“That Developer Left the Project!”: An Introduction and Case Study of Turnover Technical Debt

Roberto Verdecchia
University of Florence
Italy
roberto.verdecchia@unifi.it

Edoardo Sarri
University of Florence
Italy
edoardo.sarri@edu.unifi.it

Enrico Vicario
University of Florence
Italy
enrico.vicario@unifi.it

Abstract

Throughout the years, the attention on technical debt experienced a steady growth, and can now boast to be a consolidated concept within the software engineering field. Despite the growing academic and industrial interest in the topic, a category of technical debt appears to date to be unexplored. In this paper, we introduce the concept of turnover technical debt, *i.e.*, technical debt that arises when developers leave projects with software artifacts other developers will struggle to work with. Our contribution presents an initial formulation of the phenomenon, which intuitively revolves around two co-occurring properties in software artifacts, namely (i) centralized ownership and (ii) low understandability and suboptimal documentation. Based on these two properties, we present an approach outlining a conceptual basis to quantitatively observe turnover technical debt *via* a mix of repository mining and static analysis. We complement our contribution with a mixed-method case study conducted with the twofold goal of assessing the viability of the proposed measurement approach and collecting initial insights from practitioners on the phenomenon. The gathered results point to the relevance of turnover technical debt in practice, and to promising avenues to more precisely measure the phenomenon.

1 Introduction

> “*There was a developer who wrote a component that nobody knows how it works, and so we are all afraid of touching it. It works well for now, but if something stops working, or we have to touch that, for example to implement some new functionality, we could have a problem*”. R&D Director [1].

Technical debt can take various shapes and originate from heterogeneous sources. In the literature a plethora of different debt items have been considered, ranging from the evergreen code debt, to less studied debt types such test, documentation, and versioning debt [2]. Similarly, numerous research efforts studied what can be at the origin of technical debt, uncovering prominent causes such as time pressure, unsuitable implementation decisions, and the passing of time [1, 3, 4].

Despite the evergrowing corpus of literature, we noticed a potential cause of technical debt that remains to date unexplored. To put it in the words of the Engineering Productivity Research Team Staff: “*A large proportion of code written by someone no longer on the team might indicate that the team lacks expertise*” [5]. The idea introduced in this paper gravitates around such intuition, and is intuitively referred to as *Turnover Technical Debt* (TTD). TTD embodies a potential source of technical debt, and manifests itself as a software component that is hard to evolve and maintain when the developer(s) responsible for it stops providing support for it. As a simple example, TTD can arise when a single developer takes exclusive ownership of a software component as expedient for faster development, while neglecting to ensure the component is also maintainable by other developers. By considering the rapid adoption of Large Language Models (LLMs) for code generation tasks, the concept of TTD can be ported to unrevised generated code that no developer has a clear understanding of.

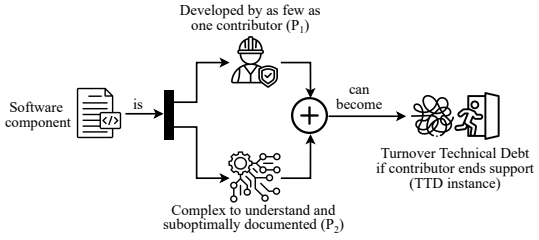

Figure 1: Overview of the turnover technical debt phenomenon

In this research we make the first steps to understand the TTD phenomenon. The main contributions presented are:

- The introduction of the Turnover Technical Debt phenomenon as a potential source of technical debt,
- A coarse intuition on how the phenomenon could be quantitatively measured, and
- A mixed-method case study to assess the usefulness of the introduced concepts and refine them.

The research is intended to support (i) researchers by opening the discourse on a phenomenon that may lie at the root of some technical debt items, supported by a sketch of an approach to measure it for future researchers to build upon, and (ii) practitioners by bringing to light and raising awareness on a phenomenon that may already be, or could become, a source of their technical debt.

The replication package of this study is available at: https://figshare.com/s/6447a47b52d8fd808e5f.

TechDebt '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2485-5/2026/04
https://doi.org/10.1145/3794915.3795779

2 Turnover Technical Debt: The "That Developer Left the Project!" Phenomenon

In this section we describe the intuition behind the TTD phenomenon, with Figure 1 depicting an overview of its key concepts.

In short, we define TTD as follows:

Turnover technical debt manifests itself as a software component, or set thereof, that becomes hard to evolve and maintain once the developer responsible for it stops supporting its development. More specifically, we hypothesize there are two key properties a software components needs to possess in order to be a TTD instance, namely:

- **Centralized Ownership (P_1):** *The software component was created and maintained by a small fraction of the development team, with as few as one contributor.*
- **Low Understandability and Suboptimal Documentation (P_2):** *The software component is characterized by low understandability and suboptimal documentation.*

The formulation of the first property (P_1) reflects the centralized knowledge of few developers regarding the implementation details of a TTD instance. In other words, the internal workings of a TTD item are known only by a small fraction of the development team, and are not widespread knowledge. The fewer the developers who are familiar with the implementation details of a TTD, the higher the probability such knowledge will be lost due to developer turnover.

The rationale behind the formulation of the second property (P_2) instead assures that the centralized knowledge embodied by P_1 cannot be obtained through means other than asking the knowledgeable developers for assistance, *e.g.*, through simple code inspection or available documentation. In simple terms, P_2 captures the property of a TTD instance to be difficult to understand for a developer who is not familiar with it.

TTD instances start in most cases their lifecycle as "dormant" technical debt items [1], *i.e.*, they do lead to any immediate consequence. A TTD item starts impeding development activities when two subsequent events occur. As first event (E_1), the developers knowledgeable about the TTD implementation stop providing support for its maintenance and evolution for various reasons, *e.g.*, they left the project or simply lost interest in providing assistance for it. As second event (E_2), the TTD item needs to be modified, or it is less directly involved in a software maintenance or evolution task (*e.g.*, due to dependencies).

Intuitively, TTD can manifest itself in various ways, ranging from teams avoiding working on TTD components no one fully understands, to the need of heavily refactoring or ever rewriting TDD instances from scratch to completely pay off the debt.

As for technical debt in general, the ability to systematically identify and measure TTD instances is a paramount factor for its management [6–8]. Ideally, TTD occurrences are discovered before the first triggering event occurs (E_1), so that the developer(s) familiar with the TTD items are given the possibility to pass their knowledge and/or improve the understandability of such instances. Therefore, based on such conjecture, TTD instances are best managed *via* a preemptive strategy, by identifying and mitigating such occurrences before these start to impede development activities.

To move the intuition of TTD beyond its introduction and take a more proactive stance, in the next section we present our coarse intuition on how the phenomenon could quantitatively be measured. The research is further supported by a small case study (Section 4) conducted to both assess the viability of the presented approach and gain preliminary insights on the topic from developers.

3 On Measuring Turnover Technical Debt

In this section, we briefly sketch how TTD instances could be quantitatively identified and measured *via* static software artifact analysis. As disclaimer, by no means we claim the presented measurement approach to be accurate, polished, or complete. Rather, the presented approach should just be interpreted as an initial intuition on which other researchers can built upon. As further motivation of the pilot approach, its preliminary results allow us to bootstrap through concrete examples interviews with developers, which are conducted as complementary portion of this study (see Section 4). Regarding the metrics used in our preliminary approach, we also warn that these should be interpreted as coarse proxies of the construct we are set to observe. A thorough discussion how such metrics could be refined to more precisely measure TDD is documented Section 5.

Regarding how TDD can be measured, based on our formulation (see Section 2), TTD can be identified by the co-occurrence of two distinct properties, namely centralized ownership (P_1) and low understandability and suboptimal documentation (P_2). We therefore split our TTD measurement approach in two distinct measurement strategies to assess the presence of P_1 and P_2.

3.1 Measuring Centralized Ownership (P_1)

To measure centralized ownership, we leverage the commit history of files. More specifically, for every file of a project we consider the total code churn [9], *i.e.*, lines of code added, deleted, and modified, weighted by the churn of each individual contributor as proxy for their ownership of files. In other words, we measure centralized code ownership as:

$$CO_{\text{dev}_i}(\text{file}) = \frac{\sum_{d \in D} \text{churn}_d(\text{file})}{\text{churn}_{\text{dev}_i}(\text{file})}$$

where D is the set of all developers who edited a file, $\text{churn}_d(\text{file})$ is the total lines added, removed, or modified in the file by developer $d \in D$, and dev_i is the specific developer being considered.

3.2 Measuring Low Understandability and Suboptimal Documentation (P_2)

To measure P_2, we rely on the Cognitive Complexity (*CogC*) metric [10]. As an evolution of the infamous cyclomatic complexity metric, cognitive complexity is designed to measure code understandability by considering different source code constructs, such as breaks in the linear flow and nesting. As the metric focuses on a core facet of P_2, namely understandability, and was empirically validated [11], we choose this metric as exemplary proxy.

3.3 Combining Centralized Ownership with Low Understandability and Suboptimal Documentation

Trivially, in our preliminary approach we rank TTD severeness by adopting the lexicographic order on the tuple $(CogC, CO_{\text{dev}_i})$,

where *CogC* serves as the primary criterion and $CO_{\text{dev}_i} \forall i$ as the secondary criterion. The choice to adopt such ranking strategy derived from its trivial transparency and support in sampling files for interviews conducted for our case study (see Section 4.2.2). As improvement opportunity, future research can consider more refined solutions to combine the measurements of P_1 and P_2, *e.g.*, by utilizing a Pareto frontier.

4 Case Study

In this section we report the case study [12] conducted with the twofold objective of (i) assessing the viability of our proposed approach (see Section 3), and (ii) collecting initial insights from practitioners of the TTD phenomenon.

4.1 Case Study Object

As object of our case study, we select the PMD project, an open source extensible cross-language static code analyzer hosted on GitHub.[1] In its current state[2], the PMD project accounts for 273.7k lines of code (LOC), primarily written in Java (210.5k LOC, 76.9%), Apex (24.6k LOC, 9%), and Kotlin (24k LOC, 8.0%). Over 14 years of development, the project counts as of October 29, 2025 more than 300 contributors spanning over 108 releases and 30.3k commits, with the most recent commit pushed to the repository less than a week ago. The project was starred 5.2k times and forked 1.5k, demonstrating substantial community engagement.

4.2 Case Study Research Process

Our case study research process is divided in two main phases, as depicted in Figure 2 and further detailed below.

4.2.1 Phase 1: Quantitative Phase. In a first research phase, we run our approach (see Section 3) on the case study repository. To measure P_1, we mine the repository commits to calculate the centralized ownership of each file present in the repository as defined in Section 3.1. To calculate P_2 instead, we analyze *via* PMD all Java source files of the repository to calculate their *CogC*.[3] We opt to focus on the Java files of the repository as they constitute the primary language in which the project is implemented (see Section 4.1), and hence is w.h.p. familiar to the most active contributors to be interviewed. This assumption is later corroborated by a developer interviewed during Phase 2, who stated "*I'm not that interested in the modules that are not Java*". The final output of this phase is a ranking of source files that are more likely to be affected by TTD (see Section 3.3 for more information on the final ranking process).

4.2.2 Phase 2: Qualitative Phase. In the second research, we conduct semi-structured interviews with two developers of our case study object. We select the developers to be interviewed, referred to as D_1 and D_2, *via* purposive sampling by selecting the ones with the highest count of commits to the case study object (respectively 35.1% and 25.1% of the total commits), as they are intuitively the individuals most familiar with the project. The interviews open with an introduction of the TDD concept and this research, followed by a discussion of six files sampled from the output of Phase 1. For each file, participants are asked to indicate whether they consider it a TDD instance. Follow-up questions are posed to gain more insights into each answer. The files are selected *via* stratified sampling to cover four distinct scenarios of decreasing TDD potential, namely (S_1) high cognitive complexity and a single contributor, (S_2) high cognitive complexity and multiple contributors, (S_3) low cognitive complexity and a single contributor, and (S_4) low cognitive complexity and multiple contributors. To enrich our results, for scenarios S_1 and S_3 we consider two different situations, *i.e.*, two distinct files for each scenario, namely one where the interviewee is the sole developer of the file, and one where the sole developer is another contributor. An overview of the sampled files is reported in Figure 3. During the interviews, source code of the considered file is displayed *via* screen sharing to provide further context to the interviewees. The interviews conclude with an open-ended question inviting participants to share additional comment and reflections on the topic. The interviews are automatically transcribed and analyzed *via* reflexive thematic analysis [13] with a round of initial coding to identify meaningful incidents, a round of open coding, and subsequent axial coding round. To provide additional depth to the results, sub-themes are created for some incidents through an additional round of axial coding. The final output of this phase is a set of coded incidents, grouped into themes and sub-themes.

4.3 Main Case Study Findings

In this section, we provide a brief overview of the main findings of our case study, while leaving further details, their discussion, and implications to the next section (Section 5).

Regarding the quantitative results (Phase 1, see Section 4.2.1), an overview of the distribution of case study object Java files according to their CO_{dev_i} and *CogC* is reported in Figure 3. We observe that CO_{dev_i} shows strong variation (min=0, median=22.58, mean=36.56, max=100, σ=35.59), indicating that some files are monopolized by a single contributor, while others are heavily co-edited. The distribution of *CogC* instead (min=0, median=0, mean=2.95, max=92, σ=7.11) indicates that, while most files are easy to understand, a small fraction of files presents exceptionally high values, *e.g.*, 89 files (1.13% of the total) possess *CogC* $\geq$30.

Regarding the qualitative results (Phase 2, see Section 4.2.2), an overview of the interview codes resulting from the thematic analysis is reported in Figure 4 and are further described in the following. Both participants reported that the TDD phenomenon is relevant for industrial practice. By quoting D_2: "*I think that [TDD] is a very interesting idea to have an eye on these pain points in your project. That could be really useful.*" Participants also confirmed their contribution role to the project, agreed on the TDD sampled file ranking presented during the interview, and in some instances dwelt into describing specifics of the project and its source code.

Interestingly, during the interview, both participants hinted to several concepts related to software metrics through which the pilot TDD measurement approach (see Section 3) could be improved. Among other topics, participants mentioned considering commit activity, dependencies of TDD files, and knowledge deprecation through time. Such improvement avenues are further discussed in the next section (Section 5).

[1] https://github.com/pmd/pmd. Accessed October 29, 2025.
[2] https://github.com/pmd/pmd/commit/450dfe6. Accessed October 29, 2025.
[3] As clarification, PMD serves a dual role in our study, as it is both the object of our case study and a tool used to perform part of our approach (*i.e.*, we run the PMD analysis on its own codebase).

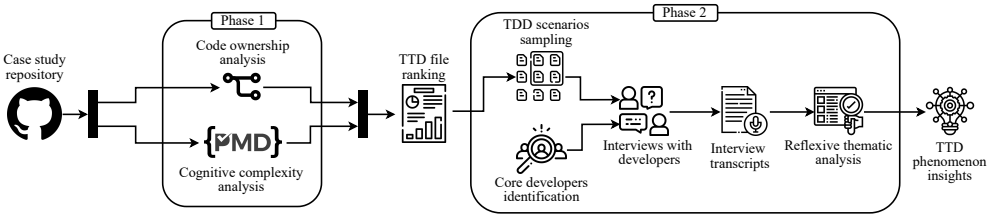

Figure 2: Case study Research Process

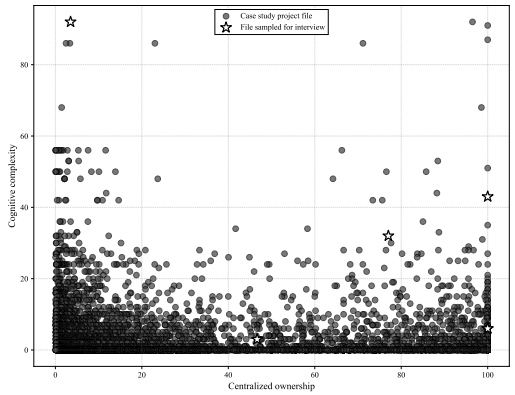

Figure 3: Cognitive complexity and centralized ownership distribution of case study files.

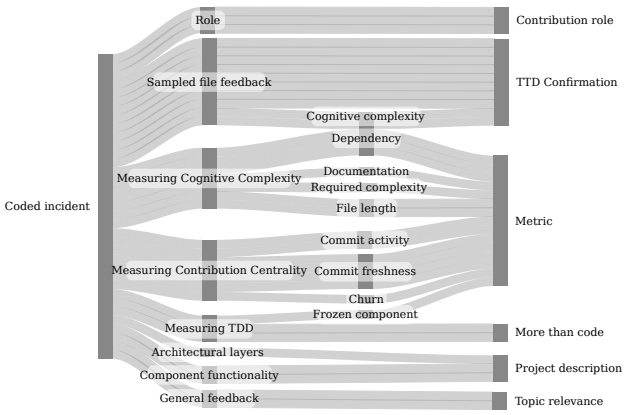

Figure 4: Overview of the interview codes

As final salient incident, one interviewee remarked that, other than source code, TDD might influence several other software artifacts. D_2 stated: *"There are other aspects difficult to be replaced by others that are not code. [Anonymized name] is doing all of the infrastructure, the CI, releases, the website, and stuff like that. I'm really glad that he's there".*

4.4 Threats to Validity and Limitations

Several threats could have influenced our results. By considering the most prominent ones [14], *CogC* and CO_{dev_i} are coarse approximations used to observe the considered TDD construct. Similarly, the small case study scale provides little guarantees regarding generalizability. Finally, while reviewed by another researcher, the reflexive thematic analysis is conducted by a single individual, hence raising the probability of introducing subjective bias in the coding process. Overall, the reported results need to be interpreted in light of the type of contribution of this paper, that is, to informally introduce a new concept that still needs to be more rigorously built upon.

5 Discussion

From the case study conducted to support our intuition, we learn that the TDD phenomenon is relevant for industrial practice. Also, the introduced theoretical framework of the two co-occurring properties identifying TDD instances (P_1 and P_2, see Section 2) appears to be viable.

Despite this optimistic outlook, building on our new idea, the case study pointed to a wide range of improvement opportunities for the TDD measurement approach of Section 3.

Regarding centralized ownership (P_1), the "freshness of commits", *i.e.*, the time elapsed from the commits, should be factored in the measurement of P_1. This is crucial to reflect the current knowledge that contributors possess about a file. Quoting D_2: *"After four weeks you can assume the original author doesn't know the code anymore"*

> **Measuring centralized ownership requires considering commit freshness.** Attributing software artifact knowledge to developers needs to account for the natural degradation of knowledge due to the passing of time.

Of a more semantic and complex nature, the activity of the commits should also be accounted for, *e.g.*, automated refactoring activities might improperly attribute the knowledge of a file to the developer conducting it. D_2 explained: *"At some point we changed the code style, so if you only look at who is the committer of a file that's not enough".*

> **Not all commits are equal.** To precisely identify the developers most knowledgeable about a software artifact, the activity of commits needs to be accounted for.

Regarding measuring low understandability and suboptimal documentation (P_2), our approach disregards code comments and their quality. These facets could nevertheless be crucial to support understandability, *e.g.*, a source file with high *CogC* might be very

well documented through comments, and hence be trivial to be understood. Similarly, other documentation artifacts, *e.g.*, README and CONTRIBUTING files, could ease familiarization with code incorrectly identified as hard to understand by using *CogC* alone.

> **Multiple factors influence understandability.** Documentation files, code comments, and their quality, need to be considered when measuring code understandability.

As pointed out by both participants, low understandability cannot be measured by considering a file in isolation, as P_2 could be inherited from other source files it depends on. As stated by D_1: *"It [understandability] is also about the context, what files the file depends on and their complexity"*

> **Low understandability can be inherited.** Understandability cannot be measured in isolation, as it could be inherited from the artifacts a software component depends on.

Finally, the viewpoint of TTD presented in this research is primary of a source-code-centric nature. However, as pointed out by an interviewee (see Section 4.3), heterogeneous software-centric artifacts of a project such as the continuous integration processes can be influenced by the phenomenon.

> **Turnover technical debt affects more than just source code.** Turnover technical debt can affect heterogeneous artifacts across the DevOps pipeline, *e.g.*, release management, continuous integration practices, testing strategies, and issue management.

6 Related Work

To the best of our knowledge, the TTD phenomenon has not yet been directly addressed in the existing body of literature. The concept of "knowledge debt", defined by Behutiye *et al.* as "missing knowledge or inadequate (not up-to-date) documentation" [15], appears closely related to TTD. However, despite an extensive literature search, we find such concept only briefly mentioned, with no studies explicitly focusing or even just thoroughly defining it. Similarly, TTD relates closely to the "bus / truck factor", a metric measuring how resilient a project is to sudden engineer turnover [16–20]. TTD takes a different and potentially more encompassing viewpoint on perhaps the same phenomenon, by considering code authorship as only one of its two core constituents (see P_1 and P_2), leading to a considerable enrichment of the facets to be considered to fully understand the phenomenon (see Section 5). Widening our related literature focus, TTD shares deep connections with various concepts related to technical debt. At its root, TDD naturally emerges when social debt is present, *i.e.*, in "sub-optimal" development communities [21–25], *e.g.*, if knowledge concentration or uncooperative contributors are present. Similarly, TTD is more probable to appear in contexts where documentation debt is an established issue [26–30], as P_2 is partly dependent on it. In a more loose manner, certain facets of TDD resemble code technical debt [2], especially when considering measurement approaches based on complexity metrics [8, 31–34] and code ownership [35–38]. As however TDD can span more than just source code artifacts (see Section 5), we argue that an equal connection could made with other debt types, such as build [39–41], infrastructure [42–44], and architectural debt [45–47]. Overall, the TDD phenomenon presented in this research builds upon the existing literature by explicitly and exclusively focusing on TTD, a concept originating at the intersection of different existing areas of technical debt knowledge.

7 Conclusions and Future Work

In this research, we introduce the phenomenon of turnover technical debt. As a first step to study the phenomenon, we hypothesize two co-occurring properties that identify TTD, and outline a conceptual basis on how the phenomenon could be measured in practice. With a complementary case study, we uncover different facets that need to be considered to effectively observe TDD, such as commit freshness and activity, low understandability inheritance, and considering also artifacts other than source code. As future work, we plan to further our understanding of the phenomenon *via* a qualitative research, and subsequently port the findings to design a more precise approach to effectively measure TDD quantitatively.

We share this research in the hope it can spark a discourse within the technical debt research community, with the final goal of joining forces to shed light on the phenomenon, understand how it manifests itself, how it can be measured, and ultimately how it can be managed in practice.

Replication Package

The source code and data of Phase 1 is available in our replication package at: https://figshare.com/s/6447a47b52d8fd808e5f. As common with qualitative research, the the raw interview transcripts cannot be released to protect participants from disclosure of sensitive information.

Acknowledgments

We express our sincere gratitude to Clément Fournier and Andreas Dangel for participating in our study and for their significant contribution to the PMD tool, without both of which this study would not have been possible.

References

[1] R. Verdecchia, P. Kruchten, P. Lago, and I. Malavolta, "Building and evaluating a theory of architectural technical debt in software-intensive systems," *Journal of Systems and Software*, vol. 176, p. 110925, 2021.
[2] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of systems and software*, vol. 101, pp. 193–220, 2015.
[3] N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman, "The most common causes and effects of technical debt: first results from a global family of industrial surveys," in *International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10, 2018.
[4] A. Martini, J. Bosch, and M. Chaudron, "Architecture technical debt: Understanding causes and a qualitative model," in *Conference on Software Engineering and Advanced Applications*, pp. 85–92, IEEE, 2014.
[5] C. Jaspan and C. Green, "Defining, measuring, and managing technical debt," *IEEE Software*, vol. 40, no. 03, pp. 15–19, 2023.
[6] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *PJoint meeting on foundations of software engineering*, pp. 50–60, 2015.
[7] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*, vol. 82, pp. 25–46, Elsevier, 2011.
[8] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, *et al.*, "An overview and comparison of technical debt measurement tools," *Ieee software*, vol. 38, no. 3, pp. 61–71, 2020.

[9] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 24–31, IEEE, 1998.

[10] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *International Conference on Technical Debt*, pp. 57–58, 2018.

[11] M. Muñoz Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in *ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pp. 1–12, 2020.

[12] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[13] V. Braun and V. Clarke, "Thematic analysis: A practical guide," pp. 36–54, 2021.

[14] R. Verdecchia, E. Engström, P. Lago, P. Runeson, and Q. Song, "Threats to validity in software engineering research: A critical reflection," *Information and Software Technology*, vol. 164, p. 107329, 2023.

[15] W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun, "Analyzing the concept of technical debt in the context of agile software development: A systematic literature review," *Information and Software Technology*, vol. 82, pp. 139–158, 2017.

[16] E. Jabrayilzade, M. Evtikhiev, E. Tüzün, and V. Kovalenko, "Bus factor in practice," in *International Conference on Software Engineering: Software Engineering in Practice*, pp. 97–106, 2022.

[17] M. Ferreira, T. Mombach, M. T. Valente, and K. Ferreira, "Algorithms for estimating truck factors: a comparative study," *Software Quality Journal*, vol. 27, no. 4, pp. 1583–1617, 2019.

[18] V. Haratian, M. Evtikhiev, P. Derakhshanfar, E. Tüzün, and V. Kovalenko, "Bfsig: Leveraging file significance in bus factor estimation," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1926–1936, 2023.

[19] G. Avelino, M. T. Valente, and A. Hora, "What is the truck factor of popular github applications? a first assessment," 2017.

[20] F. Ricca, A. Marchetto, and M. Torchiano, "On the difficulty of computing the truck factor," in *International Conference on Product Focused Software Process Improvement*, pp. 337–351, Springer, 2011.

[21] D. A. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, "What is social debt in software engineering?," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 93–96, IEEE, 2013.

[22] H. Chen, R. Kazman, G. Catolino, M. Manca, D. A. Tamburri, and W. van den Heuvel, "An empirical study of social debt in open-source projects: Social drivers and the "known devil" community smell," in *Hawaii International Conference on System Sciences*, 2024.

[23] E. Caballero-Espinosa, J. C. Carver, and K. Stowers, "Community smells—the sources of social debt: A systematic literature review," *Information and Software Technology*, vol. 153, p. 107078, 2023.

[24] H. Saeeda, M. O. Ahamd, and T. Gustavsson, "A multivocal literature review on non-technical debt in software development: An insight into process, social, people, organizational, and culture debt," *e-Informatica Software Engineering Journal*, vol. 18, no. 1, p. 240101, 2024.

[25] H.-M. Chen, R. Kazman, G. Catolino, M. Manca, D. A. Tamburri, and W.-J. Van Den Heuvel, "An empirical study of social debt in open-source projects: Social drivers and the "known devil" community smell," 2024.

[26] T. S. Mendes, M. A. de F. Farias, M. Mendonça, H. F. Soares, M. Kalinowski, and R. O. Spínola, "Impacts of agile requirements documentation debt on software projects: a retrospective study," in *ACM symposium on applied computing*, pp. 1290–1295, 2016.

[27] H. F. Soares, N. S. Alves, T. S. Mendes, M. Mendonça, and R. O. Spinola, "Investigating the link between user stories and documentation debt on software projects," in *2015 12th International Conference on Information Technology-New Generations*, pp. 385–390, IEEE, 2015.

[28] N. Rios, L. Mendes, C. Cerdeiral, *et al.*, "Hearing the voice of software practitioners on causes, effects, and practices to deal with documentation debt," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pp. 55–70, Springer, 2020.

[29] L. Mendes, C. Cerdeiral, and G. Santos, "Documentation technical debt: A qualitative study in a software development organization," in *Brazilian Symposium on Software Engineering*, pp. 447–451, 2019.

[30] L. Silva, M. Unterkalmsteiner, and K. Wnuk, "Towards identifying and minimizing customer-facing documentation debt," in *International Conference on Technical Debt*, pp. 72–81, IEEE, 2023.

[31] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100, IEEE, 2012.

[32] F. A. Fontana, R. Roveda, and M. Zanoni, "Technical debt indexes provided by tools: A preliminary discussion," in *International workshop on managing technical debt (MTD)*, pp. 28–31, IEEE, 2016.

[33] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *International Conference on Evaluation and Assessment in Software Engineering*, pp. 42–47, 2013.

[34] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *International conference on software engineering*, vol. 2, pp. 179–188, IEEE, 2015.

[35] E. Zabardast, J. Gonzalez-Huerta, and B. Tanveer, "Ownership vs contribution: Investigating the alignment between ownership and contribution," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pp. 30–34, IEEE, 2022.

[36] R. Alfayez, P. Behnamghader, K. Srisopha, and B. Boehm, "An exploratory study on the influence of developers in technical debt," in *International conference on technical debt*, pp. 1–10, 2018.

[37] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *ACM SIGSOFT symposium and European conference on Foundations of software engineering*, pp. 4–14, 2011.

[38] F. Zampetti, G. Fucci, A. Serebrenik, and M. Di Penta, "Self-admitted technical debt practices: a comparison between industry and open-source," *Empirical Software Engineering*, vol. 26, no. 6, p. 131, 2021.

[39] Morgenthaler *et al.*, "Searching for build debt: Experiences managing technical debt at google," in *International workshop on managing technical debt*, pp. 1–6, IEEE, 2012.

[40] T. Xiao, D. Wang, S. McIntosh, H. Hata, R. G. Kula, T. Ishio, and K. Matsumoto, "Characterizing and mitigating self-admitted technical debt in build systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4214–4228, 2021.

[41] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *Journal of Systems and Software*, vol. 171, p. 110827, 2021.

[42] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," in *2013 4th International Workshop on Managing Technical Debt (MTD)*, pp. 8–15, IEEE, 2013.

[43] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *International Workshop on Managing Technical Debt*, pp. 1–7, IEEE, 2014.

[44] E. Gama, S. Freire, M. Mendonça, R. O. Spínola, M. Paixao, and M. I. Cortés, "Using stack overflow to assess technical debt identification on software projects," in *Brazilian Symposium on Software Engineering*, pp. 730–739, 2020.

[45] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *International conference on software engineering*, pp. 488–498, 2016.

[46] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: The research landscape," in *Proceedings of the 2018 International Conference on Technical Debt*, pp. 11–20, 2018.

[47] R. Roveda, F. A. Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *Euromicro Conference on Software Engineering and Advanced Applications*, pp. 408–416, IEEE, 2018.