On maintainability and microservice dependencies: How do changes propagate?

✉ Tomas Cerny[1][a], Md Showkat Hossain Chy[2][b], Amr S. Abdelfattah[2][c], Jacopo Soldani[3][d], and Justus Bogner[4][e]

[1]*Systems and Industrial Engineering, University of Arizona, 1127 East James E Rogers Way, Tucson, 85721, Arizona, USA*

[2]*Computer Science Department, Baylor University, 1420 S 5th St, Waco, 76798, Texas, USA*

[3]*University of Pisa, Lungarno Antonio Pacinotti, 43, 56126 Pisa PI, Italy*

[4]*Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, Netherlands*

tcerny@arizona.edu

Keywords: Microservices, Maintainability, Dependencies

Abstract: Modern software systems evolve rapidly, especially when boosted by continuous integration and delivery. While many tools exist to help manage the maintainability of monolithic systems, gaps remain in assessing changes in decentralized systems, such as those based on microservices. Microservices fuel cloud-native systems, the mainstream direction for most enterprise solutions, which drives motivation for a broader understanding of how changes propagate through such systems. This position paper elaborates on the role of dependencies when dealing with evolution challenges in microservices aiming to support maintainability. It highlights the importance of dependency management in the context of maintainability deterioration. Our proposed perspective refines the approach to maintainability assurance by focusing on the systematic management of dependencies as a more direct method for addressing and understanding change propagation pathways, compared to traditional methods that often only address symptoms like anti-patterns, smells, metrics, or high-level concepts.

1 INTRODUCTION

The evolution of software engineering practices over the past few decades has been fundamentally driven by the quest for more maintainable, robust, and adaptable systems. This journey, marked by significant milestones, has transitioned from the era of structural programming to the adoption of modular design principles, profoundly influencing how software is conceptualized, developed, and maintained. These advancements, as documented in pioneering works (Parnas, 1972; Parnas et al., 1985), have laid a solid foundation for the software engineering discipline, emphasizing the importance of structure and modularity in achieving maintainable software systems.

In parallel, the advent of design patterns, as elegantly described by Gamma et al. (Gamma et al., 1993), and the philosophical underpinnings of software construction articulated by Dijkstra (Dijkstra, 1982), have further enriched our understanding of software maintainability. These contributions have not only advanced our conceptual framework for software design but have also provided practical tools and methodologies for enhancing software reusability, understandability, and, consequently, its maintainability. The ISO 25010 standard (iso, 2023) codifies these insights by defining maintainability as the degree of effectiveness and efficiency with which a system can be modified to correct faults, improve performance, or adapt to changing environments. This standard underscores the critical attributes of modularity, reusability, analysability, modifiability, and testability, framing them as essential for the long-term health and evolution of software systems.

The complexity of modern software systems, coupled with the accelerating pace of technological innovation, necessitated a paradigm shift towards

[a] https://orcid.org/0000-0002-5882-5502
[b] https://orcid.org/0009-0006-6978-399X
[c] https://orcid.org/0000-0001-7702-0059
[d] https://orcid.org/0000-0002-2435-3543
[e] https://orcid.org/0000-0001-5788-0991

more dynamic and flexible architectural models. This shift was epitomized by the emergence of service-oriented architecture (SOA) (Cerny et al., 2018), which introduced a new way of thinking about software as a collection of independently deployable services. SOA paved the way for the modularization of functionality, allowing systems to be composed of loosely coupled services that can be developed, deployed, and maintained independently. This architectural style represented a significant leap forward, enabling greater agility and scalability in software development and deployment processes.

Building upon the foundations laid by SOA, the microservice architecture emerged as a natural evolution, driven by the need for even more granular modularity and scalability. Microservices take the principles of SOA to the next level, advocating for small, self-contained services that are built around business capabilities and independently deployable by fully automated deployment machinery (Cerny et al., 2018). This approach has facilitated the development of complex, scalable, and resilient systems that can rapidly evolve in response to changing business requirements. The widespread adoption of microservices (ado, 2023) is a testament to their effectiveness in addressing the challenges of modern software development, including the need for continuous integration and delivery, scalability, and system resilience.

However, the transition to microservice architectures has not been without challenges. While microservices offer significant advantages in terms of scalability and flexibility, they also introduce complexity, particularly in terms of managing the dependencies between services. As systems become more distributed, the interconnections between microservices become more intricate, posing new challenges for maintainability. This complexity is further compounded by the diversity of technologies, frameworks, and languages used across different microservices, making system-wide changes more difficult to implement and manage. Moreover, in a microservice ecosystem, each service is typically owned by a separate team, operating with a high degree of independence.

This position paper questions how the interdependencies between system components influence the overall architecture's maintainability when modifications are introduced. It navigates through the complexities of microservice systems, spotlighting the essential role of dependency management in preserving system integrity amidst evolution. We call for a reevaluation of traditional microservice analysis techniques, suggesting a move towards methodologies that probe the underlying reasons for changes. The discourse pivots towards identifying microservice dependencies as a critical focal point for assessing change impacts, proposing their meticulous management as a key to enhancing system resilience and maintainability. By focusing on the strategic management of inter-service dependencies, our goal is to foster the development of sophisticated tools and frameworks. These innovations are intended to enhance impact analysis and support continuous architectural quality improvement, thereby empowering developers and architects to more effectively navigate the intricacies of microservice ecosystems and achieve higher levels of system resilience and maintainability.

The rest of this manuscript is organized as follows. Section 2 presents established approaches aiding maintenance or detecting degradation in software applications. Section 3 argues the importance of dependencies and reasons about the symptoms and causes. Section 4 considers different types of dependencies and their management to drive change impact and improve maintenance. Section 5 concludes the paper with open-ended questions.

2 BACKGROUND AND RELATED WORK

The foundational principles of modular design, encapsulation, and separation of concerns significantly enhance software maintainability. Parnas's early contributions ((Parnas, 1972)) and further studies ((Parnas et al., 1985)) have shown that clearly defined module responsibilities and interfaces simplify maintenance and comprehension. The concept of separation of concerns by Dijkstra ((Dijkstra, 1982)) and the focus on encapsulation by Gamma et al. ((Gamma et al., 1993)) underpin a design philosophy that enhances modifiability and minimizes error risk during system evolution.

However, ensuring software quality encompasses more than these design principles. It involves adherence to coding standards ((Bass et al., 2021)), application of best development practices ((Gamma et al., 1993)), proper assignment of responsibilities ((Larman et al., 1998)), and effective dependency management. Together with comprehensive documentation, version control, and thorough testing, these practices form the core of exemplary design and development.

As software systems grow in complexity, the significance of architecture in maintainability becomes more pronounced. Effective architecture

facilitates easy modifications and guards against quality degradation. Yet, the drive for rapid feature development often undermines long-term architectural integrity, leading to technical and architectural debt ((Besker et al., 2018; Martini et al., 2018; Das et al., 2022; Azadi et al., 2019; Fontana et al., 2016; Haendler et al., 2017)), where quick fixes harm future maintainability and scalability.

Strategic approaches are essential for addressing architectural degradation, as systematic literature reviews have identified ((Baabad et al., 2020)). These include detecting architectural smells, enforcing architectural rules, and mitigating architectural degradation. Yet, existing research primarily targets monolithic systems, overlooking the unique challenges posed by microservices.

Microservices introduce complexities such as multiple moving parts, a disconnected codebase, scattered concerns, and autonomously operating development teams, as highlighted by Conway's law ((Conway, 1968)). Despite the advantages of loosely coupled components, unavoidable dependencies can lead to co-changes and ripple effects ((Bogner et al., 2021)), making decentralized systems more susceptible to architectural degradation.

In a decentralized microservices architecture, the modular design's benefits are challenged by the complexity of managing independent yet interdependent components. This independence complicates measuring changes and managing dependencies, potentially causing costly ripple effects across multiple teams. While microservices operate as distinct applications, they collectively form a single system, necessitating a system-level management approach. Many issues become apparent only when viewed from this holistic perspective, rather than through analyzing individual components in isolation.

The concept of architecture recovery or Systematic Architecture Reconstruction (SAR), as mentioned in the literature ((Baabad et al., 2020)), is pivotal for microservices' success, offering a system-centered analysis approach ((Bogner et al., 2021)). However, implementing SAR effectively is daunting, requiring significant, ongoing effort ((Rademacher et al., 2020)). While static analysis ((Walker et al., 2020)) may aid in approximating this reconstruction, a deep understanding of the interconnections between components is essential for identifying dependencies and relationships.

Microservices have specific smells (Cerny et al., 2023) that can manifest across microservices (e.g., cyclic dependency); their detection could use known anti-patterns that could manifest through SAR or by accessing codebases through simple rules (Tighilt et al., 2023). Yet metrics for microservices are in their infancy. We can emphasize works looking into structural coupling (Panichella et al., 2021), logical coupling (d Aragona et al., 2023), team collaboration dependency (Lenarduzzi and Sievi-Korte, 2018), etc. However, we could question if these are the essential dependency causes or just symptoms of such cases.

While existing approaches provide a foundation for maintaining software quality within monolithic and decentralized systems alike, the unique challenges posed by microservice architectures—such as decentralized management, scattered concerns, and complex dependencies—remain inadequately addressed. Our position paper proposes a new perspective, emphasizing the critical role of dependency management in navigating the evolution of microservices. By highlighting this gap, we aim to spur further research and development of tools and methodologies that specifically cater to the nuanced demands of microservice ecosystems.

3 THE ROLE OF DEPENDENCIES: SYMPTOMS AND CAUSES

When reasoning about how microservices interconnect, we can easily think of what is obvious - "explicit" inter-service calls. Such calls align the path for dependencies, which are obvious to developers. However, what about other dependencies that are less obvious? For instance, as demonstrated by Walker et al. (Walker et al., 2021), data dependencies could be considered. In their work, they construct the canonical data model from across microservices based on similar data entities (names or structures). Such dependencies (Figure 1) might exist due to the same data microservices exchange or as a legacy of their monolith migration.

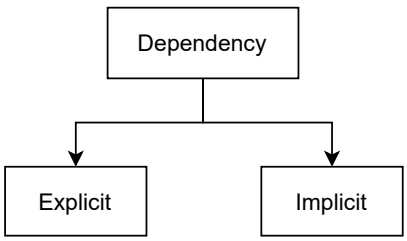

Figure 1: Types of Dependencies

However, there are other "implicit" dependencies that developers might miss when reviewing change

impact. For example, dependencies sourcing from implicit invocation involve events through message systems. We can also consider another perspective for such dependencies that involve policies and rules applicable to the entire system. These could be, for instance, compliance with the General Data Protection Regulation (GDPR) or access rights that can not be modularized and reused because it would violate the principles of cloud-native systems and introduce bottlenecks. As a result, we are left with the option of a scattered concern and thus to restate and customize the policy specifically to a given microservice while scattering the concern across multiple components and then dealing with finding each of these when the policy changes. There are other implicit dependency types, and likely, these will be missed by developers assessing change impact propagation to other microservices. Careful dependency management could guide developers on potential implications.

Dependencies in microservices, defined as the reliance of one module on another for proper functioning, often lead to challenges like co-change requirements and ripple effects. These challenges are not the root causes but rather symptoms of underlying dependencies. Researchers are thus motivated to explore beyond traditional dependency models to address these symptomatic challenges directly.

The distinction between the causes, affected artifacts, and symptoms of dependencies can be nuanced. For example, version control commits often tied to data dependencies between microservices result in logical coupling ((d Aragona et al., 2023)), a symptom of such dependencies. Likewise, team collaboration issues may stem from shared resource dependencies ((Lenarduzzi and Sievi-Korte, 2018)).

Software architecture reconstruction (SAR) aimed at change impact analysis typically uncovers only control dependencies through dynamic analysis, which might be obvious to developers. Manual SAR can be overwhelming, pushing towards evident paths, yet static analysis presents broader opportunities. It can, for instance, uncover code clones across microservices, indicating potential change pathways. These clones, whether syntactic or semantic, highlight the need to navigate polyglot diversity, despite the absence of significant data on polyglot prevalence in cloud-native systems.

Beyond clones, static analysis aids in identifying data dependencies to gauge change impacts on interconnected components. Addressing policies and rules introduces additional complexity due to the lack of a uniform method for their analysis, necessitating strategies like rule annotation or employing rule engines like Drools for a standardized expression. Static analysis also extends to identifying technology dependencies, where changes in one component might necessitate adjustments elsewhere. A comprehensive resource dependency check would include shared libraries, configuration files, data sources, and cloud-native infrastructure elements such as API gateways and service discovery.

The complexity of addressing both functional and non-functional requirements, which dictate the responsibilities of system components and necessitate modifications in the source code, often surpasses the capabilities of conventional automation techniques. This challenge, however, finds a promising solution in the model-driven development (MDE) approach, as highlighted by Terzić et al. (Terzić et al., 2018). Despite its potential to streamline these processes, the adoption of MDE has not yet become a widespread practice within the industry, signaling a gap between its theoretical benefits and its practical implementation.

4 ARCHITECTURE DEGRADATION MITIGATION STRATEGIES IN THE CONTEXT OF DEPENDENCIES

To mitigate architectural degradation, four key strategies are identified: metrics-based detection, smell detection and prioritization, architectural recovery, and addressing architectural rule violations ((Baabad et al., 2020)). As depicted in Figure 2, these strategies offer a comprehensive approach to mitigating architectural degradation, emphasizing the importance of a nuanced understanding of system dependencies.

4.1 Metrics-based strategies

Metrics-based detection in source code offers insights into software quality and maintainability, assessing effectiveness and architectural instability throughout its evolution. Key metrics focus on instability, modularity, coupling, and cohesion. High code churn signals potential instability and architectural concerns. Modularity metrics, such as the average number of modified components per commit (ANMCC), Index of Package Changing Impact (IPCI), and Index of Package Goal Focus (IPGF), indicate degradation risks (Li et al., 2014). Cyclomatic Complexity (CC) reveals code complexity and potential maintenance challenges, while high duplication levels highlight modularity

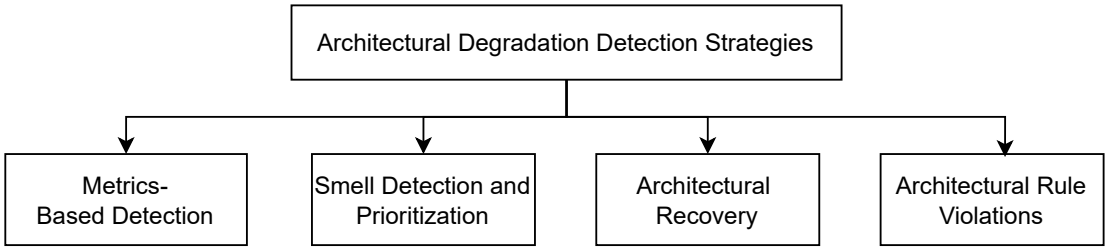

Figure 2: Architecture degradation strategies in the context of dependencies

issues, complicating consistent updates and risking inconsistent behavior.

When considering code metrics in the context of cloud-native systems, we must account for the decentralized codebase or at least for self-contained code modules that do not have direct connections across microservice components, which opens gaps. When we use remote calls as the dependency indicator, it is likely not enough to get a reliable perspective because there are other dependencies across microservices, as we suggested in the previous section.

At the same time, we may argue that practitioners do not pay attention to such non-obvious or non-tracked dependencies. In explicit dependencies, the source code explicitly states and traces the relationship. For example, when one method calls another, the code clearly states what is being called, making it explicit. Implicit dependencies lack direct connections or textual context that would lead a developer to recognize them. It is similar to implicit innovation when components do not directly invoke other components (i.e., message queue). As Garland and Shaw (Garlan and Shaw, 1993) suggest, the implicit invocation is highly scalable and makes components reusable, but we also lose execution control and need data exchange encapsulated within an event.

Recognizing various types of dependencies (apart from the explicit ones) across microservice components would open new pathways to re-apply known metrics with better reliability measures towards the overall decentralized system rather than using the individual microservice perspectives without details on their inter-dependencies. One could question such dependencies qualitatively when two microservices are dependent and what weights can be expressed on different types of dependencies. Certainly, different dependency types will present different weights but also other properties alongside their quantities. All these could be used when reasoning about system evolution and the change impact propagation across components and specific artifacts to drive reviewers.

Highlights:

- Emphasizing dependency management as key to addressing architectural degradation in microservices.

4.2 Smell detection strategies

Smell detection and prioritization have been the practice for many years, and most research would point to tools like SonarQube or alternative tools for the detection of anti-patterns. A smell is typically an indicator of a deeper problem, not knowing what the specific cause is, so it can be seen as a symptom. Anti-pattern is a manifestation of a specific problem.

While most tools are designed to operate within monolithic repository systems, challenges arise when dealing with multiple repositories. Multiple disjoint repositories make it difficult to detect microservice anti-patterns (Cerny et al., 2023) that span across multiple components. Basic approaches use dependency graphs (Al Maruf et al., 2022; Walker et al., 2020), call graphs or syntax trees; however, also plain rule-based marching checking has been (Tighilt et al., 2023). If we consider the approaches using service dependency graphs, we can generalize such graphs as an intermediate system representation and a product of architectural recovery; thus, the strategies presented in (Baabad et al., 2020) blend. It is obvious that the greater the intermediate system representation we can get, the better analysis can be performed. Thus, by recognizing and tracking all dependencies across code and artifacts in microservices, more reliable results can be provided to reviewers.

The second question to raise with regards to intermediate system representation is whether it can fit various frameworks and platforms; however, tools like Oracle's GraalVM show such direction is possible, and other approaches (Schiewe et al., 2022) indicate this applied to cloud system development frameworks as well.

Still, the major question is whether smells and anti-patterns are the proper strategies to use. While

it has been established as an approach to use, we must notice that smells are symptoms. Moreover, anti-patterns are only identified and described for a limited number of problems after they reoccur multiple times with negative consequences. They seem to be a result of identified degradation that is generalizable. Unless we detect them in a new system version, they will be a very discrete indicator of degradation.

On the other hand, dependencies are apparent at a much lower level and do not need a prior description of combined rules or graph glyphs in code. Proper identification and management of dependencies would likely indicate a problem even without knowing there is an anti-pattern but not vice versa. If we compare the changed system versions, the impact can be more quantified than using a collection of anti-patterns that will always be limited. Likely, with proper details of different types of dependencies, more anti-patterns could be identified, and vice-versa, known anti-patterns could be marched to dependency graphs.

Highlights:

- Highlighting the need for advanced metrics and analysis tools tailored to the microservice architecture's unique challenges.

4.3 Architecture reconstruction strategies

Architecture reconstruction and recovery in microservice environments are pivotal for maintaining system integrity and comprehending the evolution of the system architecture over time. The distributed nature of microservices complicates this process, necessitating the analysis of dependencies and interactions across multiple services and infrastructure components. The task of architecture reconstruction in microservice environments involves not just the mapping of services and their interactions but also understanding the underlying dependencies and the rationale behind architectural decisions. Recent advancements focus on automating the extraction and analysis of architectural information, combining both static codebases and dynamic runtime behaviors (Alshuqayran et al., 2018).

The architecture reconstruction/recovery direction (Walker et al., 2021), as mentioned with smell and anti-pattern detection section, can be used for question answering or as a documentation of the system; there are typically multiple system views extracted to enable human experts to reason about the system. Alternatively, direct questions about the system can be asked and answered. However, our experience is that while we can recover individual views, we do so with explicit dependencies or control flows. There are no clues of how individual pieces depend on each other and why. It is important to augment these views, models, or representations that result from the reconstruction with dependencies that would enable stronger analysis on top of the reconstruction process.

Highlights:

- Highlighting the need for advanced strategies in detecting smells and identifying anti-patterns tailored to microservice dynamics.

4.4 Architectural Rule Violation Detection Strategies

Architectural rule violations highlight deviations from the intended architecture, guiding developers to adhere to specific design principles. For instance, a common rule might enforce that service layers only communicate through defined interfaces, preventing direct data layer access from the UI layer. Prioritizing these rules by severity helps developers focus on the most critical architectural integrity issues.

The effectiveness of these rules depends on their ability to comprehensively capture the system's architectural dependencies. If certain dependencies are overlooked, the rules' capacity to enforce the architecture weakens. Thus, keeping track of all system dependencies is essential for the rules to remain expressive and powerful.

Additionally, managing architectural rules requires attention similar to system code, as these rules can become outdated as the architecture evolves. This necessitates a maintenance strategy for the rules themselves, ensuring they are kept current and reflective of the system's architectural standards and practices.

Highlights:

- Emphasizing the need for innovative methods in architecture reconstruction and detecting rule violations to preserve system integrity.

4.5 Implications

Concerning the dependencies, the established strategies to detect architectural degradation seem to be positioned at a higher level of abstraction. Moreover, these strategies are difficult to apply in

a highly decentralized system with independent services. For instance, to perform proper architecture reconstruction, we need to access and review each microservice codebase and manual approach of the topic, given the pace with which we evolve these systems. Similar efforts are needed to assess the violation of architectural rules. While an approximation can be provided by static analysis approaches, we still need to deal with platform diversity, and tooling support is in its infancy.

As opposed to presented strategies, dependencies are at the lower level and more fundamental elements to build new tools on top of. They open new pathways to identify issues, causes, and alternate anti-patterns without their prior cataloging. They can be used with architectural rules but will not leave the burden of required maintenance if used as a sole indicator of degradation. Moreover, using augmented intermediate system representation with identified and tracked dependencies, current metrics could be utilized on the overall system rather than just its components.

5 MAINTANABILITY IN THE CONTEXT OF MICROSERVICE DEPENDENCIES

Maintainability within microservice architectures emphasizes the facility to modify, extend, or "changes software system with minimal effort", aiming to minimize error introduction during changes. Essential to this concept is change impact analysis, which primarily serves to detect and assess the potential effects of modifications, rather than merely limiting their ripple effect. This analysis is vital for understanding the implications of modifications on system maintainability, particularly in microservices where dependencies are intricate.

Dependencies serve as a precise indicator of how changes may influence maintainability, offering insights into the specific areas of the system affected. This precision aids in identifying the root causes of potential maintainability issues, enabling decision-makers to weigh the benefits of changes against their impacts. Unlike composite metrics and smells, dependencies elucidate the direct relationship between changes and their effects, guiding the consideration of alternative designs or strategies to mitigate adverse outcomes.

Bass et al. and ISO 25010 define maintainability through attributes such as modularity, reusability, analysability, modifiability, and testability ((Bass et al., 2021)). In the realm of microservices, adept management of dependencies is crucial for bolstering these attributes, highlighting the importance of comprehensive change impact analysis to sustain system integrity and flexibility. As illustrated in Figure 3, understanding the quality attributes for maintainability as outlined by ISO 25010 is crucial.

Modularity, a key maintainability attribute, entails dividing a complex system into smaller, independent, and interchangeable components, aiming for minimal impact on other components when one is changed. In the context of microservices, this translates to designing each service to handle a specific function, allowing them to operate cohesively within the larger system. Although modularity seeks to enhance maintainability, reusability, and clarity, achieving it in cloud-native systems is challenging due to the interconnected nature of microservices and the distribution of global policies and knowledge (e.g., role-based access control) across the system. This necessitates careful tracking of dependencies to manage scattered modularity effectively. One proposed solution is to centralize shared logic in importable libraries, though this approach may conflict with cloud-native principles like those outlined in the Twelve-Factor App methodology.

Modularity: Dependency tracing is crucial due to complex inter-service interactions and policy dispersion in microservices.

Reusability is the practice of designing software components to be used multiple times across different system parts or various projects, aiming to save development time and resources by utilizing existing, validated components. This concept, fundamental to software engineering, enhances efficiency and reduces redundancy. While modularity lays the groundwork for reusability, the evolution towards service-oriented architectures initially amplified its application. However, the shift towards scalable, self-contained microservices introduces complications, such as latency and bottlenecks, which challenge the straightforward application of reusability principles. Against this backdrop, the serverless approach offers a promising pathway to circumvent the limitations posed by microservices. Distinguished from traditional server-centric models, the serverless approach focuses on executing backend services on an as-used basis without requiring the application developers to manage server infrastructure. This paradigm allows for the creation and deployment of functions that execute in response to events, facilitating the reuse of these functions across different parts of a system or even across projects. By abstracting away the underlying infrastructure management, the serverless approach

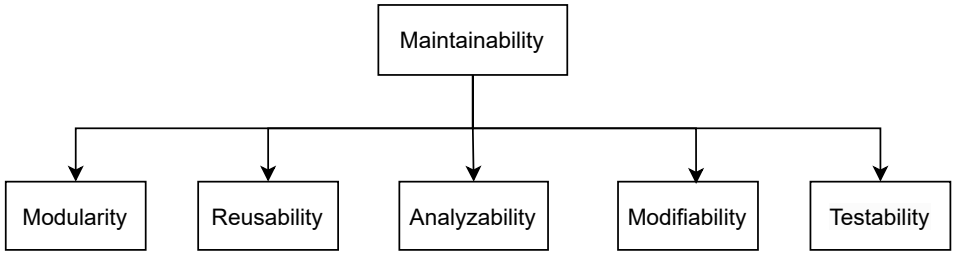

Figure 3: Quality attribute for maintainability according to ISO 25010

mitigates common microservice challenges, making it an effective strategy for achieving reusability in the context of modern software development demands.

Reusability: Scalability and independence in microservices challenge reusability, pointing towards serverless solutions.

Analysability indicates the ease with which software can be analyzed or examined for various purposes, such as understanding its structure, identifying potential issues, debugging, and making improvements. An analyzable system facilitates effective inspection, diagnosis, and comprehension by developers and other stakeholders. Microservices lead us to decentralization, and with the separation of duty, developers manage their realm but do not have the system's big picture. While most currently adopted approaches push for tracing, it is greatly dependent on developers' consistency in proper logging and willingness to sacrifice performance. Static analysis can give a fast approximation of the system, but tools lack microservices.

Analysability: The decentralized nature of microservices necessitates improved tracing and static analysis for effective system comprehension.

Modifiability is the ease with which a system can be modified or adapted to meet changing requirements. It is a crucial quality attribute as software evolves to accommodate new features, address bugs, and respond to changing user needs. A highly modifiable system allows developers to make modifications efficiently, minimizing the risk of introducing errors and reducing the time and effort required for updates. In this perspective, we see the greater potential of dependency management, which is proper change impact analysis can reduce the negative effect of improper design. However, to enable such analysis, the system must be analyzed first, pointing to the importance of the prior concept.

Modifiability: Robust dependency management facilitates efficient and error-minimizing modifications in microservices.

Testability is the ease with which a software system can be tested to ensure that it meets its specified requirements and behaves as intended. A highly testable system is designed in a way that facilitates the creation, execution, and maintenance of tests. While not directly related, it brings a great discussion. First, tests are expensive, and static analysis might provide answers with much less resource demands. Tests need maintenance, and there are dependencies between tests and the system, so there are dependencies between system changes and tests. That said, system change might not require a complete system test, but with properly managed dependencies between tests and modular system design, specific tests could be triggered to run to minimize resource usage. Moreover, changes in the system could indicate which specific tests need to be changed, which would be especially useful for end-to-end testing.

Testability: Dependency-aware testing strategies are key to optimizing test maintenance and resource use in microservices.

Figure 4 illustrates the concept of architectural degradation and its influence on system maintainability. The roots represent system dependencies, essential for nourishing and stabilizing the architecture, shown as the tree's trunk. As the trunk branches out, signifying the system's architecture, the leaves depict the aspect of maintainability. The barren branches indicate an unmaintained system, suggesting that neglecting the underlying dependencies leads to the deterioration of the system's structural integrity. This visual metaphor emphasizes that maintaining the system's architecture depends on the careful management of its dependencies, which, if mishandled, can lead to architectural degradation and a consequent decline in maintainability.

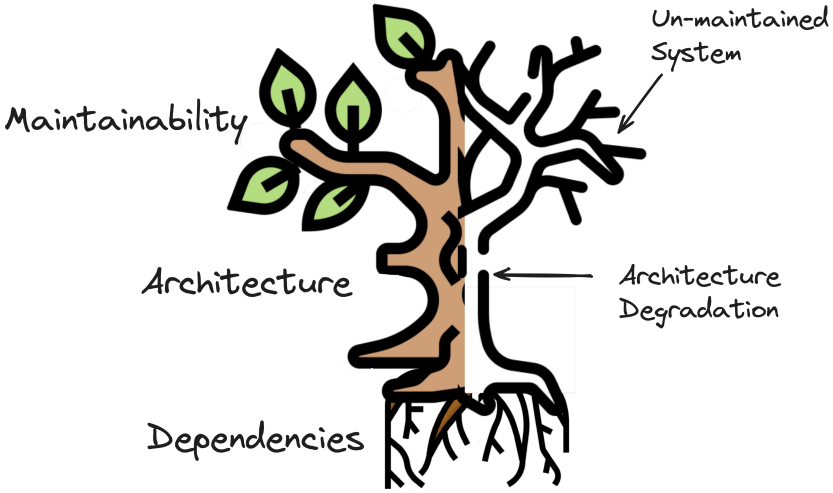

Figure 4: Visualizing root cause of architectural degradation

6 CONCLUSIONS

In this paper, we have highlighted the importance of dependency management in microservice systems when dealing with maintainability. While evolving systems must be managed to mitigate architectural degradation and various approaches exist for mono-repo systems, there is still a gap in strategies that would meet the needs of microservices. We highlight the potential of dependency management in microservices to aid change impact analysis and discuss the perspective within the context of currently recognized strategies. Using managed dependencies as an instrument to assess changes in decentralized systems could aid developers in their efforts and help them reach better outcomes more effectively. However, with this perspective, more research is necessary.

Further research is essential to bridge the current gaps in dependency management strategies for microservices, aiming to devise solutions that are both scalable and adaptable to the rapid pace of technological advancements. Pursuing these research directions will not only address the immediate challenges faced by developers but also lay the groundwork for more resilient and maintainable microservice architectures in the future.

ACKNOWLEDGEMENTS

This work is based upon work supported by the National Science Foundation under Grant No. 2409933.

REFERENCES

(2023). Iso 25000 portal. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability. Accessed: 2023-12-20.

(2023). Microservices adoption in 2020. https://www.oreilly.com/radar/microservices-adoption-in-2020/. Accessed: 2023-12-20.

Al Maruf, A., Bakhtin, A., Cerny, T., and Taibi, D. (2022). Using microservice telemetry data for system dynamic analysis. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 29–38. IEEE.

Alshuqayran, N., Ali, N., and Evans, R. (2018). Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709.

Azadi, U., Fontana, F. A., and Taibi, D. (2019). Architectural smells detected by tools: A catalogue proposal. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops. IEEE Press.

Baabad, A., Zulzalil, H. B., Hassan, S., and Baharom, S. B. (2020). Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709.

Bass, L., Clements, P., and Kazman, R. (2021). *Software Architecture in Practice: Software Architect Practice_c4*. Addison-Wesley.

Besker, T., Martini, A., and Bosch, J. (2018). Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, page 105–114, New York, NY, USA. Association for Computing Machinery.

Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2021). Industry practices and challenges for the

evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104.

Cerny, T., Abdelfattah, A. S., Maruf, A. A., Janes, A., and Taibi, D. (2023). Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*, 206:111829.

Cerny, T., Donahoo, M. J., and Trnka, M. (2018). Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45.

Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

d Aragona, D. A., Pascarella, L., Janes, A., Lenarduzzi, V., Penaloza, R., and Taibi, D. (2023). On the empirical evidence of microservice logical coupling. a registered report.

Das, D., Maruf, A. A., Islam, R., Lambaria, N., Kim, S., Abdelfattah, A. S., Cerny, T., Frajtak, K., Bures, M., and Tisnovsky, P. (2022). Technical debt resulting from architectural degradation and code smells: A systematic mapping study. *SIGAPP Appl. Comput. Rev.*, 21(4):20–36.

Dijkstra, E. W. (1982). *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY.

Fontana, F. A., Roveda, R., Vittori, S., Metelli, A., Saldarini, S., and Mazzei, F. (2016). On evaluating the impact of the refactoring of architectural problems on software quality. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops, New York, NY, USA. Association for Computing Machinery.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7*, pages 406–431. Springer.

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. In *Advances in software engineering and knowledge engineering*, pages 1–39. World Scientific.

Haendler, T., Sobernig, S., and Strembeck, M. (2017). Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In *Proceedings of the XP2017 Scientific Workshops*, XP '17, New York, NY, USA. Association for Computing Machinery.

Larman, C. et al. (1998). *Applying UML and patterns*, volume 2. Prentice Hall Upper Saddle River.

Lenarduzzi, V. and Sievi-Korte, O. (2018). On the negative impact of team independence in microservices software development. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–4.

Li, Z., Liang, P., Avgeriou, P., Guelfi, N., and Ampatzoglou, A. (2014). An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, page 119–128, New York, NY, USA. Association for Computing Machinery.

Martini, A., Sikander, E., and Madlani, N. (2018). A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. *Information and Software Technology*, 93:264–279.

Panichella, S., Rahman, M. I., and Taibi, D. (2021). Structural coupling for microservices. *arXiv preprint arXiv:2103.04674*.

Parnas, D., Clements, P., and Weiss, D. (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Rademacher, F., Sachweh, S., and Zündorf, A. (2020). A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326, Cham. Springer International Publishing.

Schiewe, M., Curtis, J., Bushong, V., and Cerny, T. (2022). Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761.

Terzić, B., Dimitrieski, V., Kordić (Aleksić), S., and Luković, I. (2018). A model-driven approach to microservice software architecture establishment. pages 73–80.

Tighilt, R., Abdellatif, M., Trabelsi, I., Madern, L., Moha, N., and Guéhéneuc, Y.-G. (2023). On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. *Journal of Systems and Software*, 204:111755.

Walker, A., Das, D., and Cerny, T. (2020). Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10(21):7800.

Walker, A., Laird, I., and Cerny, T. (2021). On automatic software architecture reconstruction of microservice applications. *Information Science and Applications: Proceedings of ICISA 2020*, 739:223.