Received 31 July 2021; | Revised day month year; | Accepted day month year

DOI: xxx/xxxx

SPECIAL ISSUE PAPER - PERSPECTIVE

Systematic Literature Review on Microservices: Challenges and Technologies

Keerthana Sanala Prakash[1] | Henrique Rocha*[2] | Marco Túlio Valente[3] | Serge Demeyer[1] | Anthony Cleve[4]

[1]Department of Computer Science, University of Antwerp, Belgium
[2]Department of Computer Science, Loyola University Maryland, USA
[3]Deparment of Computer Science, UFMG, Brazil
[4]Namur Digital Institute, University of Namur, Belgium

Correspondence
*Henrique Rocha. Email: henrique.rocha@gmail.com

Present Address
Present address

Abstract

As microservice architecture is a new research area, the need for a systematic literature review is important to summarise the state of the art and identify the potential for future studies. Although various studies have been conducted which cover the microservice challenges, it is still strenuous to get a clear image of all the existing challenges one needs to understand while adopting microservices. In this paper, we conducted a systematic literature review to understand the challenges in adopting microservices and the technologies used in implementing microservices. In our systematic literature review, we collected 81 primary studies. We identified that the most common challenges in adopting microservices are migration (14 references), performance (11 references), scalability (10 references), and testing (10 references). For the technologies used in implementing microservices, container technologies (i.e., docker, kubernetes) are the most cited (14 references). Moreover, the second most cited are the programming languages and Other technologies used to develop microservices (13 references). The resulting study can serve as a foundation for researchers and practitioners to plan and implement microservices.

KEYWORDS:
microservices, systematic literature review, service oriented architecture, internet of things.

1 | INTRODUCTION

During the last ten years, cloud computing, and the relatively low cost of server renting services like amazon web services, azure, and google cloud have opened the opportunity to build businesses around cloud technologies. This model also brings new challenges to scale and maintain systems[1]. Inspired by service-oriented computing, microservices are small applications with a single responsibility that can be deployed, scaled, and tested independently. The concept was created as service-oriented architecture (SOA) 10 years back. It is about fragmenting complex applications into small pieces and a fluid delivery model that delivers services on demand, thus improving performance[2].

The design, development, and operation of microservices are picking up more momentum in the industry. At the same time, academic work on the topic is at an early stage[3,4,5]. Companies are working day-by-day on the practical implementation of microservices[6,7]. For instance, the microservices architecture allowed netflix[8] to greatly speed up the development and deployment of its platform and services.

The academia is joining microservices architectural patterns to other disciplines. For example, devOps and internet of things (IoT)[9]. However, there is still no clear perspective of emerging recurrent solutions or design decisions in microservices both in industry and academia[3]. Despite the hype for microservices, both industry and academia still lack consensus on the adequate conditions to embrace and benefit from this new paradigm[4]. It also brings new challenges in scaling and maintaining the system as fast as we are moving towards using a microservices architecture.

While many organizations like netflix[8], uber[10], and amazon[11] have proposed solutions to certain challenges, they focus only on their organizational perspective. It is also suggested that every challenge is tailored for each company and the solutions proposed by one may not be well-suited to others[6]. Many aspects of the practical challenges in microservices are still unexplored. This makes it difficult for researchers or practitioners to know where to start the adoption process. The goal of this paper is to characterize the possible overall challenges faced when adopting microservices and technologies involved in microservice implementation. To achieve this goal, we conducted a systematic literature review[12]. More specifically, we designed two research questions, the first one related to the challenges and the second to technology solutions in implementing microservice, to guide our literature review.

The main contribution of the paper includes all the recent challenges after the introduction of microservice and a list of technologies that have been used in leveraging the implementation of microservice. We also discuss proposed solutions we found in the literature to address the challenges.

The rest of the paper is structured as follows. In Section 2, we describe background information on microservice and monolith; we also explain the importance to migrate from monolith to microservices. In Section 3, we describe the method and protocol followed in our systematic literature review. In Section 4, we answer the research questions and cover the challenges and technologies used in microservice. In Section 5, we discuss possible solutions for challenges mentioned in the studied literature. In Section 6, we cover the threats to the validity of our study. In Section 7, we present the related work. Finally, in Section 8, we conclude the paper and outline future work ideas.

2 | BACKGROUND

Microservices are an architectural style that structures an application as a collection of independent modules. These modules are highly maintainable, testable, loosely coupled, independently deployable, and each isolates a functionality. Therefore, we can split the application into distinct independent services[13]. Recently, microservice architecture has evolved as a paradigm shift in decomposing large monolithic applications into smaller manageable services with their code base and deployment infrastructure[14]. In microservices, every application function is its service, own container, and communicate via Application programming interface (API)[15].

We can describe microservices as simple and stateless. In a microservices world, business requirements are divided into independent features and each of them is built as an individual service when deployed in separate containers. Therefore, it is easy to build and deploy, especially when it is containerized[16,17]. Microservices are loosely coupled because they can be developed and released independently of each other. Since services interact over technology-agnostic protocols like HTTP, services could use any technology[18]. Microservices bring many benefits to applications. Although, service level independent scalability has been considered as the most value-adding advantage for choosing microservices architecture for applications.

For several years, a monolithic architecture was the widely-used architecture for building web and mobile applications. The server-side system is based on a single application and easy to develop, deploy, and manage[15]. These applications are mostly characterized by individual programs handling multiple functionalities[19]. They often have one database that is shared by all departments. This has the advantage that if we want to make a change in all services, it is enough to implement this change in one place[16]. It is this simplicity at the start that seems to be the greatest advantage of monoliths. We do not have to think about the division of services and functions. Though monolithic applications are known to be easier to operate, as the systems grew bigger, they also increased the complexity for coding, deployment, and maintenance stages of the software development life cycle[20,21]. Single point of failure, technology lock-in, and limited scalability are a few other drawbacks of monolithic applications. Growing companies are considering other architecture styles such as microservices[22,23,24]. The most common reasons teams migrate to microservices are resilenced and continuous delivery, amongst other challenges[25].

Adopting microservices comes with a cost[26] at the information technology (IT) operational level because of the expenses incurred for deployment, load balancing, process monitoring, and scalability for each service. If not adopted well, one might end up having technical debt. On the other hand, container platforms like docker and kubernetes provide these functions as part of their features[27,28]. These platforms help in scalability, making it the compelling architecture choice for applications.

Microservices use http as the protocol for synchronous communication and multiple service endpoints may give malicious attackers more opportunities for system penetration. Hence, it is important to secure microservices-based applications with proper authentication and authorization methods[18,4].

3 | STUDY DESIGN

We conducted a systematic literature review (SLR) conforming to the guidelines presented by Kitchenham and Charters[12]. A systematic literature review (SLR) is a method of reviewing data and results from research about a particular question in a standardized systematic way[29]. According to Garousi et al.[30], it is important to complement scientific papers in an systematic literature review by also analyzing grey literature. Therefore, we decided to adapt our systematic literature review protocol to include grey references.

In this section, we describe the designed protocol for our systematic literature review which was enhanced with grey literature. We divided our protocol for the systematic literature review into the following steps: (A) Preparation, (B) Research questions, (C) Search procedure, (D) Selection criteria, (E) Grey literature, (F) Data extraction, and (G) Data synthesis.

3.1 | Preparation

This step is called background by Kitchenham and Charters[12]. After we defined the protocol, we started by reading papers about microservices to understand the current scope of microservices research.

The first papers we study were a systematic literature review[31] and systematic mapping studies[1,19] since both types provide an overview of the state of the art on microservices. Then, we performed a backwards snowballing procedure to gather more references for this initial knowledge on microservices research[32,33,34,4,6,2,9,8,35,3,5].

Based on the studied references, we developed the rationale for this study (Sections I and II). Moreover, with an understanding of the research state-of-the-art, we were prepared to elaborate the research questions we aimed to investigate.

3.2 | Research Questions

The leading goal of this research is to analyze the possible challenges in adopting microservices. Moreover, we have also deemed it important to cover the technologies used in implementing microservices and solving such challenges. Therefore, we elaborated the following research questions to guide our systematic literature review:

- RQ#1: What are the main challenges when adopting microservices?
- RQ#2: What are the main technologies used for implementing microservices?

3.3 | Search Procedure

This step describes the search procedure we followed to acquire scientific references. We planned our procedure to search for microservices papers that would discuss challenges and technologies (i.e., tackle our research questions) but without biasing the search results. We performed a manual search using google scholar as the search engine. We chose google scholar because it combines results from multiple digital libraries. Moreover, citations and patents were excluded from the search.

We developed a strategy to choose the appropriate search string for our research. Two researchers (the main author and the supervisor) would brainstorm and propose candidate strings. For each candidate string, we analyzed if the top results seem relevant for our research. We also needed a string that would not give too many results for the authors to handle manually. Therefore, the total number of results was also a factor in choosing the string. Moreover, we did not want the search string to be biased. After one week of analyzing candidates strings, we chose the following in our search:

- `microservices "architecture style" design`

The above string produces 471 references at the time we collected its results (2020-11-12). Figure 1 shows the search results divided per year.

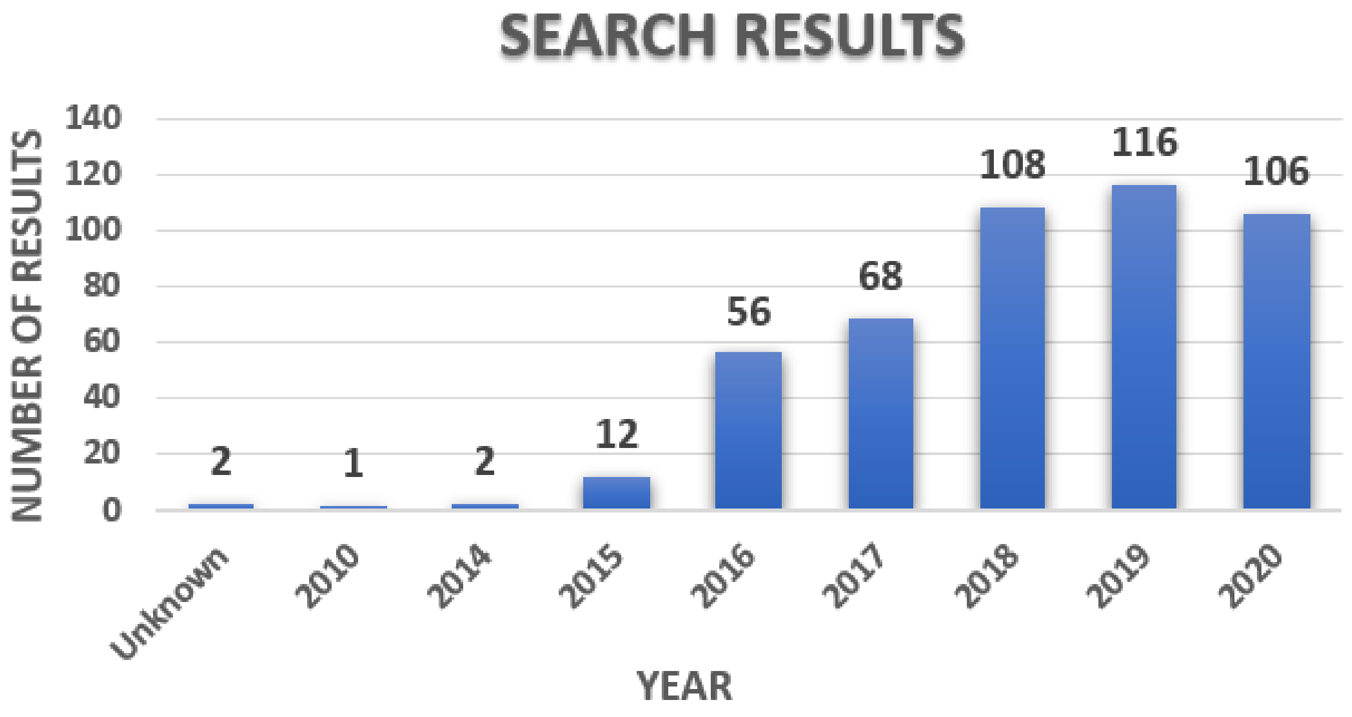


FIGURE 1 Search results per year before applying the selection criteria.

3.4 | Selection Criteria

Selection criteria determine which studies to include or exclude from a systematic literature review. Since the search provided a good amount of results, we focused on defining rejection criteria to exclude irrelevant results. Table 1 summarizes our selection criteria which we detail as follows:

TABLE 1 Selection Criteria

Exclusion Criteria	R1: Duplicate papers R2: Not written in English R3: Not peer-reviewed R4: Published before 2017 R5: Select Extended versions R6: Unrelated based on title R7: Unrelated based on abstract

- **R1: Duplicated papers.** The search engine we used acquires papers from multiple libraries. Therefore, the results can contain the same paper more than once but from different sources. Then, we reject all duplicated instances of a paper that is already in our study.
- **R2: Not written in English.** A paper can have an English abstract and keywords even though the paper content is written in another language. The search engine may still present such papers based on their abstracts.

- **R3: Not peer-reviewed.** Some digital libraries allow users to post papers that were not peer-reviewed. Since the selection criteria are to choose scientific papers, peer-review is very important to maintain academic integrity. We rejected all papers that were not peer-reviewed.
- **R4: Publication from 2017 to 2020.** We exclude papers published before 2017. We wanted this study to focus on the most recent studies on microservices.
- **R5: Multiple versions.** When a paper has multiple versions of itself, we reject the previous versions and keep only the most current one. For example, if a paper published on a conference is later extended to a journal paper, we would reject the conference version and keep the journal as it is the most current.
- **R6: Unrelated based on title.** For this criterion, we read the title of the paper to assess if it was indeed a relevant study to answer our research questions. The titles deemed unrelated were excluded.
- **R7: Unrelated based on abstract.** For this criterion, we read the abstract of the paper to verify if it would help to answer our research questions. The abstracts that were not related to our research were rejected.

We apply the selection criteria in the same order as shown, i.e., we apply R1 first then followed by R2, and so forth. From the described selection criteria, only the last two (R6 and R7) are subjective. To mitigate the subjectiveness of such criteria, we had two persons (the main author and the supervisor) working separately to review them. In case of disagreement, the paper would be included in study. Therefore, we only rejected papers if both persons marked it as unrelated.

As we previously described, the search returned a total of 471 results. After applying our selection criteria, we obtained 60 studies (i.e., 311 papers were rejected). Figure 2 shows the number of papers rejected by each criterion, and 'A' marks the total of accepted papers after the rejection criteria were applied.

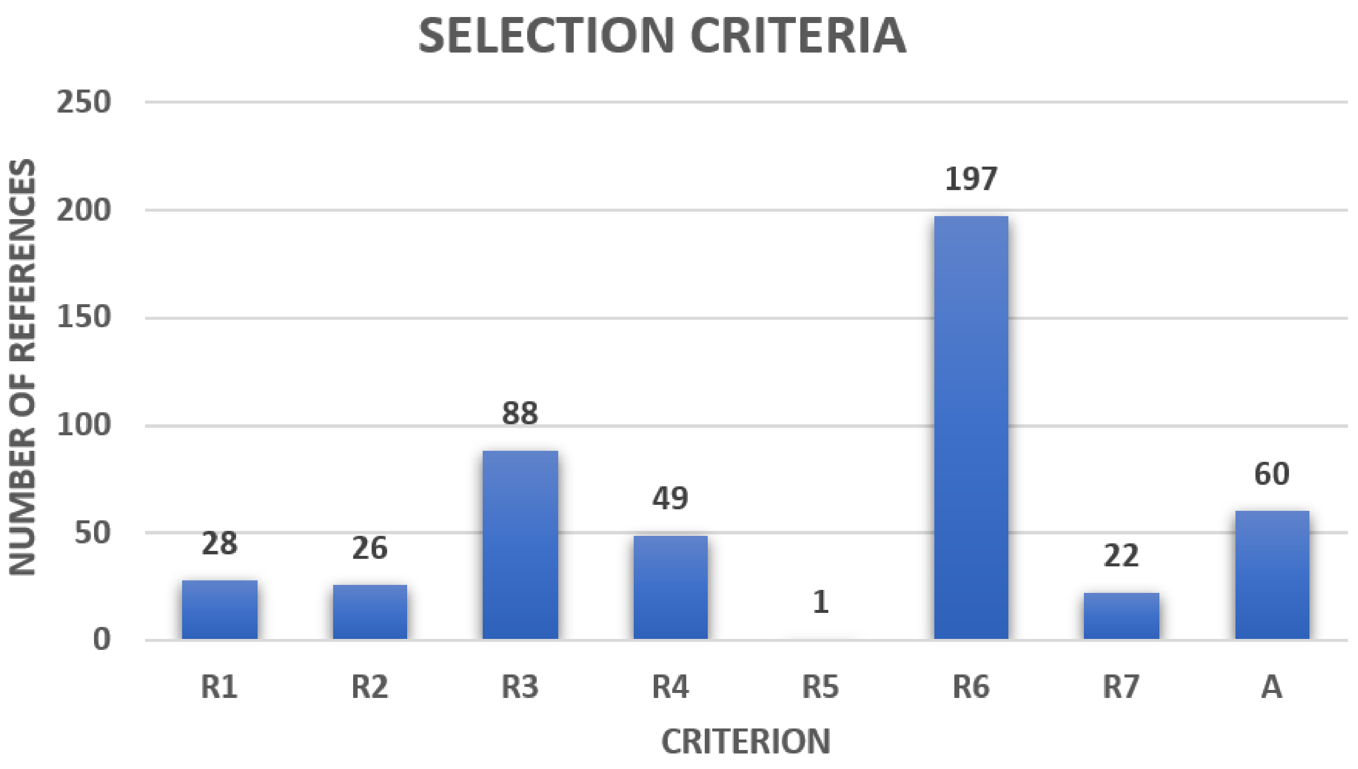


FIGURE 2 Selection criteria showing the number of scientific references rejected by each exclusion criterion (R1 to R7) and the number of accepted (A) references.

3.5 | Grey Literature

We decided to complement the scientific references that passed our selection criteria with grey literature[30]. Since microservices is a popular topic amongst practitioners, we believe it is important to include their knowledge in our research. The use of grey literature can serve as evidence of practitioners' knowledge and experience[36].

Intended as materials and research studies produced outside at the organization of traditional commercial and academic is important. grey literature publications include news, blogs, videos, etc. However, grey literature could be of a risk that there might be no technical evidence or evaluation. Although for the scope of this study, we tried to include legitimate speakers and bloggers from companies.

For this study, we started by collecting videos from the goto conference[1] which is an event for practitioners. Then we searched for microservices content by respectable sources.

Guidelines for grey literature review[37] state it is important to define stop criteria when gathering grey literature sources. One of the reasons for stopping rules is the large volume of data one can encounter when searching for grey literature[37].

We decided to set our stop criteria when our grey reference numbers achieved more than one-third of the scientific references we selected in the previous phase, i.e., 21 references[2] in total. Therefore, we collected 21 references from grey sources.

The 21 grey literature references were added to our 60 scientific references, for a total of 81 references (Figure 3). In the next steps of the protocol, we use the total references from both Scientific and grey literature.

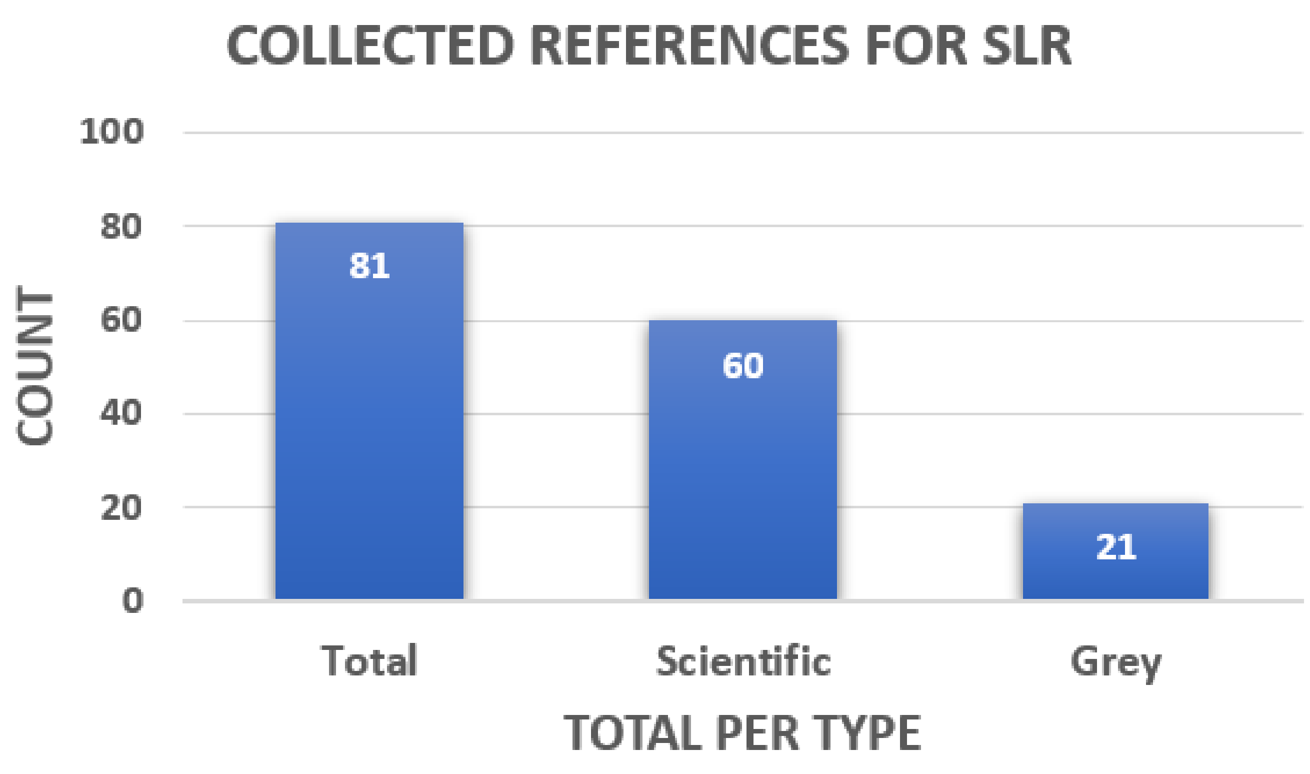


FIGURE 3 Total number of collected references used in our systematic literature review (SLR).

[1] http://gotocon.com/aboutjaoo/

[2] We collected 60 scientific references in the previous step. Therefore, one-third of 60 would be 20 references. Since we defined our criterion to be more than one-third (instead of exactly one-third), we needed at least one more reference.

3.6 | Data Extraction

For the data extraction procedure, we read all the 81 papers that we collected. For each paper, we focused on understanding and cataloging information that would address our research questions. We categorized the extracted information and stored it on a spreadsheet for easy access.

More specifically, the data extraction scheme had three main constituents. First, we extracted data about general information of the publication. Second, we extracted data for RQ#1 and then for RQ#2. Data for RQ#1 is specifically related to all the challenges in adopting microservices. Data extracted for RQ#2 is about the technologies which are used in implementing the microservices and solutions to the challenges are explained in the discussion (Section 5).

By the end of the extraction process, we identified 18 distinct categories related to challenges (i.e., RQ#1), and 44 different technologies which we grouped into 6 major categories (i.e., RQ#2).

3.7 | Data Synthesis

For the synthesis, we grouped the categories we extracted from the data extraction into another spreadsheet. That way, we have condensed information where the challenges and technologies/solutions are the main protagonist. For instance, by using this spreadsheet, we can easily find which papers presented a particular challenge.

We present the outcomes and findings of the synthesized data in Section 4.

4 | RESULTS

In this section, we present the results from our systematic literature review. First, we show a general overview of the collected paper's characteristics (Section 4.1). Then, we present our results to answer our first research question related to challenges (Section 4.2). Finally, we describe our findings to answer our second question on the topic of technologies (Section 4.3).

4.1 | General Analysis

Figure 4 shows an overview of the collected literature references we used in our systematic literature review. In our study, we have a total of 81 references, of which 60 are from scientific sources and 21 come from grey literature.

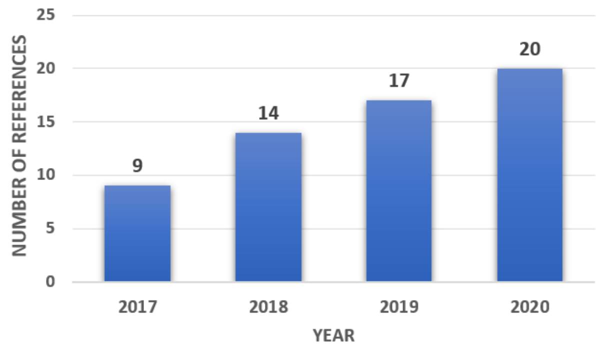
We can see the number of scientific references we deemed relevant for our study to grow each year after 2017 (Figure 4 a). Even though we reject papers from before 2017 from our study, we did not design our remaining selection criteria to favor more current papers. Therefore, the rise in the number of references per year is probably just a coincidence. On the other hand, most of the grey literature we used in this study is from 2016 (Figure 4 c). The reason is that the goto conference was our initial starting point to collected grey references. Particularly in 2016, many speakers were discussing microservices in the goto conference.

When we look at the types of literature, we have different classifications for scientific (Figure 4 b) and grey (Figure 4 d). Since all scientific references were papers, we classified them by their publication type (e.g., journal paper, conference paper, thesis). For grey, we classified the references by their media type and publishing format (e.g., video, blog). From the selected references, we could see there is a good variety of publications.

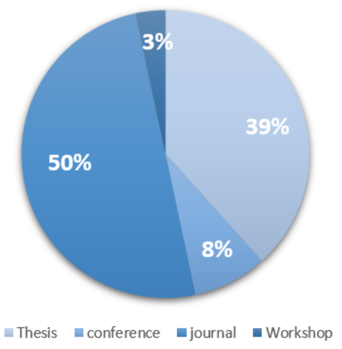
4.2 | RQ#1: What are the main challenges when adopting microservices?

In this section, we answer our first research question by analyzing the collected data. To answer RQ#1, we examined the spreadsheets created in the data synthesis phase from our 81 references.

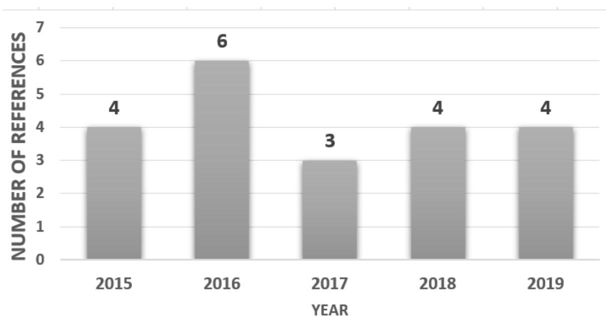
Figure 5 shows the challenges in adopting microservices that we discovered in our systematic literature review. It also shows how many distinct references in our selection discuss such challenges. According to our classification, we identified 18 challenges. The challenges mentioned on the most number of references are migration and performance, followed by scalability and testing. In our collected data, those are the major challenges of adopting microservices.



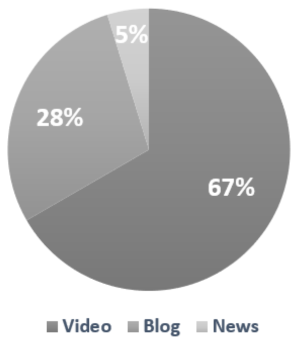
(a) Scientific references per year



(b) Types of the scientific references



(c) Grey references per year



(d) Types of the grey references

FIGURE 4 collected references Used in the systematic literature Review

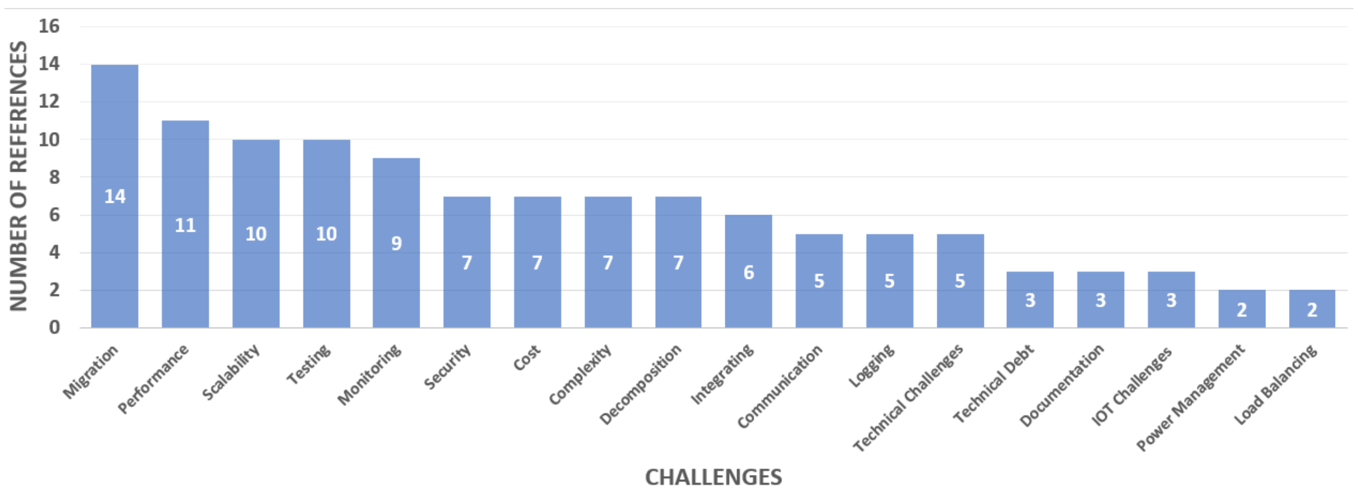


FIGURE 5 Challenges in adopting microservices

4.2.1 | Migration

When migrating an application the codebase update [3] are difficult tasks as observed from the publication by Tuuli and Wang[38,39].

Database migration is the main problem and risks if the application has a database, and especially its location inside the application docker container. To ensure the stability and maintainability of the application database, it would have to be migrated from its current location to a better alternative [38].

[3]Before undertaking any major structural changes, the codebase would have to be updated and unified to make any further changes easier. The technical problems of the outdated codebase, the difficulty of adding new features, and the inferior performance of parts of the application would be targeted

Hosting service migration is seen as the main problem in focus during this step were the uneven server load, as well as the difficulty of applying updates to the application[38]. Many organizations are planning and migrating their on-premise software to the cloud, starting with the Infrastructure as a service (IaaS) and Software as a service (SaaS) models. Nonetheless, they are facing some challenges and difficulties, mainly in the Platform as a service (PaaS) model, related to the complexity in integrating the legacy and internal systems[40]. It requires a different way of thinking compared to the traditional way of software architectures, often leading to the migrations and creations of these architectures being long and costly processes[41].

System application migration is becoming an emerging issue with different challenges. Migration of the system to microservice optimizes decentralization, replaceability, and autonomy of software architectures. Although researchers are not convinced of any specific definition of microservice, its modeling techniques, and its properties, it is aware of system migration to microservices[42].

Migration of the resulting application towards the new intended state is still a (partially) manual and labor-intensive step[43]. If on the one hand microservices can help in achieving a good level of flexibility (e.g., by promoting low services coupling, higher maintainability), on the other hand adopting a microservice-based architecture may bring higher complexity[44]. The microservice architectural style has become an essential element for the development of applications deployed on the cloud and for those adopting the devops practices. Nevertheless, while microservices can be used to develop new applications, there are monolithic ones, that are not well adapted neither to the cloud nor to devops. Migrating these applications towards microservices appears as a solution to adapt them to both.

Companies have been widely investing in modernizing the software architecture and development processes of their products. By migrating away from software monoliths towards the emerging microservice architecture style, an agile and robust software system with a compatible development process can be accomplished. Despite this, modernization can be highly challenging from a practical point of view. Available software solutions, which aim at supporting the restructuring of software monoliths into microservices, often do not satisfy the arising demands of the developers. While these products offer diverse static analysis functionalities, dynamic analysis aspects are mostly missing. Yet, especially a dynamic analysis can provide invaluable information about overlooked characteristics of the software system[22].

Serious efforts have been undertaken to move from monolithic architectures to microservice architectures. The common problem in these efforts is to identify from monolithic applications the candidates of microservices which includes the programs or data that can be turned into cohesive, standalone services; this is a tiresome manual effort that requires analyzing many dimensions of software architecture views and often heavily relies on the experience and expertise of the expert performing the extraction[45 46].

4.2.2 | Performance

Goldsmith[6] talks about challenges related to monitoring, latency, and problems with completely autonomous teams. The distributed aspect of a microservice system is a significant challenges[35]. Its distributed nature impacts the performance of the system. Calls to a single service might trigger a cascade of additional calls to other services distributed across a network, each adding its latency. The increase in resource usage may cause a microservices-based application to execute slower[47,48]. It can be hard to achieve the same level of performance as with a monolithic approach because of latencies between services. The communication between multiple microservices can introduce performance issues if the services are too fine-grained.

Representational state transfer application programming interface (REST API) have been favored by most software developers compared to all its previous approaches. But there is concern over its effect on performance when the size of the applications on the client-side grows[49]. Through an experimental setup, it is reported that the performance of a microservice model is lower than that in a monolithic model[50]. The complex dependencies between services bring new challenges to the monitoring analysis and quality assurance of system performance[51]. Monitoring the performance of the microservices are still challenging[52,28].

Although microservice offers opportunities for accommodating ever-growing workloads in the cloud, its true scalability potential has not been exploited yet. The reason behind this is the heterogeneity of microservices is never well exposed to the data center management layer, unavoidably causing power allocation imbalance and power capacity waste. Oftentimes, they overlook the sensitivity of performance to power budgeting of each microservice. As a result, it could waste precious power budget on some less critical microservices while leaving inadequate power budget to the most important ones[53]. Ensuring our application runs smoothly post-migration, and that the user experience was not negatively impacted, we need a way to compare performance metrics from pre-and post-migration and this can be very difficult.

4.2.3 | Scalability

The versatility of microservices is a strongest characteristics, but that versatility comes at a price. Scaling can involve handling several different components and services. This means that either all the components need to scale at the same time, or we need a means of identifying which individual components to scaleup, and a method of ensuring it and it should still integrate with the rest of the system[8].

Microservices require a very different approach to monolithic systems when it comes to scaling. When scaling microservices, one needs to consider both the individual components and the system as a whole. Doing so requires that the dependencies of each microsystem also scale with it. A modern and successful microservice system can expect a steady rise in traffic, and therefore resource demands, overtime[47,54]. Scaling a website to handle more traffic at peak times without wasting resources[55] is extensive research to any web company that has issues with rising costs as demand for their website increases. Some of the scalability challenges include enabling efficient scaling and high availability for services for the business requirement. Each module having different scalability criteria could be overpowering[56].

The hindrance of how to most effectively and efficiently auto-scale a web application to optimize for performance while reducing costs and energy usage is still a hurdle. In particular, this problem has new relevance due to the continued rise of internet of things and microservice-based architectures. A key concern, that is often not addressed by current auto-scaling systems is the decision on which microservice to scale to increase performance[57].

4.2.4 | Testing

An essential part of every software project is testing. Testing microservices can become challenging, particularly the integration tests[58]. To write an effective integration test case, the Quality assurance (QA) engineer should have good knowledge of each of the services that are a part of the solution. Another reason why testing a microservices-based application is difficult is because such applications are generally asynchronous. It is challenging to estimate the reliability of microservices, which is difficult to perform before release due to frequent releases/service upgrades, dynamic service interactions[59]. In contrast with a monolithic architecture, it is easier for microservices to test small, independent components.

Nevertheless, testing the system, in general, becomes more challenging. A large number of integrational tests should be implemented to verify that the system is correctly working[17]. Splitting a single process application into multiple services causes the testing process to be more challenging[60]. "It has been an incredible challenge getting failure testing instilled as a requirement" claimed Ranney[35]. Korbes[61] stated "I want my monolith back!" to demonstrate the sentiment often echoed because developing multi-service, multi-container systems in the kubernetes world which lacks a lot of the convenience monoliths used to have. Straight-forward builds, trivial testing between components is needed. Some industrial blogs claimed that testing is the biggest challenge with microservices[62,54].

4.2.5 | Monitoring

The traditional forms of monitoring and diagnostics will not align well with microservices since we have multiple services making up the same functionality previously supported by a single application. When a problem arises in the application, finding the root cause can be challenging. If we do not have a means of monitoring and tracking the path a specific request took, like how many and which microservices were traversed for a specific request coming from a user interface. Monitoring systems now need to provide integrations with a large and dynamic ecosystem of third-party platforms to provide complete observability[48]. In containerized workloads, we have to monitor multiple layers, dimensions, and the power consumption it makes[63]. Microservices require continuous and automated monitoring.

Monitoring one application running in multiple instances is easier than monitoring multiple services running in multiple instances. Typically with microservices, the number of instances is much higher than with monoliths[20]. Compared with the traditional monolithic architecture, the microservice architecture style divides a system into different microservices that run in the distributed system. The complex dependencies between services bring new challenges to the monitoring analysis and quality assurance of system performance[51,28].

Microservice monitoring brings specific challenges. They are often short-lived, which means monitoring over a longer period can be more complicated, and there may be more pathways through which the service is reached, potentially exposing issues such as thread contention[64]. Monitoring microservices can be a key factor to detect service failure earlier. However, few studies have been done on monitoring and analysis of microservice performance[52,65]. Although powerful tools exist for monitoring microservices, they are usually complex and suitable for monitoring large and complex microservice systems. The dashboard is also too complex so it makes the tools not easy to understand for novice users[66].

4.2.6 | Security

Microservice architectures meet organizations' need for speed, but the tradeoff is the introduction of new security challenges. Each separate service must then be able to communicate and interact with the other and this is achieved through the cloud. It is simple to see how the architectural differences can impact security. We have moved from securing a single application kept within a single operating system to a multiplicity of parts dispersed in a multi-cloud environment. There is a greater area for attack[17]. Microservices is triggering a closer interaction between the development and operation teams to support the applications lifecycle. Both development and operations need to have a good understanding of the processes involved and to ensure any security risks can be reduced[67,11,68].

Data generated in a microservices architecture moves, changes, and is continuously interacted with. Data is also stored in different places and for different purposes. Owners of data assets need insight into the life cycle and the dynamics of data to avoid breaches. Data leaks might happen[69]. Security is a major challenge that must be carefully thought of in microservices architecture. Services communicate with each other in various ways creating a trust relationship.

For some systems, a user must be identified in all the chains of service communication[65].

4.2.7 | Cost

'Scalability comes with costs' states Koschel et al because communication between microservices becomes more complex. It is no longer possible for components to communicate with each other via simple method calls. Instead, inter-process communication mechanisms are required. This has an impact on how to design the interfaces. Method calls are fast and can be made often without running into any problems. But remote calls are expensive and have high latency compared to simple method calls [70] [55].

From the survey by Zhang et all, we understand that microservices can handle multiple diversity of technology stacks however, the cost of setting up the technical framework was expensive and it took developers a long time to cope up with the obstacles of excessive technology stacks. Foreign technologies and tools spent much effort and cost on setting one by one. To understand the legacy system the practitioners have no choice but to rely on the costly manual code reading and analysis of the dependency among components [64].

The complexity of supporting both the microservice and the monolith will linger for a long since it takes a long time to entirely replace the monolith. The longer the migration process takes, the more it costs to maintain the two infrastructures [71] [?] ,[33].

In the survey by Leo et all, the participants mentioned that respondents seem reluctant on migrating towards a microservice architecture because of the reasons that microservices are complex, new, and not yet standardized enough and that the migration is too costly and time-consuming[?] .

4.2.8 | Complexity

Microservices might sound simple as separate individual services but it is the place for complexity. Designing microservices that tackle the complexity not only of the services but of the system as a whole is challenging because a microservice-based application is a network of different services that often interact in ways that are not obvious. The overall complexity of the system tends to grow. Many organizations are moving from on-premise software to the cloud but they are facing some challenges and difficulties mainly in Platform as a service model, related to complexity in integrating legacy and internal systems[40,17].

The adoption of microservices does not reset a system's development and maintenance complexity but is often viewed as a tradeoff between inner and outer complexity. Microservices make complexity more visible and unambiguous hence facilitating its proper handling and management. This represents a total shift in complexity from inside the services to the connections between them and their management[71,72].

Microservices add complexity in the application architecture because of the sheer number of moving parts, hence without automation and use of devops kind of model which encourages continuous integration, continuous delivery, and provisioning, etc., it is very challenging to manage microservice-based applications. Without adopting devops culture, organizations will not be able to realize the value of microservices[73].

With a growing focus on adopting microservices, developers often find it frustrating when someone modifies an application programming interface in a microservice. It is incredibly difficult to fully understand the impact of that change, which makes it difficult to throw blame around. It is complex task to search through all microservices calling that application programming interface[74].

4.2.9 | Decomposition

Decomposing a system into independent subsystems is a task that has been performed for years in software engineering. Recently, the decomposition of systems took on another dimension and especially microservices[75,48]. In microservices, every module is developed as an independent and self-contained service. Decomposing a monolithic system into independent microservices is critical and complex tasks and several practitioners claim the need for a tool to support them during the slicing phase to identify different possible slicing solutions[14]. The decomposition is usually performed manually by software architects[64,76]. One of the main challenges is to design microservice and creating services that are not too large or too small and contain the right amount of functionality. Many authors proposed domain driven design (DDD) as the best modeling approach that could help overcome this challenge of designing. However, how to apply this idea in practice is not clear to everyone[77].

It can be determined that the fine-grained splitting of the microservice framework can promote the reasonable sharing and effective integration of data. However, how to determine the granularity of microservice splitting is a multiparameter and multiobjective decision-making problem, which is also a key basic problem to be solved urgently both in academic research and application[78].

4.2.10 | Integrating

The microservices architecture fosters building a software application as a suite of independent services[40]. When we have to realize a given business use case, we often need to implement the communication and coordination between multiple microservices. Therefore, integrating microservices and building inter-service communication have became challenging tasks.

It is not recommended to tie the integration between services to some specific technology, because developers might want to use different programming languages when implementing services. There are also multiple additional challenges when thinking about the integration of microservices. The interface of microservice should be simple to use and it should have good backward compatibility so when new functionalities are introduced, the clients using the service do not have to be necessarily updated[79,64,20]. The integration of the model environment and the runtime environment is seen as crucial to achieving fine-grained evolution[43].

4.2.11 | Communication

Nowadays, there are software architectures following a microservice style. A microservice architecture consists of several independently developed and operated services. Often, issues (e.g. interface model changes) or design decision changes must be communicated between multiple teams. However, this is difficult and current approaches to communicate issues affecting multiple projects or teams come with a communication overhead[80].

In a distributed system, the components should be able to communicate with each other and in microservices-architecture, it is no exception. In a monolithic application, the components invoke one another through method or function calls[79]. In contrast, a microservices-based application is distributed in nature with each service running confined from another service. Hence, services in a microservice-based application must communicate using inter-process communication. However, inter-service communication in a microservices-based application poses numerous challenges. Since microservices are distributed in nature, the remote invocation of these services is a challenge and one should understand the necessary patterns to overcome the challenges involved. Since services in a microservice-based application communicate with one another using Inter process comunication (IPC) mechanisms one should consider issues such as how the services will interact, and how to handle failures[64].

4.2.12 | Logging

Logging is a perfect example of a cross-cutting concern. Code that needs to span numerous modules at different levels of the codebase[67]. When we split our application into silos, logging is also split over every service. Since the log messages generated by microservices are distributed across multiple hosts, without a good strategy for logging we will be unable to understand the issues that might occur in the application[35,20,39,64].

4.2.13 | Technical challenges

Microservice architecture provides also multiple new challenges that have to be solved to get the benefit from them. These challenges are such as the handling of distributed transactions, communication between microservices, separation of concerns in microservices[20,81].

- Data consistency and distributed transactions: There is only a single database with a monolith where all the transactions are applied or roll-backed if there is an error in the middle. When we migrate to the microservices architecture, they can no longer ensure the same atomicity, consistency, isolation, durability (ACID) properties. Since the original data schema is decomposed in multiple services, most of the time each service with its database. atomicity, consistency, isolation, durability properties are important because they ensure accuracy, completeness, and data integrity.
- Setup and execution of the initial prototype: The initial setup of the microservices architecture demands more effort than a monolithic system. The microservice architecture implies that multiple services are developed, deployed, and executed in the production environment. With a monolith, this is simpler as there is a single service to set up and execute.
- Decomposition of the pre-existing system with the proper granularity and low coupling: The existent monolith supports an entire business model in a single executable component, usually with a single database schema for persistence. When adopting the microservices architecture, one of the first steps is to decompose this single piece into multiple services, each with its well-defined boundaries, to achieve low coupling in a distributed system modeled around a business domain[82,83].
- Successful tools like docker are frameworks built around container engines that allow containers to act as a portable way to package applications to run it. It means that it covers an application tier or node in a tier, which results in the problem of managing dependencies between containers in multi-tier applications[27].

4.2.14 | Technical debt

Technical debt is when long-term code quality is traded for short-term gain[84]. Usually, it is attributed to shortcuts and workarounds in the source code of the software, where developers choose quick and messy implementations instead of spending time on code quality and maintainability[38]. With swiftly evolving technologies, the risk of accumulating technical debt becomes more habitual and must be taken into account at various stages of a software project. Technical debt can cover a lot of various areas, from lack of documentation to architectural design flaws. The main types of technical debt are traditional, or code quality, debt, and architectural debt[85,10].

4.2.15 | Documentation

The microservices are highly beneficial to use. However, it introduces a high level of complexity and new challenges regarding enterprise architecture (EA) model maintenance[6]. During the conducted survey in the German information technology, market to analyze the status quo in the adaption of microservices and what challenges organizations face while documenting microservice-based information technology landscape from an Enterprise Architecture perspective.

The identified challenges are synthesized into four classes: content, assignment, tooling, and business-related challenges. The content-related challenges are mainly because of documenting manually and the documentation was either wrong or out of date. Assignment challenges are usually due to which business-related assignment is unknown and technical assignment is unknown. The tooling challenges are when the tools are not up-to-date with microservice architecture and no appropriate visualization for different stakeholders. Business and organization challenges are due to missing motivation and clarification of responsibilities[86,46].

4.2.16 | Internet of things (IoT) challenges

The internet of things has many problems and challenges regarding data exchange in large-scale heterogeneous networks and interoperable elements. Nevertheless, achieving high-availability in such a distributed system is not without challenges. When addressed competently those benefits come with challenges like discovering services over the network, security management, communication optimization, data sharing, and performance[87].

The challenging question is how to properly size microservices and how to properly deal with data persistence to avoid sharing of data across services. These two concerns are closely related. But composing the right level of service component granularity is still a challenge. Some problems originate due to the lack of chosen development framework knowledge and communication protocol limitations[56].

The difficulties of scalability and interoperability, defining and implementing a microservices-based middleware platform with the orchestration of different internet of things system components such as devices, data sources, data processors, storage, etc. The difficulty is the data exchange in large-scale heterogeneous network elements to achieve interoperability[75].

4.2.17 | Power management

In a power-constrained data center, blindly budgeting power usage could lead to a power unbalance issue. Microservices on the critical path may not receive an adequate power budget. This unavoidably hinders the growth of cloud productivity. The emerging trend of decomposing cloud applications into microservices has raised new questions about managing the performance/power trade-off of a data center at a microsecond scale[88].

4.2.18 | Load balancing

There are two types of load balancing which are server-side and client-side load balancing. In server-side load balancing, the instances of the service are deployed on multiple servers and then a load balancer is put in front of them. All the incoming requests traffic firstly comes to this load balancer acting as a middle component. It then decides to which server a particular request must be directed to based on some algorithm. The challenge with this load balancer is that it acts as a single point of failure and the complexity of the system increases[89]. The server-side load balancer's logic is a part of the client itself, it holds the list of servers and decides to which server a particular request must be directed based on some algorithm. The drawback here is that the load balancer's logic is mixed up with the microservice code[74].

4.3 | RQ#2: What are the main technologies or solutions used for implementing microservices?

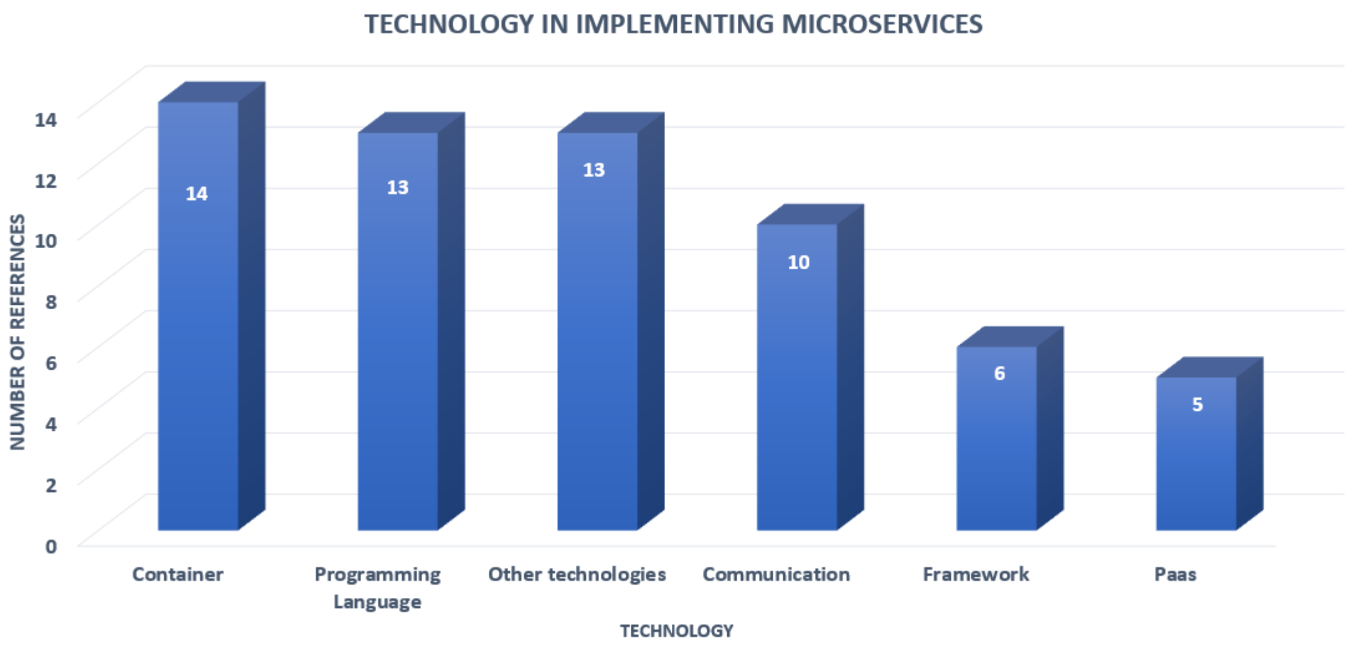


FIGURE 6 Technologies per group used for Implementing Microservices

In this section, we discuss RQ#2, which is about the technologies used in the implementation of microservices according to our studied sources. Figure 6 shows the major groups of technologies that appeared in our studied references. We classified the technologies into 6 major groups. Figure 7 shows the distinct technologies proposed to implement microservices. In total, we

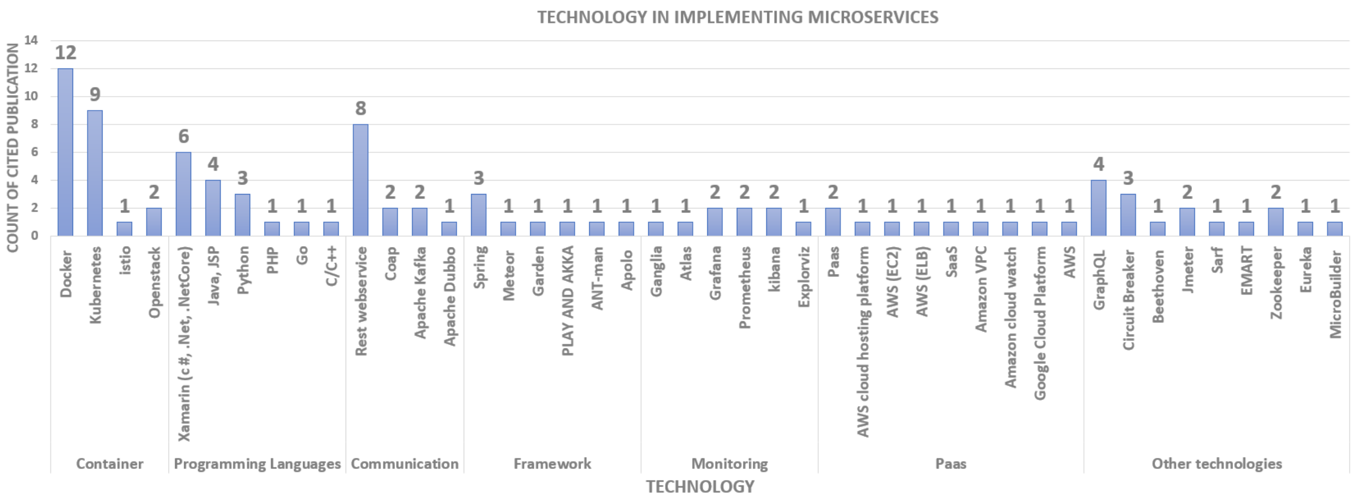


FIGURE 7 Technologies in implementing microservices

discovered 44 different technologies in our data. The number of references of the groups showed in Figure 6 is the union of the references of the distinct technologies (Figure 7) we classified in each group.

4.3.1 | Containers

Containers encapsulate discrete components of application logic provisioned only with the minimal resources needed to do their job. Containers and microservices enable developers to build and manage self-healing microservice-based applications more easily.

Docker: Docker is a platform for developing, shipping, and running applications. Developers can implement applications very fast by using docker[27,63,56]. Moreover, it is simpler to create required services separately and manage them as microservices without affecting other services[41,88,20,90]. We can create a docker image for service and by running some docker commands, we can view our microservice application inside a docker container. Docker swarm applications can be deployed as services in a swarm cluster[83,28,57].

Kubernetes: Kubernetes is a container orchestration system that is well-suited to automate the management, scaling, and deploying of microservice applications[17,63,56]. It is also possible to run kubernetes locally using MiniKube[41,20]. Kubernetes ensures that the desired state (i.e., the state that we want the system to be in) and the actual state (i.e., the state that the system is actually in) are always in sync. Kubernetes continuously monitors the health of the cluster and ensures that the system is self-healing[90,83,28].

OpenShift: Red hat openshift is an open-source container application platform based on the kubernetes container orchestrator for enterprise application development and deployment[50,90].

4.3.2 | Programming Language

When it comes to microservices development, a natural question is what language to choose. Developers have many options regarding programming languages to develop microservices. The programming languages mentioned in the citations are shown below.

Xamarin (C#, .net, .netcore): asp.net, the web framework for .net, makes it easy to create the application programming interface to become microservices. Asp.net comes with built-in support for developing and deploying your microservices using docker containers[79,91,25,50,82,83].

JAVA (Java, jsp): WSO2MicroservicesFramework for java has been used in implementing microservices[27,56,81,28].

Python: Python programming language uses flask application for building microservices[49,56,88].

PHP: PHP is also another programming language that can be used for implementing microservices[55].

Go: Golang, also known as ’go’ is popular for its concurrency and application programming interface support in terms of microservices architecture. It can be a good choice for microservices development. [79].

C/C++: C++ is a programming language with imperative and object-oriented features that help developers write fast, portable programs. C++ is especially popular in areas where performance is crucial. Also, the compilation time and execution time of c++ is much faster than most other programming languages[49].

4.3.3 | Communication

The goal of the microservice architecture is to create loosely coupled services, and communication plays a key role in achieving that. Therefore, services must interact using interprocess communication protocols such as http, amqp, or a binary protocol like transmission control protocol (TCP), depending on the nature of each service. Below we present the communication protocols used in the studied publications and grey literature.

REST API: REST API stands for 'representational state transfer' and It is a set of rules that developers follow when they create their application programming interface[71,64]. One of the rules states that a user should be able to get a piece of data by using a specific URL[70,74]. Microservices function as the building blocks of the application by performing various services, while restful application programming interface function as the communication that integrates the microservices into an application.[79,17,91,50].

CoAP: CoAP and http are both based on the rest model and can be used as communication protocols to expose restful Web Services in a mobile device cloud. We will compare coap with http to better explain it. coap is a restful web transfer protocol optimized for communication between resource-constrained networks and nodes. CoAP uses the User datagram protocol (UDP) as its transport protocol, unlike http which operates on top of the reliable transmission control protocol and can be too complex for constrained environments. CoAP is not a blind compressed version of http, but a subset of rest common, with support of uniform resource identifier (URI) and http verbs, that gear towards machine-to-machine applications, hence it is an effective protocol for micro-services hosted on mobile devices[79,87].

Apache Kafka: Apache kafka is an open-source distributed event streaming platform. Apache kafka is a powerful instrument for microservice architectures, which solves a variety of problems such as low-latency ingestion of large amounts of event data[39,92].

Apache Dubbo: Apache dubbo is a remote procedure call or remote procedure control-based service framework for programming. Dubbo is also a service governance framework, which provides service governance solutions such as service discovery and traffic scheduling for distributed microservices[64]

4.3.4 | Framework

Microservices can be implemented with a plethora of frameworks and using frameworks can help speed up the development of microservices. Below we present the frameworks used in implementing microservices in the analyzed references.

Spring framework: Spring is an open-source framework based on the java platform. Spring boot is part of the spring framework family to fastly create stand-alone applications. The distributed nature of microservices brings challenges. Spring helps us mitigate these. With several ready-to-run cloud patterns, spring cloud can help with service discovery, load-balancing, circuit-breaking, distributed tracing, and monitoring. It can even act as an application programming interface gateway [81,46,21].

Meteor framework: Meteor is a framework meant to facilitate the whole web application development, encompassing the frontend, backend, as well as database. As such, it can be considered a full-stack web application framework. Meteor is written in javascript and is based on Node.js. The concept of having a proper microservices architecture using multiple meteor applications should be achieved reliably [38].

Apollo framework: Apollo is a great fit with microservice architectures and modern user interface frameworks like React. It serves as an abstraction layer that decouples services and applications so that each can be developed independently of the other, in any language and on any platform [6].

Garden framework: Garden framework works like a spinnaker for implementing microservices[61].

ANT-man framework: an auto, native and transparent power management framework that can exploit fine-grained microservice variability for system efficiency[88].

4.3.5 | Monitoring

Monitoring is a process of reporting, gathering and storing data. Some of the technologies/tools which are discussed before for monitoring microservices.

Ganglia: Ganglia is one of the distributed monitoring tools for high-performance computing systems[63].

Atlas: Atlas is a cloud monitoring tool which is newly introduced by netflix[48].

Explorviz: Explorviz is a monitoring and visualization approach, which uses dynamic analysis techniques to provide a live trace visualization of large software landscapes[22].

Play AND Akka: The play framework is a web framework for the java virtual machine (JVM) that breaks away from the servlet specification. Play embraces a fully reactive programming model through the use of futures for asynchronous programming, work stealing for maximizing available threads, and akka for distribution of work. They are useful in building individual services, and leveraging both http and kafka for inter-service communication[87].

Grafana: Grafana is an open-source platform for data visualization, monitoring and analysis[81,20].

Prometheus: Prometheus is an open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach[81,20]

Kibana: Kibana is a free and open user interface that lets us visualize your elasticsearch data and navigate the Elastic Stack[81,20]

4.3.6 | Platform as a Service (PaaS)

Running microservices on a platform as a service fabric decreases solution fragility, reduces operational burden, and enhances developer productivity[40,93].

Amazon web services (AWS) cloud hosting platform: Among the primary benefits of a microservices architecture are the utilization and cost benefits associated with deploying and scaling components individually. While these benefits would still be present to some extent with on-premises infrastructure, the combination of small, independently scalable components coupled with on-demand, pay-per-use infrastructure is where real cost optimizations can be found[55].

AWS auto-scaling group (EC2): Amazon elastic compute cloud is a part of amazon cloud-computing platform, amazon web services, that allows users to rent virtual computers on which to run their computer applications[55].

Amazon AWS elastic load balancer: Elastic load balancing automatically distributes incoming application traffic across multiple targets[55].

Software as a service (SaaS): Microservices are the resulting standalone services after breaking a software application down into separate components that perform their functions without being embedded in the application itself. Microservices are perfectly suited for saas, where each service is assumed to be part of a larger system[25].

Amazon VPC: Amazon virtual private cloud (Amazon VPC) is a service that lets us launch aws resources in a logically isolated virtual network that we define. Amazon[11] uses amazon vpc to secure the traffic across cloud.

AWS Lamdba: AWS lamdba is a compute service that lets us run code without provisioning or managing servers. Lambda runs the code only when needed and scales automatically. AWS lambda is used to reduce infrastructure costs[26].

4.3.7 | Other Technologies

GraphQL: Graph query language (GraphQL) is considered as an alternative for Representational state transfer application programming interface. It is most useful when it is able to combine different data sources into one and serve that up as one unified application programming interface. GraphQL hides the fact that we have a microservice architecture from the clients. From a backend perspective, we want to split everything into microservices, but from a frontend perspective, we would like all our data to come from a single application programming interface. Using graphgl is the best way that lets us do both[49,39,43,72].

Circuit breaker: Hystrix, circuit breaker pattern allows us to build a fault-tolerant and resilient system that can survive gracefully when key services are either unavailable or have high latency[20],[24],[10].

Beethoven: Beethoven is a platform composed of a reference architecture and a domain-specific language for expressing microservice communication flows[94].

Apache JMeter: Apache jmeter is an apache project that can be used as a load testing tool for analyzing and measuring the performance of a variety of services. It is also used for performance-test in the microservice applications [53 50].

SArF : SArF is an sofware clustering algorithm and it has two characteristics. First, sarf eliminates the need of the omnipresent-module-removing step which requires human interactions. Second, the objective of sarf is to gather relevant software features or functionalities into a cluster. Thus, it is used to find the candidates for microservice from the source code [45].

EMART: Enhanced microservice adaptive reliability testing (EMART) is a tool for testing the applications[59]

Apache Zookeeper: Apache zookeeper is an open-source server for highly reliable distributed coordination of cloud applications. However, this architecture makes it hard to scale out to huge numbers of clients. ZooKeeper node called an 'observer' which helps address this problem and further improves zooKeeper's scalability[20,81].

Eureka Server: Eureka server is an application that holds the information about all client-service applications. Every micro service will register into the eureka server and Eureka server knows all the client applications running on each port and internet protocol address[10].

MicroBuilder: MicroBuilder is the tool used for the specification of software architecture that follows representational state transfer microservice design principles[74]

5 | DISCUSSION

In this section, we describe the possible solutions for challenges that we encountered during our systematic literature review. Whenever possible we also link the technologies we presented to the appropriate challenge. Therefore, we discuss the links between the results we presented from our two research questions (Sections 4.2 and 4.3), focusing on how to address the challenges, i.e., we show the challenges we encountered and discuss the solutions and technologies used to address each challenge.

5.1 | Migration

Before splitting or migrating the application into multiple services we need to understand its scope and architecture. Components that are used by most users should be considered first for migration.

To find the candidates for microservice a method that identifies the candidates of microservices from the source code by using software clustering algorithm SArF with the relation of "program groups" and "data" can be used. The method also visualizes the extracted candidates to show the relationship between extracted candidates and the whole structure. The candidates and visualization help the developers to capture the overview of the whole system and facilitated a dialogue with customers[45].

Database migration, mongoDB and meteor framework, both provided the necessary tools for updating the upcoming migration host. Google cloud platform (GCP) is an option for the application destination. Public cloud offered many services that would help with the goals of this optimization, it was a logical choice to migrate the application hosting server onto a Cloud Service Provider (CSP)[38]. Also, GCP offers its Kubernetes as a service (GKE) as part of its services. It was suitable as the target destination for the application containers[25].

Migration triggers are handled by different strategies as follows[43]:

- Change of model by modeler - all existing models could be changed instead of a single model.
- Change to the transformation environment - updated packages need to be redeployed to upgrade the instances.
- Change to the runtime environment - the application needs to be migrated and then Incremental migration.

The migration process can be carried out by following ideas below:

- Microservice identification where available software artifacts (e.g., source code, documentation, execution traces, etc.) of the monolithic application are to be analyzed to determine the corresponding microservices.
- Microservice packaging where the necessary transformations are to be performed on the source code of the identified[46].
- Many organizations consider a gradual process of moving to the cloud with hybrid cloud architecture[93].
- For managing common code, the study enumerated several options employed by the participants. While most participants rely on promises and challenges of Microservices[39], an exploratory study shared libraries which are the emerging sidecar technology could be a promising and elegant microservice-native solution to this problem.
- Numerous options for managing application programming interface (API) are there. API gateways and client libraries, although not adopted by most of the practitioners and have a high potential to mitigate the burden related to managing API changes[39].
- Options for supporting product variants are there. Each of these solutions has advantages and disadvantages, with most practitioners applying a role-based access model to support different product offerings within the same code base.

5.2 | Performance

The increase in resource usage may cause a microservices-based application to execute slower. We can overcome this challenge by introducing additional servers. Logging in any application can be a part of the solution - we can log performance data and detect the problems. We should take advantage of logging to store performance data in a repository so that we can analyze the data at a later point in time. We can implement throttling, handle service timeouts, implement dedicated thread pools, implement circuit breakers and take advantage of asynchronous programming to boost the performance of microservice-based applications[49,50,51].

Apache JMeter can be used to performance-test our microservice applications. Microservice criticality factor metric (MCF) to measure the overall impact of performance scaling on a microservice from the whole application's perspective[53]. Beethoven (a platform composed of a reference architecture) could be used to measure and increase the performance of microservices[94].

5.3 | Scalability

To scale successfully, each microservice needs to scale both individually, and as part of a larger system. Doing so requires that the dependencies of each microsystem also to scale with it. We can containerize each microservice using docker[57,90].

Kubernetes can manage containers. Kubernetes can be configured to auto-scale based on the load. Kubernetes can identify the application instances, monitor their loads, and automatically scale up and down[55,56]

5.4 | Testing

The best approach to solving testing challenges is adopting various testing methodologies, tools and leveraging continuous integration capabilities through automation and standard agile methodologies. Some of the options to consider are enhanced microservice adaptive reliability testing (EMART)[59], Apache Jmeter[50], testing as a service[95], integration testing, test doubles[60], end-to-end testing, consumer-driven contract (CDC) testing,[4] a/b testing[17,58]. Good quality tooling is needed like chaos monkey which will do destructive testing in the production environment to understand the resilience of system[48].

5.5 | Monitoring

We can monitor microservices by using open-source tools like prometheus with grafana APIs by creating gauges and matrices, google cloud platform (GCP) stackdriver, kubernetes monitoring, amazon cloudwatch[20,64,65,28].

Ganglia monitoring system is one of the distributed monitoring tools for high-performance computing systems[63]. The proposed monitoring framework includes an integrated scheduling tool, data collector, big data storage, elastic scaling manager[51]. For device failure detection in microservices kieker trace analysis tool is used to analyze the application and the trace tool used for visualizing are graphViz, gnuplotzunit[52].

The conceptual model of a dashboard for monitoring is proposed and this model consists of components, the interaction between components, and the requirements for each component. A black-box approach and monitoring of the microservices through its endpoint is done[66].[6] heroic tool is used for monitoring at Spotify. Atlas - cloud monitoring framework is a newly introduced tool by netflix is getting popular[48].

ExplorViz uses kieker (dynamic analysis tool) to instrument software systems for a dynamical analysis of their runtime behavior[22]. Instead of using kieker's analysis methods, explorviz uses its own and instructs kieker to send the gathered monitoring data called records. The Analysis service creates traces from these records which are sent to the Landscape service which creates the models which are visualized in the frontend service of explorViz. The assessment confirmed that one of the central prerequisites for the effective usage of explorViz is a thorough prior knowledge of the entire monolithic architecture. The evaluation also demonstrated that finding a suitable division into self-contained systems or microservices is a highly complex process that requires experience and time.

[4]Consumer-driven contract testing ensures that a provider is compatible with the expectations that the consumer has of it

5.6 | Security

The first step to a secure solution based on microservices is to ensure security which should be included in the design. Some fundamental tenets[5] for all designs are:

- Encrypt all communications (using https or transport layer security).
- Authenticate all access requests.
- Do not hard code certificates, passwords, or any form of secrets within the code.
- Use devsecops tools designed for microservice architecture environments to scan code as it is developed.
- Define the APIs and strictly make sure all communications comply.

In a typical microservices architecture, communication between the services can be in the same or different machines or even between different data centers. Due to this complexity in the communication, security in a microservices-based application is important to authorize access to a protected resource. In a monolith, the facade pattern aggregates the data that is retrieved from multiple services. On the contrary, in a microservices-based application, the API Gateway is used for the same purpose. The API Gateway pattern can be used in a microservices-based application to secure access to the microservices by abstracting the underlying microservices from external clients. The other strategies that are adopted include implementation of ssl, oauth, and containerization[17,69,65].

Some of the best practices in securing microservices are building security from the start, deploying security at container level, multifactor authentication, user identity and access tokens such as oauth 2.0 and openid[68]. Amazon[11] uses amazon vpc to secure the traffic across cloud.

5.7 | Cost

Cost could be an significant factor when building microservices[70,48,33]. Adopting a microservices architecture quickly defray those costs by returning large amounts of business and technical value[55,41]. A microservice architecture, with fewer application dependencies and simple APIs will immediately reduce the time and money spent on application maintenance. Savings on application maintenance have shown to be more than enough to cover the initial costs within a few years[96]. A good set of guidelines and practices at the company level is needed. After the load was removed, the instances should be terminated automatically by aws to save cost[55].

Testing the off-the-shelf hardware to reduce hardware costs. Invest in the team to create white box solutions that focused on reducing costs, using a reference architecture. AWS Lambda functions can be triggered based on events ingested into amazon sqs queues, s3 buckets where aws manages the polling infrastructure on our behalf with no additional cost[71,64,70]. The use of services specifically designed to deploy and scale microservices, such as aws lambda is used to reduce infrastructure costs by 70% and Microservice implemented with play reduce up to 13% cost[26].

5.8 | Complexity

Microservices architecture can be more complex than legacy applications. To handle this complexity and reduce the risks involved, we should use the right tools and technologies in place. Complexity can be addressed for each and every situation. If the complexity is based on integrating the legacy and internal system on the cloud services then we can use the platform as a service (PaaS) model[40] like follows application platform as a services (aPaaS), database platform services (dbPaaS), function platform services (fPaaS), business analytics platform services (baPaaS), business process management services (bpmPaaS), business rule platform services (brPaaS), enterprise horizontal portal services (Portal PaaS), communications platform services (cPaaS). When the applications are moving on-premise to software as a service (Saas), directly changing the code is not feasible because many customers share one instance of an application code. Chauvel and Solberg[91] propose an approach to enable deep customization on multi-tenant saas using intrusive microservices.

Another possible solution is adding a service mesh to microservices that can improve visibility, monitoring, management, and security[17,71]. A service mesh allows developers to make changes without touching the application code itself. It provides the ability to mirror and monitor traffic on multiple versions of the same service, which lets developers test capabilities before deployment and determine the best way to route traffic through the system for specific types of use patterns[73,72].

MicroBuilder is the tool used for the specification of software architecture that follows REST microservice design principles. MicroBuilder comprises MicroDSL and MicroGenerator modules. The MicroDSL module provides the MicroDSL domain-specific language used for the specification of microservice architecture[74].

5.9 | Decomposition

In transitioning an existing monolith to a microservices, we would typically need to decompose the existing application into granular microservices[14]. During this transitioning process, we would typically need to decompose the monolith to building more and more granular microservices to suit the business needs. Once this is accomplished, there would be more moving parts in the application[76]. As a result, this would lead to operational and infrastructural overheads, i.e., configuration management, security, provisioning, integration, deployment, monitoring, etc. One of the ways to reduce these complexities is by using containerization. In using containerization, provisioning, configuration, and deployment of microservices would be simplified[64].

To define the functional scope of microservices many authors propose using domain-driven design[77,82,84]. The key concepts of domain-driven design that helps in defining scopes are:

- Step 1: analyze domain.
- Step 2: define bounded context.
- Step 3: define entities, aggregates, and services.
- Step 4: finally identify microservices.

A service can have the scope of microservice. Microservice can have the scope of a bounded context. For inter-microservice interaction, we can use domain events with asynchronous messaging API calls, and service data replication.

Some factors must be considered when defining the size of a microservice and when we are defining a microservice granularity. Moreover, several more factors must be considered and chosen wisely to achieve a successful implementation and performance. The most important point is balancing. Although we may consider different factors to define the level of granularity, the key purpose is to analyze all the applicable criteria and to evaluate the possible tradeoffs that will have to be made in this process. Key points to consider are as follows[78]:

- Get a clear picture of the solution architecture.
- Functional decomposition patterns must be applied, this helps in defining services and splitting them.
- Separate reusable activities into reusable services. To achieve the above, the key technologies used were service registration and discovery, remote service calling, circuit-breaker mechanism, service link tracking, and annotation interfaces.

5.10 | Integrating

Integration of APIs, services, data, and systems has been a challenging yet essential requirement in the context of enterprise software application development. Before integrating all of these disparate applications in point-to-point style was done, which was later replaced by the enterprise service bus (ESB) style, alongside the service oriented architecture (SOA). Integration platform services (iPaaS) provides a platform in the cloud to support the application, data, and system-to-system integrations, using a mix of cloud services, mobile apps, on-premises systems, and internet of things integrations.

A message-Oriented middleware services (momPaaS) was proposed to support the communication and integration between the different microservices implemented in the respective system, thus supporting the message exchange with different protocols. Enterprise horizontal portal services (Portal PaaS) can be used to provide a B2B portal that is integrated with the microservices layer. Integration platform services (iPaaS) was recommended for situations when the integration and exchange of information between cloud applications and on-premise and legacy applications are required [40].

Architecture style makes it possible to achieve fine-grained incremental migration. The integration of this architecture style manifests itself in two ways. First of all, the runtime environment should be able to host distributed microservices. The benefits of this are an improved upgradability, scalability, resilience, and resource sharing. Second, the transformation environment should consist of independent microservices. This enables the incremental transformation of the model through the pattern incrementality by traceability[79,43].

Continuous integration and continuous delivery both go hand in hand with microservices. Without these two practices, it becomes very hard to handle multiple services, their deployments, and validating the actions of the service[64,20].

5.11 | Communication

Two types of communication are taken into consideration. One is the communication between teams and the other is communication between services.

Often, enhancement or corrective maintenance issues (e.g. interface model changes, design decision changes) must be communicated between multiple teams. 'Multi-project coding issues' as a solution approach for communicating issues concerning multiple projects/services and teams in a qualitative way. A multi-project coding issue is a coding issue that can concern more than one project at the same time. They can link other coding issues that could concern other projects/services[80].

Unlike the internals of a monolith, microservices communicate over a network. In some circumstances, this can be seen as a security concern. Istio solves this problem by automatically encrypting the traffic between microservices[64]. Modules could be communicated using coap which has minimal overhead[79]. Apache kafka is one of the right solutions for communication between microservices and it is been used at ebay[68].

5.12 | Logging

One of the main criteria for a mature microservice-based development process is the robustness of the logging. It is better to be set up as early as the project starts[39]. Detecting failures quickly for automatically restoring services is critical and is typically achieved using logging and monitoring tools, such as grafana, kibana, logstash, and elastic stack[81].

Grafana has released loki, a solution meant to complement the main tool to better parse, visualize and analyze logging[20,64]. It is hard to identify issues between microservices when services are dependent on each other and they have a cyclic dependency. Correlation id can be passed by the client in the header to REST APIs to track all the relevant logs across all the pods/docker containers on all clusters. Zap app is been used at uber for logging[35].

5.13 | Technical challenges

Microservice architecture can be very useful but it also comes with technical challenges that have to be solved to get complete benefit from them. These challenges could be handling of distributed transactions, communication between microservices, separation of concerns in microservices etc.[20,81]. We provide some ideas on how to overcome technical challenges below as follows.

- For data consistency and distributed transactions- a saga patterns [5] could be implemented and avoid a two-phase commit [82].
- Testing complexity - integration tests and following domain driven design concepts.
- Setup and execution of the initial prototype - When the microservices architecture is being developed from scratch the team should try to use the monolith-first approach and then migrate.
- Creating uniformity across multiple services - static code analysis tools can be used.
- Distributed monitoring - the concept of Log aggregation should be used.
- Decomposition of the pre-existing system with the proper granularity and low coupling - most successful microservices migrations identified used Domain Driven Design to deal with this issue.

[5]Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions to alleviate these problems the notion called saga is proposed. A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. The database management system guarantees that either all the transactions m a saga are successfully completed or compensating transactions are run to amend a partial execution

- Database challenges - usage of circuit breaker.
- Communicational challenges - REST API.
- Testing challenges - unit testing, integration testing and end-to-end testing.
- Observability challenges - zookeeper, eureka server, docker.
- Organizational challenges - experimenting should not be restricted[20,81].

  For some of the application system it is good to mainly focus on technology choice. Strong knowledge of devops and cloud architecture patterns is requirement[83].

An orchestration plan describes components, their dependencies, and their lifecycle in a layered plan. A paas then enacts the workflows from the plan through agents. Paas can support the deployment of applications from containers[27]. In paas, there is a need to define, deploy and operate cross-platform capable cloud services using light-weight virtualization, for which containers are a solution. There is also a need to transfer cloud deployments between cloud providers, which requires lightweight virtualized clusters for container orchestration. Some paas are lightweight virtualization solutions in this sense.

Specific or generic solutions for paas platforms include

- Linux: Docker, lxc linux containers, openvz, and others for variants such as bsd, hp-ux, and solaris.
- Windows: sandboxie
- Cloud paas: Warden/Garden (in cloud foundry), lxc (in openshift).

Kubernetes is a more comprehensive solution that would tackle orchestration of complex application stacks that could involve docker orchestration based on the topology-based service orchestration standard tosca, which is for instance supported by cloudify paas[90,83**?** ,41,20,17,63,56].

5.14 | Technical Debt

The technical debts can cover a lot of areas but we address only the technical debt in code quality, architectural debt. There are two solutions, the first one is clean up, and the second is prevention.

For cleaning up, we can perform parallel rewrite, partial rewrite, and phaseout; for prevention: code reviews, acceptance criteria, workflow, and tests[38]. domain-driven design is one of the solutions to reduce technical debt. It enables us to organize a testable layer of code that is similar to our problem domain and is responsible for all the business logic of our app[84]. If the microservices configuration is done badly then the common way to fix this is to use configuration servers like Spring cloud config which handles the configuration by itself. Service discovery tools like eureka and consul could help too. When the debt is due to libraries used by some developers then the solution is to use multiple different libraries that have well-defined boundaries[10].

5.15 | Documentation

Documentation of microservices can be time-consuming and challenging that is why we need to follow some of the solutions like comprehensive documentation which should include description, architecture diagram, endpoint, dependencies, run books, contact information, onboarding guide, etc. The other solution is to update documentation as a part of the development cycle. Using a central location for all microservice documentation[46].

There are couple of challenges such as content, assignment, tooling, and business-related when documenting the microservice-based information techology landscape from an enterprise architecture (EA) model maintenance perspective. To document the EA model, Kleehaus[86] suggests a solution of automatically documenting the runtime data from the monitoring tool. However, the tool used for automated documentation is not yet described in his publication and he mentions to publish it in the future works.

5.16 | IoT Challenges

Containers are particularly useful to enable portability and heterogeneity for high-degree automation in testing, deployment, and operations. Apache kafka and spark for application resiliency. The Akka, and Play framework are useful in building individual services, and leveraging both http and kafka for inter-service communication[87].

The combination of edge and cloud computing is going to make the internet of things (IoT) rapid, light, and more reliable. IoT and cloud-edge computing are distinct disciples that have evolved separately over time. However, they are increasingly becoming interdependent, and are what the future holds. A crucial aspect is how to design a compound of cloud and edge computing architectures, and implement IoT effectively. An implementation of container-based virtualization which constructs kubernetes minion (nodes) in the socker container service independently for each service on the edge side can address IoT challenges. Moreover, docker and kubernetes when combined together provide the best solutions by effectively organizing containerized applications[63].

5.17 | Power management

There are couple of way to manage the power consumption which are described in the auto, native and transparent power management framework (ANT-man) solution. First, an auto power budgeting scheme for reducing the power coordination latency at the datacenter level. It can proactively determine the power budget tailored to each microservice. Second, ANT-Man proposes a native and transparent power control scheme to overcome the power configuration latency for each microservice. It enables super-fast power budget enforcement with nanosecond-scale performance scaling[88].

To unleash the performance potential of the cloud in the microservice era, we model it using a bipartite graph and a metric called microservice criticality factor (MCF) to measure the overall impact of performance scaling on a microservice from the whole application's perspective proposed. A novel system framework called servicefridge that leverages MCF to jointly orchestrate software containers and control hardware power demand is also proposed. servicefridge allows the data center to reduce its dynamic power by 25% with slight performance loss. It improves the mean response time by 25.2% and improves the 90th tail latency by 18% compared with existing schemes[53].

5.18 | Load balancing

Load balancers are used to distribute incoming traffic to the network by efficiently distributing across multiple servers. By doing this reliability and high availability are maintained. It makes it easier to use in adding and removing servers in the network as per demand.

The API gateway can also handle authorization and load balancing for the microservices. The benefits of using an API gateway was reported to be extensibility and backward compatibility because of the additional level of freedom the customized API can provide. The reported downsides[14] were scalability issues and increased complexity in the architecture. Because graphql enables clients to define their customized queries, a single graphql endpoint can be used by multiple clients, thus graphql suits well to be used in the API gateway pattern[89].

To resolve challenges like load balancing, user acceptance, routing, and microservice auto-discovery. A redundant program code that covers the same functionality needs to be written at different layers of the software architecture. This often leads to mistakes, as a developer introduces unintentional errors to the repetitive code constructs. Moreover, the configuration of microservice architecture is not a trivial task and it requires intricate and redundant work to be performed. Therefore, to mitigate and speed up such a process, it could be beneficial for a developer to have a language with a concise set of concepts that are specific to the domain of REST microservice architecture development. Such a language should allow developers to have a single specification of a microservice without writing any boilerplate or redundant code. microbuilder tool frameworks provide a set of tools such as (i) zuul, for user request routing and filtering (ii) eureka, for microservice auto-discovery and registry, (iii) ribbon, for resilient and intelligent inter-process communication and load balancing, and (iv) hystrix, for isolated latency and fault tolerance among microservices. These tools are utilized to resolve some of the challenges caused by a large number of microservices per application. The remaining challenges concerning redundant coding and not so trivial configuration of microservice architectures, using these tools can be resolved by using microBuilder too[74].

6 | THREATS TO VALIDITY

In this section we acknowledge the threats to validity in the process we use in our systematic literature review. We discuss each threat and its mitigation strategy based on the guidelines presented by Kitchenham and Charters[12].

Construct validity: Construct validity concentrates on the sufficiency of the study's design to address the research questions. Many trials and discussions were carried out to define a search string and mitigate any subjectiveness in our study. The search string that resulted in the most relative number of results was selected. To mitigate threats related to the study selection strategy, the strategy was based on the software engineering systematic review guidelines presented by Kitchenham et al[97].

Internal validity: Internal validity is concerned with the conduct of the study. To mitigate internal validity threats during the study selection process, we followed the guidelines presented by Kitchenham and Charters[12] to construct the search strategy and prevent any systematic error. The procedure and its implementation were discussed by the main author and the supervisor to mitigate any subjectiveness in our study. The research questions we defined helped in selecting relevant studies. However, the chosen inclusion and exclusion criteria might have led to missing contributions that could have inspired the microservices field. While extracting videos for inclusion in the study results, we tried to include videos as much as we could using the search keywords the author might have mentioned during his presentation. We mitigated this threat by having two persons (the main author and the supervisor) going over the selection process separately. Moreover, if one person involved voted to include a reference, it was included. Therefore, any subjective bias on the inclusion or exclusion of references was lessened.

External validity: External validity is concerned with the generalizability of a study's findings[12]. Since our primary studies are obtained from a large extent of online sources, our results and observations may be only partly applicable to the broad area of practices and general disciplines of microservices. Hence to mitigate this threat we performed multiple iterations of backward snowballing to expand the search scope. Though the aim was to cover a representative body of implementation and challenges of microservices prioritization literature, the findings may not have prioritized specific challenges or implementation technologies outside of the primary studies. Moreover, there is a risk of having missed relevant industrial studies, because concepts related to those included in our search strings are differently named in such studies (e.g., a study discussing architecture of microservices may not employ the terms "challenges" but rather some synonyms). To mitigate this threat to validity, we have explicitly included all relevant synonyms in our search strings by looking at results for every iteration of chosen search string and finding enough articles to address the research questions.

Conclusion validity: Threats to conclusion validity are related to issues that affect the ability to draw the correct conclusions from the study[12]. From the reviewers' perspective, a potential threat to conclusion validity is the reliability of the data extraction categories from the selected sources. The technologies and challenges collected during the systematic literature review might be limited to the collected evidence. We are aware that the microservices challenges could be addressed by a wide range of solutions but we have answered according to the solutions we could find using the studied references. Although we did not explicitly address this threat, we claim it was mitigated due to the heterogeneity of the references we used. Additionally, to ensure validity multiple sources of data were analyzed, i.e., articles, blogs, news, and videos.

7 | RELATED WORK

Motivated by discipline of microservices, several studies have been conducted to examine the existing literature in the field. Some of the studies on microservices are discussed in this section. An interesting systematic literature review by Ghani et al.[31] focuses on the testing challenges and quality-related aspects concerning testing. They also provide some solutions for the issues with testing. We decided to have a more broad overview of all the challenges instead of focusing on just one. Although, we did find that testing is one of the most cited challenges in our studied literature.

Francesco et al.[19] motivation are on the publication trends of architecting microservices and the focus of architecting with microservices. One of their research questions is similar to our research question about the potential for industrial adoption of existing research on architecture with microservices. However, this is again focused on industrial needs whereas our study consists of a broader scope with both practitioner and academic material. Pahl and Jamshidi[1] conducted a systematic mapping on microservices and it is the classification of research directions in the field and highlights the perspectives considered by researchers.

We also analyzed two survey studies. Dragoni et al.[4] performed a survey on microservices. The survey gives an overview of software architecture, mostly providing the reader with references to the literature, and guiding the readers in the itinerary towards the advent of services and microservices. We focus on the adoption and implementation of microservices. We followed a systematic literature review method which is easier to reproduce. Ghofrani et al [98] study are similar to finding the challenges in adopting microservices but limited to survey conducted to the industrial peers. They do provide solutions from the experts to improve the aspects of the architecture. Although, it focuses on only artifacts from third parties, and not all challenges are addressed by the authors. Only the challenges faced in that company are considered by them.

8 | CONCLUSION

In this paper, we reported the results of our systematic literature review in microservices. We collected and read a total of 81 references we deemed relevant to answer our research questions. Our work can be used as groundwork and complementary to the existing literature reviews to guide researchers to open issues and challenges in microservices and offer an overview of solutions to consider.

The first research question addresses the challenges in adopting microservices. Among the collected publications, we identified 18 different challenges. The most mentioned challenges were migration (14 references), performance (11 references), scalability (10 references), and testing (10 references). The least mentioned were power management (2 references) and load balancing (2 references).

The second question addresses the technologies used in implementing microservices. We distinguished a total of 44 reported technologies which we grouped into the following categories: container, programming languages, and Other technologies, communication, framework and platform as a service. The most mentioned technology categories were Container (14 references), programming languages (13 references) and Other technologies (13 references). When we look at the ungrouped technologies, the most cited are docker (12 references), kubernetes (9 references), and REST (8 references).

We also discussed the possible solutions for each of the challenges. The solutions may vary based upon the requirement and necessity of adoption of microservices. According to the gathered solution, a necessary aspect to remember is to analyze the entire system before migrating to microservices.

Future work includes (i) adding more grey literature and valuable resources; (ii) conducting a survey with the industrial peers; and (iii) providing some broader scope to the collected materials about the aspects and architecture of microservices.

References

1. Pahl. C, Jamshidi. P. Microservices: A Systematic Mapping Study. In: INSTICC. SciTePress; 2016: 137-146

2. Larrucea X, Santamaria I, Colomo-Palacios R, Ebert C. Microservices. *IEEE Software* 2018; 35(3): 96-100. doi: 10.1109/MS.2018.2141030

3. Soldani J, Tamburri D, Heuvel WJ. The Pains and Gains of Microservices: A Systematic Grey Literature Review. *Journal of Systems and Software* 2018; 146: 215 - 232. doi: 10.1016/j.jss.2018.09.082

4. Dragoni N, Giallorenzo S, Lafuente AL, et al. *Microservices: Yesterday, Today, and Tomorrow*: 195–216; Cham: Springer International Publishing . 2017

5. Zimmermann O. Microservices tenets. *Computer Science-Research and Development* 2017; 32(3-4): 301–310.

6. Goldsmith K. Microservices at Spotify. In: ; 2015. youtu.be/7LGPeBgNFuU.

7. Alpers S, Becker C, Oberweis A, Schuster T. Microservice based tool support for business process modelling. In: IEEE. ; 2015: 71–78.

8. Meshenberg R. Microservices at Netflix Scale: Principles, Tradeoffs and Lessons Learned. Goto Conference. 2016. youtu.be/57UK46qfBLY.

9. Márquez G, Osses F, Astudillo H. An Exploratory Study of Academic Architectural Tactics and Patterns in Microservices: A systematic literature review. In: Genero M, Kalinowski M, Molina JG, et al., eds. *Proceedings of the XXI Iberoamerican Conference on Software Engineering, Bogota, Colombia, April 23-27, 2018*Curran Associates; 2018: 71–84.

10. blog UO. Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow. 2015. https://eng.uber.com/service-oriented-architecture/.

11. Dhanasekaran K. Migrating Applications from Monolithic to Microservice on AWS. 2017. https://aws.amazon.com/blogs/apn/migrating-applications-from-monolithic-to-microservice-on-aws.

12. Kitchenham B, Charters S. Guidelines for performing systematic literature reviews in software engineering. 2007.

13. Thönes J. Microservices. *IEEE software* 2015; 32(1): 116–116.

14. Taibi D, Systä K. From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining.. In: ; 2019: 153–164.

15. Danbettinger . What are Microservices. 2019. youtu.be/CdBtNQZH8a4.

16. Uros Pavlovic GA. Docker: Top 7 Benefits of Containerization. In: ; 2020. https://hentsu.com/docker-containers-top-7-benefits/.

17. Zaytsev A. Continuous integration for kubernetes based platform solution. 2018.

18. Kavungal J. Benefits and Challenges of Adopting Microservices Architecture. In: ; 2017. https://dzone.com/articles/benefits-and-challenges-of-adopting-microservices.

19. Di Francesco P, Lago P, Malavolta I. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 2019; 150: 77–97.

20. Kalske M. Transforming monolithic architecture towards microservice architecture. In: ; 2018.

21. Santos N, Silva AR. A Complexity Metric for Microservices Architecture Migration. In: IEEE. ; 2020: 169–178.

22. Lenga S. *Modernization of Monolithic Legacy Applications towards a Microservice Architecture with ExplorViz*. PhD thesis. Kiel University, 2019.

23. Ramaswamy J. Cloud Migration with a Microservice Architecture: A Coca Cola Case Study. 2017. https://www.brighttalk.com/webcast/8481/255521/cloud-migration-with-a-microservice-architecture-a-coca-cola-case-study.

24. Schaefer R. From Monolith to Microservices at Zalando. 2016. youtu.be/gEeHZwjwehs.

25. Haugeland SG. Migrating a Monolithic Architecture to a Customization-Ready Multi-tenant Microservice Architecture. Master's thesis. University Of OSLO. 2020.

26. Villamizar M, Garcés O, Ochoa L, et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications* 2017; 11(2): 233–247.

27. Eldein AIES. *A Container-based Architecture for the Design of Portable Cloud Applications*. PhD thesis. Sudan University of Science and Technology, 2019.

28. Venugopal M. Containerized Microservices architecture. *International Journal of Engineering and Computer Science* 2017; 6(11): 23199–23208.

29. Götz S. Supporting Systematic Literature Reviews in Computer Science: The Systematic Literature Review Toolkit. In: MODELS '18. Association for Computing Machinery; 2018; New York, NY, USA: 22–26

30. Garousi V, Felderer M, Mäntylä MV. The Need for Multivocal Literature Reviews in Software Engineering: Complementing Systematic Literature Reviews with Grey Literature. In: EASE '16. Association for Computing Machinery; 2016; New York, NY, USA

31. Ghani I, Wan-Kadir WM, Mustafa A, Babir M. Microservice Testing Approaches: A Systematic Literature Review. *International Journal of Integrated Engineering* 2019; 11: 65-80.

32. Attia S. Systematic Literature Review. 2020.

33. Bryzek M. Design Microservice Architectures the Right Way. QCon Plus. 2018. youtu.be/j6ow-UemzBc.

34. Toledo dSS, Martini A, Przybyszewska A, Sjøberg DI. Architectural technical debt in microservices: a case study in a large company. In: IEEE. ; 2019: 78–87.

35. Ranney M. What I Wish I Had Known Before Scaling Uber to 1000 Services. Goto Conference. 2016. youtu.be/kb-m2fasdDY.

36. Kamei FK. The Use of Grey Literature Review as Evidence for Practitioners. *SIGSOFT Softw. Eng. Notes* 2019; 44(3): 23. doi: 10.1145/3356773.3356797

37. Garousi V, Felderer M, Mäntylä MV. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 2019; 106: 101-121. doi: https://doi.org/10.1016/j.infsof.2018.09.006

38. Sarantola T. Migrating a Modern Web Application to the Cloud; Modernin Web-Sovelluksen Migraatio Pilviympäristöön. g2 pro gradu, diplomityö. 2020-08-18.

39. Wang Y, Kadiyala H, Rubin J. Promises and Challenges of Microservices: an Exploratory Study. tech. rep., 2020. Authors' copy. To appear in the Empirical Software Engineering (Springer).

40. Rosa F. *Analysis of requirements and technologies to migrate software development to the PaaS model*. PhD thesis. NOVA Information Management School, 2018.

41. Leo Z. Achieving a Reusable Reference Architecture for Microservices in Cloud Environments. bachelor's thesis. 2019. Malardalen University, School of Innovation, Design and Engineering, Vasteras, Sweden.

42. Ghayyur SAK, Razzaq A, Ullah S, Ahmed S. Matrix Clustering based Migration of System Application to Microservices Architecture. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS* 2018; 9(1): 284–296.

43. Overeem M, Jansen S. Continuous Migration of Mass Customized Applications.. In: ; 2018: 61–65.

44. Di Francesco P. Architecting microservices. In: IEEE. ; 2017: 224–229.

45. Kamimura M, Yano K, Hatano T, Matsuo A. Extracting Candidates of Microservices from Monolithic Application Code. In: IEEE. ; 2018: 571–580.

46. Selmadji A, Seriai AD, Bouziane HL, Mahamane RO, Zaragoza P, Dony C. From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach. In: IEEE. ; 2020: 157–168.

47. Thomas-Betts . How and Why Etsy Moved to an API-First Architecture. 2016. https://www.infoq.com/news/2016/08/etsy-api-first-architecture.

48. blog NO. A Microscope on Microservices. 2015. https://netflixtechblog.com/a-microscope-on-microservices-923b906103f4.

49. Ghebremicael ES. Transformation of REST API to GraphQL for OpenTOSCA. Master's thesis. University of Stuttgart. 2017.

50. Johansson G. Investigating differences in response time and error rate between a monolithic and a microservice based architecture. Master's thesis. KTH ROYAL INSTITUTE OF TECHNOLOGY. 2019.

51. Wang Z, Xia Y, Sun C, Cheng L. Research on Microservice Application Performance Monitoring Framework and Elastic Scaling Mode. In: . 1617. IOP Publishing. ; 2020: 012048.

52. Saman B. Monitoring and analysis of microservices performance. *Journal of Computer Science and Control Systems* 2017; 10(1): 19.

53. Hou X, Liu J, Li C, Guo M. Unleashing the Scalability Potential of Power-Constrained Data Center in the Microservice Era. In: ; 2019: 1–10.

54. Stuart T. Inside a SoundCloud Microservice. 2016. https://developers.soundcloud.com/blog/microservices-and-the-monolith.

55. McElhiney PR. Scalable Web Service Development with Amazon Web Services. Master's thesis. Univeristy of New Hampshire, Durham. 2018.

56. Khan M, others . Scalable invoice-based B2B payments with microservices. Master's thesis. Aalto University, School of Science. 2020.

57. Coulson NC, Sotiriadis S, Bessis N. Adaptive microservice scaling for elastic applications. *IEEE Internet of Things Journal* 2020; 7(5): 4195–4202.

58. Savchenko D. Testing microservice applications. 2019.

59. Pietrantuono R, Russo S, Guerriero A. Testing microservice architectures for operational reliability. *Software Testing, Verification and Reliability* 2020; 30(2): e1725.

60. Huttunen J, others . Micro service Testing Practices in Public Sector Software Projects. 2017.

61. Korbes E. Bringing Magic To Microservice Architecture Development. 2018. youtu.be/accEvqeUJWs.

62. blog oK. How we build microservices at Karma. 2016. https://blog.karmawifi.com/how-we-build-microservices-at-karma-71497a89bfb4.

63. Kristiani E, Yang CT, Huang CY, Wang YT, Ko PC. The implementation of a cloud-edge computing architecture using OpenStack and Kubernetes for air quality monitoring application. *Mobile Networks and Applications* 2020: 1–23.

64. Zhang H, Li S, Jia Z, Zhong C, Zhang C. Microservice architecture in reality: An industrial inquiry. In: IEEE. ; 2019: 51–60.

65. Aguiar Monteiro dL, Almeida WHC, Hazin RR, Lima dAC, Silva eSKG, Ferraz FS. A Survey on Microservice Security–Trends in Architecture, Privacy and Standardization on Cloud Computing Environments. *International Journal on Advances in Security Volume 11, Number 3 and 4, 2018* 2018.

66. Utomo P, Falahah F. Conceptual Model of a Dashboard for Monitoring Microservices. *EAI Endorsed Transactions on Cloud Systems* 2020; 6(18).

67. Bedra A. Security and Trust in a Microservices World. 2018. youtu.be/ZKswxdPcdsE.

68. Gonchar G. Secure Microservices Adoption. MicroXchg conference.. 2017. youtu.be/3rdsp4Z9gPM.

69. Tenev T. SECURITY PATTERNS FOR MICROSERVICE DATA MANAGEMENT. In: ; 2019: 575.

70. Koschel A, Astrova I, Dötterl J. Making the move to microservice architecture. In: IEEE. ; 2017: 74–79.

71. Ndungu M. ADOPTION OF THE MICROSERVICE ARCHITECTURE. 2019.

72. Gözneli B. *Identification and Evaluation of a Process for Transitioning from REST APIs to GraphQL APIs in the Context of Microservices Architecture*. PhD thesis. Technische Universität München, 2020.

73. Premchand A, Choudhry A. Architecture Simplification at Large Institutions using Micro Services. In: ; 2018: 30-35

74. Terzić B, Dimitrieski V, Kordić S, Milosavljević G, Luković I. Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures. *Enterprise Information Systems* 2018; 12(8-9): 1034–1057.

75. George F. Challenges in Implementing Microservices. 2015. youtu.be/yPf5MfOZPY0.

76. Carvalho L, Garcia A, Assunção WKG, Mello dR, Lima dMJ. Analysis of the Criteria Adopted in Industry to Extract Microservices. In: CESSER-IP '19. IEEE Press; 2019: 22–29

77. Merson P, Yoder J. Modeling Microservices with DDD. In: IEEE. ; 2020: 7–8.

78. Li Y, Wang CZ, Li Yc, Su J. Granularity Decision of Microservice Splitting in View of Maintainability and Its Innovation Effect in Government Data Sharing. *Discrete Dynamics in Nature and Society* 2020; 2020.

79. Liu Q. *Integrating Game Engines into the Mobile Cloud as Micro-services*. PhD thesis. University of Saskatchewan, 2018.

80. Speth S. Issue management for multi-project, multi-team microservice architectures. Master's thesis. University of Stuttgart. 2019.

81. Kalske M, Mäkitalo N, Mikkonen T. Challenges When Moving from Monolith to Microservice Architecture. In: ; 2017.

82. Neves JCRD. *Technical Challenges of Microservices Migration*. PhD thesis. Insititute of Porto, 2019.

83. Falatiuk H, Shirokopetleva M, Dudar Z. Investigation of Architecture and Technology Stack for e-Archive System. In: ; 2019: 229-235

84. Zrzavy W. *Strategies for IT Product Managers to Manage Microservice Systems in Enterprises*. PhD thesis. Walden University, 2020.

85. Händel L. Microservices in the context of a fast-growing company. 2020.

86. Kleehaus M, Matthes F. Challenges in Documenting Microservice-Based IT Landscape: A Survey from an Enterprise Architecture Management Perspective. In: IEEE. ; 2019: 11–20.

87. Khan A. Microservices in context: Internet of Things - Infrastructure and Architecture. Master's thesis. Linnaeus University, Faculty of Technology, Department of computer science and media technology (CM). 2017.

88. Hou X, Li C, Liu J, Zhang L, Hu Y, Guo M. ANT-man: towards agile power management in the microservice era. In: IEEE Computer Society. ; 2020: 1098–1111.

89. Touronen V, others . Microservice architecture patterns with GraphQL. 2019.

90. Bahadori K, Vardanega T. Designing and Implementing Elastically Scalable Services. Master's thesis. University of Padova, Italy. 2018.

91. Chauvel F, Solberg A. Using intrusive microservices to enable deep customization of multi-tenant SaaS. In: IEEE. ; 2018: 30–37.

92. Gonchar G. Microservices and Kafka. 2019. https://ebaytech.berlin/microservices-and-kafka-part-1-614767d27b20.

93. Boronin M. Hybrid Cloud Migration Challenges. A case study at King. Master's thesis. University of Uppsala. 2020.

94. Monteiro D, Maia PHM, Rocha LS, Mendonça NC. Building orchestrated microservice systems using declarative business processes. *Service Oriented Computing and Applications* 2020; 14(4): 243–268.

95. Vänskä MA. Automated testing for microservices. Master's thesis. Tampere University. 2019.

96. Otharson H. Reality Check: 6 Cost-Benefit Considerations When Adopting Microservices. 2019. https://thenewstack.io/reality-check-6-cost-benefit-considerations-when-adopting-microservices/.

97. Kitchenham BA, Budgen D, Brereton P. *Evidence-based software engineering and systematic reviews*. 4. CRC press . 2015.

98. Ghofrani J, Lübke D. Challenges of Microservices Architecture: A Survey on the State of the Practice.. In: ; 2018: 1–8.

How to cite this article: Prakash K, Rocha H, Valente M.T, Demeyer S, and Cleve A (2021), Systematic Literature Review on Microservices: Challenges and Technologies, *J Softw Evol Proc*, *2021;00:X–Y*.