The Journal of Systems & Software 180 (2021) 111014

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Deployment and communication patterns in microservice architectures: A systematic literature review☆,☆☆

Işıl Karabey Aksakalli [a,b], Turgay Çelik [c], Ahmet Burak Can [d,*], Bedir Tekinerdoğan [e]

[a] Hacettepe University, Graduate School of Science and Engineering, Beytepe, Ankara, Turkey
[b] Erzurum Technical University, Department of Computer Engineering, Turkey
[c] MilSOFT SoftwareTechnologies Corp, Turkey
[d] Hacettepe University, Department of Computer Engineering, Turkey
[e] Wageningen University & Research, Information Technology, The Netherlands

ARTICLE INFO

Article history:
Received 27 January 2020
Received in revised form 3 May 2021
Accepted 19 May 2021
Available online 7 June 2021

Keywords:
Microservice architectures
Microservice deployment
Communication patterns of microservices
Deployment challenges
Communication concerns
Research directions

ABSTRACT

Context: Microservice is an architectural style that separates large systems into small functional units to provide better modularity. A key challenge of microservice architecture design frequently discussed in the literature is the identification and decomposition of the service modules. Besides this, two other key challenges can be identified, including the deployment of the modules on the corresponding execution platform, and adopted communication patterns.
Objective: This study aims to identify and describe the reported deployment approaches, and the communication platforms for microservices in the current literature. Furthermore, we aim to describe the identified obstacles of these approaches as well as the corresponding research directions.
Method: A systematic literature review (SLR) is conducted using a multiphase study selection process in which we reviewed a total of 239 papers. Among these, we selected 38 of them as primary studies related to the described research questions.
Results: Based on our study, we could identify three types of deployment approaches and seven different communication patterns. Moreover, we have identified eight challenges related to the deployment and six challenges related to the communication concerns.
Conclusion: Our study shows that in addition to the identification of modules, the deployment and communication approaches are equally crucial for a successful application of the microservice architecture style. Various deployment approaches and communication patterns appear to be useful for different contexts. The identified research directions in the literature study show that still more research is needed to cope with the current challenges.

1. Introduction

Cloud computing presents opportunities for scaling applications to virtual servers effectively by allowing enterprises to dynamically adjust their computing resources. Many companies usually deploy their applications on Infrastructure as a Service (IaaS), Platform as a Service (PaaS) environments, or use Software as a Service (SaaS) applications. One of the main motivations to use cloud solutions is to scale applications on-demand easily. However, this becomes cumbersome in the case of monolithic applications that consist of large modules. Since monolithic applications have many services with different loads, scaling these applications is very difficult. Even if only a few demanding services of the applications need to be scaled, the entire service set must be scaled at the same time (Dragoni et al., 2018). This situation leads to inefficient use of resources in the cloud environment. Moreover, monolithic applications are generally developed by more than one developer. Since developers work on a single code base, a developer must ensure that none of the other services are broken when adding or changing a service (Newman, 2015). Update or change in any service affects the other services, and thus, the entire system must be tested and deployed all over again (Sakovich). Further, since each added service increases the application's complexity, it becomes more challenging to develop new versions and features (Namiot and Sneps-Sneppe, 2014). When a new version of an application is deployed, the users who are currently using the application can also not access the system until the redeployment is completed, and the entire service cluster is restarted. Also, since a monolithic

☆ Editor: Shane McIntosh.
☆☆ This publication is a part of the first author's PhD thesis.
* Corresponding author.
E-mail address: abc@hacettepe.edu.tr (A.B. Can).

https://doi.org/10.1016/j.jss.2021.111014
0164-1212/

application has a single code base, a single point of failure on the application causes the whole system to crash.

To overcome these drawbacks, software companies started migrating their monolithic products to microservices architecture and developing new applications with this approach. The microservice concept is an architectural style that composes small services to form complex software applications. Each microservice is responsible for accomplishing a task representing a low business capacity and can be developed with various programming languages by separate teams on different periods. These services can communicate with each other using messaging protocols/services like ActiveMQ (Apache), OMG DDS (omgwiki), AWS SQS (Amazon), or language-independent protocols like Representational State Transfer (REST) (Wilde and Pautasso, 2011). In recent years, large Internet companies such as Amazon, Netflix, eBay, and LinkedIn use microservice architectural patterns to deploy their large applications over the cloud in the form of small, independently testable, deployable, processable, and upgradable services. By taking advantage of this architectural style, companies can now enable more controlled and faster production, scale computing resources easily, reduce unit complexity, and create more scalable development teams (Daya et al., 2016).

Adopting the microservice architecture style, however, is also not trivial and several challenges can be identified in practice. An often reported key challenge of microservice architecture design is the decomposition of the monolithic application and herewith the identification of the microservice modules and the proper architecture decomposition (Guidi et al., 2017; Krylovskiy et al., 2015; Lin et al., 2017; Potvin et al., 2016; Sill, 2016). Besides primary studies, several systematic mapping studies have also been conducted with various research questions primarily related to the microservice decomposition problem (Alshuqayran et al., 2016; Di Francesco et al., 2019; Francesco et al., 2017; Li, 2017; Pahl and Jamshidi, 2016). Although modularity is indeed an important problem, it is not the only problem for successfully applying the microservice architecture style. We can identify two other key challenges, including the deployment of the modules on the corresponding communication platform, and the adopted communication patterns. Once the microservice modules have been identified, these need to be deployed on the corresponding platform. Both the deployment process and the properties of the platform have an impact on the overall performance of the application. Further, the microservices need to communicate and exchange data with each other. Although microservice architecture helps to reduce the unit complexity by enabling high cohesion and low coupling, it requires communication platforms and deployment mechanisms due to the distributed nature.

The overall objective of this paper is to describe the currently adopted deployment and communication approaches for microservices. We address the following research questions.

RQ1m — What are the currently adopted deployment approaches for microservices?

RQ2 — What are the currently adopted communication patterns for microservices?

RQ3 — What are the identified issues and obstacles related to the communication and deployment of microservices?

RQ4 — What are the identified research directions concerning communication and deployment of microservices?

We have conducted a systematic literature review (SLR) and applied a multiphase study selection process. This scope of the review includes the published literature from 2014 to 2019. We have reviewed 239 papers, which were discovered using a well-planned review protocol, and 38 of them were selected as primary studies related to our research questions. Our study shows that in addition to the identification of modules, the deployment and communication approaches are equally crucial for a successful application of the microservice architecture style. Various deployment approaches and communication patterns appear to be useful for different contexts. The identified research directions in the literature study show that still more research is needed to cope with the current challenges.

We believe that the results of this review will be beneficial for practitioners as well as researchers. For practitioners and researchers, this study will provide information contributing to the development of microservice architectures in terms of communication and deployment patterns. Besides, researchers will get ideas about the research areas and open issues in microservice architectures. Furthermore, the data extraction process used to characterize this study can lead to future studies in architecting microservices.

The remainder of the paper is organized as follows. Section 2 provides a short background on the microservices architecture. Section 3 describes the adopted research method. Section 4 presents the results of the SLR. Section 5 presents the discussion and the threats to validity. Section 6 describes the related work, and finally, Section 7 concludes the paper.

2. Conceptual model for microservices

Fig. 1 provides a conceptual model for microservices that includes the key concepts and the relations to support the overall understanding of deployment and communication patterns used for the microservices. The conceptual model has been derived based on a domain analysis of microservice architectures. The concepts in the model are defined as follows.

A *MicroserviceApplication* consists of (1) one or more *MicroserviceInstances* and (2) one or more *InfrastructureElements*. MicroserviceInstances can be deployed to different InfrastructureElements such as *BareMetal*, *VirtualMachine*, *Container*, and *OrchestrationPlatforms* (e.g., Kubernetes, Docker Swarm, etc.). Orchestration platforms are generally adopted to improve the resiliency and scalability of application deployment and operation in the cloud computing environment.

The microservice instances running in an application change dynamically, and instances are dynamically assigned to the network locations. Therefore, in case a client requests a service, the service discovery mechanism is activated. The fundamental mechanism of the service discovery is the *ServiceRegistry*, which is typically a repository of available service instances. *ServiceRegistry* provides a management API for registering and deregistering service instances and a query API to discover available service instances. Examples of service registry applications are Netflix Eureka (Netflix), Apache Zookeeper (Apache), and Consul (HashiCorp). Container orchestration systems such as Kubernetes (Kubernetes) and Marathon (Mesosphere) provide service registry functionality implicitly.

Microservice utilizes some functionalities (*CrossCuttingFunctionality*) such as logging, metrics collection, and health check, etc. They are cross-cutting by their nature and handled as separate services generally. In addition to increasing the modularity, extracting such side functionalities to separate services helps creating more robust systems. For example, a failure at logging service is not going to degrade the system's core functionalities if it is implemented as a separate service. Third-party external services (*ExternalSystem*) might be needed to enable communication of microservices located on different servers. Microservices communicate with external services via *ServiceCommunicationEndpoints*, which can be based on different technologies, e.g. (1) *Remote Procedure Invocation protocols* such as REST (2) *Publish/Subscribe architectures* such as OMG DDS (3) *Messaging Protocols* such as Apache Kafka (Apache).

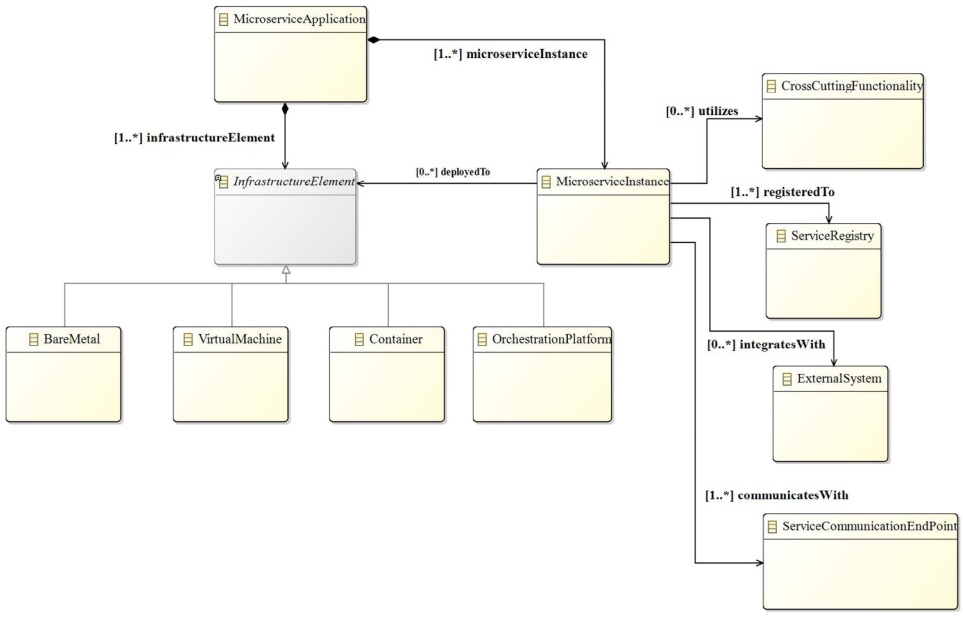


Fig. 1. Conceptual model for microservices.

3. Research method

We adopt the systematic literature review (SLR) protocol as proposed by Kitchenham et al. (2009), Kitchenham and Brereton (2013) and Kitchenham and Charters (2007). The review process is explained in the following subsections.

3.1. Review process

The workflow of the adopted SLR protocol is given in Fig. 2. First, the search scope and strategy of the SLR is defined. This stage is crucial in terms of finding studies that are appropriate for the research questions identified. After this stage, the studies to be included or excluded from the systematic review are determined. Hereby, a set of criteria is defined and the studies that match these criteria are selected as primary studies among all the studies evaluated in the search result. Then the quality checklist is defined, and these studies are passed through the quality evaluation process. After this step, a data extraction form that describes how the information is extracted from each study is created. In the final stage, the data obtained during the data extraction phase and related results are presented in the data synthesis part.

3.2. Objective and research questions

The objective of this study can be formulated as follows: What are the feasible communication and deployment patterns for microservices? In the context of this objective, the research questions are as follows:

- RQ1. What are the currently adopted deployment approaches for microservices?
- RQ2. What are the currently adopted communication patterns for microservices?
- RQ3. What are the issues and obstacles related to the communication and deployment of microservices?
- RQ4. What are the research directions concerning the communication and deployment of microservices?

The studies published between 2014 and 2019 have been indexed to assess the identified research questions. Especially, the venues where high-quality studies are published have been chosen. We use the following electronic search databases: ACM Digital Library, IEEE Xplore, Springer, ISI Web of Knowledge. Also, many studies from Google Scholar have been reviewed to pervade all studies related to microservices. These venues are shown in Table 4. In the first search phrase, journal papers, magazines, Ph.D. thesis, conference papers, white papers, and workshop papers are targeted before selecting primary studies.

In our study, both automated and manual searches are performed over the electronic search databases mentioned in Section 3.3. An automated search is performed on the search engine using the key terms associated with the research questions. Manual search is performed manually by browsing important resources related to microservices such as conferences, articles, workshops, etc. As a result of this search process, many articles have been reached, and among them, irrelevant studies are eliminated (for high precision), and the relevant ones are selected (for high recall). In SLR studies, either the recall or precision criteria can be preferred by researchers. This study focuses on high recall as a search strategy, but this method requires a lot of manual effort to deal with irrelevant articles and requires a very precise search strategy to avoid missing relevant articles. To overcome this challenge, the quasi-gold standard proposed by Zhang et al. (2011) is applied for the rigorous elimination of irrelevant studies. Therefore, before the search query is defined, a manual survey of the publications is carried out and the search strings used in these studies are analyzed.

Search strings are created for the automated search and consist of logical connectives. The keywords have been selected based on three key terms related to microservices, deployment, and communication, respectively. The adopted search strings are shown in Table 1. To look at a wide range of all studies on microservices area, general concepts have been used as much as possible during the search process phase.

3.3. Study selection process

In our study selection process, 77 papers related to microservices are selected using manual search to generate different

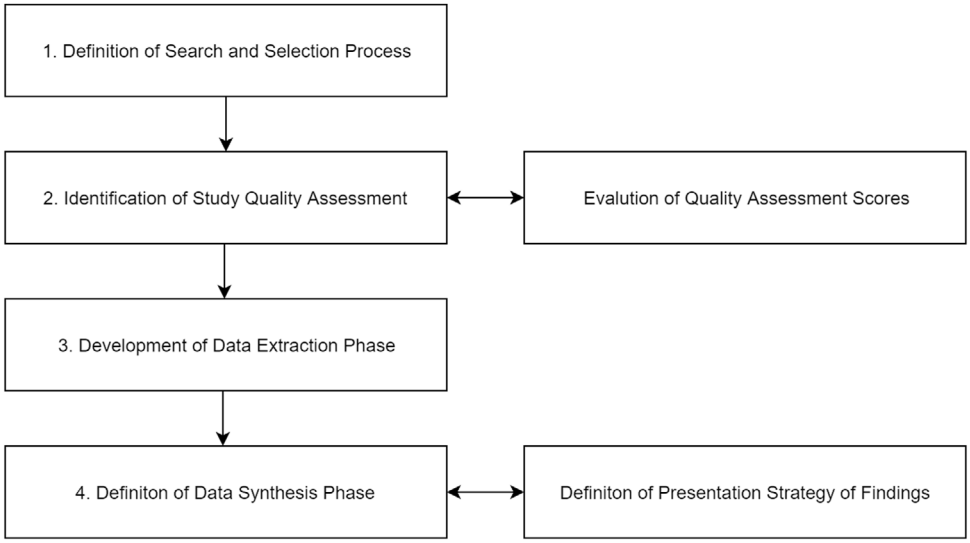


Fig. 2. The protocol for the review process.

Table 1
Search strings.

((Microservices OR Microservice-based application OR Microservice architecture) AND ((Deployment OR Allocation OR Placement) OR (Communication OR Communication patterns OR Communication protocols OR Inter-process communication)))

Table 2
Inclusion/exclusion criteria.

Criteria type (Inclusion/Exclusion)	IID	Description
Inclusion (IC)	IIC1	Title or abstract/keywords include key terms
	IIC2	The abstract of the study indicates that the study is about architecting microservices
	IIC3	The study is written in the English language
	IIC4	The study contains communication or deployment patterns specific to microservices
Exclusion (EC)	EEC1	Abstracts or titles that do not mainly discuss the provision of microservices
	EEC2	Abstracts or titles that do not propose an approach to microservices
	EEC3	Papers where the full text is not available
	EEC4	Papers that do not explicitly relate to or discuss microservices
	EEC5	Papers which are SLR papers or surveys
	EEC6	Papers that discuss only the application of microservices and do not critically reflect on the microservices communication or deployment

search strings. According to the selected keywords, 363 unique papers are identified by the automatic database search method. Then, 52 papers are eliminated since they are matched with 77 manually selected papers. 149 studies are also eliminated as irrelevant studies using the quasi-gold standard. As a result, 34 primary studies are selected by applying inclusion and exclusion criteria proposed by Kitchenham and Charters (2007) to 239 studies among the studies allocated to online databases. The overall inclusion (IC) and exclusion criteria (EC) are listed in Table 2. The detailed study selection process before and after applying inclusion and exclusion criteria is shown in Fig. 3.

At the end of the study selection process, we applied backward and forward snowballing activities when selecting the primary studies (Wohlin, 2014). The titles in the reference list of the studies considered in the backward snowballing are examined and the abstracts of the related studies are evaluated. In forward snowballing, the studies citing the selected study are obtained using Google Scholar. Four primary studies compatible with inclusion and exclusion criteria have been selected through snowballing. After the first author of this study performed the overall search process, a total of 38 primary studies among 239 papers are selected with the common decision of all authors according to identified inclusion/exclusion criteria.

3.4. Study quality assessment

After specifying the inclusion/exclusion criteria and selecting the primary studies, a quality assessment of these studies has been carried out. Among the objectives of these assessments, there are some items such as providing more detailed inclusion/exclusion criteria, investigating whether quality differences affect the results of the research questions, guiding interpretation of findings, and guiding recommendations for future research. In this study, we specified a checklist overlapping with our research questions after we examined the quality checklist provided by Kitchenham and Brereton (2013). This checklist is shown in Table 3. The first three questions in Table 3 are dedicated to calculating the score of the quality metric of reporting, the fourth, fifth, and sixth questions for calculating the rigor score. The credibility quality scores are calculated based on the evaluation of the seventh and eighth questions, and the last two questions are used for assessing the relevance of evidence scores. The quality items are numbered on a numerical scale, and the ranking and classification of the studies are ensured according to the overall quality score. Therefore an assessment is performed in a form of "yes" = 1 "somewhat" = 0.5, "no" = 0. The results of the assessment can be found in Appendix B. The results of these quality assessments help us in the process of extracting and synthesizing data.

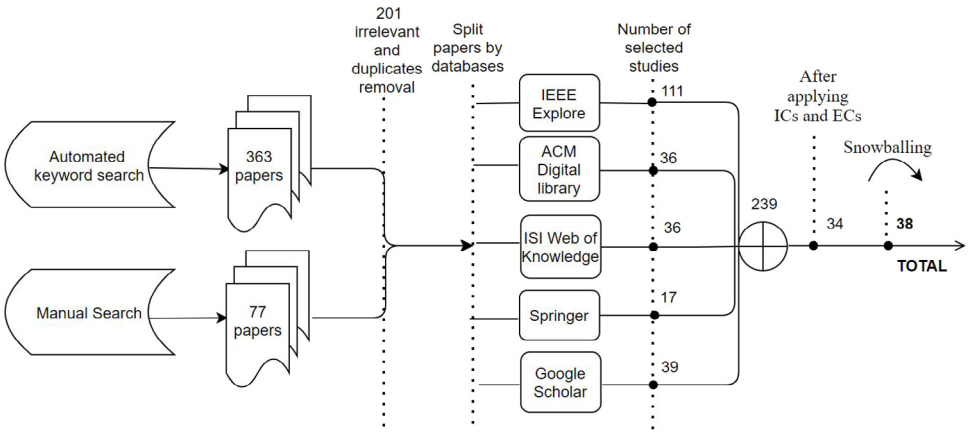


Fig. 3. The study selection process.

Table 3
Quality checklist for primary studies.

Quality metric	No	Question
Reporting	Q1	Is the objective of the study clearly stated?
	Q2	Are the scope and context of the study clearly defined?
	Q3	Is the proposed solution clearly explained and validated by an empirical study?
Rigor	Q4	Are the variables used in the study likely to be valid and reliable?
	Q5	Is the research process documented adequately?
	Q6	Are all the study questions answered?
Credibility	Q7	Are the negative findings presented?
	Q8	Are the main findings stated clearly in terms of credibility, validity, and reliability?
Relevance	Q9	Do the conclusions relate to the objective of the study?
	Q10	Does the report have implications in research and/or practice?

3.5. Data extraction

The data collection process is performed by reading the full text of the selected primary studies. To efficiently extract data from the selected primary studies and facilitate the management of these data, we specify different parameters for addressing each research question and extract data through these parameters. These information and study quality criteria are brought together in a data extraction form. Since study quality criteria are important parts of data analysis, they are kept in the same form with parameters specified for research questions. Data extraction form includes study ID, title, authors, year, publication type, a summary, and space for additional notes as a basis. Then this form is expanded by adding parameters specified to get responses to the research of the questions such as research strategy, quality attributes, industrial side of the paper, tool support, findings, constraints/limitations, implications for future research, and major conclusions.

For our SLR the related parameters for each of the research questions are given below:

Currently adopted deployment approaches (RQ1):

- Is the research proposing a novel approach for deploying microservices?
- Is the research referring to existing industrial management tools for deployment?

Currently adopted communication patterns (RQ2):

- Is the research referring to common communication technologies used in the industry?
- Is the research referring to specific communication technologies used in microservices architectures?

Issues and obstacles related to communication and deployment (RQ3):

- What are the missing features extracted from the study as a result of the applied approaches?

Research directions concerning communication and deployment (RQ4):

- What are the "constraints and limitations of the proposed solutions" extracted from research directions?

3.6. Data synthesis

The data synthesis process summarizes and synthesizes the data extracted from the primary studies to answer the research questions. In this phase, the primary objective is to classify, analyze, and understand the current research addressing microservices architecture. Additionally, qualitative and quantitative analyses are carried out on the obtained data and the findings from this analysis are interpreted and elaborated. While qualitative analysis is used to understand the basic principles of a subject better and create ideas about the proposed approach presented in the study, the quantitative analysis focuses on the amount, such as assigning numerical data to the findings and converting the obtained data into statistical data.

In our data synthesis process, it has also been investigated whether qualitative results can help to explain quantitative results. For instance, a study that examines the deployment of microservices architecture can help to interpret other deployment methods quantitatively. We intend to use tabular representations in our analyzes as much as possible to make a comparison of studies easier. Furthermore, we aim to infer the implications for future research and consequently, the existing research directions within the domain of microservices architecture thanks to quantitative analysis.

Table 4
Number of primary studies according to publication channels and occurrences.

No	Publication channel	Type	No. of studies
1	2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing	Conference	1
2	Cognizant 20-20 Insights | May 2017	Journal	1
3	Society for Modeling and Simulation International (SCS)	Conference	1
4	SAC'18, April 9–13, 2018, Pau, France	Conference	1
5	IEEE Symposium on Computers and Communications (ISCC)	Symposium	1
6	IEEE International Conference on Software Quality, Reliability and Security (Companion Volume)	Conference	1
7	IEEE Aerospace Conference	Conference	1
8	14th International Joint Conference on Computer Science and Software Engineering (JCSSE)	Conference	1
9	IEEE INFOCOM 2018 — IEEE Conference on Computer Communications	Conference	1
10	International Conference on Information and Communication Technology Convergence (ICTC)	Conference	1
11	IEEE International Conference on Web Services	Conference	1
12	10th Computing Colombian Conference (10CCC)	Conference	1
13	IEEE International Conference on Autonomic Computing	Conference	1
14	IEEE Symposium on Service-Oriented System Engineering	Symposium	1
15	International Journal of Open Information Technologies ISSN: 2307–8162 vol. 2, no. 9, 2014	Journal	1
16	The 3rd International Conference on Future Internet of Things and Cloud (FiCloud)	Conference	1
17	Addison–Wesley Professional, by Eberhard Wolf	Book	1
18	IBM Redbooks	Book	1
19	[Technical Report] Inria Sophia Antipolis.	Journal	1
20	Theory and Practice of Formal Methods, Springer, pp.194–210, 2016, Lecture Notes in Computer Science	Journal	1
21	IEEE 30th International Conference on Advanced Information Networking and Applications	Conference	1
22	United States Patent	Patent	1
23	IEEE International Conference on Industrial Technology (ICIT)	Conference	1
24	2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)	Conference	1
25	IEEE Communications Magazine, September 2018	Journal	1
26	Trans Emerging Tel Tech. 2018;29:e3445, wileyonlinelibrary.com/journal/ett	Conference	1
27	Procedia Computer Science 68 (2015) 163–172	Conference	1
28	2017 Fifth International Conference on Advanced Cloud and Big Data	Conference	1
29	International Conference on Computing, Communication, and Automation (ICCCA2017)	Conference	1
30	Journal of Network and Computer Applications 119 (2018) 97–109	Journal	1
31	9th International Conference on Knowledge and Smart Technology (KST) (Feb 2017)	Conference	1
32	2014 IEEE 17th International Conference on Computational Science and Engineering	Conference	1
33	IEEE COMMUNICATIONS LETTERS, VOL. 21, NO. 3, MARCH 2017	Conference	1
34	J Grid Computing (2018) 16:113–135	Journal	1
35	Proceedings of the Conference on Research in Adaptive and Convergent Systems	Conference	1
36	45th Euromicro Conference on Software Engineering and Advanced Applications	Conference	1
37	Springer Science+Business Media	Journal	1
38	Journal of Internet Services and Applications	Journal	1

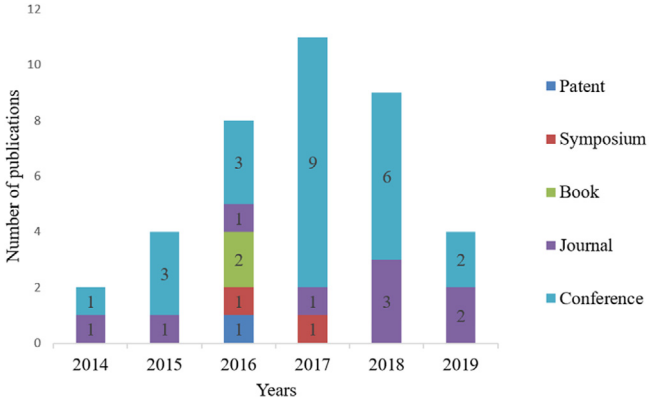


Fig. 4. Year-wise distribution of the number of primary studies.

4. Results

4.1. Overview of selected papers

The list of primary studies is shown in Appendix A. These primary studies have been chosen without looking at the publication year and venue. Fig. 4 presents the distribution of the primary studies according to publication years.

When we examine Fig. 4, it can be seen that the number of related academic studies in 2014 is very few since the microservices concept was newly emerging to the market. In the following years, the number of studies seems to have increased significantly. However, it is seen that the primary studies answering the research questions decreased after 2017. Likewise, the selected primary studies are analyzed according to the publication channel and the article type together with the number of studies that fall into the publication channels accordingly. This information is presented in Table 4. All of the selected studies have been published in different venues.

4.2. Research methods

In the selected primary studies, the adopted research methods are listed to analyze the proposed approaches and answer the identified research questions. Table 5 shows that there are five types of research methods among the selected number of 38 studies. The results of the review show that most of the studies are based on the case study, and three of them include evaluation or experiments on microservices.

4.3. Methodological quality

In a systematic literature review, research results should be rigorously examined methodologically. In this context, the quality checklist for the evaluation of the selected primary studies is given in Table 3. These studies are evaluated according to reporting, relevance, rigor, credibility qualities, and finally, the scores of all qualifications are used to calculate the overall score. According to the quality checklist in Table 3, the quality evaluation of each study is presented in the form of the clustered bar chart in Figs. 5, 6, 7, and 8. Additionally, quality assessment reports of all studies

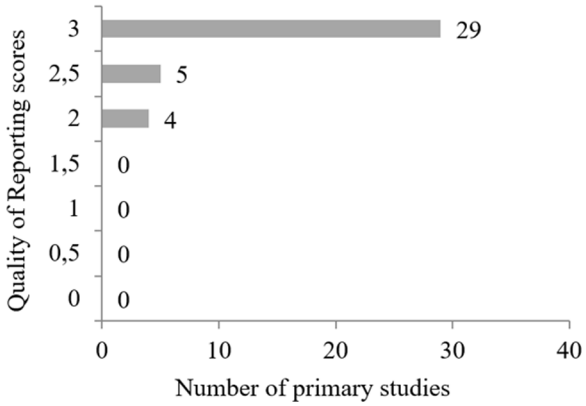


Fig. 5. Reference reporting quality of the primary studies.

are given as a table in Appendix B. The numbers that are given in the table mean yes = 1.0, somewhat = 0.5, and no = 0. Fig. 5 shows the bar chart of reporting quality results based on the first three questions. The figure indicates that most of the primary studies are good according to reporting quality. Fig. 6 shows the relevance quality score of the primary studies and focuses on the 9th and 10th assessment questions. It is determined that 68.4% of the studies (26 studies) are directly related to the research area and other studies are indirectly related to the research area with a minimum score of 1 point.

As well as these assessments, it is also evaluated on the rigor of the studies and validation and reliability of findings. The accuracy of the studies scaled between 0 and 3 is assessed with at least 1 point. Generally, most studies (21 studies) have a rigor quality of 2.5 points. Eight studies yield the maximum score in terms of rigor (see Fig. 8).

As a final criterion, the reliability of the evidence that the findings and baseline studies are mostly clear, valid, and meaningful is assessed. Fig. 7 shows the bar chart of quality scores based on the credibility of the evidence of the primary studies. As shown in the graph, only seven of the studies achieve the maximum score. After all these assessments, overall quality scores are calculated by summing up all scores, and this score represents a general quality assessment of the primary studies. Fig. 9 shows the overall quality scores in terms of four criteria: reporting, relevance, rigor, and credibility of the evidence. None of the primary studies achieve the highest score (10 points), because most of the studies do not mention negative findings of their proposed approaches. Furthermore, none of the studies answer all research questions together. Studies with a score of 9 to 10 have a high quality. In this study, 11 (%28.9) of the primary studies have high quality, 18 (%47.4) studies with a score of 7.5–8.5 have good quality, and 9 (%23.7) studies with a score of 4–7 have poor quality.

4.4. Systems investigated

In this section, results relevant to our research questions are extracted. For each research question, the data extracted from primary studies are presented under a separate subsection.

4.4.1. RQ1: What are the currently adopted deployment approaches for microservices?

While it is easy to deploy a one-piece application in the monolithic approach, the effective deployment of thousands of microservices to limited capacitated resources is not easy when the number of microservice instances is high. Thus deploying microservices to multiple resources is one of the hot research topics. In this section, we discuss current deployment approaches that we derived from selected primary studies.

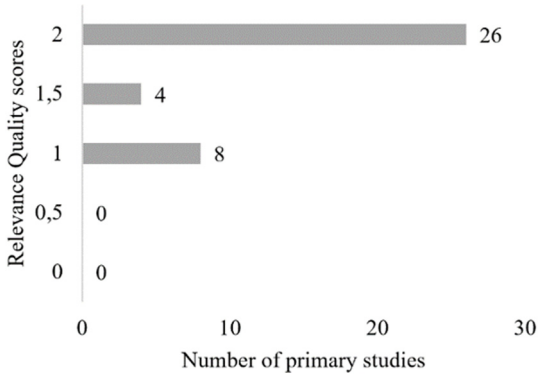


Fig. 6. Relevance quality of the primary studies.

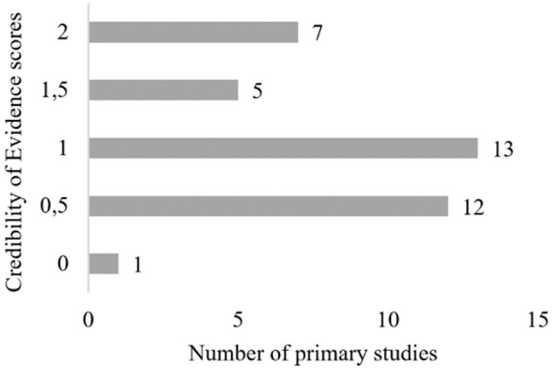


Fig. 7. The credibility of the evidence of the primary studies.

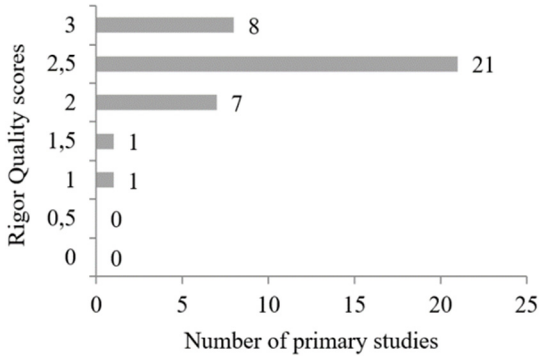


Fig. 8. Rigor quality of the primary studies.

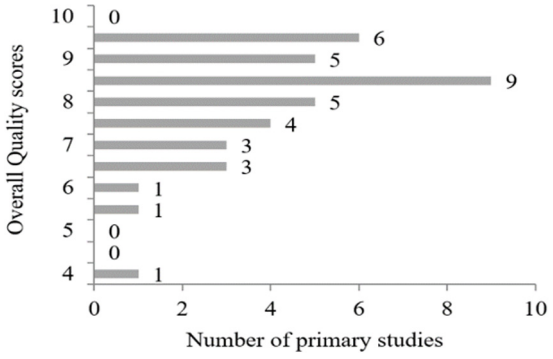


Fig. 9. Overall quality of the primary studies.

Table 5
Primary studies by research methods.

Research method	Studies	Total number of studies	Percent
Case study	P1, P3, P4, P5, P8, P9, P11, P13, P14, P16, P19, P20, P21, P22, P23, P25, P27, P28, P29, P30, P32, P33, P34, P35	24	63.16%
Survey	P2, P7, P15, P17, P18	5	13.16%
Evaluation	P6, P10, P24, P36, P37, P38	6	15.79%
Experiment	P12, P26, P31	3	7.89%
Discussion	–	0	0%

4.4.1.1. Serverless deployment. Cloud service providers (Amazon AWS Lambda, Azure, Google Cloud, etc.) can be used to deploy microservices. They provide serverless technology to users, and they can run services without any server configuration and provisioning. Serverless deployment focuses on application development without being responsible for the information technology infrastructure. Pricing is performed depending on the time taken and the amount of memory consumed by paying only for working services. This approach is handled only in paper P1 among the primary studies.

In P1, an approach is presented to model deployment costs along with computation and I/O costs of microservice-based applications deployed on public clouds. The graph-based cost model called CostHat supports services, which use new cloud environments such as AWS Lambda (Amazon) as well as microservices deployed in traditional IaaS or PaaS clouds. In particular, it is discussed how to use this model to analyze the cost impact of changes in workloads and to estimate the cost effects of new application features or re-organizations, especially when cloud users utilize the model effectively. Four types of costs are considered: computing cost (payment for CPU time to process service requests), cost of API calls (payment for each API call for a request), cost of I/O operations (payment for database or file system read/write operations, etc.), and cost of additional options. As a result, while the developer works on the code, tools continuously re-evaluate the costs in the background in real-time.

4.4.1.2. Service instance per VM. In this approach, each service is packaged as a VM image, such as the Amazon EC2 AMI. It is used by Netflix primarily to deploy its video processing services. VMs enable running each service instance completely isolated from other services. A VM has an explicitly defined CPU and memory limit, which guarantees not consuming the resources that are reserved for other services. The model has become a solution to the isolation and security problems of the multi-service approach with this feature. One of the other advantages of using VMs is the availability of cloud infrastructure. The clouds like AWS provide some features such as load balancing and automated scaling. Furthermore, the encapsulation of a service by the VM ensures that the service is protected from external threats such as sensitive data leaks, phishing, etc. VM's management software library is used in service deployment, and this deployment becomes simple and reliable. In the following, the papers that perform deployment using VMs are summarized.

Villari et al. (P5) work on deployment orchestration of distributed services in various environments such as fog computing and present a federation management component named OpenStack Federation Flow Manager (OSFFM). This component allows users to select the regions, in which microservices will be deployed through geolocation deployment restrictions. OSFFM interacts with OpenStack Federated Clouds for managing the deployment of distributed applications. Services of an edge computing system are deployed on a federal cloud network environment via a Heat Orchestration Template (HOT) service manifesto. The HOT service manifesto is analyzed by an Orchestration Broker, which can automatically extract all elements defining how microservices are deployed with Heat modules in federated clouds. Orchestration Broker analyzes specific files containing geographic location restrictions and configurations related to user-defined microservices. The most important feature of the approach is that the geographic location of federated clouds, which allow the deployment of microservices on the network edges, can be selected target federated Clouds. With this approach, the user can precisely identify the geographic area of a microservice deployed by an Orchestration Broker.

In P12, a JMeter (Rahmel, 2013) based analysis is performed on the AWS cloud to analyze monolithic and microservices architectures' performance and compare the average response time of these architectures. A monolithic enterprise application and a microservice solution that is developed with the Play web framework (Hunt, 2018) are compared as a case study. During the test, the monolithic architecture requests are sent to REST services published by the web application, and requests in the microservice architecture are sent to the REST services published by the gateway. The study represents an average response time and 90% line response time (the value of the requests below 90%) measured with JMeter (Rahmel, 2013) in the performance tests. It has been observed that both architectures meet business requirements, and using more hosts does not significantly affect the response times of microservices. On the other hand, it is mentioned that there are some challenges, such as agility, cost, and granular scaling brought by distributed systems.

Paper P19 states that there is a need to define the distinction of software components on existing resources by explaining the cost of complex applications in cloud computing and the difficulty of deploying a large number of packages. Therefore, a toolchain called Aeolus Blender is proposed, which automates the deployment and convergence of complex component-based software systems in the cloud. Blender automates real-life cloud applications entirely on OpenStack infrastructure by using a knowledge base composed of software components defined in the Mandriva Armonic tool suite. The final deployment is guaranteed to provide user requirements and software dependencies with a minimal number of virtual machines. Instead of predefined configuration recipes, Blender relies on reusable component descriptions as building blocks to synthesize a fully functional configuration, which can satisfy the system requirements. It has an easy to use Web Interface and requires only certain configuration parameters that cannot be removed from component descriptions. The web interface is an XMPP client that reveals its functions using ad hoc commands and a software pipeline that combines three independent tools called Zephyrus, Metis, and Armonic. It supports system architects and managers from the design stage to deployment with this toolchain.

The paper P28 addresses the deployment problem of microservices and states that developers of SaaS infrastructures commonly have difficulties such as scalability, availability, and increased test/development costs due to multiple tenants and large numbers of users. In this study, the need for a comprehensive and robust deployment platform enabling continuous scale is emphasized. Thus, a new platform called BigVM is proposed to isolate SaaS developers from the deployment process and close the gap between best practices and real-world applications. BigVM

provides microservice-oriented deployment kits to enable SaaS developers to create, customize, and distribute SaaS solutions in a multi-layer microservice base. BigVM focuses on the identification of highly reusable and standardized microservices, especially in multi-tenant applications. This platform can provide scaling based on resource usage and non-functional requirements, such as time constraints. It also utilizes fault tolerance and provides resource optimization. Experiments have shown that Docker containers can achieve the desired performance in terms of CPU workload and file I/O operations.

4.4.1.3. Service instance per container. In this deployment approach, each service is packaged as a container image and is isolated from the other services by running different containers. A container is a virtualization mechanism that runs at the operating system level. It consists of one or more processes running in the sandbox. Containers have their port namespace and root file systems. Their memory and CPU resources can be limited. One or more containers are started after the service is packaged as a container image. Generally, more than one container is run on each physical or virtual server. A cluster manager, such as Kubernetes or Marathon can also be used to manage containers. A cluster administrator treats servers like a pool of resources. It decides where each container should be deployed, depending on the resources available on each server and the resources required by the container. The advantages of containers are similar to those of VMs. This approach helps easy tracking of the resources consumed by each container due to isolating the service instances from each other. Besides, containers present a secure deployment environment for encapsulating services like VMs. The container management API also manages services as well as containers. In the following, the papers that perform deployment using container technologies are summarized.

In P2, an approach to design and develop microservices based on the adoption of the RESTful services is described. A tool (COSMOS) based on best practices of microservices and a container-based technology stack is proposed instead of following design checklists and development guidelines. It is emphasized that each microservice application should not only be designed for eliminating errors but should also have built-in loopback mechanisms to reduce the service call chain with minimum effect. The proposed open-source COSMOS tool offers a solution to add new deployment platforms through plug-ins. Thus, any changes made to application components can directly trigger the CI/CD (Continuous Integration/Continuous Delivery) process. Furthermore, this tool supports service packaging and creating deployment bundles for different environments.

Mittal and Risco-Martin (P3) have provided a mechanism for developing SOA-based distributed Discrete EVent Systems (DEVS) environments. The authors have proposed a new methodology to deploy DEVS Simulation Farm quickly using microservices and containerization. It is stated that the scalable microservices architecture has discrete modeling and simulation layers, and deploying both simulation and modeling layers on the cloud requires automation. The authors integrate Docker with the granular service-oriented architecture (SOA) and microservices paradigm. They improve the interoperability of models and simulations using the DEVS Modeling Language Stack (DEVSML).

In P4, a static analysis approach that supports verification of service deployment identifiers is proposed. The static analysis mechanism uses the hierarchy of deployment descriptors to verify a specific deployment identifier and correct errors automatically. The proposed approach is validated using the automation of user communities' guidelines and by analyzing 20.000 real deployment descriptors. The study focuses on how to support a series of continuously changing guides while developing descriptors and how to deal with the problems of the black-box reuse of containers. To address the first problem, starting the appropriate frame for the properties of the Docker platform is described. To address the second challenge, a technology-independent formal model is described. Furthermore, a conflict detection mechanism is defined at the formal level to address both challenges using the first-order logic that supports the detection of overlapping rules.

The paper P13 focuses on adding cross-cutting functionalities such as self-healing to microservices by using container technologies without modifying the source code of a microservice. The authors define a new abstraction layer that injects an autonomous intelligence agent in Docker containers with an approach called Gru. The autonomous agents named Gru-Agents located in nodes communicate with each other to learn about other nodes before making an autonomous decision. Gru can manage an application consisting of microservices deployed to Docker containers. After performing tests to verify the capabilities and potential of Gru, it is observed that Gru can scale containers while keeping the response time under a certain threshold value.

In P14, a Platform as a Service (PaaS) solution, Cloudware, which provides a microservices-based container solution is proposed. Without any modification on this platform, it is possible to deploy traditional software that directly provides services to users via a browser. Traditional software is connected to the OS and libraries, but with the expansion of the cloud system, the software needs to be developed and operated adaptable with the cloud, and also, it must be open to changes in the cloud. It is inferred from the paper that deployment in the Cloudware platform is identical to the deployment of microservices using container technology.

Gabrielli et al. (P20) mentioned that microservices are suitable for developing distributed systems on the cloud, but due to the dynamic nature of these services, appropriate methods for automatic deployment are required. They have proposed a tool named JRO, which is written in Jolie language for the automated and optimized deployment of microservices. JRO consists of three components: Zephyrus, Jolie Enterprise (JE), and Jolie Reconfiguration Coordinator (JRC). Starting with a partial and abstract definition of the target application, Zephyrus generates all details of the architecture, showing how many components are required and how they are deployed to virtual machines and how to bind each other. Jolie Enterprise is a distributed framework for managing and deploying microservices written in the Jolie language. JRC is a tool to provide a desired configuration and content for a deployment provided by JE, which interacts with Zephyrus to produce an optimized deployment plan.

Ciuffoletti (P27) provides a Docker-based solution for automated microservice deployment with the help of monitoring infrastructure. The study defines all steps starting from the design of an on-demand microservice-based monitoring service to the deployment. The monitoring service deploys probes in a model that defines the infrastructure. All steps are shown in a proof of concept application and an environment, where the results can be reproduced and expanded using the resources in the Docker storage. The reference architecture used for the monitoring configuration includes two applications, the sensor, and the collector, which are located in separate containers. One of the disadvantages of the proposed sensor/collector scheme is that a single collector cannot feed more than one sensor, because a collector needs an OCCI (Open Cloud Computing Interface) connection. In other words, the output of a collector cannot be broadcasted to more than one sensor. The second drawback is that it is not possible to manage asynchronous alarms because the scheme is designed to be simple to deal with periodic measurements.

Singh and Peddoju (P29) analyze the challenges of the continuous integration and deployment of microservices. They have proposed an automated system that helps to deploy and continuously integrate microservices to overcome these challenges.

The proposed microservice architecture is deployed on Docker containers and tested using a social networking application. The results are compared with the monolithic approach in terms of parameters such as response time, throughput, and deployment time. For microservice deployment experiments, HaProxy, Consul, Jenkins, GIT Repository, and Docker Registry tools are used together with design components such as API Proxy, Configuration Storage, Build Server, Source Repository, and Container Repository. As a result of the experiments, it has been seen that the application developed using the microservice approach and the deployment made with the proposed design reduce the time and effort for deployment and continuous integration of microservices.

In P30, the authors aim to satisfy service delay requirements for applications, while minimizing application costs as well as deployment cost. A suboptimal algorithm and a communication-efficient framework (ADMD) are recommended to solve the container deployment and task assignment problem. Since the proposed algorithm executes in a distributed and incremental form, it is claimed that the algorithm can be scaled to vast physical resources and various applications under the framework. Application requests in the framework of ADMD are processed as microservices on Execution Containers, and Microservices Controller (MSC) separates and manages resources for applications. The authors compare the existing three strategies in Docker Swarm using real traces in Google Cluster and verify their solutions' efficiency. The ADMD problem is divided into sub-problems to reduce the computational complexity. Each ADMD part tries to minimize the cost of distribution for a single application that is subject to the restrictions on the existing resources. Based on the result of experiments obtained from the ADMD, the authors claim that they can reduce the cost and improve flexibility compared to existing strategies.

In P31, it is mentioned that Docker technology is becoming popular today both as an operating system-level virtualization and application delivery platform. However, it is stated that the scheduling algorithm that comes with the SwarmKit, the orchestration toolkit of Docker, is not performing well when sources are not uniform. Additionally, it is mentioned that the timer's optimization may be possible with metaheuristic use, such as the Ant Colony Optimization (ACO) method. To implement a new scheduling mechanism for Docker, an ACO-based algorithm is proposed that deploys application containers over Docker hosts. This algorithm tries to balance resource utilization and improve application performance. As a result of the experiments, it is observed that the ACO performs about 15% better than the greedy algorithm on the same host configuration. Since cloud customers desire higher quality services, cloud providers are interested in resource utilization in a wide variety of resource sharing environments. Thus, P32 recommends a new resource scheduling approach for container virtualization to reduce the response time of clients' operations and to establish providers' resource usage rate. This scheduling approach decides the placement to select the most appropriate physical server to deploy the containers. To deploy containers to physical machines optimally, Stable Matching Theory (Boyle and Echenique, 2009; Gale and Shapley, 2013) is implemented, and an algorithm is proposed to solve the resource scheduling problem at the container level. Simulations show that the proposed approach increases system performance in terms of response time, CPU, memory, and disk utilization by improving resource usage.

In P33, the Docker container-based resource allocation framework is designed to reduce the cost of deployment in data centers by optimizing the resource allocation. Additionally, this framework aims to support automatic scaling while the workload of the cloud application changes. The developed AODC (Application Oriented Docker Container) resource allocation problem is modeled by considering various application requirements and resources available in cloud centers and Docker features. The proposed AODC-based framework includes a new scheduler along with two types of containers named Pallet Containers (PCs) and Execution Containers (ECs) executing tasks and decoupling resource management for each application. In contrast to the Hypervisor-based VM deployment, in this system, the number of ECs and the ECs' demands are dynamically identified not only by the workload of the application but also by the resources available in a data center. When deploying an application, resources are allocated to the application as container sets deployed over multiple physical machines (PMs). When activating the application, the scheduler creates a PC and assigns resources depending on the available resources and application requirements. The authors design a distributed AODC algorithm called by PCs for finding the appropriate PMs to reduce the deployment cost and make resource allocation decisions. The proposed method is compared with the Hypervisor-based VM placement and is proved to be more efficient in terms of acceptance rate and total cost.

In the paper P34, it is mentioned that container resource allocation is a critical factor for cloud providers as it affects system performance and resource consumption. Therefore, the authors aim to optimize container allocation and elasticity management and propose a new genetic algorithm approach using the Non-dominated Sorting Genetic Algorithm-II. In the first optimization target, the threshold distance metric is calculated and minimized to optimize the allocation of containers into physical resources. This calculation is based on the difference between the threshold value of a microservice and the container's resource consumption. In the second optimization target (Balanced Cluster use), it is recommended to use the standard deviation of the resource usage on physical nodes to evaluate cluster balance. In the third target (System failure), the reliability of systems is measured by the failure rate of applications. As the final optimization target (Total network distance functions), network distance values between the microservices are calculated by using the average network distance between all consumer pairs and provider containers. As a result of the experiments, it is stated that this technique is an appropriate solution to handle container allocation and elasticity problems by obtaining better objective values from the container management policy implemented in Kubernetes. In the paper P36, Kubernetes is preferred as an orchestration platform in order to realize fault-tolerant and elastic deployment of ALIDA platform components proposed for BDA (big data analysis) applications.

In P37, an automated SLA-aware microservice deployment framework named SmartVM is presented. The architecture of SmartVM has a container scheduling layer, and Scheduler is responsible for running containers on multiple processors. The authors use Docker-Swarm as a Scheduler in this framework.

In the paper P38, an adaptation mechanism called REMaP is proposed for managing the automatic deployment of microservices. The proposed mechanism can automatically change the service placement at runtime using the relationships (communication costs) and resource usage histories of services with each other. The authors implement the approach for the dynamic allocation of microservices by extending Kubernetes's scheduling framework. They carry out an open-source microservice-based application named Sock-Shop for empirical evaluation. Firstly, the microservices are deployed using the default Kubernetes scheduling strategy. Then, the placement of microservices at runtime are optimized using the proposed ReMaP mechanism. The authors evaluate two deployment strategies named "fully-distributed (1–1)" and "Partially distributed (N-1)" and microservices are deployed by Kubernetes in both of the strategies. The experimental results show that ReMaP uses up to 80% fewer hosts by performing replacement of microservices at runtime.

According to our review, three major deployment approaches have been identified. The deployment approaches handled in the related studies, and the comparison of the studies in terms of proposed methods, deployment environments, objectives, and industrial vs. open source are given in Table 6. In the selected studies, one service instance per container pattern appears to be the most commonly used deployment approach. Since this approach has more advantages than the other approaches, it is popular in both industrial and academic fields. The Serverless approach is relatively new compared to the other approaches, developers have almost no control over the infrastructure, and has not been widely adopted yet. Therefore, this approach has been the least used deployment pattern in primary studies. In the Service instance per VM approach, efficient resource usage is limited, and each service instance might have the overhead of VM licensing costs, including the operating system costs. In a general IaaS, a VM cannot be utilized efficiently because the sizes of the VMs are fixed. Additionally, the IaaS charges for virtual machines, regardless of whether virtual machines are busy or idle, which increases the operational costs. Another disadvantage of the approach is that deploying a new version of a service is often slow, as they are slow to render due to the size of VM images. Additionally, VMs are typically slow to restart due to their size when the service capacity needs to expand. Unlike VMs, containers are lightweight technologies, and creating and launching container images are often very fast. Furthermore, containers provide isolation at the operating system level, and more than one container can use a single operating system. This reduces the overhead of licensing costs compared to the VM. Since VMs take up more space than containers, the same physical server can hold more containers than VMs. Therefore, containerization is often preferred over the other approaches.

When the comparison of the studies according to the characteristics given in Table 6 is examined, resource allocation or resource scheduling algorithms are proposed in 27.3% of 22 studies (P30, P31, P32, P33, P34, P38) among the primary studies. The main aim of these studies to ensure effective resource usage during deployment, to decrease the deployment cost, and to increase performance. 18.18% of the studies (P13, P19, P20, and P29) adopt the automated deployment approach and aim to increase performance as a common objective. Besides, it is seen that 54.5% (P3, P4, P13, P27, P28, P29, P30, P31, P32, P33, P37, P38) of the studies that adopt container technology as the deployment environment use Docker-based technologies. In this context, it can be inferred that Docker is the most popular container technology among these primary studies.

4.4.2. RQ2: What are the currently adopted communication patterns for microservices?

For rigorous communication in a monolithic architecture, determining the relationships between database tables and mapping this database to an object model is sufficient. However, in microservice architectures, as each service has a separate database, they need to communicate with each other using appropriate communication patterns. In this section, we aim to identify these communication patterns for microservice architectures in the literature. Based on the results of the data extraction process, seven different communication patterns have been identified, and these patterns are explained below, along with a summary of the primary studies.

Synchronous communication: In synchronous communication mechanisms such as HTTP-based REST or request/response-based Apache Thrift, the client waits for a response from the service to proceed. The synchronous communication pattern is mentioned in studies P7, P1, P16, P17, P21, P23, P26. Additionally, the paper P7 mentions a one-to-one interaction type using synchronous communication called request/response. The client waits for the response after sending a request. If a response does not return to the client in a predefined timeout duration, the client receives a notification about the failure.

Publish/subscribe communication: This communication pattern is based on event-driven architecture. It provides instant event notifications for microservice-based applications. All events published on a topic are broadcast to subscribers. Studies P7, P8, P15, P16, P24, P26 use this type of communication pattern as an inter-service communication model.

Combination of HTTP and Message queue: To communicate two microservices that call each other, a hybrid model can be used by using both synchronous and asynchronous communication patterns. Firstly, the data exchange is performed using synchronous communication. If one of the services offers web service functionality, another service can send data requests to this service over HTTP and take a response containing the requested data. Synchronous communication pattern ensures that the desired response is transmitted instantly, but requires the other service to be available to use a connected service. If the requested service is not available, the user who sent the request should be informed. In this case, an asynchronous communication pattern is preferred alternatively. The incoming request can be placed in the message queue, so a hybrid approach is presented to the user. The P9 study introduces this approach as a solution to load balancing problems.

Communication using message-oriented middleware (RabbitMQ, ActiveMQ, etc.): The papers P10, P25, and P35 propose microservices communication methods using a message-oriented middleware. The paper P10 compares the communication using RabbitMQ and RESTful API in a microservices web application. In a message-oriented middleware, an API Gateway forwards incoming requests from a user to the related services by collecting the addresses of all services using RESTful API. When API Gateway receives the request, it checks the method and URL primarily then transmits the request to the related service through RESTful API. The service, which receives the request, checks the method and URL in the same way, and then it handles this request by the defined REST API. In this type of communication, service instances do not have to communicate with each other, they can just transmit their responses to the gateway. However, API Gateway causes a single point of failure when only one gateway is used in the system. To avoid this, it is recommended to replicate the API Gateway instance. In the other communication type, RabbitMQ is used instead of API Gateway. RabbitMQ includes the Exchange Function module, and this module publishes a message to the indicated queue. After a message is received by the Exchange Function module, the message is forwarded to the corresponding queue. A service module that polls the queue receives the next message and processes it.

Asynchronous communication: In asynchronous communication, the client sends the request via an asynchronous media such as a messaging queue and cannot block the communication while waiting for the response. The response from the service is later delivered to the client. While message queues such as RabbitMQ (RabbitMQ), Apache Kafka (Apache), and Apache ActiveMQ (Apache) are popular open-source messaging systems that are widely adopted in the industry to implement asynchronous communication, they are not the only alternative. The paper P7 discusses asynchronous communication patterns called request/asynchronous and publish/asynchronous. In the request /asynchronous pattern, the client sends a request to a service, and the service gives a response asynchronously while the responses are created by a client that publishes a request and then waits a certain time for the response of the related services in publish/asynchronous pattern. Along with P7, P18, and P26

Table 6
Comparison of the related studies according to deployment approaches.

Paper no.	Deployment approach[a]	Proposed method	Deployment environment	Objectives	Industrial case/Open source
P1	S	CostHat: an approach to model deployment cost	AWS	Minimizing deployment cost	Open source
P2	C	COSMOS: an open-standards-based solution	Apache Camel, Spring REST/ JAX-RS	Microservices design and delivery	Open source
P3	C	DEVSML 3.0: rapid deployment of devs farm	Docker	Performance	Open source
P4	C	Static analysis mechanism and automated rewriting approach	Docker	Security	Open source
P5	V	OSFFM (OpenStack Federation Flow Manager)	Edge computing environments	Orchestration, monitoring	Industrial
P12	V	Development of a case study to evaluate monolithic vs. microservice architecture	AWS	Infrastructure cost, performance	Industrial
P13	C	Gru: an approach to add autonomic capabilities	Docker	Performance	Industrial
P14	C	New Cloudware PaaS platform	Cloudware PaaS	Scalability, auto-deployment, disaster recovery, elastic configuration	Industrial
P19	V	Aeolus Blender: a toolchain that automates the deployment	OpenStack Cloud	Fault tolerance, performance	Open source
P20	C	JRO (Jolie Redeployment Optimizer)	AWS	Managing deployment cost, performance, resource consumption	Open source
P23	C	A modular and scalable architecture for IIoT[b]	Cloud and end-device	Scalability, availability, fault tolerance	Industrial
P27	C	Open Cloud Computing Interface	Docker	Monitoring, management	Industrial
P28	C	BigVM: Multi-layer-microservice-based platform	Docker	Performance	Industrial
P29	C	Automated system for deployment and continuous integration	Docker Swarm	Performance, scalability	Industrial
P30	C	ADMD[c] framework and EPTA[d] algorithm	Solaris 10, Linux Vserver, Docker	Managing deployment cost, performance	Industrial
P31	C	ACO-based algorithm	Docker	Resource usage, performance	Industrial
P32	C	A stable matching resource scheduling approach	Docker	Performance, resource scheduling	Industrial
P33	C	AODC[e] resource allocation algorithm	Docker	Resource usage, performance	Industrial
P34	C	Genetic algorithm	Kubernetes	Reliability, performance, resource usage	Industrial
P36	C	ALIDA: a microservice-based platform for composition, deployment and execution of big data analytics (BDA) apps	Kubernetes	Resource usage, performance	Industrial
P37	C	SmartVM: Microservice-centric framework	Docker Swarm	Deployment cost, resource usage	Industrial
P38	C	REMaP (RuntimE Microservices Placement)	Kubernetes, Docker Swarm	Resource usage, performance	Industrial

[a]Serverless: S, Service instance per VM: V, Service instance per container: C.
[b]Industrial Internet of things.
[c]Application Deployment using Microservice and Docker container.
[d]EC Placement and Task Assignment algorithm.
[e]Application Oriented Docker Container.

also discuss asynchronous communication using message brokers. A message broker is an asynchronous communication media, where a message producer service sends messages to a queue and consumers receive messages from the queue asynchronously. This type of communication separates message generators from message consumers and an intermediate message broker stores messages until consumers process them. Thus, producer and consumer microservices are completely unaware of each other and there is an asynchronous communication between them.

Point-to-point communication: In this communication model, message routing logic stays on each endpoint, and each service instance can communicate directly with other services using APIs

such as REST. However, point-to-point based systems' performance significantly degrades as the number of requests and the complexity of services increase. Furthermore, the development, operation, and maintenance of point-to-point communication models get exponentially harder when the number of services increases. The paper P26 proposes point-to-point communication as an alternative to synchronous and asynchronous communication models for microservice-based applications.

Communication using binary protocols (e.g., Apache Thrift): In a microservice architecture, the number of microservices that reveal more than one API can grow rapidly. In this case, it is necessary to maintain a well-defined interface for each call. In this case, the binary protocols become more efficient than the other communication protocols. Interface Definition Language IDL, such as Thrift and Protobuf obliges service owners to publish interface definitions so that seamless communication between the services developed in many different languages can take place. Papers P7 and P17 mention the usage of Apache Thrift as an alternative communication protocol.

The summarization of the papers, which discuss these communication patterns, is in the following:

Bakshi (P7) discusses different aspects of microservices architecture, including potential use cases for the aerospace industry. Besides, the research is focused on containerization, service communication, and related architectural components among technical implementations of microservices. The author mentioned that the Inter-Process Communication (IPC) mechanism is widely used in microservices communication. The first architectural decision is selecting a synchronous or asynchronous communication model when an IPC mechanism is selected for service.

Study P8 focuses on the publish/subscribe communication model for microservices. In their scenario, the reservation service subscribes to events that come from a client service using a publish/subscribe channel. When a client's data changes in the system or new clients are created, the client publishes service messages. Then, the reservation service gets events, and it behaves according to these events. The authors indicate that the specification gap occurs in designing microservices and inter-service communication, which can be decreased or eliminated using model-based approaches.

In study P9, it is mentioned that existing load balancing strategies used in microservices deployment increase networking overhead significantly. The paper handles the microservice communication techniques in two categories named HTTP and message communication. Since HTTP communication needs to know the exact location of the microservices instances for sending requests, it relies on Service Registry such as Zookeeper to explore API gateway and instances. Besides, HTTP communication does not support asynchronous or one-to-many models. Compared to this communication technique, message queues support all communication patterns except synchronous mode. A hybrid technique can combine HTTP and message queue as a solution for load balancing problems. However, this technique brings additional functional complexity, and thus, the authors try to find a better solution for load balancing problems. They propose Chain-driven Load Balancing Algorithm (COLBA), which only relies on the message queue. It balances the load based on microservice requirements of the chains to minimize response time. Using a convex optimization method with rounding, a solution is provided to the NP-hard problem. The results of the simulation experiments show that the overall average response time decreased by 13% compared to the current load balancing strategies.

Hong et al. (P10) compare RabbitMQ and RESTful API usage as message-oriented middleware in different use cases and with various users. The purpose of the study is to select the appropriate technique according to the requirements of service providers by understanding the advantages of these two communication techniques. According to the results from a microservices application deployed in different communication modules, it is stated that RabbitMQ is more stable than RESTful API as a message-oriented middleware when a large number of users simultaneously send requests to the web application.

In study P11, a tool named Pipekit, which is similar to Docker Compose is proposed for presenting simple container orchestration language. The Pipekit tool contains instructions to define when a service is ready. This proposed tool enables a communication layer between services with a shared file system mounted to each service. A file system provides a simple interface for writing and reading dynamic data. This data is generated by different services at deployment time. The shared storage model greatly simplifies the Pipekit synchronization process, which uses semaphores at the file system level.

Namiot and Sneps-Sneppe (P15) present application patterns of microservices with a high-level evaluation of the microservices architecture and specifically focus on microservices communication patterns. The first communication pattern is that the application uses services with direct calls. However, the biggest problem is the delay potential for remote calls. To decrease the number of remote calls, API Gateway is used to serve as a bridge between applications and services. This model is more traditional for IoT applications, and also hides some limitations for legacy devices. Besides, it is mentioned that the use of service buses is the third pattern. This pattern is appropriate for most services due to the asynchronous nature of IoT applications. Data reading is asynchronous for most of the services. Therefore, the service bus allows an application to send a request and read the response later. Furthermore, service bus deployment can use load balancing and clustering to increase scalability by deploying the workload between nodes. As another alternative, the Publish/Subscribe model is often used in IoT applications. With this model, new components can be added without making any changes to the existing components. Any new functionality can be deployed step by step, and this process does not affect already deployed components.

In study P16, a Smart City IoT platform is designed using microservices architecture. The purpose of the project is to build a service platform and to create energy efficiency applications for different stakeholders at the city level. The platform contains heterogeneous sensor technologies (middleware services) and district information models used by smart city applications. The middleware mentioned in the paper stores sensor data by providing historical data-store service and realizes service discovery by providing a message broker for Publish/Subscribe communication of service catalog and service data. Message broker uses Message Queue Telemetry Transport (MQTT) protocol and while the other services use the HTTP request/response model. The authors claim that the Publish/Subscribe communication model is an excellent fit for IoT-based applications. They utilize the IoT Data Gateway, which provides a high-level API using an application gateway pattern for different kinds of consumers. This gateway offers an appropriate interface to IoT data desired by clients, which forwards client requests to platform services and returns the requests to the related consumers as a suitable form. Additionally, based on the generated load and needs of different consumers, API Gateway functionality can be a single service using HTTP content negotiation to distinguish independent services or consumers.

In the book chapter of P17, it is mentioned that REST (REpresentational State Transfer) is one of the ways to provide communication between microservices. In REST technology, clients only need to know entry points to explore all systems. Therefore, the services are transported for clients transparently in a REST-based

architecture. The clients can receive new links without relying on any central coordination. HATEOAS (Hypermedia as the Engine of Application State) is an important component of REST, which uses HTTP and provides relationships between resources modeled by links. Therefore, the client has to know the entry point; it can continue navigating at any time and can find all data step by step. The authors mention that HATEOAS is suitable for microservices since communication is provided over links, and it does not require central coordination. As another communication method, SOAP uses POST messages to transfer data to the server. The last communication option mentioned in the paper is the Apache Thrift. Thrift uses protocol buffers to forward requests from one process to other processes through a network. Since Thrift has a flexible interface, different client and server technologies can communicate with each other.

In the book chapter of P18, it is emphasized that communication among microservices is important in terms of providing value to the clients. In this chapter, synchronous and asynchronous microservice communication types are mentioned. As explained in the previous studies, a response should be given to the client immediately in synchronous communication. Asynchronous communication messages do not require any immediate response. While an asynchronous form of messaging is more prevalent, synchronous API is preferred in case one service is triggering other services. Asynchronous messaging is used for segregated coordination. An asynchronous event is used only if the creator of the event does not need a response. Besides, the proposed platform guides the developer when deciding about asynchronous or synchronous communication. On the platform, when a user sends an update room request to map service, it waits for a confirmation, which indicates that the request must be synchronous. This synchronous request notices a response to the user about whether the update is successful or the update is in a queue. After the update is performed, the room has been updated in the database. The map service informs all other services about this update. Since this update does not require a response, it is sent using an asynchronous event.

Safina et al. (P21) aim to develop the Jolie programming language based on the microservices paradigm, which lacks some structures to deal with the message types and data manipulation available in service-oriented computing. The communication of processes in the Jolie language is carried out by two patterns, the one-way (the end-point only receives the message), and the request–response method (the end-point receives the message and sends the response back to the caller). In the Jolie language, one-way is used to get a message for an operator located in variable *x*. The R-Response (Request/Response) receives the message for the operator in variable x and executes a process, and sends the response to the caller with variable *y*. Notification and Solicit-Response messages are used as outputs corresponding to One-way and R-Response messages. Additionally, the output statements include OPort, which defines an output port of the end-point.

In the study P22, a new MicroServices Communication Platform named MSCP is proposed, and this platform is used in service-based networking solutions to facilitate communication. MSCP is implemented as an alternative to peer to peer communication network solutions. It includes microservices configurations and media communication microservices for numerous entities. The inclusion of an entity to the platform includes entity configuration settings for the use of the platform system. According to the entity configuration, a request of microservice is processed, where a microservice request means the use of at least one microservice on behalf of the entity.

Rufino et al. (P23) provide modular, distributed deployment with Docker technology and facilitate management by presenting modular and scalable architecture. In this architecture, containerization is used to isolate services from each other, and different Docker tools allow containerized services to scale. Modularity and decentralization are provided by dividing the applications into small microservices and deploying these services to various components of the system. Furthermore, interoperability between devices and machines can be abstracted using a database, which is deployed in the gateway for REST-based protocols and/or the communication mediator. It is stated that the proposed architecture can be extended to resource-restricted devices by reducing the load of the system. To achieve this, it is stated that docker images should be reduced, and different protocols should be used for inter-service communication. HTTP as REST is used for interacting with the database of microservices. The authors attempt to implement CoAP (Constrained Application Protocol) to improve communication.

The study P24 states that the interest in event-driven communication has increased in recent years, and there is a paradigm shift from traditional client/server thinking to information-centered networks focusing on named content. The study presents a multi-protocol broker solution that makes it possible to interoperate a wide range of Publish/Subscribe industrial standards (AMQP, STOMP, MQTT) by bridging. In the publish/subscribe messaging pattern, which is frequently used at IoT solutions, lightweight Publish/Subscribe protocols such as MQTT are used for sharing information among the sensors and gateways. Although MQTT is probably the most widely adopted application in the IoT domain, Extensible Messaging Presence Protocol (XMPP) and Advanced Message Queuing Protocol (AMQP) are also used. Besides, it is indicated that multiple protocols can be used for complex IoT applications. For instance, a single sensor can use MQTT, while others are using XMPP. A gateway can use a more productive protocol such as AMQP to communicate with backend systems. The OKSE 2.0 broker system developed in the study is designed to translate messages between Publish/Subscribe clients using different protocols. The system includes a server for each supported protocol, a database for authentication, and a broker core that handles message distribution between protocol servers. Each protocol service transmits messages to the internal broker core, which serves as an internal distribution service among the protocol servers. Broker core delivers the message to related protocol services for distribution to subscriber clients.

Alam et al. (P25) propose an architecture for microservices that provides middleware for endpoint discovery. The gateways in the middleware provide local communication endpoints to reduce the delay for local services. The gateways also push the state of connected edge devices to the cloud, convert the received data, perform simple analyzes, and virtualize the network functions to ease the development of new services. Another provided component is the message bus used between Docker and Mediation service. The Mediation is a multi-tiered service that supports scaling under heavy message loads. The microservices are isolated from each other, and each of them may have a particular communication protocol with the middleware.

The objective of the study P26 is to provide cooperation between industry and academia and discuss different microservice deployment, discovery, and communication options to create complete service chains for service providers. In this paper, synchronous and asynchronous modes are explained, and three microservices communication alternatives (point-to-point communication, communication over API Gateway, Message Broker Style) are briefly described.

Smid et al. (P35) propose an automated data streaming system to address the data synchronization between databases belonging to monolithic and microservice architectures. To handle database changes, message queues such as Apache Kafka (Apache) and

a data change capture platform (Debezium) are used. Besides, they address the performance improvement of communication between microservice instances by using a message broker and distributed cache. The authors experienced that the proposed architecture is not useful for complex relational databases since Debezium can only fetch data changes from a particular table. On the other hand, it is observed that the usage of a message broker is efficient in communication between microservices.

As a result of our analysis of primary studies, we have identified 7 specific types of communication patterns for microservices. Table 7 lists the identified communication patterns.

When the communication patterns identified in the studies are examined, it is seen that *synchronous* and *publish/subscribe communication* are the most preferred patterns. However, it is not possible to determine the most popular communication pattern, like deployment approaches. The selected communication pattern can change according to the need, and thus a communication pattern that is suitable for the architecture must be determined. While it is easy to implement *synchronous communicatio*n (e.g. HTTP-based REST, request/response-based, API Gateway, point to point), it creates a strong coupling between services. Thus, this type of communication may face timeouts since the communicating services must always be up and available to finish transactions. To overcome this problem, more than one instance of critical services can be deployed behind a load balancer, which directs each incoming request to one of the available services. Synchronous communication can be a good choice for idempotent operations, such as data fetching. On the other hand, *asynchronous communication* (Message Broker, message-oriented middleware RabbitMQ, RESTful API, etc.) makes services loosely coupled, and it is more appropriate for using state-changing operations. According to our investigation in the selected studies, asynchronous communication via one-to-one transactions is called request/asynchronous. Also, publish/subscribe and publish/asynchronous patterns are identified as one-to-many interactions. In the publish/*subscribe communication pattern*, any notification messages are published to a publish/subscribe channel and the messages are received by all of the channel subscribers. Publish/asynchronous responses are performed by a client that publishes a request and then waits a certain time to respond to the related services. Furthermore, to use the advantages of both synchronous and asynchronous communication, *a combination of HTTP requests and message queues* is adopted in the literature and the industry.

Point-to-point communication among the communication patterns works for basic microservice-based applications since there is direct communication between services. However, as the number of services increases, this communication model may be a disadvantage for the architecture. Finally, the *"communication using binary protocols pattern"* enables effective and reliable communication by supporting different programming languages such as Java, Python, C++, Perl, JavaScript, etc. Additionally, this pattern enables developers to define data types and interfaces in a language-independent format. Therefore, it may be preferable in case of many services created with different programming languages.

4.4.3. RQ3: What are the identified issues and obstacles related to the communication and deployment of microservices?

4.4.3.1. Deployment challenges. Architectural complexity: Developing and deploying microservices independently cause complexity in different aspects such as continuous development/delivery, monitoring, scaling, recovery. Thus, managing microservices manually is not feasible in practice (S, 2018). In P2, the authors indicated that if the microservices' design and deployment process must be validated and enforced on an ongoing basis, the implementation of microservices will be much easier. In study P19, it is mentioned that deploying complex systems with microservices is a serious problem due to the need to deploy a large number of packages, interdependencies of services, and the need for software components to allocate available computing resources optimally.

Independently failed microservices (Fault tolerance): Deployment of services independently in microservices architecture also enables isolation of service failures to avoid catastrophic failure of the system (Janssen, 2017). In this case, the developer must consider fault tolerance when deploying microservices.

Deployment coordination: There is an interconnected graph model in which a service is connected to more than one service. This graphic should be kept as a Directed Acyclic Graph (DAG) to prevent endless looping of services. Otherwise, the user can face overflow errors (Usman, 2016).

Distributed logs: Deployment of microservice instances in an application causes the logs to be distributed. When there is a problem in a microservice-based application, operators of the application can have difficulty in analyzing all distributed logs to solve the problem (Janssen, 2017).

Deployment cost: In microservice-based applications, each team should be aware of the operational and implemental costs of their services when an update is performed in these services. P1 mentioned that it is a challenge to predict the deployment cost impact of this change on the other related services. For instance, a change in a service may cause an additional load on other services, and these services need to be scaled out. To handle this, a CostHat model is proposed in study P1, and an approach has been developed to analyze the cost impacts of changes in a workload (depending on an increasing number of users). Furthermore, this approach estimates the cost effects of adding new features or new updates to the services. In another study, P30 indicates that the deployment cost of microservices in clouds is the main concern for service providers and cloud operators. Thus, the authors aim to minimize the operational and deployment cost of microservices besides preserving service delay requirements for applications. Many primary studies are addressing the deployment costs of microservices (P19, P20, P26, P28, P29, P33, P37).

Container image vulnerability: Since microservices have the principle of collaboration in large-scale systems, there is often a need for the deployment of new services and updated versions of existing services. Container-based technologies (e.g., Docker, Kubernetes, etc.), which are among the most popular technologies used for deploying microservices, promote the use of black-box reuse to support off-the-shelf deployments. P4 emphasizes that the black box reuse style prevents making sure that the information being transported in the container, where the service will be deployed, is authentic. The authors propose a formalism to use container technologies for deploying microservices more safely and statically analyze the service deployment artifacts.

Required specific configurations: In the paper P12, it is stated that the deployment of microservices architecture on cloud servers (e.g., AWS) requires the deployment of many independent applications, such as gateways and services. When a new gateway or service version is published, it is very easy to break external dependencies. Thus, it is emphasized that maintaining the service versioning is crucial for microservices architecture, and there is a need for specific configurations on the application and services offered by the cloud server. Furthermore, it is indicated that when new services are added or existing services are removed or updated, different teams responsible for various services collaborate for the upgrade process to avoid breaking issues.

Continuous Integration/Continuous Delivery (CI/CD): The paper P29 indicated that while the use of microservices brings various

Table 7
Types of communication patterns.

Communication pattern	P7	P8	P9	P10	P11	P15	P16	P17	P18	P21	P23	P24	P25	P26	P35
Synchronous communication (HTTP-based REST, request/response-based Apache Thrift)	✓				✓		✓	✓		✓	✓			✓	
Publish/subscribe communication	✓	✓				✓	✓					✓		✓	
Combination of HTTP and Message queue			✓												
Communication using message-oriented middleware (RabbitMQ, RESTful API, etc.)				✓									✓		✓
Asynchronous communication	✓								✓					✓	
Point-to-point communication														✓	
Communication using binary protocols (e.g. Apache Thrift)	✓							✓							

Table 8
The allocation of deployment challenges to the studies.

Deployment challenge	Described in the primary study
Architectural complexity	P2, P6, P12, P13, P19
Fault tolerance	P2, P6, P12, P18, P23, P36
Deployment coordination	P1, P2, P3
Distributed logs	P2, P5, P12
Deployment cost	P1, P19, P20, P26, P28, P29, P30, P32, P33, P34, P37
Container image vulnerability	P2, P4
Required specific configuration	P2, P12, P14
Continuous Integration/Continuous delivery (CI/CD)	P2, P29

advantages together, continuous integration and delivery are the main concerns in microservices-based applications because of their short release cycle. Furthermore, the authors state that updates in a service or a new microservice addition need to integrate with an existing application in less effort and less downtime. The studies addressing deployment challenges are listed in Table 8.

When the deployment challenges summarized in Table 8 are analyzed, it is seen that the most handled challenge is the *deployment cost*. Deployment cost refers to the overhead on other services due to updates in services and the total cost of communication and execution operations resulting from the allocation of services to resources. The second most addressed challenges are *architectural complexity and fault tolerance*. In the architectural complexity problem, the number of services developed independently complicates the system. This makes it difficult to deploy services to resources in a feasible way. In the fault-tolerance problem, the errors that occur during the deployment of independent services affect the whole system's operation. Among other challenges, *deployment coordination, distributed logs,* and *required specific configuration* have been addressed in three studies. In order not to expose the user to overflow errors, it is a challenge to ensure deployment coordination that prevents the interconnected services from entering the infinite loop. The logs of the services developed by different teams are also kept in a distributed manner. Analysis of all logs for a problem in any service causes overhead in the system. A possible infrastructure update or reconfiguration requirement is also a deployment challenge in the microservices architecture since updating a service may need infrastructure change. Therefore, using an external configuration server can be more dynamic for service-specific configuration. Finally, two studies deal with *container image vulnerability* and *Continuous Integration/Continuous Delivery (CI/CD)* challenges. Although the container technologies mostly used in the industry are more reliable than VMs, some attacks (access control exploits, container privilege escalations, tampering container images, application code exploits, etc.) can occur through these technologies. Therefore, unauthorized access to the packaged services in the container should be prevented during deployment. *Continuous Delivery* is the ability to manage many changes, such as configuring new features, bug fixes safely and quickly. It is one of the main challenges to be addressed during the deployment of microservice architectures. Similarly, in Continuous Integration, code changes in services are often combined in one major branch, automatic compilation and testing processes require that the code in the major branch always be of production quality. In this context, ensuring continuous integration is among the important challenges.

4.4.3.2. Communication challenges. Discovery of services: One of the most critical issues in communicating microservices is the discovery of available services. A simple solution for service discovery is to register each service to the DNS, but it is not a feasible solution, especially if the number of services is high (Hester, 2017). Service orchestration solutions, such as Consul, are widely adopted in the industry for service registration and discovery (Hester, 2017).

Replicated service instances: Replicating service instances leads to an update of replicated data. Replicating service instances is a challenge since it needs to fairly distribute the requests or messages to all existing instances of the desired service. If the data is non-static and the service constantly needs up-to-date data, doing replication asynchronously is not an easy task (Janssen, 2017).

Load-balancing: In server-side load balancing, the user relies on the load balancer, which determines the most appropriate instance of the target service to send requests. User requests are met by many microservices that reside behind the load balancer. In paper P9, it is stated that the homogeneous user requests pass through the sequential set of microservices and then, these requests are presented in a chain. It is also mentioned that microservice instances can provide multiple chains, and the chains are mutually competitive. Furthermore, the user requests of chains bring different QoS requirements along with varying processing times. It is stated that existing load balancing solutions for microservices fail to provide heterogeneity or inter-chain competition. There is a need for chain-oriented load balancing to balance user requests. However, the authors state that the existing communication patterns, such as HTTP with a message queue, make difficult interconnection management between microservice instances and bring extra challenges to apply chain-oriented load balancing.

Replicating data: Although microservices can be developed independently, the dependence on some services is inevitable. Replication of data in a service can be performed asynchronously if the data is static or the service can work with data that is not updated. However, the smaller the grace period for replication, the more complicated the process will become (Janssen, 2017).

Remote calls: Synchronous remote calls are potential candidates to slow down the system. They can even cause cascading

Table 9
The allocation of communication challenges to the studies.

Communication challenge	Described in the primary study
Discovery of services	P8, P17
Replicated service instances	P11, P15, P16
Load balancing	P6, P7, P9, P10, P25, P26
Replicating data	P6, P11, P18, P22
Remote calls	P11, P15, P16
Relation between tables	P14, P15, P18, P35

system failures if the called service depends on the data or functionality of another service (Janssen, 2017). Furthermore, in paper P15, the authors indicate that remote calls performed for microservices are very expensive, and service developers must pay attention when developing services.

The relation between tables: One of the biggest challenges in the transition from a monolithic architecture to a microservice architecture is to change the communication mechanism. When working on a monolithic application with a relational database, the relationships between tables can be appropriately designed and then matched with object models (P14). When the application is transformed into microservices-based architecture, it is important to divide this complex structure into independently developed and deployed services that create a network with many communication links. Usually, the partition is not as easy as it appears, and every component that should adopt a table-related logic may not become a separate microservice. The communication challenges mentioned according to the studies and references are listed in Table 9.

When the communication challenges summarized in Table 9 are analyzed, *Load Balancing* is seen as the most studied problem. While directing requests from users to related service instances, systems that balance the workload between service instances are needed. The second common challenge, *Replicating data*, is required when one service needs to work with data before another service is updated. Among other challenges, *Replicated service instances* is the challenge of asynchronous replication of service data that requires continuous updating. If the operation of the service called remotely during the communication of microservices depends on the data or functionality of another service. Remote calls cause the system to slow down. The *Relation between tables* is another challenge for microservices. To change the communication mechanism of monolithic applications, the relationship between tables can be regulated since a shared relational database is used. However, this operation is challenging for microservice-based applications. Since these applications consist of many services that use different databases, creating many relations according to the determined communication mechanism is a significant challenge. Finally, although there are some solutions adopted in the industry for the *Discovery of services*, it is difficult to discover the appropriate service in a system where the number of services has increased substantially. Thus, effective service orchestration systems are needed.

4.4.4. RQ4: What are the identified research directions with regard to communication and deployment of microservices?

By addressing this research question, we aim to determine research directions that can help to resolve open issues in the field of communication and deployment of microservices. The open issues that arise with the use of the microservices architectures are listed below:

Complexity control: In the transition from monolithic architectures to microservices, as the number of independently deployed services increases, operational complexity also increases (P6, P12). Hereby, more messages and more interaction between services are needed. Thus, managing all connections among microservices and deploying thousands of services effectively are becoming essential research issues. There is a need for management tools to control the complexity of microservices for easily deploying and communicating them (P2, P13, P19).

Monitoring: There is a need for monitoring tools to closely monitor the system if there is any problem in the communication or deployment of services, such as the crash of a service to be communicated or cascading failure of remote calls during communication. Since any team working in different parts of a microservice application will not be able to fix this problem, the service where the failure originated in the system should be identified immediately using monitoring tools. Many studies try to handle monitoring (Cinque et al., 2018; Fridelin et al., 2018; Haselböck and Weinreich, 2017; Mayer and Weinreich, 2017) include the primary studies P20, P27, but monitoring requirements must be reevaluated because of noisy and reactive monitoring problems. In the proposed monitoring infrastructures for microservices, there are too many false positives with scaled-out services and alert noises with individual health checks and alert fatigue. It is challenging to determine a normal behavior in microservices architectures, as there are often no "steady-states" such as updates, scaling actions, virtualization. Therefore, detection techniques for performance anomalies may produce many false alarms (Heinrich et al., 2017). As a solution, Heinrich et al. (2017) suggested that the logs of change events should be examined to determine whether a difference in the run-time behavior of the system is an anomaly or not.

Services resilience: To avoid deterioration of the working mechanism of the entire system, team members should be prepared for failures, and despite the failure, a good strategy is needed to ensure that the services remain in an operating state. If a service instance has failed, a backup storage system that is publicly shared outside the user-specific service data must be provided to the user. Thus, the user is not affected by the failure, and user interactions continue without any problem. Additionally, the backup storage system can be a solution against the communication challenge of *Replicated service instances*. If a service is updated, another service that deals with the non-updated data of that service can easily use this data. Furthermore, it is stated in P2 that built-in fallback mechanisms are needed in microservice-based applications without negatively affecting the service invocation chain during communication between services. It is mentioned in P2 and P18 that some structures such as circuit breakers and bulkhead can prevent cascading failures.

Efficient logging: Since log messages generated by microservices are deployed to multiple hosts, there is a need for a good logging strategy for helping team members to analyze and resolve the problems that may arise in an application (P2). This system can be a solution for the deployment challenges of *Distributed logs*. Analyzing all logs for a problem in any service can be solved using an efficient logging strategy that brings less overhead to the system. Some strategies are proposed by Janapati (2017), such as centralized and externalized log storage, log-structured data, correlation IDs, dynamic logging levels, async logging, and making logs searchable, etc.

Performance improvement: In a microservice architecture, the communication model needs to consider how the number of requests will affect the performance and whether improvements can be made. It is a research direction for the communication of microservices to improve system performance in this respect. As the demand for cloud resources increases, the efficient use of resources and performance optimization becomes important in terms of cost (P1). That is why the methods proposed for improving performance during the deployment of microservices are important in means of deployment cost and deployment

coordination challenges. Another important performance aspect in means of communication is remote calls. Since remote calls are slower than in-process function calls, if remote calls are restricted to speed up the system, collaboration services should be called as at least one or more call chains (P2). In this case, it is stated in P2 and P9 that there is a need for components that can monitor the service performance and trace the service-call chain.

Minimizing service dependencies: When designing the communication architecture of microservices, it is necessary to mitigate the dependencies of services to each other as much as possible (P29). When the service dependencies cannot be handled, the advantage of using microservices is eliminated. In this context, there is a need for approaches to reduce dependencies when designing microservices' communication architecture. Furthermore, enabling microservice independence may help to reduce *architectural complexity* and *Continuous Integration/Continuous Delivery (CI/CD)* challenges when deploying microservices (P26).

Fault tolerance: In microservices, managing the failure of an individual service is an important process. A current research topic is defining the system's behavior when a service cannot receive a response from a related service (P18). Services are overloaded as a result of the client's concurrent requests. Therefore, resources are occupied until receiving responses from other services. This case causes a cascading failure of the system. Even if failure in only one service cannot be avoided, all other services that rely on that service will gradually fail. This problem is solved by a circuit breaker pattern (Montesi and Weber, 2016). Using this pattern, failure in a single service does not affect other related services, so that the whole system is not affected negatively (P12). The main purpose is to quickly detect the failure, stop waiting for a service response if the service does not respond. The circuit breaker reduces the tasks of the overloaded services by limiting the waste of resources during the waiting period of unresponsive services and provides stability and flexibility (P2). Developing fault-tolerant resistor architecture helps to solve *remote calls* challenges during communication, also *continuous integration and continuous delivery* challenges during deployment.

Enabling security: Microservices that can be developed by different teams using various technologies and data stored in different databases. This situation increases the possibility of an attack on microservices architectures (P2). Besides, the deployment of microservices in a cloud environment can lead to misuse or disclosure of private information of cloud users (P4). Using a security mechanism during the deployment of microservices to servers can be a solution to the *Container image vulnerability* challenge. Since container-based technologies (e.g., Docker) use a black-box reuse mechanism to support off-the-shelf deployments, this mechanism prevents making sure that whether the service to be delivered is attacked or not in the container during the deployment (P4). To prevent the disadvantages of deploying services to the external world; container safety, secure data communication between microservices, and authenticity detection can be studied as future research directions.

Automated deployment system: According to information obtained from the research questions about deploying microservices, the deployment of microservices to resources is generally done with expert judgment. The main problems of expert judgment are that the expert needs to know the system's internal dynamics to perform the deployment. Furthermore, when the number of microservices or resources increases, the problem cannot be solved with brainpower. For instance, an expert who knows the system can probably deploy 10–15 microservices to 3–5 sources, but it is unlikely that the expert can deploy 400 microservices to 50 sources optimally. Thus, there is a need for automated deployment tools that deploy microservices without detailed user configuration and knowledge of an expert. Developing such automated deployment tools is an important research direction in terms of deploying a large number of packages and services, their detailed interdependencies, and defining the optimal allocation of software components to cloud resources (P19).

The summary of the identified research directions is shown in Table 10. The findings obtained for this research question are presented as open issues for researchers/practitioners to solve the deployment and communication challenges of microservices. Among these issues, *Complexity control, Monitoring, Service resilience, Efficient logging,* and *Fault tolerance* are about preventing and managing failures. As the number of services increases, the whole system becomes harder to control, and tools that minimize system complexity as much as possible are required to minimize errors during communication and deployment. *The monitoring system* is also required for complexity control and intervening to errors occurring in a part of the system immediately. Thus, it prevents errors from affecting the operating performance of the system. In this context, it is necessary to have a monitoring system for every microservice-based application for the application's uninterrupted operation. *Service resilience in the system*, another research direction, is essential to ensure that services are as flexible as possible against any errors occurring in the system and that it can be returned to the last working version of the system when necessary. Also, the *logging system* is required for the teams responsible for different services to be aware of the errors occurring in the services. Systems with fault tolerance should be developed to minimize service errors during deployment.

Another research direction for the deployment of microservices is the development of systems that allow the automatic deployment of a large number of services to resources with limited capacity. Although industrial deployment management tools such as container technologies are used, these tools operate according to the user configuration. As the number of services increases, finding feasible configuration becomes difficult for the user. *Performance improvement* is required for an architecture consisting of many services to prevent the negative effect of the communication cost among the services. It also explains the need to reconstruct the model created to improve performance during performance degradation. The *high cohesion and low coupling* approach is one of the most critical issues that should be considered during the design of a microservice-based application. Although the services need each other for the system to work as a whole, systems that minimize the interdependence of the services as much as possible during architectural design are needed. Finally, like any architectural infrastructure, the system needs a security mechanism for attacks coming from the outside or inside of the cloud deployment environment. In this context, *Enabling security* is one of the important research directions. It is foreseen that the solution suggestions that the researchers/practitioners will offer for these research directions will provide technical and operational benefits to the developing microservice infrastructure.

5. Discussion

This systematic literature study reports on the current deployment approaches for microservice-based applications, the advantages and disadvantages of these approaches compared with each other, the communication patterns between services, and the selection of communication patterns according to the designed architecture. Further, the SLR highlights the challenges related to deployment and communication approaches for microservices and as such it paves the way for further research. In fact, this is a secondary study/systematic literature review that aims to

Table 10
Summary of the identified research directions.

Research direction	Description
Complexity control	Managing hundreds of microservices is a critical challenge for developers. Therefore, there is a need for useful tools to deal with complexity.
Monitoring	If there is a problem in any of the services, monitoring tools are needed to solve the problem without affecting the other services in a short time.
Services resilience	A microservice must be flexible against failures and need a storage system for availability. This flexibility allows the microservice to return to the recorded state before the fault and successfully restart this service.
Efficient logging	There is a need for a good logging strategy to instantly resolve the problems that may arise in the application.
Performance improvement	The performance of the communication model is a critical factor in the data exchange of microservices.
Minimizing service dependencies	To take advantage of microservice architectures, it is necessary to minimize the interdependence of services.
Fault tolerance	Since microservices are deployed independently, service instances can often fail. If the number of interactions between microservices is high, it could have a serious effect on the system in case of failures. To prevent this, a system should be designed as fault-tolerant to operate at a certain level of satisfaction when faults occur.
Enabling security	A microservices-based system must be secure against attacks caused by cloud computing security issues.
Automated deployment system	As the number of services increases, it becomes difficult to deploy a large number of services to limited capacity resources efficiently. Thus, there is a need for automated deployment mechanisms to perform feasible deployment of microservices.

Fig. 10. The summary of all findings.

analyze and synthesize the reported approaches on deployment and communication patterns of microservice architectures. No prior systematic review has been provided on these two critical topics, and in this sense, it is unique. The systematic literature review has been carefully performed according to a well-known and stable protocol. The summary of all findings obtained from our research questions is shown in Fig. 10. Furthermore, the following key findings can be identified:

- *Deployment and communication approaches are still under development*

The findings show that the deployment and communication infrastructures are still developing, and they are in a formative stage. We have reported on the currently used deployment and communication approaches, but given the trends that we have seen in our studies, these might further develop in the future.

- *The majority of the deployment approaches uses container technology*

According to the data extracted from the 22 primary studies for deployment approaches, 18 (81.82%) studies use container technology, while 3 (13.64%) studies use VM technology to deploy microservices. Only 1 (4.54%) study uses serverless technology.

- *The widely used communication approaches are synchronous and publish/subscribe*

According to the data obtained from 21 primary studies for the communication patterns of microservices, *synchronous* and *publish/subscribe communication* approaches are mentioned in 7 (33.33%) (P7, P11, P16, P17, P21, P23, P26) and 6 (28.57%) (P7, P8, P15, P16, P24, P26) studies respectively. These approaches are determined as the most widely used communication patterns. Then, Communication using message-oriented middleware (P10, P25, P35) and asynchronous communication approaches (P7, P18, P26) are each mentioned in 3 (14.29%) studies. In 2 (9.52%) studies (P7, P17), Communication using binary protocols approaches are utilized. Finally, in one study (4.76%), a combination of HTTP and message queue (P9) and Point-to-point (P26) approaches are selected as the communication protocol.

- *Service instance per container approach is preferred to support communication and deployment*

According to the findings obtained, the "service instance per container" approach is a practical approach for those who adopt microservice architectures, as this approach has a lower resource footprint, faster startup/destroy times, faster deployment, and higher I/O performance than "service instance per VM" approach. Containers help protect software components from safety hazards and decrease the conflicts between different teams working on the same infrastructure since they isolate software from environmental factors. Besides, it is deduced from the findings that containers ease the deployment, management, and scaling process in terms of developmental and operational aspects compared to VMs.

- *Serverless deployment has challenges*

As discussed in Section 4.4.1, the serverless deployment approach has been emphasized in recent years, but this method does not allow the configuration of deployment infrastructure by the development team. Further, some functions (e.g., AWS Lambda functions) are not suitable for long-running services, such as a video upload operation that may take more than the timeout limit (Zhang et al., 2020). Moreover, due to vendor lock-in problems, serverless functions can only be written in one of the supported languages selected by the serverless platform. Despite the disadvantages, we can also see that the current infrastructures are continuously improved. For example, the timeout limit for single execution is remarkably enhanced. Serverless architectures may gain more popularity if the vendors continue to invest and improve their frameworks.

- *Hybrid communication methods can be useful*

Although the most common communication pattern identified is synchronous, the hybrid communication method called "Combination of HTTP and message queue" can be preferred because of the time-performance in synchronous communication and the high retention nature of message queues. In large-scale microservice-based applications, the use of asynchronous communication protocols for interactions among microservices can be preferred. On the other hand, it is more convenient to process the requests from the client in real-time by using synchronous (HTTP) communication, especially when integrating with third-party external services. For small-scale microservice-based applications, point-to-point communication infrastructure can be an alternative to synchronous and asynchronous communication protocols by enabling faster communication of service instances with each other directly. More details about this discussion can also be found at the end of Section 4.4.2.

- *Finding minimum cost for deployment is the most important metric, followed by fault tolerance and load balancing*

Among the identified challenges in RQ3 research question, finding the minimum cost of deployment is essential for a microservice architecture. To deploy services with the minimum cost by using servers efficiently in a cloud environment based on the pay-as-you-go principle is necessary for both developer and cloud resources. The study shows that the deployment cost is the most common deployment challenge with 11 (28.9%) of all primary (38) studies; the secondary ones are fault tolerance and load balancing, where each handled in 6 (15.8%) studies among the identified challenges.

- *Several open research challenges still exist; automated deployment and fault tolerance are considered key important challenges*

In the final research question RQ4, various research directions and open issues for microservices-based applications are extracted from the identified challenges. Among these issues, an automated deployment system is considered an important research challenge since the deployment is generally left to the user configuration using the industrial management tools. Besides, since microservices are deployed independently, the fault tolerance issue is another important research direction. The design of a fault-tolerant microservice-based system may also improve communication efficiency among microservices and helps to avoid cascading system failures when one or more services fail.

In this study, four potential threats of validity based on a standard checklist developed by Wohlin et al. are discussed, i.e., internal validity, construct validity, conclusion validity, and external validity.

Internal validity: The most critical threat to validity in this systematic literature review is that the communication and deployment challenges of microservices are not adequately addressed in the research publications, and the recommended methods and studies to overcome these challenges are very few. Besides, although our search scope and search terms are wide, it is observed that the majority of selected studies dealing with communication and deployment patterns tend to publish positive results. To evaluate the negative aspects of the studies, all studies have been investigated in-depth and negative results have been tried to be

Table A.1

Paper number	Authors	Year	Title	Source
P1	Leitner P., Cito J., Stöckli E.	2016	Modeling and Managing Deployment Costs of Microservice-Based Cloud Applications	2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing
P2	Sengupta D., Bagchi H., Kanti Gri P	2017	Accelerating Microservices Design and Development	Cognizant 20-20 Insights | May 2017
P3	Mittal S., Risco-Martin J.	2017	Devsml 3.0 Stack: Rapid Deployment Of Devs Farm In Distributed Cloud Environment using Microservices and Containers	Society for Modeling and Simulation International (SCS)
P4	Benni B., Mosser S., Collet P., Riveil M.	2018	Supporting Micro-services Deployment in a Safer Way: a Static Analysis and Automated Rewriting Approach	SAC'18, April 9–13, 2018, Pau,France
P5	Villari M., Celesti A., Tricomi G., Galletta A.	2017	Deployment Orchestration of Microservices with Geographical Constraints for Edge Computing	IEEE Symposium on Computers and Communications (ISCC)
P6	Richter D., Konrad M., Utecht K., Polze A.	2017	Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice	IEEE International Conference on Software Quality, Reliability and Security (Companion Volume)
P7	Bakshi K.	2017	Microservices-Based Software Architecture and Approaches	IEEE Aerospace Conference
P8	Petrasch R.	2017	Model-based Engineering for Microservice Architectures using Enterprise Integration Patterns for inter-service Communication	14th International Joint Conference on Computer Science and Software Engineering (JCSSE)
P9	Niu Y., Liu F., Li Z.	2018	Load Balancing across Microservices	IEEE INFOCOM 2018 — IEEE Conference on Computer Communications
P10	Jun Hong X., Sik Yang H., Han Kim Y.	2018	Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application	International Conference on Information and Communication Technology Convergence (ICTC)
P11	de Guzm'an P. Gorostiga F., S'anchez C.	2018	Pipekit: a Deployment Tool with advanced scheduling and Inter-Service Communication for Multi-Tier Applications	IEEE International Conference on Web Services
P12	Villamizar M., Garcés O., Castro H., Gil S.	2015	Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud	10th Computing Colombian Conference (10CCC)
P13	Florio L., Di Nitto E.	2016	Gru: an Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures	IEEE International Conference on Autonomic Computing
P14	Guo D., Wang W., Zeng G., Wei Z.	2016	Microservices Architecture based Cloudware Deployment Platform for Service Computing	2016 IEEE Symposium on Service-Oriented System Engineering
P15	Namiot D., Sneps-Sneppe M.	2014	On Micro-services Architecture	International Journal of Open Information Technologies ISSN: 2307–8162 vol. 2, no. 9, 2014
P16	Krylovskiy A., Jahn M., Patti E.	2015	Designing a Smart City Internet of Things Platform with Microservice Architecture	The 3rd International Conference on Future Internet of Things and Cloud (FiCloud)
P17	Wolff E.	2016	Microservices: Flexible Software Architectures	Addison–Wesley Professional
P18	Hofmann M., Schnabel E., Stanley K.	2016	Microservices Best Practices for Java	IBM Redbooks
P19	Di Cosmo R., Eiche A., Mauro J., Zavattaro G., Zacchiroli S., Zwolakowski J.	2015	Automatic Deployment of Software Components in the Cloud with the Aeolus Blender	[Technical Report] Inria Sophia Antipolis
P20	Gabbrielli M., Giallorenzo S., Guidi C., Mauro J., Montesi F.	2016	Self-Reconfiguring Microservices	Theory and Practice of Formal Methods, Springer, pp.194–210, 2016, Lecture Notes in Computer Science
P21	Safina L., Mazzara M., Montesi F., Rivera V.	2016	Data-driven Workflows for Microservices	IEEE 30th International Conference on Advanced Information Networking and Applications
P22	Lawson J., Wolthius J.	2016	System and Method For Providing a Micro-Services Communication Platform	United States Patent
P23	Rufino J., Alam M., Ferreira J., Rehman A., Fung Tsang K.	2017	Orchestration of Containerized Microservices for IIoT using Docker	IEEE International Conference on Industrial Technology (ICIT)
P24	Bertelsen E., BerthlingHansen G., Bloebaum T., Duvholt C., Hov E., Johnsen F., Morch E., Weisethaunet A.	2018	Federated Publish/subscribe Services	9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)

(*continued on next page*)

Table A.1 (*continued*).

Paper number	Authors	Year	Title	Source
P25	Alam M., Rufino J., Ferreira J., Ahmed S., Shah N., Chen Y.	2018	Orchestration of Microservices for IoT Using Docker and Edge Computing	IEEE Communications Magazine September 2018
P26	Bhamare D., Samaka M., Erbad A., Gupta L.	2018	Exploring microservices for enhancing internet QoS	Trans Emerging Tel Tech. 2018;29:e3445, wileyonlinelibrary.com/journal/ett
P27	Ciuffoletti A.	2015	Automated deployment of a microservice-based monitoring infrastructure	Procedia Computer Science 68 (2015) 163–172
P28	Zheng T., Zhang Y., Zheng X., Fu M., Liu X.	2017	BigVM: A Multi-Layer-Microservice Based Platform for Deploying SaaS	Fifth International Conference on Advanced Cloud and Big Data
P29	Singh V., K Peddoju S.	2017	Container-based Microservice Architecture for Cloud Applications	International Conference on Computing, Communication and Automation (ICCCA2017)
P30	Wan X., Guan X., Wang T., Bai G., Choi B.	2018	Application deployment using Microservice and Docker containers: Framework and optimization	Journal of Network and Computer Applications 119 (2018) 97–109
P31	Kaewkasi C., Chuenmuneewong K.	2017	Improvement of Container Scheduling for Docker using Ant Colony Optimization	9th International Conference on Knowledge and Smart Technology (KST) (Feb 2017)
P32	Xu X., Yu H., Pei X.	2014	A Novel Resource Scheduling Approach in Container Based Clouds	IEEE 17th International Conference on Computational Science and Engineering
P33	Guan X., Wan X., Choi B., Song S., Zhu J.	2017	Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers	IEEE Communications Letters, Vol. 21, No. 3, March 2017
P34	Guerrero C., Lera I., Juiz C.	2018	Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture	J Grid Computing (2018) 16:113–135
P35	Smid, A., Wang, R., Cerny, T.	2019	Case Study on Data Communication in Microservice Architecture	Proceedings of the Conference on Research in Adaptive and Convergent Systems
P36	Profeta, D., Masi, N., Messina, D., Dalle C.D., Bonura, S., Morreale, V.	2019	A Novel Micro-Service Based Platform for Composition, Deployment and Execution of BDA Applications	45th Euromicro Conference on Software Engineering and Advanced Applications
P37	Zheng T., Zhang X., Zhang, Y., Deng, Y., Dong, E., Zhang, R., Liu, X.	2019	SmartVM: a SLA-aware microservice deployment framework	Springer Science+Business Media
P38	Sampaio Jr, A., Rubin, J., Beschastnikh, I., Rosa, N.	2019	Improving microservice-based applications with runtime placement adaptation	Journal of Internet Services and Applications

extracted. There might be some missing studies in the search phase which the researchers could not access. To address the lack of search bias, which is another type of threat to validity, the keyword list is repeated continuously. When the search engine lists studies that are unrelated to the given keyword or that are too far from the desired topic, the keyword is deleted from the previously created keyword list, and new related keywords are added to the list.

Construct validity: This type of validation relates to the assessment of difficulties encountered during the data extraction phase (Wohlin, 2014). To overcome data extraction bias, studies that could contribute significantly to review results are selected and evaluated. Furthermore, to make data extraction modeling in the best way, the studies that contain the closest answers to the research questions are identified as primary studies. After the data extraction process is completed, a data extraction form is created, and then unnecessary details or unrelated results are eliminated. To minimize data extraction bias, all selected studies have been repeatedly evaluated until reaching the final data extraction model. Which study is appropriate for which research question and the communication and deployment hints extracted from these questions are added and removed several times until the results converge.

Conclusion validity: As mentioned in Section 3.6, qualitative and quantitative data analyses are carried out to ensure the rigor and repeatability of this SLR study. While deployment approaches and communication patterns are listed by identifying 34 primary studies, the challenges encountered while applying these approaches and research directions for future studies are extracted from the selected studies.

External validity: This type of validity concerns the applicability of the results of the study in a more general context. We have focused on the adopted deployment approaches and communication patterns that were discussed in the identified primary studies. The result of the SLR is a synthesis of the findings of these primary studies. Novel deployment approaches or communication patterns might be defined in the future. However, for the period that we have covered, we believe that we have captured all the adopted deployment approaches and communication patterns.

6. Related work

Several secondary studies have been carried out on microservices architecture including SLRs and systematic mapping studies. Alshuqayran et al. (2016) report on a mapping study based on 33 articles from 2014 to 2016. They handle three kinds of research questions, including (i) architectural challenges that microservices face, (ii) architectural diagrams/views used to represent microservices, and (iii) quality attributes related to microservices presented in the literature. Our paper is similar concerning quality attributes, but while Alshuqayran et al. (2016) only aims to recognize and disclose the gap in the current research, we have also reviewed the proposed techniques or tool support to improve the quality attributes. In Li (2017), Li aims to provide knowledge

Table B.1

Paper	Quality of reporting			Rigor			Credibility		Relevance		Quality of reporting	Rigor	Credibility	Relevance	Total
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10					
P1	1	1	0.5	0.5	1	0.5	0	1	0.5	0.5	2.5	2	1	1	6.5
P2	1	1	0.5	0.5	1	0.5	0	0.5	1	1	2.5	2	0.5	2	7
P3	1	1	1	0.5	1	0.5	0	1	1	1	3	2	1	2	8
P4	1	1	1	1	1	0.5	1	0.5	1	1	3	2.5	1.5	2	9
P5	1	0.5	0.5	0.5	1	0.5	0	0.5	1	1	2	2	0.5	2	6.5
P6	1	1	1	1	1	1	0	0.5	1	1	3	3	0.5	2	8.5
P7	1	0.5	0.5	0	0.5	0.5	0	0	0.5	0.5	2	1	0	1	4
P8	1	1	1	1	1	0.5	0	0.5	1	1	3	2.5	0.5	2	8
P9	1	1	1	1	1	0.5	1	1	1	1	3	2.5	2	2	9.5
P10	1	1	1	1	1	0.5	0	0.5	1	1	3	2.5	0.5	2	8
P11	1	1	1	0.5	0.5	1	0	0.5	1	1	3	2	0.5	2	7.5
P12	1	1	1	1	1	0.5	0	0.5	1	1	3	2.5	0.5	2	8
P13	1	1	1	1	1	0.5	1	0.5	0.5	1	3	2.5	1.5	1.5	8.5
P14	1	1	1	1	1	0.5	0	0.5	0.5	0.5	3	2.5	0.5	1	7
P15	1	1	0.5	0.5	1	0.5	0	0.5	0.5	0.5	2.5	2	0.5	1	6
P16	1	1	1	1	1	0.5	1	1	1	1	3	2.5	2	2	9.5
P17	1	1	0	1	1	1	1	1	1	0.5	2	3	2	1.5	8.5
P18	1	1	0	1	1	1	0	1	1	0.5	2	3	1	1.5	7.5
P19	1	1	1	1	1	0.5	0	1	1	1	3	2.5	1	2	8.5
P20	1	1	1	1	1	0.5	0	1	1	1	3	2.5	1	2	8.5
P21	1	1	1	1	1	0.5	1	1	1	1	3	2.5	2	2	9.5
P22	1	1	1	1	1	0.5	0	1	1	1	3	2.5	1	2	8.5
P23	1	1	1	0.5	1	0.5	1	0.5	0.5	0.5	3	2	1.5	1	7.5
P24	1	1	1	1	1	0.5	0	1	1	1	3	2.5	1	2	8.5
P25	1	1	1	1	1	1	0	1	1	1	3	3	1	2	9
P26	1	1	1	1	1	1	0	1	1	1	3	3	1	2	9
P27	1	1	1	1	1	0.5	0	1	1	1	3	2.5	1	2	8.5
P28	1	1	1	1	1	0.5	0	1	0.5	0.5	3	2.5	1	1	7.5
P29	1	1	0.5	0.5	0.5	0.5	0	0.5	0.5	0.5	2.5	1.5	0.5	1	5.5
P30	1	1	1	1	1	1	0	1	1	1	3	3	1	2	9
P31	1	1	1	1	1	1	0.5	1	1	1	3	3	1.5	2	9.5
P32	1	1	1	1	1	0.5	0	0.5	0.5	0.5	3	2.5	0.5	1	7
P33	1	1	1	1	1	0.5	0	1	1	1	3	2.5	1	2	8.5
P34	1	1	1	1	1	0.5	1	1	1	1	3	2.5	2	2	9.5
P35	1	1	0.5	1	1	0.5	1	1	0,5	1	2.5	2.5	2	1.5	8.5
P36	1	1	1	1	1	0.5	0	0.5	1	1	3	2.5	0.5	2	8
P37	1	1	1	1	1	0.5	0.5	1	1	1	3	2.5	1.5	2	9
P38	1	1	1	1	1	0.5	1	1	1	1	3	2.5	2	2	9.5

of quality attributes and quality improvements in a microservice architecture. This paper presents a high-level discussion of quality attributes. Our paper differs from Li (2017) in terms of identifying quality attributes and detailed expressions of their solutions.

Pahl and Jamshidi (2016) conduct a systematic mapping study of 21 primary studies from 2013 to 2015. Their study (i) focuses on the main practical motivations behind using microservices (ii) different types of microservices architectures involved (iii) the existing methods, techniques, and tool support to enable microservice architecture development and operation, and (iv) the existing research issues and what should be the future research agenda. Our paper differs from this paper in the following terms, (i) this study (Pahl and Jamshidi, 2016) does not investigate the potential for industrial adaptation of existing research on architecting microservices and also does not mention the quality attributes of microservices. When the answers to the questions that are common in both studies are compared, our study clearly describes the level of provided tool support and how the proposed techniques work in the literature.

Francesco et al. (2017) investigate three types of research questions, which are (i) Publications trends (ii) Focus of research, and (iii) Potential for industrial adoption. The authors define a classification framework and select 71 studies as primary. As main findings, they report that the number of publications on architectural microservices increases, and publication venues are scattered to different topics, and many architecture solutions are proposed in selected papers. Additionally, the author classifies papers according to the quality attributes of microservices, architectural activities such as recovery architecture, synthesis architecture, architectural maintenance, and evaluation architecture, etc.

Soldani et al. (2018) explore the technical and operational gains and problems of microservices by addressing the gap between academia and industry in their systematic literature review. As a result of the study, it is concluded that the intrinsic complexity of microservice-based applications causes the problems of microservices. Managing the distributed storage and implementing tests are identified as the primary problems during development. On the positive side, microservices are developed independently by different developers. Thus, a developer can select the technology stack and choose the best compatible database type with the service. Furthermore, it is mentioned as one of the most essential benefits of microservices architecture is the ability to scale and deploy services independently.

7. Conclusion

In this SLR study, seven different types of communication patterns and three deployment approaches are identified. Publish/subscribe communication (one-to-many interactions) appeared to be the most frequently used method among the identified communication patterns. Furthermore, Service Instance per Container Pattern is the most preferred deployment approach in the selected studies. The deployment appears to be carried out manually in most cases, although it is not a trivial process. Despite

the success and potential of microservices, we have also identified several obstacles that indicate the need for further research. We have identified eight challenges related to the deployment and six challenges related to communication concerns. Among the identified communication challenges, load balancing is the most frequently addressed problem in the primary studies. As a result of the evaluations, it is concluded that the existing load balancing solutions developed for microservices could not provide heterogeneous and inter-chain competition. Moreover, among the deployment challenges, the deployment cost of microservices is identified as the most discussed issue. In microservice-based architectures, even if the teams know the operational and building costs of the service they are responsible for, any updates to the services require updating and scaling of the other related services (Farcic, 2016). However, the teams may not accurately estimate the cost of updating and scaling on the other services. Therefore, considering the deployment costs is an important issue in microservice architectures. Finally, nine different research directions have been reported to guide researchers in the field of microservice architectures. The results of this study pave the way for further research of the microservice architecture. In our future work, we will focus on several of these challenges and enhance the microservice architecture accordingly.

CRediT authorship contribution statement

Işıl Karabey Aksakalli: Data curation, Writing - original draft, Visualization, Investigation. **Turgay Çelik:** Data curation, Conceptualization, Writing - review & editing, Visualization. **Ahmet Burak Can:** Supervision, Data curation, Conceptualization, Visualization, Writing - review & editing. **Bedir Tekinerdoğan:** Supervision, Conceptualization, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. List of primary studies included in the SLR

See Table A.1.

Appendix B. Study quality assessment

See Table B.1.

References

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture. In: Paper Presented at the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA).

Amazon, 2020a. Amazon Simple Queue Service. Retrieved from https://aws.amazon.com/tr/sqs/.

Amazon, 2020b. AWS Lambda. Retrieved from https://aws.amazon.com/tr/lambda/.

Apache, 2020a. Activemq. Retrieved from http://activemq.apache.org/.

Apache, 2020b. Kafka. Retrieved from https://kafka.apache.org.

Apache, 2020c. Zookeeper. Retrieved from https://zookeeper.apache.org/.

Boyle, E., Echenique, F., 2009. Sequential entry in many-to-one matching markets. Soc. Choice Welf. 33 (1), 87–99, Retrieved from www.jstor.org/stable/41107997.

Cinque, M., Corte, R.D., Iorio, R., Pecchia, A., 2018. An exploratory study on zeroconf monitoring of microservices systems. In: Paper Presented at the 2018 14th European Dependable Computing Conference (EDCC).

Daya, S.V.D., Nguyen, Eati, Kameswara, Ferreira, Carlos M., Glozic, Dejan, Gucer, Vasfi, Gupta, Manav, Joshi, Sunil, Lampkin, Valerie, Martins, Marcelo, Narain, Shishir, Vennam, Ramratan, 2016. Microservices from Theory to Practice: Creating Applications in IBM Bluemix using the Microservices Approach. IBM Redbooks.

Di Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. J. Syst. Softw. 150, 77–97. http://dx.doi.org/10.1016/j.jss.2019.01.001.

Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L., 2018. Microservices: How to make your application scale. In: Petrenko, A.K., Voronkov, A. (Eds.), Perspectives of System Informatics, Psi 2017, Vol. 10742. pp. 95–104.

Farcic, V., 2016. The DevOps 2.0 Toolkit. Packt Publishing.

Francesco, P.D., Malavolta, I., Lago, P., 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: Paper Presented at the 2017 IEEE International Conference on Software Architecture (ICSA).

Fridelin, Y.Y., Albaab, M.R.U., Besari, A.R.A., Sukaridhoto, S., Tjahjono, A., 2018. Implementation of microservice architectures on SEMAR extension for air quality monitoring. In: Paper Presented at the 2018 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC).

Gale, D., Shapley, L.S., 2013. College admissions and the stability of marriage. Amer. Math. Monthly 120 (5), 386–391. http://dx.doi.org/10.4169/amer.math.monthly.120.05.386.

Guidi, C., Lanese, I., Mazzara, M., Montesi, F., 2017. Microservices: A language-based approach. In: Mazzara, M., Meyer, B. (Eds.), Present and Ulterior Software Engineering. Springer International Publishing, Cham, pp. 217–225.

Haselböck, S., Weinreich, R., 2017. Decision guidance models for microservice monitoring. In: Paper Presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).

HashiCorp, 2020. Retrieved from https://www.consul.io/.

Heinrich, R., Hoorn, A.v., Knoche, H., Li, F., Lwakatare, L.E., Pahl …, C., Wettinger, J., 2017. Performance Engineering for Microservices: Research Challenges and Directions. Association for Computing Machinery, L'Aquila, Italy.

Hester, A., 2017. Communicating with microservices. Retrieved from https://medium.com/high-alpha/communicating-with-microservices-92a2c59d697c.

Hunt, J., 2018. A Beginner's Guide to Scala, Object Orientation and Functional Programming. Springer.

Janapati, S.P.R., 2017. Distributed logging architecture for microservices. Retrieved from https://dzone.com/articles/distributed-logging-architecture-for-microservices.

Janssen, T., 2017. Communication between microservices: How to avoid common problems. Retrieved from https://stackify.com/communication-microservices-avoid-common-problems/.

Kitchenham, B., Brereton, P., 2013. A Systematic Review of Systematic Review Process Research in Software Engineering, Vol. 55. Butterworth-Heinemann.

Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S., 2009. Systematic Literature Reviews in Software Engineering - A Systematic Literature Review, Vol. 51. Butterworth-Heinemann.

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

Krylovskiy, A., Jahn, M., Patti, E., 2015. Designing a smart city internet of things platform with microservice architecture. In: Paper Presented at the 2015 3rd International Conference on Future Internet of Things and Cloud.

Kubernetes, 2020. Retrieved from https://kubernetes.io/.

Li, S., 2017. Understanding quality attributes in microservice architecture. In: Paper Presented at the 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW).

Lin, J., Mauro, J., Røst, T.B., Yu, I.C., 2017. A model-based scalability optimization methodology for cloud applications. In: Paper Presented at the 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2).

Mayer, B., Weinreich, R., 2017. A dashboard for microservice monitoring and management. In: Paper Presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).

Mesosphere, 2020. Marathon. Retrieved from https://mesosphere.github.io/marathon/.

Montesi, F., Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. arXiv preprint arXiv:1609.05830.

Namiot, D., Sneps-Sneppe, M., 2014. On micro-services architecture. Int. J. Open Inf. Technol. 2 (9), 24–27.

Netflix, 2020. Eureka. Retrieved from https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance.

Newman, S., 2015. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc.

omgwiki, 2020. dds. Retrieved from https://www.omgwiki.org/dds/.

Pahl, C., Jamshidi, P., 2016. Microservices: A systematic mapping study. In: Paper Presented at the CLOSER (1).

Potvin, P., Nabaee, M., Labeau, F., Nguyen, K.-K., Cheriet, M., 2016. Micro service cloud computing pattern for next generation networks. In: Smart City 360°. Springer, pp. 263–274.

RabbitMQ, 2020. Rabbitmq. Retrieved from https://www.rabbitmq.com/.

Rahmel, D., 2013. Testing a site with ApacheBench, JMeter, and Selenium. In: Advanced Joomla!. Springer, pp. 211–247.

S, D., 2018. Microservice issues, challenges, and hurdles. Retrieved from https://dzone.com/articles/microservice-issues-challenges-and-hurdles.

Sakovich, N., 2020. Monolithic vs. microservices: Real business examples. Retrieved from https://www.sam-solutions.com/blog/microservices-vs-monolithic-real-business-examples/.

Sill, A., 2016. The design and architecture of microservices. IEEE Cloud Comput. 3 (5), 76–80. http://dx.doi.org/10.1109/MCC.2016.111.

Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J., 2018. The pains and gains of microservices: A systematic grey literature review. J. Syst. Softw. 146, 215–232.

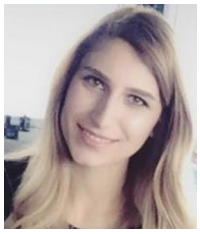
Usman, İ., 2016. Challenges of micro-service deployments. Retrieved from http://techtraits.com/microservice.html.

Wilde, E., Pautasso, C., 2011. REST: From Research to Practice. Springer Science & Business Media.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Paper Presented at the Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering.

Zhang, H., Babar, M.A., Tell, P., 2011. Identifying relevant studies in software engineering. Inf. Softw. Technol. 53 (6), 625–637.

Zhang, W., Fang, V., Panda, A., Shenker, S., 2020. Kappa: a programming framework for serverless computing. In: Paper Presented at the Proceedings of the 11th ACM Symposium on Cloud Computing.



Işıl Karabey Aksakallı graduated from Gazi University in 2013 and started to work as a research assistant at Atatürk University in 2014. After receiving her M.Sc. degree from Atatürk University in 2015, she was appointed as a research assistant to Erzurum Technical University. She started her Ph.D. in 2016 at Hacettepe University and still continues to this program. Her research topics include microservice architectures, optimization methods, distributed systems, machine learning and deep learning techniques.

Turgay Çelik received his BS (2003),M.Sc. (2005) and Ph.D. (2013) degrees in Computer Engineering from Hacettepe University, Turkey. From 2003 to 2005 he served as a research assistant in Hacettepe University. Currently, he is a Lead Software Engineer in MilSOFT Inc. Turkey, where he has been working since 2005. He has 10 years of professional experience in software engineering research and software development. His research topics include distributed systems, infrastructure and middleware technologies, modeling and simulation, software architecture modeling, model-driven software development, software design optimization,and software performance profiling and optimization.

Ahmet Burak Can is currently affiliated with Department of Computer Engineering at Hacettepe University, Turkey. He received the Ph.D. degree in Computer Science from Purdue University, West Lafayette. He has BS and MS degrees in Computer Science and Engineering from Hacettepe University. He is a member of the IEEE. His main research areas are computer vision, distributed systems, and network security.

Bedir Tekinerdogan is a full professor and chair of the Information Technology group at Wageningen University in The Netherlands. He received his M.Sc. degree (1994) and a Ph.D. degree (2000) in Computer Science, both from the University of Twente, The Netherlands. He has more than 25 years of experience in software/systems engineering and is the author of more than 300 peer-reviewed scientific papers. He has been active in dozens of national and international research and consultancy projects with various large software companies, whereby he has worked as a principal researcher and leading software/system architect.