UMEÅ UNIVERSITY

Towards Self-Driving Microservices

Mohammad Reza Saleh Sedghpour

Doctoral Thesis, May 2023
Department of Computing Science
Umeå University
Sweden

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

`https://people.cs.umu.se/msaleh/`
msaleh@cs.umu.se

ISBN: 978-91-8070-022-1 (print)
978-91-8070-023-8 (digital)
ISSN: 0348-0542
UMINF: 23.04

Cover illustration by Ida Åberg.

The papers in this thesis have been re-typeset to match the overall style of the thesis with permission granted by the copyright holders.

Printed by Cityprint i Norr AB, Umeå, 2023

"In the name of the God of rainbows."
In remembrance of Kian Pirfalak.

Abstract

In recent years, microservice architecture has become a popular method for software system design and development. This involves creating applications with multiple small services, each with multiple instances, operating as independent processes. Due to the distributed nature of microservices, communication between services presents a challenging task that becomes increasingly complex as the number of services grows. This complexity can even lead to short-term failures that can degrade application performance. Therefore, the auto-tuning of inter-service communication is necessary to prevent such failures. Service meshes were introduced to offer the necessary technical capabilities that can be employed in such scenarios. In essence, a service mesh is an infrastructure layer that includes a set of configurable proxies integrated into microservices. This enables the provision of traffic management policies such as circuit breaking and retry mechanisms to enhance microservice resilience against transient failures. However, static configuration or misconfiguration of these mechanisms is unsuitable for the dynamic environment of microservices and can lead to serious issues and performance problems, such as retry storms.

The goal of this thesis is three-fold. First, it aims to investigate the impact and effectiveness of service traffic management on application reliability and availability in the presence of transient failures. Second, it focuses on auto-tuning of service traffic management to increase carried throughput and maintain carried response time. Third, this research aims to propose measures that can improve research reproducibility in the area of distributed systems ensuring that the findings can be independently verified by others. In this thesis, we aim to offer detailed guidelines on best practices for implementing research software.

To achieve these goals, this thesis delves into the current state-of-the-art in service meshes and eBPF-powered microservices, identifying current challenges and potential future directions. It analyzes the effects of circuit breaker and retry mechanisms on microservice performance and proposes adaptive controllers for both. The results show the need for such controllers that increase throughput while maintaining the tail response time of the application. Additionally, it proposes a microservice benchmark generator to enable systematic microservice benchmark generation and improve reproducibility. It also provides recommendations for improving artifact evaluation in distributed systems research by compiling all existing recommendations.

Sammanfattning

Mikrotjänster har de senaste åren blivit en populär arkitekturmodell för programvara. Modellen innebär att man skapar applikationer med flera små tjänster, var och en med flera instanser som fungerar som oberoende processer. Mikrotjänsters distribuerade natur gör kommunikationen mellan tjänster mer utmanande. Denna komplexitet kan även ge upphov till tillfälliga fel på grund av lastobalans eller överbelastning och även försämra applikationers prestanda. Av denna anledning är dynamisk konfiguration av kommunikationen mellan mikrotjänster nödvändig. En service mesh är en teknikplattform för att hantera hur mikrotjänster kommunicerar, med funktioner för att enkelt kryptera kommunikation mellan tjänster, mäta prestanda och finkorningt styra kommunikationsflöden. En service mesh implementeras ofta som en uppsättning konfigurerbara proxies. Detta möjliggör trafikhanteringspolicies baserade på mekanismer som kretsbrytning och omsändningar. Statisk konfiguration av dessa mekanismer kan dock ge allvarliga prestandaproblem såsom låg genomströmning och/eller omfattande omsändningar.

Denna avhandling har tre mål. För det första undersöker den hur trafikhantering för mikrotjänster påverkar tillförlitligheten och tillgängligheten för applikationer vid tillfälliga störningar. För det andra fokuserar den på adaptiv reglering av trafikhantering av mikrotjänster för att öka genomströmningen och samtidigt bibehålla acceptabla svarstider. För det tredje syftar den till att förbättra reproducerbarheten i forskning inom distribuerade system och se till att forskningsresultat enklare kan verifieras av oberoende.

För att uppnå dessa mål undersöker avhandlingen den tekniska frontlinjen inom service mesh och mikrotjänster driva av eBPF-tekniken. Avhandlingen analyserar vidare hur användandet av kretsbrytare och omsändningsmekanismer påverkar mikrotjänsters prestanda. Adaptiva reglersystem för att hantera konfiguration av båda dessa mekanismer föreslås och utvärderas i omfattande experimentent. Resultaten visar att sådana regulatorer är nödvändiga för att öka genomströmningen och samtidigt bibehålla (högre percentiler av) applikationers svarstider. Adaption är särskilt viktigt då faktorer som totala trafikmängden, applikationers prestanda, tillfälliga fel, etc. kan förändras snabbt.

Avhandlingen introducerar även ett verktyg för att generera godtyckliga testapplikationer för att kunna genomföra mer heltäckande utvärderingar av olika typer av forskningsprogramvara som hanterar mikrotjänster. Avhandlingen bidrar även till reproducerbarhet genom att studera hur programvaru-artefakter bäst bör utvärderas inom forskningsområdet distribuerade system. Detta sker genom att sammanställa, och utöka, befintliga rekommendationer inom området.

Preface

The main goal of the thesis is to enhance the resiliency of microservices and enable them to be self-driving, particularly in case of transient failures. The aim is to make microservices more robust and able to recover from failures quickly and independently, without the need for manual intervention by a human operator. To achieve this goal, the thesis includes six papers that explore different aspects of microservices resiliency that are listed below:

Paper I	**M. R. Saleh Sedghpour** and P. Townend. Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions. *Proceedings of the 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE, 2022)*, IEEE, pp. 176-184, 2022.
Paper II	**M. R. Saleh Sedghpour**, C. Klein, and J. Tordsson. An Empirical Study of Service Mesh Traffic Management Policies for Microservices. *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering (ICPE, 2022)*, ACM, pp. 17-27, 2022.
Paper III	**M. R. Saleh Sedghpour**, C. Klein and J. Tordsson. Service Mesh Circuit Breaker: From Panic Button to Performance Management Tool. *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems (HAOC, 2021)*, ACM, pp. 4-10, 2021.
Paper IV	**M. R. Saleh Sedghpour**, D. Garlan, B. Schmerl, C. Klein, and J. Tordsson. Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry. *Submitted*, Umeå University, 2023.
Paper V	**M. R. Saleh Sedghpour**, A. O. Duque, X. Cai, B. Skubic, E. Elmroth, C. Klein and J. Tordsson. HydraGen: A Microservice Benchmark Generator. *Submitted*, Umeå University, 2023.
Paper VI	**M. R. Saleh Sedghpour**, C. Klein, A. V. Papadopoulos, and J. Tordsson. Artifact Evaluation for Distributed Systems: Current Practices and Beyond. *Submitted*, Umeå University, 2023.

In addition to the papers included in this thesis listed above, the following presentations arose from work conducted by Mohammad Reza during his doctoral studies:

Talk I	**M. R. Saleh Sedghpour**. Tune your Service Mesh. *ServiceMeshCon at KubeCon + CloudNativeCon EU*, 2022.
Talk II	**M. R. Saleh Sedghpour**, S. I. Ulfsparre. Integration of Research software into the EOSC infrastructure: Lessons learned from Computer science. *European Open Science Cloud Symposium (EOSC Symposium, 2022)*, 2022.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and some works supported by the Google Cloud Research Credits program.

Acknowledgements

It is with great gratitude and appreciation that I extend my sincerest thanks to those who have helped me along the way, guiding me through the twists and turns of my academic journey. I am honored to have had their support and would like to express my sincere appreciation.

Firstly, I would like to express my gratitude to Johan Tordsson, my supervisor, for his unwavering support, guidance, mentorship, and, most importantly, his patience throughout my academic journey. Johan's insightful perspectives, constructive feedback, and constant encouragement have played a vital role in shaping my research and enabling me to achieve my academic objectives. Johan's unrelenting dedication to my success, and his willingness to go the extra (Swedish) mile to provide assistance and support, have been truly remarkable. I consider myself fortunate to have had such a committed and experienced mentor who has demonstrated a genuine interest in my academic and personal growth. I am grateful for the numerous opportunities that Johan has provided me with, and I intend to apply the knowledge and skills that I have gained from his mentorship throughout my life. Once again, thank you, Johan, for your invaluable contributions to my academic journey.

Secondly, I would like to express my heartfelt appreciation to my co-supervisor, Cristian Klein, for his invaluable contribution to my academic journey. Throughout our collaboration, Cristian has been a source of inspiration, guidance, and support, helping me to navigate the challenges of my research. Cristian's teaching has been transformational; he has helped me to develop a more robust and analytical approach to solving technical problems. I will always cherish our conversations on various technological advancements that have been both enlightening and fun.

During my time in Umeå, I was fortunate to have the opportunity to collaborate with a group of highly intelligent and talented individuals. Paul Townend, my first collaborator, has been an exceptional mentor who offered me the chance to co-author a paper with him. In addition, he provided me with training on academic evaluations, and I had the privilege of assisting him in courses. Under his guidance, I have gained a wealth of knowledge, and his support has been immeasurable. One of my primary collaborators was Aleksandra Obeso Duque, a fellow Ph.D. student, with whom I worked for more than a year to actually make an impact. Along with Aleksandra, I had the pleasure of collaborating with Erik Elmroth, Björn Skubic, and Xuejun Cai, who all brought unique perspectives and insights to the project. In addition, I was privileged to meet Alessandro Vittorio Papadopoulos, who offered me invaluable guidance and support. Alessandro's extensive knowledge and expertise were invaluable in helping me develop my research skills and improve the quality of my work. Finally, I had the honor of collaborating with David Garlan and Bradley Schmerl from Carnegie Mellon. Working with these experienced researchers, I learned to think outside the box and approach problems from new angles. Their support and guidance were instrumental in helping me to overcome some of the most challenging obstacles in my research.

I would like to express my gratitude to all current and former members of ADSLab for their invaluable support and contribution to my academic journey. First, our lab leader, Erik Elmroth, deserves special thanks for creating such an exceptional lab. I am especially thankful to Ali and Javad, who embarked on this journey with me, engaging me in insightful conversations and helping me learn a lot. Petter provided me not only with a comfortable living space during my initial days in Umeå but also offered valuable advice that helped me transition smoothly. Abel and Chanh's positive attitudes and sense of humor brought a welcoming energy to our lab, for which I am grateful. Antonio, as a senior researcher, inspired me with his amazing talks and encouragement, while Sourasekhar and Anindya were fantastic lab mates with whom I shared enjoyable moments. Charlie's extensive vocabulary and willingness to teach always fascinated me, and Monowar and P-O's valuable insights inspired me to experiment with new ideas. I would also like to express my gratitude to Simon for the fun times we shared. Lastly, I am grateful to Lars, Xuan-Son, Lidia, Jon, Tobias, Atakan, Adil, Magdalena, Sagar, Oliver, Obaidullah, Pim, and Yashwant for their valuable insights and contributions to our lab. Lastly, I would like to extend a special thanks to Tomas for his exceptional technical support, even during holidays, for which I am deeply grateful.

I would like to acknowledge and thank several individuals from the Department of Computing Science who have supported and contributed to my academic journey, including Paolo, Lili, Kai-Florian, Michele, Maarten, and Anne-Lie. I express my deep appreciation to Reza Berangi and Hamid Faragardi, whose help was instrumental in enabling me to come to Sweden as a Ph.D. student and start my academic journey. Additionally, I am grateful to the individuals who co-founded the Open Science Community, including Olivia, Birgitte, and particularly Sanna Isabel, whose insights and knowledge have been invaluable to me during our conversations and collaborations.

I am deeply grateful to my family, starting with my parents, Bahram and Nazi, who laid the foundation for everything I am today and supported me with unwavering patience throughout all of my decisions. I would also like to thank my sister, Farnaz, and my brother, Alireza, who have always believed in me and my potential. And a special thanks to my super smart nephew, Hosna.

Last but not least, I would like to express my deepest gratitude to my wife and partner, Fateme. With her by my side, as my best friend, I know that

anything is possible, even eternity. We have shared the journey of our Ph.D. studies together, supporting each other with love and having fun along the way.

I want to extend my heartfelt gratitude to everyone who has helped me along this journey, and I am incredibly happy with where I am. In every meaningful way, I feel that we did it together, and for that, I am truly grateful.

Mohammad Reza Saleh Sedghpour
Umeå, May 2023

Contents

Chapter 1

Introduction

Today, software is an essential tool that has transformed the way we work, communicate, and perform our daily tasks, largely due to remarkable advancements in hardware and undeniable progress in software engineering practices. These practices enable the timely production of high-quality computing products, further contributing to the widespread adoption and impact of software in various domains.

Software is however a victim of its own success. The evolution of software technology has brought a new level of complexity to software systems, which makes them more prone to vulnerabilities, errors, and malfunctions. The software systems of today are constantly exposed to new and demanding requirements, such as the need for high availability, reliability, security, scalability, and observability. This puts great pressure on software development and operation teams, making it more challenging to produce and maintain software systems that meet these requirements. In particular, the maintenance of software systems is becoming increasingly difficult and time-consuming due to their complexity and interdependence with other systems. These challenges have resulted in a growing demand for innovative practices and tools that can address the complexities and ensure the continuous delivery of software-based services. This has led to the emergence of DevOps, which seeks to integrate development and operations to improve the software development lifecycle [1].

DevOps is a software engineering approach that emphasizes collaboration and communication between software development and operations teams. It aims to facilitate the continuous delivery, deployment, and maintenance of software applications. DevOps combines practices such as agile software development, continuous integration, continuous delivery, and automated testing to provide a unified and streamlined approach to software lifecycle management [2] (see Figure 1.1). By implementing DevOps practices, organizations can achieve faster delivery of high-quality software, increased efficiency in development and deployment, reduced development costs, and improved collaboration between teams.

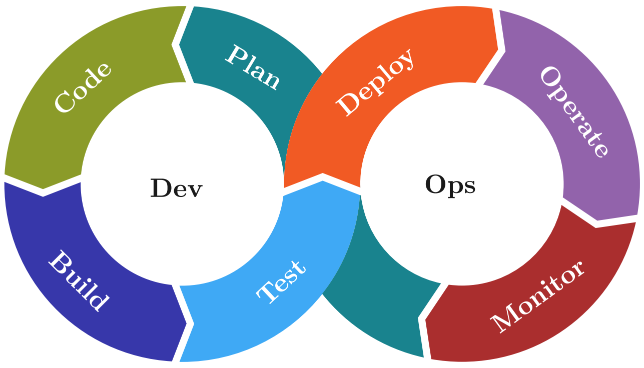


Figure 1.1: DevOps lifecycle.[1]

The design and construction of any complex software system, apart from its lifecycle, entail another crucial aspect, its architecture. This refers to the set of structures needed to reason about the system that comprises software elements, relations among them, and properties of both [3]. Over the past three decades, there has been a growing interest in software architecture, that has become a significant subfield of software engineering [4]. A well-structured architecture plays a significant role in ensuring that a system meets its critical functional and quality requirements, such as performance, reliability, portability, scalability, and interoperability. On the other hand, a poorly designed architecture can have disastrous consequences [3].

In software architecture, an architectural style is a collection of guidelines and patterns that direct the arrangement and development of a software system. In essence, the fundamental idea behind an architectural style is that it specifies the vocabulary of components and connectors that can be used in instances of that style, accompanied by a set of constraints on how they can be combined [5]. The most common architectural style is the layered architectural style. A layered architecture is a design approach that separates an application into different horizontal layers, with each layer having a specific role within the application. By separating the application into distinct layers, changes to one layer can be made without affecting the others, making it easier to maintain and update the application. Additionally, the layered architecture pattern promotes code reusability, as modules can be reused in other applications or in different layers of the same application [6]. While the layered architectural style does not prescribe a specific quantity or category of layers necessary for its implementation, a majority of layered architectures commonly include the presentation, business, persistence, and database layers (see Figure 1.2). The most commonly recognized instances of layered architecture styles are the layered communication protocols [7].

[1]Image by Kharnagy, licensed under CC BY-SA 4.0, modified by author. Original image available at https://commons.wikimedia.org/w/index.php?curid=51215412

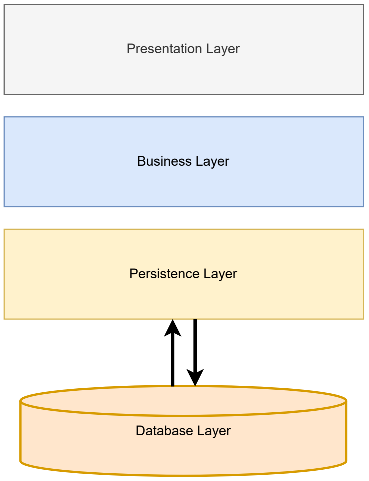


Figure 1.2: The most common layered architecture.

1.1 Monolithic Architectures

A common option in traditional software development for implementing the layered architectural style is monolithic architecture. A monolithic application is a software application that comprises a single and indivisible executable unit, where all the components are tightly integrated [8]. One of the main advantages of monolithic architecture is that it offers a simple and straightforward development process. Because all the components are developed as a single unit, it is easier to manage dependencies and ensure that the system is working as a whole. In addition, the monolithic architecture allows for easier debugging and testing, as the entire system can be tested at once rather than testing individual components in isolation. This architecture is still widely used in many applications, particularly in small to medium-scale projects where simplicity and ease of development are prioritized over scalability and flexibility. For instance, some projects have found that the complexity of managing a system with multiple components outweighs the benefits and have migrated back to monolithic architectures [9].

On the other hand, the scalability of a monolithic application is usually limited as the life cycle, resource allocation and security configuration are shared for the whole application. As a result, increasing the inbound requests of the application requires the creation of new instances of the same application, making the allocation of new resources inconvenient for certain modules. Moreover, monolithic applications tend to have dependency issues that make it challenging to identify and fix bugs in scale, and adding new features can be difficult due to the tightly coupled nature of the components [10].

In addition, monolithic applications require the rebooting of the entire application whenever any change is made to a module. This process can result in considerable downtimes, making it difficult to test, develop, and maintain the project. Deploying monolithic applications is also challenging due to the conflicting resource requirements of the constituent models, which can result in sub-optimal deployment. Furthermore, as development teams scale or release frequency increases, monolithic applications can become increasingly difficult to manage and deploy, leading to longer release cycles and higher risks of errors or bugs. This can ultimately hinder the project's growth and scalability.

Additionally, monolithic applications tend to create a technology lock-in for developers, as they are bound to use the same language and frameworks used in the original application. This lock-in makes it difficult to switch to newer and better technologies that may offer better performance and features. As a result, developers may be unable to take full advantage of the latest and best tools available for developing software.

1.2 Service-Oriented Architectures

To mitigate the drawbacks of monolithic architecture, Service-Oriented Architectures (SOAs) were introduced in the late 1990s. They strive to employ services as basic building blocks in the development of applications. These services are characterized as self-contained, reusable, and transferable, promoting swift, cost-effective, secure, and dependable distributed applications [11], [12]. Each service is responsible for a range of functions, from simple requests to complicated processes, and can expose its interface through standard protocols to be invoked. SOAs became a popular approach for developing complex and distributed software systems back in the early 2000s. The use of SOA enables organizations to create more complicated software systems [13]. SOAs have several benefits, including reduced development time, cost, and complexity, and improved flexibility and scalability of the system [14]. In monolithic application development, building a large and complex system requires extensive planning, design, and development time that can result in delays, higher costs, and a rigid system that is difficult to change or update. In contrast, SOAs enable developers to focus on building smaller, reusable services that can be combined to create more complex applications. This modular approach to development can significantly reduce the time and effort required to build a system.

However, the widespread adoption of SOA was hindered by technical challenges, particularly with the use of the Simple Object Access Protocol (SOAP) as the standard for communication between services [15]. These challenges included the need for extensive customization, high development, and maintenance costs, and difficulty in integrating with existing systems. As a result, many organizations turned away from SOA in the mid-2000s, seeking alternative approaches that were less complex and more flexible [8]. These alternative approaches include Representational State Transfer (REST) and microservices architectures.

1.3 Microservice Architectures

Microservices, take the idea of SOA a step further by breaking down the application into services that can be deployed and scaled independently [16]. In microservices, services are typically smaller, more lightweight, and highly independent, whereas in SOA, services tend to be larger and more tightly coupled. This enables developers to build complex applications using loosely coupled services that can be developed and deployed independently, leading to greater agility, scalability, and fault tolerance [8]. This approach offers several advantages over SOAs and traditional monolithic applications, such as easier maintenance and updates, more flexibility in terms of technology selection, and better resource utilization by scaling only the necessary services. The design of microservices encourages modularity, which facilitates the isolation and testing of individual services, making maintenance and updates easier to implement. Additionally, the use of lightweight services allows for greater flexibility in terms of technology selection, as services can be developed and deployed independently with minimal dependencies. The fault-tolerant nature of microservices is a result of their distributed and decoupled design, which enables individual services to fail without impacting the overall system, ultimately leading to improved overall system reliability.

1.4 Challenges of Microservices

Microservices architecture has become increasingly popular in recent years due to its ability to simplify the development and deployment of service-oriented architectures. It enables organizations to create and manage large-scale applications by breaking them down into smaller, independent services that can be developed and deployed separately. However, microservices come with certain challenges that must be carefully considered, particularly in terms of operational challenges during runtime.

One of the primary operational challenges of microservices is that they introduce a constantly evolving infrastructure landscape of software components that are ephemeral and may change location and communicate with each other in non-intuitive ways. The combination of this dynamic environment with the DevOps approach can be difficult to manage and monitor, especially for human operators who may struggle to keep up with the pace of change. As the deployment of new software releases to production environments can occur very frequently, with thousands of deployments per day reported [17], [18], the challenge of managing and monitoring a large number of independent services can quickly become complex and require specialized tools and skills.

Figure 1.3: Topologies of microservices at Netflix. Used with permission from AWS re:Invent 2015[3].

Furthermore, to ensure the reliability, availability, and security of a microservices system can also be challenging, particularly as the number of services and instances grows. And as the number of microservices in a system increases, so does the complexity of their interactions. This complexity can result in issues such as service and/or system downtime, increased latency, and reduced system performance, all of which can have a significant impact on end users and overall system goals.

One of the popular approaches to DevOps is the use of container runtimes and orchestration platforms such as Kubernetes [19]. Containers provide a lightweight and portable environment that can be easily deployed and scaled, making it an ideal solution for managing microservices. Kubernetes as *de facto* standard provides a framework for managing containers at scale, allowing developers to deploy, manage, and scale microservices with ease. Additionally, Kubernetes provides features such as service discovery, load balancing, and rolling deployments that simplify the management and operation of microservices[4].

[3]The terms of use for this figure are available at https://aws.amazon.com/events/terms/.

[4]To promote clarity and brevity, the terms *microservice*, *container*, and *service* will henceforth be used interchangeably throughout this thesis. This choice is made due to the close relationship between these terms and their frequent use in the context of software development and deployment.

1.5 Rise of Cloud-Native

Cloud-native is a term used to describe a set of practices and technologies that enable the development and delivery of applications in modern, cloud-based environments. At its core, cloud-native is about building and running applications that leverage the capabilities of cloud computing platforms, such as elasticity, scalability, and high availability, to deliver business value more efficiently and effectively [20].

Cloud-native architecture takes the DevOps and microservices approach further, by leveraging cloud-based services and infrastructure to enable a more dynamic and agile approach to build and operate software systems. Cloud-native applications often consist of numerous microservices. As an example, Figure 1.3 shows an overview of the microservices that constitute Netflix [21]. Building and operating at such a scale could be a challenging task for human operators. Moreover, these microservices may be developed using different programming languages, belong to multiple teams within the same organization, and have thousands of constantly changing service instances. This can, upon fault and outages, lead to increased Mean Time To Repair (MTTR) due to the complex service dependencies, and the potential for services to become temporarily unavailable to their consumers. Furthermore, effective traffic management is crucial for the runtime operation of microservices, as the behavior of individual microservices can be impacted by the flow of traffic between them, ultimately affecting the performance of the entire application [22]. As a result, managing communication and traffic between microservices in such dynamic environments can be complex. To address this complexity and facilitate management, observability, and communication, service meshes were introduced. Essentially, a service mesh comprises an infrastructure layer that is integrated directly into the microservices as a collection of configurable proxies. This abstraction of the network allows for a single point of network interaction for each service [23].

1.6 Improving Resilience of Microservices

The distributed nature of microservices makes them susceptible to various issues such as service failures, network latency, and resource overutilization, which can lead to degraded application performance and reduced user satisfaction. Such failures can be transient or permanent. Failures in a system can be classified as either transient or permanent. Transient failures are temporary issues that can be resolved by retrying the operation, such as network glitches or timeouts. In contrast, permanent failures are more severe and require fixing the underlying issue. These can include bugs in the code, incompatible dependencies, or database inconsistencies. It's important to note that certain operational conditions, such as long-term overload, may also be considered permanent failures if they cannot be resolved by simply retrying the operation. However, the severity of such conditions may vary depending on the specific context and may require different types of solutions, such as autoscaling or optimizing the system architecture.

There are various studies for mitigating permanent failures in a microservice architecture such as auto-scaling horizontally or vertically [24]. These approaches have been shown to improve the availability and resilience of microservice-based applications.

Despite the recent interest in microservices both in academia and industry, few studies have investigated microservice resiliency during transient failures. There are various well-known resiliency patterns that are designed to help microservices to recover from transient failures, minimize the impact of failures, and continue to operate effectively in the face of changing conditions. The service mesh in cloud-native architecture is responsible for providing various resiliency patterns for microservices, including circuit breaking and retry mechanisms, to enhance the applications' robustness and resilience towards network failures or dependent services. Circuit breaking helps to protect the latency of incoming requests at the cost of availability, making it possible to respond more quickly to overloads and load spikes than through capacity auto-scaling. On the other hand, the retry mechanism limits the maximum number of times a service attempts to connect to another service after the initial call fails. The interval between retries prevents the called service from being overwhelmed with requests [25].

Misconfiguration of circuit breakers and retry mechanisms in a microservice architecture can lead to disastrous performance issues. If the circuit breaker is not configured correctly, it may fail to drop the incoming requests when a service becomes unavailable or unresponsive, resulting in a cascading failure that can bring down the entire application. On the other hand, if the circuit breaker is too aggressive and drops requests too frequently, it caps throughput at a level below the service capacity. And if the retry mechanism is being used, it can lead to unnecessary retries and increased latency, negatively impacting performance. This can create a vicious circle where excessive retries cause the circuit breaker to trigger, which in turn triggers more retries, further increasing latency and reducing system performance. Similarly, if the retry mechanism is not configured correctly, it can lead to increased load on the system, resulting in decreased performance and potentially overloading the system. Therefore, it is essential to properly configure these mechanisms to ensure optimal system performance and avoid potential disasters.

1.7 Research Goals and Objective

This thesis investigates the management of inter-communication among microservices in this context to enhance the performance and resilience of microservice-based applications. To accomplish this, service meshes were employed as they are an emerging technology that provides a cohesive approach for managing inter-communication and security aspects of microservice-based applications, and contain all the necessary mechanisms for traffic management. Thus, this research aims to understand the impact and effectiveness of different traffic management policies provided by service meshes on application reliability and availability in the presence of transient failures.

Furthermore, this research seeks to propose measures to enhance research reproducibility in the area of distributed systems. Reproducibility is a fundamental aspect of scientific research that ensures that the results obtained can be independently verified by others. In this thesis, we aim to provide detailed descriptions of best practices for providing research software in the distributed systems community.

These perspectives are reflected in the following research objectives:

RO1 To improve the availability and reliability of a microservice-based application.

RO2 To manage the microservice inter-communication autonomously.

RO3 To study the impact of microservice architecture and improve the reproducibility of distributed systems research.

1.8 Thesis Outline

The thesis is structured as follows: Chapter 2 provides an overview of service mesh and its potential for improving the performance of microservices. Chapter 3 presents autonomic computing as a promising approach for automating maintenance operations. Chapter 4 focuses on the importance of reproducibility in distributed systems research by presenting the recent efforts in research software and research data. Chapter 5 summarizes the contributions of the scientific works included in the thesis, presents the results of empirical studies conducted, and discusses the insights gained from these studies. Finally, Chapter 6 concludes the thesis with a discussion of the implications of the findings presented in the thesis, and proposes ideas for future research.

Chapter 2

Service Meshes

The advent of microservice technology has undoubtedly improved the efficiency and agility of software service delivery. However, this progress has come at a cost: the operational complexity of modern applications has increased manifold. To alleviate this complexity, service mesh has been introduced, which adds an infrastructure layer between microservices [26]. At a glance, a service mesh infrastructure comprises two planes - the data plane and the control plane. The data plane is made up of a set of configurable proxies, which are typically deployed as sidecars [27] (as depicted in Fig. 2.1). The sidecar proxy abstracts the microservice design from underlying network infrastructures and provides a common set of functionalities needed to connect distributed components, such as authentication and discovery. This approach aims to spare microservice developers from having to rewrite this commonly required functionality and provide a single interface through which all this functionality can be configured.

From another perspective, the control plane governs and configures the proxies for traffic routing. It also configures corresponding components to enforce policies and gather telemetry. The control plane, therefore, manages and controls the service mesh. By separating concerns between the data and control planes, the service mesh can provide improved observability, security, and reliability. Furthermore, it can enable granular control over the communication between microservices, allowing for better performance and more streamlined management of the system.

2.1 Service Mesh Features

In general, a service mesh is designed to provide a set of fundamental features that can be managed with the centralized control plane. This makes it easier for administrators to manage these features and ensures consistency across the entire microservices architecture. The application transparently benefits

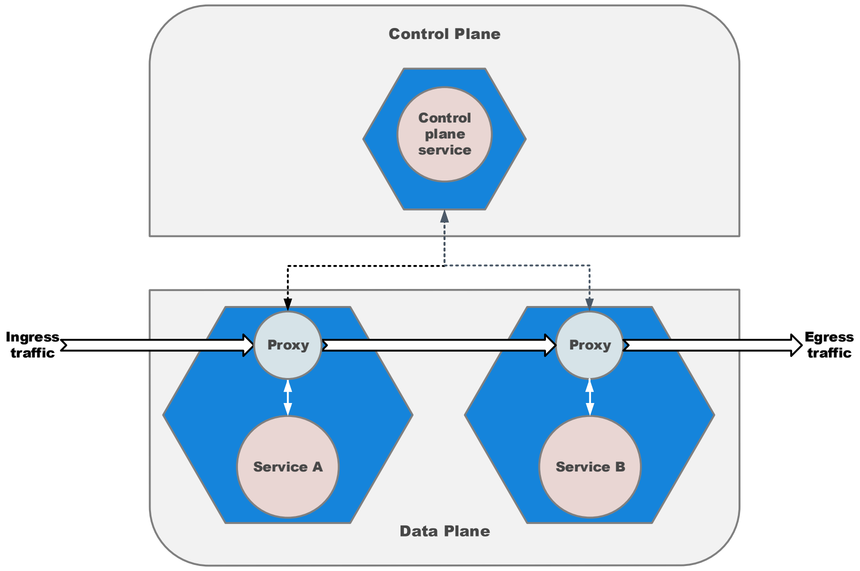


Figure 2.1: Service mesh architecture: The data plane uses proxies deployed as sidecars to control communications between microservices.

from these features, which include observability, security, extensibility, traffic management, and resiliency patterns.

2.1.1 Observability

The observability provided by a service mesh enables developers to gain visibility into their microservices and the interactions between them, which is essential for monitoring performance, troubleshooting problems, and detecting issues before they become serious [28].

Service meshes typically offer features such as distributed tracing, logging, and metrics to help understand the behavior of microservices. Distributed tracing involves tracking the flow of requests as they move through a distributed system, in order to identify and diagnose issues with latency or errors. Service mesh platforms can provide detailed tracing information, including tracing spans that link various microservices, enabling the quick identification of bottlenecks and errors.

Logging refers to the practice of capturing and storing information about events that occur within a system and it is another essential feature of observability provided by service mesh, as logs provide a detailed record of what is happening within microservices, allowing developers to troubleshoot problems quickly. Service mesh platforms typically offer centralized logging, which provides easy access to all the logs generated by microservices.

Metrics are quantitative measurements of various aspects of a system's performance, such as response time, throughput, error rates, or resource utilization. Service mesh platforms offer metrics collection, enabling developers to monitor the behavior of their microservices and quickly detect issues.

2.1.2 Security

Service mesh provides a variety of security features that help ensure the confidentiality, integrity, and availability of microservices [29]. One of the most important security features of a service mesh is mutual Transport Layer Security (mutual TLS or mTLS), which provides encryption and authentication between microservices. With mutual TLS, each microservice is required to have a valid certificate to establish a secure connection with another microservice. This prevents unauthorized access to microservices and protects in case the network is untrusted, such as when stretching the service mesh across data centers.

Service mesh also provides fine-grained access control policies to manage traffic between microservices. With access control policies, administrators can specify which microservices can communicate with each other and under what conditions. This helps prevent unauthorized access to sensitive data and ensures that only authorized microservices can access critical resources.

2.1.3 Extensibility

Service meshes are designed to be extensible and easily customizable to meet the specific needs of an organization's microservices architecture [30]. One approach to extending a service mesh is through the use of WebAssembly (Wasm). Wasm is a binary instruction format that is designed to run in a variety of environments, including web browsers and server-side environments [31]. Service mesh vendors are leveraging Wasm as a way to extend the functionality of their service meshes without requiring changes to the underlying infrastructure [32].

With Wasm, developers can create custom extensions to service meshes that can be run in a sandboxed environment within the service mesh data plane. These extensions can be written in any language that can be compiled to the Wasm format, such as C, C++, Rust, or AssemblyScript. This flexibility allows developers to use their preferred language and development tools when creating extensions for the service mesh.

Wasm extensions can be used to add new functionality to the service mesh, such as custom traffic routing, load balancing, or telemetry. They can also be used to integrate with third-party systems, such as security tools or observability platforms. Additionally, Wasm extensions can be used to implement custom policies and filters for traffic management, enabling administrators to enforce specific rules for traffic routing or modify traffic in real-time based on specific conditions.

The use of Wasm for extending service meshes offers several benefits. It provides a standardized way of extending the functionality of a service mesh that is agnostic to the underlying infrastructure. This allows developers to create extensions that can be easily deployed across different service mesh implementations. Additionally, because Wasm extensions run in a sandboxed environment, they are isolated from the service mesh control plane and other extensions, providing an added layer of security and reliability.

2.1.4 Traffic Management

Service meshes offer a variety of traffic management features that provide greater control over the flow of traffic between microservices [33]. One of the most fundamental traffic management features is load balancing, which distributes incoming traffic across multiple instances of a microservice, ensuring that the service can handle increased traffic volumes without becoming overloaded. With load balancing, administrators can specify policies for traffic distribution, such as round-robin or least connections, and can also configure the load balancing algorithm to adjust automatically based on traffic patterns.

Another traffic management feature of service meshes is traffic splitting, which allows administrators to send a percentage of traffic to different versions of a microservice. This feature is particularly useful for canary testing, where a small percentage of traffic is directed to a new version of a microservice to test its performance and reliability before rolling out the new version to all users [34].

2.1.5 Resiliency Patterns

Service meshes also provide a range of resiliency patterns that are the main focus of this thesis, to ensure the reliability and availability of microservices. One such pattern is fault injection, which allows administrators to deliberately introduce faults into the network to test the resilience of the microservices. This can help identify potential failure points and ensure that the system can recover gracefully from unexpected errors [35]. This thesis mainly discusses two resiliency patterns that are supported by service meshes: retrying and circuit breaking.

Circuit Breaker Pattern

The circuit breaker is a widely used design pattern in software development that serves to identify failures and encapsulate the logic of preventing recurring failures during maintenance, temporary external system outages, or unexpected system difficulties. This enables the system to fail faster. The Hystrix library [36] is one of the initial implementations of circuit breakers, which involves wrapping Java code in a procedure that can be regulated by the circuit breaker. Service meshes provide the same benefits but without requiring modifications to the application code. There are various implementations for this pattern in service meshes like Istio and the most important ones are:

- **Maximum connections:** In circuit breaker configuration, one important parameter is the maximum number of TCP connections that can be allowed to a specific service. This parameter is crucial because it helps prevent overloading of the service and ensures that the service is available to handle requests within its capacity. By setting a maximum number of connections, the circuit breaker can limit the traffic sent to a service and prevent it from being overwhelmed with requests. This helps maintain the overall performance of the system and prevents downtime due to service failures [37].
- **Maximum pending requests:** This parameter limits the number of requests that are allowed to be waiting and queued for a response from a downstream service. This parameter is designed to prevent the overloading of service and to ensure that resources are used efficiently. When the maximum number of pending requests is reached, the circuit breaker trips, and any new requests are rejected, returning an error to the caller. This allows the upstream service to gracefully degrade and avoids cascading failures [37].
- **Maximum requests:** Maximum requests refer to the maximum number of requests per second that are allowed to pass through the circuit breaker to the service. This is an important parameter because allowing too many requests to pass through a failing service can lead to an overload of the service, causing it to crash or slow down further. By setting a maximum number of requests, the circuit breaker can limit the amount of traffic going to the failing service, giving it time to recover or allowing requests to be redirected to other healthy instances [37].

Retry Mechanism

Retry is a common resiliency pattern in service mesh that enables automatic retries of failed requests to downstream services. With retry, if a request fails due to a temporary issue such as network latency or unavailability of a microservice, the request is automatically retried after a configurable delay period. This helps improve the availability of microservices and reduces the impact of temporary failures on end-users [38]. There are two main configuration parameters for retries:

- **Attempts:** The parameter determines how many times a sidecar proxy will attempt to establish a connection to a service if the initial call fails [39].
- **Timeout per attempt:** Specifies a timeout per retry attempt including the initial attempt [39].

Both the circuit breaker and retry mechanisms are essential for ensuring the resilience and availability of microservices. However, misconfiguring these mechanisms can result in significant performance issues. The circuit breaker is designed to protect against latency issues by dropping the requests between microservices when there are repeated failures. This protects the system from resource depletion but at the expense of availability. On the other hand, the retry mechanism is designed to improve availability by retrying failed requests, which can lead to increased latency. When these mechanisms are misconfigured in a large-scale microservice application, the system may experience disastrous performance issues. For example, if the circuit breaker threshold is set too low, it may drop requests prematurely and reduce availability unnecessarily. Similarly, if the retry mechanism is configured to have a high number of retry attempts, it may result in increased latency and reduced performance or even *retry storms.* It is, therefore, essential to tune these mechanisms to ensure desired performance and prevent issues in large-scale microservice applications.

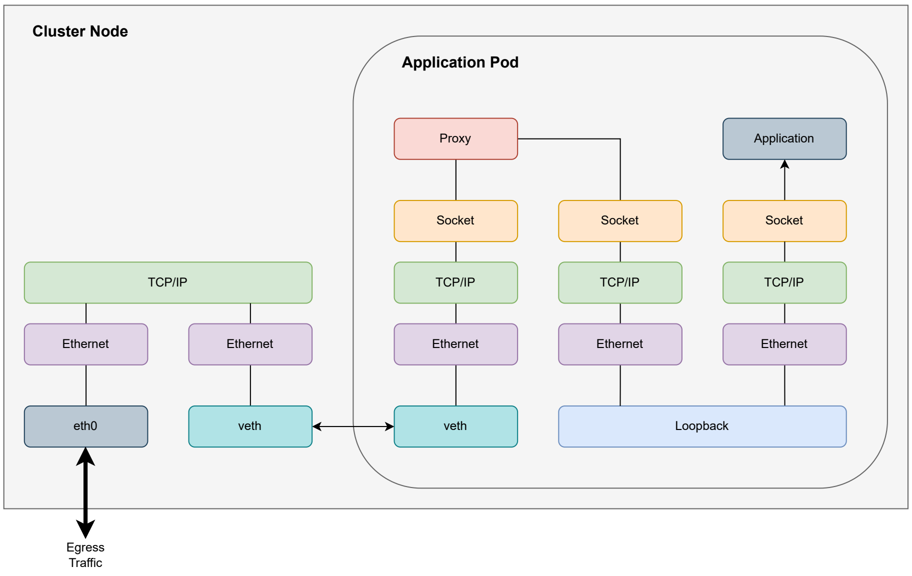
2.2 Service Mesh Architectures

Service meshes offer a solution for managing the complexity of modern applications by introducing an abstracted communication layer between microservices. One approach to building a service mesh is the sidecar proxy model, where each microservice instance has a proxy running alongside it in the same Pod. A Pod is the smallest deployable unit that consists of one or more containers with shared storage and network, and represents a logical host for those containers. The sidecar proxy in the same Pod is responsible for managing communication between microservices, providing traffic management, service discovery, and security policy implementation. While this model is flexible and can be integrated with existing applications, it has some drawbacks, particularly with respect to traffic routing.

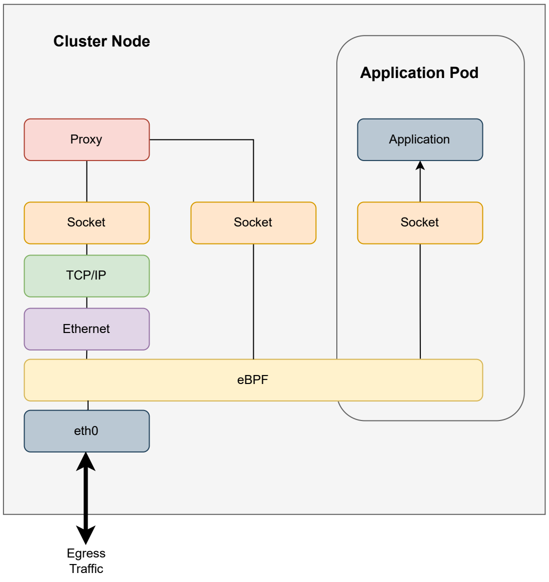
As illustrated in Figure 2.2(a), when an inbound packet arrives, it first passes through the host TCP/IP stack to reach the Pod's network namespace via a virtual Ethernet connection. This involves multiple layers of the host operating system's network stack, adding extra overhead to the process. Then, the packet goes through the Pod's network stack to reach the proxy, which forwards the packet via the loopback interface to the application. This path adds complexity to the packet's journey, leading to increased latency, especially when handling high volumes of traffic or in latency-sensitive applications [40].

Another drawback of the sidecar proxy model is that traffic has to flow through a proxy at both ends of the connection. This results in additional latency compared to non-service mesh traffic, which can adversely affect application performance. The increased complexity of the packet's path and added latency can negatively impact the performance of the application.

To improve the performance of service meshes, the community is moving towards the per-Node proxy model. A Node is a physical or virtual machine in this context and provides the underlying computing resources. In this model, a single proxy is deployed per Node instead of per microservice instance. This means that the proxy manages communication between all the microservices



(a) The route that each inbound packet should traverse to reach the application when there is a sidecar proxy per Pod.



(b) The route that each inbound packet should traverse to reach the application using the proxy per Node and eBPF.

Figure 2.2: Different service mesh architectures.

running on that Node, simplifying the deployment process and reducing the number of proxies needed. However, this model can be less flexible than the sidecar proxy model.

The extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine for packet filtering. eBPF introduces various architectural improvements in comparison with BPF to improve performance. When an event or hook is triggered, the eBPF application is executed with extremely low overhead. A more recent architecture for service meshes is the eBPF-based model, which leverages eBPF technology to provide a more efficient and flexible way of managing network traffic between microservices (see Figure 2.2(b)). The eBPF-based model enables fine-grained control over network traffic and can be easily integrated with existing applications. It is also more lightweight and scalable than other service mesh architectures. However, implementing this model requires more technical expertise and may not be suitable for all use cases [40]. Moreover, because eBPF operates at such a low level in the networking stack, it can be challenging for users to reason about how their custom code will affect the behavior of the system as a whole. Instead of thinking in terms of high-level networking abstractions like layers 2 and 3 of the OSI model, iptables rules, or routing tables, users need to think in terms of how their code will modify or replace the lower-level kernel functions that handle packet processing. This can require a different mental model and a deeper understanding of the networking stack than is typically needed for configuring network policies using more traditional tools.

2.3 Service Mesh Implementations

There are various implementations of service meshes available, each with its own unique features and functionality. Some of the most popular implementations are listed below. These implementations use different architectures and technologies to provide features.

- **Istio:** Istio is an open-source service mesh that provides a powerful and flexible platform for managing microservices. It was originally developed by some of the most famous cloud providers and is now maintained by the Cloud Native Computing Foundation (CNCF). Istio provides a range of features including traffic management, service discovery, load balancing, and security making it a popular choice for organizations with diverse technology stacks. Istio offers both sidecar proxy and proxy per Node model. In sidecar proxy, each microservice has an Envoy Proxy [41] alongside it, and in per Node model, each Node has two proxies, one for the layer 7 tasks (Waypoint proxy) and one for the layer 4 tasks (ztunnel). This allows Istio to provide fine-grained control over network traffic and implement complex policies, while also abstracting the network infrastructure from the application. Additionally, Istio integrates with a range of tools and platforms, including Kubernetes and Prometheus, making it a powerful and flexible option for managing microservices [27].

- **Linkerd:** Linkerd is an open-source service mesh that is designed to be lightweight, fast, and easy to use. It is built on top of the Rust programming language and leverages the Kubernetes platform to provide features such as service discovery, load balancing, and traffic management. Linkerd is often praised for its simplicity and ease of use, as it can be easily installed and configured without requiring any major changes to the underlying infrastructure. Additionally, Linkerd is often considered a good option for organizations with smaller service mesh deployments or those who are just starting to explore the world of service meshes [42].

- **Consul:** Consul Service Mesh is a popular implementation of a service mesh that provides features such as service discovery, traffic management, and security policies. It is built on top of the Consul service discovery and configuration tool, which allows it to provide a high level of integration with other Consul features. Consul Service Mesh uses a sidecar proxy model, with each microservice instance having a proxy deployed alongside it. The proxies communicate with the central Consul server to enable features such as service discovery and traffic routing. Consul Service Mesh also provides a dashboard for monitoring and managing the mesh, as well as integration with popular observability tools such as Prometheus and Grafana [43].

- **AWS App Mesh:** AWS App Mesh is a service mesh implementation provided by Amazon Web Services (AWS). It is designed to simplify the management of microservices in cloud-native applications by providing a way to control and monitor the communication between services. AWS App Mesh uses Envoy Proxy as the data plane for managing traffic between services. It also provides a centralized control plane for managing service discovery, routing, and security policies. AWS App Mesh is mainly compatible with other AWS services but it can also be used with other container orchestration platforms such as Docker and Kubernetes running on-premises or on other cloud platforms [44].

- **Traefik Mesh:** Traefik Mesh is another open-source service mesh built on top of Traefik, a popular cloud-native edge router, and load balancer. It provides a lightweight and easy-to-use solution for managing microservices communication within a Kubernetes cluster. Traefik Mesh offers features such as automatic service discovery, traffic management, and observability, all while minimizing the amount of configuration and maintenance required. Traefik Mesh is a promising option for organizations looking to adopt a service mesh solution without a steep learning curve or extensive infrastructure requirements [45].

- **Kuma:** Kuma is an open-source service mesh that is designed to simplify the management of service-to-service communication in modern applications. Kuma is built on top of the Envoy proxy and supports both Kubernetes and non-Kubernetes environments. It offers features such as traffic routing, service discovery, and security policies, as well as a flexible architecture that allows users to customize the mesh to their specific needs. Kuma also includes a control plane that provides a centralized interface for managing and monitoring the service mesh [46].

- **Open Service Mesh:** Open Service Mesh (OSM) is an open-source service mesh that provides a lightweight, scalable, and flexible way of managing microservices in a distributed system. Developed mainly by Microsoft, OSM is built on top of Kubernetes and leverages standard Kubernetes primitives, making it easy to integrate with existing Kubernetes-based applications. OSM offers features such as traffic management, service discovery, security policies, and observability, providing a comprehensive solution for managing microservices [47].

- **Cilium:** Cilium is a popular open-source service mesh implementation that uses eBPF technology to provide network security and observability for microservices. It offers a range of features such as HTTP and gRPC proxying, service discovery, and application-aware network security policies. Cilium also has a powerful networking stack that provides fast and efficient communication between microservices. Additionally, it can be integrated with Kubernetes and other container orchestration platforms, making it a popular choice for organizations that are already using these tools. Cilium's eBPF-based approach allows it to scale easily and handle high volumes of traffic with low latency, making it well-suited for large-scale deployments [48].

Table 2.1 summarizes different service mesh implementations. The thesis primarily relies on Istio for conducting experiments due to its widespread usage as a service mesh, active and large community, and its status as a CNCF Incubating project. Therefore, the findings and conclusions of this thesis can be extended to other service meshes that exhibit comparable resilience patterns.

Table 2.1: Summary of different implementations of service meshes.

Feature	**Istio**	**Linkerd**	**AWS App Mesh**	**Consul**	**Traefik Mesh**	**Kuma**	**Open Service Mesh**	**Cilium**
License	Apache License 2.0	Apache License 2.0	Closed Source	Mozilla License	Apache License 2.0	Apache License 2.0	Apache License 2.0	Apache License 2.0
Initiated by	Google, IBM, Lyft	Buoyant	AWS	HashiCorp	Traefik Labs	Kong	Microsoft	Cilium
Service Proxy	Envoy Proxy	Linkerd2-proxy	Envoy Proxy	Envoy Proxy exchangeable	Traefik Proxy	Envoy Proxy	Envoy Proxy	Envoy Proxy
Supported Protocols	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC	TCP HTTP/1.1+ HTTP/2 gRPC
Architecture	Sidecar proxy model Proxy per Node	Sidecar proxy model	Sidecar proxy model	Sidecar proxy model	Proxy per Node model	Sidecar proxy model	Sidecar proxy model	Proxy per Node model with eBPF
Platform	Kubernetes	Kubernetes	ECS Fargate EKS EC2	Kubernetes Nomad VMs ECS Lambda	Kubernetes	Kubernetes VMs ECS	Kubernetes	Kubernetes
Access Log Generation	yes	no	yes	yes	yes	yes	no	yes
Integration	Prometheus Grafana Jaeger Zipkin	Prometheus Grafana Jaeger	AWS X-Ray	Prometheus	Prometheus Grafana Jaeger	Prometheus Grafana Jaeger Zipkin	Prometheus Grafana Jaeger	Prometheus Grafana
Dashboard	yes	yes	yes	yes	no	yes	no	yes
Load Balancing	yes	yes	yes	yes	yes	yes	yes	yes
Traffic Splitting	yes	yes	yes	yes	yes	yes	yes	yes
Traffic routing	yes	planned	yes	yes	no	yes	yes	yes
Circuit Breaking	yes	planned	yes	yes	yes	yes	yes	yes
Retry & Timeout	yes	yes	yes	yes	yes	yes	no	yes
Fault Injection	yes	yes	no	no	no	yes	no	no
mTLS	yes	yes	yes	yes	no	yes	yes	yes

Chapter 3

Autonomic Computing

Autonomic computing is a paradigm that has been widely explored in the area of distributed systems to achieve self-management, self-configuration, self-optimization, and self-healing of complex software systems. The autonomic computing approach has become increasingly relevant due to the ever-growing scale and complexity of modern software systems, particularly those based on microservices architecture. In this chapter, we explore the relevance of autonomic computing in this context and how it can contribute to addressing the challenges related auto-tuning of resources in the cloud environment.

In 2001, IBM coined the term *autonomic computing* to describe a computer system that can adapt to changes as a new paradigm for computing [49]. Autonomic computing represents a promising paradigm that addresses the challenge of rapidly increasing software complexity [50]. This challenge poses significant challenges for both industry and academia, as the deployment, management, and maintenance of these systems become increasingly difficult for IT staff. Consequently, the cost of management also increases, and if not appropriately and timely managed, the performance of the system may decline or even result in system failure. In addition, increasing complexity redirects attention from improving the system and developing new innovative applications to handling management issues. Therefore, it seems reasonable to develop autonomic computing solutions that enable systems to self-manage and self-tune, thus reducing the burden on human operators and allowing them to focus on more value-added tasks.

Autonomic computing was inspired by the autonomic nervous system, which constantly regulates and protects our bodies subconsciously [51], allowing us to focus on other tasks. Similarly, autonomic computing aims to create systems that are aware of their environment, continuously monitor themselves, and adapt with minimal human interventions. Human administrators would only need to specify high-level policies that define the general behavior of the system. This approach would reduce the cost of management, improve performance, and enable the development of new innovative applications. Autonomic computing

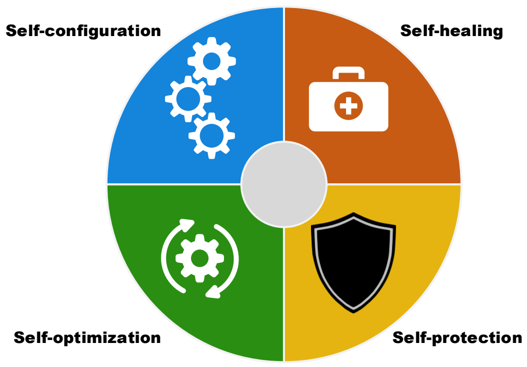


Figure 3.1: Basic properties of a self-managing system.

is not intended to replace human administrators entirely, but rather to enable systems to automatically adjust and adapt themselves to reflect evolving policies defined by humans.

Moreover, in the context of transient failures, autonomic computing approaches can be highly effective. As they enable us to detect, diagnose, and automatically repair and/or even prevent such failures. They also reduce the need for manual intervention and improve system availability and reliability. In addition, these approaches can enable systems to adapt to changing conditions and evolving requirements, helping to ensure that they remain robust and resilient over time.

3.1 Properties of Self-Managing Systems

IBM introduced a set of principles that an autonomic system should possess to be able to self-manage effectively. These properties are often referred to as *self-** properties and there are four main properties identified by IBM shown in Figure 3.1.

- ***Self-configuration*:** An essential characteristic of an autonomic system is the ability to self-configure based on the current environment and available resources. Such a system should possess the capability to continuously reconfigure itself and adapt to changes. This property is commonly referred to as self-configuration and enables the system to respond to variations in its operating environment dynamically. With self-configuration, the system can identify and allocate resources appropriately, optimize its performance, and maintain a high level of service quality.

- ***Self-optimization*:** An autonomic system should have the ability to monitor itself continuously and tune its performance automatically, ensuring that it operates at optimal levels of performance and cost. This property is known as self-optimization, and it involves collecting data from various sources, analyzing it to identify performance metrics, and adjusting system settings accordingly. Self-optimization helps to reduce the cost of management and improves the overall performance of the system. This property is particularly important in large-scale systems that involve many components, where manual optimization can be time-consuming and error-prone. By continuously tuning itself, the autonomic system can adapt to changes in the environment and avoid performance degradation, thus ensuring that it meets the requirements of the user.

- ***Self-healing*:** Self-healing is another crucial property of a self-managed system that ensures the system's ability to detect and recover from failures without human intervention. The system must be equipped with mechanisms to detect and diagnose failures and to automatically take corrective actions to restore the system to a functional state. These actions may include restarting failed components, reconfiguring the system to avoid the cause of the failure, or reallocating resources to ensure the system's continued operation. The ultimate goal of self-healing is to minimize downtime and maintain the system's availability and performance levels.

- ***Self-protection*:** Self-protection for a self-managing system involves the ability to detect, identify, and respond to security and system attacks. It is essential to ensure the security and integrity of the system, especially in the face of increasing cyber threats. A self-managed system should be able to continuously monitor itself and its environment for any signs of security or system breaches. Once detected, the system should take immediate action to isolate and mitigate the damage caused by the attack. This property requires a combination of reactive and proactive measures to ensure the system's protection. Reactive measures include mechanisms to detect and respond to known attacks, while proactive measures involve anticipating potential threats and implementing preventative measures to mitigate them. Self-protection is critical to ensure the resilience of the system and maintain its availability, integrity, and confidentiality.

3.2 The Autonomic Computing Reference Architecture

IBM has proposed a reference model for autonomic control loops, known as the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop, to attain autonomic computing [52]. This model, as depicted in Figure 3.2, is increasingly being used to communicate the architectural aspects of autonomic systems and to classify the work being done in this field.

In the MAPE-K autonomic loop, the *managed element* refers to any software or hardware resource that is granted autonomic behavior by linking it with an

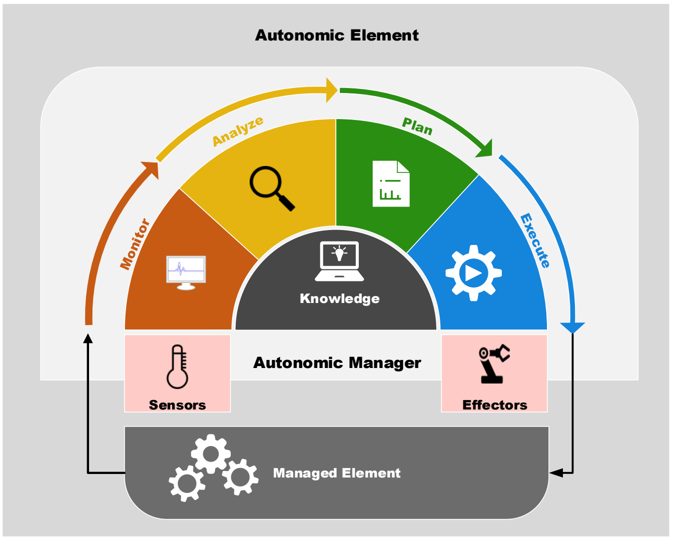


Figure 3.2: IBM's MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) reference model for autonomic control loops.

autonomic manager. This means that the *managed element* can take various forms, such as a web server or a database, a particular software component in an application like the query optimizer in a database, the operating system, a group of machines in a cloud environment, a set of hard drives, a wired or wireless network, a CPU, or a printer, etc.

Sensors, probes, or gauges gather data about the *managed element*. For instance, the information collected about a web server may include response time to client requests, CPU and memory usage, and network and disk usage.

Effectors and actuators are responsible for implementing changes to the managed element, which can range from old-fashion coarse-grained actions such as adding or removing servers from a web server cluster [53], to the gradual deployment of services to the cloud [54].

The *Autonomic Manager*: It uses data collected by *sensors* to monitor the managed element and perform changes via effectors and acutators. It is a software component that can be configured by human administrators with high-level goals. It utilizes the monitored data from sensors and the internal knowledge of the system, often an architectural model of the managed element, to plan and execute low-level actions that are necessary to achieve the goals. The goals are usually expressed through event-condition-action (ECA) policies, goal policies, or utility function policies [55]. ECA policies follow the format "when an event occurs and a condition is met, then execute an action", such as “when the response time of 95% of web servers exceeds 100 milliseconds and resources are available, increase the number of active servers”. In a MAPE-K loop, each function in the *autonomic manager* is defined as:

- The *monitor* function involves gathering, consolidating, refining, and presenting information, such as metrics and topologies, obtained from a managed resource.
- The *analyze* function includes mechanisms for correlating and modeling complex situations, such as time-series forecasting and queuing models. These mechanisms enable the autonomic manager to learn about the IT environment and make predictions about future situations.
- The *plan* function is responsible for devising the necessary actions to achieve set goals and objectives. This is achieved through the use of policy information to guide its work.
- The *execute* function encompasses the mechanisms that oversee the implementation of the plan while taking into account dynamic updates.
- The *knowledge* source serves as an implementation of a registry, key-value store, or database that provides access to management data with architected syntax and semantics, such as symptoms, policies, requests for change, and change plans. The knowledge source is accessed through interfaces prescribed by the architecture. By storing this knowledge in a knowledge source, it can be shared among different autonomic managers.

3.3 Approaches to Autonomic Computing

Various approaches have been proposed for autonomic computing by both industry and academia. Each approach has its own strengths and weaknesses, and its suitability depends on the specific requirements of the system and the environment in which it operates. In this context, this thesis provides a comprehensive overview of the three high-level approaches to autonomic computing, their main characteristics, and their applications.

3.3.1 Knowledge-Driven Approach

The knowledge-driven approach to autonomic computing is based on the idea of encoding expert and pre-existing knowledge in the form of rules or heuristics. These rules are then used to make decisions about system behavior and control. In this approach, the system is designed to understand the relationships between the different components and their behaviors and to use this understanding to make decisions about how to respond to different situations. The goal is to create a system that can make intelligent decisions without requiring human intervention.

This approach can be quite useful to manage the complexity of the distributed nature of microservices and the cloud. For instance, in a microservice-based application, the number of components and services that make up a system can be very large, and the interactions between these components can be complex. By encoding expert knowledge about these interactions in the form of rules or heuristics, the system can make decisions about how to respond to different situations without requiring human intervention.

One application of this approach for microservices is in the area of load balancing and resource allocation. There may be a large number of services running in parallel, each with different resource requirements and utilization patterns. By encoding knowledge about these requirements and patterns in the form of rules, the system can make decisions about how to allocate resources to different services in order to optimize overall system performance. This can be particularly important in cloud environments where resources are typically shared across multiple tenants.

Another application of this approach in this context could be in the area of fault detection and recovery. In such a dynamic and distributed environment, failures can occur at any system level, from individual services to entire applications. By encoding knowledge about these failure modes in the form of rules, the system can detect when failures occur and take appropriate actions to recover from them. For example, the system might automatically restart a failed service or migrate it to another Node in the cluster.

As an example of this approach, a recent study introduces a framework that enables the development of scalable IoT applications that comply with a specific set of rules defined by both the users and the deployment environment [56].

An example of this approach is demonstrated in **Paper III**, which proposes an auto-tuning mechanism for configuring a circuit breaker in a single-service application based on existing domain knowledge.

3.3.2 Data-Driven Approach

A data-driven approach to autonomic computing utilizes data analytics techniques such as machine learning to make decisions and take actions based on data collected from the system. This approach is characterized by the use of large amounts of data, often in real time, to improve the performance, reliability, and security of the system. In the context of microservices or cloud computing, data-driven autonomic computing has become an important research area due to the complexity and dynamic nature of these systems.

In large-scale and distributed systems, data-driven autonomic computing offers several advantages, including its ability to handle the complexity and variability of such environments. These systems typically consist of a large number of interconnected components that can vary in their performance and behavior over time. By collecting and analyzing data from these components, data-driven autonomic systems can make informed decisions about how to manage and optimize the system, such as scaling resources, load balancing, and fault tolerance.

Moreover, data-driven autonomic computing can adapt to changing conditions and environments. The data collected by the system can be used to detect changes in the system and adjust it accordingly. For example, if a sudden increase in traffic is detected, the system can automatically scale up resources to meet the demand, and then scale them back down when the demand subsides.

In microservices or cloud environments, data-driven autonomic computing has numerous applications. For instance, in a microservices architecture, data-driven autonomic computing can be used to optimize resource allocation and scheduling, reduce latency and response time and increase system availability and reliability. At the infrastructure level, data-driven autonomic systems can be utilized to optimize resource utilization and reduce costs, enhance security by detecting and preventing attacks, and ensure compliance with service level agreements (SLAs).

As an example of this approach, a recent study proposed a manager vertical elasticity of Docker containers [57]. This manager uses IBM's autonomic computing MAPE-K principles and scales up and down both CPU and memory assigned to each container based on the application workload. If there are not enough resources on a host, the manager migrates the target container to another host. The main motivation of this work is to improve the performance of the application.

3.3.3 Hybrid Apporach

The hybrid approach to autonomic computing combines the strengths of both knowledge-driven and data-driven approaches. The knowledge-driven approach relies on pre-defined rules and policies to make decisions, while the data-driven approach uses machine learning algorithms to derive insights from data. The combination of both approaches can lead to more effective decision-making and improved system performance.

In the context of microservices and cloud computing, the hybrid approach can be applied to enhance the scalability, reliability, and security of the system. The knowledge-driven approach can be used to define policies for resource allocation, load balancing, and fault tolerance. For example, policies can be defined to ensure that the microservices are allocated sufficient resources based on their requirements and usage patterns. Policies can also be defined to ensure that the system can handle unexpected failures and recover quickly.

On the other hand, the data-driven approach can be used to gather and analyze data from the system in real-time to detect and predict anomalies and performance issues. Machine learning algorithms can be used to learn patterns and trends in the data, and use this knowledge to make decisions and take actions. For example, machine learning algorithms can be used to predict the expected load on the system based on historical data, and automatically scale up or down the resources to ensure optimal performance.

The hybrid approach to autonomic computing can also be used to improve the security of microservices and cloud systems. The knowledge-driven approach can be used to define security policies and rules, such as access control and authorization. The data-driven approach can be used to detect and prevent security attacks and threats, by analyzing network traffic and user behavior. Machine learning algorithms can be used to learn from past attacks and incidents, and use this knowledge to identify and prevent future attacks.

There are various works employing this approach. A recent study proposed Gru, an approach for incorporating autonomic capabilities into microservices without requiring any changes to their implementation [58]. The approach involves the use of a single Gru agent on each physical or virtual host, which is responsible for making horizontal cloud decisions for the microservices running on that host.

Furthermore, **Paper IV** employs this approach by performing a sensitivity analysis to get insights about the direction of impact of different parameters for both circuit breaker and retry mechanisms on the performance of microservice applications. It then proposes an auto-tuning mechanism to control these two mechanisms for different microservice applications.

3.4 Auto-Tuning for Cloud Environments

Dynamic performance management in cloud environments has received significant attention, with several reviews of the literature focusing on autoscaling [59], scheduling [60], and the field in general [61]. Zhou et al. proposed the DAGOR overload control system for microservices, which uses the average waiting time of requests in the pending queue to profile the load status of a server [62]. Qui et al. proposed a framework that leverages machine learning to detect the root causes of Service Level Objective (SLO) violations before taking action to mitigate those causes via dynamic reprovisioning [63].

Several notable individual studies have contributed to the field of dynamic performance management in clouds. For instance, Tu et al. proposed a load-shedding technique to discard information, which resulted in reduced delay violations in a database system [64]. Babcock et al. proposed a load-shedding technique for data streaming systems [65].

A recent study provides a comprehensive survey of recent literature on resource management in cloud environments, covering over 250 publications [61]. It highlights the challenges posed by the scale, heterogeneity, and unpredictability of cloud resources, and the diverse objectives of actors within the cloud ecosystem. The authors present a conceptual framework for cloud resource management and use it to structure their state-of-the-art review. Based on their analysis, they identify five key challenges for future research, including achieving predictable performance for cloud-hosted applications, enabling global manageability of cloud systems, designing scalable resource management systems, understanding economic behavior and pricing in the cloud, and addressing the mobile cloud paradigm.

In the context of service meshes, autonomic computing approaches offer a promising solution to automate the maintenance and tuning of service meshes. Autonomic computing can leverage the observability and management capabilities of service meshes to provide automated and intelligent management of the service infrastructure, thereby ensuring optimal performance and reducing the burden on human operators [26]. By employing autonomic computing approaches, it is possible to achieve the goal of self-driving microservices, where the service mesh automatically adjusts to changing conditions in the environment and ensures the desired level of availability and reliability.

Both **Paper III** and **IV** in this thesis leverage the observability and resiliency pattern capabilities of service meshes. They propose controllers based on existing domain knowledge and insights extracted from analysis. The proposed controllers perform auto-tuning for both the circuit breaker and retry mechanisms for all services in the microservice applications.

Chapter 4

Reproducibility in Distributed Systems Research

To validate the hypothesis, this thesis extensively utilized experimentation. However, reproducing these experiments, particularly in the field of distributed systems, presents challenges. This chapter discusses the reproducibility gaps in distributed systems and how they relate to the thesis.

In the scientific process, repeatability and reproducibility are crucial factors as they ensure the accuracy and validity of reported findings, prevent flawed results from being disseminated, and foster trust in scientific research [66]. However, these terms have been defined differently by various authors [67]–[70]. **Repeatability** refers to the ability to replicate an experiment using the same procedure on a system that is either identical or comparable to the original one and obtain similar results [71]. On the other hand, **reproducibility** entails the ability to confirm a scientific hypothesis independently by a different research team [71].

It is essential to note that although these concepts are closely related, they differ in their level of rigor and the conditions required to satisfy them. In computer science, where reproducibility is more challenging to achieve due to the complexity of systems, algorithms, and software, it is necessary to distinguish between these concepts explicitly. Nonetheless, studies have shown that many research findings cannot be reproduced, indicating a reproducibility crisis [72], [73]. Furthermore, peer-review processes alone are inadequate in ensuring the reproducibility and repeatability of research findings [66], [74], [75].

An effective strategy to improve the reproducibility and repeatability of research is to provide researchers with incentives to publish their findings with evidence of reproducibility [76]. To this end, several conferences and journals have implemented a systematic process for artifact assessment and badging, known as the *artifact evaluation track*, to emphasize the importance of reproducibility in experimental research [77]. This process was introduced in 2011 and is now widely adopted by conference organizers [77], [78]. The process requires that all research artifacts pass a rigorous audit [76]. In this context, an *artifact* is defined as a digital item developed by the authors of a publication that was either used in the authors' study or generated during their experiments. Based on our study in **Paper VI**, this approach has gained popularity over the years, with the number of submitted artifacts growing from a mere 8 across the field of computer science in 2015 to 614 in 2021.

There are various reasons why researchers choose to submit their artifacts for evaluation. For some, artifacts serve as *supplementary material* that provide evidence to reviewers that the reported results were obtained in good faith, particularly in cases where data collection infrastructure is unavailable to most other researchers. For others, artifacts are seen as a means to accelerate research by enabling the reuse of old artifacts in new experiments, similar to how a software library can be utilized to speed up future software development projects. Moreover, there is anecdotal evidence that articles that include an evaluated artifact receive greater attention from the scientific community [79], [80].

The field of distributed systems research faces unique challenges in achieving reproducibility due to the complex and dynamic nature of distributed systems and the difficulty in replicating experiments across different environments. To address these challenges, this thesis aims to contribute to the improvement of reproducibility in distributed systems research by following the existing best/better practices in open science and current guidelines in artifact evaluation tracks. To this end, we begin by introducing the FAIR principles for research software. Then the most important lessons learned regarding reproducible research software are presented.

The subsequent section focuses on microservice benchmarking, highlighting the limitations of existing benchmarks in terms of their topology and limited scalability. Furthermore, the need for a benchmark generator is discussed, and our approach to achieving this is presented.

4.1 FAIR Principles

The FAIR principles were introduced in 2016 as a set of guiding principles for scientific data management and stewardship [81] The acronym FAIR stands for **Findable**, **Accessible**, **Interoperable**, and **Reusable**, and the principles are designed to help researchers make their data more discoverable and usable, as well as to promote collaboration and data reuse across disciplines and research communities. The FAIR principles have been implemented by several publicly accessible archival repositories such as Zenodo [82], FigShare [83], or Dryad [84].

The first principle of FAIR data is **findability**, which means that data should be easily discoverable by both humans and machines. This includes providing sufficient metadata and persistent identifiers to ensure that datasets can be located and accessed over time.

The second principle, **accessibility**, emphasizes the importance of making data openly available, either through public repositories or other mechanisms that ensure that datasets can be accessed by anyone who needs them.

The third principle of FAIR data is **interoperability**, which requires that data be formatted and described in a way that allows it to be easily combined and analyzed with other datasets. This includes the use of common standards and protocols for data exchange, as well as the provision of clear and consistent metadata that accurately describes the data and its provenance.

Finally, the fourth principle of FAIR data is **reusability**, which emphasizes the importance of ensuring that data can be used and repurposed for different purposes over time. This requires the use of standard and open licenses, as well as the provision of clear and complete documentation and metadata that allows others to understand the context and potential uses of the data.

Research software or artifact can be considered a type of data as it often contains information that is critical for reproducing the research results [85]. Therefore, the FAIR principles, which are designed to improve the Findability, Accessibility, Interoperability, and Reusability of data, can be applied to research artifacts and software to enhance their reproducibility and reuse. By making research software and artifacts FAIR, researchers can ensure that their work is more easily discoverable, accessible, and interoperable with other research. This, in turn, enables researchers to build upon previous work, accelerating the pace of scientific progress while increasing the transparency and rigor of research. Furthermore, applying FAIR principles to research software and artifacts can help ensure that they are preserved over time, providing a valuable resource for future researchers who wish to build upon or replicate the original research.

4.2 Reproducible Research Software

Reproducibility is a fundamental principle of scientific research, and it plays an essential role in ensuring the reliability and validity of research findings. One of the key aspects of reproducibility is the ability to reproduce experimental results using the same data and methods. In this context, research software or artifacts used to analyze data are a critical component of experimental methods. Therefore, ensuring the reproducibility of research software is essential to ensure the reproducibility of research findings. Despite the increasing focus on reproducibility in academic research, there remains a wide variation in the practices followed by researchers. This can make it difficult to evaluate and reproduce research findings.

During the course of this thesis, the author played the dual roles of artifact evaluator and artifact author. This gave him a unique perspective on the challenges faced by both parties in ensuring reproducibility. Based on his experiences, the author identified a number of best practices that were missing in the current approach to ensuring reproducibility. These insights could be valuable for researchers looking to improve their own reproducibility practices (**Paper VI**).

4.3 Microservice Benchmarking

In cloud computing research, it is crucial to evaluate the performance and efficacy of the proposed systems and methods. One common approach for evaluating cloud systems is through the use of application benchmarks. However, with the increasing popularity of microservices as an application architecture, it has become important to develop benchmarks specifically tailored for microservices. Microservices present unique challenges and characteristics, such as their distributed nature and complex dependencies, which may not be adequately captured by traditional application benchmarks. Therefore, to ensure that new cloud research artifacts are rigorously evaluated and tested, it is essential to develop microservice benchmarks that accurately reflect the characteristics and requirements of microservice-based applications. By using appropriate microservice benchmarks, researchers can more effectively evaluate the performance and scalability of their proposed methods and systems and make more informed decisions about their design and implementation [86]. They use a wide range of microservice benchmarks such as [87]–[94]. However, these benchmarks have limited complexity in terms of their architectural scale and communication topology, and hence their applicability is restricted to simpler scenarios.

For comprehensive and fine-grained performance sensitivity analysis, it is crucial to modify benchmark characteristics such as communication patterns, topological architectures, and resource usage characteristics. This approach can help identify potential bottlenecks affecting inter-service communication performance and understand how different topological architectures influence the optimal tuning of certain resource management policies. By adjusting resource usage characteristics, it is also possible to gain a deeper understanding of how proposed resource management methods perform under various conditions.

There are various benchmarking tools that were created to provide a simple and easy way for users to gain experience with cloud-native platforms. They are generally related to e-commerce use cases, such as TrainTicket [88], [95], SockShop [89], OnlineBoutique [90], and DeathStarBench-HotelReservation [91],[96]. However, many of these tools have a basic topology comprising only a few microservices and are designed as demo examples serving computationally simple applications, e.g., Bookinfo, CloudSuite [93], [94], [97], TeaStore [98], [99], JPetStore [100], PetClinic [101], AcmeAir [102], SpringCloudDemo [103], BiFrost [104]. More elaborate benchmarks, such as DeathStarBench, μSuite [92], [105], and CloudSuite, allow experimentation with various canonical architectures. Of the single and suite benchmarks, only TrainTicket and DeathStarBench enable experimental evaluation of performance impact at a larger scale. However, they are designed with a fixed architecture topology that is difficult to customize, especially if the benchmark's source code is not openly accessible.

By utilizing a configurable architecture benchmark, researchers can validate a broader range of hypotheses by systematically generating benchmarks with varying computational complexities and topologies. This allows for the generation of customized microservice-based applications and facilitates thorough and systematic experimental evaluations of performance and scalability implications, including diverse application topologies, computational complexities, and inter-service complexities for cloud-native resource management mechanisms (**Paper V**).

Chapter 5

Summary of Contributions

This chapter provides a comprehensive overview of the papers included in this thesis, highlighting their relevance to the research goal and objectives. Initially, an outline of the overall framework is presented, followed by papers in order of their relevance to the research objectives, including a detailed description of the author's contributions.

5.1 Outlines of Contributions

As outlined in Chapter 1, the research objectives of this thesis are aimed at achieving the goal of enhancing the resilience of microservice-based applications, advancing our knowledge of autonomic microservice inter-communication management, and promoting the reproducibility of distributed systems research. To this end, the following research objectives have been identified:

RO1 To improve the availability and reliability of a microservice-based application.

RO2 To manage the microservice inter-communication autonomously.

RO3 To study the impact of microservice architecture and improve the reproducibility of distributed systems research.

Figure 5.1 illustrates the specific problems addressed by each paper in this thesis. The figure provides a clear visual representation of how the various papers contribute to achieving the research objectives.

Paper I contributes to achieving **RO1** by highlighting the challenges and opportunities of using service meshes as a networking solution for microservices. The paper discusses how service meshes can help improve the availability and reliability of microservice-based applications by providing a set of configurable proxies that are responsible for the management, observability, and security of microservices. However, the paper also acknowledges that service meshes

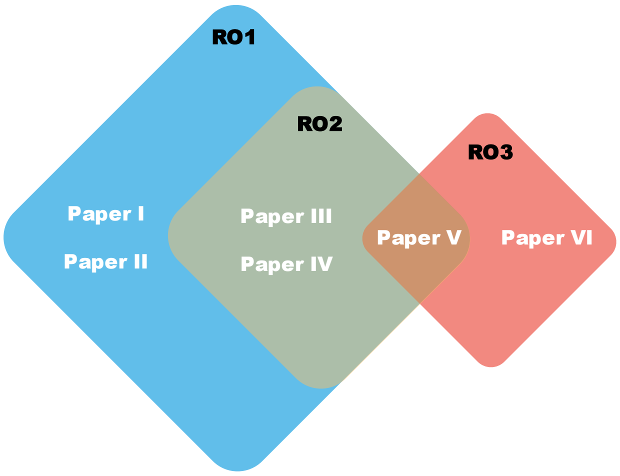


Figure 5.1: Mapping of the contents of the included paper to the research objectives.

introduce overhead into a system, which can be significant for low-powered edge devices. This highlights the need for addressing performance issues caused by service mesh proxies, which is crucial for achieving the goal of enhancing the resilience of microservice-based applications.

Paper II presents a set of experiments aimed at investigating the impact of traffic management policies such as circuit breaking and retries on the performance and robustness of a microservice application deployed in a service mesh cluster. The empirical results reveal effective configurations of circuit breakers and retries.

By identifying effective configurations of traffic management policies, **Paper II** contributes to achieving **RO1** by improving the availability and reliability of microservice-based applications. The findings presented in the paper can be used by engineers to configure such traffic management policies more systematically, allowing them to enhance the resilience of their microservices and improve the overall availability and reliability of their applications. Additionally, the paper proposes research on autonomic traffic management for microservices, providing insights into the impact of traffic management policies on microservice performance and robustness.

Paper III proposes an adaptive circuit-breaking mechanism that enhances the resilience of microservice-based applications by addressing the **RO1** of improving the availability and reliability of such applications using service meshes. The mechanism is implemented through an adaptive controller that not only avoids overload and mitigates failure but also keeps the tail response time below a given threshold while maximizing service throughput in comparison to static configuration.

Furthermore, **Paper III**'s proposal of an adaptive controller also contributes to the **RO2** of achieving autonomous management of the inter-service communication. The adaptive controller is designed to automatically adjust the circuit breaker configuration based on the observed traffic patterns, thereby reducing the need for manual intervention in configuring microservice traffic management.

Paper IV makes significant contributions to achieving the first two research objectives of this thesis. Firstly, the paper proposes a controller that dynamically adjusts the number of retry attempts and retry intervals to improve the throughput of microservice-based applications. This approach addresses **RO1** by enhancing the availability and reliability of microservices through improved throughput.

Secondly, the paper evaluates the proposed retry controller alongside the circuit breaker controller proposed in **Paper III**. Through multiple experiments involving different applications and traffic scenarios, the paper demonstrates that employing both controllers significantly improves the throughput and response times of the entire microservice application. This evaluation addresses **RO2** by promoting the autonomous management of inter-service communication, as the controllers enable the system to self-adapt and self-heal in the event of transient failures or noisy neighbors.

Paper V contributes to **RO1** by proposing HydraGen, a tool that enables researchers to systematically generate microservice benchmarks with different computational complexities and topologies. By doing so, the tool allows for more realistic and flexible evaluations of resource management mechanisms for microservice-based architectures. This could potentially lead to improved availability and reliability of microservice applications as cloud researchers can now evaluate the performance and scalability of their management mechanisms at scale with a focus on inter-service communication.

Paper V also contributes to **RO2** by demonstrating how HydraGen can enrich the evaluation of cloud management systems. By generating benchmarks with different topologies, researchers can use HydraGen to evaluate how different cloud management systems handle traffic engineering in a microservice-based architecture. This could potentially lead to the development of more autonomous cloud management systems that can dynamically adjust to changes in the system's traffic patterns.

Finally, **Paper V** is mainly contributing to **RO3** by highlighting the limitations of current microservice benchmarks, including static computational complexity, limited architectural scale, and fixed topology. The paper provides a tool that can help address these limitations and improve the reproducibility of distributed systems research. Additionally, HydraGen's open-source nature makes it more accessible to other researchers and potentially contributes to the development of a more standardized approach to microservice benchmarking.

Paper VI makes a significant contribution to **RO3** by addressing the challenge of improving the reproducibility of distributed systems research. The paper highlights the importance of repeatability and reproducibility and the growing concern over a reproducibility crisis. In particular, the paper points out that distributed systems research lags behind other computing disciplines in terms of artifact evaluation procedures and guidelines, and argues that current artifact assessment criteria are insufficient for the unique challenges of this field. To address these challenges, **Paper VI** examines the current state of the practice for artifacts and their evaluation in distributed systems research and provides recommendations for artifact authors, reviewers, and track chairs.

5.2 Paper I

M. R. Saleh Sedghpour and P. Townend. Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions. *Proceedings of the 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE, 2022)*, IEEE, pp. 176-184, 2022.

Paper Contributions

This paper provides a comprehensive survey of the use of service meshes as a networking solution for microservices and the integration of service meshes with eBPF. This paper discusses the challenges of this movement, explores its current state, and provides insights into future opportunities for microservices.

The paper introduces the concept of service meshes as an infrastructure layer built directly into microservices or the Nodes of orchestrators. Service meshes consist of configurable proxies that manage, observe, and secure microservices. However, the paper also acknowledges that the use of service meshes can introduce overhead into a system, especially for low-powered edge devices. To mitigate this issue, the industry is exploring the integration of service meshes with eBPF for faster and more efficient responses.

The paper proposes that the integration of service meshes with eBPF is the next key trend in the evolution of microservices. This integration will enable the use of service meshes as a full networking solution for most of the required features by the industry. The paper highlights the challenges of this movement, explores its current state, and discusses future opportunities in the context of microservices.

Author's Contributions

Mohammad Reza defined the problem and conducted the literature review and analysis, gathering and organizing the relevant research materials, and drafting the initial manuscript. Paul provided guidance and direction throughout the research process, and helped to edit and review the final manuscript.

5.3 Paper II

M. R. Saleh Sedghpour, C. Klein, and J. Tordsson. An Empirical Study of Service Mesh Traffic Management Policies for Microservices. *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering (ICPE, 2022)*, ACM, pp. 17-27, 2022.

Paper Contributions

This paper provides a systematic analysis of the impact of traffic management policies on microservice performance and robustness within a service mesh environment. The complexity of microservice architecture, with its numerous loosely coupled services and multiple instances, is acknowledged. To manage this complexity, service meshes offer a range of traffic management policies, such as circuit breaking and retry mechanisms, which enhance the resilience of communication between microservices. However, there is a lack of systematic studies exploring the impact of these mechanisms on microservice performance, and the exact effects of tuning parameters for circuit breaking and retries remain unclear.

To address these issues, a large number of experiments were conducted using a representative microservice benchmark in a service mesh cluster. The results reveal optimal configurations of circuit breakers and retries that are effective in protecting microservices against overload and enhancing their robustness. The findings have practical implications for engineers who aim to configure service meshes in a more systematic manner and also create new opportunities for research in the area of service meshes for (autonomic) microservice resource management. Thus the results of this paper were presented as a set of practical guidelines for engineers and service mesh users in ServiceMeshCon EU, 2022.

Author's Contributions

The problem was identified by Mohammad Reza, in collaboration with Cristian and Johan. Mohammad Reza was responsible for implementing the entire testbed setup, including experiment automation, conducting all experiments, and writing the paper. Throughout the process, Cristian and Johan provided feedback at each stage of experimentation to paper writing.

5.4 Paper III

M. R. Saleh Sedghpour, C. Klein and J. Tordsson. Service Mesh Circuit Breaker: From Panic Button to Performance Management Tool. *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems (HAOC, 2021)*, ACM, pp. 4-10, 2021.

Paper Contributions

The paper presents a solution to the limitations of circuit breaker technologies in large and dynamic distributed systems. The proposed approach involves a dynamic controller that adjusts circuit breaker queue-length thresholds based on performance metrics like response times to protect the tail response time of services. The paper extensively evaluates the proposed dynamic controller and compares it to the traditional static circuit breaker through multiple experiments in an isolated environment. The results demonstrate that the dynamic controller outperforms the static circuit breaker in terms of response time and throughput during overload conditions.

The contributions of the paper are significant in addressing the limitations of existing circuit breaker technologies and proposing a novel approach to performance management in microservices architecture. The dynamic controller can be seamlessly integrated with service mesh technologies such as Istio and Linkerd to enhance service performance in large and dynamic systems. The paper's findings can provide valuable insights to practitioners and researchers in the field of microservices architecture and performance management for distributed systems.

Author's Contributions

Mohammad Reza, Cristian, and Johan collaborated to identify and address the problem. Mohammad Reza was responsible for implementing the testbed setup and conducting experiments. All three authors worked together in writing the paper, with Cristian and Johan providing valuable feedback throughout the process, from the early experimentation stages to the final paper writing.

5.5 Paper IV

M. R. Saleh Sedghpour, D. Garlan, B. Schmerl, C. Klein, and J. Tordsson. Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry. *Submitted*, Umeå University, 2023.

Paper Contributions

The paper begins by examining the impact of tuning parameters for the retry mechanism in microservice architectures using a sensitivity analysis approach. The authors recognize the limitations of static configurations and propose an adaptive controller for the retry mechanism. The controller is designed to address the dynamic nature of microservice environments by continuously monitoring and adapting to changes in workload and resource availability. The proposed controller is evaluated under various scenarios, including transient overload and noisy neighbors, to assess its effectiveness in improving system performance. The results show that the adaptive controller can improve throughput and maintain response time even under challenging conditions. This study contributes to the development of adaptive solutions for managing microservices in dynamic environments, where traditional static configurations may be insufficient.

The results of this study suggest that the proposed adaptive controller for retry mechanisms can be easily integrated with service mesh technologies like Istio and Linkerd, thereby improving service performance in large and dynamic systems. These findings are highly relevant to practitioners and researchers working in the field of microservices architecture and performance management for distributed systems, providing them with valuable insights and potential solutions to manage the challenges of dynamic environments.

Author's Contributions

Mohammad Reza collaborated with Cristian and Johan to identify the problem. Mohammad Reza took charge of implementing the entire testbed setup, including experiment automation, conducting all experiments, and writing the paper. David, Bradley, Cristian, and Johan were actively involved throughout the process, providing feedback at every stage, from the early stages of experimentation to paper writing.

5.6 Paper V

M. R. Saleh Sedghpour, A. O. Duque, X. Cai, B. Skubic, E. Elmroth, C. Klein and J. Tordsson. HydraGen: A Microservice Benchmark Generator. *Submitted*, Umeå University, 2023.

Paper Contributions

The paper emphasizes the significance of realistic and adaptable microservice benchmarks for evaluating enhanced resource management mechanisms in large-scale software systems. However, current benchmarks have limitations in terms of their computational complexity, architectural scale, and fixed topology. Furthermore, they mainly concentrate on typical online tasks in the e-commerce domain, which fails to capture the intricacy of several real-world applications.

In response to these limitations, the paper introduces HydraGen, a tool that overcomes the shortcomings of existing microservice benchmarks by allowing researchers to generate benchmarks with varying computational complexities and topologies systematically. HydraGen's main objective is to facilitate the experimental evaluation of performance at scale for web-serving applications, particularly in terms of inter-service communication.

HydraGen is tailored to support the experimental evaluation of the performance and scalability of cloud management systems. The paper demonstrates how HydraGen can accurately reproduce an existing microservice benchmark while preserving its architectural properties. The authors also provide a case study related to traffic engineering to demonstrate how HydraGen can enhance the evaluation of cloud management systems.

Author's Contributions

Mohammad Reza and Aleksandra collaborated to formulate and solve the problem. They jointly designed and developed HydraGen, established the testbed setup, performed the experiments, and wrote the entire paper. According to their agreement, credit for the paper's first authorship is evenly divided between Mohammad Reza and Aleksandra.

Cristian and Johan provided valuable feedback from the initial experimentation phase to the paper-writing stage. Xuejun, Björn, and Erik offered guidance and direction throughout the study and assisted in reviewing the final manuscript.

5.7 Paper VI

M. R. Saleh Sedghpour, C. Klein, A. V. Papadopoulos, and J. Tordsson. Artifact Evaluation for Distributed Systems: Current Practices and Beyond. *Submitted*, Umeå University, 2023.

Paper Contributions

The paper analyzes the current status of artifact evaluation in distributed systems research and notes its inferior position compared to other computing disciplines. Furthermore, it recognizes the unique challenges posed by distributed systems research, which are currently not adequately addressed by existing artifact assessment criteria.

To address these issues, the paper offers recommendations for improving the quality and quantity of submitted artifacts. The authors hope that these recommendations will initiate a discussion among the community and enhance the overall quality of artifacts over time by providing a crucial starting point for researchers to tackle this problem.

The paper's primary contribution lies in compiling recent artifact evaluation procedures and guidelines, which serve as a valuable resource for researchers aiming to enhance the quality of their artifacts. Moreover, the authors emphasize the need for a more unified and coordinated approach to artifact evaluation in distributed systems research, which may lead to improved repeatability and reproducibility of results. The main results of this paper were presented as a set of best practices in computer science that could be integrated with European Open Science Cloud infrastructure in EOSC Symposium, 2022 [106].

Author's Contributions

Mohammad Reza, Cristian, Alessandro, and Johan worked together to formulate and solve the problem. Mohammad Reza took charge of compiling and organizing all the artifact evaluation guidelines from recent years. Meanwhile, Cristian, Alessandro, and Johan provided feedback starting from the early stages all the way to the writing of the final paper.

Chapter 6

Conclusion

In the last few years, there has been a growing trend toward microservices architecture as it offers a streamlined approach to developing and implementing service-oriented architectures. This approach allows companies to handle complex applications by dividing them into separate, self-contained services that can be developed and deployed independently. A microservice architecture may consist of hundreds or even thousands of individual services. On the other hand, the integration of the microservice approach with the DevOps methodology has led to faster development and deployment of each service. As a result, the software infrastructure landscape is constantly evolving with ephemeral software components that may change location and communicate with each other in ways that are not immediately apparent. The dynamic nature of microservices environments makes managing communication and traffic between services a complex task. To simplify this process and enhance management, observability, and communication, service meshes were introduced as a solution. A service mesh is essentially an infrastructure layer that is integrated directly into the microservices as a set of configurable proxies. This abstraction of the network enables each service to interact with a single point of the network, resulting in improved network communication and traffic management [23].

Service meshes offer various traffic management policies, including circuit breaking and retry mechanisms, which can enhance the resilience and robustness of applications against transient failures of dependent services or network issues. Circuit breaking is a traffic management policy that rejects incoming requests to protect latency at the expense of availability. A retry mechanism, on the other hand, specifies the maximum number of times a sidecar proxy will attempt to connect to a service if the initial call fails. The interval between retries prevents the server-side proxy from being overwhelmed with requests.

It's crucial to properly configure circuit breakers and retry mechanisms in a microservice architecture to avoid disastrous performance issues. Misconfigurations in these policies can lead to cascading failures that bring down the entire application, resulting in significant financial losses.

The main goal of this thesis was to utilize an autonomic computing approach to *self-configure* the tuning parameters of resiliency patterns in a microservice architecture. The focus was on adapting the configuration parameters of these patterns dynamically to enhance the throughput and latency of the microservice application. The proposed approach aimed to optimize the performance of the microservices by automatically adjusting the configuration of the circuit breaker and retry mechanisms based on real-time system feedback. By doing so, the system could adapt to changing network conditions and unpredictable failures, maintaining the overall reliability and performance of the microservice application.

In order to accomplish the main objective, the approach taken in this thesis involved several studies. The first paper focused on exploring the current state-of-the-art in-service meshes and their integration with eBPF technology. The second paper involved an empirical investigation into the impact of circuit breaking and retrying on the performance and resilience of a microservice application. In the third paper, an adaptive controller was proposed for the circuit breaker pattern to ensure a specific response time for a single service application. The fourth paper proposed a retry controller based on a sensitivity analysis of these patterns and evaluated the effectiveness of adaptive retrying and adaptive circuit breaking when used together. Throughout the first four papers, the need for a systematic microservice benchmark generation became apparent, and thus an open-source microservice benchmark generator (HydraGen) was developed in the fifth paper to address this issue and improve the reproducibility of distributed systems research. Finally, in the sixth paper, existing best practices for research software (artifact) were studied and compiled into guidelines, while considering the limitations of distributed systems, in order to further enhance the reproducibility of research in this field.

6.1 Outlook

Looking at service meshes from an industrial standpoint, it appears that they are becoming more like complete networking solutions. While there are still discussions about the architecture of service meshes, the different architectures available each have their own use cases. For example, the combination of eBPF and a proxy per node model appears to be promising for edge computing.

On the other hand, from an academic perspective, service meshes are seen as providing observability benefits without any additional burden. This has resulted in increased adoption of service meshes in research. With service meshes providing a standard way of handling microservices communication, researchers can focus on the performance and resiliency aspects of microservice applications. This, in turn, can lead to better insights and new ideas for improving microservice architectures.

With the rapid growth of microservices and distributed systems, there is an increasing demand for intelligent and adaptive solutions that can automatically optimize system performance and resilience in dynamic and complex environments. As such, autonomic computing has the potential to play a vital role in the development and management of resilient microservice applications. In the future, we can expect to see more research and development focused on autonomic computing approaches for resiliency patterns and service meshes as service meshes provide the observability themselves. This may include the development of more sophisticated and adaptive controllers, the integration of machine learning techniques, and the exploration of novel approaches for managing and optimizing the performance of distributed systems. Additionally, as the use of service meshes continues to grow, we may see more standardization and consolidation in the industry, with a move towards more unified and interoperable solutions.

Apart from the pure system contribution of this thesis, the importance of reproducibility and open science has been widely recognized in the research community. As research becomes more complex and data-intensive, the need for a rigorous and transparent evaluation of research artifacts is becoming increasingly important. The future of artifact evaluation and reproducibility is likely to involve a greater emphasis on open data, open-source software, and open standards for research artifacts. This will require researchers to adopt best practices for software engineering and data management, as well as to provide detailed documentation of their research processes and results. In addition, the development of new tools and platforms for sharing and evaluating research artifacts is likely to facilitate more collaborative and transparent research practices. Ultimately, the goal is to create a culture of open science where researchers are encouraged and supported to share their research data and methods, leading to more reproducible and impactful research outcomes.

Bibliography

[1] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic computing: principles, design and implementation.* Springer Science & Business Media, 2013.

[2] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of DevOps concepts and challenges," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–35, 2019.

[3] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: Views and beyond," in *25th International Conference on Software Engineering, 2003. Proceedings.*, IEEE, 2003, pp. 740–741.

[4] D. Garlan, "Software architecture: A travelogue," in *Future of Software Engineering Proceedings*, 2014, pp. 29–39.

[5] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in software engineering and knowledge engineering*, World Scientific, 1993, pp. 1–39.

[6] M. Richards, *Software architecture patterns.* O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA …, 2015, vol. 4.

[7] G. R. McClain, *Open systems interconnection handbook.* McGraw-Hill, Inc., 1991.

[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, Cham: Springer International Publishing, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[9] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, "The monolith strikes back: Why Istio migrated from microservices to a monolithic architecture," *IEEE Software*, vol. 38, no. 05, pp. 17–22, 2021.

[10] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014, ISSN: 1075-3583.

[11] Y. Wei and M. B. Blake, "Service-oriented computing and cloud computing: Challenges and opportunities," *IEEE Internet Computing*, vol. 14, no. 6, pp. 72–75, Nov. 2010, ISSN: 1089-7801. DOI: `10.1109/MIC.2010.147`.

[12] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Proceedings of the 7th International Conference on Properties and Applications of Dielectric Materials (Cat. No.03CH37417)*, IEEE Comput. Soc, 2003, pp. 3–12, ISBN: 0-7695-1999-7. DOI: `10.1109/WISE.2003.1254461`.

[13] W. Binder, D. Bonetta, C. Pautasso, *et al.*, "Towards self-organizing service-oriented architectures," in *2011 IEEE World Congress on Services*, IEEE, 2011, pp. 115–121.

[14] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE software*, vol. 34, no. 01, pp. 91–98, 2017.

[15] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[16] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018, ISSN: 0740-7459. DOI: `10.1109/MS.2018.2141039`.

[17] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE software*, vol. 32, no. 2, pp. 50–54, 2015.

[18] J. Jenkins, *Velocity culture*, Velocity 2011, 2011. [Online]. Available: `https://www.youtube.com/watch?v=dxk8b9rSKOo`.

[19] Kubernetes Community, *Kubernetes: Production-grade container orchestration*. [Online]. Available: `https://kubernetes.io/`.

[20] Cloud-Native Computing Foundation, *Cloud native definition*, 2023. [Online]. Available: `https://www.cncf.io/about/who-we-are/`.

[21] D. Hahn, *A day in the life of a Netflix engineer using 37% of Internet*, AWS re:Invent, 2015.

[22] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *ICPE '19*, USA: ACM, 2019, pp. 25–32.

[23] K. Ponomarev, "Attribute-based access control in service mesh," in *Dynamics '19*, Russia: IEEE, 2019, pp. 1–4.

[24] K. Rzadca, P. Findeisen, J. Swiderski, *et al.*, "Autopilot: Workload autoscaling at Google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[25] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[26] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 122–1225.

[27] Istio Community, *Simplify observability, traffic management, security, and policy with the leading service mesh.* [Online]. Available: `https://Istio.io/`.

[28] D. Cha and Y. Kim, "Service mesh based distributed tracing system," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, 2021, pp. 1464–1466.

[29] R. Chandramouli, Z. Butcher, *et al.*, "Building secure microservices-based applications using service-mesh architecture," *NIST Special Publication*, vol. 800, 204A, 2020.

[30] A. El Malki and U. Zdun, "Guiding architectural decision making on service mesh based microservice architectures," in *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*, Springer, 2019, pp. 3–19.

[31] A. Haas, A. Rossberg, D. L. Schuff, *et al.*, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[32] A. O. Duque, C. Klein, J. Feng, X. Cai, B. Skubic, and E. Elmroth, "A qualitative evaluation of service mesh-based traffic management for mobile edge cloud," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, IEEE, 2022, pp. 210–219.

[33] A. Khatri and V. Khatri, *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul.* Packt Publishing Ltd, 2020.

[34] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini, "CanaryAdvisor: A statistical-based tool for canary testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 418–422.

[35] N. Mendonca and C. Aderaldo, "Towards first-class architectural connectors: The case for self-adaptive service meshes," in *Brazilian Symposium on Software Engineering*, 2021, pp. 404–409.

[36] Netflix, *Hystrix: Latency and fault tolerance for distributed systems*, 2023. [Online]. Available: `https://github.com/Netflix/Hystrix/`.

[37] Envoy Community, *Circuit breaking / Envoy documentation*, 2023. [Online]. Available: `https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/circuit_breaking`.

[38] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, "Automated testing and resilience of microservice's network-link using Istio service mesh," in *2022 31st Conference of Open Innovations Association (FRUCT)*, 2022, pp. 79–88.

[39] Istio Community, *Istio / Virtual service - HTTPRetry.* [Online]. Available: `https://Istio.io/latest/docs/reference/config/networking/virtual-service/#HTTPRetry`.

[40] Cilium Community, *Try eBPF-powered cilium service mesh.* [Online]. Available: `https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta/`.

[41] Envoy Proxy Community, *Envoy Proxy.* [Online]. Available: `https://www.envoyproxy.io/`.

[42] Linkerd Community, *The world's lightest, fastest service mesh.* [Online]. Available: `https://linkerd.io/`.

[43] Consul Community, *Service mesh on Consul.* [Online]. Available: `https://developer.hashicorp.com/consul/docs/connect`.

[44] Amazon Web Services Inc., *AWS App Mesh.* [Online]. Available: `https://aws.amazon.com/app-mesh/`.

[45] Traefik Labs, *Traefik Mesh - The simplest service mesh.* [Online]. Available: `https://traefik.io/traefik-mesh/`.

[46] Kuma authors, *Kuma - the universal Envoy service mesh for distributed service connectivity.* [Online]. Available: `https://kuma.io/`.

[47] Open Service Mesh Authors, *Open Service Mesh.* [Online]. Available: `https://openservicemesh.io/`.

[48] Isovalent, *Cilium service mesh beta.* [Online]. Available: `https://github.com/cilium/cilium-service-mesh-beta`.

[49] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," 2001.

[50] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[51] P. Van Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, and T. Coupaye, "Self management for large-scale distributed systems: An overview of the SELFMAN project," in *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures 6*, Springer, 2008, pp. 153–178.

[52] IBM, "An architectural blueprint for autonomic computing," Tech. Rep., 2006, pp. 1–37.

[53] B. Schmerl and D. Garlan, "Exploiting architectural design knowledge to support self-repairing systems," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, 2002, pp. 241–248.

[54] J. L. da Silva, M. M. Assis, A. Braga, and R. Moraes, "Deploying privacy as a service within a cloud-based framework," in *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, IEEE, 2019, pp. 1–4.

[55] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.*, IEEE, 2004, pp. 3–12.

[56] E. Goynugur, G. de Mel, M. Sensoy, K. Talamadupula, and S. Calo, "A knowledge driven policy framework for Internet of Things," in *Proceedings of the 9th International Conference on Agents and Artificial Intelligence*, vol. 2, 2017, pp. 207–216.

[57] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of Docker containers with ELASTICDOCKER," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 472–479. DOI: 10.1109/CLOUD.2017.67.

[58] L. Florio and E. D. Nitto, "Gru: An approach to introduce decentralized autonomic behavior in microservices architectures," in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, 2016, pp. 357–362. DOI: 10.1109/ICAC.2016.25.

[59] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, Oct. 2014. DOI: 10.1007/s10723-014-9314-7.

[60] A. Arunarani, D. Manjula, and V. Sugumaran, "Task scheduling techniques in cloud computing: A literature survey," *Future Generation Computer Systems*, vol. 91, pp. 407–415, 2019, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.09.014.

[61] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.

[62] H. Zhou, M. Chen, Q. Lin, *et al.*, "Overload control for scaling WeChat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 149–161, ISBN: 9781450360111. DOI: 10.1145/3267809.3267823.

[63] H. Qiu, S. Banerjee, S. Jha, Z. Kalbarczyk, and R. Iyer, "Firm: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*, USENIX Association, 2020, pp. 805–825.

[64] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao, “Load shedding in stream databases: A control-based approach,” in *VLDB ’06*, Seoul, Korea: VLDB Endowment, 2006, pp. 787–798.

[65] B. Babcock, M. Datar, and R. Motwani, “Load shedding for aggregation queries over data streams,” in *Proceedings. 20th International Conference on Data Engineering*, IEEE, 2004, pp. 350–361.

[66] C. Collberg and T. A. Proebsting, “Repeatability in computer systems research,” *Commun. ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016, ISSN: 0001-0782. DOI: `10.1145/2812803`.

[67] A. Abedi, A. Heard, and T. Brecht, “Conducting repeatable experiments and fair comparisons using 802.11N MIMO networks,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 41–50, 2015.

[68] C. Drummond, “Replicability is not reproducibility: Nor is it good science,” in *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, National Research Council of Canada Montreal, Canada, vol. 1, USA: ACM, 2009.

[69] D. G. Feitelson, “From repeatability to reproducibility and corroboration,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 3–11, 2015.

[70] B. R. Jasny, G. Chin, L. Chong, and S. Vignieri, *Again, and again, and again. . .* 2011.

[71] J. Vitek and T. Kalibera, “Repeatability, reproducibility, and rigor in systems research,” in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT ’11, Taipei, Taiwan: Association for Computing Machinery, 2011, pp. 33–38, ISBN: 9781450307147. DOI: `10.1145/2038642.2038650`.

[72] M. McNutt, “Reproducibility,” *Science*, vol. 343, no. 6168, pp. 229–229, 2014. DOI: `10.1126/science.1250475`.

[73] M. Baker, “1,500 scientists lift the lid on reproducibility,” *Nature*, vol. 533, no. 7604, pp. 452–454, May 2016, ISSN: 1476-4687. DOI: `10.1038/533452a`.

[74] S. Krishnamurthi and J. Vitek, “The real software crisis: Repeatability as a core value,” *Commun. ACM*, vol. 58, no. 3, pp. 34–36, Feb. 2015, ISSN: 0001-0782. DOI: `10.1145/2658987`.

[75] D. Delling, C. Demetrescu, D. S. Johnson, and J. Vitek, “Rethinking experimental methods in computing (dagstuhl seminar 16111),” *Dagstuhl Reports*, vol. 6, no. 3, D. Delling, C. Demetrescu, D. S. Johnson, and J. Vitek, Eds., pp. 24–43, 2016, ISSN: 2192-5283. DOI: `10.4230/DagRep.6.3.24`.

[76] ACM, *Artifact review and badging version 1.1*, Online, Aug. 2020. [Online]. Available: `https://www.acm.org/publications/policies/artifact-review-and-badging-current`.

[77] S. Krishnamurthi, “Artifact evaluation for software conferences,” *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 3, pp. 7–10, 2013.

[78] B. R. Childers, G. Fursin, S. Krishnamurthi, and A. Zeller, “Artifact evaluation for publications (dagstuhl perspectives workshop 15452),” in *Dagstuhl Reports*, vol. 5, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 29–35.

[79] B. R. Childers and P. K. Chrysanthis, “Artifact evaluation: Is it a real incentive?” In *2017 IEEE 13th international conference on e-science (e-Science)*, IEEE, USA: IEEE, 2017, pp. 488–489.

[80] R. Heumüller, S. Nielebock, J. Krüger, and F. Ortmeier, “Publish or perish, but do not forget your software artifacts,” *Empirical Software Engineering*, vol. 25, no. 6, pp. 4585–4616, 2020.

[81] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, *et al.*, “The FAIR guiding principles for scientific data management and stewardship,” *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.

[82] Zenodo, *Zenodo - research. shared.* 2022. [Online]. Available: `https://zenodo.org/`.

[83] Figshare, *Figshare - credit for all your research*, 2022. [Online]. Available: `https://figshare.com/`.

[84] Dryad, *Dryad - publish and preserve your data*, 2022. [Online]. Available: `https://datadryad.org/`.

[85] A.-L. Lamprecht, L. Garcia, M. Kuzak, *et al.*, “Towards FAIR principles for research software,” *Data Science*, vol. 3, no. 1, pp. 37–59, 2020.

[86] A. Detti, L. Funari, and L. Petrucci, “μBench: An open-source factory of benchmark microservice applications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 968–980, 2023. DOI: `10.1109/TPDS.2023.3236447`.

[87] Istio Community, *Istio / Bookinfo application*, 2022. [Online]. Available: `https://Istio.io/latest/docs/examples/bookinfo/`.

[88] X. Zhou *et al.*, “Benchmarking microservice systems for software engineering research,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, USA: ACM, 2018, pp. 323–324. DOI: `10.1145/3183440.3194991`.

[89] Weaveworks, *Sock Shop: A microservices demo application*, 2022. [Online]. Available: `https://github.com/microservices-demo/microservices-demo`.

[90] Google Cloud Platform, *Online Boutique: A cloud-first microservices demo application*, Last checked: 2023-02-26, 2023. [Online]. Available: `https://github.com/GoogleCloudPlatform/microservices-demo`.

[91] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS '19*, Providence, RI, USA: ACM, 2019, pp. 3–18, ISBN: 9781450362405.

[92] A. Sriraman and T. F. Wenisch, "μSuite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, USA: IEEE, 2018, pp. 1–12. DOI: `10.1109/IISWC.2018.8573515`.

[93] M. Ferdman, A. Adileh, O. Kocberber, *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, Mar. 2012, ISSN: 0362-1340. DOI: `10.1145/2248487.2150982`.

[94] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, USA: IEEE, 2016, pp. 122–132. DOI: `10.1109/ISPASS.2016.7482080`.

[95] X. Zhou *et al.*, *Train Ticket: A benchmark microservice system*, 2022. [Online]. Available: `https://github.com/FudanSELab/train-ticket`.

[96] Y. Gan *et al.*, *Deathstarbench: Open-source benchmark suite for cloud microservices*, 2022. [Online]. Available: `https://github.com/delimitrou/DeathStarBench`.

[97] M. Ferdman *et al.*, *Cloudsuite: A benchmark suite for cloud services*, Last checked: 2022-10-10, 2022. [Online]. Available: `https://github.com/parsa-epfl/cloudsuite`.

[98] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018, pp. 223–236. DOI: `10.1109/MASCOTS.2018.00030`.

[99] U. of Würzburg, *Teastore: A micro-service reference test application*, Last checked: 2023-02-27, 2023. [Online]. Available: `https://github.com/DescartesResearch/TeaStore`.

[100] R. Jung and M. Adolf, "The JPetStore suite: A concise experiment setup for research," *Softwaretechnik-Trends*, vol. 39, no. 3, pp. 40–42, Nov. 2019, (Proceedings of the 9th Symposium on Software Performance (SSP 2019)), ISSN: 0720-8928.

[101] Spring, *Spring PetClinic: Distributed version of Spring Petclinic built with Spring Cloud*, Last checked: 2023-02-27, 2023. [Online]. Available: `https://github.com/spring-petclinic/spring-petclinic-microservices`.

[102] *Acme Air sample and benchmark*, Last checked: 2023-02-27, 2023. [Online]. Available: `https://github.com/acmeair/acmeair`.

[103] *Spring Cloud example project*, Last checked: 2023-02-27, 2023. [Online]. Available: `https://github.com/kbastani/spring-cloud-microservice-example`.

[104] *Bifrost microservices sample application*, Last checked: 2023-02-27, 2023. [Online]. Available: `https://github.com/sealuzh/bifrost-microservices-sample-application`.

[105] A. Sriraman and T. F. Wenisch, *μSuite: A benchmark suite for microservices*, Last checked: 2022-10-10, 2022. [Online]. Available: `https://github.com/wenischlab/MicroSuite`.

[106] M. R. Saleh Sedghpour and S. I. Ulfsparre, *Integration of research software into the eosc infrastructure: Lessons learned from computer science*, EOSC Symposium 2022, 2022.