STM JOURNALS

Journal of
Software Engineering Tools & Technology Trends

ISSN: 2394-7292
Volume 12, Issue 1, 2025
January–April
DOI (Journal): 10.37591/JoSETTT

https://journals.stmjournals.com/josettt

Research JoSETTT

# Balancing Tech Debt and Architectural Evolution–Role of Engineering Management

Ashwin Chavan*

***Abstract***

*There is a severe issue recognized today in software development, namely technical debt that impacts system extensibility, speed, and support. It emanates from choices for fast, second-rate remedies for instant gratification, always at the detriment of systemic stability. Architectural modifications are essential to keep up with evolving business landscapes and emerging innovations. Such interaction between technical debt and architectural change highlights some concerns engineering managers face concerning innovation and system stability. This article will discuss how technical debt arises and its various forms and measure its effects on performance, morale, and adaptability at work. It underlines the importance of engineering managers acting as technical debt champions, recognizing the use of tools and frameworks for early detection, and incorporating decision-making about technical debt across an organization. Artificial intelligence code analysis and cloud-native development are the key approaches to technical debt management, along with several other rising trends. The article shows how engineering leaders can coordinate architectural decisions for business goals to correlate short-term deliverables with long-term system coherence. Agile and lean approaches can address technical debt cyclically while keeping the door open to new ideas. Packed with numerous examples of standard traps and practical advice combined with an overview of future developments in technical debt research, this article provides engineering managers with actionable knowledge to handle the challenges of technical debt management effectively. Since the article explains how to address this pressure at every step, it is a guide for managing the balance between speed, optimization, and architectural integrity so that a system can always remain healthy, extensible, and ready for expansion.*

**Keywords:** Technical debt, engineering management, architectural evolution, software development, code quality, system scalability, innovation, maintenance costs

## INTRODUCTION

The idea of technical debt as one of the most challenging issues engaging engineering groups and corporations has become more important for the constantly growing speed of software development. Technical debt can be defined as short-term costs incurred due to taking shortcuts in developing specific code, design, infrastructure, or architecture while using imperfect, second-best elements because it was faster or more convenient to do so quickly than to take the necessary time to evaluate and implement better-quality options for achieving the same result. However, these are gained in the short term at the cost of system maintainability, performance, and scalability in the long term [1, 2]. To summarize, technical debt is as close as one can get to taking a loan in the future – postponing work that must be done in favor of the work that can be

***Author for Correspondence**
Ashwin Chavan
E-mail: ashwin.chavan@gmail.com

Software Architect, Pitney Bowes, Texas, USA

Received Date: January 29, 2025
Accepted Date: February 13, 2025
Published Date: March 18, 2025

**Citation:** Ashwin Chavan. Balancing Tech Debt and Architectural Evolution–Role of Engineering Management. Journal of Software Engineering Tools & Technology Trends. 2025; 12(1): 36–59p.

done now. Continuous enhancement is the gradual transformation and improvement of a software system's architecture over a specific timeframe. This applies because as business needs evolve or technology grows, it becomes important for organizational structures to evolve, too. Maintaining system sustainability, which requires evolving the architecture, becomes increasingly onerous when one has technical debt. Managed simplex environments occur over time when legacy systems and outdated solutions collect, making the updating or evolving architectures more complex and costly to maintain balance.

Technical debt and architectural change are two interconnected concepts. Technical debt as an embedded concept is frequently encountered when an organization demands the systems an organization operates to advance. At the same time, such architecture may lead to new types of debt that will slow the system's evolution even more. It is at the core of what delivery teams must do to balance, on the one hand, technical debt against architectural progression on the other. This becomes more than an arena for skilled developers to craft and coordinate; a balance between technical debt and architectural change is key. The process requires active direction and management and solid engineering support and initiative. Engineering managers are central to decisions that manage technical debt and how architecture decisions support an organization's strategic planning. They are responsible both for optimizing project performance within the short time frame of the project and for preserving system integrity and stability in the long run. A successful engineering manager often becomes important to identify and quantify technical debt quite early, lest it snowball into the development process. However, they need to promote innovation and improvement, which will contribute to changing the software's architecture. Simplifying, through cultural interventions that encourage technical debt assessment, the engineering manager directs the team's decision-making regarding the cost of technical debt related to short-term delivery and company growth.

In addition, engineering managers are directly involved in presenting the consequences of technical debt and architectures to the other members. Those two types of responsibilities can help them successfully promote or integrate engineering with other organizational goals. This makes it possible to deal with technical debts as they come rather than waiting for them to accumulate to the point that a bottleneck or redesign would be necessary.

This article attempts to understand how engineering management is critical in managing technical debt or architectural drift. They will help engineering leaders understand how to address the various challenges resulting from technical debt without compromising on systems' integrity and innovativeness. This article will present the implications of technical debt on aspects of the software development lifecycle and emergent strategies for managing deleveraging in different architectures. The purpose is to familiarize the target audience, the engineering managers, with the knowledge and tools that enable one to manage the tension between innovation speed and system architecture stability. Finally, it should be a valuable resource for managers and engineers who aspire to steer an organization's technical direction through the frequent curves and turns of technical debt and architecture refinement.

## TECHNICAL DEBT CONCEPTS

### What Is Technical Debt?

It is a process emerging from the shortcuts that are implemented in the development process in an attempt to implement a suboptimal solution. It is also used in a context similar to accounting debt because the cost of paying for it escalates. Since teams focus on increasing delivery and short-term benefits, many 'workarounds' or near-optimal approaches create longer-term problems. Technical debts in software engineering go beyond just coding mistakes but comprise everyday architectural and design decisions and sourcing strategies that accumulate to become a massive problem in the future as shown in Figure 1.


**Figure 1.** An overview of technical debt.

Concerning the dynamics of software systems, technical debt slows down the development process and decreases the manageability of the final product. The disposition of development speed over development quality leads to the accumulation of a technical debt that, if not managed, can impose degradation in system performance, system extensibility, and plausibly the morale of developers. Thus, technical debt is best controlled by striking a fine line between short-term tactical requirements and long-term strategic asset execution of higher-quality code.

## Types of Technical Debt

### *Code Debt*

Technical debt is where the developers choose the simplest way to get through coding rather than the best and most sustainable option. These are generally in light of time constraints or organizational circumstances. These changes mean that source code may become more challenging to comprehend, modify, integrate into tests, or upgrade after a while. This means that it increases the chances of encountering bugs and security breaches, resulting in future poor performance.

### *Design Debt*

Design debt is incurred when design is compromised or not good enough to accommodate the system's needs as it grows and changes over time. This spectrum of technical debt commonly arises from wrong decisions on architectural designs that cannot accommodate change or growth later on [3]. The design captured on the growing system results in architectures that a code is complex to change or enhance without disrupting its present performance. The software must be redesigned to overcome technical debts and meet the business's future adaptability and expansion needs.

### *Infrastructure Debt*

Maintenance debt is the lack of investment and care to update the foundational parts, including the databases, servers, or networks. Though delaying updates and concentrating on development work is often beneficial, infrastructure is critical to fail or become slow, unscalable, and buggy or force you to spend the time you could use for development. Maintaining infrastructural debt entails constant assessment and revising the system's fundamental architectures to support the needs of current applications.

### *Architectural Debt*

Design debt reflects a similar idea as design debt but is focused on higher-level architectural decisions that affect the architecture of the software in the future. Failure may also arise when teams fail to observe

some basic architectural principles in an attempt to take shortcuts. Different patterns of architectural debt mean that these applications are not very modular, the integration points are suboptimal, and the applications are complex to scale [4]. This weakens the team's flexibility or opportunity to add new features without a significant rewrite or refactoring [5]. Maintaining the architectural debt involves frequent check-ups and changes in the architectural layer of the system to support future evolution and technological enhancement as shown in Figure 2.

## Causes of Technical Debt

### *Speed-to-Market Pressures*

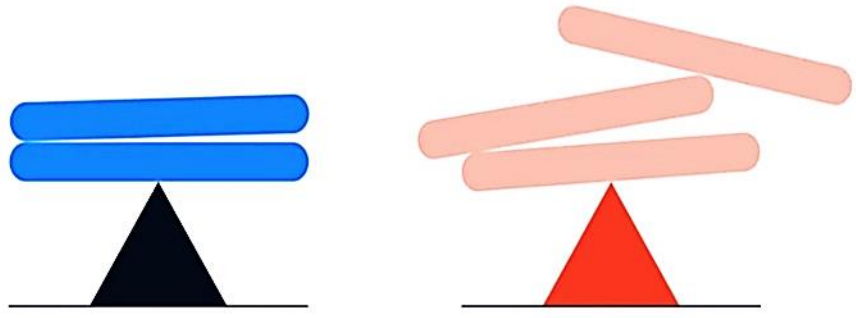
Management pressure to deliver certain products on the market or within a particular timeframe also leads to technical debt. In the course of organizational operations, many organizations aim at finding means to work faster in order to outperform competitors. While this approach causes rapid output to be delivered initially, it results in a buildup of technical debt in suboptimal, messy code, negatively impacting performance and sustainability in the long run [6]. People may decide not to refactor or optimize their code when working to a deadline because of time constraints, but this creates technical debt, which has to be paid off in terms of higher costs when maintaining code and slower rates of developing other new programs.

### *Lack of Resources*

Lack of resources, particularly human resources, time, or financial resources, are also sources of technical debt. When they do not have resources to carry out thorough testing, review the code, or create appropriate and quality systems, they patch up in several ways that add to technical debt, this shortcoming may also result from decisions that organizations make that put short-term wins above the system's best interest in the long run. The technical debt increases, and system quality degrades, reducing the possibility of innovating or scaling the application [7].

### *Evolving Requirements*

Over time, the business requirements get modified, and the initial specifications employed during the software designing do not suffice to cater to the organizational needs. Dynamic features like updates, changes of features, or even integration with a third system necessitate alteration of the codebase, frequently in a short time, with little consideration of the broader architecture [8]. Such changes may inflict discrepancies in abilities that the system can support at present and the new business requirements and can ultimately lead to the rise of technical debt. To avoid this, the different development teams need to constantly review the architecture and compare it with the current needs of the business.



**Intentional**

Occurs when an organization makes a conscious decision to optimize for the present rather than for the future.

**Unintentional**

Occurs when an organization makes an unavoidable mistake.

**Figure 2.** Other types of technical debt.

## Challenges in Managing Technical Debt

### *Balancing Business Goals and Innovation*

One of the most significant issues when dealing with technical debt is balancing growing business value and the delicacies of software maintenance. Enjoying a quick delivery of new features deemed critical for business performance inevitably leads to a constant growth of technical debt that constrains the further evolution of the product. The engineering managers need to decide when and how to handle and rectify all the technical debts, primarily not impacting their business goals [9]. Technical debt management cannot be a top priority since it means redirecting resources from developing important new features that, in return, seem less critical or helpful.

### *Managing Productivity*

Another is managing productivity when dealing with the issue of technical debt. In the long run, technical debt will reduce development velocity as maintainability and scalability costs increase. The debt leads to the protraction of testing cycles, recurrent bug repair, and issues delivering new capabilities – all unfavorable for team efficacy. It is the role of the engineering manager to make sure that any effort to reduce the technical debt has no impact on the morale or efficiency of the team. There is always a risk in choosing between getting more work done and creating more technical debt – the key is never to favor one over the other.

## Risks of Unmanaged Technical Debt

### *Reduced Performance and Scalability*

In the following tutorials, experts shall see how unmanaged technical debts affect the performance and scalability of information systems. The increased number of technical debts slowly exposes a substantial strain on the code, along with the potential slow and rigid speeds for processing, downtimes, and general complications with scaling as demands require. Performance problems can arise due to ineffective coding, algorithm problems, or insufficient platform, resulting from accumulated technical debt. Left unmitigated, these problems can worsen, making the system less effective and unable to expand as the business requirements grow.

### *Increased Maintenance Costs*

A final toxic effect of unmanaged technical debt is on costs and, more specifically, the increment in overall retention costs. The cost of maintaining a system with a large amount of technical debt is high in the long run because additional resources must be spent to diagnose, correct, and optimize the codebase. Taking care of technical debt can sometimes mean overhauling or redesigning, increasing the tone. The increasing stock of technical debt results in the degradation of developer productivity and an ever-rising cost regarding resolving problems [10]. Hence, effective technical debt management is necessary to control the extent of maintenance costs that are likely to reduce the organization’s profitability.

## IMPACT ON VARIOUS ASPECTS OF SOFTWARE ENGINEERING

Technical debt can influence multiple facets of software engineering, including runtime and architecture, software teams, and the application's end user. Engineering managers need to be fully aware of these impacts to avoid certain risks while striking a balance between an organization's corporate wants and the sustainable needs of the future.

## Operational Impact

This is the case because technical debt puts a software system at risk of poor operational performance when the technical debt becomes too much. When old or suboptimal code is present in the system, it will inevitably cause the system to run slower and require more resources. This leads to delayed system performances and increased downtime, affecting the systems' overall functionality [11, 12]. In enterprise systems, technical debt also affects maintenance tasks and makes performing required updates or patches harder. Operational acceleration is even more significant when technical debt stays

in legacy systems, which creates problems for scaling and adapting to new demands due to infrastructures' obsolescence.

**Architectural Impact**

It will be appreciated that technical debt has a direct impact on the architectural quality of a system. Developing bad architecture decisions at the early phase or some periods can lead to software architecture- requirements misfit. In the long run, such misalignments might weaken the flexibility of the system resource, thereby complicating the process and raising the cost of future architectural modifications. For instance, lousy refactoring in the early phases can create systems that cannot efficiently handle more load or integrate new functionality without a steep cost of technical debt [13]. These impacts are significant, especially in systems that have been developed to create monolithic structures where the more the technical debts are accumulated, the more the system becomes rigid to develop.

**Development Impact**

In the case of application development, for example, technical debt introduces many barriers to coding efficiency and code quality. Having to frequently go back to the existing code or deal with less-than-ideal solutions slows the developers and can cause additional time on the project schedule. Developers may be unhappy with improving unstable or poorly constructed code bases, which, in turn, affect morale and the ability to avoid burnout. These effects are further compounded in applications where development is rapid, as in agile development, progressive enhancement is hampered by time and resources devoted to technical debt. The system expands and develops, and the issues with handling and resolving the debt increase, which forms a negative circle known as the 'pay now, delay later' approach accompanied by a decrease in the quality of service.

**Team Dynamics and Morale**

Technical debt also plays a role in the relations among team members. The primary internal challenge is dealing with an increasing number of technical debts and responding to new feature demands and deadlines. Such constant switching can often be frustrating and lead to lower job satisfaction. Demotivation is likely to occur in teams once the members realize that long-term strategies are being compromised for short-term preferences in their output. This breaches trust in the management and reduces engineers' feelings of ownership over the product when they are constantly pressured to make tactical decisions that prioritize short-term business values before code quality [6]. In the long run, this affects higher turnover rates as developers demand healthy work conditions that would enable them to develop solid and sustainable systems as shown in Figure 3.

**Innovation and Technological Evolution**

Technical debt can prevent a team from adopting new technologies and pursuing new directions: the more significant the technical debt, the less the team can experiment and do new things. When resource continuity is devoted to managing debts, less can be used to search for a new solution that can yield more profits for the organization in the long run. Specifically, the concept of evolution can be slowed by the accumulation of layers of technological debris, such as old code and worn-down frameworks [14]. This results in the company persisting in supporting and sustaining suboptimal systems rather than adopting new and better innovative ones. For example, suppose it is located in such components as a database or API (application programming interface). In that case, it might restrict the usage of new technologies, such as cloud-native apps, machine learning, or artificial intelligence. The result is that the company cannot progress in developing new technology.

**Organizational Agility and Development Velocity**

The flexibility of an organization and the speed of its development are linked to the problem of teams' responsiveness to further conditions in the market or changes in the requirements of customers. Technical debt hinders this kind of responsiveness because time spent repairing and addressing this type of debt would otherwise be used to advance the development of new features or changes.

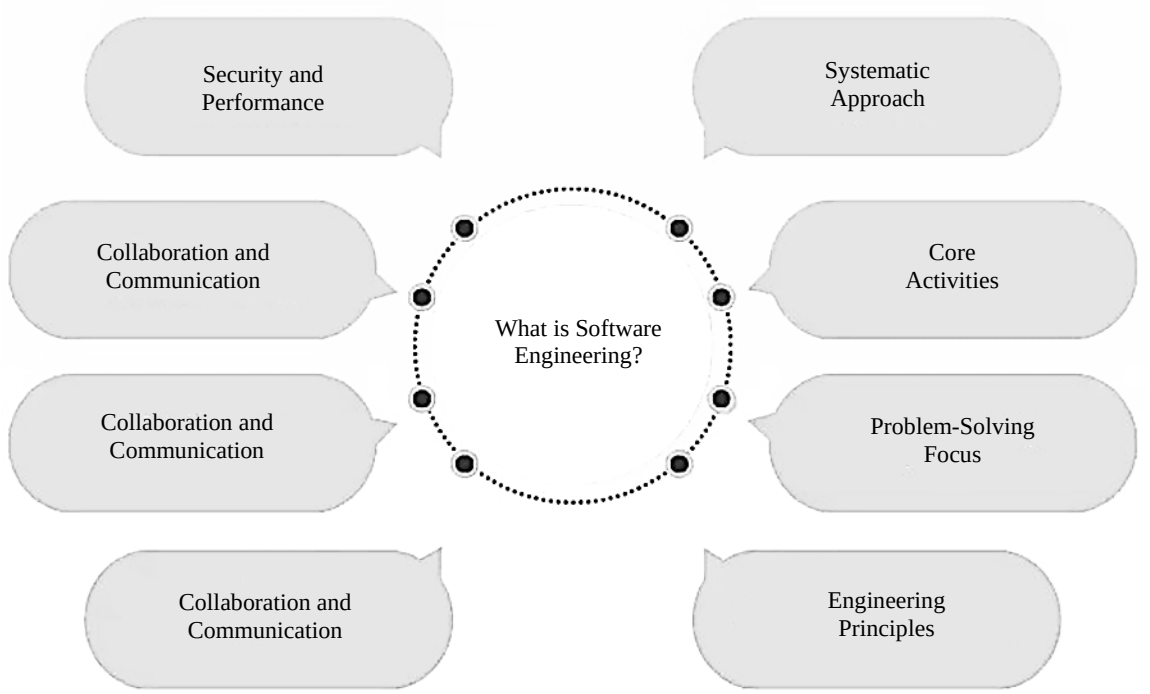



**Figure 3.** The importance of software engineering.

The greater the amount of technical debt, the harder it is to find new work against which to charge off the debt. This creates an environment where the requirement of keeping things going must hinder change. In high-velocity settings – for example, in an e-commerce or fintech setting – this can carry dire implications, with the enterprise in question may well lose out on notable market trends.

### Impact on Software Quality

Software quality is the first negative consequence of falling due to technical debt. As the system increases in size and complexity, it becomes far more challenging to manage the assurance of quality processes. Old code is riddled with errors and redundancies, and newer, kludged code that is poorly understood can create an increased risk of failure. In addition, technical debt hinders the introduction of needed refactorings, which is critical for preserving a clean and well-structured code base. This leads to the creation of substandard software that cannot deliver on the provider's or the consumer's expectations and expectations of the business. The decreased software quality effects are realized in several aspects, from customers to security and compliance.

## TECHNICAL DEBT IN DIFFERENT CONTEXTS

### Microservices and Distributed Systems

Over time, microservices architectures have received a popular reception based on scalability and flexibility. They also bring different technical debt issues. Compared to a monolithic architecture, where the application code is tightly coupled, microservices involve handling multiple services, each deployable on its own. This makes it easier for the programmers to duplicate code, have inconsistent interfaces, and have integration problems. Consequently, technical debt in a microservices environment will likely compound quickly and is considered more challenging.

One of the greatest difficulties lies in maintaining the continuity of information exchange and documentation between the services. In distributed systems, latency and network failures are normal and common occurrences that result in inefficiencies and performance issues, which are the main causative factors of technical debt [15]. Adopting distinct technologies and frameworks may lead to problems of inconsistency and dispersion within the development process.

Some of the best practices to recommend when managing technical debt in microservices include getting precise technical communication with the definition of communication standards, designing a standard interface of the microservices, and recommending and implementing sophisticated testing and monitoring tools. Through CI/CD (continuous integration/continuous deployment) pipelines, such technical debt can be detected early enough before it rises and becomes unmanageable. In addition, frequent refactoring and architectural reviews check that services are still compact and easily scalable as the services develop as shown in Figure 4 [16].

**Monolithic Systems**

Managing technical debt in monolithic systems is a slightly different proposition. Even though these systems are often integrated, having a single code base at the start of their development, they become more complex and challenging to maintain as they grow and evolve. In monolithic applications, technical debt problems are often expressed as dependency on other components, un-optimized code, and non-modular designs. The inherent difficulty can decrease the velocity at which applications can be built and scaled because alterations in one facet frequently alter the others.

Performance issues are also familiar with monolithic systems, especially in large-scale applications with relatively large data or user volumes. Controlling technical debt in such systems entails an approach such as converting the complex system into several smaller systems that are more manageable and less dependent on each other, a process often known as 'modularization'. It is easy to maintain and scale this approach in the long run. Making unit tests automatic and implementing code quality can help stop the formation of additional technical debt and sustain the system's stability.

**Legacy Systems**

Legacy systems are particularly problematic because the software is technologically outdated and incorporates no recent advancements in development. These systems often employ outdated programming languages and technology architectures and are very challenging to alter, enhance, and interface with more modern information systems. The technical debt incurred in technologies and infrastructure that are no longer current in the organization can compromise the performance, software maintenance cost is high, and it poses a higher risk of project failure.

It makes the case that to pay off the technical debt in existing systems, engineering managers must schedule remediation through modernization. There are specific approaches like code refactoring that address the legacy code and make the code structure nimble and cleaner [17]. The constant replacement of suboptimal parts with modern tools does not interrupt the business processes [18]. Transitioning to a cloud-based system can provide more flexibility and scalability than usual with older infrastructure. It is crucial to adhere to a serious approach toward the testing of all changes and carry out processes that might take place during the modernization phase.

**Enterprise Systems**

Enterprise resource planning (ERP) systems, commonly big and sophisticated software applications that deal with organizational core processes, produce considerable amounts of technical debt and are therefore under considerable pressure on this issue. Many of these systems are inherently intertwined with other organizational processes, which complicates attempts at altering these systems without impacting the overall enterprise environment. The direct implication of technical debt management in enterprise systems is the deceleration of the system's performance due to the increased database size, user dissatisfaction due to delays and slower systems, and the capacity to implement new features or adapt to new organizational needs.

Technical debt in enterprise systems refers to the ability of the architecture and development practices in relation to the firm's overall business. One goal is to apply modularity and service-oriented architecture (SOA) to prevent the complex from shaking when integrating new features.

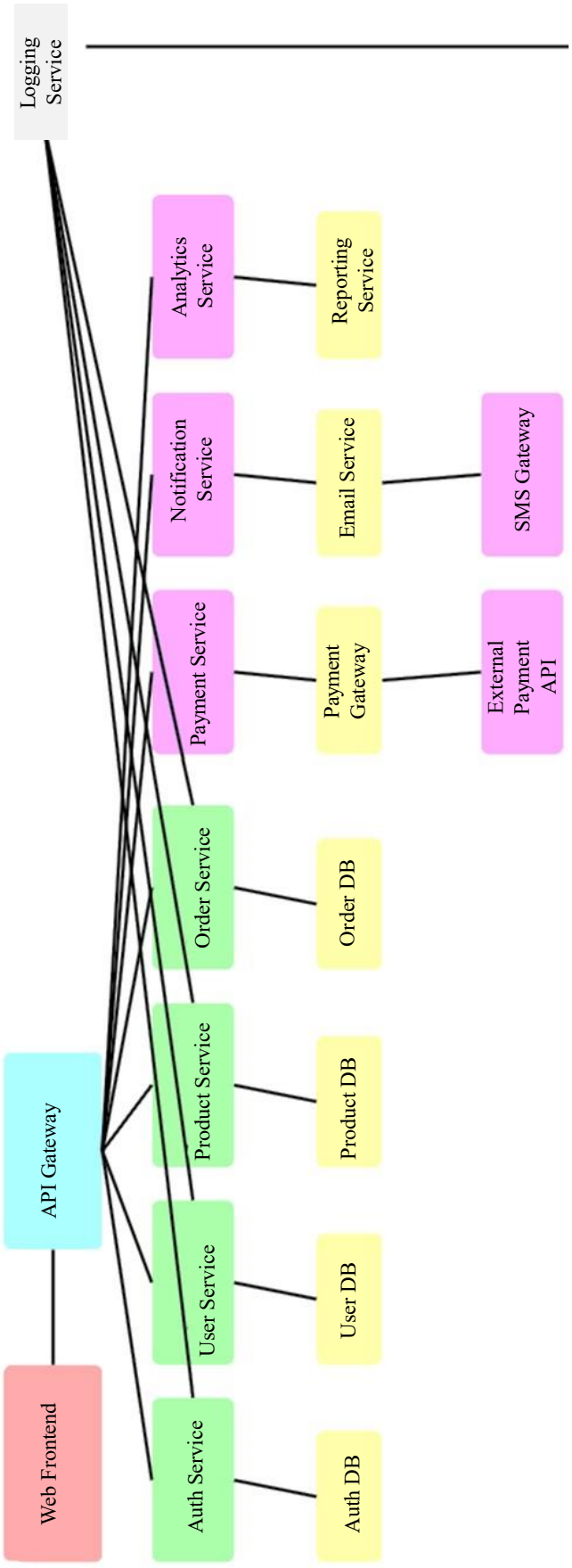



**Figure 4.** Microservices as technical debt.

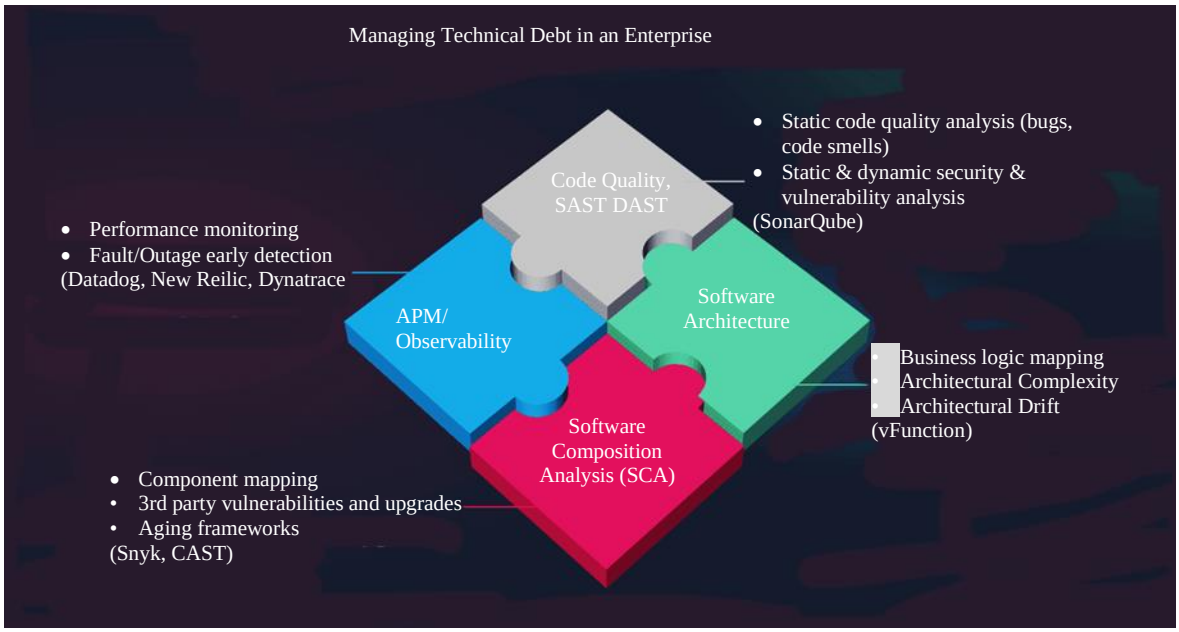



**Figure 5.** Architectural technical debt and its role in the enterprise.

The use of agile methods in developing enterprise systems may also provide a way to accelerate the redevelopment of iterations and the constant assessment and management of technical debt [19]. Enterprise systems should also look for debt areas using tools such as application under test (AUT), Static Code Analysis, and monitoring tools as shown in Figure 5.

## ROLE OF ENGINEERING LEADERSHIP/MANAGEMENT

### Cultivating a Culture of Technical Debt Awareness

There is always a technical debt associated with a product, and Proof of technical debt is that there is always a technical debt. It is possible to enter a state of technical debt psychologically. Cultivating technical debt consciousness. Organizational leadership in the engineering discipline is responsible for developing an organization's technical debt understanding culture. Technical debt is essential to management and the foundations of long-lasting software solutions. This can be implemented by regularly updating the team members about the consequences of technical debt on performance and the ability to scale up the system. This can be done through periodic talks or seminars, awareness creation through sessions that explain technical debts, and workshops to make those within the organizations understand the dangers of amassing technical debts. When this awareness becomes a 'norm' of development teams, the engineering managers can ensure that the developers consider the consequences of their design and coding choices.

Formal communication should be made about how the leadership is managing technical debt, and in the same way, organizational members should be made aware of the project's progress. When engineers comprehend the impacts of technical debt on not just the code base but also the objectives of an organization, like product delivery or customer satisfaction, it will force them to be proactive about the problem. Such a shift in mindset can foster an organizational culture where managing technical debt becomes a culture rather than a desperate repair process whenever issues arise.

### Balancing Innovation with Managing Technical Debt

Another important issue that any engineering leader is facing is how to innovate fast while at the same time taking care of technical debt. While innovation is critical to remain relevant in the market, such technological advancements will almost always build up technical debt over time. Engineering leadership then has the responsibility of ensuring that such innovations do not compromise the system's sustainability [20]. They need to offer the proper guidance for decision-making that allows teams to

assess if a new feature or a technology adds to a sustainable architecture or contributes to technical debt as shown in Figure 6.

The best engineering leaders promote practices within the teams that enable them to innovate while simultaneously creating practices for the system that are possible to sustain. They have to help their subordinates focus on tasks, minimizing existing debts as well as on the introduction of novelties. Continuous integration and continuous delivery practices can help introduce innovation at the same time as catching technical debt where it is early on in the SDLC (software development life cycle). By introducing slight changes to the system, engineering managers can encourage teams to improve the design while keeping the architecture stable.

**Proactive Identification and Management of Technical Debt**

Minimizing the effect of technical debt relies on identifying such a risk at an earlier stage and its management. While managing a team of developers, the engineering leaders have to personally find those places in the codebase that may end up with technical debt, especially during the initial stages of product construction. Having code reviews, architecture assessments, and performance reviews can effectively identify technical debt early enough in its life cycle. Leaders must also ensure that developers feel that whatever problems they encounter, they voluntarily report before whatever issue they escalate becomes a significant problem.

To prevent being overwhelmed by debt or to possess a better strategy for keeping track of it, engineers should begin to familiarize themselves with the best practices concerning the early identification of debts. One of them is the definition of done, which states that re-payment of the technical debt is one of the activities that should be in every cycle. Apart from conventional manual assessments of the technical debt, one can use automated static code checkers or applications whose primary function is to track the growth of the technical debt. These mechanisms may monitor coverage, size, and architecture problems and give feedback to the developers in real time on how the code debt increases.

***Strategies for Prevention***

It is much more effective if the technical debt does not accrue in the first place for a manager to deal with. Engineering leadership can avoid debt by giving proper directions on how to code, encouraging quality work, and allowing adequate time and resources for quality and scalable results. These should become a part of the development process rather than compromising quality for speed in the longer term [21].

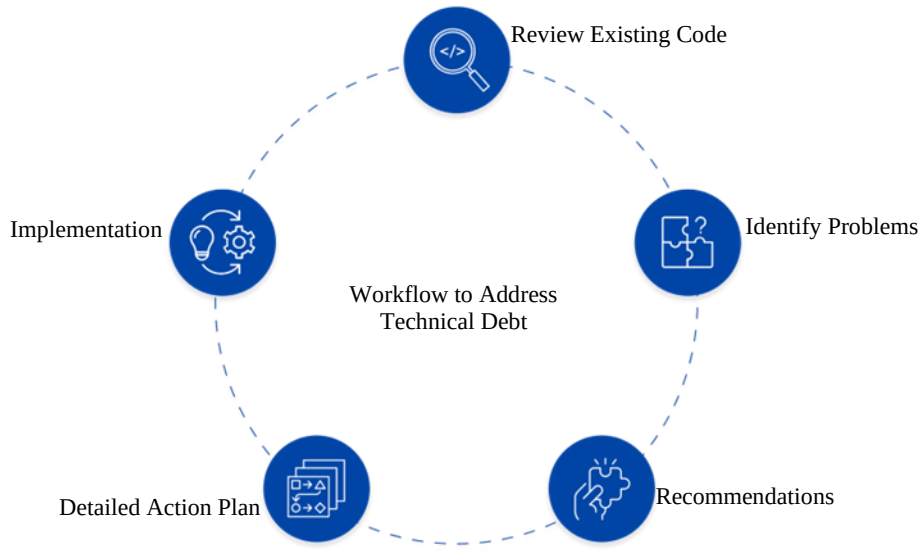



**Figure 6.** Technical debt management.

Another benefit is that refactoring can best be incorporated into the continuous delivery process, where changes occur slowly without a sharp disruption of the delivery flow. They also need to work towards achieving correct timing and expectations of deliverables on specific team deadlines. Given that expectations are sometimes unrealistic, decisions are made in haste, which ends up creating more technical debts. This allows engineering leaders to find a reasonable way to bring innovations fast while keeping design and code quality in mind to prevent the creation of a significant amount of technical debt.

### Embedding Technical Debt Awareness into Organizational Culture

Implementing technical debt awareness into the organizational culture is seen to need the commitment of leadership. Engineers, being managers, must be in a position to sell or convince top management and other organizational stakeholders of the need to manage technical debt and the effects of failure. In order to do this, leadership should create formal procedures that ensure that the technical debt is monitored, related to other corporate goals, and included in project reviews.

Various strategies for communication are essential in ensuring that technical debt is communicated throughout the organization. Engineering managers can use visual dashboards to show the status of technical debt and this on the project schedules and systems established [22]. This is because this transparency enables key stakeholders to picture the future consequences of technical debt. Another way could be scheduling presentations or reports on the status of different technical debt items to ensure that everyone in the organization, the developing team, and understand Code Entropy (CE) and its implications.

#### *Tools, Techniques, and Processes for Effective Communication*

Another strategy implicating communication is the capacity to explain technical debts to the different levels of the organization. Leadership can use Confluence or Jira to log and report on technical debt so everybody can hear. These tools allow engineers to record problems, rank tasks to do with credit, and analyze the work on remediation [23]. Furthermore, using graphs and charts to present information can also distort technical data and make it comprehensible for people not involved in technical work. Thereby, they will comprehend the risks. By institutionalizing technical debt tracking and communication, engineering managers guarantee that technical debt is treated at par with other concerns that define development [24].

### Early Detection and Mitigation Strategies for Technical Debt

Managing the technical debt is, therefore, significant to avoid the impacts of delayed action. Engineering leadership should invest in the tools that would receive the code by automatically running checks needed and reporting emerging debt areas at the beginning of development. The code metrics can, therefore, indicate areas of concern that need urgent attention, and this is through regular reviews. This causes a build-up of what is commonly referred to as technical debt, where efforts that could have been spent on other value-adding work are spent on rectifying earlier decisions.

Dispelling the myth of agile refactoring and architectural improvements should be balanced with other mitigation strategies. Managers need to ensure that the remediation processes do not take time away from the development process but instead form part of it. Teams find it easier to address debt within formidable sprints by adopting agile techniques like refactoring and integrating them into sprint planning as shown in Figure 7.

## ROLE OF ENGINEERING LEADERSHIP/MANAGEMENT WITH RESPECT TO ARCHITECTURE

### Influence of Technical Debt on Software Architecture

Evolution technical debt arising from decisions made in software development with a view of attaining some near-term objectives to create longer-term problems affects the evolution of software architecture in several ways.

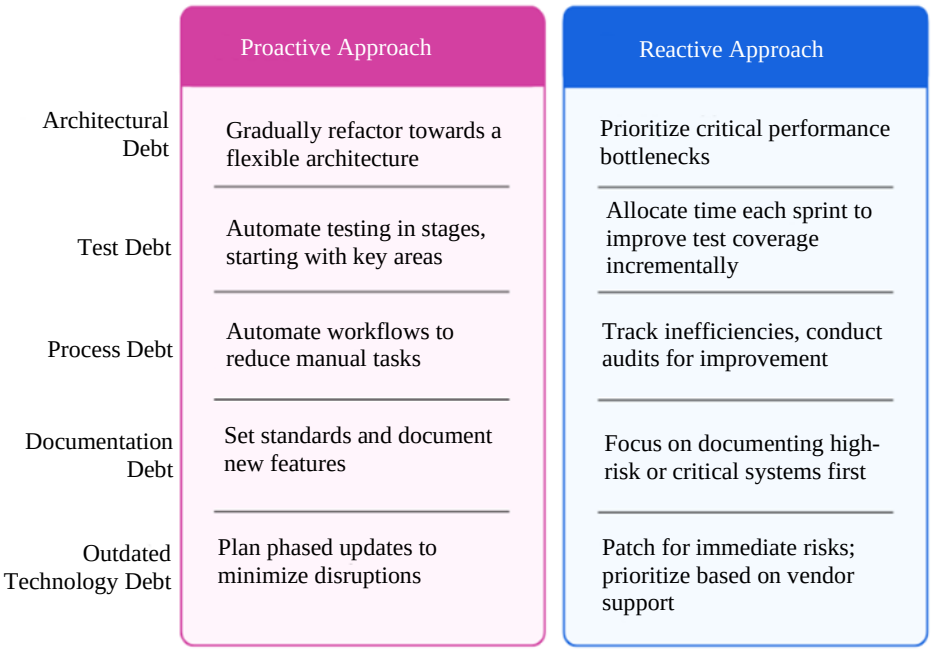



**Figure 7.** Proactive and reactive approaches of managing technical debt.

Over time, as more and more systems are developed on top of others, this leads to a complex architecture that becomes hard to manage and brittle over time, and legacy issues result in major inefficiencies. Since unmanaged debt poses significant risks that must be managed, engineering leadership decides on the right direction for this evolution. Technical debt is usually taken when teams decide about features and faster time to market than keeping the codebase clean and easily maintainable [25]. In the long run, these shortcuts lead to the degradation of architectural designs, meaning that decisions made at architectural levels are later distorted to fit problems not foreseen at design time.

Engineering managers must accept that the technical debt that is not regulated leads to the architectural structure becoming cumbersome, hard to scale, and immobile. As a result, the task of managers is to ensure that the modification of architectural decisions will not negatively affect the system's cost in the future. This often implies making rational decisions between requirements usually valuable to the business in the short term and architectural wellness across numerous systems [5]. If a proactive strategy to address the technical debt is implemented, the system remains flexible, and the structure is adjusted depending on the business needs.

**Engineering Leadership in Architectural Decisions: Guiding Architectural Changes**

Engineering management is crucial in overseeing architectural concerns, particularly where architectural choices are compelled to maneuver amid OS (operating system) improvement missions and short-term goals. Leadership teams should appreciate a close link between technical debt and software architecture since it affects the rate of development and the ability to develop new solutions [26]. Engineering managers must identify tactics that control and steer architectural alterations in compliance with short-term/strategic business goals and the laid-down futuristic course of the chosen system.

Engineering leadership responsibilities in architectural directions involve facilitating the interactions between the involved parties, such as the architects, developers, and product managers. By such approaches, leadership can be assured that technical debt is acknowledged and integrated into decision-making. Implementing an architecture improvement is done with an outlook toward scalability [27]. These discussions help managers direct their teams to define which parts of the system are the most appropriate to refactor and which architecture components need to be rearranged to avoid creating more

technical debt. Managers are also mandated to set the pace for architectural vision and explain how it will get there. This means that only changes to meet long-term objectives must be implemented rather than temporary fixes. It also means that leadership must understand the organization's technical and strategic roles.

**Role of Code Quality Standards and Continuous Refactoring**

One of the typical tasks of engineering management is compliance with code quality standards, which implies further development of the system's architecture. When mitigating technical debt, high-quality code is crucial to implementing and evolving the system in the future. Engineering managers should ensure that they always promote reforms against the accumulation of technical debt through practicing activities such as testing, Code reviews, and documentation. They are good practices that help keep code bases clean, manageable, and easily extensible to accommodate the change in architecture and the need for the new 'horsepowers' in the product as shown in Figure 8.

Another important factor in limited and controlled liability for architectural overgrowth is the strategy of constant refactoring. Refactoring involves altering the source code of a program. At the same time, it is preserved free from any behavioral change. It is necessary throughout the progression of the life cycle of programming to make the program better in design and less sophisticated in programming. Although refactoring is currently pursued in a controlled manner within the engineering organization, the engineering managers must drive the initiative to ensure that its implementation becomes a regular part of the development process [28]. This can be done by integrating refactoring activities into the team's performance and development process so that high software quality is maintained and the architecture of a software system aligns with the organization's growth plan. Continuous refactoring helps engineering leaders keep the technical debt in check and not try to rearchitect too many things at once in the future. This practice also makes it easier to carry forward the codebase to the next set of technologies or patterns when business requirements change.

**Importance of Architectural Reviews**

Architecture reviews are essential to monitoring the health of sustainable architecture, which should be able to adapt to changes over time instead of being trapped in or overwhelmed by heavy debts. These reports offer engineering managers a chance to learn about the state of the architecture and how it is aligned with technology and business objectives within firms. Architectural reviews should be done more frequently to spot any technology built-up and areas where refactoring should be done.

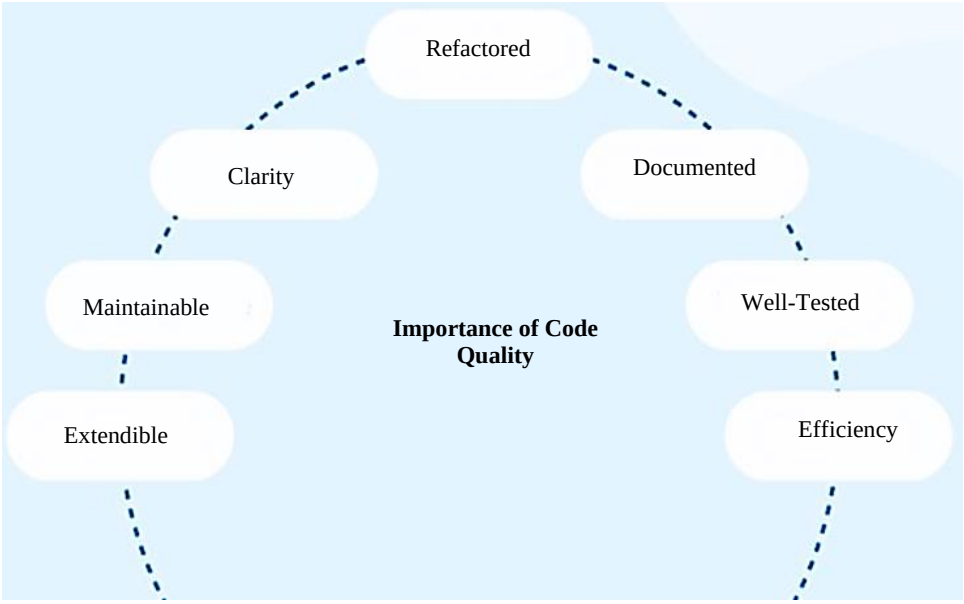



**Figure 8.** Importance of code quality standards.

Structured reviews allow managers to help teams set the correct architectural direction based on probable future requirements instead of solving current issues. These reviews also provide an avenue through which it is possible to ensure that every architectural change is assessed in terms of its impact on performance, scalability, maintainability, and security. Engineering leadership should also receive input from several team members to track whether architectural evolution is on track or not and whether there are any issues with managing technical debt at the early stages.

Architectural reviews are also helpful feedback mechanisms. They let engineering managers know when some of their past decisions have resulted in more technical debt or in options that do not meet the needs of sustaining long-term system architecture. This kind of feedback prepares for subsequent architectural choices and strengthens organizational learning within the engineering team.

## ROLE OF ENGINEERING LEADERSHIP/MANAGEMENT CONCERNING BUSINESS GOALS

### Strategic Alignment of Architecture with Business Objectives

The role of managers and leaders of engineering is to strategically match software architecture with the goal of the business, which is one of its critical duties. If the underlying architectural decisions align with the broader goals of the business, the engineering teams are enabled to deliver the most sustainable and malleable systems. The design and evolution of the architecture involve the engineering manager to ensure that they capture business needs such as growth opportunities, time to market, and differentiation [29]. As much as possible, engineering leads should involve the other side of the business, such as product managers and executives, to determine which architectural decisions would offer maximum value for the business.

Effective technical debt management means that despite the architecture being fully aligned with strategic imperatives, it counts as an effective way and does not prevent the achievement of objectives. Managers have to be on guard about how technical debt grows with regard to business aims and objectives and make sure that immediate and long-term targets are met concurrently with system health.

### Balancing Immediate Business Needs with Long-Term System Sustainability

It is also acknowledged that in satisfying immediate business needs, more emphasis should be put on the further significance that the engineering managers must not also allow system sustainability engines in pursuit of fulfilling the business needs in the short run. Short-term business needs compel organizations to adopt measures that create technical debt that can hinder the system's development in the long run. It is a regrettable necessity for engineering leaders to decide which capability should be delivered first, and nobody wants a system architecture to maintain a long-term ability to adapt and evolve without introducing substantial technical debt as shown in Figure 9.

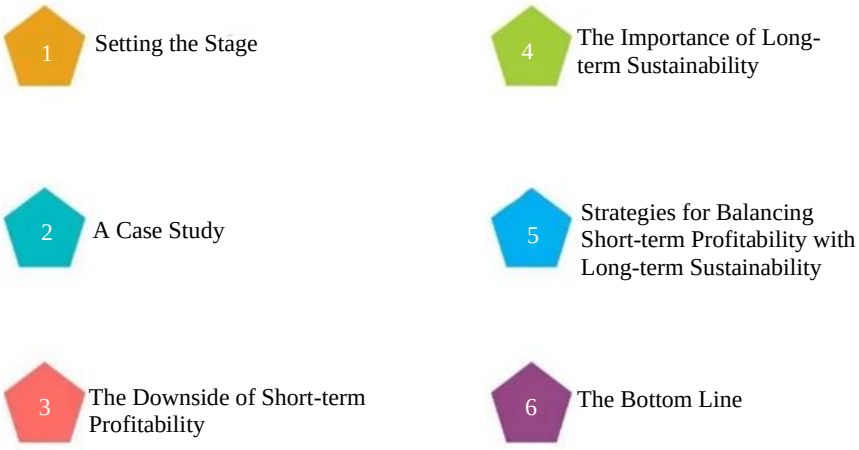



**Figure 9.** Balancing short-term profitability with long-term system sustainability.

Well-intentioned managers who look out for the long-term health of the systems help the organization. Customer expectations and engineering realities also mitigate the tensions between the speed of product delivery and system stability by clearly specifying expectations and resource allocation and promoting alignment between engineering and business functions. The technology of sustainable architecture is significant for the possibility of the development of the technology stack and for managing the adjustment of the organization in terms of a changing market trend without accumulating technical debt.

**Decision Frameworks and Prioritization Techniques**

They are to work with the team to find out how decision-making tools and prioritization of work can be applied to manage technical debt and meet the needs of the business. Engineering managers employ various analytical tools when deciding whether to take technical debt and when to take it, including cost and benefits analysis, risk management, and architectural fitness. These frameworks assist the managers in comparing the possibilities of the consequences of the decisions being made on the business and the system architecture.

Engineering managers must consider technical and business when prioritizing technical debt. It provides help to concentrate on the most important type of debt with the least possible negative effect on business and technical architecture using such tools as MoSCoW (Must have, Should have, Could have, and Will not have) or Value versus Complexity matrix. When used in connection with business objectives, these methods will help managers make informed decisions about what kind of technical debt is acceptable and what might have negative consequences.

**Tracking Debt and Its Impact on Project Health**

Measuring and monitoring technical debt and its relation to project health status are key tasks of engineering management. When organizations expand, technical debt becomes tricky to manage, and if it is not tracked, it might cause severe project dysfunction, expensive maintenance costs, and underperforming systems. Managers across engineering disciplines must design mechanisms to help them identify technical debt levels within a given development project so that this kind of debt is not permitted to grow uncontrolled.

*Possible examples of tracking tools:* A debt register, which belongs to tracking that identifies documented technical debt and prioritizes them. These tools allow groups to keep abreast of the debt within code, infrastructure, and design [30]. It is also good practice for managers to do avid debt checks during code reviews, architecture reviews, and while doing retrospective meetings. In this way, with the necessary transparency of tracking technical debt, engineering leaders can understand the state of project health and make changes, if necessary, when technical debt is still at a smaller scale.

**Metrics and Key Performance Indicators for Managing Technical Debt**

Metric and key performance indicator (KPI) utilization is needed to keep track of technical debt levels and the effectiveness of management actions undertaken by engineering managers. These include code quality measures, such as cyclomatic complexity, code churn, and code coverage, all of which speak to a system's maintainability. Besides, it can be measured using the specific weights related to the frequency of refactoring, the bugs rate, and the frequency of deploying new versions, which will give the idea of the level of technical debt and its capital intensity as shown in Figure 10.

Other motivation metrics for technical debt management may or may not depend on business-oriented factors like time to market new features, customer satisfaction, and systems availability. To a certain extent, these KPIs directive – debt management supports business objectives. Engineering managers must monitor technical and business performance KPIs to understand how technical debt impacts your organization. They guarantee that technical debt does not become an impetus for failure while ensuring a reasonable level of sustainable architecture.

| METRICS | TARGET VALUE |
|---|---|
| Cyclomatic Complexity | The lower, the better |
| Depth of Inheritance | The lower, the better |
| Class Coupling | The lower, the better |
| Lines of Code | The lower, the better |
| Maintainability Index | MS Visual Studio considers values normal in the range of 20..100 but we recommend to target 85..100 range (or 65..85 at minimum) |

**Figure 10.** Metrics for measuring technical debt.

### Balancing Innovation and Long-Term Architectural Health

The delicate thing here is always how to encourage that progress in itself, yet at the same time, work to prioritize the longevity of architectural health. The managers managing engineers need to encourage the teams to try new things that are not necessarily being done in such a manner that new things lead to higher technical debt. Innovation and product development always entertain the thought of implementing solutions that may appear more urgent and less costly in the short run but accrue vast liabilities in the longer run.

Leading in this area includes the culture of constant improvement and refactoring, where innovation can be done but under exemplary architecture. Engineering managers must insist on test-driven development, continuous integration, and code reviews to keep system quality high while facilitating innovation. In this way, engineering managers help the organization satisfy current business requirements with the help of existing system architecture while guaranteeing its sustainable development as the system evolves.

## TECHNICAL DEBT AND AGILE METHODOLOGY

Through this research, agile methodology in software development has been seen to manage technical debt successfully. Agile is inherently professional due to its flexible and iterative processes, which incorporate improvement in each subsequent cycle. It provides structured methods for planning, controlling, and measuring technical debt, together with a set of principles that can be easily implemented in a lean environment. This section details these dimensions with support from theory, empirical studies, and proven best practices.

### Agile Approaches to Reduce Technical Debt over Time

Agile methodology advocates for iterative development; hence, it caters to technical debt, allowing the team to refine suboptimal code and design choices. Iterations or sprints offer control points for dealing with a debt built up over a certain period. Among more essential Agile activities is refactoring, which aims to improve the existing working code. Refactoring is crucial in controlling technical debts because the code should remain close enough to change. During the sprints, the teams can use the 'swarms' to continue refactoring to moderate the propensity to have debts that increase exponentially as shown in Figure 11.

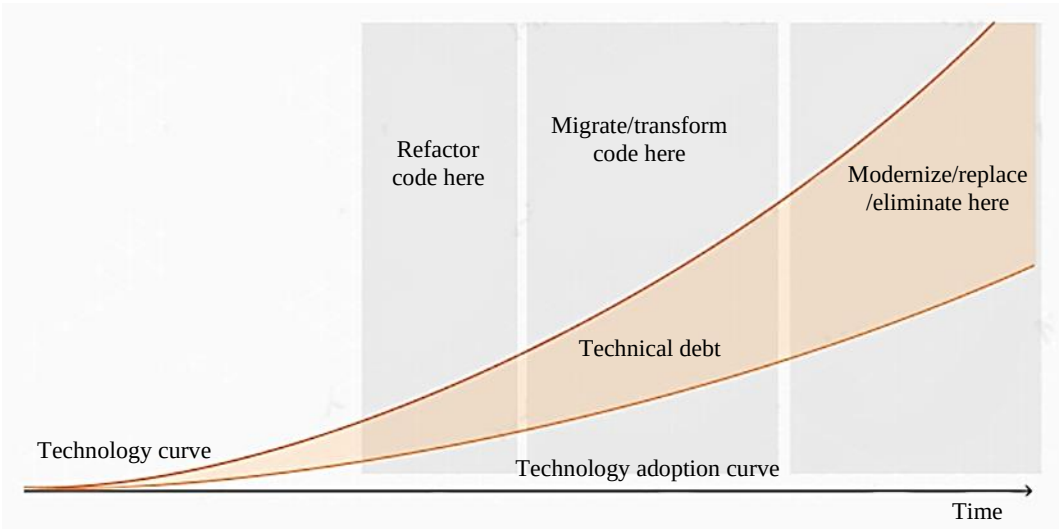



**Figure 11.** Reducing technical debt.

The cross-functional team nature of Agile development helps identify possible debt at a relatively early stage. Significant issues in design and shortcuts are pointed out during design or as planning is underway, and developers, testers, and product owners can easily communicate. Agile heavily adopted user stories, which are tremendously typical in capturing non-functional requirements, which tips the scales between feature delivery and long-term system maintainability. Integrated in that way, the debt management tasks will fit into the Agile backlog, meaning that while teams add new features and capabilities, they will also tackle known problems concurrently [31]. The popular Agile framework Scrum also backs debt reduction since teams are required to reflect on their progress in Scrum retrospective meetings. Such reviews allow teams to discuss technical debt freely and implement it in the upcoming sprints. The inherent practice of daily, weekly, and other schedule feedback in Agile also discourages any likelihood of accruing debt, besides promoting accountability and learning [30].

**Processes and Tools for Monitoring and Managing Debt**

One of the biggest strengths of Agile is the frequent use of numerical indicators and devices to measure work, which applies to the management of technical debt. Debt resolution, on the other hand, may be depicted on a burndown chart and or the cumulative flow diagram used by agile teams to visualize work. By using these tools, managers can pinpoint different areas that are likely to be affected by unpaid debts and, hence, be able to approve appropriate resources.

SonarQube and Checkstyle help estimate the technical debt as applied to Agile processes. These tools deliver third-party, scientifically valid data on code quality and identify sources of debt, such as code flaws, nonadherent coding, and more. Consequently, it can be integrated into sprint planning to assign tasks associated with the problems discovered. Incorporating such tools in CI processes fits well with Agile practices since it will detect technical debt and fix it as it is accumulated.

Safe is a key ingredient of Agile that is used to enhance test automation frameworks like Selenium and JUnit. Automated testing helps monitor product quality and minimize technical debt since defects and discrepancies can be detected during development. These tools allow application developers to get feedback on the state of living code, always catching problems before they worsen. The ability to perform debt management consistently is a positive long-term factor for the system's sustainability, which aligns with the Agile program's goals of delivering sustainable products.

**Integration of Lean and Agile Principles in Debt Management**

The idea that in Lean, most of the value is delivered continuously, without waste, corresponds to Agile's technical debt management. The approach used by proponents of lean thinking deems added or

unnecessary complexity as anathema, and most technical debt is the product of such complexity. Having studied many lean processes, teams can discover areas that lead to debt formation in the approval process by applying a technique like value stream mapping. When adopted in Agile, Lean helps to balance development investment on value delivery while not sacrificing the sustainability of these developments in the future.

Education on the choice of a strategy for technical debt tasks as part of the Lean and Agile hybrid is one method used to integrate the two methodologies. Sometimes, cross-functional agile teams use a board and techniques like Kanban to see work in progress (WIP) to see if there are opportunities to incorporate debt reductions with features. Since Kanban is initiated mostly on a pull basis, it is possible to oppose the changes while preserving the sustainability of the development process. The lean concept titled “building quality in” is in harmony with Agile's view of delivering constant, incremental enhancements [32]. Quality assurance processes have to be integrated into every development phase to minimize the factors that lead to the introduction of new debt. CI/CD pipelines, as one might expect when working in an Agile environment, allow this approach because small changes are tested and released often. This makes it easier to identify possible problems than they get solved, hence decreasing the long-term effects of debt.

Another aspect of Lean and Agile frameworks is encouraging the teams' ownership culture. By adopting and encouraging acidic practices, organizations can also decrease the application rate of “gets the job done” solutions that create technical debt. Championing this mindset leads to better debt management that improves system sustainability. Utilizing Agile methodology in partnership with lean principles is the perfect approach to managing technical debt. Due to its nature of repeating cycles, emphasis on cooperation, and incorporation of instruments helpful in tracking and controlling debts, the teams can solve technical issues while not undermining creativity and system quality. Agile technical debt tasks allow technical debt initiatives to be seamlessly incorporated into Agile while applying lean, eliminating waste, creating a balance that can be productive in the short term, and maintaining system health in the long term as shown in Figure 12.

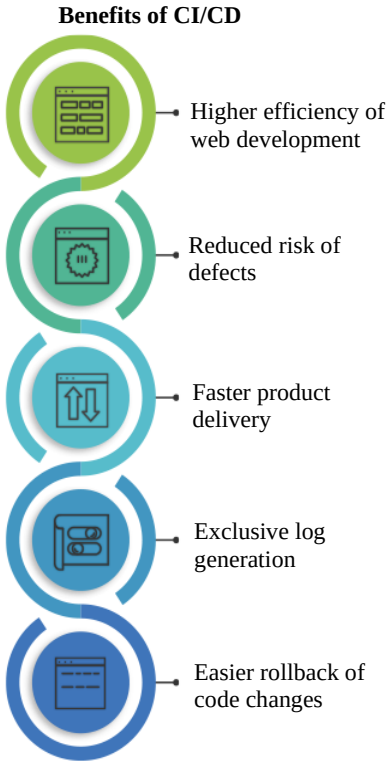



**Figure 12.** Some of the benefits of continuous integration/continuous deployment (CI/CD).

## MISCELLANEOUS CONCEPTS

### Common Pitfalls in Managing Technical Debt

Technical debt management remains one of software development's most important but least defined and least understood aspects. The most basic weakness in managing technical debt is the poor documentation and visibility of the technical debt. Unfortunately, with many projects not having this aspect well addressed through comprehensive tracking systems, it becomes almost impossible for the teams to remember that there is accruing debt that needs to be paid off later on. It is clear that much technical debt ends up being poorly managed due to the lack of metrics and frameworks in place to track it in organizations and, as a result, contributing to more severe maintenance issues and decreased operational performance.

The third risk is cutting sustainable forms of value delivery in favor of short-term tangible deliverables. To save time or meet the set deadlines or when an organization wants to launch new features, most eliminate or reduce servicing of technical debts, which means that the quality of systems is likely to degrade. Systems become inefficient. This short-term thinking poses significant challenges regarding system design for growth and program and policy experimentation.

Another related problem emerges from the lack of understanding of technical debt and the lack of guidance for managing it among engineering departments. When developers and managers fail to understand the effects of technical debt, they may be inclined to make decisions that worsen it. According to Ernst et al. [33], systematic training sessions that raise awareness about technical debt concepts and their consequences can lead to much better long-term training debt results.

The fourth issue is companies' tendency to underfund technical debt repair efforts. As budgets are mainly given to feature development, technical debt has rarely been treated, resulting in long-term low productivity. To avoid these risks, organizations require a preventive approach; this means early identification and documentation of the debts, general employee training, and appropriate dedication of funds for debt resolution.

### Emerging Trends in Engineering Management and Technical Debt

Recent developments in software engineering and management, in particular, have unveiled some radical measures that can be employed to tackle the issue of technical debt. One important trend is a greater incorporation of automated tools for identifying and evaluating liabilities. The addition of SonarQube and the CAST Application Intelligence Platform assures teams of a direct approach to measuring technical debt and visibility of all problematic areas. They not only reveal the problems that exist but also suggest which problems should be fixed first.

Another promising avenue is the debt quantification framework approach. The technical debt conceptual model (TDCM) is applicable for defining and assessing the influence of technical debt on reaching organizational objectives. Frameworks like this empower engineering managers by providing a context in which they can understand and make decisions between delivering features and maintaining systems as shown in Figure 13.

The use of machine learning, including artificial intelligence (AI), has also started to feature in technical debt management. These tools can estimate which part of a business is most prone to debt generation so appropriate intervention can be instituted. For example, AI-driven automated code analysis systems can identify prospective architectural issues, thereby limiting the chances of creating more technical debt. Another emerging trend is coupling technical debt management activities with the agile and DevOps approaches. When organizations adopt iterative development models, integrating debt management practices in those models has been successful in managing long-term risks. CI/CD pipelines, especially those explicitly implemented with technical debt in mind, constantly pay down that debt in portions so that it does not accumulate to be detrimental [34, 35].

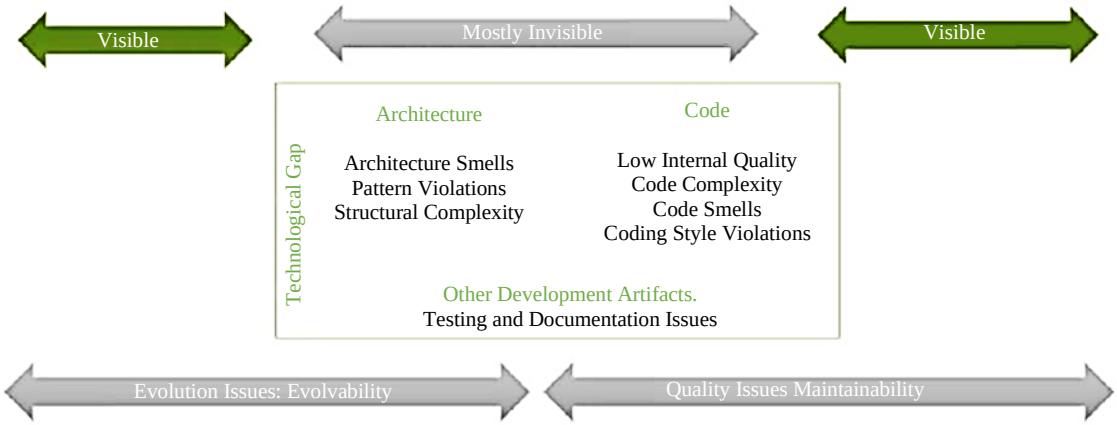



**Figure 13.** The future of managing technical debt.

***Impact of New Technologies***

Recent technologies, including cloud computing, containerization, and microservices, have changed how technical debts are managed. These technologies can reduce debts, but they also bring certain complexities. For instance, cloud-native architecture allows teams to work on preselected parts of technical debt rather than whole systems. They also entail much experience and capital that can be impractical for smaller institutions.

Tools like Docker and Kubernetes are important at the container level. They help teams break down applications into parts so everyone can easily see which areas need technical debt fixing. These ideas add more flexibility and robustness to the overall structure since failure can be contained at this modular level. Adopting such technologies makes creating new technical debt possible. For instance, if dealing with microservices architectures, their management may lead to interdependency and barely manageable for upgrade and maintenance purposes.

Other examples of the dynamic environment of technical debt management are such emerging technologies as blockchain and edge computing. Although these technologies bring new opportunities in capability development, they bring an added system architecture layer and increased requirements for architecture and engineering [36]. Engineering managers staying abreast with these technologies requires constant updating of formal technical education and constantly updating approaches on how to apply these technologies best. The risks and unresolved issues of managing technical debt and new trends need a balanced approach. Through embracing integrated automated tools, quantification frameworks and new technologies organizations can easily manage technical debt as well as ensure the systems are scalable and reliable. It must be the assertion of engineering managers to promote the awareness of technical debt in order for teams to be ready to face the dynamics of today's software project management.

## CONCLUSION

Technical debt still continues to be one of the most significant issues in today’s software engineering process, as it links short-term business objectives with system stability. This article has enlightened me in terms of the way technical debt can be well controlled and managed for architectural progression. By learning the relationship between technical debt and its impacts, engineering managers can do more focused and targeted anticipation of risks, improve team efficiency, and correlate development approaches with the organization’s goals and objectives. The primary tension is between the often-competing goals of the speed of innovation and the delivery of solutions, on the one hand, and, on the other hand, the sustainability and scalability of the established system. The level of performance of organizations required by today’s markets tends to favor short-term profits, resulting in decisions that produce second-rate code, design, or infrastructure, thus creating technical debt. Although these

approaches may help to speed up development, they are detrimental to the system in terms of efficiency, manageability, and flexibility in the long run. When selecting a strategic approach, engineering managers must remember the inherent sacrifices made when such decisions are made and the risks accumulated over time.

Technical debt management is a complex process involving technical and management skills. Tools like SonarQube and the CAST Application Intelligence Platform allow for the precautionary and effective identification of technical debt elements, enhancing the visibility of probable debt areas. Combined with models such as the technical debt conceptual model (TDCM), it allows for prioritizing work according to its influence on organizational objectives and the further impact of technical debt. The possible solutions to the technical debt problem based on new technologies and methods are described. The Agile and DevOps methodologies, in conjunction with CI/CD pipelines, can be used to reduce technical debt in steps as time elapses. It stresses cycles of construction and testing, feedback and refinement, reducing the chances of excessive levels of technical debt. As examples, in the same way, cloud computing, containers, and microservices bring the ability to encapsulate and scale out aggressively technical debt along with other subsystems. They also come with new concerns, for example, the coupling between micro frontend and microservice components that should be addressed. They also found out that the human factor is as important when it comes to technical debt. Most engineering managers usually maintain high technical debt or become heavily involved in advocating for its reduction and ensuring that teams comprehend it and work towards mitigating the problem. Debt management awareness can be fostered through regular training, practical communication, and the incorporation of debt processes and procedures. Engineering managers can incorporate the management of technical debt into the culture to ensure that the efforts of a team are in harmony with business objectives and the creation of value for the organization without compromising the integrity of the systems in use.

Considering future trends, the application of engineering management regarding the interaction with technical and architectural debts will remain one of the growing aspects. This is an important consideration as technologies improve and systems get increasingly complicated, often requiring fresh techniques and strategies to meet new needs. Therefore, by promoting the use of sustainable practices, creating awareness, and ensuring technical decision-making is in harmony with the business objectives, engineering leaders will be able to effectively resolve issues of technical debt as well as architectural evolutions. Managing technical debt goes beyond just a technical concern; it is a critical organizational challenge that directly impacts a company's ability to innovate, expand, and stay competitive. If engineering managers carry out proper strategies, use other available tools, and cultivate a sense of responsibility, technical debt can be a mechanism for change rather than a hindrance. The future of engineering management is to provide such balance by maintaining the strength, flexibility, and coherence of the systems implemented in an organization.

## REFERENCES

1. Bansal A. System to redact personal identified entities (PII) in unstructured data. Int J Adv Res Eng Technol. 2020; 11 (6): 133. doi: 10.34218/IJARET.11.6.133.
2. Bansal A. Deployment strategies to make AI/ML accessible and reproducible. J Artif Intell Cloud Comput. 2022; 1: E179. doi: 10.47363/JAICC/2022(1)E179.
3. Fairbanks G. Just Enough Software Architecture: A Risk-Driven Approach. Boulder, CO, USA: Marshall & Brainerd; 2010.
4. Sas D, Avgeriou P. An architectural technical debt index based on machine learning and architectural smells. IEEE Trans Softw Eng. 2023; 49 (8): 4169–4195.
5. Bansal A. Identifying hallucination in retrieval-augmented generation. Int J Adv Res Eng Technol. 2023; 14 (7): Article 007.
6. Bansal A. Revolutionizing call centers through ASR and advanced speech analytics. J Artif Intell Cloud Comput. 2022; 1: E178. doi: 10.47363/JAICC/2022(1)E178.

7. Behutiye WN, Rodríguez P, Oivo M, Tosun A. Analyzing the concept of technical debt in the context of agile software development: a systematic literature review. Inform Softw Technol. 2017; 82: 139–158.
8. Gill A. Developing a real-time electronic funds transfer system for credit unions. Int J Adv Res Eng Technol. 2018; 9 (1): 162–184.
9. Codabux Z, Williams B. Managing technical debt: an industrial case study. In: 2013 4th International Workshop on Managing Technical Debt (MTD), San Francisco, CA, USA, May 20, 2013. pp. 8–15.
10. Bijekar S, Padariya HD, Yadav VK, Gacem A, Hasan MA, Awwad NS, Yadav KK, Islam S, Park S, Jeon BH. The state of the art and emerging trends in the wastewater treatment in developing nations. Water. 2022; 14 (16): 2537.
11. Nyati S. Revolutionizing LTL carrier operations: a comprehensive analysis of an algorithm-driven pickup and delivery dispatching solution. Int J Sci Res. 2018; 7 (2): 1659–1666.
12. Nyati S. Transforming telematics in fleet management: innovations in asset tracking, efficiency, and communication. Int J Sci Res. 2018; 7 (10): 1804–1810.
13. Krasner H. The Cost of Poor Software Quality in the US: A 2020 Report. Boston, MA, USA: Consortium for Information & Software Quality; 2021.
14. Köhler A. End-of-Life Implications of Electronic Textiles – Assessment of a Converging Technology. MSc Thesis. Lund, Sweden: Lund University; 2008.
15. Rios N, de Mendonça Neto MG, Spínola RO. A tertiary study on technical debt: types, management strategies, research trends, and base information for practitioners. Inform Softw Technol. 2018; 102: 117–145.
16. Shamsi JA, Khojaye MA. Big Data Systems: A 360-Degree Approach. London, UK: Chapman and Hall/CRC Press; 2021.
17. Vernon V, Jaskula T. Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture. Boston, MA, USA: Addison-Wesley Professional; 2021.
18. Purushothaman MB, Seadon J, Moore D. A relationship between bias, lean tools, and waste. Int J Lean Six Sigma. 2022; 13 (4): 897–936.
19. Kumar A. The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. Int J Comput Eng Manage. 2019; 6 (6): 118–142.
20. Mihelcic JR, Zimmerman JB. Environmental Engineering: Fundamentals, Sustainability, Design. Hoboken, NJ, USA: John Wiley & Sons; 2021.
21. Smith PG, Reinertsen DG. Developing Products in Half the Time: New Rules, New Tools. Hoboken, NJ, USA: John Wiley & Sons; 1997.
22. Wiese M, Rachow P, Riebisch M, Schwarze J. Preventing technical debt with the TAP framework for technical debt aware management. Inform Softw Technol. 2022; 148: 106926.
23. Taulli T. The Robotic Process Automation Handbook: A Guide to Implementing RPA Systems. New York, NY, USA: Apress; 2020.
24. Yang Y, Verma D, Anton PS. Technical debt in the engineering of complex systems. Syst Eng. 2023; 26 (5): 590–603.
25. Bansal A. Optimizing RAG with hybrid search and contextual chunking. J Emerg Appl Sci Technol. 2023; 5: E114. doi: 10.47363/JEAST/2023(5)E114.
26. Bansal A. Energy conservation in mobile ad hoc networks using energy-efficient scheme and magnetic resonance. J Netw. 2015; 3 (Special Issue): 15. doi: 10.11648/j.net.s.2015030301.15.
27. Putnik G, Sluga A, ElMaraghy H, Teti R, Koren Y, Tolio T, Hon B. Scalability in manufacturing systems design and operation: state-of-the-art and future developments roadmap. CIRP Ann. 2013; 62 (2): 751–774.
28. Özkan O, Babur Ö, Van Den Brand M. Refactoring with domain-driven design in an industrial context: an action research report. Empirical Softw Eng. 2023; 28 (4): 94.
29. Ross JW, Weill P, Robertson D. Enterprise Architecture as Strategy: Creating a Foundation for Business Execution. Boston, MA, USA: Harvard Business School Press; 2006.
30. Morris K. Infrastructure as Code. Sebastopol, CA, USA: O'Reilly Media; 2020.

31. Moran A. Managing Agile. Strategy, Implementation, Organisation and People. Cham, Switzerland: Springer; 2015.
32. Ng PL, Maqsood T, Khalfan M, Rahmani F. AgiBuild: a scaled Agile framework for building adaptation projects. Buildings. 2023; 13 (12): 3019.
33. Ernst N, Kazman R, Delange J. Technical Debt in Practice: How to Find It and Fix It. Cambridge, MA, USA: MIT Press; 2021.
34. Chintale P. DevOps Design Pattern: Implementing DevOps Best Practices for Secure and Reliable CI/CD pipeline. English Edition. New Delhi, India: BPB Publications; 2023.
35. Preciado Orozco S. Technical Debt Analysis and Project Architecturization of a Jenkins Platform Based on Groovy. Barcelona, Spain: Bachelor's Thesis. Universitat Politècnica de Catalunya; 2022.
36. Banafaa M, Shayea I, Din J, Azmi MH, Alashbi A, Daradkeh YI, Alhammadi A. 6G mobile communication technology: requirements, targets, applications, challenges, advantages, and opportunities. Alexandria Eng J. 2023; 64: 245–274.