

## Categorization of Technical Debt in Software as a Service Applications

Özden ÖZCAN TOP\*<sup>ID</sup>, Yasemin KURANEL\*<sup>ID</sup>, Altan KOÇYİĞİT\*<sup>ID</sup>

(Alınış / Received: 08.09.2023, Kabul / Accepted: 09.11.2023, Online Yayınlanma / Published Online: 30.12.2023)

### Keywords

Technical Debt,  
Software as a Service,  
Technical Debt in SaaS,  
SaaS

**Abstract:** Technical debt (TD) refers to the outcomes of making poor decisions or choosing simpler solutions generally for the sake of faster code delivery in software development projects. Eventually, TD is repaid and usually the long-term costs overweigh the short-term benefits. Therefore, proper identification and management of technical debt is crucial in any software development effort. Although the technical debt concept in the software domain has been extensively studied in the literature, there is a limited understanding of the indicators of TD in Software as a Service (SaaS) applications. Hence, this study investigates the indicators and major categories in organizations that provide support for SaaS applications to their customers. To this end, an exploratory case study has been conducted in a SaaS development vendor partner and consultancy company. Interviews have been conducted with nine employees in the company and an in-depth analysis of collected data have been carried out to identify the major technical debt categories specific to SaaS applications. The results revealed that TD categories and problems in traditional software applications are also valid for SaaS projects. There are also new categories of TD specific to SaaS. We identified 15 unique SaaS-specific technical debt categories associated with 23 relevant indicators. Additionally, we noted the presence of 81 generic software technical debt indicators, classified under 23 categories.

## Hizmet Olarak Yazılım Uygulamalarında Teknik Borcun Kategorizasyonu

### Anahtar Kelimeler

Teknik Borç,  
Hizmet Olarak Yazılım,  
Hizmet Olarak Yazılımda  
Teknik Borç,  
SaaS

**Öz:** Teknik borç (TB), yazılım projelerinde genellikle daha hızlı kod teslimi için daha basit çözümler seçme veya kötü kararlar alma sonucunda ortaya çıkan durumları ifade eder. Sonuçta TB ödenir, ancak uzun vadeli maliyetler, kısa vadeli faydaları aşar. Bu nedenle, teknik borcun uygun bir şekilde tanımlanması ve yönetilmesi her yazılım geliştirme projesinde önemlidir. Yazılım alanında teknik borç kavramı literatürde kapsamlı bir şekilde incelenmesine rağmen, Hizmet Olarak Yazılım (SaaS) uygulamalarındaki TB göstergeleri oldukça sınırlıdır. Bu nedenle, bu çalışma, müşterilerine SaaS uygulamaları için destek sağlayan organizasyonlardaki TB göstergelerini ve başlıca kategorileri araştırmaktadır. Bu amaçla çalışmada, bir bağımsız SaaS ortağı olan danışmanlık şirketinde keşif amaçlı bir vaka çalışması gerçekleştirilmiştir. Bu kapsamda şirkette dokuz çalışanla görüşmeler yapılmış ve toplanan verilerin derinlemesine analizi ile SaaS uygulamalarına özgü başlıca TB kategorileri belirlenmiştir. Sonuçlar, geleneksel yazılım uygulamalarındaki TB kategorilerinin ve sorunlarının SaaS projeleri için de geçerli olduğunu göstermiştir. SaaS'a özgü 15 TB kategorisi ile ilişkili 23 TB göstergesi tespit edilmiştir. Buna ek olarak, genel yazılım projelerinde de geçerli olacak 81 TB göstergesi, 23 kategori altında gözlenmiştir.

\*İlgili Yazar, email: ozdenoz@metu.edu.tr

### 1. Introduction

Metaphors are cognitive lenses that help us make sense of situations and reality [1]. Technical Debt (TD) metaphor is used to describe the effects that result from making suboptimal decisions or taking shortcuts for various reasons during software and system development [2]. Although suboptimal decisions and solutions may save time and effort in the short term; they lead to issues in software product, requiring more time and effort for improvement and bug fixing in the long term [3][4].

Swift decisions and quick measures taken during software development life cycle impacts the final product quality [5]. Major factors contributing to suboptimal decisions that result in technical debt may include customer pressure, poor communication, low levels of code testing coverage, compromised or rigid software architecture, and insufficient documentation. These issues are also common in cloud computing service models, as they increase with platform-specific boundaries and limitations, and ultimately lead to problems in software processes and malfunctions in the end product delivered to customers [6][7][8].

Software as a Service (SaaS) is a cloud-based software delivery model where applications are provided to users by a service provider such as Google Cloud, Microsoft Azure or Salesforce [9]. Users of SaaS applications pay for the services on a subscription basis [10][11]. There has been a growing interest to such applications. According to Gartner's recent report, it is estimated that SaaS end-users is expected to spend around 195 Billion U.S. dollars worldwide in 2023 [12]. The key characteristics of SaaS applications include accessibility from any internet-enabled device, automatic updates to services, and scalability when needed. They also eliminate the need for businesses to install, run and maintain on-premises software and enable significant savings on hardware, software, and personnel costs [13].

There has been extensive research on technical debt from a general software development perspective. Nevertheless, understanding technical debt specific to SaaS applications is essential due to the unique challenges that this software delivery model presents. SaaS platforms, especially multitenant ones deliver scalable, reliable and customizable software to their customers. The complexity of cloud-based architectures, frequent updates to satisfy rapidly evolving customer expectations, and continuous delivery often pose distinct challenges and can cause unusual forms of technical debt. Addressing these challenges is essential as accumulating technical debt can lead to slower feature delivery and degraded quality of service. Moreover, as the SaaS business model typically relies on regular user subscriptions, maintaining a high quality of experience is critical to retain customers and attract new ones, thereby improving return on investment and ensuring profitability. Any slowdown or degradation in service quality due to technical debt can lead to customer dissatisfaction and eventually result in loss of revenue [14].

The purpose of this research is to specify and categorize the technical debt indicators and categories specific to Software as a Service application development projects. In order to achieve these purposes, we performed an exploratory case study in a SaaS-based customer relationship management application development company by interviewing nine employees who have various roles and experiences in product development and maintenance. We performed the interviews based on the ISO/IEC 12207 Software Life Cycle Processes categories to follow a systematic approach, and analyzed the interview data using the qualitative coding technique. This study contributes to the literature by specifying novel TD categories for SaaS projects and augmenting the existing knowledge about technical debt for general software development projects. The SaaS-specific TD categories and indicators can be used for facilitating more effective technical debt management throughout the software development life cycle.

The rest of the paper is structured as follows. Section 2 gives background information on technical debt and SaaS applications and provides the review of the literature. Section 3 explains the case study design and conduct stages, and the validity threats. Section 4 presents the results. The findings and discussions are provided in Section 5. Finally, Section 6 concludes with major findings of the study and future work.

## **2. Background and Related Work**

### **2.1. Software as a Service (SaaS)**

SaaS, also known as on-demand software, is a way to deliver software applications running on cloud platforms over the Internet as a service. Unlike on-premises software, which basically requires installation and maintenance by users, Software as a Service enables users to readily access the required services provided on cloud platforms via the Internet. SaaS solutions are usually preferred because of their lower upfront costs when compared to traditional or on-premise software [15].

SaaS applications typically use a multitenant architecture, where all applications and users share a single infrastructure and code base. This essentially enables faster and easier innovation of SaaS. As the software is delivered through the Internet, SaaS applications are ubiquitous. The ability for each user to access the services wherever and whenever they need them with adequate quality of service improves productivity and forms trust and satisfaction [16]. SaaS applications not only differ from packaged software based on cost advantage, but also their benefits on scalability, rapidness in implementation, accessibility, managing update frequencies, and ensuring security [17].

A noteworthy difference between a packaged software and SaaS is the ease of customization, which plays a significant role concurrently supporting multiple organizations with diverse needs and varying business processes with the same system. Most SaaS solutions provide no-code solutions or high-level abstractions, thereby saving both from development time, and the cost of maintaining a code base. Additionally, no-code automation and customization prevent harmful effects on the common infrastructure, enabling organizations to achieve success in their business processes [9][17]. While making the changes, users take advantage of using SaaS solutions, benefitting from documented and ready-to-use solutions for their use cases if the service provider offers the specifications of the platform, software or service. Organizations can handle these development and maintenance work through in-house teams, external support, or independent software vendors (ISVs) [9]. ISV partners refer to companies that specializes in creating and distributing software applications hosted in the cloud and offer services to customers under a subscription model. These vendors maintain their autonomy while forming strategic partnerships with larger platform providers, such as cloud hosting services to enhance their service delivery.

## 2.2 Technical Debt

Technical debt occurs when a suboptimal solution adds value to the software but leads to undesirable consequences. In other words, taking shortcuts during analysis, design, implementation, testing or even documentation phases of a project might end up in more effort and time spent on the tasks to resolve a defect or to enhance the quality of the end product [18]. The “technical debt” term was coined by Cunningham (1992) [2] in an experience report as follows: “...Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite.... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.” Since then, the term technical debt is used to define (usually) conscious technical decisions that will help in short-term or with a more urgent request, but that may have negative outcomes in the future for other dimensions.

Studies prove that technical debt must be seriously taken into consideration and carefully managed to prevent everlasting payment of the “debt”. A survey study conducted by Besker et al. [19] indicates that 36% of overall development time on average is wasted due to the technical debt. Noteworthy, it was stated that most time is spent on understanding the root cause of technical debt.

High TD awareness is the first step towards managing it [20], and its management becomes easier when causes and impacts of TD are identified. In prior research, various methods for identifying and managing TD have been explored. Falessi and Kazman [21] and Ramač et al. [22] conducted surveys among software practitioners to identify common TD issues, their frequency, and causes. They also reported on practitioners’ familiarity with the TD concept. Notably, multiple case studies have compared technical debt management across different companies [23][24][25][26]. Kruchten et al. [3] surveyed the evolution of technical debt, and proposed an organizational solution.

Technical debt categorization is crucial for making informed decisions and proper management of TD in software development projects. It helps teams understand the specific type of debt they are dealing with, prioritize their solution efforts based on the debt’s impact, and align these efforts with the project’s overall goals. This organized approach also enhances communication among stakeholders, guides resource allocation, and aids in tracking the progress of debt reduction [3][27].

Alves et al. [28] performed an SLR to reveal different TD types and indicators, considering the nature of the development process. Zazworka et al. [6] advocated using automated tools over manual methods, while Ramasubbu and Kemerer [29] considered the TD types arising from client-vendor interdependencies during system maintenance and Codabux et al. [4] took an empirical approach by gathering insights from practitioners on TD management.

Specifically, we can talk about three technical debt categorization levels. The first level technical debt categorization can simply be performed on the *intention level* as suggested by McConnell [30]. Based on this categorization, each technical debt can be categorized as *Intentional* or *Unintentional*. At the second categorization level, Fowler [31] classifies the technical debt based on the intention of the person who creates the debt but in a more detailed way than McConnell’s classification: *Deliberate, Reckless, Inadvertent and Prudent*. At the third level, Alves et al. [28] presented 13 technical categories at the process level based on a systematic literature review study. These are *Architecture Debt, Build Debt, Code Debt, Defect Debt, Design Debt, Documentation Debt, Infrastructure Debt, People Debt, Process Debt, Requirement Debt, Service Debt, Test Debt, Usability Debt* and *Versioning Debt*.

In our previous research on detecting technical debt on ambiguous issue statements (*the citation was removed intentionally to ensure double blind review process*), we specified that most of the choices leading to architectural or structural debt was unintentional, as the effects of these decisions often become clear only in the later stages of the software development cycle. However, almost all suboptimal solutions chosen purposefully to address customer or stakeholder needs result in decreased product quality.

### 2.3 Technical Debt in SaaS

Despite these extensive studies on TD in software domain, our understanding of TD in SaaS applications and enterprise-level software, such as customer relationship management software, remains limited. Klinger et al. [26] conducted interviews with technical architects related to enterprise solutions, while Alzaghoul and Bahsoon [25] analyzed the option-based approach to clear TD in cloud-based solutions. Ramassubbu and Kemerer [29] quantified the negative impact of TD on system reliability. Kumar et al. [32] worked on identifying and estimating TD in SaaS clouds, emphasizing the distinction between "good debt" and "bad debt" in the context of SaaS applications.

In SaaS projects, it has to be ensured that the customizations and developments are compatible with the system's architecture and infrastructure. Proper documentation of changes made by external providers, ISVs, or in-house teams is necessary. Undocumented or poorly executed changes, especially by less-experienced staff, can lead to issues similar to those in on-premise services, or specific to the SaaS application and its infrastructure [33]. SaaS providers often address these problems through regular updates or bug-fixes, and users must follow these updates closely [34]. It is also vital to document customizations and business processes to avoid technical debt, which can become a problem in SaaS applications if not managed carefully.

As understood from our first case study (*the citation was removed intentionally to ensure double blind review process*) in identifying TD based on ambiguous issue statements, we determined that Level 3 categorization holds greater significance for comprehending the underlying causes of technical debt. This level of categorization yielded more detailed insights, aligning closely with the phases of the software development life cycle. However, Level 3 categorization needs to be specifically examined in the context of SaaS and CRM applications to better understand the nature of technical debt in these systems.

## 3. Case Study Design and Conduct

In order to determine technical debt categories specific to SaaS applications, we preferred to follow the qualitative research approach [35]. To gain a deeper understanding of a phenomenon in a specific field and communicate with research participants in their natural settings, we conducted an exploratory case study [36] and aimed to answer the following research questions:

**RQ1:** What are the indicators highlighting TD in SaaS application development?

**RQ2:** What are the technical debt categories specific to SaaS applications?

Below, we describe the case study design and conduct stages.

### 3.1 Case, Process and Participant Selection

We used the purposive sampling technique in determining the case organization [37]. In our purposive sampling process, we aimed at finding a company that primarily focuses on developing SaaS applications. Such a company should fulfil essential software development life cycle activities, such as software analysis, design, implementation, testing and project management to a certain level. Additionally, a key criterion was the presence of an active SaaS development project within the company. As the practitioners were expected to discuss the problems that they experience in software development based on their current project, they would be more precise about the issues and the root causes of these issues.

Below, the case study components and the decisions we made at the design stage are described.

**The Case Company:** The company we selected for conducting the exploratory case study specializes in asset and file management and is involved in various implementation and integration projects on multiple platforms. The company has several teams that are competent and work on various digital platforms. The number of employees of the company is around 100. Apart from integration and platform-based solutions, there are also software products that the company works on and offers to the market. The main purpose of most of these products is to facilitate asset management and to create cost-effective solutions. The remaining products aim to provide an ease of integration for customers and to increase customer communication.

**The Case Project:** For the case project, we selected an ongoing SaaS-based CRM (customer relationship management) project. The platform that has been used in the case product is Salesforce. Salesforce company is specialized in CRM applications, with a focus on sales, marketing and customer services. The company both offers SaaS and platform as a service (PaaS). The CRM product that we choose for the case has been released to the market for about 15 years ago and new feature enhancements and maintenance work continue since then. The goal of the product is to provide a cost-effective customer file management system for a specific SaaS platform. Various developers and experts have contributed to the product over the years. Specifically, product and platform-based development activities have been going on in our case company by a team of 8-10 people for the last five years. The product that the case company offers to the market not only includes platform-oriented components, but also 3<sup>rd</sup> party packages and products. Some of which are managed by the core team. Since the product was developed for the platform, it must comply with the limitations, requirements and security obligations imposed by the platform. Due to these characteristics, it was considered that the project is a very good candidate for observing various TD categories.

**The Software Life Cycle Processes:** In order to understand the nature of the TD in SaaS applications, the first step was to design an interview questionnaire. The main purpose of the interview was to access wide range of different technical debt issues experienced by people working in organizations like our case company, having various experiences and roles in the project. In order to follow a structured approach in specifying the interview questions, we used the ISO/IEC 12207 [38] software life cycle process standard as the basis. In accordance with the goals of this study, we focused on the following technical management and technical process sub-group given in Table 1.

**Table 1.** Selected ISO/IEC 12207 Software Development Life Cycle Processes

Process Group	Process Name	Referring ISO/IEC 12207 Process ID
Technical Management Processes	Project Planning Process	6.3.1
	Project Assessment and Control Process	6.3.2
Technical Processes	Stakeholder Needs and Requirements Definition Process	6.4.2
	Systems/Software Requirements Definition Process	6.4.3
	Architecture Definition Process	6.4.4
	Design Definition Process	6.4.5
	Implementation Process	6.4.7
	Integration Process	6.4.8
	Verification Process	6.4.9
	Validation Process	6.4.11

**The Interviewees:** While choosing the interviewees, we aimed to access employees who have participated in the selected ISO/IEC 12207 Software Life Cycle development processes given in Table 1. The interviewees were not only defined by their roles, but also their expertise level in SaaS and software development. We contacted the potential participants through the company’s point of contact. In total, we have agreed with nine employees in the project who are willing to discuss their experiences and issues that they experience in SaaS development process. The roles, experience levels and TD awareness levels of these participants as gathered from interviews are summarized in Table 2. TD Awareness levels have been categorized based on the definitions given below:

- **Low TD Awareness:** The participant might be unfamiliar with the TD metaphor entirely. However, they might still provide general examples of TD during the interview. Alternatively, they may understand the metaphor but remain uncertain about what is recognized as TD in the processes they are a part of.
- **Intermediate TD Awareness:** The participant is familiar with the TD metaphor and offers examples, though these might lack depth. They don't articulate the impact of TD on processes or share views on managing it.
- **High TD Awareness:** The participant has a clear understanding of the TD metaphor, provides specific examples of TD, recognizes the indicators, and is aware of issues arising from TD. They also offer insights on strategies to address or manage TD.

**Table 2.** Roles, experience levels, and TD awareness levels of each interviewee

ID	Role	Experience Level in Software Development (in years)	Experience Level in SaaS Development (in years)	TD Awareness
----	------	---	---	--------------

1	Consultant	3	3	Low
2	QA Engineer	5	5	High
3	Developer	4	4	High
4	Developer	5	2	Intermediate
5	Team Lead	13	3	High
6	Developer	5	2.5	Low
7	Developer	3	3	Intermediate
8	Release Manager	2	2	Low
9	Customer Support Specialist	5	2.5	Intermediate

### 3.2 Interview Questions

We identified the interview questions given in Table 3 to retrieve as much as possible information from the interviewees. These questions were designed in an open-ended structure and applicable for all processes given in Table 1. We specifically aimed not to emphasize the technical debt term presuming that the initial awareness of the interviewees on the subject are low. The interview questions were prepared in English and Turkish, as there were people in the team with nationalities other than Turkish as well.

**Table 3.** Interview Questions

No	Questions
1	What is your role in the organization, and what kind of responsibilities do you have in the company and the projects?
2	Can you describe the project and the technologies being used in the project?
3	Which of the following Software Life Cycle Processes do you participate in the project? Project Planning, Project Assessment and Control, Stakeholder Needs and Requirements Definition, Systems/Software Requirements Definition, Architecture Definition, Design Definition, Implementation, Integration, Verification, Validation
4	Do you experience any issues with the processes/practices that you participate in the project?
5	Do you practice the tasks/activities defined for each process specified in ISO/IEC 12207?
6	Do you think there are tasks that you fail or don't achieve completely?
7	What kind of problems occur in the project due to these issues?
8	How did you realize these problems? Do you have any further observations on the problems?
9	Do you know if these problems are caused by the SaaS provider or the development team that you belong to?
10	Do you have any recommendations or a way to solve/prevent these problems?
11	Do you experience any further challenges regarding the usage of SaaS applications in the project?
12	Do you sometimes choose easy/limited solutions to problems in your project, instead of using a better approach?
13	Do you have rework or refactoring cycles for your source code?
14	Do you have any problems that you choose not to fix/solve and only monitor the course/trend over time?
15	Are you familiar with the technical debt metaphor?
16	Do you measure the technical debt that is present in your project, using any tools or processes?
17	Do you mostly monitor or take further action on technical debt issues?
18	How do you decide the managerial activities on the technical debt issues?
19	Do you have any metrics or scale to prioritize the technical debt issues?
20	Do you have any technical debt that is expected to become worse in the future?
21	To what extent does technical debt affect your motivation?
22	Is your management aware of technical debt and are they taking any action?
23	Do you have additional issues that you would like to mention/highlight?
24	Do you think that there's room for improvement for your project/organization?

### 3.3 Validity Threats

In the design stage of this research, another aspect that we considered was the mitigation strategies to the validity threats. As a qualitative research methodology, the case study approach is subject to certain validity challenges. To ensure the integrity of our findings, we have determined and implemented measures to address potential threats to internal, construct, and external validity.

### 3.3.1. Construct Validity

Construct validity assesses whether the research tools accurately measure the variables or constructs they aim to evaluate [36]. For our study, it was necessary to ensure our interview questionnaire was capable of identifying SaaS-related TD categories without oversight. To this end, we utilized the ISO/IEC 12207 [38] software life cycle process standard as our foundation. We selected ten processes that spanned both technical and managerial categories from ISO/IEC 12207.

Another element that could influence construct validity is the potential ambiguity in terms and definitions used in the research. In particular, as presented in Table 2, team members demonstrate different levels awareness regarding TD. Recognizing that members of the SaaS project development team might interpret the term “technical debt” in various ways, we reframed our questions. Instead of asking, “What specific technical debt issues do you encounter?”, we posed, “What kind of problems do you experience in relation to a specific process?”.

### 3.3.2. Internal Validity

Internal validity considers the degree to which the observed effects in the research are actually caused by the manipulated variables or interventions, rather than some other external factors [36]. In our case study, we considered the possibility that the SaaS development team members might provide answers they believe the researchers want to hear. To mitigate this potential bias, we assured the participants that their responses would be kept confidential, and any data published would be anonymized. We believe this approach minimizes participant bias. Furthermore, instead of prompting for confirmatory answers, we encouraged participants to give detailed, descriptive, open-ended responses to our interview questions, elaborating on problematic situations as comprehensively as possible.

### 3.3.3. External Validity

External validity concerns the generalizability of the research findings outside the specific context or sample in which the research was conducted [36]. As the study focuses on a single CRM application, the findings might not be generalized to other SaaS applications or other domains. In the findings and results sections, we clearly stated the research findings. These insights are highly valuable; however, they pertain primarily to the specific CRM application studied. Additional qualitative and quantitative research is needed to explore technical debt categories in SaaS application for other domains. We performed the interview with nine employees in a SaaS project and covered a wide diversity of the roles and experience levels within the development team. This has provided a more comprehensive view of the technical debt landscape.

## 3.4 Case Study Conduct

In this section, we explain the data collection and data analysis stages of the case study.

### 3.4.1 Data Collection

The data was obtained from semi-structured interviews based on the questions provided in Table 3. We followed the techniques recommended by Patton [39] to ensure comprehensive coverage of every process listed in Table 1, while also retrieving participants’ experiences in a flexible way. In total, nine face-to-face interview sessions were conducted, with each taking around 45 minutes. Out of the nine interviewees, two preferred to conduct their interviews in English. Each team member was responsible for at least one SDLC process listed in Table 2. The SDLC processes experienced by the organization, in terms of the number of participants responsible for each process, are detailed below.

At the start of each interview session, the participants were asked about the processes for which they were responsible. To aid in this, the names and purposes of each process were presented to the participants. If a participant requested examples for any process, these were also provided by referring to the activities defined in the ISO/IEC 12207 standard. Table 4 given below shows the SDLC process involvement for each interview participant.

**Table 4.** SDLC process involvement for each interviewee

Process Name / Interviewee #	1	2	3	4	5	6	7	8	9
Project Planning Process					X			X	X
Project Assessment and Control Process								X	
Stakeholder Needs and Requirements Definition Process			X	X	X		X	X	X
Systems/Software Requirements Definition Process			X	X	X	X	X		

Architecture Definition Process		X	X	X		
Design Definition Process		X	X	X	X	X
Implementation Process	X	X	X	X	X	X
Integration Process	X	X	X	X	X	
Verification Process	X		X	X	X	X
Validation Process	X				X	X

### 3.4.2 Data Analysis: Coding and Categorization

In analyzing the data, we applied open and axial coding techniques to identify themes, as recommended by [40] and Lochmiller [41]. First, the interview data was first transcribed. The coding and categorization processes were carried out in three iterative and incremental stages.

During the first stage, open coding helped us specify initial TD indicators within the project. Next, we established connections between the codes, focusing on process levels and keywords highlighted by the participants. These stages enabled us to detect TD categories specific to SaaS development, addressing RQ1 and RQ2 respectively. While determining TD categories, we sought patterns that could define TD category themes. To ensure consistent and accurate coding, two review sessions were held: one after the open coding and another at the end of the analysis.

The TD indicators identified from the interview transcripts, along with the resulting TD categories, will be presented in the following section.

## 4. Findings

SaaS applications are intrinsically complex. They are built to serve clients with diverse needs and designed to be universally accessible. Therefore, they frequently subject to updates and modifications. As businesses race to introduce new features or adapt to market demands, there is often a tendency to make swift decisions, sometimes overlooking long-term implications. Such decisions can inadvertently introduce technical debt.

Table 5 given below presents technical debt indicators and categories for both SaaS applications and generic software development projects. In total, we identified 38 TD categories, and 15 of them are specific to SaaS projects. We also found 104 TD indicators in total, 23 being related only to SaaS projects.

These indicators were derived directly from responses given by interview participants during our case study. Once the indicators were defined, the categories were determined over three iterations. Multiple instances of TD indicators have been removed from Table 5.

The table's first column specifies if the TD category and its associated indicators are common in generic software development or are unique to SaaS projects. This is followed by columns providing the TD category name, its description, and the corresponding TD indicators. The table is organized in ascending order based on the TD categories. SaaS-specific technical debt categories are highlighted with an orange background.

**Table 5.** TD categories, TD descriptions and associated TD indicators

TD Type SaaS-Specific / Generic	TD Category	Category Description	TD Indicators
Generic	1. Communication Issues	Represents any problems related to conveying or understanding information among team members, stakeholders, or customers.	a. Communication barriers on work tracking b. Miscommunication on customer requests c. Miscommunication with stakeholders/team members
Generic	2. Design Adaptability Challenges	Indicates the difficulties in adjusting designs based on evolving customer requests or requirements.	a. Customer driven volatile design

<b>Generic</b>	3.	Design and Implementation Mismatch		Denotes scenarios where the design and the actual implementation don't align or match as intended.	<ul style="list-style-type: none"> <li>a. Incomplete implementation wrt design</li> <li>b. Insufficient design maintenance</li> <li>c. Insufficient design verification/validation</li> </ul>
<b>Generic</b>	4.	Design Inadequacy		Addresses instances where the design is incomplete, flawed, or based on incorrect assumptions.	<ul style="list-style-type: none"> <li>a. Design with incomplete/incorrect requirements</li> <li>b. Incomplete design</li> <li>c. Lack of solid design principles</li> </ul>
<b>Generic</b>	5.	Design Inefficiency		Covers situations where the design process or outcome is inefficient, overly complex, or doesn't adhere to standard practices.	<ul style="list-style-type: none"> <li>a. Overengineering</li> </ul>
<b>Generic</b>	6.	Design Process Inefficiency		Refers to instances where the design process is flawed or doesn't yield optimal results.	<ul style="list-style-type: none"> <li>a. Poor design process</li> </ul>
<b>Generic</b>	7.	Document Maintenance Deficiency		Indicates a lack of adequate updates, maintenance, or attention to various project or system documents.	<ul style="list-style-type: none"> <li>a. Insufficient maintenance of design documentation</li> </ul>
<b>Generic</b>	8.	Documentation Deficiency		Refers to situations where critical documentation is missing or inadequate, potentially leading to confusion or mistakes.	<ul style="list-style-type: none"> <li>a. Inadequate architecture definition</li> <li>b. Inadequate design definition</li> <li>c. Insufficient design documentation</li> <li>d. Insufficient requirements documentation</li> <li>e. Missing documentation</li> </ul>
<b>SaaS-Specific</b>	9.	Explorative Challenges	SaaS	Challenges that emerge when attempting to innovate or implement new methods, features, or processes within a SaaS environment.	<ul style="list-style-type: none"> <li>a. Exploration of new approaches in SaaS</li> </ul>
<b>SaaS-Specific</b>	10.	Inadequate Support	SaaS	Occurs where the SaaS provider does not offer enough assistance, guidance, or customer service for users facing issues.	<ul style="list-style-type: none"> <li>a. Inadequate support from SaaS provider</li> </ul>
<b>Generic</b>	11.	Inadequate Technical Management	Debt	This theme refers to the challenges of managing, addressing, or accumulating technical debt in the project.	<ul style="list-style-type: none"> <li>a. Bug-focused releases</li> <li>b. Customer-driven quick fixes</li> <li>c. Delayed problem identification</li> <li>d. Developer-initiated refactoring</li> <li>e. Incomplete solutions</li> <li>f. Rework for fixed problems</li> <li>g. Lack of design refactoring</li> <li>h. Post-verification problem identification</li> <li>i. Quick unconventional fixes</li> <li>j. Refactoring cycles</li> <li>k. Rework time constraints</li> <li>l. Rigid architecture and coding structure</li> <li>m. Temporary fixes</li> </ul>

<b>Generic</b>	12.	Inefficient Work Planning/Estimation	Work	Captures issues related to misestimating work durations, improperly planning tasks, or having conflicts about priorities	<ul style="list-style-type: none"> <li>a. Decision ambiguities</li> <li>b. Estimation difficulties</li> <li>c. Issue priority conflicts</li> <li>d. Planning conflicts with upper management</li> <li>e. Priority conflicts</li> <li>f. Rising ticket resolution time</li> </ul>
<b>SaaS-Specific</b>	13.	Insufficient Communication	SaaS	A lack of clarity or adequate information from the SaaS provider. This can be in terms of potential impacts, updates, or changes that might affect users.	<ul style="list-style-type: none"> <li>a. Insufficient impact information from SaaS</li> </ul>
<b>Generic</b>	14.	Issue/Task Description Issues		Highlights problems arising from unclear or incomplete descriptions of tasks or issues.	<ul style="list-style-type: none"> <li>a. Insufficient issue details</li> </ul>
<b>SaaS-Specific</b>	15.	Known SaaS-Specific Issues		Issues that are already identified and recognized with the SaaS provider. These are often listed or documented but remain unresolved.	<ul style="list-style-type: none"> <li>a. Pre-existing issues with SaaS</li> </ul>
<b>Generic</b>	16.	Legacy System Challenges	System	This theme refers to the difficulties associated with maintaining, upgrading, or working within older systems that may no longer be efficient or compatible with newer technologies.	<ul style="list-style-type: none"> <li>a. Legacy system maintenance</li> </ul>
<b>Generic</b>	17.	Process Adherence Issues		Highlights situations where established processes are not being followed or are incompletely adhered to.	<ul style="list-style-type: none"> <li>a. Incomplete process adherence</li> <li>b. Ineffective parallel design-implementation cycles</li> <li>c. Neglected backlog maintenance</li> </ul>
<b>Generic</b>	18.	Process Definition Ambiguity		Describes situations where processes or criteria are not clearly defined, leading to confusion or misalignment	<ul style="list-style-type: none"> <li>a. Ambiguous completion criteria</li> <li>b. Undefined checkpoints</li> </ul>
<b>Generic</b>	19.	Quality Assurance Inefficiency		Encompasses challenges and inefficiencies in ensuring the quality of code, designs, or implementations.	<ul style="list-style-type: none"> <li>a. Absent code convention checklist</li> <li>b. Delayed testing cycles</li> <li>c. Ineffective code review</li> <li>d. Self-code verification neglect</li> </ul>
<b>SaaS-Specific</b>	20.	Reactive Adjustments	SaaS	The need for users to make changes, fixes, or adaptations due to unexpected or uncommunicated changes made by the SaaS provider.	<ul style="list-style-type: none"> <li>a. Adjustments needed due to SaaS alterations</li> </ul>
<b>Generic</b>	21.	Release Management Issues		Denotes difficulties related to managing product releases, updates, or ensuring customers can keep up with the release pace.	<ul style="list-style-type: none"> <li>a. Mismatch of customer and product needs</li> </ul>
<b>Generic</b>	22.	Requirement Issues		Refers to problems related to understanding, gathering, or clearly defining requirements.	<ul style="list-style-type: none"> <li>a. Inadequate requirement gathering</li> <li>b. Incomplete requirement understanding</li> <li>c. Incomplete requirements</li> <li>d. Requirement consensus lack</li> <li>e. Undefined stakeholder needs</li> <li>f. Vague requirements</li> </ul>

<b>Generic</b>	23.	Requirements and Design Mismatch	Highlights situations where there's a disparity between what's required and the resulting design.	a.	Insufficient design verification/validation
<b>SaaS-Specific</b>	24.	SaaS Configuration Overhead	Challenges related to the extensive setup, tuning, or customization required on the SaaS side. It indicates that the SaaS might have a complex or not have user-friendly configuration setup.	a.	Extensive SaaS-side configurations
<b>SaaS-Specific</b>	25.	SaaS Imposed Limitations	These are strict constraints set by the SaaS provider (such as the number of API calls per hour, storage capacity, customization options, or integration capabilities).	a. b. c.	Limitations imposed by SaaS governor Constraints imposed by SaaS provider Inability to exceed SaaS limitations
<b>SaaS-Specific</b>	26.	SaaS-Induced Modifications	Challenges that arise directly due to changes, updates, or decisions made by the SaaS provider. This can include anything from platform changes to the deprecation of certain features.	a. b. c.	Platform-driven design alterations Release-induced incompatibilities Deprecation-induced disruptions
<b>SaaS-Specific</b>	27.	SaaS-Platform Specific Limitations	Refers to the constraints that are unique to a specific SaaS platform. Salesforce has its own set of limitations distinct such as platform-specific development languages, modules, or unique architectural elements.	a.	Salesforce-specific SaaS constraints
<b>SaaS-Specific</b>	28.	SaaS Testing and Troubleshooting Limitations	Difficulties faced during the testing or troubleshooting phases because of constraints or peculiarities in the SaaS.	b. c.	Testing and troubleshooting issues due to SaaS limits SaaS-related troubleshooting difficulty
<b>SaaS-Specific</b>	29.	Setup and Replication Challenges	Difficulties related to creating, setting up, or duplicating SaaS environments. This can range from initial environment setup to trying to reproduce specific conditions for testing or troubleshooting.	a. b. c.	Time-consuming SaaS environment setup Difficulty in replicating live environments Challenges in customer environment replication
<b>SaaS-Specific</b>	30.	Shared Accountability	This theme pertains to situations where both the software vendor and the development team share the blame for issues or conflicts. It emphasizes that responsibility is not solely on one party.	a.	Accountability on both vendor and development team
<b>Generic</b>	31.	Standards and Best Practices Violation	Represents scenarios where industry standards, best practices, or internal guidelines aren't followed.	b. c. d. e. f. g. h. i.	Absence of architectural standards Absent unit test standards Best practice implementation neglect Flawed testing process Incomplete unit tests Lack of a defined design process Neglect of solid design principles Neglected code conventions

				<ul style="list-style-type: none"> <li>j. Non-conforming code practices</li> <li>k. Undefined test steps</li> </ul>
<b>SaaS-Specific</b>	32.	Technological Diversity Challenges	Issues that arise due to using a mix of different technologies, tools, or platforms. This might be the integration of varied systems or the problems that arise from them.	<ul style="list-style-type: none"> <li>a. Diverse technology hindrances</li> </ul>
<b>Generic</b>	33.	Testing Limitations	Deals with challenges in the testing phase, including time restrictions, missing tests, or lack of automation.	<ul style="list-style-type: none"> <li>a. Insufficient testing</li> <li>b. Manual testing overhead</li> <li>c. Missed edge case testing</li> <li>d. Missing unit test coverage</li> <li>e. Time-restricted testing</li> </ul>
<b>Generic</b>	34.	Time Management Issues	Encompasses challenges related to deadlines, frequent releases, or general time constraints.	<ul style="list-style-type: none"> <li>a. Deadline pressures</li> <li>b. Frequent release pressure</li> <li>c. General time constraints</li> <li>d. Missed deadlines</li> </ul>
<b>SaaS-Specific</b>	35.	Trade-offs due to SaaS Architecture	Recognizing both the advantages and disadvantages of using a particular SaaS due to its design or structure, especially with architectures like multi-tenancy.	<ul style="list-style-type: none"> <li>a. Multi-tenancy-related SaaS trade-offs</li> <li>b. SaaS limits acknowledged, but seen as not positive</li> </ul>
<b>Generic</b>	36.	Verification Inefficiency	Highlights inefficiencies or challenges in the verification phase, whether it's customer or internal verifications.	<ul style="list-style-type: none"> <li>a. Customer-identified issues</li> <li>b. Verification difficulties</li> <li>c. Verification-identified issues</li> </ul>
<b>Generic</b>	37.	Workaround Dependencies	Refers to reliance on workarounds, be it due to platform constraints, specific requests, or other limitations.	<ul style="list-style-type: none"> <li>a. Limitation workarounds</li> <li>b. Platform-specific workarounds</li> <li>c. Request-specific workarounds</li> </ul>
<b>SaaS-Specific</b>	38.	Workaround for SaaS Limitations	The effort and methods employed to find alternative solutions or bypasses for issues or limits set by the SaaS provider.	<ul style="list-style-type: none"> <li>a. Employing workarounds for SaaS limits/issues</li> </ul>

## 5. Discussions

SaaS applications have recently gained significant interest due to their adaptability and scalability characteristics. These applications can serve to many clients with varying needs. However, this flexibility can contribute to increased TD levels in SaaS projects. In this study, we identified technical debt categories specific to SaaS projects. Our study was not limited to SaaS; we also identified categories of technical debt that apply to generic software development. The SaaS-specific TD categories identified in this study are novel, whereas the generic categories are consistent with prior literature on technical debt. However, these generic TD categories are explained at a finer level of abstraction, thereby providing a deeper understanding of the associated challenges.

### 5.1 SaaS-Specific Technical Debt Indicators and Categories

Technical debt is a well-discussed topic in general software development, but SaaS environments bring their own set of challenges and characteristics. The limitations imposed by SaaS platforms play a significant role in finding

quick solutions and making swift decisions which leading to technical debt. The technical debt categories we specified in this study for SaaS development are;

1. Explorative SaaS Challenges
2. Inadequate SaaS Support
3. Insufficient SaaS Communication
4. Known SaaS-Specific Issues
5. Reactive SaaS Adjustments
6. SaaS Configuration Overhead
7. SaaS Imposed Limitations,
8. SaaS-Platform Specific Limitations
9. SaaS-Induced Modifications
10. SaaS Testing and Troubleshooting Limitations
11. Setup and Replication Challenges
12. Shared Accountability
13. Technological Diversity Challenges
14. Trade-offs due to SaaS Architecture
15. Workaround for SaaS Limitations

Within the technical debt landscape, SaaS-related TD indicators stand out mainly from the characteristics of SaaS platforms. These indicators that range from configuration overheads to platform-specific limitations highlight the balance developers need to ensure between using the benefits of SaaS and managing its inherent constraints.

The *Trade-offs due to SaaS architecture*, *SaaS imposed limitations*, *SaaS platform specific limitations* and *Technological Diversity Challenges* categories are basically related to strict nature of the SaaS platforms. The architectural decisions made during the design and implementation phases of SaaS solutions often lead to inherent trade-offs. By its nature, SaaS usually adopts architectures like multi-tenancy [16]. While this approach offers scalability and maintenance advantages, it can also introduce constraints. For instance, while one tenant (i.e. the client organization) might benefit from a specific feature addition or update, another might find it disruptive. Moreover, customization becomes challenging when trying to adjust to the diverse needs of various tenants without affecting the base architecture. The architectural decisions to use such shared resources might lead to performance bottlenecks or security concerns in SaaS projects. Recognizing these trade-offs is essential to ensure that the advantages of a SaaS solution outweigh its potential limitations. Developers and decision-makers need to be continually be aware of these trade-offs to make informed choices, ensuring that the software remains resilient, secure, and effective.

The SaaS development landscape also involves multiple stakeholders, including SaaS providers, developers, third-party integrators, and end-users [17]. This dynamic nature can lead to situations where no single entity has full ownership of a technical challenge or its resultant debt. The *Shared Accountability* category highlights these blurred lines of responsibility. For instance, a limitation imposed by the SaaS provider might lead the development team to implement a workaround. If this workaround causes performance issues, it would not be clear who is responsible for the issue. This shared accountability can complicate the resolution process and make it challenging to specify the root cause of issues. For effective technical debt management in SaaS projects, it is crucial to establish clear lines of responsibility and communication pathways between all involved parties.

In SaaS projects, the complicated relationship between the *Inadequate SaaS Support*, *Insufficient SaaS Communication*, and *Known SaaS-Specific Issues* categories highlights the shared challenges that developers face. Although there is a close and symbiotic dependence between SaaS providers and developers, this relationship may lean towards a state of imbalance, often leading to the accumulation of technical debt. Even if SaaS providers communicate well, SaaS developers might still have to deal with known problems with the platform. Knowing about these issues does not always make them easy to avoid.

The *Explorative SaaS Challenges* technical debt occurs when developers try innovated operations within the SaaS environment. Whether employing a novel API or integrating a new feature, uncertainties associated with these actions may cause the development team to adopt shortcuts. The absence of established SaaS development guidelines may increase the technical debt levels.

The categories of *Inadequate SaaS Support* and *Insufficient SaaS Communication* often interact hand-to-hand. Insufficient support from SaaS providers may be in the form of limited guidance or insufficient documentation. This can increase the problem by forcing developers into unexpected actions. Especially, poor SaaS provider communication on platform updates, new features or other changes may result in misinterpreted features or misunderstood functionalities, both of which contribute to an increase in technical debt.

Similarly, the *Known SaaS-Specific Issues* and *Reactive SaaS Adjustments* TD categories represent potential challenges that are recognized but may not be readily resolved. Developers may search for quick, reactive code changes to overcome these issues. While such adjustments serve as temporary solutions, they inadvertently contribute to the accumulation of technical debt.

The *SaaS Configuration Overhead, Setup and Replication Challenges* and *SaaS-Induced Modifications* TD categories refer to the additional workload required by the configuration and customization of a SaaS platform to meet particular needs. Especially, a complex configuration structure without sufficient guidance can contribute to the accumulation of technical debt. Lastly, the categories of *SaaS Testing and Troubleshooting Limitations* TD category points to restrictions in effective software verification processes. Impediments in setting up suitable test environments or replicating conditions can lead to less rigorous testing, thereby causing further technical debt. A comprehensive understanding of these areas is critical for effective management of technical debt which is basically comprised of seven stages: Identification, Measurement, Prioritization, Resolution, Monitoring, Communication, and Prevention [7][42]. Awareness of potential technical debt is critical for a well-executed TD management life cycle. For instance, awareness on the *SaaS Configuration Overhead* technical debt category would enable teams to recognize specific type of indicators, making timely and accurate categorization in a SaaS context. This TD categorization will aid in the efficient allocation of resources, accurate issue prioritization for resolving TDs, and facilitate the development of resolution strategies.

### 5.3 Generic Software Technical Debt Categories

Our case study revealed 81 distinct indicators associated with 23 generic technical debt categories, as illustrated in Figure 1. This graph, which includes redundant occurrences, aims to capture the frequency of each technical debt category. From the data, we can note that the top three technical categories are *SaaS-Related Technical Debt* (elaborated in Section 5.1), *Inadequate Technical Debt Management*, and *Violations of Standards and Best Practices*, with 23, 13, and 11 instances respectively.

While Alves *et al.* [28] identified 13 high-level technical debt categories based on a systematic literature review, our study takes this a step further. We provide a deeper level of categorization associated with indicators. We argue that just referencing process names in TD categories does not suffice; each technical debt category should clearly hint potential challenges.

In addition to SaaS related TD categories, our study has highlighted a new area of concern in technical debt: *Inadequate Technical Debt Management*. Neglecting to manage technical debt effectively can have a snowball effect, leading to even more debt. The importance here is not just recognizing how technical debt accumulates but also understanding how to manage it properly. We recommend teams not only be cautious about occurring technical debt but also prioritize its effective management through regular audits, targeted sprints, and long-term planning. Lastly, we find categories like 'Verification Inefficiency' and 'Testing Limitations' to be pivotal. These emphasize the crucial role that verification and testing play in managing technical debt. A lack of rigor in these areas can worsen existing technical problems, making them increasingly complex to identify and resolve.

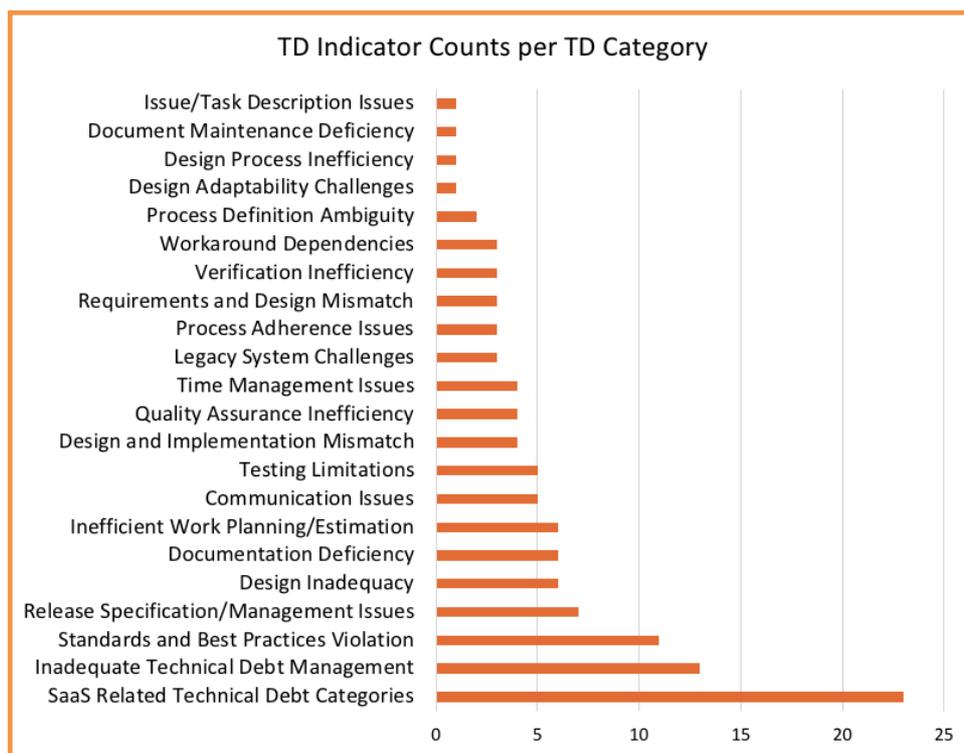


Figure 1. TD indicator counts per TD category

## 6. CONCLUSIONS

While SaaS development holds significant potential and benefits, it comes with its unique challenges. Such challenges highlight the need for a deeper understanding and strategic approach to SaaS development.

In this study, we approached to the problem from technical debt perspective and aimed to understand SaaS-specific technical debt indicators and categories. We performed an exploratory case study by conducting interviews with nine employees in a SaaS vendor. Based on the data we collected from interviewees, we identified 15 distinct SaaS-specific technical debt categories and 23 corresponding indicators. Additionally, we observed 81 generic software-specific technical debt indicators categorized into 23 types.

Addressing SaaS-specific categories is not just about acknowledging their existence, our study points out several categories that are unique to SaaS development. This emphasizes the distinct challenges of SaaS development and the need for specific strategies to manage its technical debt.

These granular categorizations are valuable in terms of enabling developers to identify, prioritize, tackle and prevent SaaS specific issues effectively. Our findings highlight the multifaceted nature of technical debt and stress the need for an encompassing approach to understanding and managing it. Therefore, our study makes a significant contribution for effective management of technical debt in SaaS applications.

Future studies can be performed with multiple case studies and on understanding the impact of SaaS technical debt on the overall software development lifecycle.

## References

- [1] Kendall, J. E., & Kendall, K. E. 1993. Metaphors and methodologies: Living beyond the systems machine. *MIS quarterly*, 149-171.
- [2] Cunningham, W. 1992. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2), 29-30.
- [3] Kruchten, P., Nord, R. L., & Ozkaya, I. 2012. Technical debt: From metaphor to theory and practice. *IEEE software*, 29(6), 18-21.
- [4] Codabux, Z., Williams, B. J., Bradshaw, G. L., & Cantor, M. 2017. An empirical assessment of technical debt practices in industry. *Journal of software: Evolution and Process*, 29(10), e1894.
- [5] Tom, E., Aurum, A., & Vidgen, R. 2012. A consolidated understanding of technical debt, *European Conference of Information Systems Proceedings*, <https://core.ac.uk/reader/301355650>.

- [6] Zazworka, N., Spínola, R. O., Vetro', A., Shull, F., & Seaman, C. 2013. A case study on effectively identifying technical debt. In Proceedings of the 17<sup>th</sup> International Conference on Evaluation and Assessment in Software Engineering (pp. 42-47).
- [7] Li, Z., Avgeriou, P., & Liang, P. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101, 193-220.
- [8] Rios, N., Spínola, R. O., Mendonça, M., & Seaman, C. 2020. The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from Brazil. *Empirical Software Engineering*, 25, 3216-3287.
- [9] Patidar, S., Rane, D., & Jain, P. 2012. A survey paper on cloud computing. In 2<sup>nd</sup> International Conference on Advanced Computing & Communication Technologies (pp. 394-398). IEEE.
- [10] Sun, W., Zhang, K., Chen, S. K., Zhang, X., & Liang, H. 2007. Software as a service: An integration perspective. In Service-Oriented Computing Proceedings, 5, 558-569, Springer Berlin Heidelberg.
- [11] Benlian, A., & Hess, T. 2011. Opportunities and risks of software-as-a-service: Findings from a survey of IT executives. *Decision support systems*, 52(1), 232-246.
- [12] Gartner Press Release 2022, <https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023>, (Access Date: 28.07.2023)
- [13] Turner, M., Budgen, D., & Brereton, P. 2003. Turning software into a service. *Computer*, 36(10), 38-44.
- [14] Vidhyalakshmi, R., & Kumar, V. 2014. Design comparison of traditional application and SaaS. In 2014 International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 541-544). IEEE.
- [15] Cusumano, M. 2010. Cloud computing and SaaS as new computing platforms. *Communications of the ACM*, 53(4), 27-29.
- [16] Chou, S. W., & Chiang, C. H. 2013. Understanding the formation of software-as-a-service (SaaS) satisfaction from the perspective of service quality. *Decision Support Systems*, 56, 148-155.
- [17] Sadiku, M. N., Musa, S. M., & Momoh, O. D. 2014. Cloud computing: opportunities and challenges. *IEEE potentials*, 33(1), 34-36.
- [18] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., ... & Zazworka, N. 2010. Managing technical debt in software-reliant systems. In Proceedings of the FSE/SDP workshop on Future of software engineering research (pp. 47-52).
- [19] Besker, T., Martini, A., & Bosch, J. 2017. The Pricey Bill of Technical Debt: When and by Whom will it be Paid?. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).
- [20] Suryanarayana, G., Samarthyam, G., & Sharma, T. 2014. Refactoring for software design smells: managing technical debt. Morgan Kaufmann.
- [21] Falessi, D., & Kazman, R. 2021. Worst smells and their worst reasons. In IEEE/ACM International Conference on Technical Debt (TechDebt) (pp. 45-54). IEEE.
- [22] Ramač, R., Mandić, V., Taušan, N., Rios, N., Freire, S., Pérez, B., ... & Spinola, R. 2022. Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry. *Journal of Systems and Software*, 184, 111114.
- [23] Zazworka, N., Izurieta, C., Wong, S., Cai, Y., Seaman, C., & Shull, F. 2014. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3), 403-426.
- [24] Iuliia, G. 2017. Technical Debt Management. In Russian Software Development Companies. Master's Thesis. St. Petersburg University Graduate School of Management.
- [25] Alzaghoul, E., & Bahsoon, R. 2013. CloudMTD: Using real options to manage technical debt in cloud-based service selection. In 2013 4th International Workshop on Managing Technical Debt (MTD) (pp. 55-62), IEEE.
- [26] Klinger, T., Tarr, P., Wagstrom, P., & Williams, C. 2011. An enterprise perspective on technical debt. In Proceedings of the 2nd Workshop on managing technical debt (pp. 35-38).

- [27] Nugroho, A., Visser, J., & Kuipers, T. 2011. An empirical model of technical debt and interest. In Proceedings of the 2<sup>nd</sup> workshop on managing technical debt, 1-8.
- [28] Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spínola, R. O. 2014. Towards an ontology of terms on technical debt. In 6<sup>th</sup> International Workshop on Managing Technical Debt (pp. 1-7). IEEE.
- [29] Ramasubbu, N., & Kemerer, C. F. 2016. Technical debt and the reliability of enterprise software systems: A competing risks analysis. *Management Science*, 62(5), 1487-1510.
- [30] Mcconnell, S. 2008. Construx,. Available from: <https://www.construx.com/resources/whitepaper-managing-technical-debt/>. (Access Date: 23.08.2023).
- [31] Fowler, M. 2009. Technical Debt Quadrant, Available from: <http://martinfowler.com/bliki/TechnicalDebtQuadrant.html>. (Access Date: 31.07.2023).
- [32] Kumar, S., Bahsoon, R., Chen, T., & Buyya, R. 2019. Identifying and estimating technical debt for service composition in SaaS cloud. In 2019 IEEE International Conference on Web Services (ICWS) (pp. 121-125). IEEE.
- [33] Agarwal, P. 2011. Continuous Scrum: Agile management of SAAS products. In *Proceedings of the 4th India Software Engineering Conference* (pp. 51-60).
- [34] Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., & Leaf, D. 2011. NIST cloud computing reference architecture. *NIST special publication*, 500(2011), 1-28.
- [35] Baxter, P., & Jack, S. 2008. Qualitative case study methodology: Study design and implementation for novice researchers. *The qualitative report*, 13(4), 544-559.
- [36] Runeson, P., & Höst, M. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14, 131-164.
- [37] Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. 2008. Selecting empirical methods for software engineering research. *Guide to advanced empirical software engineering*, 285-311.
- [38] ISO/IEC/ IEEE 12207 2017. Systems and software engineering — Software life cycle processes.
- [39] Patton, M. Q. 2002. Qualitative interviewing, in *Qualitative Research and Evaluation Methods*. Thousand Oaks, CA, USA: SAGE, pp. 344–347.
- [40] Williams, M., & Moser, T. 2019. The art of coding and thematic exploration in qualitative research. *International Management Review*, 15(1), 45-55.
- [41] Lochmiller, C. R. 2021. Conducting thematic analysis with qualitative data. *The Qualitative Report*, 26(6), 2029-2044.
- [42] Kruchten, P., Nord, R., & Ozkaya, I. 2019. Managing Technical Debt: Reducing Friction in Software Development. Software Engineering Institute.