Only merged PRs**. We considered only merged PRs, as these adhere to a baseline level of code quality and have been approved by project reviewers.
- **Merged by February 29, 2024, at 23:59:59**. A cutoff point was established to ensure data homogeneity.

Our dataset initially comprised 7,494 merged PRs (Table 1). Through successive filtering stages, we excluded: (i) PRs containing merge commits except in the final commit, and (ii) PRs that did not modify Java source files. This refinement process yielded 3,854 candidate PRs. From these, we successfully executed our analysis tool on 2,035 PRs, which constitutes our final sample. The details of the filtering process and the execution of SonarQube are provided in the following sections.

3.1.2 Dataset Preparation

After gathering the projects and PRs, we applied a five-step pre-processing pipeline to the PRs dataset. First, we filtered the PRs based on their merge dates (Step 2.1). Next, we identified the PR commits (Step 2.2) by using both the commit creation dates and the PR opening date—defining a PR commit as the most recent commit prior to the PR's creation. In

[5] https://docs.github.com/en/rest
[6] https://docs.github.com/en/graphql

Table 1. Dataset of projects and PRs.

Project	# Classes	NCLOC	# Issues	# PRs			
				Mining	After processing	Post-execution	With issues
accumulo	5,164	440,441	158,730	2,295	1,512	994	952
cayenne	4,716	318,428	1,712	429	336	55	55
commons-collections	839	67,690	2,280	241	68	31	28
commons-io	288	30,501	8,285	374	95	76	73
commons-lang	918	95,929	9,673	496	192	128	122
helix	2,106	189,487	35,978	1,374	669	278	276
httpcomponents-client	879	76,964	5,179	310	159	126	120
maven-surefire	3,036	110,481	1,833	331	71	25	25
opennlp	2,478	155,900	10,731	375	153	96	94
struts	3,419	234,331	9,877	716	407	145	140
wicket	5,254	251,505	1,000	441	157	47	40
zookeeper	1,542	131,712	2,289	112	35	34	34
Total			247,365	7,494	3,854	2,035	1,959

Step 2.3, we excluded PRs that involved code pulls from other branches, retaining only those that either did not incorporate code pulls or did so solely in the final commit. This strategy ensured that our analysis focused exclusively on code modifications generated within the PR, thereby avoiding external influences that could skew our results.

During Step 2.4, we identified the modified files and eliminated any PRs that did not alter Java files. Finally, in Step 2.5, we determined the commits preceding the PR commits—a necessary step to run SonarQube on a commit before the PR so that issues detected can be classified as *pre-existing* (as discussed in Section 3.1.4). To identify this "preceding commit," we employed two strategies: (i) if a commit exists before the PR commit within the branch, we designate the last commit before the PR commit as the preceding commit; (ii) if no such commit is present within the branch, we use the concept of a parent commit[7], considering the parent of the PR commit as the preceding commit. In the example shown in Figure 2, c_1 is the preceding commit.

3.1.3 Dataset Processing

In our study, we employed SonarQube (version 10.0.0.68432) in conjunction with the SonarScanner[8] tool (version 4.8.0.2856). SonarScanner was used to execute all SonarQube analyses and submit the results to SonarQube. Although SonarScanner can be integrated natively with build tools (e.g., SonarScanner for Maven), it also supports standalone execution via a command-line interface (CLI), making it independent of specific build tools. Since most of the analyzed projects lacked native SonarQube integration and the necessary configurations for the SonarScanner plugin, we opted to use the SonarScanner CLI.

To perform the SonarScanner analysis (Step 3), the compiled code of the project is required. Consequently, we downloaded the project code corresponding to each commit (as described in Section 2.2) for every PR and compiled it to enable SonarQube execution. During this stage, 1,819 PRs were excluded due to compilation errors, such as missing dependencies or dependency version conflicts.

Given the large size of the projects, both the compilation and the execution of the SonarQube analysis were time-consuming — the average execution time per commit was 6 minutes and 18 seconds. Given that each PR consists of multiple commits, managing execution time posed a significant challenge. To mitigate this issue, we partitioned the PRs into subsets and executed each subset on a separate virtual machine (VM). We utilized Oracle VM VirtualBox[9] to configure four VMs, each configured as follows:

- 8 GB of RAM;
- 4 processor cores;
- 150 GB of storage;
- Ubuntu 22.04.1 LTS operating system.

Additionally, we developed Python scripts to automate the entire workflow, including downloading the commit source code, compiling it, running SonarQube, and extracting the identified issues via the SonarQube API[10].

3.1.4 Analysis

Following the execution of SonarQube, we processed data related to issues (Step 4.1), execution monitoring (Step 4.2), and PRs (Step 4.3).

In the issue processing phase, we categorized each detected issue along two dimensions: *origin* and *status*. The origin attribute distinguishes whether an issue existed prior to the PR (*pre-existing* issue) or was introduced during the PR (*new* issue). The status attribute indicates whether the issue was resolved (*fixed* issue) or remains unresolved (*unfixed* issue).

To determine an issue's origin and status, we analyze the sequence of commits described in Section 2.2. This analysis enables us to precisely determine the commit in which an issue was introduced or resolved. For instance, Pre-existing issues are identified in the commit preceding the PR commit, while new or fixed issues are detected in the commits

[7] `https://git-scm.com/docs/git-commit-tree`

[8] `https://docs.sonarsource.com/sonarqube/10.0/analyzing-source-code/scanners/sonarscanner/`

[9] `https://www.virtualbox.org/`

[10] `https://docs.sonarsource.com/sonarqube/10.0/extension-guide/web-api/`

that follow. This granular approach makes it possible to track changes in TD measurement, as new issues may be addressed within the PR itself. This method contrasts with studies that only compare code states before and after a PR, as they do not track code evolution at the commit level (Panichella et al., 2015; Zou et al., 2019; Nikolaidis et al., 2023).

For each issue, we extracted the fields described below:

rule: Identifier of the violated SonarQube rule.
severity: Severity level, on a scale from least to most severe (*INFO*, *MINOR*, *MAJOR*, *CRITICAL*, *BLOCKER*).
type: Classification of the issue (*CODE SMELL*, *BUG*, *VULNERABILITY*).
debt: Estimated effort (in minutes) required to resolve the issue.
origin: Issue's origin (*PRE-EXISTING*, *NEW*).
status: Issue status: *OPEN* if unresolved, *CLOSED* if resolved.
file: File affected by the issue.
ncloc: Non-Commented Lines of Code (NCLOC) of the affected file.

In the monitoring phase (Step 4.2), we recorded the execution time for each commit in every PR to estimate the duration required for SonarQube analyses. This data can serve as a valuable reference for other studies using the same tool, aiding in more efficient experiment planning.

For PR characterization (Step 4.3), we collected several features, including added lines, removed lines, code churn (sum of added and removed lines), time until merge, number of modified files, and number of commits. This characterization covered all 2,035 PRs analyzed.

In summary, we assembled three datasets: (i) issues, (ii) monitoring, and (iii) PRs. We then performed a detailed characterization of the issues and PRs to understand the distributions of their respective features. The issues dataset was subsequently used to address our research questions, with the analysis detailed in the following section.

3.1.5 Metrics

To address RQ1, we introduced a normalized metric for TD to account for variations in project sizes. For a given PR_i that modifies k files, where each file j contains n_j issues, the TD density (TDD_i) is given by:

$$TDD_i = \frac{\sum_{j=1}^{k} \sum_{a=1}^{n_j} TD_{aj}}{\sum_{j=1}^{k} NCLOC_j} \quad (1)$$

Here, TD_{aj} represents the TD associated with issue a in file j, and $NCLOC_j$ denotes the number of non-comment lines of code () in file j.

To assess the evolution of TD within PRs, we introduced a metric that quantifies variation in TD. This variation is classified into three categories: *reduced*, *unchanged*, or *increased*. For a given PR_i that modifies k files, the TD variation, denoted as TDV_i is defined as:

$$TDV_i = \sum_{j=1}^{k} \left(\sum_{a=1}^{n_j} TD_{aj}^{New\ AND\ Unfixed} - \sum_{a=1}^{n_j} TD_{aj}^{Pre\text{-}existing\ AND\ Fixed} \right) \quad (2)$$

In this equation, $TD_{aj}^{New\ AND\ Unfixed}$ denotes the TD associated with a new unfixed issue a in file j, while $TD_{aj}^{Pre\text{-}existing\ AND\ Fixed}$ refers to the TD related to pre-existing fixed issue a in file j. The TD variation is computed as the difference between the final (new TD) and initial (pre-existing TD) states. Since the only differences arise from the unresolved new debt and the addressed pre-existing debt, the expression simplifies to Equation 2. The TD variation is categorized as follows:

Reduced: If $TDV_i < 0$, indicating a decrease in TD after the PR.
Unchanged: If $TDV_i = 0$, indicating no change in TD.
Increased: If $TDV_i > 0$, indicating an increase in TD after the PR.

To further analyze the evolution of TD, we introduced two additional metrics: *Pre-existing TDV* and *New TDV*. These metrics enable us to distinguish TD variation by its origin. For a given PR_i that modifies k files, the *Pre-existing TDV* and *New TDV* are defined as follows:

$$TDV_i^{Pre\text{-}existing} = \sum_{j=1}^{k} \sum_{a=1}^{n_j} TD_{aj}^{Pre\text{-}existing\ AND\ Fixed} \quad (3)$$

$$TDV_i^{New} = \sum_{j=1}^{k} \sum_{a=1}^{n_j} TD_{aj}^{New\ AND\ Fixed} \quad (4)$$

Here, $TD_{aj}^{Pre\text{-}existing\ AND\ Fixed}$ denotes the TD associated with a pre-existing fixed issue a in file j, while $TD_{aj}^{New\ AND\ Fixed}$ indicates the TD related to a new fixed issue a in file j. Both $TDV_i^{Pre\text{-}existing}$ and TDV_i^{New} can be categorized as follows:

Reduced: A value greater than zero indicates that TD decreased after the PR.
Unchanged: A value of zero indicates that TD remained constant after the PR.

To address RQ2, we ranked issue types (rules) based on their positions in each project's top 10 frequency list. This approach was necessary due to the uneven distribution of issues across our dataset. For example, the `accumulo` project accounts for 64.12% of the identified issues—an expected result given its size and representation (48.60% of total PRs)—while other projects contribute less than 36% of the issues. To account for these disparities, we introduced a metric that aggregates the rule's rankings of each project's top 10. For a given rule i, its $PositionScore_i$ is defined as follows:

$$PositionScore_i = \sum_{j=1}^{k} Position_j \quad (5)$$

Here, $Position_j$ represents the ranking of the rule within the top 10 for project j. If a rule does not appear in a project's top 10, we assign it a default position of 11 as a penalty. The $PositionScore$ is then computed across all k projects—in our study, 12 projects. Under this scheme, a rule can achieve a score as low as 12 (if it ranks first in every project) and as high as 131 (if it ranks tenth in one project and falls outside the top 10 in all others). Lower $PositionScore$ values indicate a higher overall ranking. This metric allows us to rank rules consistently across all projects.

Using the resulting $PositionScore$ values, we aggregate the top 10 fixed and top 10 unfixed rules across all projects, ultimately identifying the five rules with the lowest $PositionScore$, which correspond to the most frequently encountered rules across the dataset.

Regarding RQ3, we analyzed the lifetime of each project. For that, we used the PR creation date as our time reference. Since the time span analyzed varies across repositories, we applied min-max normalization to scale the time values between 0 and 1, with 0 representing the start and 1 representing the end. In our study, min-max normalization is defined by the following formula:

$$NormTime = \frac{Date - \min(Date)}{\max(Date) - \min(Date)} \quad (6)$$

Here, $NormTime$ represents the normalized time, $Date$ is the date being normalized, $\min(Date)$ denotes the earliest date, and $\max(Date)$ denotes the most recent date. The differences are calculated as the number of days between the dates.

3.1.6 Results and Discussion

In this section, we present the results and discuss the findings of our quantitative study using SonarQube and research questions Q1-Q3.

Data Characterization

Before addressing our research questions, we performed an exploratory data analysis to gain insights into the distribution of PRs and their associated issues. This analysis aimed to characterize the typical characteristics of PRs and issues included in our study.

Our findings revealed that most PRs involve modifications to a small number of files (a median of 2 to 4) and consist of a few commits (a median of 1 to 2), although outliers modifying hundreds of files were also observed. These results underscore that a typical PR represents a brief development cycle by a single developer, targeting specific parts of the system.

Regarding issue characterization, we observed that our dataset comprises 264 distinct rules—out of the 622 rules detectable by SonarQube—with 202 rules (76.51%) having at least one fixed instance. Most of the issues (94.38%) were classified as *CODE SMELL*, while 48.43% of the issues had low severity (*INFO* and *MINOR*). However, 73.04% of total TD stems from issues with medium to high severity (*MAJOR*, *CRITICAL*, and *BLOCKER*). As expected, over 96% of issues were pre-existing, with only slightly more than 4% being new ones. Notably, just 5.88% of the issues were fixed.

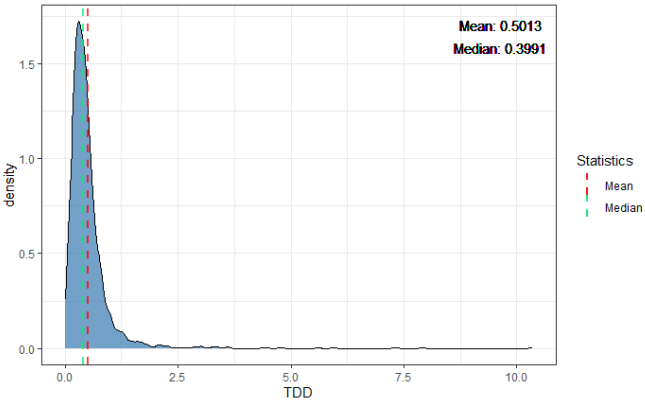

Figure 5. Distribution of TDD per PR.

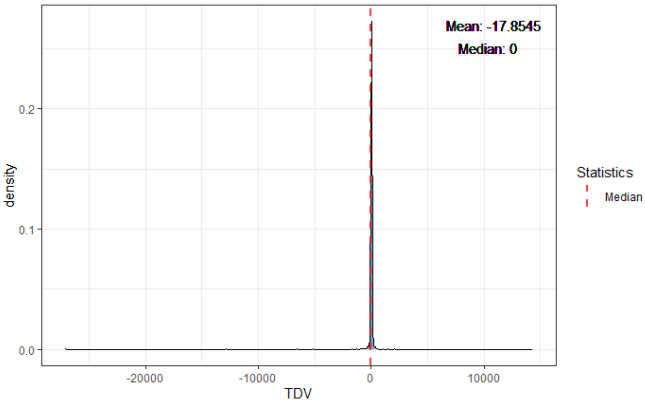

Figure 6. Distribution of TDV per PR.

In summary, the majority of issues are classified as *code smells*, highlighting code concerns that, while not directly impairing system functionality, may affect maintainability and readability. While the distribution of instances across severity levels is relatively balanced, a substantial portion of TD is concentrated in high-severity issues, likely due to their greater complexity and the effort required for resolution.

Several factors may contribute to the predominance of pre-existing issues. Although PRs generally affect only a few files, these files can be extensive and harbor numerous issues that may be overlooked or deprioritized by developers, in turn resulting in a high number of unfixed issues. Moreover, a typical PR that modifies hundreds of files further exacerbates the situation, as it becomes impractical for a developer to address issues across such a broad scope.

RQ1: How does TD evolve within PRs?

To investigate how TD evolves within PRs, we employed two metrics—TD Density (TDD) and TD Variation (TDV), as detailed in Section 3.1.5. The distributions of these metrics are illustrated in Figures 5 and 6.

TDD quantifies the amount of TD per NCLOC in a PR. Our dataset shows an average TDD of 0.5013 and a median of 0.3991. Given the median of 651 NCLOC, these values suggest that most PRs have a moderate to high TD density — translating into an estimated rework time of over four hours to resolve all TD issues. This considerable effort implies that, in many cases, addressing every issue may not be feasible; it might be more effective for developers to prioritize fixing those issues with higher severity.

`wicket` #676 exhibited the highest TDD of 10.34, representing 600 minutes of TD across 58 NCLOC. This PR had

a code churn of 27 (12 lines added and 15 lines removed) and modified two files, each containing one issue. Both issues violated rule S110[11] (*rule description: inheritance tree of classes should not be too deep*), and each incurred 300 minutes of TD. These issues were pre-existing and remained unfixed, indicating they were present in the code prior to the PR and were not resolved during it. Rule S110 sets a default threshold of 5 for inheritance tree depth, as deeper inheritance hierarchies can lead to increased complexity. For each class exceeding this threshold, the rule assigns a TD of 4 hours plus an additional 30 minutes for each level beyond the threshold. In this case, both classes had a depth of 7, resulting in significant TD concentrated in a small amount of code (NCLOC).

The TDV metric quantifies changes in TD from the start to the end of a PR. It measures the net difference in TD (in minutes): positive values indicate an increase, negative values a decrease, and zero indicates no change. Figure 6 illustrates the distribution of TDV, which is concentrated around zero (median), indicating that most PRs exhibit little or no variation in TD. Nonetheless, significant outliers exist, primarily driven by PRs that modify dozens to hundreds of files. Among these outliers, `struts` #799 recorded the lowest TDV of -27,133 minutes, representing a complete elimination of TD within the PR. This PR involved modifications to 484 files and 90,610 lines (90,609 deletions and 1 addition). According to its description, the PR removed unused files and obsolete plugins, meaning the TD reduction resulted solely from deletions rather than active resolution of issues. Among the top 10 cases with the largest TD reductions, we observed that six PRs involved file deletions, suggesting that a portion of TD reduction occurs incidentally rather than as a deliberate effort to address TD.

In contrast, the PR with the highest increase in TD was `accumulo` #2022, which saw an addition of 14,300 minutes of new TD. This PR changed 151 files and modified a total of 1,919 lines (with 985 additions and 934 deletions). It introduced 14,300 minutes of new TD, with none of this new debt being fixed, while the pre-existing TD amounted to 67,767 minutes, also with no remediation. Among the new TD issues in this PR, the most frequent violations were:

S117[12]**:** Local variable and method parameter names should comply with a naming convention (385 instances).
S2589[13]**:** Boolean expressions should not be gratuitous (383 instances).
S101[14]**:** Class names should comply with a naming convention (276 instances).
S1125[15]**:** Boolean literals should not be redundant (272 instances).

Together, these four rules accounted for 1,316 of the 2,114 new instances and 7,340 of the 14,300 minutes of TD (51.33%). Rules S117 and S101 pertain to naming conventions, whereas S1125 and S2589 address redundancy in boolean expressions.

Analysis of outliers suggests that sharp reductions in TD do not always result from developers' explicit efforts to address issues; instead, these decreases may occur incidentally. Conversely, substantial increases in TD are more often associated with the introduction of new functionalities.

To categorize PRs according to TD variation—whether reduced, increased, or unchanged—we computed the proportion of PRs with a TDV below zero (indicating reduction), equal to zero (indicating no change), and above zero (indicating an increase). Owing to the variability in the number of PRs and issues among projects, we first calculated these percentages for each project and then derived the overall average percentages.

Table 2. Percentages of TDV by projects.

Project	TDV		
	Reduced	**Unchanged**	**Increased**
accumulo	26.16%	46.95%	26.89%
cayenne	12.73%	56.36%	30.91%
commons-collections	35.71%	53.57%	10.71%
commons-io	32.88%	50.68%	16.44%
commons-lang	13.93%	78.69%	7.38%
helix	19.20%	36.96%	43.84%
httpcomponents-client	19.17%	55.83%	25.00%
maven-surefire	24.00%	68.00%	8.00%
opennlp	44.68%	32.98%	22.34%
struts	37.14%	36.43%	26.43%
wicket	10.00%	57.50%	32.50%
zookeeper	17.65%	52.94%	29.41%
Mean	**24.44%**	**52.24%**	**23.32%**

Table 2 presents the TDV percentages for each repository analyzed, along with the overall average. On average, 24.44% of the PRs reduced the TD, 52.24% kept the TD unchanged, and 23.32% increased the TD. These values approximate a 1:2:1 ratio, indicating that, for every four PRs, one reduces the TD, two keep the TD unchanged, and one increases the TD. Given that 98.57% of the PRs have at least one pre-existing issue, the PRs that leave the TD unchanged neglect a certain amount of TD. Therefore, by considering both the PRs that keep the TD unchanged and those that increase it, we observe that 75.56% of the PRs neglect some amount of TD.

Table 3. Percentages of pre-existing TDV by projects.

Project	Pre-existing TDV	
	Unchanged	**Reduced**
accumulo	57.64%	42.36%
cayenne	75.93%	24.07%
commons-collections	53.85%	46.15%
commons-io	62.50%	37.50%
commons-lang	81.15%	18.85%
helix	56.51%	43.49%
httpcomponents-client	69.49%	30.51%
maven-surefire	68.00%	32.00%
opennlp	43.96%	56.04%
struts	51.80%	48.20%
wicket	64.10%	35.90%
zookeeper	67.65%	32.35%
Mean	**62.71%**	**37.29%**

We also performed TDV analysis by issue origin, distin-

[11] `https://rules.sonarsource.com/java/RSPEC-110/`

Table 4. Percentages of new TDV by projects.

Project	New TDV	
	Unchanged	**Reduced**
accumulo	75.34%	24.66%
cayenne	90.91%	9.09%
commons-collections	66.67%	33.33%
commons-io	84.00%	16.00%
commons-lang	85.00%	15.00%
helix	75.86%	24.14%
httpcomponents-client	94.74%	5.26%
maven-surefire	100%	0.00%
opennlp	95.83%	4.17%
struts	91.43%	8.57%
wicket	94.12%	5.88%
zookeeper	78.57%	21.43%
Mean	**86.04%**	**13.96%**

guishing between pre-existing issues (those created before the PR) and new issues (those added in the current PR). Tables 3 and 4 show the variation of pre-existing TDV and new TDV, respectively. A value below zero signifies a reduction in the specific TD, while a value equal to zero means the TD remained unchanged.

For pre-existing TD, on average, 37.29% of PRs reduce some amount of TD, whereas the majority (62.71%) neglect it. For new TD, the trend is less positive: only 13.96% of PRs reduce some amount of new TD, which indicates that a significant 86.04% of these PRs fail to resolve any of the newly introduced issues before merging. For new TDV, a reduction occurs when at least one of the added issues is fixed.

A notable exception is the `maven-surefire` project, which exhibits distinct behavior. In this project, only seven PRs introduced new TD, none of which showed a reduction. Consequently, 100% of the new TD remained unchanged.

***RQ1**: Almost all PRs (96.26%) contain some level of TD, with changes across PRs following an approximate 1:2:1 ratio. This means that one in every four PRs reduces TD, two keep it unchanged, and one increases TD. Among PRs with pre-existing debt, 37.29% reduce it, whereas only 13.96% of those introducing new debt manage to reduce it within the PR.*

RQ2: Which TD issues are most commonly resolved and neglected within PRs?

To address RQ2, we aggregated the top 10 most frequently fixed and unfixed rules per project using the *PositionScore* metric (Section 3.1.5). Table 5 presents the five fixed and five unfixed rules with the lowest *PositionScore* values, i.e., the most prevalent across the projects. As observed, all these common rules are classified as *CODE SMELL* and are predominantly of medium to high severity, indicating that developers often encounter or introduce code quality issues whose remediation may be more labor-intensive.

Among these, rule S1192 has the lowest *PositionScore* for both fixed and unfixed rules. For unfixed rules, S1192 ranks first in eight projects and within the top three in the remaining four; for fixed rules, it appears in the top position in two projects and within the top three in six projects. Rule S1192

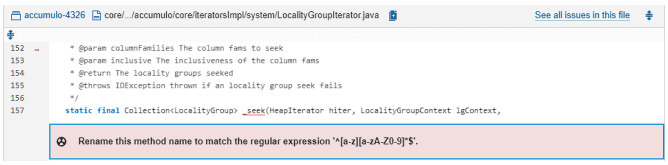

Figure 7. Example of a violation of the rule S100 in `accumulo` #4326.

concerns the duplication of `String` literals—a violation that forces developers to modify the same value in multiple locations during maintenance. The recommended solution is to extract the literal and convert it into a constant, thereby ensuring consistency.

Our results for rule S1192 suggest that developers may not always prioritize its resolution, possibly due to its perceived lower impact, especially when it appears in test classes. Anquetil et al. (2022) observed that while approximately 42.9% of regular methods use literals, over 82.2% of test methods contain them. In our study, 68.92% of the instances of this rule occurred in test classes, with only 31.08% in non-test classes.

Other TD-related rules that developers frequently overlook are S100 and S106. Rule S106 addresses the use of standard output for logging purposes, and approximately 97.38% of its 6,413 occurrences remain unfixed. Analyzing the PRs with the highest number of instances reveals that, in `accumulo` #1433, some of the issues occurred in a file named `Main.java`, which appears to be a CLI. Similarly, `accumulo` #2180 modifies code snippets associated with the mentioned issue and also seems to be related to a CLI. In contrast, robust and widely used CLIs such as Maven[16] and Gradle[17] adhere to best practices by employing dedicated loggers for output. However, in the mentioned project, this practice was not followed.

Rule S100 addresses method naming conventions. The method names must comply with the regular expression illustrated in Figure 7, meaning the first character should be a lowercase letter, and the remaining characters can be either letters (in any case) or numbers. Figure 7 provides an example from `accumulo` #4326, where a method name incorrectly begins with an underscore ("_") rather than a lowercase letter. We observed that several Apache projects include method names with underscores, suggesting that some developers do not consider this to be an issue. However, since rule S100 also ranks among the most frequently fixed, we identified some PRs with a high number of fixed instances of this type. For instance, `httpcomponents-client` #202 fixes 16 occurrences of this issue, specifically, method names starting with uppercase letters. One commit message even stated, "Use camelCase for Java method names – always," underscoring the expectation that method names follow the camelCase convention.

Rule S3776, which evaluates cognitive complexity, appears among both fixed and unfixed rules. However, our analysis of some PRs indicates that reducing complexity is often achieved by removing entire methods or classes. For example, in `struts` #657, a developer explicitly addressed this issue, as reflected in the comment "[...] good to see the cognitive complexity reduced afterwards" and the PR de-

[16] `https://github.com/apache/maven`
[17] `https://github.com/gradle/gradle`

Table 5. Top 5 fixed rules and top 5 unfixed rules by PositionScore metric.

	PositionScore	Rule	Description	Severity	Type
Unfixed rules	22	S1192	String literals should not be duplicated	CRITICAL	CODE SMELL
	85	S3776	Cognitive Complexity of methods should not be too high	CRITICAL	CODE SMELL
	100	S100	Method names should comply with a naming convention	MINOR	CODE SMELL
	100	S106	Standard outputs should not be used directly to log anything	MAJOR	CODE SMELL
	101	S117	Local variable and method parameter names should comply with a naming convention	MINOR	CODE SMELL
	PositionScore	**Rule**	**Description**	**Severity**	**Type**
Fixed rules	41	S1192	String literals should not be duplicated	CRITICAL	CODE SMELL
	84	S1874	"@Deprecated" code should not be used	MINOR	CODE SMELL
	90	S112	Generic exceptions should never be thrown	MAJOR	CODE SMELL
	94	S2293	The diamond operator ("<>") should be used	MINOR	CODE SMELL
	102	S3776	Cognitive Complexity of methods should not be too high	CRITICAL	CODE SMELL

Noncompliant Code Example

```
List<String> strings = new ArrayList<String>();  // Noncompliant
Map<String,List<Integer>> map = new HashMap<String,List<Integer>>();  // Noncompliant
```

Compliant Solution

```
List<String> strings = new ArrayList<>();
Map<String,List<Integer>> map = new HashMap<>();
```

Figure 8. Example of a compliant and non-compliant code for rule S2293.

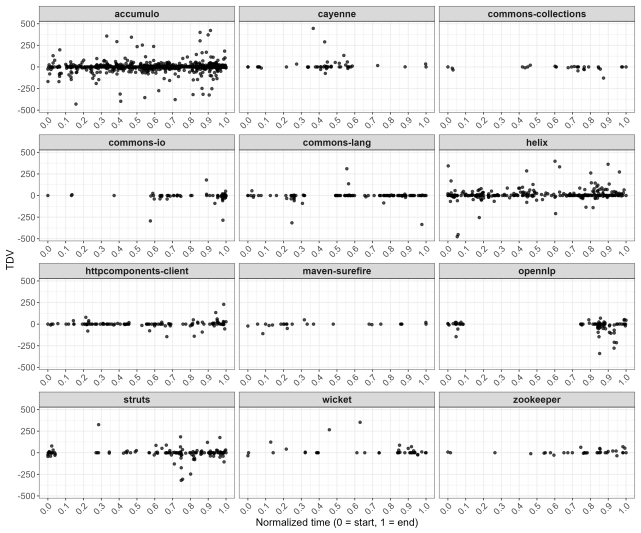

Figure 9. TDV evolution over normalized time by repository.

scription "Improve readability of XmlConfigurationProvider class." Among the projects analyzed, only `struts` officially uses SonarQube, suggesting that the developer may have been influenced by the tool's reports to identify and remedy the problem. Nevertheless, such cases are uncommon, and we believe that more attention should be directed toward cognitive complexity, given that methods with high cognitive complexity are inherently more difficult to test and maintain.

Among the rules that were exclusively associated with fixed issues, they primarily addressed obsolete code (S1874), improper exception handling (S112), and redundancy (S2293). Rule S1874 specifically targets the use of deprecated code marked with the `@Deprecated` Java annotation. In such cases, the removal of deprecated code is typically part of a planned refactoring process.

Rule S2293 emphasizes that the object type should not be redundantly declared in both the declaration and the constructor. Instead, the constructor should use the diamond operator (<>), allowing the compiler to infer the type automatically. Figure 8 provides examples of code snippets that comply with this rule and those that do not. The two PRs with the highest number of fixes for this rule were `accumulo` #2643 (107 instances) and #3604 (55 instances). In both cases, the fixes were achieved through code removals rather than direct refactoring.

Rule S112 advises against throwing generic Java exceptions, recommending that only specific exceptions intended to be handled should be caught. Relying on generic exception types forces developers to distinguish between exceptions by examining their messages, a practice that is prone to error and difficult to maintain. The PR with the most issues resolved for this rule was `accumulo` #3402, which addressed 260 instances. According to the PR description, the objective was to improve exception handling, as the author stated: "[...] try to catch more specific checked exceptions, when appropriate, instead of catching (Exception) [...]." However, this targeted approach to exception handling appears to be relatively uncommon.

> ***RQ2**: Naming conventions, literal duplication, cognitive complexity, obsolete code, improper exception handling, and redundancy are the most common TD issues. Developers tend to tolerate naming and duplication problems, and they actively address obsolete code, duplication, and exception handling; cognitive complexity, however, remains largely overlooked.*

RQ3: How does TD within PRs evolve over time?

While in RQ1 we examined TD evolution from a broad perspective, here we aim to investigate whether time influences TD variation. Figure 9 shows how TDV varies over

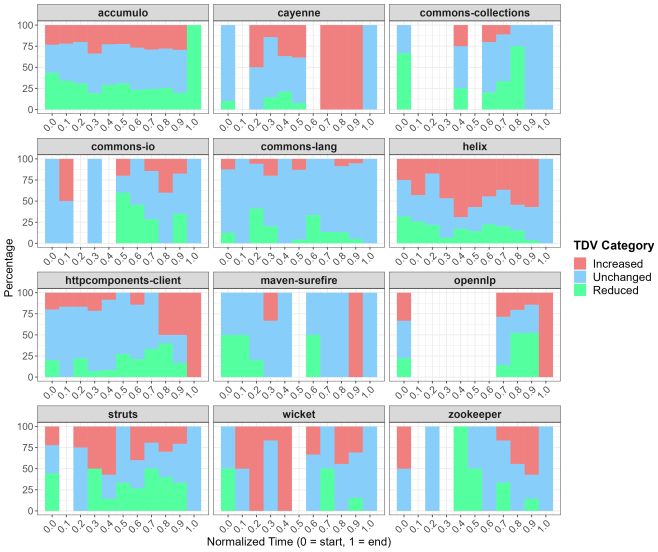

Figure 10. TDV categories evolution over normalized time by repository.

normalized time for each repository. We found no notable patterns in TD variation over the lifespan of the analyzed projects. Therefore, time alone does not explain changes in TD. This suggests that effective TD management must account for factors beyond the project timeline.

However, some projects display temporal gaps, such as `opennlp` and `commons-collections`. In the case of `opennlp`, this gap corresponds to a period of approximately four years and seven months. This was likely due to PRs from that period not executing the build process correctly, and consequently, we were not able to include them in our analysis.

When analyzing how TDV categories evolve (Figure 10), we observe that most repositories show a stable trend. However, three projects stand out: `cayenne` and `httpcomponents-clients` exhibited an increasing trend in PRs that contributed to TD towards the end of the analyzed period, while `wicket` showed a similar trend but closer to the beginning of the analyzed period. In all three projects that exhibited spikes, these increases might be attributed to higher-than-usual demands for new features. For instance, in the `wicket` project, we found an earlier spike, even though this project had already existed for over five years. Therefore, the observed spike is unlikely to be related to initial development efforts.

We also analyzed the evolution of issue severity over time (Figures 11 and 12). The first key finding is the disparity between *fixed* and *neglected* issues. Neglected issues exhibit a higher share of *CRITICAL* and *BLOCKER* cases, accounting for up to 50% of all recorded issues in some projects. This suggests that neglected TD consists primarily of severe issues that demand greater effort and time to resolve. Consequently, without proper, continuous management, TD can increase rapidly, undermining project quality and complicating future maintenance.

While neglected issues maintain a stable trend, fixed issues occasionally show spikes in high-severity cases across repositories. Periodically, developers actively target the most severe TD issues—likely those long neglected—demonstrating a degree of ongoing TD management.

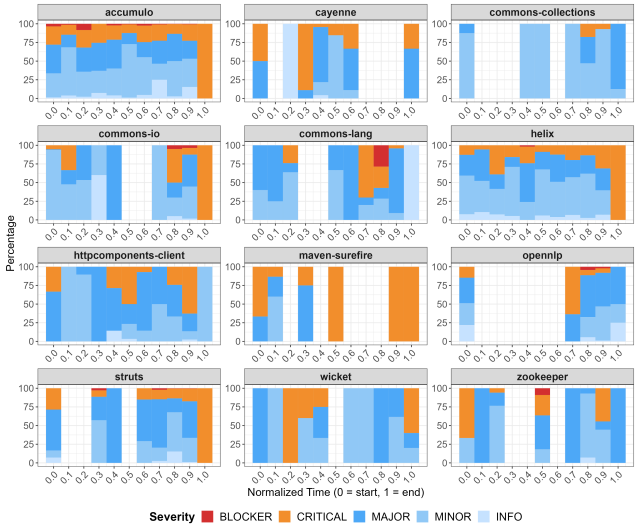

Figure 11. Fixed issues severity evolution over normalized time by repository.

Figure 12. Neglected issues severity evolution over normalized time by repository.

***RQ3**: TDV shows no consistent evolutionary pattern over time, while neglected issues within PRs consistently exhibit higher severity levels. We observed that developers periodically intensify efforts to resolve these issues during specific periods.*

3.2 Qualitative Study - The Study on PR Code Review Threads

We conducted a qualitative study of PR code review threads to assess the extent to which authors and reviewers identify, discuss, and address TD after a PR is opened. To guide this investigation, we defined the following research questions:

RQ4: Which maintenance goals are commonly linked to PRs that discuss TD? Considering common maintenance goals (bug fixing, requirement changes, and code improvement), we want to understand which kind of tasks lead to code review messages related to TD issues.

RQ5: What are the key characteristics of suggestions for addressing TD issues? We analyze how reviewers recommend resolving TD, focusing on who makes the suggestion, the type of suggestion, the level of agreement among participants, and the resolution.

RQ6: What are the primary reasons that suggestions for resolving TD issues are rejected? Assuming that some of the rejected suggestions may be related to neglected TD, we aim to identify the characteristics of these rejected suggestions and the most common reasons for their rejection.

3.2.1 Study Procedure

Inspired by previous qualitative work on code reviews (Han et al., 2022; Karmakar et al., 2022; Coelho et al., 2024), we conducted an analysis in which, for each PR review thread, the first author manually analyzed both the discussion and the associated code and answered a series of questions. The collected results were then validated by the remaining authors.

To select the sample, we first identified all PRs among the 2,035 analyzed in our quantitative study (Section 3.1) that contained at least one review thread, totaling 712 PRs. From these, we calculated the final sample using proportional stratified sampling by project, with a 95% confidence level and a 5% margin of error, resulting in a sample of 250 PRs. We adopted proportional sampling to ensure a representative sample that included projects as evenly as possible. Nevertheless, `cayenne` and `maven-surefire` projects were excluded from the sample calculation due to their minimal datasets, each containing only two PRs with review threads. This exclusion was based on their insufficient sample size rather than an arbitrary decision.

For each PR, the first author manually evaluated aspects of the PR and its associated review threads, addressing inquiries concerning both the PR content and the discussions within the review threads—each PR may have zero or more review threads, which typically begin when a reviewer selects a code snippet and initiates a discussion about it, potentially including one or more comments. We excluded standalone PR comments—those posted independently and not tied to any code snippet—because they obscure the boundaries of each discussion and provide no context about the affected code. That context is essential for us to determine whether reviewers' suggested changes were actually implemented.

We aimed to identify the main maintenance activity performed for each PR, referred to as the *primary objective*. This objective was classified based on the code change categories proposed by previous studies (Hassan, 2009; Palomba et al., 2017; Nikolaidis et al., 2023) and can be defined as: (i) *bug* - when the main activity of the PR consists of correcting a failure resulting from unexpected behavior; (ii) *improvement* - encompassing all actions aimed at improving code quality; and (iii) *requirement changes* - when a new feature is added or an existing one has its structure modified. In cases where the primary objective among these three categories was not clearly identifiable, we classified it as *unknown*. To identify each PR's primary goal, we manually reviewed its title, description, and any related issues.

Thus, in our study, we collected the following data:

Repository: The repository containing the PR.
PR number: The unique identifier of the PR.
PR URL: The link to access the PR details.
Primary objective: As described above and based on the categories from previous studies (Hassan, 2009; Palomba et al., 2017; Nikolaidis et al., 2023), classified as:

- ***BUG*:** When the primary objective is to fix bugs, that is, to correct unexpected behavior in system functionality. For example, `accumulo` #3150 was classified as a *BUG* because it implements logic to handle failures in a specific component, as indicated by the title "Modified ScanServer to correctly handle tablet failures" and clarified in the description: "The ScanServer was throwing a NotServingTabletException […]."
- ***IMP*:** When the primary objective is to improve the code quality, that is, to implement enhancements without changing the code's behavior. For instance, `accumulo` #2186 was classified as *IMP*, as its title and description indicate an objective to refactor the code to reduce duplication and address minor style issues.
- ***REQ*:** When the primary objective is to modify or implement features. For example, `helix` #2107 was classified as *REQ*, as its title indicates that it adds support for TTL and Container modes to `BaseDataAccessor` and its implementations.
- ***UNKNOWN*:** A category for PRs whose classification could not be adequately determined. For instance, `accumulo` #1934 was classified as *UNKNOWN* because there is insufficient information to determine whether it represents a bug fix or a requirement change. The title, "stop recovery if tablet is being deleted," could be interpreted either way; classifying it as a bug would require a report of an erroneous recovery occurring under those conditions.

Furthermore, we identified PRs and review threads initiated by bots. In both cases, we manually examined the user's

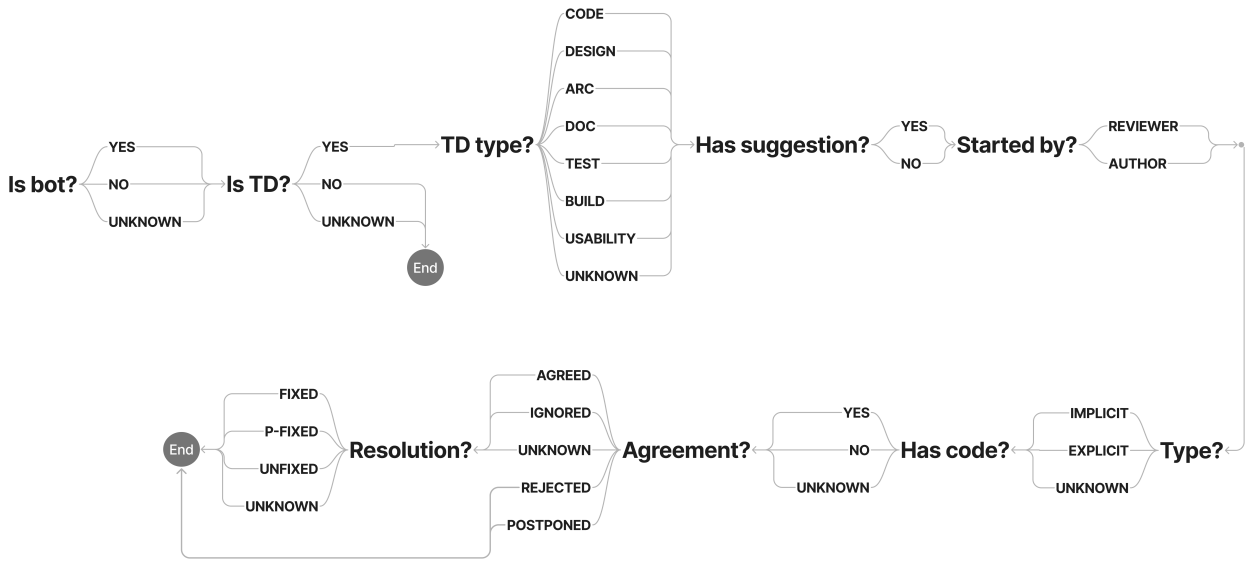

Figure 13. Review threads coding procedure.

profile and their comments to determine whether they belong to a bot. We also relied on the "bot" tag that bots can use on GitHub for identification.

For each review thread, we collected a set of data aimed at identifying whether the discussion addresses TD and, if so, determining its type. To perform this classification, we analyzed both the textual content and the associated code in the review thread. Figure 13 illustrates the coding procedure that we followed throughout this process. We adopt the TD categories defined in prior taxonomies (Kruchten et al., 2012; Alves et al., 2014; Li et al., 2015; Rios et al., 2018) and, for classification purposes, assume that TD refers exclusively to code quality issues that do not arise from requirement changes or bug fixes. Table 6 presents all types of TD identified during the manual evaluation, along with their definitions and representative examples.

It is important to note what we did not classify as TD. We do not classify bugs as TD, as Li et al. (2015) identified inconsistencies in the literature regarding whether defects should be considered a form of TD. Similarly, we exclude requirement changes, since they typically represent business needs that inherently carry priority due to their nature.

Additionally, inspired by a previous study (Han et al., 2022), we sought to identify the presence of fix suggestions—only for threads discussing TD. We classified the suggestions according to their type, author, whether they included suggested code snippets, whether there was agreement between the reviewer and the PR author, and whether the suggestion was implemented in the PR.

Regarding their type, the suggestions were classified as follows:

- **Explicit** (example in Figure 14). When the author of the suggestion explicitly points out a problem in the code in an assertive manner, often also indicating a possible solution.
- **Implicit** (example in Figure 15). When the author is unsure whether the identified issue is indeed valid; they hesitate to request changes explicitly.

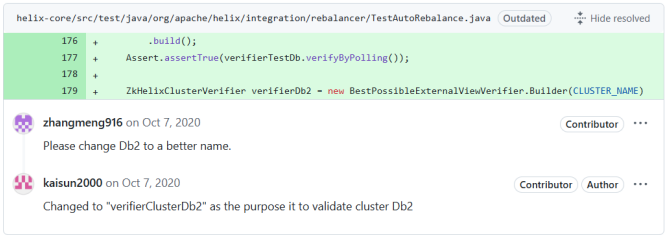

Figure 14. Example of a assertive suggestion in `helix` #1449.

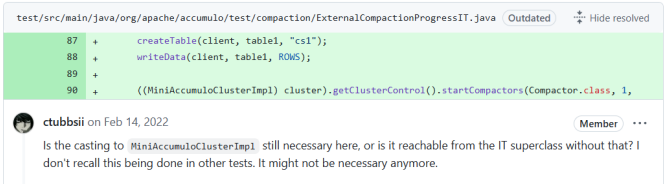

Figure 15. Example of a implicit suggestion in `accumulo` #2490.

We classified the suggestions according to the author (reviewer or PR author) and whether they included suggested code snippets (yes or no). Based on the procedure of a previous study (Han et al., 2022), we also identified whether there was agreement between the author and the reviewer on the application of the suggestion. This attribute is relevant because we want to verify whether the authors agreed with the issues raised by reviewers. Thus, the agreement was assessed solely based on the textual content of the review thread and categorized as follows:

***AGREED*:** When there is explicit agreement in a comment made by the PR author (if the suggestion was initiated by a reviewer) or by the reviewer (if the suggestion was initiated by the PR author). For example, the first review thread (ID *PRRT_kwDOAAJ058431ruQ*) of `zookeeper` #2114 was classified as *AGREED* since the reviewer noted that the author had forgotten to log the server's current state ("it seem that you are not printing the state"), and the author agreed and replied, "Yeah! Good catch, thanks!" In another example, in the first

Table 6. TD types, their definitions, and representative examples.

TD type	Definition	Examples
Code	Source-code deficiencies that hinder readability and increase maintenance effort.	"This import is not used" (`helix` #1847) "nit: looks like there is an extra space at the front" (`helix` #2588) "I'm wondering if update() could be renamed [...]" (`accumulo` #2224)
Design	Suboptimal designs in terms of efficiency, maintainability, and readability, as well as practices that diverge from essential object-oriented design principles.	"I think it might be helpful to break this function down to [...]" (`helix` #1678) "Maybe its better to just make the new SPI interfaces follow the patterns of the old ones [...]" (`accumulo` #1891)
Architectural	System-level issues that affect architectural requirements (e.g., performance, robustness, scalability).	"For less confusion, this should be in a package corresponding to the module it is contained in [...]" (`accumulo` #2549) "[...] I would like to move more logging to that package [...]" (`accumulo` #3047)
Test	Inadequate testing practices that reduce test effectiveness and compromise quality assurance.	"Can you test enable it by the new config?" (`helix` #1487) "Could use `assertThrows()` to ensure the specific line throws the desired exception [...]" (`accumulo` #2224) "A unit test for this would nice. [...]" (`accumulo` #2752)
Documentation	Absence, inadequacy, or incompleteness of project documentation artifacts.	"Let's add TODO for this and below methods." (`helix` #2127) "[...] wie should add a comment referencing JIRA-1337 [...]" (`commons-lang` #269) "[...] Maybe we should mention it in the documentation [...]" (`zookeeper` #1799)
Build	Build-system/process issues that lead to excessive complexity and hinder the ease of building the software.	"[...] Remove unused spring-webmvc-portlet [...]" (`struts` #552)
Usability	Poor usability decisions that are likely to require rework in the future.	"If we make this property of type PropertyType.BYTES, then it may be more user friendly [...]" (`accumulo` #1706)

review thread (ID *PRRT_kwDOAAMmKs42qKw0*) of `commons-lang` #1148, the author agreed to the suggestion by reacting with a thumbs-up emoji, which we interpret as agreement, and the fix was implemented in the subsequent commit.

***REJECTED*:** When the author explicitly rejects the suggestion. For instance, in the only review thread of `helix` #2597, the code under review prints the method name as a plain string; the reviewer suggested using a utility class to get the method name and print it, but the author disagreed, arguing that "[...] we just use the plan text test func name to reduce string append."

***UNKNOWN*:** When the author participates in the discussion without clearly indicating whether they agree or not. For example, in the second review thread (ID *PRRT_kwDOACaFSM4x2rKu*) of `accumulo` #3725, the reviewer suggested removing the "GC" prefix to eliminate redundancy in the log message. The author replied by asking whether a debug-level log might be more appropriate, without explicitly agreeing or disagreeing, so we classify the agreement as *UNKNOWN*.

***POSTPONED*:** When both agree that there is an issue, but it is not within the scope of the current PR. For example, in the third review thread (ID *PRRT_kwDOAPIHxc4wopCV*) of helix #2579, the reviewer suggested deferring style and refactoring changes to a separate PR to keep the focus on the core logic, which the author agreed with.

***IGNORED*:** When the author does not engage in the discussion, which is easily verified by the absence of any comments from them in the review thread.

Although the agreement indicates that a consensus was reached on the issue, it is important to verify whether it was effectively resolved in cases where the author neither rejected nor postponed the solution. Thus, we identified the resolution of the suggestion by analyzing only the code in the subsequent commits following the discussion. The resolution was categorized as follows:

***FIXED*.** When the issue was completely corrected. For ex-

ample, in the only review thread of `accumulo` #2403, the reviewer asked whether certain class attributes should be declared final. The author replied that they had made the change, and we verified in commit *149e858* that all affected attributes had been updated.

P-FIXED. When the issue was partially resolved. This classification encompasses situations where the solution can be divided and implemented in parts, with the developer leaving some parts unaddressed. For example, in the penultimate review thread (ID *PRRT_kwDOAPIHxc4xSLfM*) of `helix` #2588, the reviewer suggested replacing a large code block in a test class with a single log statement because the block would not be used. The author removed all the code from the method but did not add the log, thereby addressing only part of the suggestion.

UNFIXED. When the issue was not resolved. For example, in the penultimate review thread (ID *PRRT_kwDOAAJ0584eND-O*) of `zookeeper` #1799, the reviewer asked that the log level be changed from WARN to INFO. The author explicitly agreed, stating they had made the change to INFO, but no subsequent commit was made, and the issue was never resolved. Conversely, in the only review thread of `helix` #2666, the reviewer suggested moving a method to an abstract test class for reuse. The author did not engage further, and no commits followed, leaving the suggestion unaddressed.

UNKNOWN. When it is not possible to determine whether the issue was corrected. This category applies when it's impossible to determine with confidence that the issue was fixed. It covers cases where it is unclear what the correction should entail, or where commits containing the potential fix include too many changes, making it impossible to identify the correction with precision. For example, in the third-to-last review thread (ID *PRRT_kwDOAPMNSs439FfM*) of strus #861, the reviewer recommended improving a code segment's performance without specifying how. The author subsequently comments that the issue was fixed, but the subsequent commit bundles numerous edits, making it unclear whether the performance issue was actually addressed.

These data collection and evaluation processes generated two datasets: one for PRs and another for review threads. The PR dataset consists of 250 elements, while the review thread dataset contains 929 records. Both datasets include the features mentioned above.

3.2.2 Results and Discussion

In this section, we present the results and then discuss the findings for the research questions Q4-Q6. All results in this section are derived from manual analysis.

Data Characterization

In this study, we manually evaluated a sample of 250 PRs. In 211 of them (84.4%), we identified at least one discussion related to TD, meaning that four out of five PRs include some discussion about TD issues. When analyzing this value for each repository (Figure 16), most of them show percentages above 80%, except for the `opennlp` and `wicket` projects, both of which have a low number of PRs (2 and 5, respectively). This result indicates that PR reviewers are aware of TD-related issues.

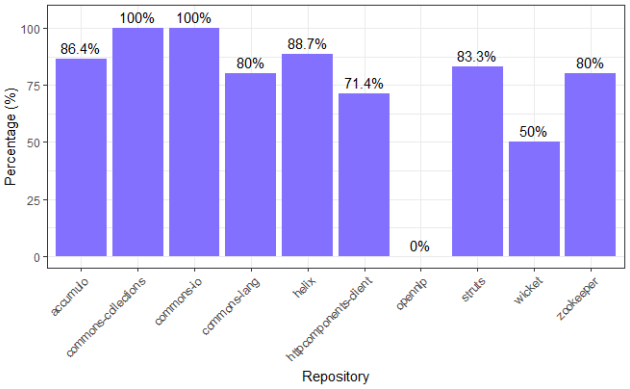

Figure 16. Percentage of PRs with TD discussion by project.

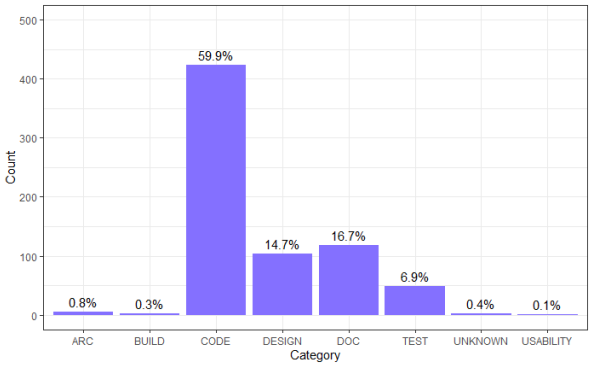

Figure 17. Review threads by TD type.

In addition, we analyzed 879 unique review threads. However, since some threads discussed more than one issue, the total number of review threads per issue, including duplicates, was 929, of which 708 discussed TD. The average number of review threads per PR was 3.72, while the median was 2. The average number of review threads discussing TD per PR was 3.35, with a median also of 2.

When comparing these values with the statistics on TD issues per PR from the quantitative study (mean = 126.3, median = 26), we can conclude that although reviewers almost always point out TD issues, the number of discussions remains relatively low compared to the total number of issues present in the code, whether pre-existing or newly introduced.

Table 7. Review threads regarding a TD discussion.

TD Discussion	Percentage (%)
No	20.5
Unknown	3.6
Yes	76.0

Table 7 shows that three out of four review threads discuss TD, while only one in five does not. These numbers reinforce prior findings, indicating that reviewers frequently identify and engage with TD-related issues during code reviews. Furthermore, we identified the most discussed types of TD (Figure 17). *Code* is the most frequently debated type,

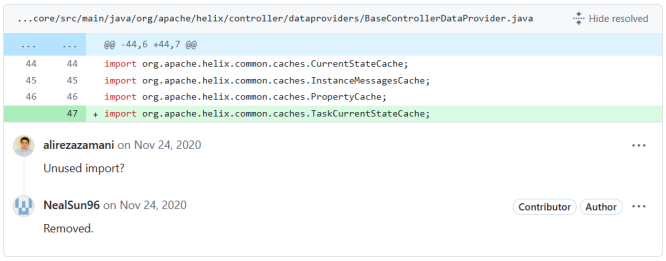

Figure 18. Review thread about TD code.

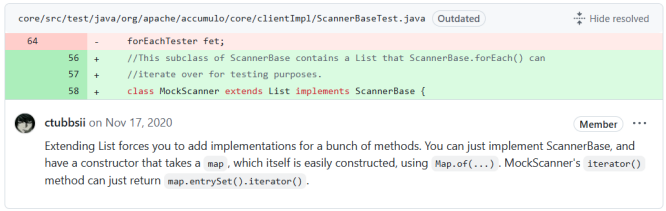

Figure 19. Review thread about TD design.

followed by *documentation*, *design*, and *testing*. The fact that code is the most common type is an expected result, as these issues are generally easier to identify and resolve, such as improper naming, style, and readability issues, among others. However, other types of TD issues are also considered by reviewers.

Figure 18 presents a typical discussion involving code that occurred in PR `helix` #1550, where an unused import issue was identified and subsequently fixed. On the other hand, Figure 19 illustrates an example of a design issue identified in `accumulo` #1765, where the reviewer proposed a better solution for the `MockScanner` class, and the author partially implemented the solution (*P-FIXED*).

Among the 708 review threads that discuss TD, 680 of them (96.32%) include suggestions. This indicates that reviewers not only identify issues but also actively propose concrete solutions to address them in the code.

Finally, we checked whether any PR or review thread was initiated by a bot. We did not find any PRs created by bots, but we identified four review threads initiated by bots: two in `struts` #861, one in `zookeeper` #1992, and one in `zookeeper` #2001. In `struts`, both review threads were initiated by a GitHub Advanced Security[18] bot, which ran SonarCloud through the code scanning feature. In `zookeeper`, both review threads were initiated by a bot from the Sonatype Lift[19] tool. The only discussion with human involvement was in `zookeeper` #2001, where the PR author pointed out that the reported issue was a false negative. All identified bots used the "bot" tag for self-identification.

RQ4: Which maintenance goals are commonly linked to PRs that discuss TD?

Table 8. Percentage of PRs by primary goal.

Primary Goal	Percentage (%)
BUG	26
IMP	40
REQ	29.6
UNKNOWN	4.4

[18] `https://docs.github.com/en/get-started/learning-about-github/about-github-advanced-security`

[19] `https://github.com/apps/sonatype-lift`

Before addressing RQ4, it is important to understand the most common primary objectives in our sample. Table 8 presents the percentage of PRs for each category.

The *IMP* category is the most frequent, accounting for 40% of PRs. This indicates that two out of five PRs focus on implementing code improvements and, consequently, resolving TD issues. However, when comparing these results with the quantitative study, which showed that 25% of PRs reduce TD, we observe a 15% difference between PRs that effectively reduce TD and those whose primary objective is code improvement.

This difference is attributable to issue types that SonarQube does not detect, including incomplete documentation, spelling errors in both documentation and code, more complex problems related to the code context, and missing test cases. Such issues are only detected during the PR review phase, highlighting the importance of code reviews in this context and the limitations of code analysis tools.

The *REQ* and *BUG* categories follow, with 29.6% and 26%, respectively. Finally, in 11 cases (4.4%), we were unable to identify the primary objective.

Table 9. Percentage of PRs with TD discussion by primary goal.

Primary Goal	Percentage (%)
BUG	76.9
IMP	86
REQ	87.8
UNKNOWN	90.9

Answering RQ4, Table 9 shows the percentage of PRs that discuss TD for each primary objective. PRs focused on adding or modifying features are the ones that most frequently include TD discussions, followed closely by PRs that improve the code.

Since new code is introduced more frequently in PRs with requirement changes, the results indicate that, in these situations, reviewers are more likely to raise TD-related issues (around 88%). On the other hand, bug-fix PRs present the lowest rate of TD discussions. Although still relatively high (77%), this slightly lower rate may be explained by the fact that reviewers tend to focus more on ensuring that the unexpected behavior caused by the bug is properly fixed rather than on code quality aspects.

> ***RQ4****: PRs that add or modify features involve TD discussions most often (87.8%), with code-improvement PRs following closely behind. Bug-fix PRs exhibit the lowest rate of such discussions (76.9%), indicating that in these cases, reviewers give slightly more priority to fixing functionality over debating code quality.*

RQ5: What are the key characteristics of suggestions for addressing TD issues?

As pointed out in the data characterization, more than 96% of the review threads discussing TD also include a fix suggestion. These suggestions can provide insights into whether, based on the TD issues raised by the reviewer, the author understands the need to pay the debt and whether they actually correct them.

Table 10. Distribution of suggestions across different categories.

Category	Distribution (%)
Author	
Reviewer	96.9
PR Author	3.1
Type	
Explicit	50.6
Implicit	49.4
With suggested code	
No	63.7
Yes	36.3
Agreement	
AGREED	45.0
IGNORED	32.9
POSTPONED	3.7
REJECTED	12.1
UNKNOWN	6.3
Resolution	
FIXED	83.6
P-FIXED	3.0
UNFIXED	11.2
UNKNOWN	2.3

Figure 20. Example of a suggestion with code suggestion (`accumulo` #2490).

Before that, it is important to understand what a typical fix suggestion looks like. Table 10 presents the distribution of suggestions by author role among other breakdowns. In 96.9% of cases, suggestions come from reviewers, while only 3.7% originate from the PR authors. This suggests that PR authors often resolve issues directly in the code without the need for an explicit discussion.

However, PR authors are more likely to initiate a discussion when they are uncertain about their implementation, unsure if a code issue exists, or are debating whether it should be addressed. In these situations, they seek a second opinion to help them decide how to proceed. For example, in `struts` #867, the author starts a discussion about the possibility of renaming some classes and interfaces in the future: "Maybe in 7.0 we can consider renaming `VelocityManager` to `StrutsVelocityManager` [...]."

Regarding *type*, both explicit and implicit suggestions appear with almost equal frequency. However, suggestions that include code snippets account for about one-third of the total,

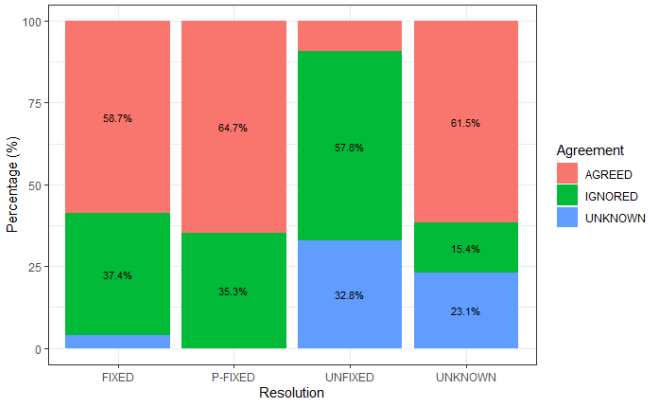

Figure 21. Suggestions by agreement and resolution.

meaning that most suggestions do not contain code. Figure 20 illustrates an example in which the reviewer includes a code snippet with suggestions to improve readability, making it easier for the developer to understand what needs to be modified.

The agreement analysis shows that in 45% of the suggestions, there is explicit consensus between the author and the reviewer. In 32.9% of the cases, the author simply ignores the discussion, while in 6.3% they participate but do not clearly state whether they agree with the suggestion. It is worth noting that in 3.7% of cases, even when consensus is reached, the correction is postponed.

Rejected suggestions account for 12.1% of cases, in which the author explicitly states in the discussion that they do not accept the suggestion. Excluding the rejected and postponed suggestions, approximately 84% of the discussed issues could potentially be fixed or not. Nevertheless, there is still room for developers to engage more actively in discussions, making it clear whether they agree with or reject the proposed suggestions.

While agreement indicates whether the author acknowledges the identified issue, it is still necessary to verify whether they actually fix the problem in the code. Table 10 also presents the distribution of suggestions by resolution. Four out of five issues are fixed, 3% of the corrections are only partially implemented, and 11.2% of the issues remain unresolved.

Complementing these findings with the results shown in Figure 21, we observe that in approximately 40% of the cases where problems were fixed, there was no explicit agreement between the author and the reviewer, yet the author still implemented the correction. In cases where the discussion was ignored, a possible explanation is that when the issue is clear, the author may simply choose to fix it directly. However, even in such cases, it would be beneficial for them to explicitly state their agreement and confirm that the issue has been fixed, as this could also serve as additional motivation for reviewers.

One aspect that requires more attention from PR authors is that in about 90% of cases where suggestions remained unfixed, reviewers were not explicitly informed of this decision. This lack of explicitness typically occurred either because the PR author did not participate in the discussion (*IGNORED*) or did not clearly state their intention to leave the suggestion unfixed (*UNKNOWN*). Consequently, reviewers were likely

unaware that their suggestions would not be addressed.

> ***RQ5**: Almost all fix suggestions (96.9%) come from reviewers, yet most—63.7% overall and 76.5% of implicit ones—lack suggested code snippets, which can obscure what needs fixing. In approximately 40% of cases, developers neither participate nor clearly express an intent to resolve the issue; for unaddressed issues, that figure climbs to nearly 90%.*

Table 11. Distribution of rejected suggestions by type and suggested code.

Category	Distribution (%)
Type	
Explicit	15.9
Implicit	84.1
With suggested code	
No	82.9
Yes	17.1

RQ6: What are the primary reasons that suggestions for resolving TD issues are rejected?

As previously discussed, the number of rejected suggestions in our sample is low (12.1%). To better understand this group, we first analyzed the distribution of rejected suggestions by type (Table 11). We found a significant increase of more than 50% in the proportion of implicit suggestions, which account for 84.1% of the total. There is also an increase in the proportion of suggestions without a suggested code snippet. We examined various cases of rejected and implicit suggestions, which revealed that, in most situations, reviewers perceived an issue. However, developers often explain why the identified point is not actually a problem.

For example, in `helix` #2153, a discussion begins with the reviewer asking: "How about a proper successful test case?", to which the author responds: "The first test case covered that." In this case, the reviewer initially believed that there were no tests covering the success case, but the developer clarified that such a test already existed.

In some cases, the issue raised by the reviewer is valid, but the suggestion is rejected because the developer is simply following project standards or because implementing the change would require significant modifications, making it inappropriate at that moment. For instance, in `zookeeper` #1799, the reviewer suggests using a final attribute instead of repeatedly parsing the same value multiple times. The author responds that, despite agreeing, this is the standard practice in the project. In another example, in `accumulo` #1945, the reviewer points out that the code does not follow a consistent naming convention for an object, sometimes referring to it as "ctx" and other times as "context," and suggests adopting a single naming pattern to improve readability. The author acknowledges the issue but states that the change falls outside the scope of the PR.

Figure 22 shows that the main types of TD involved in rejected suggestions are *code* (61%) and *design* (24.4%). When comparing these values with those obtained for all review threads discussing TD (Figure 17), we observe an increase in the rejection of design-related TD and a significant decrease in the rejection of documentation-related TD.

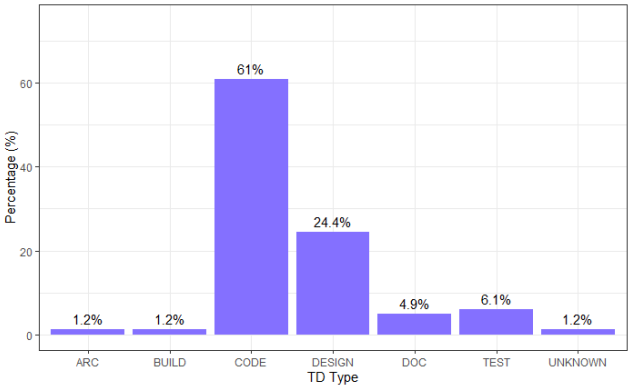

Figure 22. Rejected suggestions by TD type.

The lower rejection rate for documentation-related suggestions may be linked to the fact that such suggestions tend to be simple, straightforward, and easy to implement. In general, they involve minor modifications to existing documentation, the addition of new documentation, or spelling corrections, and reviewers often provide explicit textual suggestions. For example, in `commons-io` #530, there are two documentation suggestions, both including specific textual suggestions for inclusion: (i) "[...] You could say instead 'Always throws the exception supplied from a constructor'" and (ii) "[...] I think we should say 'in a constructor', not 'in the...'" On the other hand, in `accumulo` #2369, there is a typo correction suggestion that simply involves changing "witn" to "with."

Upon analyzing the discussion content of rejected suggestions, we observed that nearly all rejections resulted from situations where reviewers identified a problem without fully examining the code, leading developers to reject the suggestion by explaining why it was not an issue. In other cases, although the problem may be real, the suggestion was rejected because it conflicted with project standards or the required changes would entail considerable effort.

> ***RQ6**: Rejected suggestions differ significantly from common ones, with 84.1% being implicit and 82.9% lacking suggested code, alongside fewer documentation-related issues. Most rejections occur when reviewers identify issues without thorough code examination, leading developers to dismiss them by clarifying non-issues, or when suggestions conflict with project standards or require substantial effort.*

3.3 Implications

Our findings carry important implications for practitioners, researchers, and tools. For practitioners, the results reveal that issues related to code quality—classified as *CODE SMELL*—are the most prevalent, often necessitating extensive remediation efforts due to their medium to high severity.

While the TD detected by SonarQube is neglected in a large portion of PRs (about 75%), we observed that a similar proportion (76%) of PRs discuss TD during code reviews. Furthermore, a significant portion of the issues identified in these reviews is later addressed by developers (83.6%). This divergence between the two scenarios can be explained by several factors.

One factor is the adoption of ASATs. Only the `struts`

project officially integrates SonarQube, making it unlikely that developers of other projects run the tool manually on their PRs, meaning that they may not be aware of all these issues. Even when a tool is integrated, developers might not proactively use it to fix every issue. ASATs flag every detected problem within their scope, often leading developers to prioritize only the most relevant or critical issues. This behavior is consistent with prior studies (Marcilio et al., 2019; Tan et al., 2021) showing that developers tend to focus on a small subset of issue types that the tool can detect.

Finally, the limitations of ASATs play a role. An ASAT often cannot detect more complex issues or issues in other domains, such as documentation, which can further explain the divergence between tool detection and review discussions.

As practical implications, tool developers can conduct studies to expand the detection scope of ASATs, using issues identified in code reviews as a basis and potentially leveraging our dataset from the manual study. For developers and reviewers, the results indicate that while there is a good level of discussion about TD during reviews, there is still room for improvement, as reviewers cannot detect all issues on their own. Therefore, it is advisable to complement code reviews with static analysis tools.

In code reviews, we identified that in approximately 40% of the discussions, developers either do not participate (*IGNORED*) or do not clearly express their intention to fix the issue (*UNKNOWN*). This percentage rises to around 90% for issues that remain unresolved (*UNFIXED*). Given these findings, it is crucial for developers to engage more actively in discussions, clearly stating their intentions regarding whether or not to fix the issues. Meanwhile, reviewers can adopt a more direct approach by clearly pointing out issues and, whenever possible, providing code snippets with suggested solutions.

Project managers can explore the implementation of mechanisms to prevent the merging of PRs that introduce new TD beyond a predefined threshold. Additionally, for projects utilizing configurable tools like SonarQube, they could tailor the active rules set to their specific context, disabling irrelevant rules.

For researchers, further investigations could explore the tolerance for issues by gathering developers' perspectives. Such studies could contribute to the proper configuration of static analysis tools and/or the development of new tools that consider the context in which issues occur and/or adjust severity levels. Additionally, with the growing popularity of Large Language Models (LLMs), automatic code review tools like CodeRabbit[20] have emerged, offering the capability to address developers' questions during discussions. Therefore, it would be interesting to compare these tools with ASATs in terms of both detection scope and the rate at which developers address identified issues. It would also be relevant to compare human reviews with automated reviews across various dimensions, such as effectiveness, accuracy, and developer acceptance.

4 Threats to Validity

In the quantitative study using SonarQube, we are restricted to issues related to code and design (Nikolaidis et al., 2023). This limitation was mitigated through manual analysis, which enabled us to identify a variety of TD types, covering domains beyond the tool's scope. In addition, SonarQube can also present false positives. The first author conducted a manual review of several samples from the collected data and observed that the occurrence of false positives was minimal, thus not significantly impacting the conclusions drawn.

A major challenge was the necessity of compiling the code. This requirement led to the exclusion of many projects and PRs from our study due to compilation errors. Furthermore, the analysis was a time-intensive process, with an average execution time of 6 minutes and 18 seconds per commit. To address this, we parallelized the executions across four VMs, enabling the analysis of a substantial sample of 2,035 PRs within approximately two weeks.

Despite these challenges, SonarQube remains the most widely used tool in empirical studies related to TD (Digkas et al., 2017; Molnar and Motogna, 2020; Avgeriou et al., 2021; Zabardast et al., 2022; Nikolaidis et al., 2023; Dantas et al., 2023), and to the best of our knowledge, the only tool capable of estimating TD remediation time.

Alternative quality methods could also be employed to assess TD. For instance, the SQALE method quantifies TD in terms of remediation time, while models such as QMOOD (Bansiya and Davis, 2002) and Quamoco (Wagner et al., 2012) consolidate structural metrics into quality aspects. Furthermore, Curtis et al. (2012) proposed a measurement approach that considers three variables: the number of issues to be addressed, the time required to resolve each issue, and the labor cost (dollars per hour). Among these methods, only Quamoco offers an available plugin[21], although it has not been updated in the last eight years. A recent study (Özçevik, 2024) even utilized tools like SonarQube to estimate the metrics outlined by QMOOD. Thus, the SQALE method, implemented through SonarQube, proved to be the most viable approach for measuring TD in our study.

Our filtering strategy eliminated nearly half of the PRs in the initial dataset. However, this step was necessary to avoid scenarios such as handling code pulled from other branches, which could impact the results. Furthermore, we focused exclusively on Java projects from the Apache Software Foundation. Therefore, our findings may differ for projects in other programming languages or industrial contexts.

Regarding our qualitative study, its primary limitation lies in the fact that the manual analysis and classification were conducted only by the first author and revised by the remaining authors. To mitigate potential biases, we developed a detailed study guide containing all the questions we aimed to answer and guidelines for addressing them. Furthermore, the first author is an experienced developer (with more than 3 years of experience), and he chose to conduct short evaluation sessions (approximately 10 PRs per day) to minimize the impact of fatigue on the analyses. Although our sample size is limited, we calculated it to achieve a 95% confidence

[20] `https://www.coderabbit.ai/`

[21] `https://github.com/MSUSEL/msusel-quamoco-plugin`

level and a 5% margin of error.

We base our definition of TD on taxonomies established in prior studies (Kruchten et al., 2012; Alves et al., 2014; Li et al., 2015; Rios et al., 2018). The types of TD may vary across studies, but most of the categories presented here are commonly found in existing taxonomies. Each review thread was classified as discussing TD or not. For cases where the context was not sufficiently clear to determine whether TD was involved, we applied the *UNKNOWN* category. We encountered several instances where a reviewer simply wrote "same here" or "this too," referring to an issue previously reported in another review. Since we only considered the context of the thread under review, we classified all such cases as *UNKNOWN* to avoid misattributing references to prior discussions. Two examples of this are found in `commons-io` #325 and `helix` #1579.

A separate challenge was identifying cases where a comment's purpose was unclear, making it difficult to distinguish between a quality concern, a bug, or a requirement change. For example, in the third-to-last review thread of `helix` #1534, the reviewer asked, "How about the managers? Please disconnect them too." Neither the code nor the comment provided enough detail to judge whether this was a performance-related issue, a new requirement, or a bug; we therefore classified it as *UNKNOWN*. Aside from these ambiguous cases, most review threads concerned straightforward issues such as style corrections, documentation fixes, or variable renaming.

Other studies exploring changes in code reviews may employ different categorizations. For instance, Panichella and Zaugg (2020) identify only two types of maintenance activities: perfective maintenance, which encompasses code improvements, and corrective maintenance, which includes both requirement changes and bug fixes. We chose to distinguish three types, following previous similar studies (Hassan, 2009; Palomba et al., 2017; Nikolaidis et al., 2023). Additionally, they categorize change types at a finer granularity, where their perfective maintenance category includes types aligning with our definition of TD, albeit under different labels.

The scripts developed to process and analyze the data were manually validated by the authors through extensive testing. To ensure the replicability of this study, we have detailed all the main steps in the methodology and made our replication package available (Calixto et al., 2025). This package includes the datasets, scripts, and the data produced at each stage of the analysis.

5 Related Work

In this section, we review relevant literature related to our work. We begin by examining studies related to the evolution of TD (Section 5.1), outlining the level of granularity used to measure TD and the key findings derived from their analysis. Next, we explore research on code review practices within PR-based development (Section 5.2).

5.1 TD Evolution

Li et al. (2015) performed a systematic literature review focused on TD management, identifying 10 distinct types of TD, 8 TD management activities, and 29 tools used for TD management. Their findings revealed that *code debt* was the most extensively studied type of TD, aligning with the focus of our quantitative study. Similarly, Avgeriou et al. (2021) compared various tools for measuring TD, concluding that SonarQube is the most widely adopted tool, which is also employed in our quantitative analysis.

Several studies have explored TD evolution in software projects using ASATs, primarily SonarQube. Digkas et al. (2017) analyzed weekly revisions across 66 Apache Java projects over a five-year period, finding that absolute TD levels increased while normalized TD decreased. Their study identified prominent issues such as improper exception handling and code duplication. Molnar and Motogna (2020), examining three Java projects, found a strong correlation between lines of code and TD levels, highlighting that 20% of issue types accounted for 80% of TD. In contrast to their analysis, our study additionally identifies the types of issues most frequently fixed and neglected within PRs. Tan et al. (2021) investigated the evolution of TD in Python projects, noting that resolved TD was predominantly related to testing, documentation, complexity, and code duplication, with most TD resolved within two months.

While these studies provide valuable insights into long-term TD trends through project revisions, our study uniquely focuses on PRs as the unit of analysis, allowing us to examine small-scale development cycles, thereby capturing the socio-technical factors that influence TD management.

Nikolaidis et al. (2023) examined the impact of maintenance activities on TD growth, analyzing 13.5K PRs from Java projects using SonarQube. The results revealed that adding new features typically increases TD, whereas refactoring activities reduce it. Our work differs from Nikolaidis *et al.* by analyzing TD evolution within PRs specifically, identifying frequently resolved and neglected issue types, examining the temporal evolution of TD and issue severity, and exploring maintenance activities associated with TD discussions.

5.2 Code Review in PR-Based Development

Previous studies have already investigated code review in PRs, focusing on quality issues (Pascarella et al., 2019; Uchôa et al., 2020; Han et al., 2022; Coelho et al., 2024) and TD (Karmakar et al., 2022). Karmakar et al. (2022) analyzed whether PR comments indicate TD. Initially, they manually labeled comments and used this data to train machine learning and deep learning models. During this process, the authors faced several challenges, as a single PR comment often does not provide enough information about the discussed context. A TD issue may be spread across multiple comments or even discussed elsewhere, such as in issues. Considering this limitation, we chose to analyze code reviews at a higher level of granularity, using review threads. A review thread has a well-defined scope, clearly indicating where the discussion starts and ends, as well as the specific code segment

involved. This allows us to consider both the code and the discussion itself to determine whether a review thread addresses TD or not. While Karmakar et al. (2022) limit their analysis to classifying whether comments are related to TD or not, we seek to understand whether developers agree with reviewers' recommendations and whether they actually address the identified issues.

Coelho et al. (2024) conducted a qualitative study to characterize code review in refactoring-inducing PRs. One of their findings was that some review comments are more assertive, with reviewers directly point out the problem and, to some extent, demand a correction. In contrast, other comments reflect the reviewer's uncertainty about the identified issue. We took these aspects into account when classifying code review suggestions as explicit or implicit.

With the aim of investigating which code smells are identified and fixed during code reviews, Han et al. (2022) analyzed review comments in four open-source projects. Among other things, they classified the types of code smells, the actions suggested by reviewers, and how developers responded. The authors found that reviewers usually provide constructive feedback, including correction recommendations, and that developers tend to follow these suggestions. Extending this analysis, our study classifies reviewer-developer interactions by examining agreement and resolution of TD issues within review threads.

Mäntylä and Lassenius (2009) and Beller et al. (2014) observed approximately a 75:25 ratio between evolvability (TD-related) and functionality (bug-related) issues, which aligns closely with our findings regarding TD presence in PRs and review threads. Similarly, El Zanaty et al. (2018) classified review comments in two projects to determine which were related to design. They found that only 9% to 14% of reviews were design-related, which is consistent with our observations about design TD discussions.

Previous research on types of code changes (Hassan, 2009; Palomba et al., 2017; Nikolaidis et al., 2023) typically classified activities into requirement implementation, bug fixing, and quality improvement. Palomba et al. (2017) found that bug-fixing activities tend to include a higher number of refactoring operations aimed at improving maintainability and readability. On the other hand, during the implementation of new features, refactoring operations are usually more complex and focus on enhancing system design.

6 Conclusions

In this paper, we investigated how TD evolves within PRs by using a static analysis tool as a ground truth and examining TD as part of the code review process. To achieve this, we evaluated merged PRs from 12 Java projects from the Apache ecosystem.

The results show that TD issues are present in nearly all PRs (96.26%). Across the projects, the variation in TD follows a ratio close to 1:2:1 (reduction:unchanged:increment), meaning that for every four PRs, one reduces TD, two leave TD unchanged, and one increases TD. Time does not appear to influence TD variation, while neglected issues consistently exhibit high severity levels over time.

Our analysis revealed that the most frequent issues across the studied projects are associated with naming conventions, literal duplication, cognitive complexity, obsolete code, improper exception handling, poor logging practices, and redundancy. In particular, literal duplication was the most prevalent among all projects. Our findings suggest that developers may tolerate certain issues—specifically those related to literal duplication and naming conventions—while cognitive complexity stands out as a significant concern that warrants further attention, given its impact on code comprehension and testability.

Regarding code reviews, approximately 76% of them include discussions about TD issues, with the most common issues related to code, design, and documentation. Nearly all discussions include a suggested fix (around 96%), and over 80% of these issues are resolved. However, in a significant portion of the reviews, developers do not actively participate, reaching 90% of cases where issues remain unaddressed.

For future work, we plan to expand our manual study to strengthen the reliability of the coding by involving additional raters, conducting consensus meetings, and performing inter-rater agreement analysis (e.g., using Cohen's Kappa). We also plan to dive into aspects such as understanding ignored suggestions during reviews. Moreover, future studies could assess developers' and reviewers' tolerance toward certain issues through surveys. An emerging area of interest is automatic code review, where research could compare different tools, from ASATs to automated review systems, and contrast human review with automated approaches.

7 Availability of Artifacts

All data and scripts utilized in this study are publicly available to support future research and reproducibility (Calixto et al., 2025).

Acknowledgements

The first author was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spínola, R. O. (2014). Towards an ontology of terms on technical debt. In *2014 Sixth International Workshop on Managing Technical Debt*, pages 1–7.

Ampatzoglou, A., Mittas, N., Tsintzira, A.-A., Ampatzoglou, A., Arvanitou, E.-M., Chatzigeorgiou, A., Avgeriou, P., and Angelis, L. (2020). Exploring the relation between technical debt principal and interest: An empirical approach. *Information and Software Technology*, 128:106391.

Anquetil, N., Delplanque, J., Ducasse, S., Zaitsev, O., Fuhrman, C., and Guéhéneuc, Y.-G. (2022). What do de-

velopers consider magic literals? A smalltalk perspective. *Information and Software Technology*, 149.

Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., de Toledo, S. S., and Tsintzira, A. A. (2021). An overview and comparison of technical debt measurement tools. *IEEE Software*, 38(3):61–71.

Baltes, S. and Ralph, P. (2022). Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering*, 27(4):94.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17.

Beller, M., Bacchelli, A., Zaidman, A., and Juergens, E. (2014). Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 202–211, New York, NY, USA. Association for Computing Machinery.

Besker, T., Ghanbari, H., Martini, A., and Bosch, J. (2020). The influence of technical debt on software developer morale. *Journal of Systems and Software*, 167:110586.

Calixto, F., Araújo, E., and Alves, E. (2024). How does technical debt evolve within pull requests? an empirical study with apache projects. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*, pages 212–223, Porto Alegre, RS, Brasil. SBC.

Calixto, F., Araújo, E., and Alves, E. (2025). [replication kit] how developers address technical debt in pull requests: A dual study. `https://doi.org/10.5281/zenodo.15054186`.

Coelho, F., Tsantalis, N., Massoni, T., and Alves, E. L. G. (2021). An empirical study on refactoring-inducing pull requests. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12.

Coelho, F., Tsantalis, N., Massoni, T., and Alves, E. L. G. (2024). A qualitative study on refactorings induced by code review. *Empirical Software Engineering*, 30(1):17.

Coq, T. and Rosen, J.-P. (2011). The sqale quality and analysis models for assessing the quality of ada source code. In Romanovsky, A. and Vardanega, T., editors, *Reliable Software Technologies - Ada-Europe 2011*, pages 61–74, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cunningham, W. (1992). The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, page 29–30, New York, NY, USA. Association for Computing Machinery.

Curtis, B., Sappidi, J., and Szynkarski, A. (2012). Estimating the principal of an application's technical debt. *IEEE Software*, 29(6):34–42.

Dantas, C. E. C., Rocha, A. M., and Maia, M. A. (2023). How do developers improve code readability? an empirical study of pull requests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 110–122.

Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., and Avgeriou, P. (2022). Can clean new code reduce technical debt density? *IEEE Transactions on Software Engineering*, 48(05):1705–1721.

Digkas, G., Lungu, M., Chatzigeorgiou, A., and Avgeriou, P. (2017). The evolution of technical debt in the apache ecosystem. In Lopes, A. and de Lemos, R., editors, *Software Architecture*, pages 51–66, Cham. Springer International Publishing.

Eisenberg, R. J. (2012). A threshold based approach to technical debt. 37(2):1–6.

El Zanaty, F., Hirao, T., McIntosh, S., Ihara, A., and Matsumoto, K. (2018). An empirical study of design discussions in code review. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA. Association for Computing Machinery.

Giordano, G., Annunziata, G., De Lucia, A., and Palomba, F. (2023). Understanding developer practices and code smells diffusion in ai-enabled software: A preliminary study. In De Vito, G., Ferrucci, F., and Gravino, C., editors, *Joint Proceedings of the 32nd International Workshop on Software Measurement (IWSM) and the 17th International Conference on Software Process and Product Measurement (MENSURA), Rome, Italy, September 14-15, 2023*, volume 3543 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Griffith, I. and Izurieta, C. (2014). Design pattern decay: the case for class grime. ESEM '14, New York, NY, USA. Association for Computing Machinery.

Han, X., Tahir, A., Liang, P., Counsell, S., Blincoe, K., Li, B., and Luo, Y. (2022). Code smells detection via modern code review: a study of the openstack and qt communities. *Empirical Software Engineering*, 27(6):127.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88.

Karmakar, S., Codabux, Z., and Vidoni, M. (2022). An experience report on technical debt in pull requests: Challenges and lessons learned. In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '22, page 295–300, New York, NY, USA. Association for Computing Machinery.

Kruchten, P., Nord, R., and Ozkaya, I. (2019). *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley Professional, 1st edition.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21.

Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610.

Lenarduzzi, V., Nikkola, V., Saarimäki, N., and Taibi, D. (2021). Does code quality affect pull request acceptance? an empirical study. *Journal of Systems and Software*, 171:110806.

Lenarduzzi, V., Saarimäki, N., and Taibi, D. (2019). The technical debt dataset. *CoRR*, abs/1908.00827.

Lenarduzzi, V., Saarimäki, N., and Taibi, D. (2020). Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study. *Journal of Systems and Software*, 170:110750.

Letouzey, J.-L. (2012). The sqale method for evaluating technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 31–36.

Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219.

Molnar, A.-J. and Motogna, S. (2020). Long-term evaluation of technical debt in open-source software. ESEM '20, New York, NY, USA. Association for Computing Machinery.

Monteith, J. Y. and McGregor, J. D. (2013). Exploring software supply chains from a technical debt perspective. In *2013 4th International Workshop on Managing Technical Debt (MTD)*, pages 32–38.

Mäntylä, M. V. and Lassenius, C. (2009). What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448.

Nikolaidis, N., Ampatzoglou, A., Chatzigeorgiou, A., Mittas, N., Konstantinidis, E., and Bamidis, P. (2023). Exploring the effect of various maintenance activities on the accumulation of td principal. In *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*, pages 102–111.

Nugroho, A., Visser, J., and Kuipers, T. (2011). An empirical model of technical debt and interest. MTD '11, page 1–8, New York, NY, USA. Association for Computing Machinery.

Palomba, F., Zaidman, A., Oliveto, R., and De Lucia, A. (2017). An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185.

Panichella, S., Arnaoudova, V., Di Penta, M., and Antoniol, G. (2015). Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170.

Panichella, S. and Zaugg, N. (2020). An empirical investigation of relevant changes and automation needs in modern code review. *Empirical Softw. Engg.*, 25(6):4833–4872.

Pascarella, L., Spadini, D., Palomba, F., and Bacchelli, A. (2019). On the effect of code review on code smells.

Rios, N., de Mendonça Neto, M. G., and Spínola, R. O. (2018). A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102:117–145.

Saarimäki, N., Baldassarre, M. T., Lenarduzzi, V., and Romano, S. (2019). On the accuracy of sonarqube technical debt remediation time. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 317–324.

SonarQube. Metric definitions. `https://docs.sonarsource.com/sonarqube/10.0/user-guide/metric-definitions/`. Accessed: 2024-03-30.

SonarSouce (2022). Sonarsource posts record growth with its clean code solution. `https://www.sonarsource.com/company/press-releases/sonar-record-growth-2022/`. Accessed: 2023-05-04.

SonarSource. Adding code rules. `https://docs.sonarsource.com/sonarqube/10.0/extension-guide/adding-coding-rules/`. Accessed: 2024-03-31.

Tan, J., Feitosa, D., Avgeriou, P., and Lungu, M. (2021). Evolution of technical debt remediation in python: A case study on the apache software ecosystem. *Journal of Software: Evolution and Process*, 33(4):e2319. e2319 smr.2319.

Trautsch, A., Herbold, S., and Grabowski, J. (2023). Are automated static analysis tools worth it? an investigation into relative warning density and external software quality on the example of apache open source projects. *Empirical Software Engineering*, 28(3):66.

Uchôa, A., Barbosa, C., Oizumi, W., Blenilio, P., Lima, R., Garcia, A., and Bezerra, C. (2020). How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511–522.

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. C. (2018). Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49.

Vetrò, A. (2012). Using automatic static analysis to identify technical debt. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1613–1615.

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A., and Streit, J. (2012). The quamoco product quality modelling and assessment approach. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 1133–1142. IEEE Press.

Yamashita, A. and Counsell, S. (2013). Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653.

Yu, P., Wu, Y., Peng, J., Zhang, J., and Xie, P. (2023). Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 569–580.

Zabardast, E., Bennin, K. E., and Gonzalez□Huerta, J. (2022). Further investigation of the survivability of code technical debt items. *J. Softw. Evol. Process*, 34(2).

Zazworka, N., Vetrò, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., and Shull, F. (2014). Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426.

Zou, W., Xuan, J., Xie, X., Chen, Z., and Xu, B. (2019). How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. *Empirical Software Engineering*, 24(6):3871–3903.

Özçevik, Y. (2024). Data-oriented qmood model for quality assessment of multi-client software applications. *Engineering Science and Technology, an International Journal*, 51:101660.