www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 16(4), Sep-Dec 2025, 288-302. ISSN: 2007-1558. https://doi.org/10.61467/2007.1558.2025.v16i4.1150

A Systematic Literature Review on Technical Debt in Software Development: Types, Tools, and Concerns

Victor Terrón-Macias[1], *Jezreel Mejía*[2*], *Miguel Terrón*[3]
[1] Departamento de Ciencias Computacionales, Tecnologico de Monterrey
[2] Centro de Investigación en Matemáticas A.C. Unidad Zacatecas
[3] Departamento de Ingeniería en Mantenimiento Industrial, Universidad Tecnológica de Tlaxcala
A00843371@tec.mx, jmejia@cimat.mx*, miguel.terron@uttlaxcala.edu.mx

Abstract. Software projects often accumulate technical debt, which undermines code quality, maintainability, and long-term viability. This article analyses twelve categories of technical debt and examines the capabilities of existing mitigation tools. To achieve this, we conducted a systematic literature review. Based on the results, we categorise and describe each debt type as it occurs in practice, and we assess prominent tools (such as AnaConDebt, CAST, DebtFlag, Visminer TD, and TD-Tracker) with respect to the debt types they target, the programming languages they support, and their adherence to established software quality models or methodologies. Our analysis shows that most current solutions focus on code and architectural debt, whereas documentation and process debt remain largely unaddressed. Moreover, most tools do not conform to software quality models or standards, which none of them explicitly incorporate.
Keywords: Capability maturity model integration-dev, concerns, scrum, systematic literature review, technical debt types, tools.

Article Info
Received February 25, 2025
Accepted July 6, 2025

1 Introduction

Technical debt has become a significant issue in the software development industry, potentially leading to long-term negative consequences. This debt accumulates when suboptimal solutions, often taken for short-term gains, compromise software quality, resulting in higher maintenance costs, decreased performance, and inefficiencies in development cycles, among other issues. As the complexity of software systems continues to grow, managing technical debt becomes more challenging for organizations, often hindering their ability to maintain high-quality software processes. Many authors have documented this issue, highlighting the damaging effects of technical debt on performance (Curtis et al., 2012), cost (Rachow & Riebisch, 2022), and documentation (Vidoni & Cunico, 2022).

Despite advancements in software engineering practices, many organizations continue to struggle with effectively identifying and mitigating technical debt. Most existing tools offer only partial support. They typically address a single category of technical debt, and only a few provide capabilities for both identifying and mitigating it. Even fewer reference established standards when recommending remediation strategies, and many are tied to a specific programming language. For instance, CAST is focused on the identification of architectural, code, and defect debt; Among the tools surveyed, only CAST explicitly cites a formal use of standards — ISO/IEC 5055 — for evaluating technical debt (Nikolov, 2021); the rest make no mention of models or methodologies, such as CMMI-Dev, which codifies best practices for software quality (Software Engineering Institute, 2010). These limitations highlight the need for a new, standards-aligned approach that leverages modern technologies to assess and reduce technical debt in architecture, code, and documentation.

Managing TD systematically, therefore, requires robust methods and recognized frameworks to ensure consistent detection, assessment, and mitigation across projects. Aligning Technical Debt management with standards, models, or methodologies such as ISO/IEC 29110, CMMI-Dev, or Scrum would impose discipline, improve maintainability, and contain life-cycle costs, helping to mitigate technical debt because provide structured practices that can significantly reduce or mitigate technical debt by enhancing development processes and product quality. These models or methodologies approach the technical debt problem

from complementary angles: CMMI-DEV introduces organization-wide process improvements to prevent debt at its source. At the same time, Scrum instills agile practices to continually address and limit debt during iterative development (Majka, 2024).

Therefore, the present article conducts a systematic literature review that pursues five objectives (1) present the technical debt types; (2) analyze the concerns of specific roles in a software development team into technical debt-as defined in current software models such as CMMI or methodologies such as Scrum-shape debt-management activities; (3) survey the current tools used to manage technical debt; (4) identify which of these tools explicitly align with models, methodologies or standards; and (5) in the discussion section, expose the significant limitations and research gaps uncovered as a results of the analysis of the four research questions.

2 Background

Ward Cunningham introduced the *technical debt* metaphor in 1992, borrowing a concept from finance to describe the future cost of shortcuts taken to satisfy short-term delivery pressures. For example, such as *blob* classes, tangled spaghetti code, or any undisciplined practice that erodes the intended architecture, leaves the system more change- and fault-prone (Fontana et al., 2017; Asif et al., 2019).

Although deliberately incurring a limited amount of debt can be a rational strategy when time-to-market outweighs long-term considerations, unchecked accumulation inflates maintenance effort, degrades performance, and diminishes overall software quality (Saraiva et al., 2022). Because some degree of debt is inevitable as systems evolve, the core challenge is not its total avoidance but its disciplined management—deciding when to borrow, how much, and how quickly to repay—to sustain high-quality, maintainable software over the product's lifetime.

Technical debt has been acquiring significant importance due to the need to maintain the existing software systems. The Consortium for Information & Software Quality (CISQ) estimates that the cost of poor software quality in the US in 2022 is USD 2.41 trillion (Consortium for Information & Software Quality, 2022). Also, the growing impact of technical debt is an obstacle to making any change to the existing code, for it is estimated that by 2025, at least 40% of the IT budget will be spent to maintain technical debt, that is, to mitigate it, which is one of the main reasons that many modernizations of projects fail. Finally, it is estimated that 33% of the weekly hours for an average developer will be for addressing technical debt issues, that is, reducing them.

The prevalence and impact of technical debt in the software industry are alarming. According to a 2018 survey conducted by Stripe, software developers spend approximately 33% of their time addressing technical debt and maintenance issues. This percentage translates to a staggering USD 85 billion in lost developer productivity annually globally (STRIPE, 2018). Moreover, a study by OutSystems found that 60% of IT leaders consider technical debt a significant threat to innovation, and organizations with high levels of technical debt are 40% less likely to launch new products or services successfully (OutSystems, 2021). These statistics highlight the significant economic and productivity implications of technical debt, as presented in Fig. 1.

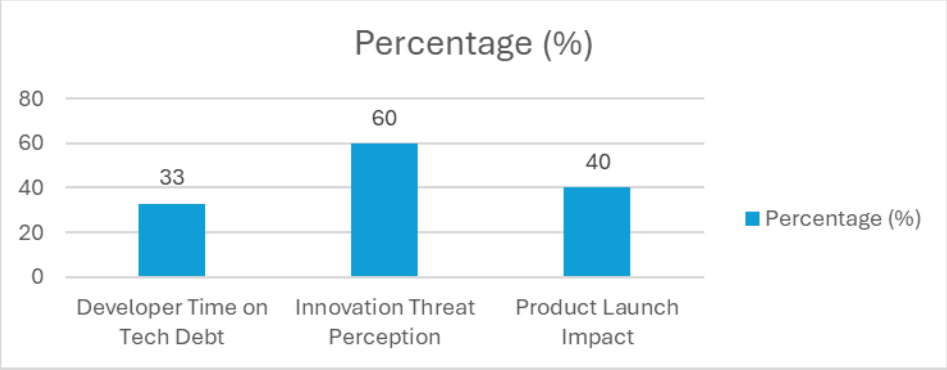

Fig. 1. Prevalence and industry impact metrics of technical debt.

Technical debt manifests in various forms across the software development lifecycle. Design debt affects approximately 25% of projects due to architectural inconsistencies (Yli-Huumo et al., 2016). Code debt is even more prevalent, with 62% of developers reporting that they work with *poorly written* code daily (Stack Overflow Developer Survey, 2021). Documentation

debt affects 48% of projects, leading to increased onboarding time and maintenance issues (Souto et al., 2016). Test debt is also significant, with 40% of organizations reporting inadequate test coverage (Capgemini, 2020), as illustrated in Fig. 2.

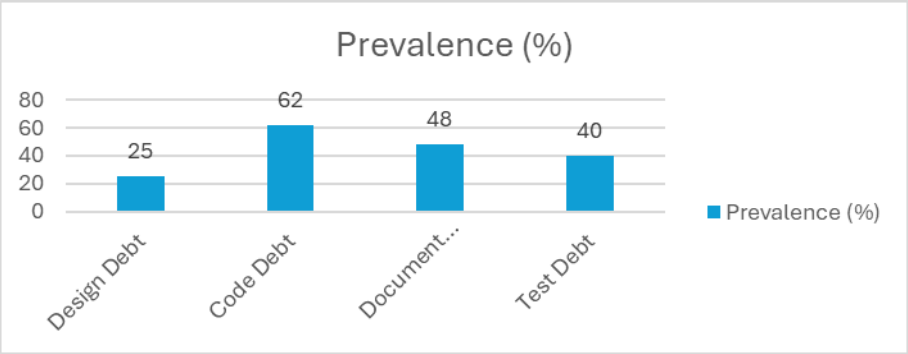

Fig. 2. Prevalence of the main categories of technical debt. Design debt affects approximately 25% of projects, code debt is encountered daily by 62% of developers, documentation debt impacts 48% of projects, and inadequate test coverage (test debt) is reported by 40% of organizations.

Looking forward, Gartner predicts that by the end of 2025, technical debt management will become a key strategic priority for 75% of CIOs (Gartner, 2024). Furthermore, effective technical debt management could unlock over $100 billion in global IT value by 2030 (APPIT, 2025). These projections underscore the increasing importance of addressing technical debt in the years to come. In this context, a set of activities can be incorporated into software development processes to identify, monitor, assess, and mitigate technical debt, which can be applied through the usage of models or methodologies (Avgeriou et al., 2016) and through the usage of technical debt management tools to address this issue.

In summary, technical debt is a multidimensional concept that affects various aspects of software development, including design, code, and documentation. The accumulation of technical debt can significantly impact the quality, maintainability, and scalability of software over time. The following section outlines our systematic review protocol, which mentions the search strategy, inclusion criteria, and data extraction procedures. Its goals are, first, to catalogue the types of technical debt, present the team roles' interest in technical debt management, identify the current tools used for managing technical debt, identify how well each tool aligns with formal standards, models, or methodologies, and finally, identify the tool's limitations.

3 Systematic Literature Review

This section presents the literature review stages, including the theoretical framework for technical debt, which encompasses its concept, types, identification, assessment, and reduction.

3.1 Systematic Literature Review Stages

A Systematic literature review is a process that examines existing literature on a specific topic to identify and interpret the evidence available in Primary Studies (PS), related to one or more research questions. In software engineering, literature reviews are essential for understanding how theoretical models, empirical studies, and tool implementations have addressed complex issues, such as technical debt. To guide this process for this research, we consider the approach indicated by Kitchenham et al., (2009). Kitchenham indicates that a Systematic Literature review has three main stages: Planning, Conducting the review, and Reporting the Review results, as shown in Fig. 3.

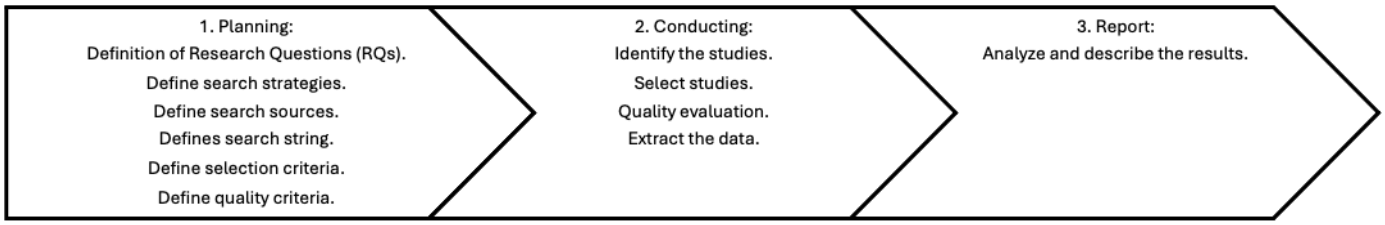

Fig. 3. Systematic Literature Review Stages.

Each stage is described below to clarify the process of the Systematic literature review.

Definition of Research Questions. The first stage involves establishing research questions to guide the research and evaluate the literature. For this research, the following research questions (RQs) were established (see Table 1):

Table 1. Research Questions

RQ1: What are the types of technical debt in software development?
RQ2: What are the concerns regarding technical debt among work team roles defined by Scrum and CMMI-Dev?
RQ3: What tools have been proposed to manage technical debt?
RQ4: What tools are aligned with software quality standards or process models (e.g., ISO/IEC 5055, CMMI-Dev) to identify, assess, and reduce technical debt according to each role?

These questions served as the foundation for designing the review strategy and determining the inclusion and exclusion criteria for the sources.

Sources and Search Strategy Design. Several search strings were established to ensure a comprehensive and unbiased collection of relevant studies, such as:

Technical Debt Types AND *Software* AND *Tools*
Technical Debt AND *Software* AND *Concerns* AND *Scrum* OR *CMMI Dev*
Models AND *standards* AND *Technical Debt Tools*

In each research string, we include the following restrictions: and LIMIT-TO (DOCTYPE, *CP*) OR LIMIT-TO (DOCTYPE, *AR*) and LIMIT-TO (LANGUAGE, *ENGLISH*)

Databases used included IEEE Xplore, ACM Digital Library, Scopus, SpringerLink, and Google Scholar. The search was restricted to peer-reviewed articles, conference proceedings, and relevant white papers published between 2019 and 2025, written in either English or Spanish, to ensure coverage of both foundational and recent work.

Selection Criteria. The selection process of the primary studies is aligned with the inclusion and exclusion criteria, which are described below (see Table 2).

Table 2. Selection Criteria

Inclusion Criteria	Articles that addressed one or more types of technical debt. Articles that describe tools, methods, or models for identifying or managing debt. Articles that discuss or are referenced to an alignment with standard, methodologies, or process models for mitigating technical debt (e.g., CMMI, ISO/IEC). Articles that addressed one or more concerns of technical debt.
Exclusion criteria	Those were applied to duplicate publications, non-English documents, or works lacking empirical evidence or technical depth.
Quality criteria	Are the objectives clearly defined? Was the research question appropriate to meet the research objectives? Is the research related to the identifying of the tools? Is the research related to the type(s) of technical debt?

Applying the search string, we initially identify 97 sources. After applying the inclusion, exclusion, and quality criteria, 42 studies were selected for a deeper analysis.

Data Extraction and Coding. From each selected PS, data were extracted and coded according to the following attributes (see Table 3).

Table 3. Attributes

Type(s) of technical debt addressed.
Tool or methodology proposed.
Reference to quality models or standards.
Empirical evidence of effectiveness.
Limitations identified.

This process allowed us to organize the literature thematically and identify patterns, redundancies, and gaps.

4. Report

This section reports the results found in the systematic literature review. The following subsections present the results from the analysis of the 42 Primary Studies (PS) related to the four established research questions.

4.1 Technical debt types in Software development

Based on the results of RQ1, the types of technical debt identified by different authors were categorized. Suryanarayana et al., (2014) indicates that technical debt has four main dimensions: code, design, test, and documentation. Moreover, several authors propose taxonomies or classifications of technical debt. Alves et al., (2016) propose a comprehensive ontology that identifies 13 types of technical debt, including architecture, code, build, and documentation debt, offering a structured vocabulary and conceptual map of the domain. Avgeriou et al., (2016) distinguish between various categories, including code, architecture, and even social debt, emphasizing the multidimensional nature of technical debt. Curtis et al., (2012) classify debt based on its impact, introducing categories like maintainability, performance, and reliability debt. After analyzing the work of several authors, a categorization of 12 types of technical debt was established, along with a corresponding description for each type. Fig. 4 shows our taxonomy.

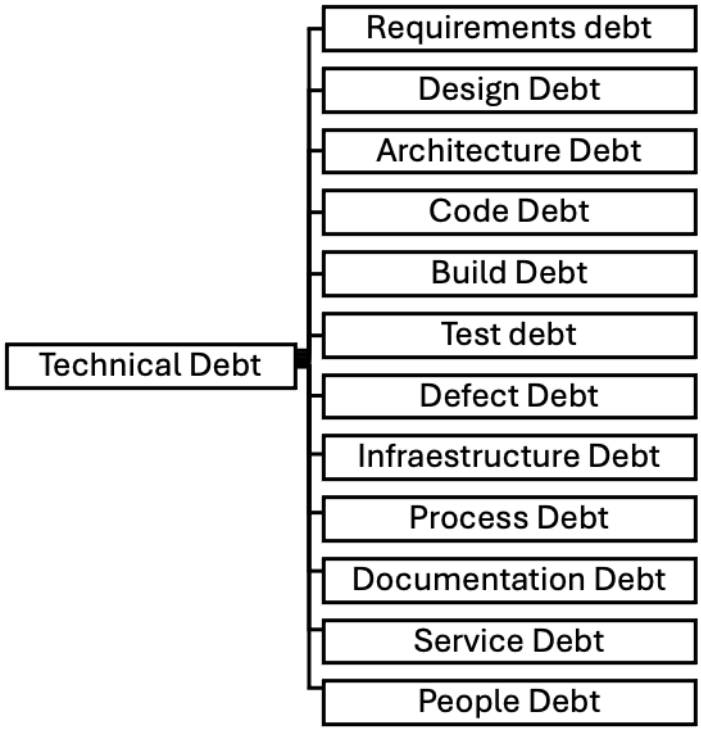

Fig. 4. Taxonomy of Technical Debt.

Technical debt of requirements occurs when specific aspects of the original specifications are deferred or neglected, often due to time or resource constraints, resulting in incomplete or partially implemented requirements. For example, edge cases may be overlooked when a system is only designed for standard input, leading to failures with unusual data. Similarly, if error handling is implemented only for common scenarios, unexpected errors may cause the system to crash. Inadequate validation and testing can result in untested code paths, allowing defects to go unnoticed. Non-functional requirements, such as performance and security, may also be postponed, leaving the system vulnerable to future issues as it scales. In some cases, integrations with external systems are partially completed, which can cause problems when the unfinished connections become necessary. Temporary workarounds, used to bypass incomplete functionality, often create future challenges when they need to be reworked. Additionally, unclear requirements may lead to incorrect assumptions during development, requiring costly adjustments later. Finally, a lack of proper documentation can hinder future development, making it more challenging for teams

to maintain or extend the system. These scenarios contribute to the accumulation of technical debt, necessitating the allocation of future resources to address it. It occurs at the beginning of the development cycle when the project requirements are defined (Alves et al., 2016).

Technical debt of design arises from design decisions that compromise long-term maintainability or flexibility. One typical example is the presence of design smells, where the architecture deviates from best practices. For instance, rigid architecture occurs when a system is designed so that even small changes require modifications across many components, making the system difficult to extend. Another scenario is excessive coupling, where modules or classes are too dependent on one another, making it hard to isolate or update individual components (Sorgalla et al., 2020). God classes, which are large classes handling multiple responsibilities, violate the single responsibility principle, making the system harder to understand and maintain (Guamán, et al., 2020). Overly complex hierarchies in object-oriented design can also introduce debt, where deep inheritance trees add unnecessary complexity, making debugging and modification harder. Additionally, premature optimization can lead to designs prioritizing efficiency over clarity or flexibility, which may not be necessary and often complicate future development. Each of these design smells highlights a deviation from core design principles, contributing to technical debt that will require future refactoring to resolve. These problematic structures often harm the overall quality of the design (Suryanarayana et al., 2014). A technical debt of design has another issue: the violation of design rules, which refers to instances where the code or architecture does not adhere to established design principles, guidelines, or best practices. A study conducted by Siemens reveals that up to 64% of software defects in enterprise software are related to technical debt in software design (SIEMENS, 2021), highlighting the importance of addressing technical debt in software design.

The technical debt of architecture refers to the violations of modularity, where system components become too tightly coupled, violating the principle of separation of concerns. Another architectural debt is building a system with a monolithic architecture, where all functionality is bundled into a single, large codebase rather than broken into independent modules or services (Alves et al., 2016).

The technical debt of code refers to suboptimal code that violates best coding practices or established rules, resulting in long-term issues with the software's maintainability and readability (Vidoni & Cunico, 2022). Code debt often results from shortcuts taken during development, such as rushing to meet deadlines or neglecting proper coding standards. Poorly written code can be more challenging to understand, debug, and modify, thereby increasing the complexity of future updates (DongGyun et al., 2022). Concrete examples of code debt include inconsistent naming conventions, which make it difficult for developers to understand the purpose of variables or functions, and a lack of proper documentation, leaving future developers to spend unnecessary time deciphering how a particular piece of code works. Another example is overly complex functions that perform multiple tasks, rather than breaking them down into smaller, more manageable pieces. This violates the single responsibility principle and makes the codebase harder to maintain and extend. This debt includes syntax errors, code smells, performance issues, security vulnerabilities, race conditions, memory leaks, overly complex code structures, infrastructure-as-code misconfigurations, and outdated practice (Suryanarayana et al., 2014). A technical debt of code has another issue: the inconsistent coding style, which refers to varied coding practices within a team, can result in a codebase that’s difficult to decipher and update efficiently (DongGyun et al., 2022). A study developed by Microsoft reveals that at least 30% of technical debt is related to code debt (Kim et al., 2014).

The *technical debt of the build* refers to inefficiencies in the build process that unnecessarily complicate or slow it down, often due to poorly managed dependencies or redundant code (Alves et al., 2016). These inefficiencies increase the time and resources required for building and deploying a project, affecting productivity. For example, a build process that includes unnecessary dependencies—libraries or modules not critical to the project's functionality—can slow compilation times and consume valuable developer time. Another example is a building that includes excess code not needed by the end user, making the process more resource intensive. Addressing build debt often involves optimizing dependency management, removing unneeded components, and streamlining the build process to improve efficiency and reduce complexity.

According to Alves, Ribeiro, Caires, Mendes, and Spínola, test technical debt is divided into test automation debt and test debt. *Test automation debt* occurs in testing automation and reflects the lack of automated tests for previously developed functionality. It is crucial for continuous integration and rapid development. *Test debt* refers to issues in the quality of testing activities that can affect the thoroughness and accuracy of testing (Alves et al., 2016).

The *technical debt of defects* refers to the accumulation of known defects in a system that have been identified but postponed for future resolution due to competing priorities or limited resources (Alves et al., 2016). These defects are typically discovered through testing activities or reported by users via bug tracking systems. While the Change Control Board (CCB) acknowledges

that these issues should be resolved, other priorities often take precedence over their deferral. As these unresolved defects accumulate, they can significantly increase technical debt, making the system more challenging to maintain and potentially introducing further issues over time (Alves et al., 2016). For example, a minor performance bug identified in early testing might be deprioritized in favor of delivering new features. However, as the system evolves, fixing this bug later could become more complex due to additional dependencies or changes made to the codebase. Another example is a known user interface glitch that has been documented but not addressed, which could become more costly to resolve if it negatively impacts user experience or leads to future compatibility issues.

Infrastructure technical debt occurs when outdated or insufficient infrastructure hinders development activities, making it more challenging to maintain, scale, or efficiently support the software (Alves et al., 2016). This debt arises, when necessary, infrastructure upgrades, such as hardware improvements, server optimizations, or network enhancements, are delayed or ignored. These delays can slow down development processes, increase operational costs, and negatively impact system performance. For example, delaying server upgrades may result in slower application response times and reduced capacity to handle increased traffic, affecting user experience. Another example is postponing network or storage enhancements, which could limit the team's ability to efficiently manage large data sets, affecting testing, deployment, and overall development speed. As this infrastructure debt accumulates, it can become more costly and time-consuming to address, eventually slowing down the entire software lifecycle.

The *technical debt of the process* arises when the development process becomes inefficient or misaligned with the current needs of the team or project. This occurs when effective processes become outdated or unsuitable for the project's scale, complexity, or goals. As a result, these inefficiencies can slow development, reduce productivity, and create bottlenecks in the workflow (Alves et al., 2016). For example, a manual testing process that worked well for a smaller project might become inefficient as the project grows, leading to delays in identifying defects or preparing releases. Similarly, a rigid approval process for code changes, which once ensured quality, could become a hindrance, delaying integration and deployment as the team expands or project demands increase. Addressing process debt typically requires reevaluating and optimizing workflows to ensure they remain aligned with the project's evolving needs.

The *technical debt of documentation* refers to situations where code is either insufficiently or inaccurately documented, leading to challenges in understanding and maintaining the software. When proper documentation is lacking, developers may struggle to comprehend the purpose, structure, or behavior of the code, especially as time passes or as new team members join. This type of debt can lead to increased maintenance costs, slower onboarding for new developers, and a higher likelihood of introducing errors when modifying the codebase (Silva et al., 2023). For example, a system may have complex functions or modules that lack clear explanations or comments, forcing developers to spend additional time deciphering how the code works before they can make any necessary changes. Similarly, out-of-date or incorrect documentation can mislead developers, potentially leading to bugs or faulty implementations. Addressing documentation debt involves ensuring that code is annotated correctly, with up-to-date explanations that accurately reflect its current state, making future development and maintenance easier and more efficient (Aversano et al., 2020). Another problem with documentation debt is poor documentation, which indicates a lack of a documented process for software artifacts (Rios et al., 2020). Another problem with documentation debt is outdated documentation, which refers to existing documentation that is not current with the version of the artifact being documented (Suryanarayana et al., 2014).

The *technical debt of service* arises when web services need to be updated or replaced due to evolving technical or business objectives, potentially creating new forms of technical debt during the substitution process. This type of debt can occur when a service replacement introduces new inefficiencies or complications that must be addressed over time (Alves et al., 2016). For example, a web service might be substituted to meet new business requirements. However, the process could lead to compatibility issues with other system parts, requiring additional adjustments or workarounds. Another scenario is when a third-party service is replaced, but the integration introduces performance or security risks that were not previously present. The technical debt in-service replacement can involve various dimensions, such as selecting the appropriate service, the composition or integration with existing systems, and the operation of the new service over time. Properly managing this debt involves careful planning and execution to transform the change from a liability into a value-added improvement.

Technical debt of people occurs when the development team lacks sufficient expertise, often due to delayed training or hiring, which negatively impacts productivity and efficiency. This debt arises when critical knowledge or skills are concentrated in too few team members, creating bottlenecks in the development process. (Alves et al., 2016). For example, suppose only a small number of developers possess the necessary expertise to manage key parts of the system. In that case, their availability becomes a limiting factor, slowing down progress and increasing the risk of delays. Additionally, delayed hiring of skilled personnel or

insufficient training for existing team members can result in a lack of necessary knowledge, leading to inefficient problem-solving and slower cycles of development. As this debt accumulates, it can severely impact the team's ability to meet deadlines or adapt to changing project demands, requiring significant investment in training or recruitment to resolve the issue.

To address technical debt, a solution is to use models, standards, or methodologies. For instance, model CMMI-DEV is a process maturity model that guides organizations in implementing disciplined software engineering practices. By following the CMMI-DEV model, teams establish rigorous processes for requirements management, design reviews, quality assurance, and risk management – all of which help prevent the accumulation of technical debt. For example, CMMI-DEV emphasizes thorough documentation, adherence to coding standards, continuous verification, and process audits to detect and resolve issues early in the development lifecycle. These best practices mean that shortcuts or suboptimal solutions (which often create technical debt) are less likely to be introduced. Moreover, CMMI-DEV's focus on continuous process improvement and quantitative project management ensures that any inefficiencies or defects are systematically identified and addressed before they grow into larger maintenance problems (KPMG, 2015).

On the other hand, the Scrum methodology addresses technical debt at both the team and project levels by integrating quality control into every iteration. Scrum is an Agile framework that delivers software in short cycles (sprints), with a potentially shippable product increment at the end of each sprint. A key Scrum practice for debt mitigation is maintaining a strict Definition of Done for each backlog item. In Scrum, a feature is not considered *done* until it is fully coded, tested, and integrated – in other words, ready for release. This is presented in the following subsection (Majka, 2024).
For this, it is necessary to understand the interests of each role and activity within the work team roles defined by the CMMI-Model and Scrum methodology.

4.2 Concerns Regarding Technical Debt among Work Team Roles Defined by Scrum and CMMI-Dev

This section presents a description of the CMMI-Dev model and Scrum roles, along with their relevance as outlined in the literature. CMMI-DEV and Scrum were selected for this research because they are widely recognized models that approach software process management and delivery from complementary and practical perspectives in addressing technical debt.

CMMI-Dev Model. CMMI-DEV (Capability Maturity Model Integration for Development) is a process maturity model adopted by organizations in over 70 countries to improve the quality and predictability of software development (SEI, 2010). Its structured focus on continuous improvement, along with emphasis on practices such as requirements management, design reviews, quality assurance, and risk management, helps prevent the accumulation of technical debt through rigorous process standardization and documentation. Capability Maturity Model Integration for Development (CMMI-Dev) is a model developed by the Software Engineering Institute (Chrissis et al., 2011). This model applies to any software (Viera-Bautista, & Sánchez-Gordón, 2019) and enables the establishment of processes that provide the infrastructure and stability necessary to handle changes, maximize productivity, and remain competitive in the market. Its 1.3 version contains 22 process areas focused on the developer organization's activities. As indicated earlier, this model is designed for developing products and services and comprises process areas; each is composed of a purpose statement, introductory notes, related process areas, specific goals, generic goals, specific practices, generic practices, sub-practices, and finally, work products (Khraiwesh, 2012).

In the case of CMMI-Dev there are no roles defined, nevertheless it is defined specific responsibilities that must be complied with by different actors in the organization that vary according to the process area, for instance: Requirements Management tasks usually fall to a Requirements Analyst or Product Owner; Project Planning and Project Monitoring & Control are driven by the Project Manager; Measurement & Analysis is led by a Metrics Analyst or PMO specialist. Other potential benefits of using CMMI are the reduction in the cost of quality from over 45% to under 30% over three years, improved customer satisfaction by an average of 42%, improved on-time delivery from 50% to 85%, improved account productivity in 20%, according to data of the KPMG (KPMG, 2015).

Scrum. Scrum has been adopted by more than 85% of Agile teams, according to the 14th State of Agile Report (VersionOne, 2020). It emphasizes incremental delivery, short development cycles (sprints), and a strict Definition of Done—all of which enable the early detection and correction of technical issues. Furthermore, Scrum fosters shared responsibility and continuous improvement through its key roles (Product Owner, Scrum Master, and Developers), who are directly involved in maintaining product quality and minimizing technical debt in every iteration. By analyzing both frameworks together, it is possible to gain a comprehensive understanding of how different roles and practices contribute to mitigating technical debt from both a structured (CMMI-DEV) and Agile (Scrum) perspective. Scrum is an agile framework used for managing and developing complex software projects. It emphasizes iterative progress, collaboration, and flexibility in response to changing requirements. Work is

structured into short cycles called sprints, typically lasting two to four weeks, during which a potentially shipping product increment is delivered.

The Scrum methodology revolves around three primary roles. The Product Owner is responsible for defining the product vision and managing the backlog to ensure the team builds the right product. The Scrum Master acts as a facilitator, helping the team follow Scrum principles, removing obstacles, and promoting continuous improvement. The Development Team is a cross-functional group that plans, builds, tests, and delivers the product increment during each sprint. Scrum also includes ceremonies such as sprint planning, daily stand-ups, sprint reviews, and retrospectives to support communication and continuous feedback.

Other potential benefits of adopting Scrum include a 67% acceleration in delivery speed, a 61% uplift in product quality—with some large-scale benchmarks reporting up to 250% fewer defects when teams follow full-Scrum practices—plus a 45% rise in customer satisfaction and 300%–400% gains in team productivity once the framework is embedded. The impediments backlog is actively burned down. These figures come from multi-year industry datasets compiled by Digital.ai (State of Agile 2024), CA Technologies' Impact of Agile study, and Jeff Sutherland's longitudinal research on hyper-productive Scrum teams.

Roles, Responsibilities, and Technical Debt Concerns: According to CMMI-DEV, the core roles are tied to seven Process Areas: Requirements Management, Project Planning, Project Monitoring and Control, Supplier Agreement Management, Measurement and Analysis, Process and Product Quality Assurance, and Configuration Management. The Table 4 includes the CMM-Dev L2 roles and Table 5 includes the SCRUM roles.

Table 4. Roles, responsibilities, and technical debt interest in CMMI-DEV L2 Model

Role	Responsibilities	Technical debt concerns
Project Manager	Planning, monitoring, and controlling the project; managing risks and resources.	Process Debt, Documentation Debt, and Infrastructure Debt: Concerned with how process inefficiencies and missing documentation affect project timelines and costs. Poor infrastructure planning can lead to delays and increased risk (Szczepańska-Woszczyna & Gatnar, 2022).
Requirements Engineer / Requirements Manager	Eliciting, documenting, managing, and tracking requirements.	Requirements Debt and Documentation Debt: Incomplete or unclear requirements lead to misunderstandings, changes, and rework. Poor documentation increases the risk of requirements changes (Kasauli et al., 2021).
Configuration Manager	Maintain artifact integrity, managing version control and configuration baselines.	Configuration Debt, Process Debt, and Built Debt: Issues with configuration management lead to version inconsistencies and an increased risk of integration problems. Inefficient build processes can also add to project delays (Sandrin et al., 2022).
Quality Assurance Analyst / Quality Manager	Ensure quality through testing, reviews, and audits; identify defects.	Test Debt, Defect Debt, and Documentation Debt: Insufficient test coverage and unresolved defects increase risk. Lack of documentation can hinder effective testing and quality assurance (Govob & Zuieva, 2023).
Measurement Analyst / Process Analyst	Collect and analyze metrics related to performance and process improvements.	Process Debt and Measurement Debt: Inadequate metrics or analysis processes lead to poor decision-making. Gaps in measurement make it challenging to accurately assess the impact of technical debt (Fauzi & Andreswari, 2022).
Supplier Manager	Manage contracts and relationships with external suppliers/vendors.	Service Debt and Infrastructure Debt: Poor-quality or non-compliant deliverables from suppliers increase technical debt, especially when services are unreliable, or infrastructure does not meet project needs (Fauzi & Andreswari, 2022).

Table 5. Roles, responsibilities, and technical debt interest in Scrum Methodology

Role	Responsibilities	Technical debt concerns
Product Owner	Define product vision, manage the Product Backlog, and prioritize features.	Requirements debt, people debt, and code debt: Lack of stakeholder alignment leads to unclear requirements. Prioritization that overlooks technical debt can lead to its accumulation, particularly in code quality (Cristina, 2022).
Scrum Master	Facilitating Scrum events, removing impediments, and coaching the team.	Process Debt, People Debt, Test Debt: Inefficient Scrum processes or a lack of team skills can contribute to these types of debt. Guides the team in managing technical debt and maintaining good testing practices (Majka, 2024).
Development Team	Designing, coding, testing, and delivering product increments.	Code Debt, Design Debt, Architecture Debt, Test Debt: Makes trade-offs between quick fixes and sustainable solutions. Design flaws, poor architecture, or lack of refactoring contribute to debt. Inadequate testing adds risk (Majka, 2024).

4.3 What tools have been proposed to manage technical debt?

As a result, regarding RQ3, thirteen tools have been identified for managing technical debt, as shown in Table 5**Error! Reference source not found.**, reveals several critical limitations in addressing these issues.

1. AnaconDebt, designed to detect both code and architectural debt (Martini, 2018).
2. CAST AIP identifies architecture, code, and defect debt, offering compatibility with eleven programming languages (Lenarduzzi et al., 2021).
3. CodeScene focuses on design and code debt, also supporting eleven languages (Lenarduzzi et al., 2021).
4. DebtFlag is restricted to detecting code debt and only supports Java (Amanatidis et al., 2020).
5. DebtGrep specializes in identifying debt related to people, design, architecture, and programming language agnostic (Amanatidis et al., 2020).
6. DV8 targets architecture and code debt, supporting eight languages (Amanatidis et al., 2020).
7. Kiuwan is focused on software code quality, detecting only code debt, with support for eleven languages (Guamán et al., 2020).
8. NDepend can identify design, architecture, code, and test debt (Guamán et al., 2020).
9. SonarQube covers design, architecture, code, test, and defect debt, and supports twelve programming languages (Lenarduzzi et al., 2021).
10. Sqore detects code and test debt and is compatible with twelve languages (Guamán et al., 2020).
11. TD-Tracker stands out by detecting design, documentation, code, test, defect, and infrastructure debt, supporting six languages (Borante Foganholi et al., 2015).
12. TEDMA focuses on architectural and code debt but is limited to Java (Fernández et al., 2017).
13. VisminerTD offers the most comprehensive coverage, identifying requirements, people, design, architecture, documentation, code, build, test, and defect debt. However, it also only supports Java (Martini, 2018).

Table 5. Technical Debt Coverage of Tools

#	Tool's Name	Requirements Debt?	Process	People Debt?	Design Debt?	Architecture Debt?	Documentation debt?	Service Debt?	Code Debt?	Built Debt?	Test Debt?	Defect Debt?	Infrastructure Debt?
1	AnaConDebt	○	○	○	○	●	○	○	●	○	○	○	○
2	CAST AIP	○	○	○	○	●	○	○	●	○	○	●	○
3	CodeScene	○	○	○	●	○	○	○	●	○	○	○	○
4	DebtFlag	○	○	○	○	○	○	○	●	○	○	○	○
5	Debtgrep	○	○	●	●	●	○	○	●	○	○	○	○
6	DV8 (SCITools)	○	○	○	○	●	○	○	●	○	○	○	○
7	Kiuwan	○	○	○	○	○	○	○	●	○	○	○	○
8	Ndepend	○	○	○	●	●	○	○	●	○	●	○	○
9	SonarQube	○	○	○	●	●	○	○	●	○	●	●	○
10	Sqore	○	○	○	○	○	○	○	●	○	●	○	○
11	TD-Tracker	○	○	○	●	○	●	○	●	○	●	●	●
12	TEDMA	○	○	○	○	●	○	○	●	○	○	○	○
13	VisminerTD	●	○	●	●	●	●	○	●	●	●	●	○

Reviewing the literature, it is evident that most tools focus primarily on code debt, which refers to the support provided by programming languages. Tools, such as DV8 (Tool 6) and Sqore (Tool 10), offer compatibility with a diverse range of programming languages, including Java, Python, SQL, and others. This makes them better suited for teams that work with varied technology stacks.

4.4 Tools Used to Manage Technical Debt and their alignment with models or methodologies and interests by role

Identification of technical debt refers to the process of recognizing and documenting instances where shortcuts, suboptimal design choices, or compromises were made during the software development process (Allman, 2012). For the identification of technical debt, different models, such as Software Quality Assessment based on Lifecycle Expectations (SQALE), the Checking Quality Model (CQM), or the ISO/IEC 5055 standard, are considered for estimating not only technical debt but also its interest or model Software Improvement Group model (SIG model) (Molnar & Motogna, 2022). Those tools utilize static code analysis to identify issues based on rules related to quality attributes (see Table 6).

Table 6. Technical Debt and its Alignment with Models, Standards, or Methodologies

	Tool	Explicitly mentions that integrates a Model Standard or Model?	User Oriented
1	AnaConDebt	○	Project Managers, Software Architects
2	CAST AIP	●	Project Managers, Developers, QA Engineers, Software Architects
3	CodeScene	●	Developers, DevOps Engineers, Project Managers
4	DebtFlag	○	Software Architects, Developers
5	Debtgrep	○	Developers, Software Architects
6	DV8 (SCITools)	○	Software Architects, Developers
7	Kiuwan	○	Developers, QA Engineers, Project Managers, Analysts
8	Ndepend	○	Developers, Software Architects
9	SonarQube	○	Developers, QA Engineers, Project Managers
10	Sqore	○	Developers, Project Managers
11	TD-Tracker	○	Project Managers, Software Architects
12	TEDMA	○	Software Architects, Developers
13	VisminerTD	○	Technical Leads, Software Architects, Developers

Reviewing the literature, the tools are often not aligned with: 1) established quality models or methodologies like CMMI-Dev, Scrum among others limiting their effectiveness in comprehensive process improvement; and 2) technical debt assessment, it

refers to evaluating the impact, scope, and risk associated with a software system; that allow to understand its severity, costs, and how it affects the software (Allman, 2012).

After reviewing the tools, only CAST and CodeScene align their metrics with ISO/IEC 5055 standard; in these tools, the aligned quality factors are: testability, reliability, changeability, efficiency, usability, security, maintainability, portability, and reusability; the final value is weighted by severity as can be seen in Table 6.

5 Discussion

The results of this study highlight the growing complexity of technical debt and underscore the need for context-aware, role-sensitive solutions. While technical debt is widely acknowledged as a pervasive problem in software development, many existing tools and methodologies offer only partial coverage, particularly regarding forms of debt beyond code, such as documentation, design, and process debt.

A key observation from the results of the Systematic literature review is the disproportionate focus on code-related technical debt. Although code debt is a critical dimension, this emphasis often overshadows other equally significant types, such as documentation processes and design debt, which have clear implications for maintainability and long-term project sustainability. Documentation debt, in particular, remains one of the most underrepresented categories, despite its recognized impact on developer onboarding, knowledge transfer, and defect mitigation.

Based on the identified roles within CMMI-DEV and Scrum—each with distinct responsibilities and technical debt concerns—it becomes evident that the available tools offer varying degrees of coverage aligned with role-specific needs. Tools like SonarQube and NDepend provide robust support for code, design, test, and architecture debt, aligning well with the needs of developers, Scrum Masters, and QA roles. In contrast, TD-Tracker and VisminerTD stand out for their broader coverage, addressing not only code-related debts but also documentation, infrastructure, and even people and requirements debt. These tools are particularly valuable for roles such as project managers, requirements engineers, and configuration managers.

However, there are notable and persistent gaps. Process debt, which is critical for project managers, process analysts, and Scrum Masters, is vastly underrepresented. Similarly, measurement debt and service debt, important for analysts and supplier managers, are inadequately addressed. While DebtGrep makes a meaningful contribution to identifying design debt, it lacks coverage for documentation or service-related concerns. Among all the tools evaluated, VisminerTD emerges as the most complete solution, supporting a wide range of technical debt types and addressing the needs of various roles. In contrast, tools like Kiuwan and DebtFlag offer only partial support and may be more appropriate in narrowly scoped environments due to limited detection capabilities and role alignment.

This uneven distribution of coverage reflects a broader disconnect between most tools and formal software quality models or process improvement frameworks. Except for tools like CAST, few integrate standards such as ISO/IEC 5055 or adopt practices from structured models like CMMI-Dev. This represents a missed opportunity to standardize technical debt management and to ground mitigation strategies in proven quality assurance principles. Tools that fail to incorporate these models risk limiting their usefulness in structured or process-intensive environments.

Furthermore, the analysis reveals how most tools tend to be developed with developers as their primary audience, often overlooking the perspectives and responsibilities of roles defined in methodologies such as Scrum or frameworks like CMMI-Dev. This is a significant gap, as technical debt is not merely a technical artifact but a reflection of systemic practices across project management, requirements engineering, quality assurance, and configuration management. Addressing technical debt holistically thus requires tools that support the concerns of diverse team members, not just developers.

Another challenge is the limited language and environment support found in many tools. Several are tightly coupled to Java ecosystems, making them less applicable in organizations that use polyglot environments, such as Python, C#, or JavaScript. In addition, the assessment criteria used by most tools lack standardization. Many fail to communicate the severity or business impact of the detected debt, which hinders prioritization and planning.

Taken together, these findings suggest that managing technical debt effectively requires more than technical analysis. It demands a structured, organization-wide strategy aligned with well-defined roles, quality standards, and process models. Frameworks such as CMMI-Dev and Scrum offer valuable guidance by emphasizing proactive planning, role-based responsibilities, risk management, and performance metrics—all essential to comprehensive debt mitigation.

Finally, although many tools claim to assess technical debt, their evaluation mechanisms often lack clear criteria or severity metrics that can guide decision-making, making it difficult for organizations to prioritize debt reduction tasks.

6 Conclusions

Technical debt remains a significant challenge to software sustainability, manifesting prominently in the areas of design, code, process, and documentation. This systematic literature review highlights several critical gaps in current technical debt mitigation approaches.

Tool Limitations. Existing tools, such as AnaConDebt and DebtFlag, predominantly focus on isolated dimensions of technical debt, particularly code debt, and fail to provide comprehensive, multi-dimensional analyses. CAST and CodeScene are integrating the ISO/IEC 5055 standard, whereas most other tools lack alignment with robust standards or models, which limits their effectiveness across diverse software projects.

Neglect of Process Debt. A notable oversight in current tools is the absence of support for managing process debt—inefficiencies in workflows and processes that exacerbate technical debt accumulation and negatively impact software quality and development efficiency.

Lack of Methodological Consistency. The identification and assessment of technical debt are fragmented, lacking standardized approaches, metrics, and models, which complicates consistent debt prioritization and effective repayment strategies.

To address these critical challenges effectively, this study proposes several strategic future directions.
Adoption of Standardized Metrics. Establish consistent use of recognized frameworks, such as ISO/IEC 5055, the SIG model, Scrum, or the CMMI-Dev model, across organizations to standardize debt quantification and facilitate benchmarking, leading to more informed and transparent debt management decisions.

Inclusion of Process Debt Management. Extend technical debt management tools to capture and analyze process debt explicitly, addressing workflow inefficiencies such as bottlenecks in manual testing, inefficient resource allocation, and other systemic issues. This broader scope will contribute significantly to overall process improvement and organizational efficiency.

Technical debt management must be perceived not merely as a technical issue but as a strategic organizational imperative. Embedding effective management practices within recognized models or methodologies, such as CMMI-Dev or Scrum, transforms technical debt into a strategic opportunity. As Gartner predicts, by 2025, organizations that strategically manage technical debt will distinguish themselves as industry leaders, leveraging technical debt management as a critical factor for long-term success and sustainability.

Future research and tool development should focus on closing these gaps. Emerging technologies, particularly artificial intelligence, and large language models (LLMs), present a promising avenue for innovation. These technologies can enhance automation, contextualization, and adaptability in technical debt management, enabling intelligent code analysis, real-time feedback, and alignment with formal frameworks. Role-aware recommendations, tailored to the specific concerns of development teams and organizational practices, could become a standard feature of next-generation tools. By aligning technical solutions with both team roles and process frameworks, future approaches can significantly improve software quality and team productivity.

References

Allman, E. (2012). Managing Technical Debt. Communications of the ACM, 55(5), 50–55. https://doi.org/10.1145/2160718.2160733.

Alves, N., Ribeiro, L., Caires, V., Mendes, T., & Spínola, R. (2016). Towards an Ontology of Terms on Technical Debt. IEEE International Workshop on Managing Technical Debt, 1–7. https://doi.org/10.1109/MTD.2014.9.

Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., & Angelis, L. (2020). Evaluating the Agreement Among Technical Debt Measurement Tools: Building an Empirical Benchmark of Technical Debt Liabilities. Empirical Software Engineering, 1–44. https://doi.org/10.1007/s10664-020-09869-w.

APPIT. (2025). The 80/20 Rule of Application Development: Focus on What Truly Matters. Retrieved March 16, 2025, from https://appitventures.com/blog/80-20-the-universal-truth-about-software-development.

Asif, M., Ilyas, A., Sajjad, S., & Masood, A. (2019). Automatic Alert Generation against Pre-defined Rules-set for Perimetric Security of Sensitive Premises using YOLOv3. 2019 15th International Conference on Emerging Technologies (ICET), 1-6. doi:10.1109/ICET48972.2019.8994686.

Aversano, L., Carpenito, U., & Iammarino, M. (2020). An Empirical Study on the Evolution of Design Smells. Information, 11(7), 1–21. https://doi.org/10.3390/info11070348.

Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (2016). Managing Technical Debt in Software Engineering. Dagstuhl Seminar 16162, 110–138. https://doi.org/10.4230/DagRep.6.4.110.

Borante Foganholi, L., Garcia, R., Eler, D. M., Messias Correia, R. C., & Junior, C. O. (2015). Supporting Technical Debt Cataloging with TD-Tracker Tool. Advances in Software Engineering, Article 898514, 1–12. https://doi.org/10.1155/2015/898514.

Capgemini. (2020). World Quality Report. Retrieved January 16, 2024, from https://www.capgemini.com/es-es/wp-content/uploads/sites/16/2022/12/World-Quality-Report-2019-20.pdf.

Chrissis, M., Konrad, M., & Shrum, S. (2011). *CMMI for Development: Guidelines for Process Integration and Product Improvement* (3rd Edition ed.). United States of America: Carnegie Melon University. Retrieved February 02, 2025, from https://www.sei.cmu.edu/library/cmmi-for-development-guidelines-for-process-integration-and-product-improvement-third-edition/

Cristina, A. (2022). The Product Owner Role in a Contemporary Agile Team. Economic and Applied Informatics, 28(1), 78. https://doi.org/10.35219/eai15840409248.

Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the Principal of an Application's Technical Debt. IEEE Software, 29(6), 34–42. https://doi.org/10.1109/MS.2012.156.

DongGyun, H., Chaiyong, R., & Krink, J. (2022). Does Code Review Really Remove Coding Convention Violations? 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 43–53. https://doi.org/10.1109/SCAM51674.2020.00010.

Fauzi, R., & Andreswari, R. (2022). Business Process Analysis of Programmer Job Role in Software Development Using Process Mining. Procedia Computer Science, 197, 701–708. https://doi.org/10.1016/j.procs.2021.12.191.

Fernández, C., Humanes, H., Garbajosa, J., & Díaz, J. (2017). An Open Tool for Assisting in Technical Debt Management. Euromicro Conference on Software Engineering and Advanced Applications, 1–4. https://doi.org/10.1109/SEAA.2017.60.

Fontana, A., Dietrich, J., Bartos, W., Yamashita, A., & Zanoni, M. (March de 2016). Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 609-613. doi:10.1109/SANER.2016.84.

Gartner. (2024). Priorities CIOs Must Address in 2025, According to Gartner's CIO Survey. Retrieved January 18, 2025, from https://www.gartner.com/en/articles/priorities-cios-must-address-in-2025-according-to-gartner-s-cio-survey.

Govob, D., & Zuieva, O. (2023). Examining Software Quality Concept: Business Analysis Perspective. System Analysis and Decision-Making Theory, 9–14. https://doi.org/10.20998/2079-0023.2023.02.02.

Guamán, D., Pérez, J., Garbajosa, J., & Rodríguez, G. (2020). A Systematic-Oriented Process for Tool Selection: The Case of Green and Technical Debt Tools in Architecture Reconstruction. Product Focused Software Process Improvement, 237–253. https://doi.org/10.1007/978-3-030-64148-1_15.

Kasauli, R., Knauss, E., Horkoff, J., Liebel, G., & de Oleira Neto, F. (2021). Requirements Engineering Challenges and Practices in Large-Scale Agile System Development. Journal of Systems and Software, 172, Article 110851. https://doi.org/10.1016/j.jss.2020.110851.

Khraiwesh, M. (2012). Risk Management Measures in CMMI. International Journal of Software Engineering & Applications, 1–15. https://doi.org/10.5121/ijsea.2012.3111.

Kim, M., Zimmermann, T., Nagappan, N., & Zimmermann, T. (2014). An Empirical Study of Refactoring Challenges and Benefits at Microsoft. IEEE Transactions on Software Engineering, 1–19. https://doi.org/10.1109/TSE.2014.2318734.

Kitchenham, B. (2004). Procedures for Performing Systematic Reviews (Version 1.0). Keele University. https://www.inf.ufsc.br/~aldo.vw/kitchenham.pdf.

KPMG. (2015). CMMI and its Potential Benefits – Improve Performance. Retrieved September 19, 2024, from https://assets.kpmg.com/content/dam/kpmg/in/pdf/2016/08/CMMI-and-its-potential-benefits.pdf.

Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., & Fontana, F. (2021). A Systematic Literature Review on Technical Debt Prioritization: Strategies, Processes, Factors, and Tools. The Journal of Systems & Software, 171, Article 110827. https://doi.org/10.1016/j.jss.2020.110827.

Majka, M. (2024). Common Challenges for New Scrum Masters. Retrieved May 16, 2025, from https://www.researchgate.net/publication/386553317_Common_Challenges_for_New_Scrum_Masters.

Martini, A. (2018). Anacondebt: A Tool to Assess and Track Technical Debt. ACM/IEEE International Conference on Technical Debt, 1–2. https://doi.org/10.1145/3194164.3194185.

Molnar, A.-J., & Motogna, S. (2022). An Exploration of Technical Debt over the Lifetime of Open-Source Software. In H.

OutSystems. (2021). Retrieved January 8, 2024, from https://www.outsystems.com/news/study-reveals-technical-debt-is-threat-to-innovation/.

Rios, N., Mendes, L., Cerdeiral, C., Magalhães, A. P., Perez, B., Correal, D., & Spínola, R. O. (2020). Hearing the Voice of Software Practitioners on Causes, Effects, and Practices to Deal with Documentation Debt. In Requirements Engineering: Foundation for Software Quality (pp. 55–70). Springer. https://doi.org/10.1007/978-3-030-44429-7_4.

Saraiva, J., Gameleira, J., Kulesza, U., Freitas, G., Reboucas, R., & Coehlo, R. (08 de July de 2022). Exploring Technical Debt Tools: A Systematic Mapping Study. Enterprise Information Systems (ICEIS 2021), 280-303. doi:10.1007/978-3-031-08965-7_14.

Sandrin, E., Forza, C., Leitner, G., & Trentin, A. (2022). Configuration Manager: Describing an Emerging Professional Figure. ACM International Systems and Software Product Line Conference, B, 193–200. https://doi.org/10.1145/3503229.3547049.

SIEMENS. (2021). Refactoring For Design Smells 2021. SIEMENS Corporate Technology. Retrieved September 12, 2024.

Silva, L., Unterkalmsteiner, M., & Wnuk, K. (2023). Towards Identifying and Minimizing Customer-Facing Documentation Debt. 2023 ACM/IEEE International Conference on Technical Debt (TechDebt), 72–81. https://doi.org/10.1109/TechDebt59074.2023.00015.

Sorgalla, J., Sachweh, S., & Zündorf, A. (2020). Exploring the Microservice Development Process in Small and Medium-Sized Organizations. PROFES, 453–460. https://doi.org/10.1007/978-3-030-64148-1_28.

Souto, T., Farias, M., Mendoza, M., Frota, H., & Kalinowski, M. (2016). Impacts of Agile Requirements Documentation Debt on Software Projects: A Retrospective Study. 1290–1295. https://doi.org/10.1145/2851613.2851761.

STRIPE. (2018, September). The Developer Coefficient. Retrieved January 8, 2025, from https://stripe.com/files/reports/the-developer-coefficient.pdf.

Suryanarayana, G., Samarthyam G., & Sharma, T., (2014). Refactoring for Software Design Smells: Managing Technical Debt (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Szczepańska-Woszczyna, K., & Gatnar, S. (2022). Key Competences of Research and Development Project Managers in High Technology Sector. Forum Scientiae Oeconomia, 3, 107–130. https://www.ceeol.com/search/article-detail?id=1101347.

Vidoni, M., & Cunico, M. L. (2022). On Technical Debt in Mathematical Programming: An exploratory study. Mathematical Programming Computation, 14, 781–818. https://doi.org/10.1007/s12532-022-00225-1.

Viera-Bautista, D., & Sánchez-Gordón, S. P. (2019). Mapping between CMMI-DEV v1.3 and ISO/IEC 90003:2014: Process areas of "management of decisions and suppliers" and "creating a culture of excellence. International Conference on Information Systems and Software Technologies (ICI2ST), 134-140. doi:10.1109/ICI2ST.2019.00026.

Yli-Huumo, J., Maglyas, A., & Smolander, K. (2016). How do Software Development Teams Manage Technical Debt? An Empirical Study. Journal of Systems and Software, 120, 195–218. https://doi.org/10.1016/j.jss.2016.05.018.