American Impact Review

Original Research

Self-admitted technical debt in modular open-source Ruby on Rails ecosystems: a multi-project empirical study

Anna Topalidi*

Independent Researcher
* Corresponding author
ORCID: 0009-0003-8831-9912

OPEN ACCESS

Citation:
Anna Topalidi (2026). Self-admitted technical debt in modular open-source Ruby on Rails ecosystems: a multi-project empirical study. *Am. Impact Rev.* 10.66308/air.e2026037

Received: April 8, 2026
Accepted: April 19, 2026
Published: April 20, 2026

DOI:
10.66308/air.e2026037

ISSN: 3071-124X

Abstract

Background: Self-Admitted Technical Debt (SATD) - intentional shortcuts documented by developers in source code comments - has been extensively studied in Java and Python, but the Ruby on Rails ecosystem remains unexamined despite its scale in civic tech, e-commerce, and social networking.
Aim: This study investigates the prevalence, distribution, and types of SATD in five prominent Ruby on Rails open-source projects, focusing on how modular architectures based on Rails engines shape SATD distribution.

Method: We mined 1.55 million lines of Ruby across Decidim, Discourse, Solidus, Spree, and Mastodon, extracted 907 SATD comments using keyword-based heuristics, analyzed density and module-level correlations, and manually classified a stratified sample of 400 comments.

Results: We report SATD under two scenarios: inclusive (treating rubocop:disable as formalised SATD) and classical (excluding it). The mean density is 0.50 SATD/KLOC in the inclusive scenario and 0.26 SATD/KLOC in the classical scenario - substantially below Java figures under either view. Formalised SATD - predominantly inline linter-suppression markers and dependency workarounds - accounts for 55.1% of validated instances, a category absent from Java-focused taxonomies. Module size predicts SATD count ($\rho = 0.684$, $p < 0.001$) but not density. Manual validation reveals a 9.8% overall false-positive rate, concentrated in the REVIEW and TEMPORARY keywords.

Conclusion: Ruby on Rails projects exhibit a distinct SATD profile in which debt is often expressed through RuboCop interaction rather than free-text comments. SATD taxonomies and detection tools should be extended with a formalised subclass to accommodate dynamically-typed ecosystems with strong linting cultures.

Keywords: Technical Debt, Ruby on Rails, Modular Architecture, Linter Suppression, Empirical Software Engineering, Mining Software Repositories

1. Introduction

Software systems accumulate technical debt over their lifecycle as developers make pragmatic trade-offs between short-term delivery and long-term code quality (Cunningham 1993). A particu-

larly visible form of this phenomenon is Self-Admitted Technical Debt (SATD) - instances where developers explicitly document suboptimal solutions in source code comments using markers such as TODO, FIXME, HACK, and similar annotations (Potdar and Shihab 2014).

Over the past decade, SATD has attracted significant research attention. Large-scale empirical studies have examined its prevalence and evolution across hundreds of Java projects (Bavota and Russo 2016), its detection through natural language processing (Maldonado, Shihab, and Tsantalis 2017), its identification across multiple software artifacts (Li, Soliman, and Avgeriou 2023), and its relationship with software quality (Wehaibi, Shihab, and Guerrouj 2016). However, the overwhelming majority of SATD research has focused on Java ecosystems, with limited exploration of other languages. Recent studies have examined SATD in R packages (Sharma et al. 2022), released the first SATD dataset for C++ (Pham et al. 2025), and investigated SATD in scientific software more broadly (Melin et al. 2026), yet the Ruby programming language - and the Ruby on Rails framework in particular - remains entirely unexamined in the SATD literature.

This gap is notable for several reasons. Ruby on Rails powers a substantial ecosystem of large-scale open-source platforms, including Decidim (participatory democracy, recognized as a UN Digital Public Good, deployed by over 500 institutions across 31 countries), Discourse (community forums), Mastodon (federated social networking), and major e-commerce platforms such as Solidus and Spree. Moreover, the Rails ecosystem has distinctive characteristics that may influence SATD patterns: a culture of convention over configuration, widespread use of code linters such as RuboCop, a modular architecture pattern based on Rails engines (packaged as Ruby gems), and extensive metaprogramming capabilities.

To address this gap, we formulate the following research questions:

- **RQ1:** What is the prevalence and density of SATD in large-scale Ruby on Rails open-source projects, and how does it compare with findings from Java-focused studies?
- **RQ2:** How is SATD distributed across modules in projects with modular Rails engine architectures?
- **RQ3:** What types of SATD are most common in Ruby on Rails projects, and do Ruby-specific SATD categories emerge that are absent from existing taxonomies?

This paper makes the following contributions:

- The first empirical study of SATD in the Ruby on Rails ecosystem, analyzing 907 SATD instances across 1.55 million lines of code in five major open-source projects.
- The identification of a *formalised SATD* subclass - RuboCop linter suppression markers and dependency workarounds - that constitutes 55.1% of validated SATD in the studied projects and is absent from existing SATD taxonomies. We report all key metrics under two scenarios (inclusive and classical) so that readers can interpret the results regardless of whether they accept linter suppressions as debt.
- A module-level analysis of SATD distribution in projects using the Rails engine architecture pattern, demonstrating that module size predicts SATD count but not SATD density.

- A publicly available replication package containing all extraction scripts, classified datasets, and analysis notebooks.

The remainder of this paper is organized as follows. Section 2 reviews related work on SATD. Section 3 describes our research methodology. Section 4 presents results for each research question. Section 5 discusses findings, implications, and threats to validity. Section 6 concludes the paper.

2. Related Work

2.1. Self-Admitted Technical Debt

The concept of technical debt was introduced by Cunningham (1993) as a metaphor for the long-term costs incurred by taking implementation shortcuts. Potdar and Shihab (2014) - recipient of the ICSME 2024 Most Influential Paper Award - coined the term Self-Admitted Technical Debt to describe instances where developers explicitly document such shortcuts in source code comments. Their keyword-based detection approach using patterns such as TODO, FIXME, and HACK established a foundational methodology for SATD research, and found that 2.4% to 31% of files contain SATD across four Java projects.

Bavota and Russo (2016) conducted a large-scale empirical study across 159 open-source Java projects, mining over 600,000 commits and 2 billion comments, and reported that SATD persists for more than one thousand commits on median before being removed. Maldonado and Shihab (2015) proposed a taxonomy of SATD types - design debt, defect debt, documentation debt, requirement debt, and test debt - based on manual classification of code comments. Subsequent work by Maldonado, Shihab, and Tsantalis (2017) applied natural language processing to automate SATD detection, achieving significant improvements over keyword-based approaches.

Sierra, Shihab, and Kamei (2019) provided a comprehensive survey of SATD research, identifying trends, tools, and open challenges. More recent work by Sheikhaei et al. (2024) investigated the effectiveness of large language models for SATD identification and classification, demonstrating that pre-trained models can outperform traditional machine learning approaches.

2.2. SATD and static analysis warning suppressions

A separate line of research examines *formalised* debt expressed through interaction with automated quality tools rather than natural-language comments. Rantala, Mäntylä, and Lenarduzzi (2024) compared keyword-labelled SATD with SonarQube findings across a large corpus of Java projects and reported a significant but limited overlap: only 15% of SATD comments actually address a static-analysis issue, suggesting that the two signals are complementary rather than redundant. Liargkovas, Panourgia, and Spinellis (2023) studied FindBugs/SpotBugs suppressions across 1,425 Java projects and argued that a substantial share of suppressions reflect handling of false positives and tool limitations rather than genuine technical debt. Hu et al. (2025) conducted the first large-scale study of suppressed static-analysis warnings across three languages and four analysers, and found that 50.8% of suppressions are *useless* - they do not actually suppress any real warning. These

findings directly motivate the dual-reporting strategy we adopt in Section 4: we report SATD metrics both with and without rubocop:disable directives so that readers who accept or reject formalised SATD as debt can both interpret the results.

2.3. SATD beyond Java

While Java dominates SATD research, several studies have begun exploring other language ecosystems. Sharma et al. (2022) examined SATD in R packages, finding that dynamically-typed scientific software exhibits distinct SATD characteristics compared to traditional object-oriented systems; they manually identified sixteen causes of SATD, eight of which were new to the literature. Building on this direction, Pham et al. (2025) released CppSATD, the first large-scale SATD dataset for C++, containing over half a million annotated comments across five SATD categories. Melin et al. (2026) extended the cross-ecosystem inquiry to scientific software more broadly, analysing SATD across source-code comments, commit messages, and pull requests. Despite this expanding coverage, to the best of our knowledge no empirical study of SATD in Ruby or Ruby on Rails has been published to date.

Li, Soliman, and Avgeriou (2023) proposed an approach for automated SATD identification integrating four sources - code comments, commit messages, pull requests, and issue tracking systems - across 103 open-source projects, demonstrating that SATD is evenly distributed across these sources. Zampetti et al. (2021) investigated SATD practices through developer interviews and surveys, finding that industrial developers are less likely to annotate technical debt compared to open-source contributors.

2.4. Technical debt and modular architectures

MacCormack and Sturtevant (2016) studied the relationship between system architecture and technical debt, showing that tightly coupled components in large software systems are associated with higher defect rates. The concept of architectural technical debt has been further explored by Standridge (2025), who investigated strategies for migrating object-oriented monoliths to modular architectures.

In the Ruby on Rails context, Shopify developed Packwerk, an open-source static analysis tool for enforcing boundaries between groups of Ruby files, recognizing that Rails provides limited built-in support for modular boundary enforcement. This tooling development acknowledges that modular Rails applications face unique challenges in managing coupling and technical debt across module boundaries.

2.5. Summary and research gap

While prior work has extensively examined SATD in Java projects and has begun exploring other language ecosystems, no study has investigated SATD in the Ruby on Rails ecosystem. This is a significant gap given the prevalence of large-scale Rails applications in open-source, the distinctive language features of Ruby (metaprogramming, dynamic typing, convention over configuration),

and the modular architecture patterns enabled by Rails engines. This paper addresses this gap by conducting the first empirical study of SATD across five major Ruby on Rails open-source projects.

3. Research Method

We conducted a mixed-methods empirical study following the Mining Software Repositories (MSR) methodology (Wohlin et al. 2012).

3.1. Subject selection

We selected five prominent Ruby on Rails open-source projects representing diverse application domains and architectural approaches. Table 1 summarizes the characteristics of the studied projects. Selection criteria required that projects: (1) are written primarily in Ruby on Rails, (2) have active development with commits within the last six months, (3) have at least 1,000 commits and 50 contributors, and (4) are available under an open-source license on GitHub. We included both modular projects (Decidim, Solidus, Spree), which organize functionality into separate Rails engines packaged as gems, and monolithic projects (Discourse, Mastodon), which use a conventional single-application structure.

Table 1. Characteristics of the five studied Ruby on Rails projects.

Project	Domain	Stars	Contrib.	Ruby LOC	KLOC	SATD	Arch.
Decidim	Civic tech	3K	180+	361,146	361.1	220	Modular
Discourse	Forum	43K	900+	789,755	789.8	441	Monolithic
Solidus	eCommerce	5K	800+	119,667	119.7	60	Modular
Spree	eCommerce	13K	800+	138,728	138.7	40	Modular
Mastodon	Social network	47K	800+	136,735	136.7	146	Monolithic
Total				1,546,031	1,546.0	907	

3.2. Data collection

SATD comments were extracted from Ruby source files (*.rb,* .erb, *.rake,* .gemspec, Gemfile, Rakefile) using keyword-based heuristics following the approach established by Potdar and Shihab (2014). We searched for the following primary SATD markers (case-insensitive): TODO, FIXME, HACK, XXX, WORKAROUND, KLUDGE, and TEMPORARY. Additionally, we included Ruby-specific markers: OPTIMIZE, REFACTOR, REVIEW, and rubocop:disable - inline linter suppression directives that indicate a conscious deviation from established code quality rules.

Files within vendor/, node_modules/, .bundle/, tmp/, log/, and auto-generated files (db/schema.rb) were excluded. Ruby pragmas such as # frozen_string_literal: true and encoding declarations were filtered out. For modular projects, module boundaries were determined by top-level directories containing .gemspec files (e.g., decidim-core/, decidim-proposals/ for Decidim; core/, backend/, api/ for Solidus and Spree).

Lines of code (LOC) were measured using cloc (Count Lines of Code), counting only Ruby source lines excluding blanks and comments. Module-level contributor counts were computed from git log by counting unique author email addresses per module directory.

3.3. Manual validation

To validate the keyword-based extraction and classify SATD types, we drew a stratified random sample of 400 comments (approximately 80 per project, adjusted for smaller projects). Each comment was classified by the first author into one of the following categories, adapted from Maldonado and Shihab (2015) and extended with Ruby-specific types:

- **Code/Design Debt** - suboptimal implementation or design choices acknowledged in comments.
- **Test Debt** - gaps in test coverage, test quality issues, or test workarounds.
- **Documentation Debt** - incomplete or inadequate documentation, naming issues.
- **Requirement Debt** - missing or incomplete features.
- **Ruby-Specific: Linter Suppression** - inline rubocop:disable directives suppressing code quality rules.
- **Ruby-Specific: Dependency** - workarounds for external gem or library bugs/limitations.
- **Not SATD** - false positives where the keyword appeared in a non-debt context (e.g., feature names, URL paths, test data).

To assess classification reliability, we compared the manual expert classification against the automated keyword-based pre-classification, computing Cohen's Kappa inter-rater agreement coefficient.

3.4. Analysis methods

For RQ1, we computed SATD density - SATD comments per thousand lines of code (KLOC) - denoted SATD/KLOC for each project and compared values with benchmarks from Java studies (Bavota and Russo 2016). For RQ2, we computed Spearman rank correlation coefficients between module size (LOC) and SATD count, module size and SATD density, and contributor count and SATD density. We applied the Mann-Whitney U test to compare SATD density between modular and monolithic projects, with Cliff's delta as the effect size measure. For RQ3, we analyzed the frequency distribution of SATD types across projects using the manually validated sample. All statistical tests used $\alpha = 0.05$ as the significance threshold.

4. Results

4.1. RQ1: SATD prevalence and density

We extracted a total of 907 SATD comments from 1,546,031 lines of Ruby code across the five studied projects. Table 2 presents the breakdown by SATD keyword.

Table 2. SATD keyword distribution across projects.

Keyword	Decidim	Discourse	Solidus	Spree	Mastodon	Total	%
rubocop:disable	200	57	17	9	96	379	41.8
TODO	0	194	31	8	36	269	29.7
REVIEW	3	102	0	0	1	106	11.7
TEMPORARY	10	13	0	4	7	34	3.7
HACK	3	19	3	1	4	30	3.3
WORKAROUND	3	19	3	5	0	30	3.3
FIXME	0	11	3	13	2	29	3.2
OPTIMIZE	0	11	0	0	0	11	1.2
XXX	0	10	0	0	0	10	1.1
REFACTOR	1	5	3	0	0	9	1.0
Total	220	441	60	40	146	907	100.0

The raw mean SATD density across projects - before applying any false-positive correction - was 0.59 SATD/KLOC (SD =0.29). Mastodon exhibited the highest raw density at 1.07 SATD/KLOC, while Spree had the lowest at 0.29 SATD/KLOC. After applying the per-keyword false-positive rates derived from manual validation (see Section 4.3), the corrected mean density drops to 0.50 SATD/KLOC in the inclusive Scenario A and 0.26 SATD/KLOC in the classical Scenario B (defined below). All per-project corrected values are reported in Table 3; the remainder of this section uses corrected values unless explicitly stated otherwise.

Both the raw and corrected Ruby densities are substantially lower than prevalence levels reported for Java projects in the SATD literature: Potdar and Shihab (2014) found SATD in 2.4% to 31% of files across four Java projects, and Bavota and Russo (2016) reported that SATD persists across thousands of commits in 159 Java projects, suggesting that the Ruby ecosystem exhibits a notably lower rate of explicitly documented debt.

A distinctive finding was the prominence of rubocop:disable as a SATD marker, accounting for 41.8% of all extracted SATD comments - the single most frequent marker across the entire dataset. This is a Ruby-ecosystem-specific phenomenon with no equivalent in classical Java-focused SATD studies. We discuss this as a case of *formalised SATD* - debt admitted through interaction with an automated quality tool rather than free-text comments - and in light of concerns raised by Hu et al. (2025), Liargkovas, Panourgia, and Spinellis (2023) and Rantala, Mäntylä, and Lenarduzzi (2024) we report all downstream metrics under two scenarios:

Scenario A (inclusive).

rubocop:disable is treated as SATD. Raw N=907; after applying per-keyword false-positive rates from manual validation, N≈775 true SATD, giving a mean density of 0.50 SATD/KLOC.

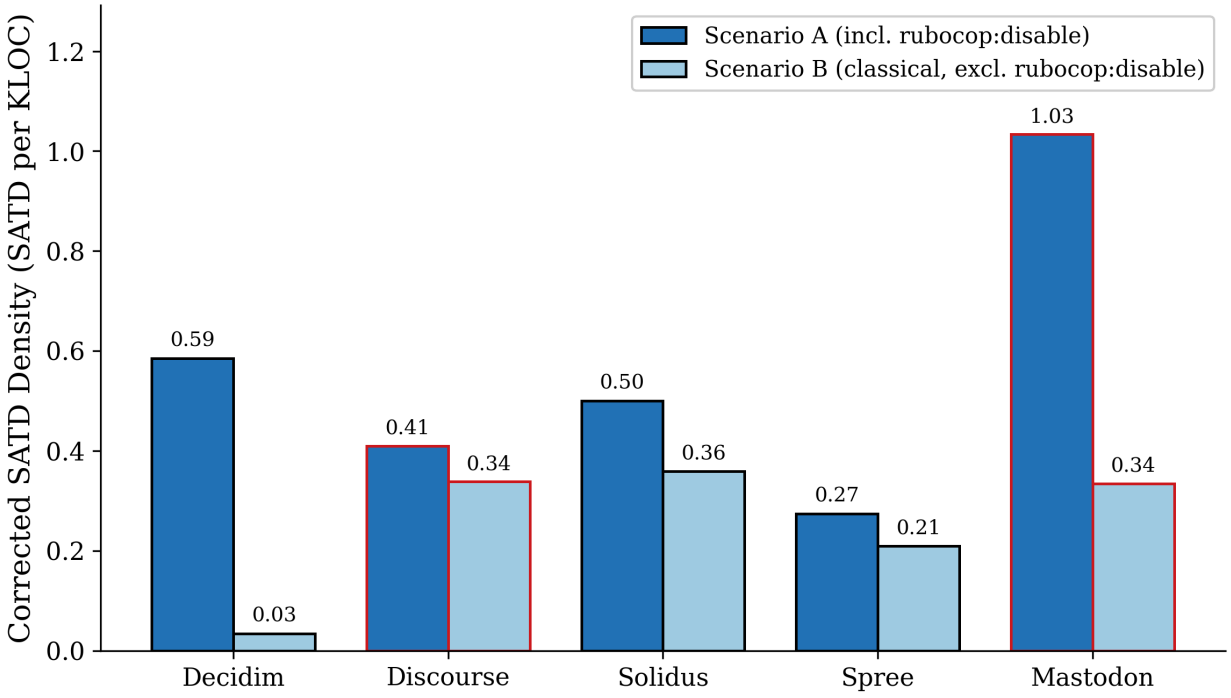
Scenario B (classical).

rubocop:disable is excluded and only TODO/FIXME/HACK-style comments are counted. Raw N=528, corrected N≈398, mean density 0.26 SATD/KLOC.

Under either scenario Ruby on Rails projects show lower SATD density than Java baselines, though the gap is larger for Scenario B. Table 3 summarises per-project densities in both scenarios. In contrast to rubocop:disable, TODO markers - which dominate Java SATD studies - accounted for 29.7% of Ruby SATD instances. The REVIEW keyword had the highest false-positive rate among all markers (93.1% in our manual sample); manual validation (Section 4.3) revealed that the majority of REVIEW instances in Discourse represented feature names and URL paths rather than genuine SATD.

Table 3. Dual reporting of corrected SATD density (per KLOC) across the five studied projects. Scenario A includes rubocop:disable suppressions; Scenario B excludes them. Corrected counts are obtained by extrapolating per-keyword false-positive rates observed in the manual-validation sample (N=400) to the full auto-extracted dataset (N=907), under the assumption that a keyword's FP rate is stable between sample and population.

Project	KLOC	% rubocop:disable	A (incl.)	B (classical)
Decidim	361.1	90.9%	0.59	0.04
Discourse	789.8	12.9%	0.41	0.34
Solidus	119.7	28.3%	0.50	0.36
Spree	138.7	22.5%	0.27	0.21
Mastodon	136.7	65.8%	1.03	0.34
Mean	309.2	44.1%	0.50	0.26



Black border: modular project (Decidim, Solidus, Spree). Red border: monolithic project (Discourse, Mastodon).

Corrected SATD density (per KLOC) across the five studied Ruby on Rails projects under both reporting scenarios. Dark bars: Scenario A (inclusive, includes rubocop:disable). Light bars: Scenario B (classical, excludes rubocop:disable). Black borders mark modular projects (Decidim, Solidus, Spree); red borders mark monolithic projects (Discourse, Mastodon). Values are after applying per-keyword false-positive corrections from the manual-validation sample.

Summary (RQ1): Ruby on Rails projects exhibit lower overall SATD density than Java projects reported in the literature under both the inclusive (0.50/KLOC) and classical (0.26/KLOC) measurement scenarios. The most common SATD marker in Ruby is rubocop:disable (41.8%), followed by TODO (29.7%), representing a significant departure from Java ecosystems where TODO typically dominates.

4.2. RQ2: SATD distribution across modules

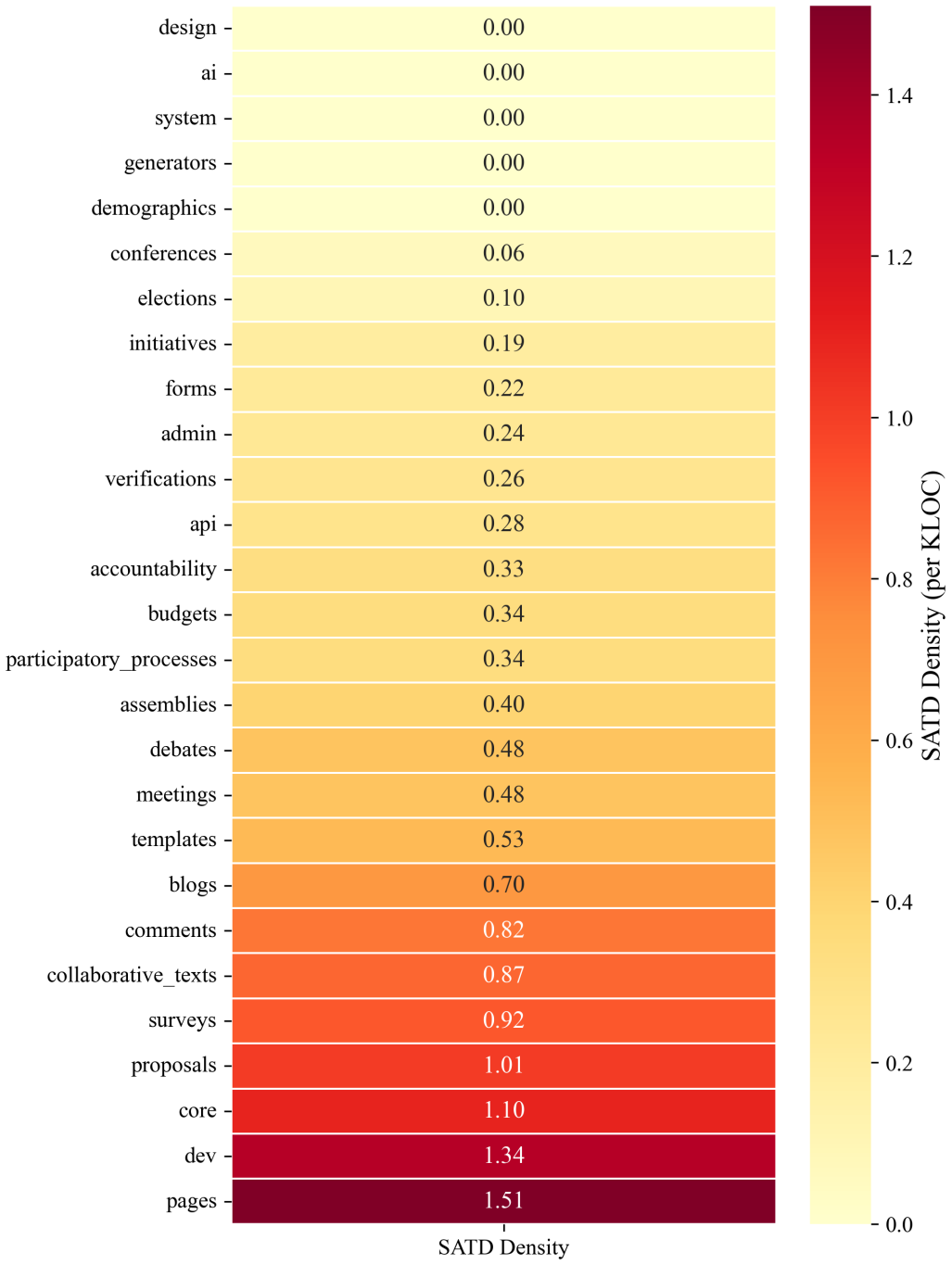
For the three modular projects (Decidim, Solidus, Spree), we analyzed SATD distribution across 38 modules. Table 4 presents the top-10 modules by SATD density.

Table 4. Top 10 modules by SATD density across modular projects.

#	Module	Project	LOC	KLOC	SATD	Density
1	decidim-pages	Decidim	1,325	1.3	2	1.51
2	decidim-dev	Decidim	3,720	3.7	5	1.34
3	decidim-core	Decidim	96,517	96.5	106	1.10
4	solidus_promotions	Solidus	14,851	14.9	16	1.08
5	decidim-proposals	Decidim	31,581	31.6	32	1.01
6	decidim-surveys	Decidim	5,449	5.4	5	0.92
7	decidim-collaborative_texts	Decidim	4,579	4.6	4	0.87
8	decidim-comments	Decidim	7,280	7.3	6	0.82
9	decidim-blogs	Decidim	4,302	4.3	3	0.70
10	solidus_core	Solidus	48,111	48.1	26	0.54

Module size (LOC) was a strong predictor of absolute SATD count (Spearman $\rho = 0.684$, $p < 0.001$), indicating that larger modules accumulate more SATD comments. However, module size did not significantly predict SATD density ($\rho = 0.234$, $p = 0.157$), suggesting that SATD is distributed roughly proportionally to code volume rather than concentrating in specific modules. The relationship between contributor count and SATD density showed a borderline trend ($\rho = 0.295$, $p = 0.072$) that did not reach statistical significance.

Comparing modular and monolithic projects, the mean SATD density was lower in modular projects (0.47 SATD/KLOC) compared to monolithic projects (0.82 SATD/KLOC). However, with only five projects, the Mann-Whitney U test did not yield a statistically significant difference ($U = 1.0$, $p = 0.400$), and this observation should be interpreted with caution.



SATD density heatmap across Decidim modules, sorted by density (highest at top).

Summary (RQ2): Module size strongly predicts absolute SATD count ($\rho = 0.684$, $p < 0.001$) but not normalized SATD density ($\rho = 0.234$, $p = 0.157$). SATD appears to scale proportionally with module size rather than clustering in specific modules.

4.3. RQ3: SATD types

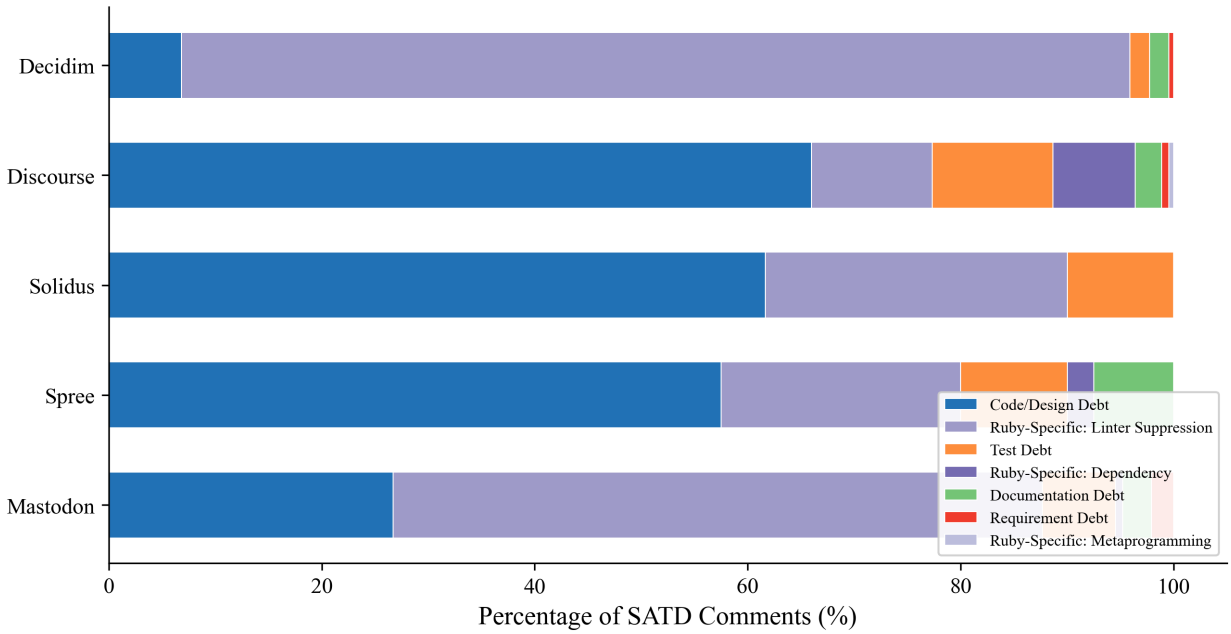
The manual classification of 400 sampled SATD comments revealed 39 false positives (9.8%), predominantly caused by the keyword REVIEW appearing in URL paths and feature names in Discourse's moderation system, and TEMPORARY appearing in parameter documentation. After removing false positives, 361 comments were classified as true SATD instances. Cohen's Kappa between the automated pre-classification and expert manual classification was $\kappa = 0.740$, indicating substantial agreement (Landis and Koch 1977).

Table 5. Distribution of SATD types based on manual classification (N = 361 validated SATD instances from a sample of 400 comments; 39 false positives excluded).

SATD Type	Count	%
Ruby-Specific: Linter Suppression	187	51.8
Code/Design Debt	129	35.7
Test Debt	26	7.2
Ruby-Specific: Dependency	12	3.3
Documentation Debt	4	1.1
Requirement Debt	3	0.8
Total	361	100.0

The most striking finding is the dominance of what we term *formalised SATD* - debt admitted through interaction with an automated quality tool rather than through free-text comments. Linter Suppression (rubocop:disable directives) alone accounted for 51.8% of validated SATD instances in the manual sample, and when combined with Dependency workarounds (3.3%), Ruby-specific debt comprised 55.1% of the validated sample. This category has no counterpart in existing SATD taxonomies derived from Java studies. We deliberately separate this formalised subclass from the classical SATD types both in reporting (Table 3) and in discussion (Section 5), so that readers who do not accept rubocop:disable as debt can still interpret the remaining findings.

In the Scenario B view - excluding rubocop:disable entirely - Code/Design Debt becomes the dominant type, matching prior Java studies, while formalised SATD accounts for the bulk of the Scenario A-vs-B gap. Overall, in the validated manual sample, Code/Design Debt accounted for 35.7% of SATD in Ruby projects, Test Debt for 7.2%, Documentation Debt for 1.1%, and Requirement Debt for 0.8%.



Distribution of SATD types across the five studied projects (manually validated sample, N = 361).

Notably, the composition of SATD varied substantially across projects. Decidim's SATD profile was overwhelmingly dominated by Linter Suppression (91.6%), reflecting the project's strict RuboCop configuration and the prevalence of Rails/SkipsModelValidations suppressions in database migration code. In contrast, Discourse exhibited a more traditional SATD profile with Code/Design Debt as the dominant type. Spree showed the highest proportion of Ruby-Specific: Dependency debt, driven by workarounds for the Mobility internationalization gem.

Summary (RQ3): Formalised SATD - predominantly rubocop:disable directives and, to a lesser extent, dependency workarounds - constitutes 55.1% of validated SATD in the studied projects, representing a novel SATD subclass absent from existing taxonomies. Under the conservative Scenario B that excludes rubocop:disable, Code/Design Debt becomes the dominant type, and the distribution resembles prior Java studies. SATD composition varies substantially across projects, influenced by code-quality tooling configuration and external dependency choices.

5. Discussion

5.1. Interpretation of findings

The lower SATD density in Ruby projects (0.50/KLOC under the inclusive Scenario A, 0.26/KLOC under the classical Scenario B) compared to Java benchmarks may reflect multiple factors. Ruby's expressive syntax and the Rails "convention over configuration" philosophy may reduce the need for workarounds that generate SATD. Additionally, the Ruby community's strong emphasis on code quality tooling (RuboCop) may encourage developers to address debt through linter configuration rather than leaving TODO/FIXME comments.

The dominance of rubocop:disable as a SATD marker reveals a fundamental difference in how Ruby developers manage technical debt. Rather than documenting debt in natural-language comments, Ruby developers interact with a structured system: the linter identifies a violation, and the developer explicitly suppresses it. Each suppression is a documented decision to deviate from an established quality rule. We call this a *formalised SATD* subclass to distinguish it from the classical SATD captured by TODO/FIXME/HACK markers, and to acknowledge the concerns raised by Hu et al. (2025) and Liargkovas, Panourgia, and Spinellis (2023) that not every suppression necessarily corresponds to an active debt item. The most frequently suppressed rule in our dataset was Rails/SkipsModelValidations, indicating that database operations bypassing Active Record validations represent a recurring and acknowledged debt pattern in Rails applications - these are, by construction, real debt items rather than tool-limitation false positives. Rantala, Mäntylä, and Lenarduzzi (2024) found in Java that only 15% of SATD comments address a static-analysis issue; our findings suggest that in Ruby the relationship is inverted, with the formalised signal accounting for the bulk of developer-acknowledged debt.

An extreme case worth highlighting is Decidim: under the classical Scenario B, Decidim's corrected SATD density drops to 0.04 SATD/KLOC - effectively zero. On 361 KLOC of Ruby code, only a handful of TODO/FIXME/HACK comments remain, while 90.9% of all extracted SATD comes from rubocop:disable directives. In other words, Decidim has almost entirely migrated its self-admitted debt out of free-text comments and into the formalised linter-suppression channel. This is itself an interesting finding: the strength of the Ruby community's linting culture appears to be strong enough, at least in some large projects, to effectively replace traditional SATD markers. It also illustrates why treating rubocop:disable as SATD is methodologically necessary for Ruby studies: under Scenario B alone, Decidim would appear to have almost no self-admitted debt at all, which clearly contradicts the lived reality of a 361-KLOC modular codebase.

The high false positive rate for the REVIEW keyword (predominantly in Discourse, where "review" is a core feature name for content moderation) highlights the importance of context-aware SATD detection. Simple keyword matching without semantic analysis can inflate SATD counts in projects where SATD markers coincide with domain terminology.

5.2. Implications for practitioners

Our findings suggest several actionable recommendations for Ruby on Rails project maintainers:

First, monitoring rubocop:disable directives should be incorporated into technical debt management practices. These markers represent quantifiable, trackable debt that can be systematically reduced through dedicated refactoring efforts.

Second, projects using modular Rails engine architectures do not appear to suffer from SATD concentration in specific modules - SATD scales proportionally with module size. This suggests that the modular architecture does not introduce additional debt management challenges beyond those of larger codebases in general.

Third, projects with heavy reliance on external gems (such as Spree's dependency on Mobility) should track dependency-related workarounds as a distinct debt category, as these represent debt whose resolution depends on upstream library fixes rather than internal refactoring.

5.3. Implications for researchers

The discovery of Ruby-Specific SATD categories - linter suppression and dependency workarounds - suggests that existing SATD taxonomies require extension to accommodate language-ecosystem-specific debt types. Current SATD detection tools, predominantly trained on Java corpora, may miss or misclassify significant portions of SATD in dynamically-typed language ecosystems.

Future research should investigate whether similar language-specific SATD patterns exist in other ecosystems with strong linting cultures (e.g., ESLint suppressions in JavaScript/TypeScript, # noqa comments in Python). The concept of "formalized SATD" - debt admitted through interaction with automated quality tools rather than free-text comments - represents a fruitful avenue for theoretical development.

5.4. Threats to validity

Internal validity. Keyword-based SATD extraction may miss debt expressed without recognised markers. Our manual validation revealed a 9.8% overall false-positive rate, almost entirely driven by keyword polysemy (REVIEW as a Discourse feature name, 93.1% FP, and TEMPORARY as a parameter-documentation word, 46.7% FP); by contrast, rubocop:disable had a false-positive rate of only 0.5% in the sample.

Including rubocop:disable as a SATD marker is a methodological choice that substantially affects the reported density. Hu et al. (2025) recently showed that 50.8% of static-analysis suppressions across three languages and four analysers are "useless" - they do not suppress any real warning - and Liargkovas, Panourgia, and Spinellis (2023) argued that a sizeable share of Java suppressions reflect false-positive handling rather than genuine debt. While our manual validation confirmed that the *syntactic* rate of spurious rubocop:disable directives in our corpus is very low, we cannot rule out the possibility that some suppressions target questionable cops rather than real quality violations. We therefore report all key metrics under two scenarios (A: inclusive, B: classical, see Table 3) so that readers can judge whether to accept the formalised subclass. Under Scenario B the conclusion - Ruby on Rails projects exhibit lower SATD density than Java baselines and no clustering of debt in specific modules - still holds.

Classification was performed by a single expert (the first author, a senior contributor to one of the studied projects); while this introduces potential bias, the substantial Cohen's Kappa agreement (κ=0.740) with automated pre-classification provides confidence in classification consistency. The same automated extraction methodology was applied uniformly to all five projects, and no project-specific adjustments were made to the analysis pipeline.

External validity. Results may not generalize beyond the five studied projects or beyond the Ruby on Rails ecosystem. The projects were selected for prominence and diversity but may not represent the broader population of Rails applications. In particular, private and commercial Rails codebases may exhibit different SATD patterns.

Construct validity. SATD density as measured through comment analysis captures only explicitly documented debt. Developers may introduce debt without documenting it, and the propensity to document debt may vary across projects and developer cultures. The inclusion of rubocop:disable

as a SATD indicator is novel and may not be directly comparable with Java-focused studies that do not include analogous markers.

Reliability. All extraction scripts, classified datasets, and analysis notebooks are available in our replication package to enable independent verification and replication of our findings.

6. Conclusions

This paper presented the first empirical study of Self-Admitted Technical Debt in the Ruby on Rails open-source ecosystem. We analyzed 907 SATD instances across 1,546,031 lines of Ruby code in five prominent projects - Decidim, Discourse, Solidus, Spree, and Mastodon.

Our key findings are:

- Ruby on Rails projects exhibit a mean SATD density of 0.50 per KLOC in the inclusive scenario and 0.26 per KLOC in the classical scenario (excluding rubocop:disable), substantially lower than densities reported for Java projects in prior studies under either measurement choice.
- The most common SATD marker in Ruby is rubocop:disable (41.8%), which we treat as a *formalised SATD* subclass - debt admitted through interaction with an automated quality tool rather than through free-text comments - a phenomenon absent from Java-focused SATD research.
- Formalised SATD (linter suppressions plus dependency workarounds) constitutes 55.1% of validated SATD instances, necessitating extensions to existing SATD taxonomies.
- In modular projects, SATD scales proportionally with module size ($\rho = 0.684$, $p < 0.001$) without concentrating in specific modules.
- Manual validation of keyword-based extraction revealed a 9.8% false positive rate, primarily due to keyword polysemy in domain-specific contexts.

Future work should extend this analysis longitudinally to examine how SATD evolves over project releases, investigate SATD in other dynamically-typed ecosystems with strong linting cultures, and develop automated detection tools adapted for Ruby-specific debt patterns. Cross-ecosystem comparative studies - analyzing the same SATD markers across Java, Ruby, Python, and JavaScript projects - would help establish whether the patterns identified here are unique to Ruby or reflect broader trends in dynamically-typed language communities.

Funding Statement

This research received no external funding.

Data Availability Statement

The dataset and analysis scripts are permanently archived on Zenodo (DOI: 10.5281/zenodo.19639725).

Conflicts of Interest

The author declares no conflicts of interest.

References

1. Bavota, Gabriele, and Barbara Russo. 2016. "A Large-Scale Empirical Study on Self-Admitted Technical Debt." In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 315 - 26. https://doi.org/10.1145/2901739.2901742.

2. Cunningham, Ward. 1993. "The WyCash Portfolio Management System." *ACM SIGPLAN OOPS Messenger* 4 (2): 29 - 30. https://doi.org/10.1145/157710.157715.

3. Hu, Huimin, Yiyang Liu, Cristian-Alexandru Staicu, and Michael Pradel. 2025. "An Empirical Study of Suppressed Static Analysis Warnings." *Proceedings of the ACM on Software Engineering* 2 (FSE). https://doi.org/10.1145/3715729.

4. Landis, J. Richard, and Gary G. Koch. 1977. "The Measurement of Observer Agreement for Categorical Data." *Biometrics* 33 (1): 159 - 74. https://doi.org/10.2307/2529310.

5. Li, Yikun, Mohamed Soliman, and Paris Avgeriou. 2023. "Automatic Identification of Self-Admitted Technical Debt from Four Different Sources." *Empirical Software Engineering* 28 (3): 65. https://doi.org/10.1007/s10664-023-10297-9.

6. Liargkovas, Georgios, Evangelia Panourgia, and Diomidis Spinellis. 2023. "Quieting the Static: A Study of Static Analysis Alert Suppressions." arXiv preprint arXiv:2311.07482. https://doi.org/10.48550/arXiv.2311.07482.

7. MacCormack, Alan, and Daniel J. Sturtevant. 2016. "Technical Debt and System Architecture: The Impact of Coupling on Defect-Related Activity." *Journal of Systems and Software* 120: 170 - 82. https://doi.org/10.1016/j.jss.2016.06.007.

8. Maldonado, Everton da Silva, and Emad Shihab. 2015. "Detecting and Quantifying Different Types of Self-Admitted Technical Debt." In *Proceedings of the 7th International Workshop on Managing Technical Debt (MTD)*, 9 - 15. https://doi.org/10.1109/MTD.2015.7332619.

9. Maldonado, Everton da Silva, Emad Shihab, and Nikolaos Tsantalis. 2017. "Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt." *IEEE Transactions on Software Engineering* 43 (11): 1044 - 62. https://doi.org/10.1109/TSE.2017.2654244.

10. Melin, Erin L., Nasir U. Eisty, Gregory Watson, and Addi Malviya-Thakur. 2026. "Multi-Artifact Analysis of Self-Admitted Technical Debt in Scientific Software." *arXiv Preprint arXiv:2601.10850*.

11. Pham, Phuoc, Murali Sridharan, Matteo Esposito, and Valentina Lenarduzzi. 2025. "Descriptor: C++ Self-Admitted Technical Debt Dataset (CppSATD)." *IEEE Data Descriptions*. https://doi.org/10.48550/arXiv.2505.01136.

12. Potdar, Aniket, and Emad Shihab. 2014. "An Exploratory Study on Self-Admitted Technical Debt." In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 91 - 100. https://doi.org/10.1109/ICSME.2014.31.

13. Rantala, Leevi, Mika Mäntylä, and Valentina Lenarduzzi. 2024. "Keyword-Labeled Self-Admitted Technical Debt and Static Code Analysis Have Significant Relationship but Limited Overlap." *Software Quality Journal* 32 (2): 391 - 429. https://doi.org/10.1007/s11219-023-09655-z.

14. Sharma, Rishab, Ramin Shahbazi, Fatemeh H. Fard, Zadia Codabux, and Melina Vidoni. 2022. "Self-Admitted Technical Debt in R: Detection and Causes." *Automated Software Engineering* 29 (2): 53. https://doi.org/10.1007/s10515-022-00358-6.

15. Sheikhaei, Mohammad Sadegh, Yuan Tian, Shaowei Wang, and Bowen Xu. 2024. "An Empirical Study on the Effectiveness of Large Language Models for SATD Identification and Classification." *Empirical Software Engineering* 29 (6): 159. https://doi.org/10.1007/s10664-024-10548-3.

16. Sierra, Giancarlo, Emad Shihab, and Yasutaka Kamei. 2019. "A Survey of Self-Admitted Technical Debt." *Journal of Systems and Software* 152: 70 - 82. https://doi.org/10.1016/j.jss.2019.02.056.

17. Standridge, Lionel. 2025. "Architectural Technical Debt Migrating Object Oriented Systems to Modular Architectures." PhD thesis, Nova Southeastern University.

18. Wehaibi, Sultan, Emad Shihab, and Latifa Guerrouj. 2016. "Examining the Impact of Self-Admitted Technical Debt on Software Quality." In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 179 - 88. https://doi.org/10.1109/SANER.2016.72.

19. Wohlin, Claes, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2.

20. Zampetti, Fiorella, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. 2021. "Self-Admitted Technical Debt Practices: A Comparison Between Industry and Open-Source." *Empirical Software Engineering* 26 (6): 130. https://doi.org/10.1007/s10664-021-10031-3.