**Technical Debt and Architectural Decay: A Comparative Study in Microservices**

**Supriya Surabhi**

SupriyaSurabhi@lewisu.edu

Lewis University

Software Archiecture and Design, CPSC,61200

Fall 2024

# ABSTRACT

The performance, maintainability, and scalability of software systems are greatly impacted by technical debt and architectural degradation, especially in monolithic and microservices architectures. Emprical investigation of six actively maintained GitHub repositories—three monolithic and three microservices—are analyzed in this study to examine the connection between technical debt and architectural deterioration. To find signs of technical debt and architectural smells, object-oriented design metrics including LCOM, WMC, and CBO are assessed using the CKMetrics tool and code smells are detected by JDeodrant. Case studies and developer interviews are used to add qualitative insights to the trend visualization of these metrics across time. The results seek to measure the development of technical debt and its consequences while providing useful suggestions for reducing architectural deterioration and enhancing software quality.

**Keywords**

Emprical investigation, Technical Debt, Architectural Deterioration, Monolithic Systems, Microservices Architecture, CK Metrics, Maintainability, Software Quality.

# 1 INTRODUCTION

A popular paradigm in contemporary software development, microservices architecture allows businesses to create modular and scalable applications. Agility and quicker delivery cycles are encouraged by this architecture, but it also has drawbacks, particularly technical debt and architectural deterioration, which can impair system performance and maintainability over the long run [1][3]. Microservices' distributed architecture and intricate interdependencies make technical debt—a metaphor for the price of making less-than-ideal choices in software development. [1] [2].

Unmanaged technical debt is frequently the cause of architectural decay, which is defined as the progressive loss of a system's design integrity [2][6]. Frequent modifications, a lack of team communication, and the absence of a cohesive design philosophy all contribute to architectural deterioration in microservices [4][6]. To quantify, control, and lessen these issues' effects on software quality and operational effectiveness, methodical methodologies are needed [5][7].

To fight technological debt and decay in microservices, existing research highlights the significance of implementing architectural patterns and continuous delivery approaches [6][8]. For software practitioners, however, striking a balance between immediate delivery demands and long-term architectural integrity continues to be a constant issue [7][8]. With an emphasis on methods to lessen their effects and preserve sustainable software architectures, this paper investigates the relationship between technical debt and architectural degradation in microservices.

**2 BACKGROUND AND RELATED STUDY**

**Introduction**

This study's theoretical foundations examine two interconnected ideas, architectural decay (AD) and technological debt (TD), particularly in relation to microservices architecture (MSA). Ward Cunningham originally used the term "technical debt" in 1992 [8]. Cunningham employed it as a metaphor to explain to stakeholders, both technical and non-technical, the need for code reworking. Originally limited to coding methods, the idea of TD has since broadened to include requirements, testing, and architectural debt—all of which take time and money away from the software development lifecycle (SDLC) [9].

Comparably, architectural deterioration describes how accumulated TD, and a lack of proactive design maintenance gradually erode a system's structural integrity [4]. Because microservices are distributed and necessitate smooth team coordination and constant adherence to design rules, this issue is especially severe in microservices [5]. These ideas and their ramifications for software projects are covered in more detail in the sections that follow.

**Technical Debt**

In software design or implementation, technical debt (TD) occurs when poor choices lead to long-term inefficiencies that require future rework [8]. If the debt is not "repaid" through redesign or refactoring, these inefficiencies, which are figuratively called "interest," mount up over time. A single, excessively complicated module, for instance, that developers are afraid to alter for fear of disrupting functioning, might greatly impede the rollout of new additions, hence increasing the debt [8].

Every second spent on badly written code raises the interest on technical debt, which, if not controlled, can ultimately cause a project to fail, as Ward Cunningham rightly pointed out.

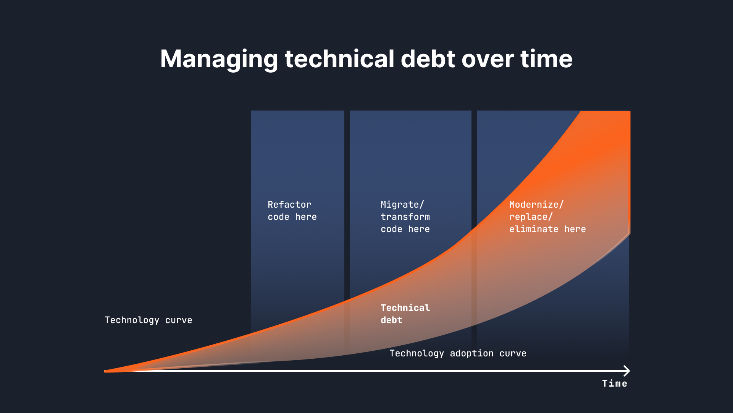
**Types of Technical Debt**

Code, architecture, and requirements are just a few of the software artifacts that are impacted by the various types of technical debt (TD). Studies by Alves et al. (2014) and Li et al. (2015) have found 13 and 10 forms of TD, respectively. The two most often mentioned varieties are architectural TD (ATD), which refers to less-than-ideal architectural choices, and code TD, which deals with problems in the software's source code. Defect TD (problems found during testing), design TD (complexity in design), documentation TD (inadequate documentation), and infrastructure TD (inadequate infrastructure selections) are other important categories. Process TD (inefficiencies in development processes), people TD (socio-technical decisions), and service TD (associated with web service selection in cloud-based systems) are less frequently researched types. The range of these debt forms demonstrates the complexity of technical debt in software development, and each one calls for a different approach to detection, tracking, and handling.

**Calculating Technical Debt**

TD is calculated by calculating the additional work needed for maintenance or adaptation (interest) and the expense of correcting less-than-ideal design choices (principal). This cost may be increased for microservices due to the intricacy of the interconnections between dispersed services [7]. The following is an expression for the TD formula:

**TD=Principal+Interest**



***Figure 1: Managing Technical Debt Over Time***

There is an urgent need for intervention when the total TD exceeds the cost of refactoring. Teams should prioritize efforts to lessen the impact of excessive TD. The phases of resolving technical debt—from reworking code to replacing or removing it—as well as the related increase in technology adoption curves are depicted in Figure 1.

**Architectural Decay**

When a system's design departs from its intended structure because of iterative adjustments, inadequate documentation, or insufficient governance, architectural decay happens [2][6]. Such deterioration is particularly likely in microservices, where services are created individually, because of rapid modifications, inconsistent implementation styles, and a lack of established communication protocols.

For example, using a single "god class" to handle cross-service interactions may make design simpler at first, but it can result in several refactoring failures. Delivery schedules are thrown off, and TD rises as a result [9]. To reduce architectural deterioration, architects must balance the trade-offs between short-term solutions and long-term design excellence.

* Architects can use tactics like these to stop architectural deterioration:
* Periodic design evaluations to determine whether architectural principles are being followed.
* Utilizing technologies that detect anti-patterns or code smells to automate quality tests.
* Establishing uniform design and documentation procedures for all teams.

**Related work**

* [4] examined the ways in which TD in microservices varies from conventional monolithic designs, pointing to a major obstacle as greater complexity.
* To reduce architectural decay, [6] looked at automated TD tracking systems and suggested integrating them into CI/CD pipelines.
* [7] hasoffered case studies showing how businesses were able to successfully strike a compromise between long-term architectural quality and delivery challenges.

# 3 METHOD AND IMPLEMENTATION

# Research Goal

This study's main objective is to investigate and comprehend the relationships between architectural deterioration (AD) and technical debt (TD) in various software architectures, with a focus on monolithic and microservices systems. Using object-oriented design metrics like LCOM (Lack of Cohesion of Methods), WMC (Weighted Methods per Class), and CBO (Coupling Between Objects), the study looks at how these two factors change over time to find quantifiable signs of technical debt and architectural smells.

Quantitative analysis of software repositories and qualitative data from developer interviews and case studies are used in the study to:

* Examine how will technical debt effect the software's long-term scalability, and maintainability in both microservices and monolithic architectures.
* Identify the trends and patterns that cause technical debt to accumulate and architectural decay to start.
* Provide recommendations for best practices to control technical debt and prevent architectural degradation for both types of systems.

This study attempts to provide practical insights for enhancing software design and architecture management by contrasting how various architectural styles manage and accrue technical debt. The goal of the research is to improve knowledge and management of software systems' long-term health.

**Research Questions**

**RQ1**: What connection exists between architectural deterioration (AD) and technical debt (TD) in microservices as opposed to monolithic systems?

RQ1 explores the relationship between TD and AD in both architectures with the goal of locating degradation trends and important contributing elements in monolithic and microservices systems.

**RQ2**: How can technical debt and architectural smells in software repositories be measured using object-oriented design metrics like LCOM, WMC, and CBO?

To get insight into software quality and possible regions of architectural decay, RQ2 investigates how design metrics such as LCOM, WMC, and CBO can quantify technical debt and architectural smells.

**RQ3**: How can unmanaged technical debt and architectural deterioration affect software performance, scalability, and maintainability over the long run in both monolithic and microservices architectures?

By contrasting their effects in microservices and monolithic systems, RQ3 investigates the long-term effects of unmanaged TD and AD on system performance, scalability, and maintainability.

**Research Process**

The purpose of this study is to examine and measure the effects of architectural deterioration (AD) and technical debt (TD) in various software architectures, with a focus on monolithic and microservices systems. We used a mix of automated tools, code analysis metrics, and qualitative research techniques, such as developer interviews and case studies, to investigate this. The research methodology and the implementation process are described in the steps that follow.

**Phase1: Repository Selection and Data Collection**

Six actively maintained GitHub repositories covering both monolithic and microservices architectures have been chosen for this study. Spring Pet Clinic [10] and SiteWhere[12] are two projects that started out with a monolithic architecture before switching to microservices. I have selected distinct branches for the monolithic and microservices versions of these repositories for a thorough comparison. This makes it possible to examine in detail how the architecture changed over time and how this migration affected the software's scalability and maintainability.

I have included the Moduliths [11] repository, which still uses a monolithic architecture, to broaden my research even more. This project offers a different viewpoint from the migrating initiatives by shedding light on the continued difficulties and benefits of keeping a monolithic system. Finally, to have a better understanding of the features and intricacies of a system built with microservices from the beginning, I have chosen the Eventuate Tram Example Customers and Orders [13] repository, a microservices-based application**.**

This collection of repositories offers a comprehensive overview of the transition from monolithic to microservices architectures, providing insightful information about the difficulties of migration, the benefits of microservices, and the continued applicability of monolithic systems in specific situations. The study intends to investigate the architectural trade-offs, performance consequences, and long-term sustainability of both approaches by looking at these projects.

**Phase 2: Code Analysis Using CKMetrics**

Object-oriented design metrics were used to find indications of architectural deterioration and technical debt in the chosen repositories. These metrics were selected because of their capacity to draw attention to possible problems with architectural design and code quality. Three main metrics were calculated using the CKMetrics tool [14]:

Lack of Cohesion of Methods, or LCOM, is a metric that assesses how tightly a class's methods relate to one another. Poor class coherence, which is frequently a sign of code debt, is generally indicated by a high LCOM value. A class with low cohesiveness may be more difficult to maintain and eventually be more vulnerable to the introduction of technical debt.

Weighted Methods per Class, or WMC, is a metric that counts a class's methods and accounts for its complexity. Higher WMC values, a common sign of accumulated technical debt, imply that a class may be unduly complex and challenging to maintain.

Coupling Between Objects, or CBO, is a statistic used to evaluate how dependent one class is on another. When a class is modified, a high CBO suggests that classes are closely related, which may result in cascading changes. This close linkage can worsen technical debt, especially in bigger systems, and is frequently an indication of architectural deterioration.

We were able to monitor the development of technical debt and pinpoint certain times when architectural smells or code quality deteriorated by computing these metrics. This analysis sheds light on the differences between monolithic and microservice architectures and offers a methodical way to comprehend how technical debt develops within software systems.

**Phase 3: Case Studies and Developer Insights**

**1.Netflix: Transitioning from Monolith to Microservices**

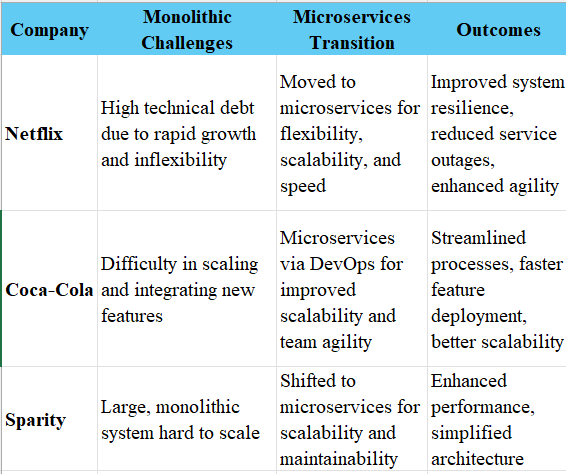
Netflix's shift from a monolithic design to microservices is among the most well-known case studies in the tech sector [15]. Initially, Netflix's monolithic structure caused a bottleneck as it grew to meet the increasing demand. The business had to deal with issues like sluggish deployments, system outages, and trouble managing massive volumes of data. Netflix was able to increase system dependability, increase deployment flexibility, and extend its services independently by switching to microservices. This change was essential for lowering technological debt and enhancing system maintainabilitygenerally, which helped Netflix better handle its expanding user base**.**

**2. Sparity: Monolithic to Microservices**

The company that moved from a monolithic to microservices architecture to address performance bottlenecks and complicated, difficult-to-maintain code is the subject of Sparity's case study [16]. The requirement for more scalability and flexibility prompted the change. Sparity was able to isolate malfunctions and enhance system maintainability by utilizing microservices. Improved fault tolerance and the capacity to upgrade specific parts without impacting the system were further advantages for the business. By streamlining the entire architecture and making upgrades simpler, the business was able to lower its technical debt through the adoption of microservices.

**3. Coca-Cola: Microservices for Scalability and Flexibility**

Coca-Cola was able to manage its extensive and variedproduct line more effectively after switching to microservices architecture [17]. Coca-Cola required an architecture that could accommodate geographically scattered and diversified teams while ensuring high availability and performance because of its global presence. To decentralize its systems and facilitate the scaling of individual components, the organization implemented microservices. Faster response times and more efficient use of resources were made possible by this move, which addressed the previous system's increasing technical debt and architectural complexity. Coca-Cola was able to maintain its competitiveness in the digital era by implementing microservices, which increased system flexibility and agility.

****

***Figure 2: Key Insights: Monolith vs Microservices in Addressing Technical Debt and Architectural Decay***

With an emphasis on technical debt and architectural degradation, Figure 2 offers a comparative analysis of the difficulties, solutions, and advantages seen by several enterprises transitioning from monolithic to microservices architectures. These case studies highlight lessons learned in controlling technical debt over time and provide insightful information about how both architectural approaches are applied in the real world.

**Developer Insights**

To shed the spotlight on the long-term effects of technical debt (TD) and architectural deterioration (AD) in both monolithic and microservices architectures, a series of interviews with a software architect and a developer were undertaken for this study. In addition to collecting useful qualitative data to support the quantitative components of this study, the purpose of these interviews was to investigate the practical consequences of TD and AD in actual systems.

The research questions (RQ1, RQ2, and RQ3) were carefully matched with the interview questions, which addressed topics like the connection between AD and TD, the measurement of technical debt using object-oriented design metrics, and the long-term impacts of unmanaged technical debt on system performance, scalability, and maintainability.

**Developer Interviews**

Experienced software developers and Software Architects who have worked with both monolithic and microservices architectures participated in a series of qualitative interviews. The purpose of these interviews was to get direct insight on how architectural deterioration (AD) and technical debt (TD) affect various system architectures. Throughout the lifecycle of both monolithic and microservice-based applications, the developers were questioned about their experiences with controlling TD, seeing architectural smells, and resolving the difficulties presented by AD.

**Interview Process**

Developers and architects with experience in both monolithic and microservices architectures participated in semi-structured interviews for the study. A fixed set of questions were asked and various ascpects of developers and architects are noted down. To better understand technical debt (TD) and architectural deterioration (AD), these developers were asked to discuss the difficulties they encountered and the methods they used to deal with these problems in various system types. The talks offered insightful viewpoints on how TD builds up and is handled in monolithic versus microservices systems, as well as how AD affects performance, scalability, and maintainability.

The developers also examined the trade-offs between the scalability and maintainability of microservices versus monolithic structures. They also discussed their migration experiences, focusing on the difficulties they encountered when switching from monolithic to microservices, especially about code quality, teamwork, and service integration. These revelations highlighted the intricacy of these migrations and the significance of early technical debt management for a seamless transition and long-term system architecture. The questionnaire and answers have been uploaded to the repository on GitHub [18]. The full set of questions and insights provided for both developers and architects is available there for further review and analysis.

Key themes were found and grouped into pertinent categories after a thematic analysis of the qualitative data from the developer interviews. One of the main themes was the difficulties in managing technical debt (TD), particularly in monolithic systems where the architecture is less flexible and more rigid, which causes TD to accumulate more quickly. The strategies for minimizing architectural degradation (AD) were another important issue, with developers discussing methods including modularization, code restructuring, and switching to microservices to increase maintainability. Themes of performance, scalability, and flexibility finally surfaced, emphasizing how microservices designs generally enhanced these elements by providing superior fault separation and scalability. On the other hand, monolithic systems occasionally offered a simpler solution in specific circumstances. These observations highlight the difficulties developers have as they design software systems and deal with related technical issues.

**Phase 4: Data Analysis and Interpretation**

**Analysis and Comparative Study**

The study's subsequent stage will entail combining the qualitative information gathered from developer interviews with the quantitative data gathered from CKMetrics. A clear image of how each architecture impacts code quality, scalability, and performance will be provided by the research by comparing the computed metrics (LCOM, WMC, and CBO) across monolithic and microservices architectures. Patterns pertaining to the building of technical debt and the deterioration of architecture over time will be found through this investigation. Monitoring the growth of monolithic systems as they shift to microservices—as demonstrated by repositories like SiteWhere and Spring Pet Clinic—will be a major area of attention. The project will investigate the effects of this migration process on performance and maintainability, providing information on how technical debt arises and how it may be controlled throughout such transitions.

**Sustainability and Developer Insights**

In the second section of the study, case studies such as Netflix, Coca-Cola, and Sparity will be used to examine the long-term feasibility of both monolithic and microservices designs. The trade-offs of the two architectures in terms of performance, scalability, and maintenance costs will be seen from a practical standpoint. The study will look at whether switching tomicroservices has more advantages thandisadvantages, especially when it comes to handling technological debt. Insights into how developers address problems like tight coupling, code duplication, and modularity in both design types will be gathered through more follow-up interviews. In addition to a forward-looking examination of new architectural trends and their effects on technical debt management, this will result in actionable suggestions for preserving software quality.

**4 RESULTS**

In this section we report and discuss the data gathered to answer our RQ questions

**Results RQ1:** What connection exists between architectural deterioration (AD) and technical debt (TD) in microservices as opposed to monolithic systems?

***Figure 3: CBO, WMC, LCOM trends in Monolithic vs Microservices***

Along with the ck metrics I have found the code smells for all the six projects using Jdeodrant [19] are results are presented below. As shown in Figure 3 strong interdependencies between classes are reflected in the higher average CBO (6.75) in monolithic systems, which results in code smells like Cyclic Dependencies, Shotgun Surgery, and Feature Envy. Because tightly connected classes make updates more time-consuming and prone to errors, these smells worsen architectural decay. Because ripple effects across interconnected modules necessitate significant work during refactoring or feature additions, this increases technical debt.

Monoliths with higher WMCs (5.17) have more complex methods, which contributes to smells like Data Clumps, Long Method, and God Class. These smells lead to big, multipurpose methods or classes in designs that are hard to maintain. Monoliths frequently suffer from Blob Classesand Divergent Changes, where unrelated responsibilities inside a class worsen maintainability and raise technical debt, in addition to their low cohesiveness (LCOM of 15.71).

The reduced WMC (3.99) and CBO (6.36) in microservices indicate improved modularity and less complicated techniques, which lessen some forms of architectural degradation. Inter-service communication, however, creates distributed dependencies that add to technological debt by introducing smells like Message Chains and Inappropriate Intimacy. As demonstrated by smells like Speculative Generality, over-engineering or superfluous abstractions can still lead to service complexity.

Although microservices are more cohesive than monoliths (LCOM of 13.74), Large Service smells might result from service bloat or unclear responsibilities. By requiring more maintenance, this type of architectural deterioration makes service boundaries more difficult to manage and adds to technical debt. Due to their unique architectural and design characteristics, these architectures display several types of ADS and TD.

**Results RQ2:** How can technical debt and architectural smells in software repositories be measured using object-oriented design metrics like LCOM, WMC, and CBO?

***Figure 4: Distribution of CK metrics in Monolithic vs Microservices***

**LCOM (Lack of Cohesion of Methods):** LCOM measures how closely related the methods in a class are. A high LCOM indicates low cohesion, suggesting that a class is responsible for unrelated tasks, which can lead to God classes or feature envy. These classes are more difficult to maintain, understand, and test, which directly contributes to increased technical debt.

**Insights from Data:** As shown in Figure 4Spring Pet Clinic (Monolithic) has a much higher LCOM value (3.28) than Spring Pet Clinic (Microservice) (0.91). This indicates that the monolithic architecture is likely to suffer from low cohesion in its classes, which could result in God classes or feature envy, adding complexity and maintenance challenges. In contrast, the Microservice architecture shows a lower LCOM, suggesting that its classes are more cohesive and thus easier to maintain and modify.

Similarly, SiteWhere (Monolithic) and SiteWhere (Microservice) have identical LCOM values (37.44), which is quite high. This suggests that the monolithic version of SiteWhere suffers from significant architectural smells, including potential god classes and low cohesion across methods. This higher LCOM indicates more potential for technical debt in the monolithic architecture.

LCOM reveals that monolithic architectures are prone to higher technical debt due to low cohesion in their classes, while microservice architectures tend to exhibit more cohesive classes, leading to a more maintainable codebase with lower technical debt.

**WMC (Weighted Methods per Class):** WMC measures the complexity of a class by counting the total number of methods and their complexities. A high WMC value indicates that a class is likely complex, difficult to understand, and harder to maintain, thus contributing to increased technical debt.

**Insights from Data:** From Figure 4 we can observe thatSpring Pet Clinic (Monolithic) has a much higher WMC value (4.85) compared to Spring Pet Clinic (Microservice) (1.55). This shows that the monolithic version is likely to contain more complex classes, leading to difficulties in maintenance and higher technical debt. The Microservice version, with its smaller, simpler classes, is expected to have less complexity and, therefore, reduced technical debt.

The SiteWhere projects both exhibit very high WMC values (7.17), suggesting that both the monolithic and microservice versions of SiteWhere have complex classes. This may point to potential bottlenecks in maintenance and higher technical debt due to the complexity of individual classes.

High WMC values in both SiteWhere projects indicate that these systems, regardless of architecture, are complex and difficult to maintain. However, the monolithic architecture tends to have even higher values in comparison to the microservice architecture in other projects (e.g., Spring Pet Clinic), which adds to its technical debt.

**CBO (Coupling Between Objects):**

CBO measures the extent to which classes are dependent on each other. A high CBO suggests tight coupling, which leads to “dependency hell,” making maintenance and future changes difficult. This is a sign of increased technical debt.

**Insights from Data:** The Spring Pet Clinic (Microservice) project has a lower CBO (6.53) than the Monolithic version (7.78) as shown in Figure 4, indicating less coupling between classes. This means that changes to one class in the microservice architecture are less likely to impact other parts of the system, resulting in easier maintenance and lower technical debt.

SiteWhere exhibits the same CBO values (7.69) for both its microservice and monolithic versions, suggesting that the coupling between classes is high in both architectures. This high coupling is indicative of dependency hell, where changes in one class require modifications to many other classes, leading to significant maintenance challenges and increased technical debt.

High CBO values, particularly in SiteWhere, suggest that the system is tightly coupled, which contributes to increased technical debt. The Spring Pet Clinic project, with lower coupling in the microservice version, benefits from easier refactoring and maintenance, leading to less technical debt in comparison.

To Conclude, the object-oriented design metrics LCOM, WMC, and CBO are effective tools for measuring technical debt and identifying architectural smells in software systems. The analysis of the Spring Pet Clinic and SiteWhere projects across microservice and monolithic architectures provides clear insights into how these metrics reflect the system's quality and maintainability.

* LCOM shows that the microservice architecture tends to have more cohesive classes, reducing the likelihood of God classes and feature envy, which are common sources of technical debt.
* WMC reveals that classes in monolithic systems tend to be more complex, contributing to higher technical debt, while microservices are simpler and easier to maintain.
* CBO indicates that monolithic systems and systems with high coupling face more significant challenges in terms of maintenance and refactoring, leading to higher technical debt.

In conclusdebt andg LCOM, WMC, and CBO metrics helps software engineers detect architectural flaws early, quantify technical debt, and improve the overall maintainability of systems by identifying areas that require attention.

**Results RQ3:** How can unmanaged technical debt and architectural deterioration affect software performance, scalability, and maintainability over the long run in both monolithic and microservices architectures?

Unmanaged technical debt (TD) and architectural deterioration (AD) can seriously impair the long-term performance, scalability, and maintainability of systems, regardless of whether they use monolithic or microservices architectures, as demonstrated by my interviews with architects and real-world case studies.

**Monolithic Architecture**: The tight interdependence between parts in monolithic systems causes technical debt to mount up quickly. This happens because of cascade changes in the closely knit software, where altering one component frequently affects other components. The system becomes more inflexible as a result, and adding new features or repairing flaws becomes more expensive over time. This speeds up architectural degradation (AD), which causes scalability issues and performance snags.

Unmanaged TD and AD's effects on Monolithic Projects:

* **Performance**: It gets more difficult to rework badly structured code and optimize inefficient algorithms as the codebase expands. Because of the interdependencies among the components, each modification may have an impact on the system, making it challenging to enhance performance without creating new problems.
* **Scalability**: Because adding new features necessitates changing the entire software, monolithic systems are more difficult to scale. Longer development cycles and trouble managing more users or traffic result from this.
* **Maintainability**: The system gets more difficult to maintain as technical debt increases. Instead of adding new features, developers spend more time debugging and resolving problems, which slows release cycles and lowers productivity.

**Microservices Architecture:**Teams can isolate and manage technological debt at the service level in microservices systems due to the design's increased flexibility and modularity. The intricacy of overseeing distributed systems, data consistency, and inter-service connectivity, however, can cause architectural degradation if left unchecked. If these difficulties are not resolved, they may lead to problems with scalability and performance that are comparable to those of monolithic systems.

Unmanaged TD and AD's effects on Microservices systems:

* **Performance**: Response time delays and bottlenecks can result from ineffective service-to-service communication and inconsistent services brought on by mismanaged debt. It gets more difficult to make sure services are optimized and operating properly as they expand.
* **Scalability**: Although microservices allow for service-level scalability, technical debt can make service scaling more difficult, particularly when managing database consistency or inter-service communication.
* **Maintainability:** As distributed services grow increasingly challenging to monitor and update, the intricacy of managing them without efficient refactoring might eventually result in high maintenance expenses, deployment delays, and an increase in technological debt.

**Survey Insights on Managing Technical Debt and Architectural Deterioration:**

I learned a lot from the architects where I spoke with regarding the impact that architectural decay and technical debt play in both monolithic and microservices systems. The key insights from the survey and interview are presented below in Figure 4.



***Figure 4: Key Insights from the Developers and architects***

# 5 THREATS TO VALIDITY

This study recognizes several risks to its validity that may affect the dependability and applicability of its conclusions:

**1)Internal Validity**

* **Metric Selection Process**: The selection of LCOM, WMC, and CBO as the main metrics might not fully account for all facets of architectural deterioration and technological debt. Other measurements may show different trends or insights.
* **Tool Dependency**: Since CKMetrics might not fully support the distinctive features or structures of all chosen repositories, relying solely on it could add tool-specific limits.
* **Sampling bias**: The six repositories examined in the study might not be entirely representative of the software project landscape. Projects that are bigger or more varied may provide different outcomes.
* **Developer Interviews**: The integrity of the qualitative data may be impacted by biases like recall bias or personal preferences that may alter the insights gleaned from developer interviews.

**2)External Validity**

* **Limited Scope of Repositories**: Only a small portion of the projects in the survey made the switch from monoliths to microservices. It's possible that the results don't apply to all software systems or sectors.
* **Ecosystem Variability**: The accumulation of technical debt and architectural deterioration can be influenced by various technology stacks, team procedures, and organizational cultures; these factors are not fully taken into consideration in this study.

**3)Construct Validity**

* **Definition Ambiguity:** Different contexts may define technical debt and architectural degradation differently, which could result in different interpretations of the results.
* **Case Study Bias:** Using well-known case studies, like Netflix, may unintentionally bias results in favor of the best-case scenario rather than the average or worst-case situation.

**4)Conclusion Validity**

* **Correlation versus Causation**: Metrics and technical debt correlations do not prove causation. The observed patterns could be influenced by outside causes.

**6 FUTURE WORK RECOMMENDATIONS**

* **Extending Repository Coverage**: To increase generalizability, future research should examine a larger and more varied collection of repositories from various programming languages and fields.
* **Including Extra Metrics**: A more comprehensive understanding of technical debt and architectural deterioration can be obtained by utilizing a greater variety of object-oriented and architectural metrics.
* **Longitudinal Research**: Longer-term research projects can better document the development of technological debt and how it affects system quality.
* **Automated Monitoring tools**: Real-time detection and resolution could be improved by creating or incorporating cutting-edge technologies to automatically monitor architectural deterioration and technical debt throughout development.
* **Team Dynamics Analysis**: Examining how team size, experience, and communication methods affect technical debt mitigation can yield useful suggestions for businesses.
* **Empirical Validation**: The usefulness of the suggested ideas can be confirmed and further improved by testing them in actual situations.
* **Cross-Architecture Comparison**: The findings' wider application might be investigated by extending the investigation to serverless systems or hybrid architectures.

**7. CONCLUSION**

This study focuses on monolithic and microservices architectures to illustrate the complex relationship between architectural degradation and technical debt in software systems. Six actively maintained GitHub repositories were empirically examined to identify trends and patterns of the accumulation of technical debt using object-oriented design criteria.

The results show that because microservices are dispersed and interdependent, they are more vulnerable to architectural deterioration even while they provide for agility and scalability. On the other hand, monolithic systems have difficulties with scalability and maintainability but tend to accrue technological debt more gradually.

The significance of regular design reviews, coding standards compliance, and proactive technical debt management are all emphasized in the beneficial concepts offered by case studies and developer insights. By putting these tactics into practice, teams can maintain architectural integrity and lessen the long-term effects of technical debt.

Further study should concentrate on broadening the analysis's scope, adding more metrics, and investigating cutting-edge tools and techniques to improve software quality and maintainability in a range of architectural contexts. In the end, our research helps create software systems that are both high-performing and sustainable by providing a basis for comprehending and resolving the issues of technical debt and architectural deterioration.

# 8 REFERENCES

1. Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. IEEE Software, 29(6), 18-21.
2. Bogner, J., Wagner, S., & Zimmermann, A. (2017). Towards a practical approach for managing architectural technical debt: Experiences from industry. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 37-40).
3. Fowler, M. (2014). Microservices - A definition of this new architectural term. martinfowler.com.
4. Gorkhali, A., Xu, Y., & Li, X. (2020). Challenges in Microservices: A Systematic Review. In Proceedings of the 2020 International Conference on Software Engineering Research & Practice (SERP).
5. Lim, E., Taksande, N., & Seaman, C. (2012). A Balancing Act: What Software Practitioners Have to Say About Technical Debt. IEEE Software, 29(6), 22-27.
6. Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Architectural Patterns for Microservices: A Systematic Mapping Study. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 1-10).
7. Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure it? Manage it? Ignore it? Software practitioners and technical debt. In Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (pp. 50-60).
8. Chen, L. (2018). Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Software, 35(2), 50-54.
9. Cunningham, W. (1992). The metaphorical model of technical debt
10. Spring Petclinic Microservices. (n.d.). GitHub. <https://github.com/spring-petclinic/spring-petclinic-microservices>
11. Moduliths. (n.d.). GitHub. <https://github.com/moduliths/moduliths>
12. SiteWhere. (n.d.). GitHub. <https://github.com/sitewhere/sitewhere>
13. Eventuate Tram Example Customers and Orders. (n.d.). GitHub. <https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders>
14. Aniche, M. (2020). CK Metrics. GitHub. <https://github.com/mauricioaniche/ck>
15. Asif, M. (2020, June 3). Monolithic vs. Microservices Architecture: Case Study on Netflix. LinkedIn. Retrieved from <https://www.linkedin.com/pulse/monolithic-vs-microservices-architecture-case-study-netflix-asif/>
16. Sparity. (n.d.). Monolithic to Microservices. Retrieved from <https://www.sparity.com/case-studies/monolithic-to-microservices/>
17. Etute, D. (2021, February 23). *Coca-Cola’s new microservices model via DevOps: A case study*. LinkedIn. Retrieved from <https://www.linkedin.com/pulse/coca-colas-new-microservices-model-via-devops-case-study-etute/>
18. Surabhi, S. (2024). Software Architecture Design Final Research [Repository]. GitHub. <https://github.com/SupriyaSurabhi/Software_Architecture_Design_Final_Research>
19. Santalis, T. (n.d.). *JDeodorant* [Computer software]. GitHub. Retrieved December 11, 2024, from <https://github.com/tsantalis/JDeodorant>