On Dependencies in Microservices: Dependency Management and Maintainability

Tomas Cerny[1(✉)], Md Showkat Hossain Chy[1], Md Arfan Uddin[1], Amr S. Abdelfattah[1], Jacopo Soldani[2], and Justus Bogner[3]

[1] University of Arizona, 1127 East James E Rogers Way, Tucson, AZ 85721, USA
tcerny@arizona.edu
[2] University of Pisa, Lungarno Antonio Pacinotti, 43, 56126 Pisa, PI, Italy
[3] Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, Netherlands

Abstract. Rapid evolution characterizes modern software systems, particularly evident with adopting continuous integration and delivery processes. However, while tools for maintaining monolithic architectures are well-established, there is a notable deficiency in methodologies for analyzing and managing changes within decentralized, microservice-based systems. As microservices increasingly become the backbone of cloud-native enterprise solutions, understanding the intricacies of how changes affect these systems becomes crucial. This paper investigates the impact of dependencies on the maintainability of microservice architectures and emphasizes the importance of managing these dependencies to prevent deterioration in system maintainability. We advocate for a systematic approach to dependency management that addresses the actual pathways through which changes propagate, providing a concrete alternative to traditional methods that often focus on symptomatic treatments such as anti-patterns and code smells.

Keywords: Software architecture · Architecture degradation · Micro-services · Maintainability · Dependencies · Service-oriented architecture

1 Introduction

The field of software engineering has experienced a profound shift over the last several decades, motivated by an ongoing quest to achieve higher levels of system maintainability, robustness, and adaptability. This progression has been marked by a significant evolution from traditional approaches like structured programming to more advanced paradigms embracing modular design principles. These changes, which are thoroughly documented in seminal publications by Parnas et al. [34,35] have played a pivotal role in shaping modern software engineering practices. They underscore the importance of structured and modular approaches, which are essential for developing software systems that are maintainable and can evolve with changing technological landscapes.

Alongside these developments, the field has been significantly influenced by the pioneering contributions regarding design patterns by Gamma et al. [20], and the philosophical perspectives on software construction put forth by Dijkstra [18]. These contributions have substantially enriched our understanding and methodologies

C. Pahl and M. van Steen (Eds.): CLOSER 2024, CCIS 2851, pp. 29–52, 2026.
https://doi.org/10.1007/978-3-032-17286-0_2

concerning software maintainability. They have provided a robust framework that has propelled forward the conceptual and practical aspects of software design, focusing on enhancing reusability, understandability, and maintainability. This framework is further supported by the ISO 25010 standard [1], which formally defines software maintainability as the capacity of a system to be efficiently modified to correct faults, improve performance, or adapt to a changing environment. This standard also highlights several critical attributes—modularity, reusability, analysability, modifiability, and testability—that are fundamental to ensuring the long-term health and evolutionary potential of software systems.

The escalating complexity of modern software architectures, coupled with the fast-paced advancements in technology, has necessitated a paradigm shift towards more dynamic and flexible architectural models. This shift is vividly illustrated by the emergence and integration of service-oriented architecture (SOA) [14]. SOA redefines the conceptualization of software, viewing it as a collection of independently deployable services, thus paving the way for systems that exemplify modularization of functionality. This approach allows for systems to be assembled from loosely coupled services that can be individually developed, deployed, and maintained. Embracing SOA not only signifies a leap forward in terms of agility and scalability in software development processes but also aligns with the need for architectures that can readily adapt to new business strategies and rapidly changing market demands.

Taking the robust framework established by SOA even further, the microservice architecture has come into prominence, targeting enhanced granularity in modularity and heightened scalability [32]. This evolution from SOA harnesses the concept of compartmentalized, self-sufficient services tailored to specific business functionalities that can be independently deployed through automated systems [14]. The successful proliferation of microservices [2] highlights their vital role in addressing key demands of contemporary software development, such as seamless integration, scalable frameworks, and robust system resilience.

Despite the advantages, the shift towards microservice architectures introduces several complexities. Notably, the management of microservice dependencies becomes increasingly challenging as the architecture grows more distributed. This growing complexity is magnified by the varied technologies, frameworks, and programming languages that microservices employ, complicating the coordination and implementation of changes across the system. Additionally, the typical microservice environment features distinct teams managing different services independently, adding layers of management complexity.

In this paper, we explore the profound impact that interdependencies have on the maintainability of software architectures, especially when modifications are necessary. By delving into the intricacies of microservice systems, we emphasize the importance of rigorous dependency management to uphold system integrity through evolutionary changes. We challenge traditional approaches to microservice analysis and advocate for innovative methodologies that address the fundamental reasons behind changes. The narrative shifts to spotlighting the meticulous management of microservice dependencies as pivotal in evaluating the ramifications of changes and bolstering the resilience and maintainability of systems. Our goal is to catalyze the creation of

advanced tools and methodologies that facilitate precise impact analysis and continuous architectural refinement, equipping developers and architects to navigate microservice ecosystems more effectively and ensure greater system stability and maintainability.

This discussion expands on our previous findings in Cerny et al. [50], where we highlighted issues surrounding microservice dependencies and their impact on system maintainability. In this extension, we provide a more detailed analysis of different types of dependencies, their detection, and practical examples that underscore the complexity of managing these interrelations effectively within microservice architectures. This additional insight fosters a deeper understanding and promotes the development of tools that support robust architectural governance and maintenance practices.

Finally, the remainder of this manuscript is structured to unfold progressively deeper insights into the topic. Section 2 revisits established methods supporting the maintenance of software applications and highlights early signs of architectural degradation. Section 3 explores the pivotal role of dependencies within software architectures, analyzing both their observable effects and fundamental causes. Section 4 examines strategies to mitigate architecture degradation in the context of these dependencies, presenting approaches to manage and counteract their effects. Section 5 delves into various types of dependencies, offering strategies for their effective management to enhance system adaptability and maintainability. Section 6 discusses the challenges of maintainability within the context of microservice dependencies, and Sect. 7 concludes the discussion with a series of open-ended questions that invite further exploration and consideration.

2 Background and Related Work

The evolution of software design has been markedly influenced by principles such as modular design, encapsulation, and the separation of concerns, which are fundamental to enhancing maintainability. The seminal works by Parnas et al. [34] illustrate the benefits of modularization, where clearly defined interfaces and responsibilities significantly ease maintenance efforts and enhance system clarity [35]. Further supported by Dijkstra's principle of separation of concerns [18] and Gamma et al.'s focus on encapsulation [20], these design philosophies collectively foster system maintainability and minimize error propagation during its evolution.

Event-driven coupling, as discussed in resources from enterprise integration patterns, elucidates a facet of microservices that significantly impacts modularity and architectural agility [23,24]. These insights emphasize the nuanced balancing act between component independence and interdependence, highlighting how event-driven architectures can both enhance and complicate microservice manageability by introducing asynchronous communication patterns that reduce direct dependencies but may obscure data flow and service interaction [23].

Quality assurance in software extends beyond these principles, requiring rigorous adherence to established coding standards [10], the implementation of proven development practices [20], precise role allocation [25], and robust dependency management. This foundation is crucial for exemplary system design and development, supported further by comprehensive documentation, diligent version control, and extensive testing.

The importance of architectural integrity grows as systems increase in complexity. Effective architecture not only facilitates easier modifications but also shields the system against quality erosion. However, the impetus to accelerate feature deployment often risks this integrity, incurring technical and architectural debt [8,11,17,19,22,30] that could undermine future maintainability and scalability.

Strategic interventions are necessary to counteract architectural degradation, with systematic literature reviews pointing to various methodologies [9], including the identification of architectural smells, adherence to design rules, and the mitigation of degradation itself. However, most existing research focuses on monolithic systems, often neglecting the distinct complexities introduced 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 [15]. The loosely coupled nature of these components, though advantageous, also breeds dependencies that can trigger co-changes and ripple effects [12], increasing the system's susceptibility to architectural decay.

In such decentralized microservice architectures, the benefits of modular design are often offset by the challenges in managing the independence and interdependence of components. These challenges make it difficult to track changes and manage dependencies effectively, potentially leading to costly disruptions across teams. While individual microservices may operate very independently, they collectively constitute a cohesive system that demands a unified management approach, revealing complexities that are often only apparent from a holistic viewpoint [12].

The success of microservices hinges on effective Systematic Architecture Reconstruction (SAR), a strategy that provides a focused system analysis approach [12]. Implementing SAR effectively is challenging and demands considerable, continuous effort [37]. Although static analysis [45] can facilitate this reconstruction, a deep understanding of component interconnections is crucial for identifying dependencies and relationships.

Microservices often exhibit distinct architectural smells such as cyclic dependencies, which are indicative of deeper systemic issues [13]. Their detection could use known anti-patterns that could manifest through SAR or through straightforward rule-based analysis of codebases [43]. Despite these methods, metrics specifically designed to evaluate microservices are still underdeveloped. Research in areas such as structural coupling [33], logical coupling [7], and team collaboration dependencies [26] has begun to shed light on these interactions. Yet, it remains a critical question whether these metrics are pinpointing the root causes of issues within microservice architectures or merely highlighting the symptoms.

This paper proposes a reimagined perspective on microservice management, emphasizing the critical role of dependency management in guiding the evolution of microservice architectures. By identifying existing gaps, we aim to encourage further exploration and development of specialized tools and methodologies tailored to meet the unique demands of microservice ecosystems.

3 The Role of Dependencies: Symptoms and Causes

Understanding the interconnectivity of microservices often starts with the most visible elements—"explicit" inter-service calls. These interactions clearly outline dependency pathways, which are immediately recognizable to developers. However, there are less apparent types of dependencies that also play a crucial role. For instance, data dependencies present a subtler form of interconnection that might not be as evident. Walker et al. [46] explore this by developing a canonical data model that spans across microservices, identifying shared data entities by their names or structures. Such dependencies, as shown in Fig. 1, often arise either from shared data exchanges among microservices or as remnants from their origins in monolithic systems. These less visible dependencies require careful consideration, as they can significantly impact the functionality and evolution of microservice architectures.

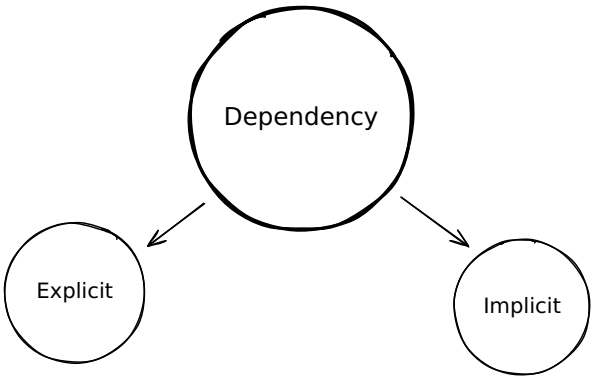

Fig. 1. Types of Dependencies.

However, the realm of "implicit" dependencies often escapes the attention of developers during the assessment of changes. These include dependencies arising from implicit invocations, such as events triggered through messaging systems. Another critical aspect of such dependencies relates to overarching policies and rules that apply system-wide. For example, adherence to regulations like the *General Data Protection Regulation (GDPR)*[1] or specific access rights may not be modularized or reused without breaching the principles of cloud-native architectures and causing potential bottlenecks. Consequently, policies must be individually tailored and applied to each microservice, dispersing the concern across various system components. This dispersion complicates the process when policies need updating, as each modification must be tracked and implemented across the distributed elements. Other types of implicit dependencies also exist and are likely overlooked during the evaluation of change impacts on microservices. Effective dependency management is crucial, as it equips developers

[1] https://gdpr-info.eu, accessed on 5 August 2024.

with the insights necessary to anticipate and mitigate potential issues arising from these less visible dependencies.

In the context of microservices, we define dependencies as the reliance of one component on another to function correctly. Dependencies affect how modifications in one microservice can impact other microservices. Such dependencies often precipitate challenges like co-change requirements and ripple effects across the system. These challenges, while significant, are merely symptoms of deeper underlying dependencies. This realization prompts researchers to look beyond conventional dependency models and address these symptomatic issues more directly.

Similarly, understanding the nuances between the causes, the artifacts affected, and the symptoms of dependencies requires a detailed analysis. For instance, version control commits, which are frequently linked to data dependencies between microservices, lead to logical coupling [7], showcasing a symptom of these underlying dependencies. Similarly, team collaboration problems may arise from shared resource dependencies [26].

Software Architecture Reconstruction (SAR) aims to analyze the impact of changes and tends to reveal only control dependencies through dynamic analysis, which are often apparent to developers. While manual SAR can be burdensome and skew towards obvious dependencies, static analysis offers a broader spectrum for discovery. It can reveal code clones across microservices, suggesting potential pathways for changes. These clones, be they syntactic or semantic, underscore the challenge of managing diversity in programming languages within cloud-native systems, despite limited data on the prevalence of such polyglot settings.

Consequently, static analysis extends beyond detecting clones; it is crucial for identifying data dependencies that affect interconnected components. Moreover, it confronts the complexity introduced by overarching policies and rules. The absence of a uniform method for analyzing these elements necessitates the use of strategies like rule annotation or the application of rule engines such as *Drools*[2], to standardize expressions. Furthermore, static analysis is instrumental in spotting technological dependencies, where alterations in one component may require changes in others. A thorough dependency check would encompass shared libraries, configuration files, data sources, and elements of cloud-native infrastructure like API gateways and service discovery tools.

Addressing both functional and non-functional requirements—which dictate component responsibilities and necessitate code modifications—often exceeds the capabilities of traditional automation techniques. However, the model-driven development (MDE) approach offers a promising solution to these challenges, as discussed by Terzić et al. [42]. Despite its potential to streamline development processes, the adoption of MDE has not been widely embraced in the industry, highlighting a disconnect between its theoretical advantages and practical application. This gap underscores the need for further research and development in deploying MDE effectively within microservice architectures. This may be due to the fact that MDE fundamentally goes against the decentralized and lightweight philosophy of

[2] https://www.drools.org/, accessed on 5 August 2024.

microservices, suggesting it might remain a situational approach rather than gaining broad support [38].

Comprehending the intricate causes and symptoms of microservice dependencies is pivotal for maintaining the efficiency and reliability of microservice architectures. This paper identifies four key areas where these dependencies manifest: architectural misalignment, resource management issues, performance degradation, and operational complexities. Each area presents unique challenges and impacts on system functionality and maintainability. This analysis delves into the specific symptoms observed, the underlying causes, and the associated artifacts, providing a comprehensive overview of the critical factors influencing microservice dependencies.

3.1 Architectural Misalignment and Design Issues

Architectural misalignment and design issues are significant contributors to microservice dependency problems. Symptoms such as cascading failures, inter-service communication complexity, and functional bottlenecks often indicate underlying architectural flaws. These issues are primarily caused by high coupling, functional centralization, and over-reliance on single endpoints, which create tight interdependencies between services. Key artifacts associated with these causes include HTTP endpoints, client request calls, and source code annotations. Abdelfattah et al. [3] emphasize that addressing these design flaws is crucial for mitigating operational disruptions and enhancing system resilience.

3.2 Resource Management and Quality of Service Issues

Resource management issues and violations of Quality of Service (QoS) present another critical area of concern. Symptoms such as large resource consumption, service conflicts, and QoS violations often stem from differentiated dependencies and challenges in service deployment. These issues are linked to multiple call graphs and service instances that complicate resource allocation and management. Lv et al. [28] highlight the significant impact of these challenges, noting how they lead to substantial resource consumption and QoS violations, necessitating improved resource management strategies to ensure efficient system operations.

3.3 Performance Degradation and Resource Allocation Challenges

Performance degradation and inefficient resource allocation are prevalent symptoms of complex service dependencies. These problems arise from dependencies' dynamic and time-varying nature, which complicate performance monitoring and resource allocation. Key artifacts include affinity matrices, performance monitoring tools, adaptive learning modules, and resource estimators. Meng et al. [31] address these challenges by exploring how service dependencies evolve, highlighting the need for dynamic management tools to mitigate performance degradation and optimize resource allocation.

3.4 Operational and Communication Complexities

Operational and communication complexities significantly impact microservice architectures, leading to symptoms such as service conflicts, uneven communication delays, and uncoordinated transmissions. Non-integrative scaling practices, co-location of microservices, and intricate service function chains cause these complexities. Artifacts such as threshold-based scaling methods, dependency topologies, and interaction relationships are crucial in understanding these issues. Song et al. [41] and Zhou et al. [49] underline the importance of addressing these operational complexities to enhance system performance and reliability.

Thus, the comprehensive analysis of the causes and symptoms of microservice dependencies underscores the need for targeted strategies to manage these challenges effectively. Future research should focus on developing predictive models and integrated management tools to mitigate the impacts of these dependencies, ensuring the robustness and maintainability of microservice architectures.

4 Architecture Degradation Mitigation Strategies in the Context of Dependencies

To counteract architectural degradation effectively, four principal strategies have been delineated [9]: metrics-based detection, smell detection and prioritization, architectural recovery, and addressing architectural rule violations. Illustrated in Fig. 2, these strategies collectively form a robust framework for addressing architectural degradation. Each strategy is designed to provide a nuanced understanding and

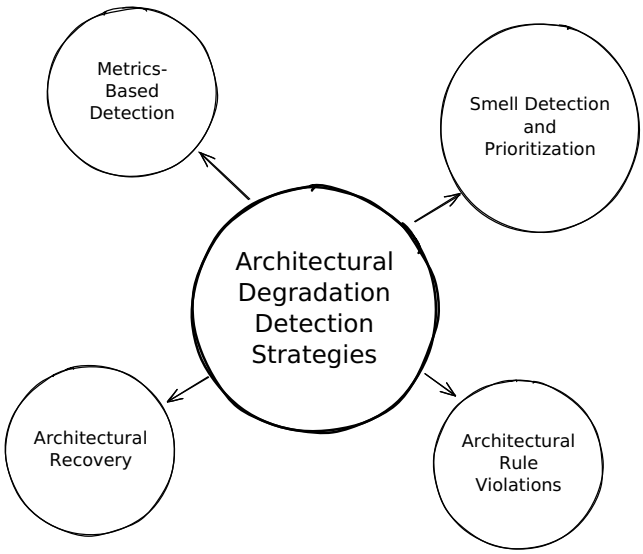

Fig. 2. Microservice architecture degradation detection strategies.

management of system dependencies, which is crucial for maintaining the integrity and performance of the architecture. This holistic approach not only identifies and corrects existing issues but also aims to fortify the architecture against potential future vulnerabilities by deeply understanding and resolving dependency-related challenges.

4.1 Metrics-Based Strategies

Metrics-based detection methods provide critical insights into software quality and maintainability by analyzing source code and assessing architectural stability throughout its development lifecycle. Essential metrics include indicators of instability, modularity, coupling, and cohesion, each providing a lens through which the health of a system can be viewed. For instance, a high code churn rate often signals potential instability and impending architectural concerns. Specific 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), offer predictive insights into the risks of architectural degradation [27]. Similarly, Cyclomatic Complexity (CC) measures the complexity of a codebase and identifies potential maintenance challenges, while metrics that quantify code duplication can reveal issues with modularity that complicate updates and foster inconsistent system behavior.

In the domain of cloud-native systems, where codebases are typically decentralized or segmented into self-contained modules without direct interconnections, traditional metrics may not adequately reflect the system's dynamics. The use of remote procedure calls as indicators of dependencies, for instance, may not provide a complete perspective because of the presence of non-obvious, implicit dependencies that span across microservices. These dependencies can be crucial for system functionality but are often not tracked or visible in the usual metrics frameworks.

Additionally, implicit dependencies pose a significant challenge as they lack direct code connections or clear textual indications, making them difficult to track and manage. Such dependencies often manifest through indirect interactions like message queues, where components communicate asynchronously, enhancing scalability and component reuse but also complicating execution control and dependency management, as highlighted by Garland and Shaw [21].

Addressing the challenge of measuring and managing these implicit dependencies within microservices requires a reevaluation of current metrics applications. A comprehensive approach that includes both explicit and implicit dependencies is necessary to provide a more accurate depiction of a decentralized system's architecture. This approach should not only focus on the immediate impacts of code changes but also consider the broader network effects and the interplay between different microservice components.

Moreover, a detailed analysis of dependency types and their respective impacts on system evolution is critical. This involves both a qualitative assessment of how dependencies interact and a quantitative measure of their effects on system stability and change propagation. By enhancing the depth and breadth of dependency analysis, organizations can better prepare for potential disruptions, plan more effective interventions, and ultimately drive more informed decision-making processes regarding system maintenance and evolution.

Highlights:

– Metrics can guide the evaluation of microservice architectures by highlighting latent dependencies not typically revealed through traditional methods, thus playing a crucial role in identifying potential points of architectural degradation.

4.2 Smell Detection Strategies

The practice of smell detection and the prioritization of code issues have been integral in software maintenance for many years, commonly facilitated by tools like *SonarQube*[3] and other platforms that specialize in identifying anti-patterns. A "smell" in the code usually signals an underlying issue but doesn't specify the cause, making it a symptomatic alert rather than a diagnostic one. Conversely, an anti-pattern represents a more concrete manifestation of a recurring problem.

Traditional smell detection tools are often optimized for monolithic repository systems, which complicates their application in environments characterized by multiple, disjointed repositories. Such environments are typical in microservices architectures, where anti-patterns can extend across various components, challenging their detection [13]. Basic methodologies involving dependency graphs, call graphs, or syntax trees [4,45], along with rule-based matching [43], offer some remedies. These techniques can be thought of as generating an intermediate system representation, which acts as a scaffold for architectural recovery, integrating strategies from established practices [9].

Therefore, this intermediate representation is critical because the more detailed and expansive it is, the more thorough the analysis that can be conducted. This approach not only helps in identifying the dependencies across codebases and artifacts in microservices but also ensures more reliable and actionable insights for developers.

A key question arises regarding the adaptability of these intermediate representations across various frameworks and platforms. Innovations like Oracle's GraalVM suggest possibilities for broader applications, and other developments indicate that similar adaptations are feasible for cloud system development frameworks [39].

However, the reliance on smells and anti-patterns as the primary strategy for identifying architectural issues warrants scrutiny. While established as a useful approach, it's important to recognize that smells often only signal symptoms of deeper problems. Anti-patterns, identified through repeated negative outcomes, offer limited insight unless they are detected afresh in updated system versions.

In contrast, dependencies provide a foundational view of system interrelations and do not require a pre-defined set of rules or visual indicators like graph glyphs to be useful. Proper management and identification of dependencies can reveal systemic issues even without prior knowledge of specific anti-patterns. Comparing different system versions can make the impact of changes more quantifiable than merely relying on a catalog of anti-patterns. By employing techniques like dependency mapping to

[3] https://github.com/SonarSource/sonarqube, accessed on 5 August 2024.

visualize service interactions and using change impact analysis to assess the potential effects of modifications, not only can existing anti-patterns be linked to specific dependencies, but new patterns of degradation can also be uncovered. These methods enhance the overall strategy for maintaining and improving software architecture by providing a clearer understanding of how changes in one area can affect the entire system.

Highlights:

– Advancing smell detection strategies with dependency analysis to effectively manage systemic issues and detect degradation patterns in microservice architectures.

4.3 Architecture Reconstruction Strategies

Architecture reconstruction and recovery are essential processes for ensuring the integrity and understanding of the ongoing evolution of microservice architectures. Due to their distributed nature, microservices present a complex landscape where multiple services and infrastructure components interact and depend on each other. The primary goal of architecture reconstruction in such environments is not only to map out the services and their interactions but also to delve deep into the underlying dependencies and the logic behind architectural choices. Recent technological advances have enabled the automation of extracting and analyzing architectural data, integrating both static and dynamic aspects of the codebase and runtime behavior [5].

As discussed earlier in conjunction with smell and anti-pattern detection strategies, architecture reconstruction can also serve as a foundation for resolving queries or documenting the system. This process typically results in multiple system views that allow experts to dissect and understand the architecture more thoroughly. These views or models can be used to extract specific information about the system's structure and operation, effectively answering questions and testing hypotheses about architectural decisions and their impacts.

However, a notable challenge in this process is the focus on explicit dependencies and control flows, which often overlook the subtler, implicit connections between components. This limitation can lead to a superficial understanding that lacks depth in explaining how different parts of the system interact and depend on each other. To address this issue, it is crucial to enrich the reconstructed views or models with detailed information about dependencies. By incorporating a comprehensive mapping of dependencies into the reconstruction process, analysts can perform a more robust and insightful analysis. This enhanced approach helps not only in understanding the current state of the architecture but also in making informed decisions about future modifications and improvements.

Highlights:

– Advanced architecture reconstruction integrates both explicit and implicit dependencies, enhancing understanding and decision-making in microservice architectures.

4.4 Architectural Rule Violation Detection Strategies

Architectural rule violation detection is critical in ensuring that the software development adheres to predefined architectural guidelines. Such rules might dictate that service layers communicate solely through specified interfaces to avoid unauthorized data layer access directly from the UI layer. By categorizing these rules based on their criticality, developers can prioritize efforts to address the most significant threats to architectural integrity.

The effectiveness of these rules largely hinges on their comprehensiveness in encompassing all relevant architectural dependencies. Missed or overlooked dependencies can diminish the rules' effectiveness, leading to unintended breaches in architectural enforcement. Consequently, thorough and ongoing cataloging of all system dependencies is imperative to maintain the efficacy and authority of these rules.

Moreover, the governance of architectural rules demands a proactive approach, akin to the management of the system's codebase. As the architecture evolves, these rules may become obsolete or misaligned with the current system design, necessitating regular updates and maintenance. Establishing a robust maintenance strategy for these rules is essential, ensuring they accurately reflect the latest architectural standards and practices and remain relevant and effective in guiding system development.

Highlights:

– Proper governance and continuous refinement of architectural rules ensure adherence to best practices, enhancing the integrity and relevance of microservice architectures.

4.5 Implications

The strategies currently used to detect architectural degradation often operate at a high level of abstraction, presenting challenges when applied to decentralized systems with independent microservices. For effective architecture reconstruction, it is necessary to individually access and review each microservice's codebase—a task that becomes increasingly complex with the rapid evolution of these systems. Similarly, the enforcement and monitoring of architectural rules require substantial effort. While static analysis provides a preliminary approximation, it often struggles to accommodate the diversity of platforms and the nascent state of supporting toolsets.

In contrast, dependencies represent more granular and fundamental elements within the system's architecture, serving as a foundational layer for developing new analytical tools. These dependencies facilitate the identification of system issues, underlying causes, and potential anti-patterns without needing a pre-existing catalog. When integrated with architectural rules, dependencies offer a more dynamic view of the system's health without imposing the significant maintenance burden typically associated with high-level strategies.

Furthermore, by employing an enhanced system representation that includes meticulously tracked dependencies, it becomes possible to apply existing metrics across

the entire system, not just isolated components. This approach allows for a more holistic view of system health and can significantly improve the precision and relevance of diagnostic efforts. Ultimately, recognizing and managing dependencies not only aids in immediate problem resolution but also enhances the adaptability and resilience of the architectural framework, ensuring it can evolve in alignment with emerging needs and technologies.

5 Navigating the Maintainability Challenges of Microservice Dependencies

Effective dependency management is critical to maintaining the integrity and performance of microservice architectures. Furthermore, various benchmarking tools have been developed to detect, analyze, and manage these dependencies, each catering to specific aspects of microservice maintainability. Dependencies can significantly affect system maintainability, leading to increased complexity, higher costs, and longer times for updates and fixes. This section discusses different types of dependencies and the benchmarking tools designed to handle them, highlighting their use cases and the specific challenges they address.

For illustration, a fraud detection system in a financial services context monitors transactions for potential fraudulent activities, as demonstrated in Fig. 3. The system consists of multiple microservices that interact with each other to detect, handle, and log fraudulent transactions. The primary service, Service A (Transaction Monitoring Service), monitors all transactions and communicates with other services to take appropriate actions upon detecting a potential fraud. Service A connects with Service B (Account Service) to fetch and update account information, ensuring the account associated with the flagged transaction is managed appropriately. It also communicates with Service C (Alert Service) to alert customers and internal security teams about fraudulent activity. Finally, Service A logs the details of the flagged transaction by communicating with Service D (Audit Service), ensuring that all activities are recorded for auditing purposes. Different dependencies will be describe from this example in the following section.

5.1 Data Dependencies

Data dependencies arise when microservices share or access the same data entities, leading to maintainability issues as changes in data schemas must be coordinated across services. Tools like the Endpoint Dependency Matrix (EDM) and Data Dependency Matrix (DDM) [3] effectively track and visualize these dependencies, helping to understand how data flows between services. For instance, a company might use EDM and DDM to map how customer data is accessed and modified across various microservices, ensuring changes are synchronized to prevent inconsistencies and system failures. In Fig. 3 highlighted as dashed line, Service A (Transaction Monitoring Service) leans heavily on Service B (Account Service) for account information to evaluate transactions for fraud. At the same time, Service C (Alert Service) and Service D (Audit Service) also bear the weight of transaction and account information. This underscores the significant responsibilities of each service in the system.

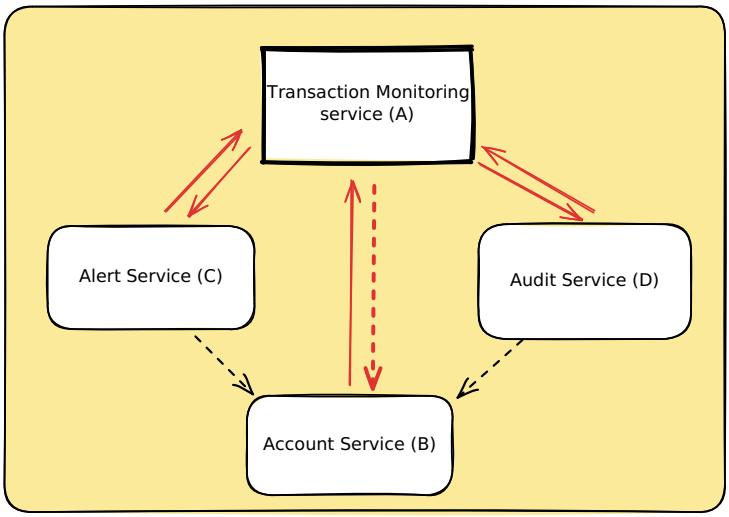

Fig. 3. A Fraud Detection System with Dependencies.

Notably, service design patterns like Tolerant Reader and Request Mapper can be employed to enhance robustness against changes in data formats. According to Daigneau [16], the Tolerant Reader pattern involves designing services that are lenient in what they accept in incoming messages, allowing the service to handle unknown or extra data gracefully without breaking. This pattern increases the resilience of services to changes in data formats and helps maintain compatibility as services evolve.

Similarly, the Request Mapper pattern transforms incoming requests into a format that the service can process effectively. This decouples the internal logic of the service from external data formats, providing a layer of abstraction that simplifies handling data dependencies. By using Request Mapper, services can adapt to changes in data structures more efficiently, reducing the maintenance burden associated with evolving data schemas. These patterns, combined with tools like EDM and DDM, offer a robust approach to managing data dependencies, ensuring services remain flexible and resilient, thus improving overall system maintainability.

5.2 Control Dependencies

Control dependencies arise when the execution of one service is dependent on the control flow or state of another. This type of dependency can complicate maintainability, as it requires a deep understanding of the interaction between services. Figure 3 illustrates that Service A (Transaction Monitoring Service), the pivotal player, controls the workflow by initiating actions in dependent services after flagging a transaction for potential fraud. This underscores the crucial role of Service A in the system.

Furthermore, TraceNet [48] can support the identification of issues in service interactions. By analyzing tracing data, it can pinpoint the root cause of problems, making it easier to visualize the flow of requests across services and locate where failures or performance bottlenecks occur. GDC-DVF [36], on the other hand, focuses on mapping invocation relationships and extracting dependencies, aiding in the

decomposition of monolithic applications into microservices. This dual-view fusion approach provides a comprehensive understanding of both the static and dynamic aspects of service interactions, essential for managing complex control dependencies.

For instance, a financial services firm can use TraceNet to trace transaction flows across microservices, identifying exactly where issues arise and understanding the sequence of service calls. GDC-DVF helps visualize and manage the interactions between services, which is particularly useful during their transition from a monolithic to a microservice architecture. This insight is critical for ensuring that the control flows are maintained correctly and that changes in one service do not inadvertently impact others.

5.3 Communication Dependencies

Communication dependencies involve service interactions, such as API calls and message passing. These dependencies are characterized by the exchange of information between services, often through synchronous or asynchronous communication mechanisms. Unlike control dependencies, which influence the execution flow directly, communication dependencies are about the exchange of data or messages between services to achieve a task. As in Fig. 3 highlighted in red lines, Service A (Transaction Monitoring Service) communicates with Service B (Account Service) to fetch account information, with Service C (Alert Service) to send details of flagged transactions, and with Service D (Audit Service) to log transaction details.

ChainsFormer [40] analyzes communication-based interactions among microservices to identify critical chains and nodes, facilitating resource provisioning and dynamic scaling. By understanding the communication patterns and identifying critical paths, ChainsFormer helps optimize the allocation of resources and improve overall system performance.

On the other hand, GSMART [29] creates service dependency graphs (SDGs) to visualize microservice-based systems, aiding in tracing relationships and selecting regression test cases. This visualization helps identify and address potential communication bottlenecks by mapping service interactions. GSMART's ability to trace and visualize these interactions makes it an invaluable tool for understanding the dependencies that arise from service communications and ensuring they are managed effectively.

To illustrate, an e-commerce platform could use ChainsFormer to analyze peak traffic times and ensure critical services are adequately resourced. By identifying the most heavily used service interactions, ChainsFormer enables the proactive scaling of resources to meet demand. GSMART, on the other hand, can be used to identify communication bottlenecks and optimize the flow of information between services, ensuring smooth operation during high-traffic periods.

5.4 Resource Dependencies

Resource dependencies are related to the shared use of computational resources such as CPU, memory, and storage. These dependencies can cause resource contention and performance degradation, making the system harder to maintain. DeepScaler utilizes

affinity matrices to scale microservices dynamically based on service affinities and resource usage patterns. This tool continuously monitors resource usage and adjusts the allocation dynamically, ensuring efficient use of resources. The sample microservice described in Fig. 3 demonstrates how all services utilize shared CPU and memory resources to run fraud detection algorithms efficiently.

SYMBIOTE [6] continuously analyzes service coupling metrics to detect potential architectural degradation and optimize resource allocation. By monitoring the degree of coupling between services, SYMBIOTE helps identify areas where resource contention might occur and pre-emptively address these issues. This continuous monitoring and adjustment help maintain optimal performance and resource utilization.

For example, a cloud service provider can use DeepScaler [31] to monitor and adjust resource allocation dynamically based on real-time usage patterns. This not only ensures that services receive the resources they need without over-allocating, but also helps in maintaining efficiency and reducing costs. SYMBIOTE helps maintain optimal service performance by monitoring and adjusting service couplings, ensuring that resource dependencies are managed effectively to prevent performance degradation.

Table 1. Overview of Dependency Types and Their Benchmarking Tools.

Dependency Type	Description	Benchmarking Tools
Data Dependencies	Occur when microservices share or access the same data entities.	Endpoint Dependency Matrix (EDM), Data Dependency Matrix (DDM)
Control Dependencies	Arise when the execution of one service is dependent on the control flow or state of another.	TraceNet, GDC-DVF
Communication Dependencies	Involve interactions between services, such as API calls and message passing.	ChainsFormer, GSMART
Resource Dependencies	Related to the shared use of computational resources like CPU, memory, and storage.	DeepScaler, SYMBIOTE

In summary, each type of dependency introduces unique challenges to the maintainability of microservice architectures where different dependencies overview is presented in Table 1. By leveraging tools like EDM, DDM, TraceNet, GDC-DVF, ChainsFormer, GSMART, DeepScaler, and SYMBIOTE, developers and architects can better manage these dependencies. Each tool offers specific capabilities tailored to different aspects of dependency management, enabling a comprehensive approach to maintaining robust and scalable microservice architectures. Understanding and applying these tools in appropriate use cases ensures that microservice systems remain maintainable, performant, and resilient over time.

6 Maintainability in the Context of Microservice Dependencies

Maintainability in microservice architectures (MSA) is crucial due to the distributed nature of the systems, where services are developed, deployed, and scaled independently. The ease of modifying, extending, or updating software systems in MSA is significantly influenced by the architecture's ability to manage complex and highly intertwined dependencies among services. Effective change impact analysis is paramount in MSA. This process extends beyond simple assessment of modifications; it actively pinpoints and evaluates potential impacts of changes on the system's overall architecture. This is critical in MSA environments, where changes in one service can affect multiple other services, potentially leading to significant system-wide impacts. Such analysis supports maintaining system integrity and minimizing error introduction during updates, ensuring the system remains robust, adaptable, and easier to maintain over time.

In these settings, dependencies act as clear indicators of how changes might affect system maintainability. They provide detailed insights into which specific areas of the system could be impacted by proposed changes. This level of precision is instrumental in pinpointing the root causes of potential issues with maintainability, thereby enabling IT leaders and developers to make informed decisions about the trade-offs involved in implementing changes.

Dependencies offer a more granular understanding than broader metrics or the identification of code smells, which might only hint at potential issues. By clearly outlining the direct relationships between changes and their impacts, dependencies guide strategic planning and the evaluation of alternative solutions or design modifications that could prevent negative outcomes. This focus on dependencies not only enhances the agility and responsiveness of the development process but also ensures that the architecture remains robust and adaptable to new requirements and challenges.

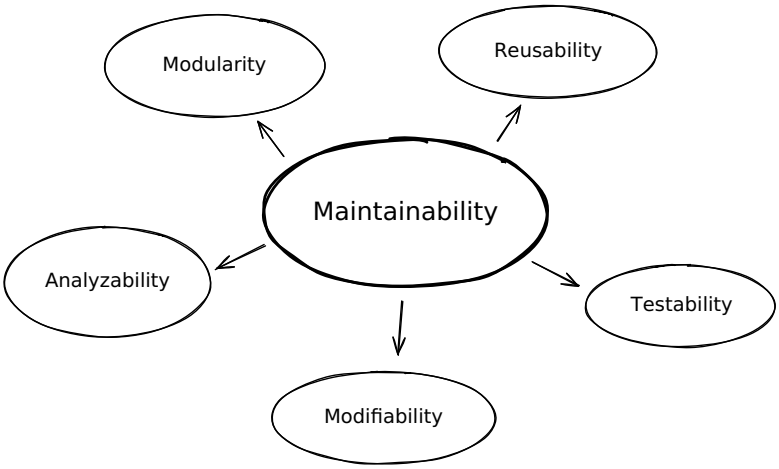

Fig. 4. Quality attribute for maintainability according to ISO 25010.

Maintainability, as characterized by attributes such as *modularity*, *reusability*, *analysability*, *modifiability*, and *testability*, as defined by Bass et al. and the ISO 25010 standard, is crucial in microservices architecture [10]. In this context, managing dependencies refers to the processes involved in identifying, understanding, and controlling the relationships and interactions between different microservices. Effective dependency management ensures that changes in one service minimize disruptions to others, thereby enhancing system integrity and flexibility. This includes using tools and practices such as service registries, automated testing, and continuous integration to handle dependencies proactively. Understanding these quality attributes, as illustrated in Fig. 4, is fundamental to maintaining and improving system performance.

Modularity is one of the cornerstone attributes of maintainability. It involves segmenting a complex system into smaller, independent, and replaceable units, designed to minimize impact on other components when changes occur. In microservices, modularity is achieved by designing each service to perform a distinct function, allowing it to operate independently yet harmoniously within the larger ecosystem. While modularity aims to improve maintainability, reusability, and clarity, realizing this attribute in cloud-native environments presents challenges. The interconnected nature of microservices and the widespread distribution of global policies and knowledge, such as role-based access control (RBAC), complicate the modular design. RBAC can enforce a degree of interdependence among services by requiring them to adhere to shared security protocols and access rules, thereby complicating the modular design by potentially introducing tight coupling based on access controls.

Understanding the dependencies between these modular units is crucial; however, simply identifying these dependencies does not mitigate the impact of changes across the system. While dependency tracking allows for a clearer understanding of potential ripple effects from changes in one service, it does not inherently reduce the broad impact of these changes. Updates to services with many dependencies might still necessitate coordinated updates across multiple consumer services, thus presenting challenges in maintaining true independence and modularity.

To effectively manage the modularity of a microservice-based system, understanding the web of dependencies across services is crucial. While meticulous tracking of these dependencies does not reduce the impact of updates on interconnected services, it does facilitate informed decision-making and strategic planning. When updates are inevitable, this awareness helps prepare for and manage potential cascading effects across the system. A potential strategy to balance modularity with the demands of cloud-native principles, such as those proposed in the Twelve-Factor App methodology [47], involves centralizing shared logic in importable libraries. This approach should be carefully executed to preserve the independence of services and align with cloud-native architectural principles. By doing so, the system not only adheres to microservices best practices but also retains the flexibility to adapt and evolve, ensuring robustness despite the inherent complexities of interconnected services.

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

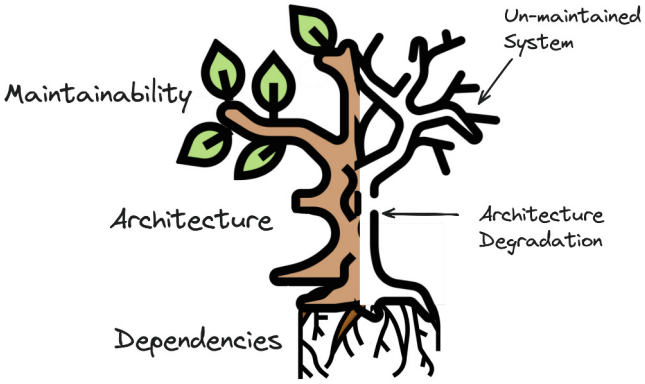

Fig. 5. Visualizing root cause of architectural degradation [50].

Reusability is a core principle of software engineering, traditionally aiming to save development time and resources by allowing components to be used across various parts of a system or in multiple projects. This principle enhances efficiency and reduces redundancy. However, in the context of microservices, excessive reusability can lead to increased dependencies, which is problematic. Microservices advocate a "share-as-little-as-possible" approach to minimize coupling and enhance each service's ability to evolve independently. Excessive reuse of services and shared libraries can complicate this, leading to tight coupling that might hinder the independent scalability and evolution of services [44].

While modularity supports reusability by defining clear, independent units, the transition to microservices often complicates its application due to potential latency and system bottlenecks. To address these challenges, the serverless computing model presents a promising solution. Unlike traditional server-based models, serverless computing executes backend services on-demand without requiring developers to manage server infrastructure. This model facilitates the reuse of functions across different parts of a system or even across projects by removing the overhead of infrastructure management, thus alleviating common challenges associated with microservices and enhancing reusability in modern software development environments.

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

Analysability measures how easily a software system can be inspected for understanding its structure, diagnosing issues, and implementing improvements. In microservice architectures, where services are decentralized and duties are separated, developers can effectively manage their specific components but often lose sight of the overall system architecture. This decentralization challenges the thorough analysability of the entire system.

Tracing and logging are commonly used to enhance analysability in microservices; however, these methods depend heavily on the consistency of the developers'

logging practices and their readiness to potentially compromise performance for better traceability. Static analysis tools, which provide quick insights into the code's structure without execution, often fall short in microservices environments due to their complexity and distributed nature.

To improve analysability in such systems, there is a need for tools and practices specifically tailored to address the unique challenges of distributed architectures. This includes developing more sophisticated tracing mechanisms and encouraging a culture of detailed and consistent logging among developers.

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

Modifiability refers to the ease with which a software system can be altered or adapted to meet evolving requirements. This attribute is vital as software must frequently evolve to incorporate new features, fix bugs, and adapt to changing user needs. A system with high modifiability allows developers to implement changes quickly and efficiently, minimizing the risk of errors and reducing the time and effort needed for updates.

In the context of microservices, effective dependency management plays a significant role in enhancing modifiability. Properly managing dependencies enables comprehensive change impact analysis, which helps to mitigate the negative effects of improper design. This analysis identifies how changes in one service affect others, allowing for more informed decision-making and reducing potential disruptions.

However, to conduct effective change impact analysis, a thorough understanding of the system's dependencies is required. This underscores the importance of initially analyzing the system to map out all dependencies. By having a clear picture of these interconnections, developers can better predict the impact of changes, ensure smoother modifications, and maintain the system's overall integrity and performance.

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

Testability refers to how easily a software system can be tested to ensure it meets its requirements and behaves as intended. A highly testable system simplifies the creation, execution, and maintenance of tests.

In microservice architectures, testability and dependency management intersect significantly. Tests are resource-intensive, and static analysis can sometimes offer insights with fewer resources. However, tests still require maintenance, and changes in the system often impact which tests need to be run or updated.

Proper dependency management can streamline the testing process by pinpointing which specific tests are affected by system changes. This allows for targeted testing, reducing the need for comprehensive system tests and conserving resources. For example, changes in one microservice might only necessitate tests related to that specific service, rather than a full system retest.

Moreover, understanding the dependencies between system components and tests helps identify which tests need modification when changes occur, especially in

end-to-end testing scenarios. This ensures that testing efforts remain focused and efficient, maintaining system integrity with optimal resource usage.

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

Figure 5 illustrates the concept of architectural degradation and its impact on system maintainability. In this visual metaphor, the roots symbolize system dependencies, which are crucial for supporting and stabilizing the architecture, represented by the tree's trunk. As the trunk branches out, it signifies the architecture of the system, while the leaves represent maintainability. Barren branches indicate an unmaintained system, suggesting that neglecting the underlying dependencies results in the deterioration of the system's structural integrity. This emphasizes that the health of the system's architecture relies heavily on the careful management of its dependencies. Poor management can lead to architectural degradation and a subsequent decline in maintainability.

7 Conclusions

In this manuscript, we have underscored the critical role of dependency management in enhancing maintainability within microservice systems. As systems evolve, mitigating architectural degradation becomes essential, and while various approaches have been developed for monolithic repository systems, there remains a significant gap in strategies tailored for microservices. We have explored the potential of effective dependency management to support change impact analysis, emphasizing its importance in decentralized systems.

By utilizing managed dependencies as a tool to assess changes, developers can achieve better outcomes more efficiently, ensuring that modifications do not inadvertently compromise system integrity. However, this perspective indicates the need for further research.

Future research should focus on developing scalable and adaptable dependency management strategies for microservices that can keep pace with technological advancements. Addressing these research gaps will not only assist developers in overcoming current challenges but also contribute to the creation of more resilient and maintainable microservice architectures.

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

References

1. ISO 25000 portal (2023). https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability. Accessed 20 Dec 2023
2. Microservices adoption in 2020 (2023). https://www.oreilly.com/radar/microservices-adoption-in-2020/. Accessed 20 Dec 2023

3. Abdelfattah, A.S., Cerny, T.: The microservice dependency matrix. In: European Conference on Service-Oriented and Cloud Computing, pp. 276–288. Springer (2023)
4. Al Maruf, A., Bakhtin, A., Cerny, T., Taibi, D.: Using microservice telemetry data for system dynamic analysis. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 29–38. IEEE (2022)
5. Alshuqayran, N., Ali, N., Evans, R.: Towards micro service architecture recovery: an empirical study. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 47–4709 (2018). https://doi.org/10.1109/ICSA.2018.00014
6. Apolinário, D.R., de França, B.B.: A method for monitoring the coupling evolution of microservice-based architectures. J. Braz. Comput. Soc. **27**(1), 17 (2021)
7. d Aragona, D.A., Pascarella, L., Janes, A., Lenarduzzi, V., Penaloza, R., Taibi, D.: On the empirical evidence of microservice logical coupling. a registered report (2023)
8. Azadi, U., Fontana, F.A., Taibi, D.: Architectural smells detected by tools: a catalogue proposal. In: Proceedings of the Scientific Workshop Proceedings of XP2016. XP 2016 Workshops, IEEE Press (2019). https://doi.org/10.1109/TechDebt.2019.00027
9. Baabad, A., Zulzalil, H.B., Hassan, S., Baharom, S.B.: Software architecture degradation in open source software: a systematic literature review. IEEE Access **8**, 173681–173709 (2020). https://doi.org/10.1109/ACCESS.2020.3024671
10. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice: Software Architect Practice_c4. Addison-Wesley (2021)
11. Besker, T., Martini, A., Bosch, J.: 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 2018, pp. 105–114. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3194164.3194178
12. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Industry practices and challenges for the evolvability assurance of microservices. Empirical Softw. Eng. **26**(5), 104 (2021). https://doi.org/10.1007/s10664-021-09999-9
13. Cerny, T., Abdelfattah, A.S., Maruf, A.A., Janes, A., Taibi, D.: Catalog and detection techniques of microservice anti-patterns and bad smells: a tertiary study. J. Syst. Softw. **206**, 111829 (2023). https://doi.org/10.1016/j.jss.2023.111829. https://www.sciencedirect.com/science/article/pii/S0164121223002248
14. Cerny, T., Donahoo, M.J., Trnka, M.: Contextual understanding of microservice architecture: current and future directions. ACM SIGAPP Appl. Comput. Rev. **17**(4), 29–45 (2018)
15. Conway, M.E.: How do committees invent. Datamation **14**(4), 28–31 (1968)
16. Daigneau, R.: Service Design Patterns: fundamental design solutions for SOAP. Addison-Wesley (2012)
17. Das, D., et al.: Technical debt resulting from architectural degradation and code smells: a systematic mapping study. SIGAPP Appl. Comput. Rev. **21**(4), 20–36 (2022). https://doi.org/10.1145/3512753.3512755
18. Dijkstra, E.W.: On the Role of Scientific Thought, pp. 60–66. Springer, New York (1982). https://doi.org/10.1007/978-1-4612-5695-3_12
19. Fontana, F.A., Roveda, R., Vittori, S., Metelli, A., Saldarini, S., Mazzei, F.: On evaluating the impact of the refactoring of architectural problems on software quality. In: Proceedings of the Scientific Workshop Proceedings of XP2016. XP 2016 Workshops, Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2962695.2962716
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: abstraction and reuse of object-oriented design. In: Nierstrasz, O.M. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 406–431. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-47910-4_21
21. Garlan, D., Shaw, M.: An introduction to software architecture. In: Advances in Software Engineering and Knowledge Engineering, pp. 1–39. World Scientific (1993)

22. Haendler, T., Sobernig, S., Strembeck, M.: Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In: Proceedings of the XP2017 Scientific Workshops. XP 2017, Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3120459.3120468
23. Hohpe, G.: Event-driven = loosely coupled? not so fast! In: Enterprise Integration Patterns (2023). https://www.enterpriseintegrationpatterns.com/ramblings/eventdriven_coupling.html
24. Hohpe, G.: The many facets of coupling. Enterprise Integration Patterns (2023). https://www.enterpriseintegrationpatterns.com/ramblings/coupling_facets.html
25. Larman, C., et al.: Applying UML and Patterns, vol. 2. Prentice Hall, Upper Saddle River (1998)
26. Lenarduzzi, V., Sievi-Korte, O.: On the negative impact of team independence in microservices software development. In: Proceedings of the 19th International Conference on Agile Software Development: Companion, pp. 1–4 (2018)
27. Li, Z., Liang, P., Avgeriou, P., Guelfi, N., Ampatzoglou, A.: 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, pp. 119–128. QoSA 2014, Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2602576.2602581
28. Lv, W., et al.: Graph-reinforcement-learning-based dependency-aware microservice deployment in edge computing. IEEE Internet Things J. **11**(1), 1604–1615 (2023)
29. Ma, S.P., Fan, C.Y., Chuang, Y., Liu, I.H., Lan, C.W.: Graph-based and scenario-driven microservice analysis, retrieval, and testing. Futur. Gener. Comput. Syst. **100**, 724–735 (2019)
30. Martini, A., Sikander, E., Madlani, N.: A semi-automated framework for the identification and estimation of architectural technical debt: a comparative case-study on the modularization of a software component. Inf. Softw. Technol. **93**, 264–279 (2018). https://doi.org/10.1016/j.infsof.2017.08.005. https://www.sciencedirect.com/science/article/pii/S095058491630355X
31. Meng, C., Song, S., Tong, H., Pan, M., Yu, Y.: Deepscaler: holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 53–65. IEEE (2023)
32. Newman, S.: Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly Media, Sebastopol (2015)
33. Panichella, S., Rahman, M.I., Taibi, D.: Structural coupling for microservices. arXiv preprint arXiv:2103.04674 (2021)
34. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972). https://doi.org/10.1145/361598.361623
35. Parnas, D., Clements, P., Weiss, D.: The modular structure of complex systems. IEEE Trans. Softw. Eng. **SE-11**(3), 259–266 (1985). https://doi.org/10.1109/TSE.1985.232209
36. Qian, L., Li, J., He, X., Gu, R., Shao, J., Lu, Y.: Microservice extraction using graph deep clustering based on dual view fusion. Inf. Softw. Technol. **158**, 107171 (2023)
37. Rademacher, F., Sachweh, S., Zündorf, A.: A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Enterprise, Business-Process and Information Systems Modeling, pp. 311–326. Springer, Cham (2020)
38. Rademacher, F., Wizenty, P., Sorgalla, J., Sachweh, S., Zündorf, A.: Model-Driven Engineering of Microservice Architectures—The LEMMA Approach, pp. 105–147. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-44412-8_5
39. Schiewe, M., Curtis, J., Bushong, V., Cerny, T.: Advancing static code analysis with language-agnostic component identification. IEEE Access **10**, 30743–30761 (2022)

40. Song, C., et al.: Chainsformer: a chain latency-aware resource provisioning approach for microservices cluster. In: International Conference on Service-Oriented Computing, pp. 197–211. Springer (2023)
41. Song, Y., Li, C., Zhuang, K., Ma, T., Wo, T.: An automatic scaling system for online application with microservices architecture. In: 2022 IEEE International Conference on Joint Cloud Computing (JCC), pp. 73–78. IEEE (2022)
42. Terzić, B., Dimitrieski, V., Kordić (Aleksić), S., Luković, I.: A model-driven approach to microservice software architecture establishment, pp. 73–80 (2018). https://doi.org/10.15439/2018F370
43. Tighilt, R., Abdellatif, M., Trabelsi, I., Madern, L., Moha, N., Guéhéneuc, Y.G.: On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. J. Syst. Softw. **204**, 111755 (2023). https://doi.org/10.1016/j.jss.2023.111755. https://www.sciencedirect.com/science/article/pii/S0164121223001504
44. de Toledo, S.S., Martini, A., Sjøberg, D.I.K.: Improving agility by managing shared libraries in microservices. In: Paasivaara, M., Kruchten, P. (eds.) Agile Processes in Software Engineering and Extreme Programming - Workshops, pp. 195–202. Springer, Cham (2020)
45. Walker, A., Das, D., Cerny, T.: Automated code-smell detection in microservices through static analysis: a case study. Appl. Sci. **10**(21), 7800 (2020)
46. Walker, A., Laird, I., Cerny, T.: On automatic software architecture reconstruction of microservice applications. In: Information Science and Applications: Proceedings of ICISA 2020, vol. 739, p. 223 (2021)
47. Wurster, M., Breitenbücher, U., Falkenthal, M., Leymann, F.: Developing, deploying, and operating twelve-factor applications with tosca, pp. 519–525 (2017). https://doi.org/10.1145/3151759.3151830
48. Yang, J., Guo, Y., Chen, Y., Zhao, Y.: Tracenet: operation aware root cause localization of microservice system anomalies. In: 2023 IEEE International Conference on Communications Workshops (ICC Workshops), pp. 758–763. IEEE (2023)
49. Zhou, J., Wang, G., Zhou, W.: Dependency-aware microservice deployment and resource allocation in distributed edge networks. In: 2023 International Wireless Communications and Mobile Computing (IWCMC), pp. 568–573. IEEE (2023)
50. Černý, T., Chy, M.S.H., Abdelfattah, A., Soldani, J., Bogner, J.: On maintainability and microservice dependencies: how do changes propagate? pp. 277–286 (2024). https://doi.org/10.5220/0012725200003711