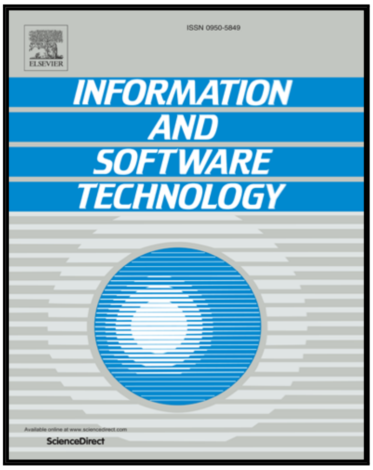
Journal Pre-proof


ISSN 0950-5849
ELSEVIER
INFORMATION AND SOFTWARE TECHNOLOGY
Available online at www.sciencedirect.com
ScienceDirect

Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review

Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, Muhammad Ali Babar

PII: S0950-5849(20)30199-3
DOI: https://doi.org/10.1016/j.infsof.2020.106449
Reference: INFSOF 106449

To appear in: *Information and Software Technology*

Received date: 20 November 2019
Revised date: 15 September 2020
Accepted date: 6 October 2020

Please cite this article as: Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, Muhammad Ali Babar, Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review, *Information and Software Technology* (2020), doi: https://doi.org/10.1016/j.infsof.2020.106449

Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review

Shanshan Li[a], He Zhang[a,*], Zijia Jia[a], Chenxing Zhong[a], Cheng Zhang[b], Zhihao Shan[c], Jinfeng Shen[a] and Muhammad Ali Babar[d]

[a] *State Key Laboratory of Novel Software Technology Software Institute, Nanjing University, Jiangsu, China*
[b] *School of Computer Science & Technology, Anhui University, Anhui, China*
[c] *Tencent Technology Co.Ltd, Shenzhen, China*
[d] *CREST, University of Adelaide, Adelaide, Australia*

ARTICLE INFO

Keywords:
Microservices
Monolith
Quality attributes
Systematic literature review

ABSTRACT

Context: As a rapidly adopted architectural style in software engineering, Microservices Architecture (MSA) advocates implementing small-scale and independently distributed services, rather than binding all functions into one monolith. Although many initiatives have contributed to the quality improvement of microservices-based systems, there is still a lack of a systematic understanding of the Quality Attributes (QAs) associated with MSA.

Objective: This study aims to investigate the evidence-based state-of-the-art of QAs of microservices-based systems.

Method: We carried out a Systematic Literature Review (SLR) to identify and synthesize the relevant studies that report evidence related to QAs of MSA.

Results: Based on the data extracted from the 72 selected primary studies, we portray an overview of the six identified QAs most concerned in MSA, *scalability*, *performance*, *availability*, *monitorability*, *security*, and *testability*. We identify 19 tactics that architecturally address the critical QAs in MSA, including two tactics for *scalability*, four for *performance*, four for *availability*, four for *monitorability*, three for *security*, and two for *testability*.

Conclusion: This SLR concludes that for MSA-based systems: 1) Although *scalability* is the commonly acknowledged benefit of MSA, it is still an indispensable concern among the identified QAs, especially when trading-off with other QAs, e.g., *performance*. Apart from the six identified QAs in this study, other QAs for MSA like *maintainability* need more attention for effective improvement and evaluation in the future. 3) Practitioners need to carefully make the decision of migrating to MSA based on the return on investment, since this architectural style additionally cause some pains in practice.

1. Introduction

Microservices Architecture (MSA) originates from agile developer communities and has received shifting focus in both industry and academia in recent years [10] after the prosperity of SOA (Service-oriented Architecture). No agreement has been reached about the relationship between MSA and SOA [29]. Some MSA advocates claim that MSA is a new architectural style, while some SOA proponents think that MSA is merely an implementation approach of SOA. Our paper follows the former claim based on the definition and nine characteristics of MSA from Fowler et.al. [10] and the seven principles defined by Newman [22].

MSA advocates decomposing the application into a set of small services and making them communicate with each other through light weight mechanisms (e.g., RESTful API or stream-based communications) [10]. Typical characteristics and benefits [29] of MSA are "componentization via services" improving *modifiability*, *scalability*, and *deployability*, "organized around business capabilities" making code understandable and easy to maintain, "infrastructure automation" for continuous delivery and DevOps, and "Decentralized governance and data management" enabling polyglot persistence promotes flexibility and suitability, etc. Inspired by the perceived benefits of MSA, many world-leading Internet enterprises such as Netflix, Amazon and eBay have migrated to MSA. Some technology stacks, e.g., Spring Cloud [1] and TARs [2], make the migration and development of microservices much easier. However, adopting microservices may also introduce complexities to the quality assurance of systems that require much effort to resolve [3, 26].

In the perspective of architecture, the quality assurance is considered as a key concern during the migration or the development of microservices-based systems [11, 2]. Many researchers have contributed to the design and the quality improvement of microservices-based systems [17, 22, 29]. Some of them even provided tool solutions for specific stakeholders to facilitate the migration when preserving highlighted quality attributes, for example, "TheArchitect"[25], a rule-based tool for software architects to achieve automatic gen-

*Corresponding author. Tel: +86 25 83621189; Fax: +86 25 83621370.

stu.shanshan.li@gmail.com (S. Li); hezhang@nju.edu.cn (H. Zhang); stu_zijiajia@163.com (Z. Jia); chenxingzhong1125@gmail.com (C. Zhong); cheng.zhang@ahu.edu.cn (C. Zhang); cycle44@gmail.com (Z. Shan); Shen_ting@outlook.com (J. Shen); ali.babar@adelaide.edu.au (M.A. Babar)

ORCID(s):

[1] https://spring.io/projects/spring-cloud

[2] https://github.com/Tencent/Tars

Table 1
Comparison between the three systematic mapping studies related to MSA and our SLR

Study	Time span	Sample size	Research questions	QAs	Tactics
[24]	-2015	21	RQ1: What are the main practical motivations behind using microservices? RQ2: What are the different types of microservice architectures involved? RQ3: What are the existing methods, techniques and tool support to enable microservice architecture development and operation? RQ4: What are the existing research issues and what should be the future research agenda?	Scalability, Deployability, Maintainability, Interoperability, Reusability, Performance, Reliability, Flexibility	✗
[2]	-2016	33	RQ1: What are the architectural challenges that microservices systems face? RQ2: What architectural diagrams/views are used to represent microservices architectures? RQ3: What quality attributes related to microservices are presented in the literature?	Scalability, Independence, Maintainability, Deployability, Reliability, Modularity, Manageability, Performance, Reusability, Portability, Agility, Security	✗
[11]	-2016	71	RQ1: What are the publication trends of research studies about architecting microservices? RQ2: What is the focus of research on architecting microservices? RQ3: What is the potential for industrial adoption of existing research on architecting microservices?	Performance, Maintainability, Security, Functional suitability, Reliability, Compatibility, Usability, Portability	✗
Ours	-2018	72	RQ1: What are the most concerned QAs for MSA? RQ2: What tactics have been proposed or discussed to improve the most concerned QAs of MSA?	Scalability, Performance, Availability, Monitorability, Security, Testability	✓

eration of MSA and the full stack platform [16] that can create microservices applications including some components for addressing certain quality attributes. Quality Attributes (QAs) are inevitably discussed in the migration practices from monolithic applications to MSA [13, 14], for instance, *maintainability*, *reusability*, and *scalability*.

However, based on our investigation, there is no relatively systematic panorama of the state-of-the-art of the QAs of MSA yet. Many aspects are still unclear, unexplored, or even with inconsistent understanding, for example, how QAs are impacted in MSA and which QAs are the most challenging to be addressed in MSA. Furthermore, there is an apparent need for a broad and systematic analysis of the tactics to improve specific QAs [11].

In contrast to the rapid and widespread adoption of and migration to MSA in the software industry, the shortage in the comprehensive understanding on the QAs of MSA turns to be significant. Hence, we performed a Systematic Literature Review (SLR), which characterizes the existing researches on the QAs related to MSA and provides a panorama for researchers and practitioners. There are two main contributions of this study: i) It identifies a list of the most commonly concerned and addressed QAs for MSA from the 72 reviewed studies; ii) It obtains an initial holistic set of findings to the concerns on these QAs in terms of improvement tactics.

The major target audience of this study is the practitioners with particular interest in specific QAs as well as effective tactics for migration and implementation of MSA of higher quality, and the researchers who expect to identify potential research opportunities in this area.

The remainder of this work is structured as follows. Section 2 presents an overview of the related work. Section 3 describes the research method. Section 4 presents the overall results of this study, including demographics, quality scores and the discussion on our first research question. Section 5 reports the findings on our second research question. Section 6 discusses the main findings identified in our research and explains the threats to validity of our SLR. Section 7 draws the conclusion and suggests the future work.

2. Related work

Most of the existing studies on MSA focus on architectural principles and application of the architectural patterns [19, 15, 4] in microservices migration practices, which can provide an analytic view of common patterns and practices used for MSA and can be considered as a starting point of our work.

In addition, with the increasing interest in microservices, some systematic mapping studies related to MSA have been recently reported. Table 1 shows the comparison between three systematic mapping studies related to MSA and our SLR from the perspectives including the time span of search, the number of selected studies, research questions, QAs identified, and discussing tactics or not.

Pahl et al. [24] conducted a systematic mapping study on the motivation, type, techniques and challenges of microservices. Focusing on taxonomically classifying and comparing their 21 identified studies (till 2015), they proposed a characterization framework for microservices. They found that the maturity of research on microservices was so low that they could not reach a conclusive analysis based on their studies. As a result, research trends and some benefits of microservices were presented. While some keywords on quality were extracted from the reviewed studies, Pahl et al. did not specifically report the QAs of MSA. Alshuqayran et al. [2] reported another systematic mapping study on microservices. Aimed at presenting an overview of the existing challenges, architectural diagrams/views and QAs, 33 relevant studies (till 2016) were identified and analyzed in their review. Similar to Pahl's mapping study [24], this study only provides a list of some QAs with little discussion about their characteristics in microservices-based systems, the tactics and the evaluation methods. Francesco et al. [11] performed a systematic mapping study on the trends, focus, and potential directions of MSA. A pre-defined classification framework was applied to the 71 identified studies (till 2017). A clear overview of the existing research, which can be useful to both researchers and practitioners, was given. The authors also investigated the QAs addressed by the identified studies, which is similar to what we did in our study. These related work confirmed the commonly so-called advantages of MSA, *scalability*, *deployability*, *maintainability*, *performance*, etc. However, they merely presented an overview of QAs for microservices-based systems without discussing the addressing tactics. Different from these mapping studies, we extended the time span of the search to the first half of 2018. Furthermore, we investigated the characteristics of the QAs as well as the tactics to address them in this SLR rather than just presenting an overview of the existing research.

There are also some empirical studies concentrating on the QAs of architectures in other domains. For instance, Mahdavi-Hezavehi et al. [21] assessed methods to handling variability in QAs of service-based systems, collected evidence of the current research that suggests the implications for practice, and identified open problems and areas for improvement. Garcés et al. [12] provided a broad, detailed panorama about the state of the art on quality models and QAs that are important in the domain of Ambient Assisted Living (AAL). Mahdavi-Hezavehi et al. [20] reviewed the current state of research regarding architecture-based methods to handling multiple QAs in self-adaptive systems, and identified the areas for improvement in the future. Ullah et al. [27] identified the most frequently reported quality attributes and architectural tactics for Big Data Cybersecurity Analytics (BDCA) systems through reviewing 74 papers. They found 12 most frequently reported quality attributes for BDCA systems and 17 architectural tactics for addressing part of the identified quality attributes, *performance*, *accuracy*, *scalability*, *reliability*, *security* and *usability*. Although the domains of these studies are different from ours, the research questions about the similar topic and the way they conducted the research and analyzed the data are valuable to be studied with respect to microservices architecture.

3. Research methodology

This study was initiated at the beginning of 2018, and strictly followed the SLR's guideline [18]. This section describes the research method applied in this study.

3.1. Research questions

To obtain a systematic understanding of QAs associated with MSA, the following two research questions are defined to drive this SLR:

RQ1: What are the most concerned QAs for MSA? This question aims to identify the most frequently emphasized QAs as well as their challenges related to MSA.

RQ2: What tactics have been proposed or discussed to improve the most concerned QAs of MSA? This question is to explore the possible tactics of the most concerned QAs for MSA identified in RQ1 from the perspectives of introduction, motivation, description, constraint, and dependency.

3.2. Roles and responsibilities

The review team is composed of: 1) four research students, one PhD student and three master students; 2) three supervisors, who both have rich experiences in the domain of software architecture and the conduction of SLRs; 3) one industrial expert, who has abundant experience with MSA design and development. Specifically, the PhD student took main responsibility of the review protocol development (from forming the research questions to establishing the data extraction schema); then the three master students joined in the rest of the review process. The entire review process was under the guidance by the two supervisors, and the students' outcomes at each review stage, i.e. primary studies selected or data extracted, were randomly cross-checked by their supervisors as well as the industrial expert. The review team held regular meetings to check the process and discussed any divergences or emerging issues over search strategy, study selection criteria, data extraction, data synthesis approaches, etc. until consensus was reached.

3.3. Search strategy and process

In order to retrieve a comprehensive set of research studies that are relevant to QAs of MSA, it is important to achieve an acceptable trade-off between the coverage of the existing research on the topic studied and a manageable number of studies to be selected, which were further analyzed in detail. During the search process, we applied the Quasi-Gold Standard approach [28] which systematically integrates manual and automated search strategies. This approach not only tries to balance search recall and effort but also suggests a relatively rigorous means to evaluating search strategy and strings in terms of sensitivity and precision, as shown in the following equations respectively. Note that the "Number of Relevant Studies Retrieved" in the two equations refers to the "quasi-gold standard (QGS)" proposed in the study of Zhang et.al. [28], which is a representative sample set of known studies from the related venues, e.g., domain-specific conferences and journals recognized by the community in the subject, for a given time span. The "Total Number of Relevant Studies" and "Number of Studies Retrieved" respectively refer to the number of relevant and all papers retrieved from the digital libraries that are searched using the strings.

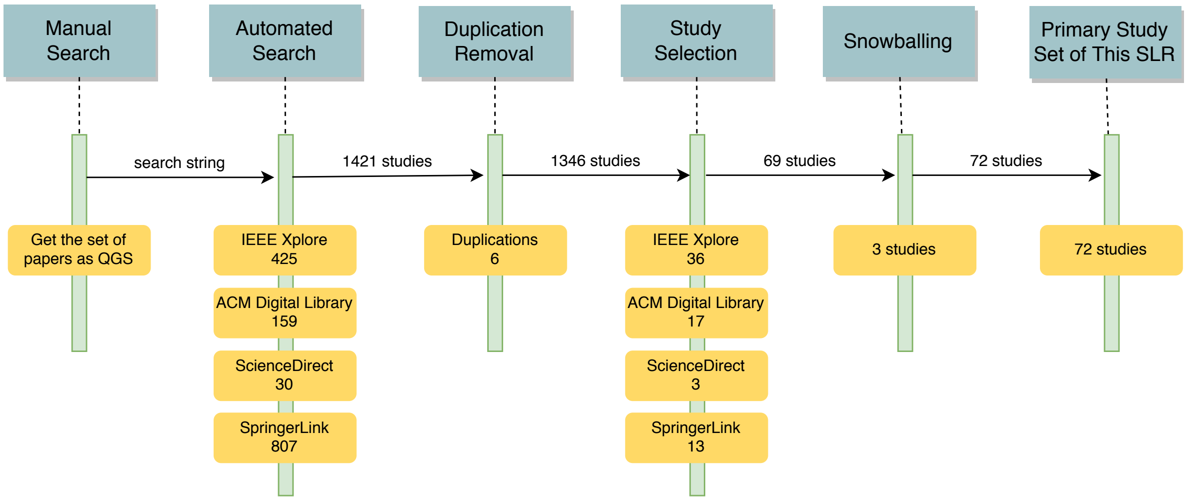


Figure 1: Study identification process

$$\text{Sensitivity} = \frac{Number\ of\ Relevant\ Studies\ Retrieved}{Total\ Number\ of\ Relevant\ Studies} \times 100\% \quad (1)$$

$$\text{Precision} = \frac{Number\ of\ Relevant\ Studies\ Retrieved}{Number\ of\ Studies\ Retrieved} \times 100\% \quad (2)$$

As shown in Figure 1, the search process consists of manual and automated search phases in sequence. In the manual search, the researchers manually checked the papers published at the International Conference on Software Architecture (ICSA), which is the best known conference related to software architecture in general and microservices architecture in particular, and finally identified four papers to be relevant. As a result, we constructed these four relevant studies as the QGS of this SLR. Then the preliminary search string was designed with reference to the previous systematic reviews [11, 2, 24] on microservices and the suggestion from the supervisors and the industrial expert. In order to capture as many relevant studies as possible, the search string concentrates on the peer-reviewed literature on microservices in generic fields, rather limited to Software Engineering (SE). Then the search string was further coded into the equivalent forms to match the specific search syntax of IEEE Xplore, ACM Digital Library, ScienceDirect, and SpringerLink. The search string for automated search is as follows:

TITLE-ABSTR-KEY (microservice **OR** micro service* **OR** micro-service*)*

Note that there exist many different terms or their combinations (e.g., concerns, quality concerns, quality properties, QAs, quality requirements, and non-functional requirements) that can refer to QAs in the literature. Moreover, in many cases the specific quality attributes (e.g., *performance* or *reliability*) might be the only clue of QA in the relevant papers. Therefore, to avoid missing any relevant studies, we used the search string in a generic form and removed any terms related to QA as recommended by Mahdavi-Hezavehi et al. [21]. Moreover, we applied the snowballing strategy in the search to ensure that we could search relevant studies.

In the automated search, with the search string defined, we used four major publishers' digital library portals in SE to retrieve the studies relevant to MSA. We restricted the search timespan initially by June 30, 2018. The retrieved studies from IEEE Xplore (425), ACM Digital Library (159), ScienceDirect (30), and SpringerLink (807, full-text search) were combined to become the raw study set including 1421 studies, of which 6 duplications were removed (as shown in Figure 1). After that, we applied the study set of QGS to validate our search string, i.e. check whether the search string was able to retrieve them from the selected literature databases. With the sensitivity of 100% the result of the automated search could be accepted while the search string remains in a relatively simple format. Then we excluded 1346 studies based on the selection criteria (as shown in Table 2) after reading the title, abstract or even the full-text of the studies. Accordingly the sample size of the selected candidate studies decreased to 69. Then we used the snowballing method to explore the references of the 69 selected studies and the studies having the reference including at least one of the 69 studies. We found and selected other three relevant papers, which brought us to the final total of 72 studies for this SLR.

3.4. Study selection

The inclusion and exclusion criteria for the relevant studies (shown in Table 2) were used to assess each study retrieved from the literature databases and make the selection decisions. The selected studies should mainly discuss QA(s), model(s), tactic(s) etc. related to MSA, and address or improve the quality requirements, for example, various "-ilities" attributes (e.g., *availability* or *scalability*), and others (e.g., *performance* or *security*). We also removed the primary studies that only discuss the low-level technical details of microservices without focusing on the quality attributes at architecture level. The selection of the primary studies for further review was jointly double-checked by the research students, and the disagreements were settled by meetings and discussion with their supervisors.

Table 2
Selection criteria

	Inclusion criteria
I1	Studies justify the influence on specific QAs of microservices architecture.
I2	Studies provide the solutions to address QA relevant problems or the QA evaluation in the domain of microservices.
	Exclusion criteria
E1	Papers are not written in English.
E2	Studies are not available as full-text.
E3	Papers are secondary or tertiary studies, e.g., systematic literature reviews.
E4	The study is an editorial, position paper, abstract, keynote,opinion, tutorial summary, panel discussion, or technical report.
E5	Studies only mention microservices in keywords or related work.

Table 3
Data extraction form

#	Field	RQ	Comments
F1	Title	Metadata	N/A
F2	Authors	Metadata	N/A
F3	Year	Metadata	N/A
F4	Venue Type	Metadata	Conference, Journal, Workshop
F5	Domain	Demography	The horizontal domain, Cloud computing and IoT, etc.
F6	Quality attributes	RQ1	List of QAs and the occurrence of QAs
F7	QAs Explanation	RQ1	The detail of QAs of MSA, e.g., challenges
F8	Motivation	RQ2	The motivation for highlighting the quality attribute(s)
F9	Architectural tactic(s)	RQ2	The architectural tactic(s) proposed in the study
F10	Constraints	RQ2	Necessary conditions or possible problems of incorporating the tactic
F11	Explanation	RQ2	Explanation of applying the tactic proposed in the study
F12	Dependency	RQ2	Dependency between the tactic in the study and other tactics
F13	Comments		N/A

3.5. Quality assessment criteria

To analyze the overall quality of each included primary study and the strength of the evidence contributed by a primary study, we established a checklist including five questions (or quality criteria), based on the quality assessment of primary studies proposed by Dybå et.al. [8].

- *Q1*: Is there a clear statement of the aims of the research? Consider: Is there a rationale for why the study was undertaken?
- *Q2*: Was the research design appropriate to address the aims of the research? Consider: Has the researcher justified the design of the research?
- *Q3*: Was the research method implemented in a way that addressed the research issue? Consider: Has the researcher discussed the process or the detail of the methods that were chosen/proposed?
- *Q4*: Is there a clear statement of findings? Consider: Has an adequate discussion/evaluation of the evidence identified or method proposed, both for and against the researchers arguments, been demonstrated?
- *Q5*: Has the limitation or future work been considered adequately? Consider: Did the researcher critically examine the limitation and future work?

Answers to each of the questions can be either yes, to some extend or no, and then numerical values are assigned to the answers (1 = yes, 0 = no, and 0.5 = to some extent). The final quality score for each primary study is calculated by summing up the scores for all the questions. The results of quality assessment criteria are used in the synthesis phase. The scores are used as an indication of primary studies validity, and may be beneficial for practitioners and researchers in the domain of microservices.

3.6. Data extraction

Data were extracted using the data extraction form shown in Table 3. As indicated, the data items (F1-F4) are for the citation purpose only. The other data items are briefly explained in the right most column.

The data extraction was mainly completed by the research students' reading the full-text of each selected study and extracting the data items concerned to answer the research questions. To standardize the data extraction among the reviewers, ten studies were randomly selected for joint extraction as a pilot. All of the four research students extracted the data from the ten studies and checked the extraction results to ensure the consistent understanding of the data items. Next, the rest studies were assigned to the four research students ensuring that each study was reviewed by at least two reviewers and the data was extracted independently. Any disagreements were discussed in pairs and resolved by the consensus meetings with supervisors or by consultation with the industrial expert.

3.7. Data synthesis

The majority of data extracted from the relevant studies in this review is qualitative, e.g., quality attributes, motivations and architectural tactics. To process the qualitative data within their specific context, we adopted the thematic analysis method, which is widely used for qualitative data analysis. Thematic analysis is to identify, analyze and report themes within data set. Following the guideline [7], we conducted thematic analysis with six phases which consist of familiarizing with the data, generating initial codes, searching for themes, reviewing themes, defining and naming themes, and producing the report. When coding, the coding technique from [23] was leveraged, which tries to identify the most basis segment or element of the raw data that is valuable for answering the research questions. The activity of searching for themes happens when all data have been coded and collated. In searching for themes, we analyzed the relationships between different codes and collated relevant coded data within themes.

The synthesis methods were used to derive the aggregated tactic results. After the pilot phase of data extraction (Section 3.6), we carried out pilot synthesis on the selected ten studies by using thematic analysis individually and discussing the synthesis results. On the basis of this, the four research students synthesized the remaining studies independently and jointly checked the results with their pair. Finally, the differences of individuals synthesis result were reviewed and solved by the review team together, until all team members aligned their understanding of the result.

4. Results

4.1. Demography of studies

The data were extracted from the reviewed studies and synthesized to answer the research questions (cf. Section 3.1). This section starts with the descriptive metadata of the studies, then reports on the important findings based upon the (qualitative) data synthesis in response to the research questions in various representation formats.

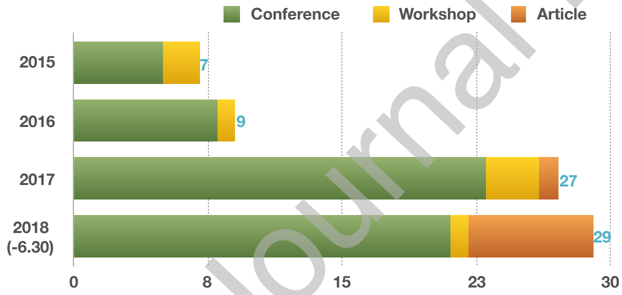


Figure 2: Distribution of reviewed studies per year.

After the search process and study selection (cf. Section 3), we identified 72 studies relevant to QAs of MSA for review. Figure 2 shows the number of the reviewed studies published per year. The first seven studies discussing the QAs of MSA appeared in 2015. After that more researchers were concerned about the QAs of MSA in 2016 (9/72, 12.5%); and the relevant studies were witnessed a three-fold increase in 2017 (27). The number of the relevant studies in the first second half of 2018 is 29, which is already more than the number of relevant studies published in the whole year of 2017 and demonstrates the growth trend of the research on the QAs of MSA.

As shown in Figure 3, more than 47% of the reviewed studies explicate their specific application domains, among which 30 studies are in **Cloud** domain. Other domains are **IoT** (1), **Cloud & IoT** (2) and **Mobile** (1). This demonstrates the obvious relationship between microservices-based systems and cloud.

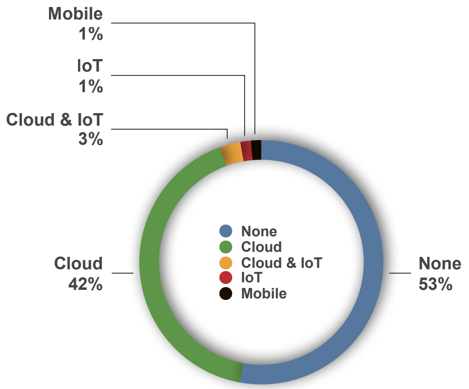


Figure 3: Domains of reviewed studies.

4.2. Quality scores

We evaluate the quality of the primary studies through the quality score, which is obtained by answering the quality assessment questions (cf. Section 3.5). The list of the primary studies and their related scores for each quality assessment question are shown in Table 14 in Appendix A. Each study received a total score between 2 and 5, having intervals of 0.5. The average score of all studies is 3.8 out of 5. Those scores in Table 14 demonstrate the quality of primary studies from different aspects and are not used for the exclusion of a study. Among the five quality assessment criteria, Q5 gains the lowest score on average, which is 0.51, for many researchers did not critically examine the limitation and future work in their studies.

4.3. Demography of QAs in MSA (RQ1)

From the 72 reviewed studies, this SLR totally identified six most concerned QAs which are related to MSA with evidence support to answer the research questions. The consistent terminology is particularly important for the identification of QAs. To ensure consistent meanings of quality attributes extracted from different studies, we used the commonly accepted standards like ISO/IEC 25010 [9] and considered specific attribute definitions provided by Bass et al. [5]. For example, both the explicit expression like "performance" in [Khaz 16] and the implicit expression like "response time" in [Sing 17] are all recognized as the QA of *performance*.

The results about the emphasized QAs relevant to MSA are summarized in Table 4, in which the frequency of each QA discussed in the reviewed studies was counted for ranking and displayed in the bottom row "Total". From the result of frequency counting we can see that *scalability* and *performance* are the two most frequently addressed QAs in MSA related studies, each of which respectively received 23 and 21 hits among the 72 relevant studies. By contrast,

Table 4
Concerned QAs for microservices architecture

Studies	SC	PE	AV	MO	SE	TE	Studies	SC	PE	AV	MO	SE	TE
[Lehv 17]	★			★			[Kook 16]			★			
[Thal 17]	★		★				[Mei 18]			★			
[Toff 15]	★	☆	★				[Pard 18]			★			
[Lope 17]	★						[Sing 17]		☆	★			
[Alip 17]	★			☆			[Furd 18]			★			
[Karw 17]	★	☆					[Kang 16]		★	★			
[Crec 18]	★			☆			[Cele 17]			★			
[Klin 18]	★						[Karg 18]			★			★
[Goti 18a]	★						[Kaki 18]			★			
[Prac 18]	★						[Heor 16]			★			★
[Mazl 17]	★						[Hase 17b]			★			
[Bare 17]	★						[Mont 18]			★			
[Ahma 16]	★						[Stub 15]			★	★		
[Hass 17]	★						[Mora 17]				★		
[Gyse 16]	★						[Maye 17]				★		
[Kryl 15]	★						[Kita 17]			☆	★		
[Chen 17]	★						[John 17]				★		
[Toff 17]	★			☆			[Maye 18]				★		
[Eski 18]	★						[Phip 18]				★		
[Goti 18b]	★						[Hase 17a]				★		
[Liu 18a]	★						[Gan 18]	☆			★		
[Khaz 16]		★					[Sun 15]					★	
[Guer 18a]	☆	★	☆				[Otte 17]			☆		★	
[Ruse 16]		★					[Fetz 17]					★	
[Krat 16]		★					[Wals 17]					★	
[Kloc 17]		★					[Tork 17]					★	☆
[Bham 17]		★					[Jand 18]					★	
[Abad 18]		★					[Yary 18b]					★	
[Zhen 17]		★					[Yary 18a]					★	
[Amar 15]		★					[Arcu 17]						★
[Guer 18b]		★					[Savc 15b]						★
[Lloy 18]		★					[Mein 15]						★
[Grib 18]		★					[Savc 15a]						★
[Kalr 18]		★					[Ma 18]				☆		★
[Fili 18]		★											
[Nico 18]		★											
[Liu 18b]		★											
[Seli 17]		★					Total	23	21	17	13	8	8

1) ★ : Primary QA addressed ☆: Secondary QA(s) addressed
2) SA: Scalability; PE: Performance; AV:Availability; MO: Monitorability; SE: Security; TE: Testability

testability of microservices gained the least discussion in these studies (seven hits). In another dimension from Table 4, we observed that there are 18 reviewed studies addressing more than one QA simultaneously, such as [Karw 17] targeted at both *performance* and *scalability*. Two studies ([Guer 18a, Toff 15]) even addressed three QAs at the same time, i.e. *performance*, *scalability*, and *availability*. In addition, we also identified other two QAs but the claims without essential evidence support, i.e. *maintainability* and *interoperability*. These QAs were only mentioned in the reviewed studies with no evidence for answering our research questions. Therefore they are excluded from the final QAs list.

Summary to answer RQ1. This study identifies six most concerned QAs from reviewed studies related to MSA: *scalability*, *performance*, *availability*, *monitorability*, *security*, and *testability*. *Scalability* and *performance* are the two most concerned QAs addressed in MSA, while *testability* gains the least concern of researchers. The correlation exists among the identified QAs and is considered by some researchers, e.g., the trade-off between *performance* and *scalability* or dependency between *monitorability* and *scalability*.

5. Findings: QAs and tactics (RQ2)

This section reports more critical findings about the six QAs for MSA and the corresponding tactics from the reviewed studies to answer the RQ2. We pay more attention to the tactics specific to QAs in the context of MSA, rather than the ones related to the QAs of software architecture in general, for example, the tactics given by Bass et al. [5]. Correspondingly, we particularly elaborate the MSA-specific tactics in this paper.

We first conceptualize each QA in the context of MSA and then analyze its addressing tactics identified from the primary studies. The tactics were codified using the tem-

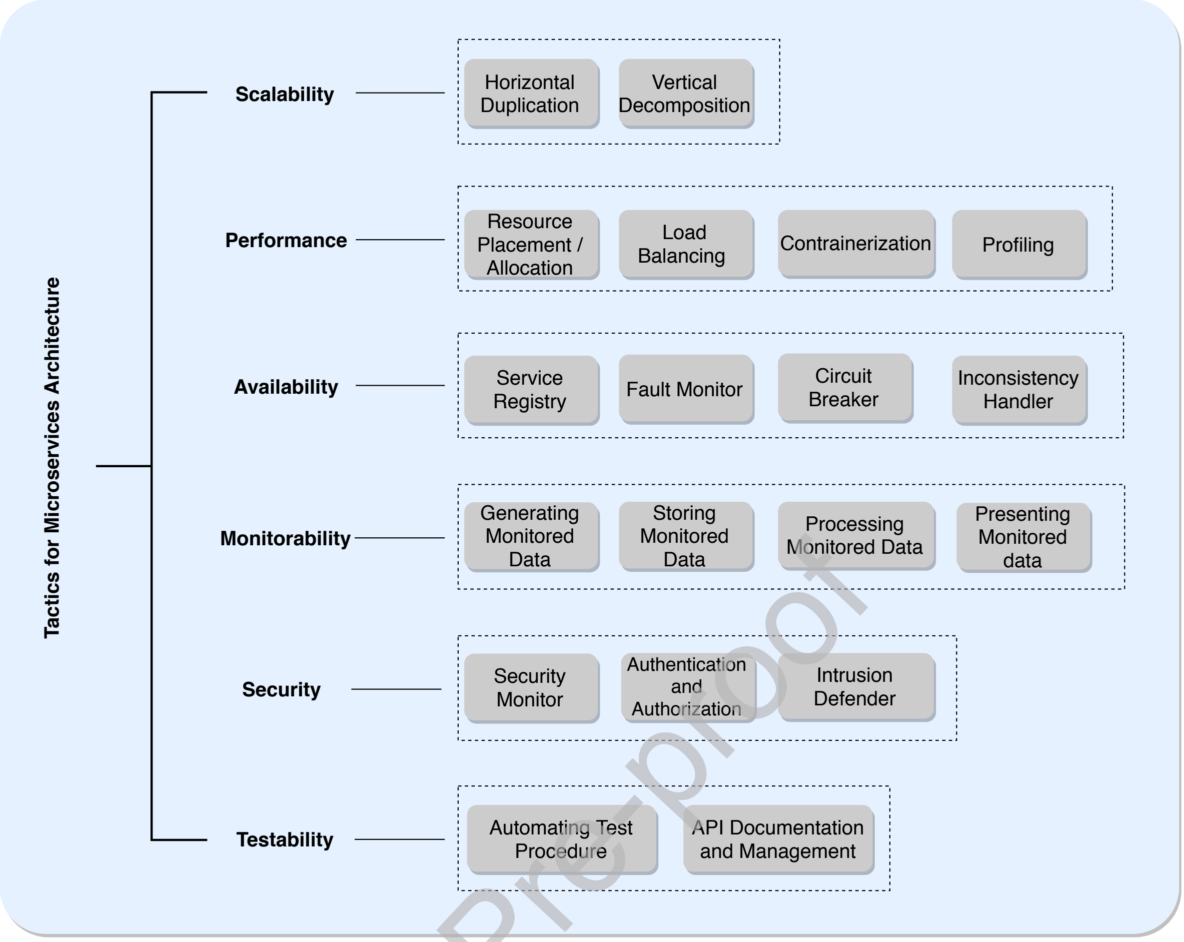


Figure 4: Tactics for microservices architecture

plate, which includes introduction, motivation, description, constraint, and dependency. Figure 4 displays the codified tactics for the six QAs related to MSA.

5.1. Scalability

Scalability is a measure of a system's ability to add resources to handle a varying amount of requests [5]. Horizontal *scalability* (also called elasticity) and vertical *scalability* are two types of *scalability*, which refer to adding more resources to logical units (e.g., servers of a cluster) and adding more resources to physical units (e.g., memory of a computer). Horizontal *scalability* is a most important property of MSA, since it decomposes the monolith into independent microservices and enables microservice instances scale out conveniently. However, we still identified more than 30% of studies devoted themselves to address the problems related to *scalability* (we refer to horizontal *scalability* in this paper), for example, automatically deciding how many instances of each microservice need to be running to grant the desired QoS [Alip 17, Karw 17, Prac 18].

We identified two *Scalability* tactics for MSA from reviewed studies: Horizontal Duplication and Vertical Decomposition, as shown in Table 5.

5.1.1. Horizontal Duplication

Introduction Horizontal Duplication is a tactic for *scalability* that is based on the theory of the X-axis in the Application Scale Cube defined by Abbott et al. [1]. This tactic guides the increase or decrease of microservice instances responding to the changing amount of requests. Solutions of the Horizontal Duplication tactic are identified from 14 studies that fall into two types: Reactive Auto-scaling and Proactive Auto-scaling. There are 11 reviewed studies using the Reactive Auto-scaling solution to address the *scalability* problems of MSA. Most of the reactive methods in these studies use threshold-based rules to decide the moment of scaling microservices. Proactive Auto-scaling is identified from three studies. This solution is similar to Reactive Auto-scaling option, which guides the increase or decrease the number of microservice instances responding to the changing amount of requests.

Motivation The main objective of the Horizontal Duplication tactic is to decide how many instances of each microservice are needed in execution to grant the desired QoS (Quality of Service) [Toff 17]. In front of unpredictable workloads, it is necessary for MSA to react fast and match the supply of microservice instances as closely as possible to

Table 5
Scalability tactics for microservices architecture

Tactics	Studies	#
Horizontal Duplication	[Lehv 17, Thal 17, Guer 18a, Gan 18, Toff 15, Lope 17, Crec 18, Klin 18, Goti 18a, Toff 17, Goti 18b, Alip 17, Karw 17, Prac 18]	14
Vertical Decomposition	[Mazl 17, Bare 17, Ahma 16, Hass 17, Gyse 16, Kryl 15, Chen 17, Eski 18, Liu 18a]	9

the request demand, on the one hand, ensuring the QoS, and on the other hand, minimizing the cost. Therefore, it needs a tactic to quickly react on conditions which reflect the need to scale with high precision in supplying the demand with the adequate capacity of microservices. On the contrary, sometimes the workload of systems can be predicted in advance from historical data. Under this circumstance, tactics can make use of the historical data for the workload prediction and decide whether to scale out or in microservice instances.

Description The two solutions related to the Horizontal Duplication tactic are described as follows:

- *Reactive Auto-scaling:* This Reactive Auto-scaling solution is responsible for scaling microservices in and out based on a threshold-based auto-scaling mechanism. A scaler decides to scale in or out according to an upper and lower threshold as well as the information periodically monitored, e.g., the workload intensity, the CPU utilization, HTTP latency, HTTP throughput or the message queue metrics [Klin 18]. If the monitored information exceeds the threshold pre-defined, the scaler will choose to increase or decrease the instances of the microservice. Since microservices are frequently modified, the scaler should allow for the adaptation process to alter the upper and lower threshold for deciding scaling operations as well as the number of a specific microservice in a particular context [Goti 18a]. Four typical examples were identified to apply the Reactive Auto-scaler solution: Sieve [Thal 17], Dynamite [Toff 17], CAUS [Klin 18], and Bosch IoT Suite [Goti 18a]. They all offer a threshold-based auto-scaling mechanism to scale microservices in and out based on system metrics and application-related information collected, e.g., CPU usage, application response time, and some message queue metrics. The threshold is usually given as a percentage of the collected metric.

- *Proactive Auto-scaling:* The Proactive Auto-scaling solution addresses the scale-out or in through some basic processes [Prac 18, Alip 17]: 1) predicting workload from history to find the number of requests for each microservice that will come to microservices in the future; 2) converting requests to the needed resource (amount of CPU core and memory) to keep microservices working under SLA; 3) matching the supply from the needed resources so as to ensure that enough resources are available and also cost-effective. The optimization algorithm is usually used to find item set that has the most probability and the lowest cost needed for resource package.

Constraint The Reactive Auto-scaling solution has two constraints: 1) Threshold-based rules need to be pre-defined to address under- and over-provisioning problems in this tactic. Rules in this context consist of a condition and an action to be executed. The condition includes a lower and upper threshold for a *performance* metric, based on which an auto-scaler will decide to scale microservice instances in or out [Goti 18a]. 2) Threshold-based rules of the auto-scaling systems should not stay the same at all time, for microservices are usually frequently modified and may impact the *performance*. A microservice developer should have degrees of freedom to configure the thresholds to achieve a specific goal regarding reaction speed, costs or *performance*. The thresholds for each metric can be optimized with a heuristic algorithm using the achieved elasticity as a fitness function to capture the accuracy and timing aspects of scaling decisions [Guer 18a].

By contrast, the workload prediction in the Proactive Auto-scaling solution depends on historical data analysis. It means that the system should have run for a certain period of time to ensure the accuracy of the prediction. New system without historical data acceleration may need some simulation methods to use this tactic [Alip 17].

Dependency The Horizontal Duplication tactic is tightly related to the Vertical Decomposition tactic (cf. Section 5.1.2), which can help to obtain the scalable units (microservices). Moreover, it also relies on the tactics for *monitorability* (cf. Section 5.4) to provide necessary information for some decisions such as the threshold configuration and the scaling operations when using the Reactive Auto-scaling approach. For example, Dynamite [Toff 17] uses its own internal monitoring system allows the application to scale on high-level *performance* metrics (e.g. 95th percentiles) that are computed at timely intervals by the Logstash component and saved to Etcd to be accessed by Dynamite; CAUS [Klin 18] obtains the chosen signal from Prometheus by using the rate function over a one-minute interval.

5.1.2. *Vertical Decomposition*

Introduction Vertical Decomposition tactic guides the decomposition of a monolith into a set of appropriate microservices including databases so as to achieve higher and more independent *scalability*. Nine studies are identified to use this tactic for *scalability* of microservices, which include three kinds of approaches: Logical Coupling, Semantic Coupling, and Contributor Coupling.

Table 6
Performance tactics for microservices architecture

Tactics	Studies	#
Resource Management and Allocation	[Khaz 16, Guer 18a, Ruse 16, Kloc 17, Bham 17, Zhen 17, Guer 18b, Lloy 18, Grib 18, Lope 17, Karw 17, Seli 17]	12
Load Balancing	[Sing 17, Abad 18, Lloy 18, Kalr 18, Fili 18, Liu 18b]	6
Containerization	[Sing 17, Krat 16, Kang 16, Amar 15]	4
Profiling	[Fili 18, Nico 18]	2

Motivation According to the Y-axis of the Application Scale Cube defined by Abbott et al. [1], the *scalability* can be improved through the separation of responsibilities, operation, and data, which means that microservices should be vertically decomposed from the system concerning the effective utilization of resources. In other words, appropriate service decomposition is a prerequisite for *scalability* of MSA [Hass 17, Chen 17]. Consequently, it is of great importance to realize more effective microservice-oriented decomposition for better *scalability* of MSA.

Description Vertical decomposition of a system into microservices includes the following solutions:

- *Logical Coupling:* This tactic uses the Single Responsibility Principle for the microservices-oriented decomposition, which states that the software elements changing for the same reason should be gathered together [Mazl 17]. The goal behind the Logical Coupling is to enforce strong module boundaries determined by the responsibilities or changing behaviors of services. Static code can be used for the Logical Coupling. For example, [Mazl 17] proposed a graph based microservice clustering approach using static and evolutionary coupling between static software classes.

- *Semantic Coupling:* This tactic makes the decomposition decisions according to the notion of bounded context originating in domain-driven design [Mazl 17]. Specifically, the boundary of each microservice is determined by examining the content and semantic of source code files, OpenAPI specifications [Bare 17] or service context and user context [Liu 18a] through information retrieval techniques. By this way, classes containing the same "things", e.g., domain model entities, are integrated together as the potential microservice candidates.

- *Contributor Coupling:* This tactic is to incorporate team-based factors into microservices-oriented decomposition [Mazl 17]. As the vertical decomposition may propose cross-functional teams of developers organized around domain and business capabilities, reducing communication overhead to external teams and maximizing internal communication and cohesion inside developer teams of the same microservice naturally become the main objective during the decomposition procedure. Owners of changes on the class files in version control history of the monolithic system are analyzed to help identify the boundary of candidate microservices.

Constraint When applying the Vertical Decomposition tactic, decisions should be carefully made to find appropriate granularity of microservices for the balance of *scalability* and the *performance*. High cohesion and loose coupling can be considered to decide the size of microservice candidates balancing between the independent *scalability* and the *performance* in terms of the network communication of services [Kloc 17]. Fine-grained and loose coupling microservices may improve the resource utilization and help to save the cost of system due to the independent *scalability* of specific services rather than the whole system. However, it does not mean that the smaller size of the microservice, the better. Too small microservices may lead to the increase of the quantity of services and also the interactions among microservices via network, which may negatively impacted the *performance* because of the longer response time to deal with the more communications of services.

Dependency As mentioned in section 5.1.1, the Vertical Decomposition tactic is a prerequisite of Horizontal Duplication tactic.

5.2. Performance

Performance is a measure of a system's ability to meet timing requirements when responding to an event [5]. *Performance* is another most critical QA for distributed microservices in MSA, which often uses lightweight and REST-based mechanisms for microservices' communication. Table 4 reveals that around 29% (21/72) of studies focused on addressing *performance* problems of MSA. On the one hand, allocating different amount and type of resources (VMs or containers) may influence the throughput or response time of dealing with specific requests [Guer 18b]. On the other hand, the frequency and complexity of the communication between microservices via the network may directly impact the *performance* of systems based on MSA. There is a trade-off between *scalability* and *performance* [Kloc 17]. The smaller size of microservices may increase *scalability*, while it can also decrease the *performance* due to the more interactions among microservices. On the contrary, the *performance* may be improved through merging two microservices and reducing the communication overhead, at the cost of *scalability* to some extent.

Performance tactics for MSA identified from the reviewed studies are divided into three categories: Resource Management and Allocation, Load Balancing, Containerization, and Profiling, as shown in Table 6.

5.2.1. Resource Management and Allocation

Introduction The most commonly used tactic addressing *performance* is Resource Management and Allocation, which operates on the response side to make the resources at hand work more effectively in handling the demands put to them [5]. This tactic is similar to the definition of tactics from Bass et al., for example, the "Increase Resources" and the "Scheduling". More than one half of the related studies (57%, 12/21) tried to improve the *performance* through the management and allocation of resources to achieve optimized *performance* of microservices.

Motivation According to Bass et al., the demand for resources of systems is not controllable [5]. Under this circumstance, effective management of these resources on the response side may be critical for the concern of *performance*. As mentioned in the Introduction, the optimized resource management and allocation can make the resources work more effectively to deal with requests with better *performance* (higher throughput or shorter response time).

Description Table 7 shows the comparative results identified from primary studies using the Resource Management and Allocation tactic. It can be applied to optimize the *performance* of microservices deployed in different infrastructures (single cloud or multiple clouds) with specific objectives, approaches, and algorithms. To use this tactic, a *performance* analytical model based on the behavior of systems is usually designed to perform what-if analysis and resource planning approaches in a systematic manner for large scale microservices with optimized objectives, for example the 3-dimensional Continues Time Markov Chain (CTMC) [Khaz 16] and the Fluid Stochastic Petri Nets (FSPN) [Grib 18]. The *performance* model includes all the servicing steps in fulfilling requests for users or other services and computes some performance objectives (metrics): latency (12/12), cost (5/12), resource utilization (3/12), reliability or repair time (2/12), and others (3/12). Based on the *performance* model, an orchestration engine are necessary for managing and allocating different resources, e.g., microservices or containers (11/12), virtual machines (4/12), or physical machines (1/12), using specific approaches. Since the management and allocation of resources can be characterized as a NP-complete problem and addressed by evolutionary approaches, such as Genetic Algorithm (GA) and its variant NSGA-II ([Guer 18a, Kloc 17, Guer 18b]), which is one of the most common solutions for multi-objective optimization.

Constraint In this tactic, resource management and allocation decisions can be obtained through the combination of relevant variables to specific objectives. To know which is the best combination for each variable, one option is to perform a number of experiments and simulations corresponding to different situations. However, simulations encompassing the different possibilities are time consuming [Kloc 17]. Therefore, efficient approximations of workloads are essential, automating the feedback loop and generation of new microservice architectures based on the current and expected variables of the system. Furthermore, this tactic supports to optimize different objectives, whose effectiveness may be constrained by the algorithms used [Bham 17, Guer 18b].

Dependency Seven out of the 12 relevant studies mentioned that the Resource Management and Allocation tactic relies on the Horizontal Duplication tactic (cf. Section 5.1.1) to scaling and adjusting resources. For example, based on the *performance* model, containers and VMs can be orchestrated by a central manager in [Guer 18b], which was in charge of deciding the scaling level of both containers (number of container instances) and VMs, the allocation of the containers in the most suitable type of VM, and the allocation of the VM in the most suitable cloud provider. Besides, an important requirement for the orchestration engine or manager of this tactic is that all *performance* metrics should allow to be traced back [Kloc 17]. Under this circumstance, tactics for *monitorability* (cf. Section 5.4) are necessary for monitoring, parameter setting, and *performance* measurement [Khaz 16].

5.2.2. Load Balancing

Introduction Load Balancing is a tactic that distributing incoming load on an individual microservice among its many instances. By this way, requests can be fulfilled in a manner that maximizes speed and capacity utilization and ensures that no one service is significantly overworked compared to others, which could dred toegrade *performance*. We found the use of the Load Balancing tactic in six studies.

Motivation In MSA, user requests are served by a series of microservice instances forming a chain. High load on one microservice instance may block the process of requests and decrease the *performance* of MSA-based systems. In other words, changing the load distribution at one microservice instance and making instances to deal with user requests in parallel can help to address the performance challenge of MSA to some extent.

Description In order to implement the Load Balancing, an application should be able to distribute the load on an individual service among its many instances, which is the duty of a Load Balancer. This component achieves an even load balancing through the following two solutions identified from relevant studies:

- *Centralized load balancing:* Centralized load balancing is the most common mechanism to manage the load of microservices. It usually sets up a centralized load balancer, which sits between the client and the services accepting incoming network and application traffic and distributes the traffic across multiple backend services in a real-time way using various push-based algorithms, for example, the Round-Robin algorithm, the JSQ algorithm, the JIQ algorithm [Abad 18], the hybrid scheduling algorithm [Fili 18], or the adap-

Table 7
Comparative results of primary studies using the Resource Management and Allocation tactic

Studies	Infrastructures	Objectives	Approaches	Models/Algorithms
[Khaz 16]	Single cloud	Latency, cost, resource usage	VM and container orchestration	3-dimensional Continuous Time Markov Chain (CTMC)
[Ruse 16]	Single cloud	Latency, throughput	Container orchestration and migration	Swarm-like Algorithm
[Grib 18]	Single cloud	Latency, cost	VM orchestration	Fluid Stochastic Petri Nets (FSPN)
[Lope 17]	Single cloud	Latency, cost	Microservice and container orchestration	Matrix and Formalized Calculation
[Karw 17]	Single cloud	Latency, cost	Container orchestration and migration	Swarm Algorithm
[Bham 17]	Multiple clouds	Latency, inter-VM traffic	Microservice, VM/PM orchestration	Fair Weighted Affinity-based Scheme (FWS)
[Zhen 17]	Multiple clouds	Latency, resource usage	Microservice and container orchestration	Kubernetes
[Guer 18b]	Multiple clouds	Latency, cost, repair time	VM and container orchestration	NSGA-II and Greedy Algorithm
[Lloy 18]	Multiple clouds	Latency	Container size reservation	–
[Seli 17]	Micro-cloud	Latency, average chunk loss	Microservice orchestration	Two phases: K-Means and BASP (Bandwidth and Availability-aware Service Placement)
[Guer 18a]	–	Latency, reliability, resource usage	Container orchestration	Non-dominated Sorting Genetic Algorithm-II (NSGA-II)
[Kloc 17]	–	Latency	Microservice orchestration	Genetic Algorithm

tive SQLB algorithm. The centralized load balancer is usually transparent to the clients and the clients do not know the service list. It is responsible for checking the health of the server pool underneath and use specific algorithms for the load's distribution. Common server-side load balancing solutions [Sing 17] are Ngnix, HA Proxy, etc.

- *Distributed load balancing:* Distributed load balancing is an approach on the upswing in MSA. The load balancer of this approach is composed of a set of schedulers or implemented by the client itself. The distributed load balancer or client gets a list of available servers and then distributes the requests to different servers according to a specific pull-based load balancing algorithms, such as FCFS algorithm [Fili 18, Liu 18b]. Different from the centralized side load balancing, client itself of distributed load balancing holds the list of service IPs that it can deliver the requests, selects an IP from the list randomly or according to a specific method, and forwards the request to certain services.

Constraint Different types of load balancing approaches may have their own constraints. The centralized load balancing requires the load balancer at the server side to save the state of the push to the client side, which possibly becomes a burden of the centralized server when facing with a large amount of clients [Abad 18]. By contrast, the decentralized load balancing can reduce the burden of storage, since the clients can save useful data locally for the backup and recovery [Fili 18]. However, the active and frequent pull of data by clients themselves may not retrieve new data every time, which will cause a waste of bandwidth. Moreover, the coordination among multiple clients is another possible complexity of using the decentralized load balancing.

Dependency This tactic may related to the Profiling tactic in Section 5.2.4 and the Horizontal Duplication tactic in Section 5.1.1, since the main responsibility of the load balancer is applying some given scheduling policies that provide allocation solutions. The load balancer component may modify the information with the workload profiling component and auto-scaling resource manager for the consideration of operation and optimization [Fili 18].

5.2.3. Containerization

Introduction Containerization is an emerging tactic for MSA, which achieves virtualization through containers. We found that four primary studies adopted this tactic for higher *performance* than virtual machines (VMs).

Motivation Virtualization has been the key to increase the *performance* of distributed applications [Sing 17]. Many researchers have worked on performance analysis of different virtualization techniques, among which VM is the most popular one. However, each VM usually has its own guest operating system (OS) and may cause overhead for microservices in different OSs.

Description Docker is the most representative technology implementing the Containerization tactic. It increases the *performance* of microservices from two aspects [Kang 16]:

Firstly, different from VM in their architecture, containers can share the same host OS and require guest processes to be compatible with host kernel. This way can reduce the overhead of communication among different guest OSs for different container processes, can thus contribute for high *performance*, less memory requirement and reduced infrastructure cost. Secondly, the lightweight characteristic of Docker containers could help to create and run more microservices contributing to higher resource utilization. Similar to VMs, containers can also share a single server to increase the resource utilization, which is more efficient than VMs when accessing memory and I/O. In terms of a single-host scenario, containers are able to run more microservices than VMs and ensure the *performance* at the same time. When it comes to the multi-host scenario, the lightweight and fast-booting container images can facilitate microservices scaling in or out easily to adapt the change of workload and thus lead to more efficient resource utilization.

Constraint The study [Kang 16] reveals constraints and open issues of the Containerization tactic in the context of containerizing infrastructure cloud services. As these services need to access some OS-level modules or data, for example, the in-memory state or kernel modules, it will become difficult for the containerization of them without sacrificing portability and security.

Dependency Containers can be the resources that are scheduled when using the Resource Management and Allocation tactic (cf. Section 5.2.1). Moreover, the Fault Monitor tactic in Section 5.3.1 and the Service Registry tactic in Section 5.3.2 achieved by mechanisms such as etcd and fleet, can collaborate with the Containerization tactic to monitor the *availability* of microservices deployed in containers (e.g., shell scripts used by sidekick for probing) and recover microservices from fault (killing failed services and instantiating a new service).

5.2.4. *Profiling*

Introduction Profiling is a tactic that guide the *performance* issues' detection and make the preparation for *performance* optimization. This tactic was applied by two reviewed studies, as shown in Table 6.

Motivation Some fundamental design decisions may impact the *performance* of microservices-based systems, e.g., the granularity of microservices, or the deployment environment selected. Tactics are necessary to help a system architect to address the sources of *performance* unpredictability across an end-to-end microservice's critical path and figure out these impacts, bottlenecks, as well as the potential optimization when designing a microservice.

Description Profiling is a tactic that analyzes the characteristics of a microservices-based system including the required memory, CPU, or bandwidth, and provides both scheduling engine and auto-scaling resource manager with essential information to optimize the *performance* [Fili 18]. Two primary solutions related to this tactic are CPU and memory profiling. Traditionally, CPU profiling is to analyze the execution time of a service and can find service "hotspots" that need to be optimized. Memory profiling is to analyze the memory state or memory allocation events. It is often used not only for finding memory leaks, but also for optimizing memory usage [Nico 18]. Netflixs Vector and Googles Google-Wide Profiling are typical frameworks to support the Profiling tactic at a system and application level and a host level respectively.

Moreover, Profiling tactic for MSA also focuses on diagnosing *performance* problems through on-demand requests rather than monitoring the state of systems, for example the ODP framework [Nico 18]. Other similar frameworks identified are ZProfiler and ZDebugger from Alibaba, which provide features including profiling and sharing of detected issues. Key components for the diagnosis purpose of the Profiling tactic may consist of a Profiler request, a Profiler API server, a Profiler service with profiler plugins, a messaging backbone, and a data pipeline and datastore [Nico 18].

Constraint To some extent, employing lightweight profilers such as Linux Perf can enable an "always-on" approach and support continuous *performance* assurance [Nico 18]. However, the *scalability* of solution for the Profiling tactic should be considered and addressed, since the "always-on" approach may also significantly increase the cost of data processing and storage. Then it is usually challenging to implement the profiling of embedded devices, e.g., mobile devices. Although there are some existing profiling tools for devices, emulators, and simulators, it needs much effort to adapt them as a Profiler plug-in. Furthermore, some technologies for virtualized or containerized services would require further permissions or processes to ensure the visibility of STP to the Profiler service and its plug-ins.

Dependency The Profiling tactic is very important for the *performance* predictability of a microservices-based system. Based on the necessary information collected by the workload profiling module, multiple policies and algorithms in the tactic Horizontal Duplication (cf. Section 5.1.1) or Resource Management and Allocation tactic (cf. Section 5.2.1) could be used to make a decision about the time and the resources of executing microservices.

5.3. Availability

Availability measures the ability of a system to repair faults so that the period of a service with the unavailable state does not exceed a required value [5]. According to the definition of Bass et al., *availability* is a broad perspective and encompasses what is normally called *reliability* [9]. Therefore, we categorized the reviewed studies related to *reliability* into the *availability* group. It is observed that 17 reviewed studies highlight the importance of *availability* for MSA systems, for the distributed microservices may be vulnerable to many failures and difficult to make the root cause analysis because of the complex calling relationship between microservices [Kita 17, Kang 16, Cele 17, Kaki 18, Toff 15].

Table 8
Availability tactics for microservices architecture

Tactics	Studies	#
Fault Monitor	[Mei 18, Kita 17, Kang 16, Cele 17, Kaki 18, Toff 15, Ma 18, Kryl 15, Stub 15]	9
Service Registry	[Kook 16, Mei 18, Sing 17, Otte 17, Tork 17, Ma 18, Hase 17b, Mont 18]	8
Circuit Breaker	[Otte 17, Tork 17, Mont 18]	3
Inconsistency Handler	[Pard 18, Furd 18]	2

Availability tactics are categorized into four types and relevant studies applying these tactics are shown in Table 8. We introduce the tactics for *availability* of MSA obtained from the reviewed studies as follows.

5.3.1. *Fault Monitor*

Introduction The Fault Monitor tactic belongs to the category of Detect Faults defined by Bass et al. [5] and is identified from nine reviewed studies. Compared with tactics in the other two categories Recover Faults and Prevent Faults [5], the Fault Monitor tactic is more commonly used for discovering the faults of microservices through continuous health monitoring using a specific component.

Motivation For the *availability* concern of microservices, it is necessary to detect or anticipate the presence of a fault before the system can take action to recover from faults. In order to achieve high *availability*, microservices-based applications require the continuous monitoring, so that their health can be analyzed to automatically and responsively react to failures with minimal human intervention.

Description For the Fault Monitor tactic in this category, three kinds of fault monitor for microservices-based systems are frequently used in the reviewed studies:

- Centralized Monitor: A centralized monitor collects the results of service invocation based on the health checks by the service discovery mechanism and updates the health status of the service instances in the service registry, which can minimize the downtime of any microservice during the update or recovery from failures [Kook 16, Mei 18, Sing 17, Kang 16]. Centralized monitor is usually implemented using a server-side style [Mont 18]. For example, a health monitoring mechanism [Kang 16] for *availability* was designed based on the Centralized Monitor solution implementing a generic method, i.e. shell scripts used by sidekick for probing.
- Symmetric Monitor [Kaki 18, Cele 17]: The monitors of a microservice in the Symmetric Monitor approach are the neighborhoods of the service itself, including successors and predecessors. For example, an IoT Watchdog [Cele 17] implements this technique that allows the neighborhood monitoring, in which the monitoring relationships are purely symmetric. For this case, the failures of a specific IoT devices are continually monitored by its neighbour IoT devices. The broken microservice instance in an IoT device will be repaired or replaced by a replica in another IoT device.
- Arbitral Monitor [Kaki 18]: An arbitrator acts as the role of detecting failures or conflicts of microservices, which is implemented as a decentralized and independent group of nodes. The failure detected by any node needs to be confirmed by a majority of nodes in the arbitrator group. Cascading failures can be addressed by this method and the number of nodes leaving the system can be reduced.

Constraint. The Centralized Monitor solution requires the deployment of an additional service, which will consume resources and may cause the single point of failure. For the rest two Fault Monitor approaches, Symmetric Monitor has the potential to create inconsistencies, however the use of the arbitrator group can keep the membership lists consistent [Cele 17, Kaki 18].

Dependency The Fault Monitor tactic is tightly related to the Service Registry tactic in Section 5.3.2, which is a mechanism that stores the location of all running services and employs a periodic message communication between a system monitor and a microservice being monitored.

5.3.2. *Service Registry*

Introduction Service Registry is a tactic that stores the location of all running services at a central point and can be used by other microservices to retrieve binding information about other microservices in MSA. This tactic was adopted by nine reviewed studies.

Motivation Microservices may have multiple instances deployed and the number of microservice instances can be changed over time. Therefore, a participated service may need a way to locate the position of a specific microservice to detect the faults with high efficiency. As referred to the idea for the Service-Oriented Architecture (SOA) [Mei 18]. registering a service may also be helpful for this purpose.

Description A service registry can be a service that automatically registers microservices and store their addresses after the startup. Then it can also remove an instance from the list when the periodic heartbeat information is not received from the instance [Mont 18]. Some companies think of a self-implemented custom solution for the client libraries. The reason for this is that they need a specific system to include services of other vendors automatically, and they also need to include services of their previous system, which was running in parallel [Mont 18]. Apart from that, we identified three approaches for the Service Registry tactic [Hase 17b]: self-registry, third-party registry, and manual registry.

- Self-registry: Through this way, each service can register and deregister itself using a local registry once it is successfully started or shut down. The implementation of this approach needs a service registry storing the locations of microservice instances and a client library registering the microservice with the registry.
- Third-party registry: This approach is implemented using a third-party registry service, which can achieve the automatic registration or deregistration of microservices. The third-party registry contains a registrar component and an additional service registry storing the locations of services and does not need the service specific library any more. Additionally, it allows checking the *availability* of services thanks to the automatic registration or deregistration of services through the registrar component. Typical cases for the service registrar component of this method are Registrator and Prana from NetflixOSS.
- Manual registry: The registration or deregistration of microservices are achieved manually at the service registry. Compared with other two approaches, this approach supports better interoperability, since it supports the users to include microservices from other systems based on their needs. However, manual registry may cause long time of responding to microservices with failures for it can not check the health of services in time. Therefore, this solution is only suitable for the systems whose microservices have been available and stable for a long period.

Constraint This tactic requires that the microservices-based system should be coupled to the service registry component for the communications among each other [Hase 17b]. If the service registry has some faults, it will become a single point of failure. In addition, the implementation of clients should adapt to all programming languages used.

Dependency This tactic can collaborate with the Containerization tactic in Section 5.2.3 and the Fault Monitor tactic in Section 5.3.1 achieved by mechanisms like etcd and fleet. The collaboration among these tactics can monitor the *availability* of microservices deployed in containers and better support the recovery of microservices from faults.

5.3.3. Circuit Breaker

Introduction Circuit breaker is a tactic that prevents requests to a service in case of a failure. When facing with a failure, the circuit breaker opens depending on a specific threshold. After a period of time pre-defiend, the circuit breaker closes again and will not open until errors are detected. This tactic is adopted by three studies [Otte 17, Tork 17, Mont 18].

Motivation Fault tolerance should be embedded in a microservice system to ensure its high *availability*, since many services with dependencies need to work together and failure of one service may cause the cascading failure of the downstream services and even the failure of the whole system. Leveraging tactics like Circuit Breaker may help to minimize the loss caused by cascading failures [Tork 17].

Description A circuit breaker, whose behavior is determined by a state machine, can intercept a target service with faults. The mainstream implementation of the circuit breaker tactic is Hystrix. There are three states for such a circuit breaker, i,e. Closed, Open, and Half-open. When a circuit breaker is in the “Closed” state, requests are allowed to be passed to the target service. Otherwise, if the count of faults and timeouts caused by the requests passes a threshold or a critical fault is detected, the circuit breaker will transition into the “Open” state and prevent requests from being passed to the target service. After a periodic observation of the service’s health, the “Half-open” state may be triggered to let the pass of the target service in limited number. Circuit breakers can be usefully implemented through three different solutions [Mont 18] as follows in terms of which party implements the breaker:

- Client-side Circuit Breaker. This approach implements a separate circuit breaker for clients to intercept requests from a service to other external services and prevent the targeted service from receiving messages. Clients are forced to use circuit breakers when adopting this method, and it needs the access to the source code of clients. Besides, clients can be malicious and the information about a service’s *availability* is local to the client. All clients might regularly ping targeted services to obtain the information about their health.
- Service-side Circuit Breaker. This approach needs an internal circuit breaker that decides whether a client requests received by a service should be processed or not. It requires resources to execute the service-side circuit breaker and receive messages even when the circuit breaker is open. Through this method, services can check the aggregated information about their responsiveness that includes requests from all clients, which can support the throttling of requests based on overall performance.
- Proxy Circuit Breaker. This approach uses a proxy service sitting between clients and services to deploy circuit breakers. The proxy can be single for multiple services or can also provides a circuit breaker for each client and service of a system. Only when the circuit breaker of both client and service is closed, a request can go through from the former to the latter. One single proxy for many services may become a *performance* bottleneck. By contrast, a proxy circuit breaker for each target service ensures that clients and services are equally protected from each other, since services are shielded against cases in which a single

client sends too many requests and clients are made more resilient against faulty services.

Constraint The Circuit Breaker tactic requires an additional request and response to perform the handshaking mechanism in front of each communication [Mont 18]. It is challenging to recognize the appropriate response in the open circuit state and coordination should be carried out with relevant stakeholders. Besides, if the response in open circuit state is not merely an exception, then the possibility of returning that response should be certified by the service provider team [Mont 18].

Dependency No Data.

5.3.4. Inconsistency Handler

Introduction Inconsistency Handler is tactic that deals with the trade-off between the consistency and *availability* of a microservices-based system. It ensures the *availability* of distributed MSA systems through using the eventual rather than the instantaneous consistency. Two studies were observed to apply this tactic in their work.

Motivation Consistency is a property of distributed systems by which users will get the same answer if they go to multiple nodes. Multiple instances of a microservice in a distributed MSA-based system interact in parallel with a data repository, creating data consistency challenges. According to the CAP theorem, consistency, *availability*, and partition tolerance of a distributed system are traded off against each other and only two of them can be kept in a failure mode. For a microservice-based system that needs to be more available (*availability*) and keep running over the network (partition tolerance), the consistency has to be lost. Therefore, a tactic is necessary to deal with the possible possible inconsistency in MSA when achieving its *availability*.

Description Inconsistency Handler is commonly adopted to deal with unreliable and unstable networks with limited bandwidth or high data volume, in which it is unfeasible to simultaneously access multiple replicas in MSA. In other words, read/write operations can only be conducted on one replica each time, or may lead to temporary data inconsistencies. The Inconsistency Handler is a tactic to address this problem through synchronization of the data replicas finally and achieve the eventual consistency of data. This tactic requires a business event storage to manage changes applied to the state of microservices. The business events of microservices can be stored in their databases using logs and can be replayed to update microservices to the latest version. Some of the events in the log are local or conceptually cross the boundary between two microservices. For inter-service communication, asynchronous messaging technology such as Kafka or RabbitMQ are usually used, which implements different publish-subscribe techniques to deliver the events to the corresponding subscribers. The implementations are based on an eventual consistency model eliminating synchronous requests from one microservice to the other using a local cache [Furd 18].

Constraint The eventual consistency may cause problems when a backup of the entire application is being performed. It is likely that the state of the application will not converge and will remain inconsistent during the backup due to three reasons [Pard 18], e.g., broken link, orphan state, and missing state. There should be further tradeoff between the availability and consistency of a microservice-based architecture for this situation. Moreover, before actually building the microservice ecosystem, an appropriate disaster recovery plan, dealing with how to perform backups (independently for individual microservices or globally for the entire system), and dealing with the consequences of restoring a system from potentially inconsistent backups (broken links, orphan state, and missing state), can avoid many headaches afterwards.

Dependency No data.

5.4. Monitorability

Monitorability is a measure of a system's ability to support the operations staff to monitor the system while it is executing [5]. Monitoring becomes an essential but more complex part of microservices-based systems due to the high level of dynamic structure and behavior of such a system, therefore it was highlighted by 13 reviewed studies (Table 4). The monitoring of a microservices-based application can be infrastructure information (e.g., VM or container), application information (e.g., response time), and environment information (e.g., network) [Maye 17, John 17, Maye 18, Alip 17, Crec 18, Hase 17a]. Improving *monitorability* may impact other QAs of a system with MSA style, e.g., the *scalability*, *performance*, and *availability*. *Monitorability* tactics are categorized as four kinds of approaches as shown in Table 9.

5.4.1. Generating Monitored Data

Introduction Generating Monitored Data is a tactic adopted in all of the 14 reviewed studies addressing the *monitorability* of microservices. This tactic guides the generation and the collection of monitored data at different levels, such as host, platform, and service, by solutions like instrumentation, logging, and distributed tracing.

Motivation The first area of *monitorability* addressing is to prepare and collect the monitoring data of MSA with complex relations among microservices and the changing behavior of the whole system. The data generated at different level relates to different concerns and requirements. Monitoring host, platform, and service metrics can help to know the runtime information of microservices from different aspects, e.g., available hosts, service response time, failure rate, resource consumption. They are important when telling the health state of microservices and doing some scheduling decisions related to *performance*. Moreover, the challenge of locating the root cause of a failure in distributed MSA requires monitoring and analyzing the interaction and calling relationship between services.

Table 9
Monitorability tactics for microservices architecture

Tactics	Studies	#
Generating Monitored Data	[Lehv 17, Mora 17, Maye 17, Kita 17, John 17, Maye 18, Gan 18, Toff 15, Alip 17, Crec 18, Ma 18, Toff 17, Hase 17a, Stub 15]	14
Storing Monitored Data	[Lehv 17, Maye 17, Gan 18, Maye 18, Toff 15, Alip 17, Crec 18, Ma 18, Hase 17a]	9
Processing Monitored Data	[Maye 17, Kita 17, John 17, Maye 18, Toff 15, Alip 17, Toff 17, Hase 17a]	8
Presenting Monitored Data	[Lehv 17, Maye 17, Crec 18, Toff 17, Ma 18, Hase 17a]	6

Description We identified three solutions, i.e. instrumentation, logging and distributed tracing, from relevant studies to generate and collect the required data. Instrumenting services is a prerequisite to collect the static and runtime information for each service at different levels, e.g., service-, interaction-, and infrastructure-related information. Distributed tracing specifically focus on collecting the call dependency among service. The none-instrumentation solution logging is a common way to record this information for further processing.

- Instrumentation: There are three main concerns related to Instrumentation tactic, which are host instrumentation, platform instrumentation, and service instrumentation [Maye 18, Hase 17a]: 1) Host instrumentation requires an agent for collecting runtime-related data to be installed at each host of a microservice runtime infrastructure. This agent automatically detects new microservice instances and starts collecting monitoring data. 2) Platform instrumentation means that each microservice's runtime environment, e.g., each Tomcat or MySQL server, is instrumented. This requires an agent for each platform used at a specific host. The agent implementation is independent from the operating system and it can be more technology-specific than an agent at the host level. 3) Service instrumentation requires an agent for each service. This allows the analysis of service runtime metrics, and service interactions, but provides no information about the service runtime environment.
- Logging: Through logging, all outgoing and incoming requests of the microservice are recorded to a local log file for the microservice instance. Each log entry may represent a specific request, comprising a timestamp, response time, response code, a unique ID of the source microservice instance, the URL of the target microservice instance, and the requested method [Kita 17, Crec 18, Toff 17]. These log files are periodically fetched and aggregated by a specific logging framework such as CloudWatch [Maye 17, Alip 17, Crec 18] and Logstash [Toff 15, Toff 17] in a configurable interval for further analysis.
- Distributed Tracing allows to determine the initiator of a sequence of microservice calls and is thus a means for root cause analysis. It is a method for just logging information on which service (or service instance) called another service (instance) [Ma 18]. The number of incoming and outgoing calls per service are stored during this process. Then this information can be combined with information about a service API and be used for determining call frequency on the level of service instances or service types (if aggregated). It also allows the analysis of communication frequency between different services. Existing tools for distributed tracing are New Relic, Dynatrace APM, Zipkin, OpenTracing, Apache HTrace, etc [Hase 17a, Kita 17].

Constraint The implementation of those above-mentioned approaches for the Generating Monitored Data tactic needs the support of additional technology stacks. For example, as described in the study [Maye 18], the effort required to support additional technology stacks depends on a specific stacks support for implementing the instrumentation for service calls and may cause an additional burden for developers.

Dependency Generating Monitored Data is the most basic tactic to make the monitoring of microservices-based systems easier. The data generated and collected set up the bridge between it and other tactics of *monitorability* or specific tactics of other QAs. For example, distributed tracing refers to the need for information for analyzing communications and interactions between services. This is typically a prerequisite for root cause analysis, which needs to identify the root cause in case of the failure of a microservice and improve the *availability*.

5.4.2. Storing Monitored Data

Introduction We identified Storing Monitored Data as an important tactic for the *monitorability* of a MSA-based system, which was used by nine studies. This tactic is to manage the data generated and collected by the aforementioned tactic in the last sub-section.

Motivation After the generation of monitoring data, storing and managing them is the next vital step for further processing. Storing data in one place is necessarily required, for collecting data from several distributed places (e.g., nodes in a system) before processing requires additional time for data collection. The management of the monitored data may have different options in terms of different concerns and requirements.

Description This tactic contains two options that provide different ways to store the generated data for the following processing activities, i.e. centralized or decentralized.

- Centralized storage: With the centralized storage, data of all running microservices generated and collected are stored at one place, usually using central storage component [Maye 17, Maye 18, Crec 18]. It allows the analysis of service interactions in a centralized style with less administration overhead, since the deployment and operation of the storage has to be performed only once. However, the centralized storage might become a bottleneck and also a single point of failure especially in large systems.

- Decentralized storage: The decentralized storage option is based on a local component at each host, platform or service to store the monitored data [Hase 17a]. Through this way, it is possible to analyze service interactions, which is a critical requirement in MSA. This solution has higher scalability, since installing new services at new hosts will also provide a new local storage. While the unavailability of a local storage (e.g., due to a crashed host) in this design will result in a loss of monitoring data. Furthermore, operating many local storages needs a lot of administration effort.

Constraint As mentioned above, the Storing Monitored Data tactic with different options may all bring some side effects: 1) the scalability bottleneck and the single point of failure for the centralized storage; 2) the potential risk of the unavailability of a local storage and the administration effort for the decentralized storage.

Dependency The Storing Monitored Data tactic is related to the Generating Monitored Data tactic, since the former tactic can be used to store the data generated and collected by the latter tactic. Moreover, the data stored can also be further processed, analyzed, or visualized by other tactics that are introduced later.

5.4.3. Processing Monitored Data

Introduction Processing Monitored Data is another tactic applied by eight reviewed studies to address *monitorability*. This tactic contains approaches processing the monitoring data in different ways regarding specific objectives.

Motivation Analyzing and processing different data monitored and collected have different aims, for example, collecting all microservice calling links for root cause localization of a failure based on service dependency relationships obtained or making the decision on scaling microservice instances in/out according to the service runtime and infrastructure information. With various kind of data collected, targeted processing and analysis tactics are required to deal with different purposes more efficiently.

Description In terms of processing monitored data, we identified two solutions for different purposes, i.e. aggregation processing and none-aggregation processing.

- Aggregation processing means that storing logs' data in aggregated form. Storing data in aggregated form is a way for reducing the required amount of storage and allowing long term information analysis [Maye 18], such as long-term service runtime metrics. Nevertheless, some kinds of analysis are impossible to carry out depending on the level of aggregation. For example, root cause analysis requires detailed information on service communication and this is not possible with aggregated request data. Elasticsearch facilitates computing of the values used by auto scaler (average metrics, percentages) offering a complex aggregation framework [Crec 18].

- None-aggregation processing means that storing data in native form enables detailed analysis. However, this may lead to a huge amount of collected monitoring data, which need a huge amount of storage, making it almost impossible to store data over a long period of time. Thus, this is mainly useful for short time failure analysis, but it does not support the generation of long term reports [Hase 17a].

Constraint Those two solutions of the Processing Monitored Data tactic has their own best fit situation. They are not mutually exclusive and can also be used in combination. Data in native form can be used for short term analysis, while aggregated data can be used for long term analysis [Hase 17a].

Dependency This tactic may rely on the Generate monitored Data and the Storing Monitored Data for the necessary information preparation. The data processed and analyzed by this tactic may deliver to the Presenting Monitored Data tactic for visualization.

5.4.4. Presenting Monitored data

Introduction Presenting Monitored data is tactic for visual monitoring of microservices-based systems and was applied by six reviewed studies (cf. Table 9). This tactic was used in these studies to guide the visualization of information concerned by relevant stakeholders.

Motivation After the information concerned are processed and analyzed, presentation and visualization of the data monitored on a particular microservices-based system are necessary for different stakeholders [Maye 17]. Different stakeholders may take different views of the system and presenting key metrics can facilitate the relevant stakeholders to be aware of the current state of their systems and make the on-spot decision according to some states.

Description Presenting Monitored Data tactic provides a way to presenting key metrics on a particular microservices-based system in a unified view [Maye 17, Ma 18, Hase 17a]. Multiple views to address different concerns and the information needs can be created for each stakeholder. Views for presenting monitoring data are mainly fivefold: 1) Service-specific metrics for analyzing response time, failure rate, throughput, as well as CPU and memory consumption per service.

2) The IT-landscape allows analysis and generation of long term reports of infrastructure-related monitoring data like available hosts and data centers, host utilization, or service allocation to hosts. 3) The infrastructure related metrics allows to analyze metrics like CPU load and memory consumption for a specific host. 4) The map of running services provides an overview of all running services, of service interactions, and potentially provides long term reports for these data. 5) All downstream calls of a service supports finding the root cause in case of service failure. Kibana [Crec 18, Toff 17], is an open source web application that provide analytics and dashboard of microservices-based systems, and is usually combined with Elasticsearch and Logstash. Logstash collects log lines, transforms them into a unified format and sends them to a pre-defined output. Information collected through the Instrumentation tactic is stored in log files, which are sent from the container in which a microservice runs to Logstash. The output lines of Logstash are transmitted to Elasticsearch which is a full-text search server. Kibana can get its input data from Elasticsearch and then display the gathered metrics in a meaningful way for human administrators.

Constraint No data.

Dependency Presenting Monitored Data through dashboards is only able to display the metrics collected and processed. The metrics displayed rely on the Generating Monitored Data by the instrumentation, logging, and distributed tracing approaches. It also relies on the Storing Monitored Data tactic and the the Processing Monitored Data in suitable ways to provide necessary information after processing.

5.5. Security

Security is a measure of a systems ability to protect data and information from unauthorized access while providing access to people and systems that are authorized [5]. MSA distributes the application logic into multiple smaller microservices, resulting in a much more complex network interaction model between services. Attackers can thus exploit this complexity to launch attacks against applications [Sun 15]. For this reason, the *security* of MSA is valued and addressed by eight reviewed studies (See Table 4). We categorize four tactics for the *Security* of MSA based on the data extracted from relevant studies, as shown in Table 10.

Table 10
Security tactics for microservices architecture

Tactics	Studies	#
Security Monitor	[Sun 15, Otte 17, Tork 17, Yary 18b]	4
Authentication and Authorization	[Fetz 17, Wals 17, Jand 18, Yary 18a]	4
Intrusion defender	[Yary 18b, Otte 17]	2

5.5.1. Security Monitor

Introduction Security Monitor is a tactic applied by four reviewed studies to detect attacks of microservice instances.

Motivation MSA introduced many security challenges to be addressed. One concern is that it uses distributed methods to communicate amongst participating microservices, whose network/infrastructure complexity may be exploited by attackers and cause the burden of detecting attacks [Sun 15, Tork 17]. Another concern is the lack of enough trust on specific microservices or among distributed microservices, since one microservice may be compromised and controlled by an adversary abusing its privileges. Therefore, a tactic is needed to flexibly monitor risky aspects of microservices or their Infrastructure and enforce policies on them in order to detect or prevent both external and internal threats targeting MSA applications.

Description *Security* Monitor is a tactic that places monitors to observe abnormal behaviors or attacks of MSA-based applications at different levels. Security monitors may differ in their effect scopes, i.e. local or external.

- Local monitor: It is a way to place local security monitors in microservices themselves to monitor the network events [Otte 17], since the network connection seen by applications are actually defined by software (i.e. via SDN). It describes the monitoring functionality for each microservice application by associating that service with a separate security service and the network events to be monitored, implementing the illusion that the security service is physically resident on the network channel. For example, the local monitor of FlowTap proposed specifically for the network monitoring in [S46] can enforce over the internal network events [Sun 15], including HTTP, message, and database access, among microservices of the application.
- External monitor: It enables more complex evaluations and actions being taken as a result of the state of the overall system, as compared to merely a single microservices, VMs, or containers. For example, the security gateway in [Tork 17], which adapts the traditional Health Endpoint Monitoring to Security Health Endpoint and implements the efficient, real-time, and external security monitoring of microservices applications. Generally, it commences the evaluation triggered by the service registry and discovery service, e.g., Netflix Eureka, Apache Zookeeper or Hashicorp Consul, retrieves service instances' OpenAPI document from the configuration server, and launches security assessment tests to continuously detect security health information. This information monitored show the most recent security assessment result with the most important aspects such as the information of vulnerabilities [Tork 17]. Vulnerabilities' information can be directly consumed by other deployed security applications and used for security tasks such as the automated configuration of Firewall-as-a-Service (FWaaS) rules and the integration of vulnerability information into Intrusion Detection Systems (IDS).

Constraint Security policies are necessary for the assessment and monitoring of specific vulnerabilities and network-related security:

- Policies for vulnerabilities' monitor: 1) Global Policy for all the microservices of an application; 2) Microservice-Specific Policy targeting specific microservice implementation technologies; 3) VM and Container Policy at the infrastructure level in a cloud native environment.
- Policies for network-related security's monitor: 1) Connection Policy deciding whether or not a microservice can have a direct connection to another microservice; 2) Request-specific Connection Policy, defining what kind of request a microservice can make to another microservice; 3) Request Integrity Policy enforcing over the content of a request to prevent compromised microservices from hijacking requests.

Dependency *Availability* is a key quality attribute, whose tactics are highly related to this tactic, especially the Service Registry tactic in Section 5.3.2 and Fault Monitor tactic in Section 5.3.1, since they can be achieved by having elements to detect (by querying, for example) services that are down. For example, the core of the security monitor's implementation in [Tork 17] directly interacts with the Service Registry and Discovery Service (Eureka server) for policy enforcement and external security assessment.

5.5.2. Authentication and Authorization

Introduction Authentication and Authorization is tactic to resist attacks in MSA and applied by four studies. Authentication is a process by which to confirm the identity a user or a party and authorization is a mechanism by which a principal is mapped to the action allowing an identity to do.

Motivation Microservices, small and focused services deployed on independent containers or virtual machines (VMs), are an increasingly common way to structure large systems. Dealing with the authentication and authorization in microservices that is to imitate the way of monolithic structure can address the security of MSA to some extent[Jand 18]. That is each service using own database or shared database that stores credential data and implements own function to verify the the user independently. This way is easy to understand while has several deficiencies. First, when joining a new service in the system every time, the authentication and authorization function needs to be re-implemented for the new service. Second, for such an ensemble to be trustworthy, it is not enough to secure microservices individually. Communications channels must also be authenticate. Authentication-the reliable determination of the endpoints of a channelis prerequisite for reliable logging and it provides a basis for authorization or enforcement of other security policies.

Description This tactic consists of three solutions as follows :

- Key exchange-based communication authentication. Executing a key (passwords, keys or asymmetric key pairs) exchange between those processes and authenticating that exchange with all of the roles available is the basic approach to set up secure and authenticated communication between two processes. The authentication of the exchange is verified by the processes and the exchanged key will be associated with all verifiable roles. The ephemeral key is usually generated by a symmetric authenticated encryption scheme for the validation of encrypted messages, which can also be tagged with roles that were verified during the key exchange [Jand 18]. Alternative technologies for the encryption are the symmetric AES and the HMAC mechanism (based on SHA256) to ensure confidentiality and integrity respectively. Messages are encrypted based on their addresses when an event is sent using a specific key. Through these two alternative ways, only receivers knowing the encryption key (AES) and integrity key (HMAC) can decrypt the message [Fetz 17].
- Client certificate [Fetz 17, Wals 17, Jand 18]. Transport Layer Security (TLS) is another approach to secure a communication between two microservice hosts. This includes the case of invocating microservices on representational state transfer (REST), in which HTTP Secure (HTTPS) based on TLS can be used. TLS can authenticate both sides of a communication channel through server and client certificates, e.g., X.509. Kevin et. al. proposed mutual authentication mechanisms relying on TLS as an underlying transport protocol, with either asymmetric or shared symmetric keys, i.e. Attested TLS, Federated Attested TLS-PSK, Centralized Attested TLS, Centralized Attested TLS-PSK, and Local Attested TLS-PSK [Wals 17].
- Federated identity. Federated identity is a solution that extends host-authenticated TLS with in-band authentication options and allows microservices to use an identity management system that store the identity of users for the authentication [Fetz 17]. A trusted third party is usually adopted to issue permits in the form of tokens, for example, JSON Web Tokens (JWT). The trusted third party can be implemented as a separate service or just using a single sign-on framework, e.g., OpenID Connect, SAML, and OAuth2. Shibboleth is a typical solution for the federated identity and *security*.

Constraint Constraints for those above solutions of Authentication and authorization tactic are: 1) Key exchange may requires the key management and moves the focus of trust from a small amount of sound and cryptographical protocols to various unspecified mechanisms. The adoption of this approach lead to the trade-offs between *performance* and security. 2) TLS-based client certificates are not used very frequently and support should be explicitly invoked by the application code, since this method does not have strict mandate on the use and validation of certificates. Furthermore, it may be insufficient to carry out the authentication only based on host name. 3) Deploying de and integrating trusted third parties that allow federated security is very complex and requires specialized knowledge.

Dependency No Data.

5.6. Intrusion Defender

Introduction The Intrusion Defender tactic is to understand the security state of an application and react to some insecure states. The understanding and reaction about this is important for the *security* especially when designing future applications or improving the version and the feature of microservices. This tactic was used by two primary studies.

Motivation Self-protection for microservices is challenging and has received scant attention in the research literature [Otte 17]. Vulnerabilities can be detected by local or external security monitor as described in Section 5.5.1. Once an intrusion or vulnerability is detected, actions should be taken timely for the defence to stop an attack and ensure the security of systems.

Description The tactic may opt for the following approaches to react to intrusions [Yary 18b]: 1) Rollback/ Restart. This solution is to destroy the existing service instance and restart a new one with the same or ever older configuration without problem. It allows the defender to mitigate attacks and bugs in the latest version. 2) Isolation/ Shutdown. This solution is to defend the intrusions by entailing physical exclusion of problematic services through shutting down them permanently. 3) Diversification. It can be done through recompilation or binary rewriting to incorporate randomness into the binaries executing in the microservice. Besides, moving a microservice to a different host can also achieve the diversification, which helps to mitigate attacks that rely on host characteristics, for example, exploits targeting hardware or malicious cloud providers. 3) Scale up and down N-Variant services. The N-variant microservice system proposed can take actions to defend intrusions and spawn additional microservices using the aforementioned diversification techniques. It feeds the results to a governor node for the comparison of consistency, which allows to detect nodes that have been infected. 5)

Constraints No Data.

Dependency This tactic may rely on the Security Monitor tactic to provide periodic health checks against MSA-based applications, which are necessary to quickly identify failures or intrusions.

5.7. Testability

Testability is a measure of a systems ability to demonstrate its faults through (typically execution-based) testing [5]. As the relationship between microservices is complex and some microservices may experience frequent modification, it is necessary to pay attention to the *testability* of this kind of systems to minimize the impact on the *performance*, *availability* and *security*. According to Table 4, there are eight reviewed studies addressing the challenges related to the *testability* of MSA, based on which we categorized two tactics, the Automating Test Procedure and the API Documentation and Management.

5.7.1. Automating Test Procedure

Introduction Automating Test Procedure is a tactic aiming to limit the complexity of testing process using some automatic techniques[3]. This tactic was applied by seven reviewed studies.

Motivation Within a microservice architecture, the testing process plays an important role in the integration of services in a pipeline. Testing the large amount of microservices by hand is impossible and time-consuming because of the large number of tests, their complexity, and the amount of checking of results that are required. Motivated by DevOps, automating the microservices-oriented testing is imperative for efficiency's improvement.

Description For the Automating Test Procedure tactic, a testing model [Savc 15b] or approach [Ma 18] for microservice systems is useful to define the automatic microservice testing process and support a continuous integration of microservices. Microservices-oriented testing methods or procedures [Ma 18] can be automatically supported by tools implementing certain techniques, as shown in Table 12. [Mein 15] and [Arcu 17] contribute for the automation of generating test cases for Black-box End-to-End and white-box service testing respectively. Gremlin [Heor 16] provides a solution for resiliency testing of MSA-based systems, which implements three components, Recipe Translator, Failure Orchestrator, and Assertion Checker, to correspondingly support the fault modeling, injection and assertion with automation. [Karg 18] supports the automatic regression testing using Diffy in combination with the continuous delivery tool pipeline Jenkins, Gitlab, Artifactory, Kubernetes, Kubectl. [Ma 18] proposes a GMAT (Graph-based Microservice Analysis and Testing) solution to help various types of testings, especially contract testing using Pact, through analyzing dependency relationships between microservices and improving the test coverage of tests. Finally, [Savc 15b] and its extended work [Savc 15a] provides a unified validation framework for MSA-based system to support the automatic testing from the source code to the continuous stability testing.

Constraints Automating test procedures depends on automatic tools and developers may require a minimal learning curve to apply them in their production applications.

Dependency The implementation of the Automating Test Procedure tactic depends on certain approaches of the Generating Monitored Data tactic addressing *Monitorability* to

[3] Limit Complexity is a category of tactics defined by Bass et. al. [5] to decrease the complexity of testing complex systems from different aspects, e.g., state space or testing process.

Table 11
Testability tactics for microservices architecture

Tactics	Studies	#
Automating Test Procedure	[Karg 18, Heor 16, Arcu 17, Savc 15b, Mein 15, Savc 15a, Ma 18]	7
API Documentation and Management	[Tork 17, Arcu 17, Ma 18]	3

Table 12
Testing methods/procedures with automation

Studies	Testing methods/procedures with automation	Approaches	Tools
[Mein 15]	End-to-End testing	Machine learning algorithm	LBTest
[Arcu 17]	Service testing	Evolutionary genetic algorithm	EVOMASTER
[Heor 16]	Resilience testing	Fault modeling, injection and assertion	Gremlin
[Karg 18]	Regression testing	Continuous Delivery pipeline	Diffy
[Ma 18]	Contract testing	Graph-based dependency analysis	GMAT
[Savc 15b]	Functional/Load/Security unit testing Functional/load/security integration testing	An unified validation framework	Concept
[Savc 15a]	Functional/Load/Security unit testing Functional/load/security integration testing Container self-testing Continuous system stability testing	An unified validation framework	MSVF

prepare essential data. For example, the assertions in [Heor 16] may need the observability of the MSA-based system to validate the behavior of microservices, therefore it can use the logging approach of the Generating Monitored Data tactic for the necessary observation. In the last step of GMAT [Ma 18], all information needs to produce a service dependency graph (SDG) document and displaying SDG in a visualized way based on the SDG document. The SDG is automatically generated using the distributed tracing approach. In the study [Ma 18], the code instrumentation was adopted to collect code coverage metrics and ensure that the high coverage test cases are generated. Moreover, this Automating Test Procedure tactic may also rely on the other *testability* tactic, API Documentation and Management, to give detailed microservice interface description, including input and output variables types, boundaries and syntax, so that those testing solutions/technologies can be performed.

5.7.2. API Documentation and Management

Introduction API Documentation and Management has been found in three of the reviewed studies [Tork 17, Arcu 17, Ma 18]. This tactic reduces the complexity of generating and managing API specifications necessary for certain testings.

Motivation MSA advocates using a lightweight mechanism such as the HTTP RESTful API for the inter-service communication. API descriptions can provide an overview of a microservices functionality and are useful for service testing, such as contract testing [Arcu 17]. However, the continuous update or upgrade of functionality requirements may cause the inevitable evolution of APIs and complicate the relevant service testings. Consequently, it will be less painful for developers and testers to find a way of managing APIs concerning about versioning and tracking changes to APIs over time.

Description To use the API Documentation and Management tactic requires certain components to implement the generation of API document specifications and further management. As reflected by [Tork 17, Arcu 17, Ma 18], Swagger is a widely-used tool provide such components that takes the hard work out of generating and maintaining the API docs, ensuring the documentation and management stays up-to-date as the APIs evolve. For example, the Swagger Inspector can help developers to automatically generate the OpenAPI file from an API definition and for any end point. The SwaggerHubs versioning system is to maintain documentation for multiple versions of the APIs. When a RESTful API is configured with Swagger, RAML can also be used to automatically provide a JSON file as a resource that will fully define which APIs are available in that RESTful service. Then further testing on such RESTful services may be efficiently conducted through retrieving the well-defined and managed Swagger JSON definition of APIs.

Constraints No data.

Dependency This tactic can be used before the Automating Test Procedure to prepare essential data for the automation of certain testings' procedure, such as the contract testing in [Ma 18] or the service testing in [Arcu 17].

Summary to answer RQ2. For the six QAs, this SLR totally identified 19 tactics including two *scalability* tactics, four *performance* tactics, four *availability* tactics, four *monitorability* tactics, three *security* tactics, and two *testability* tactics. Based on the data synthesized, we further integrate the tactics into a reference architecture for MSA-based systems, as shown in Figure 5.

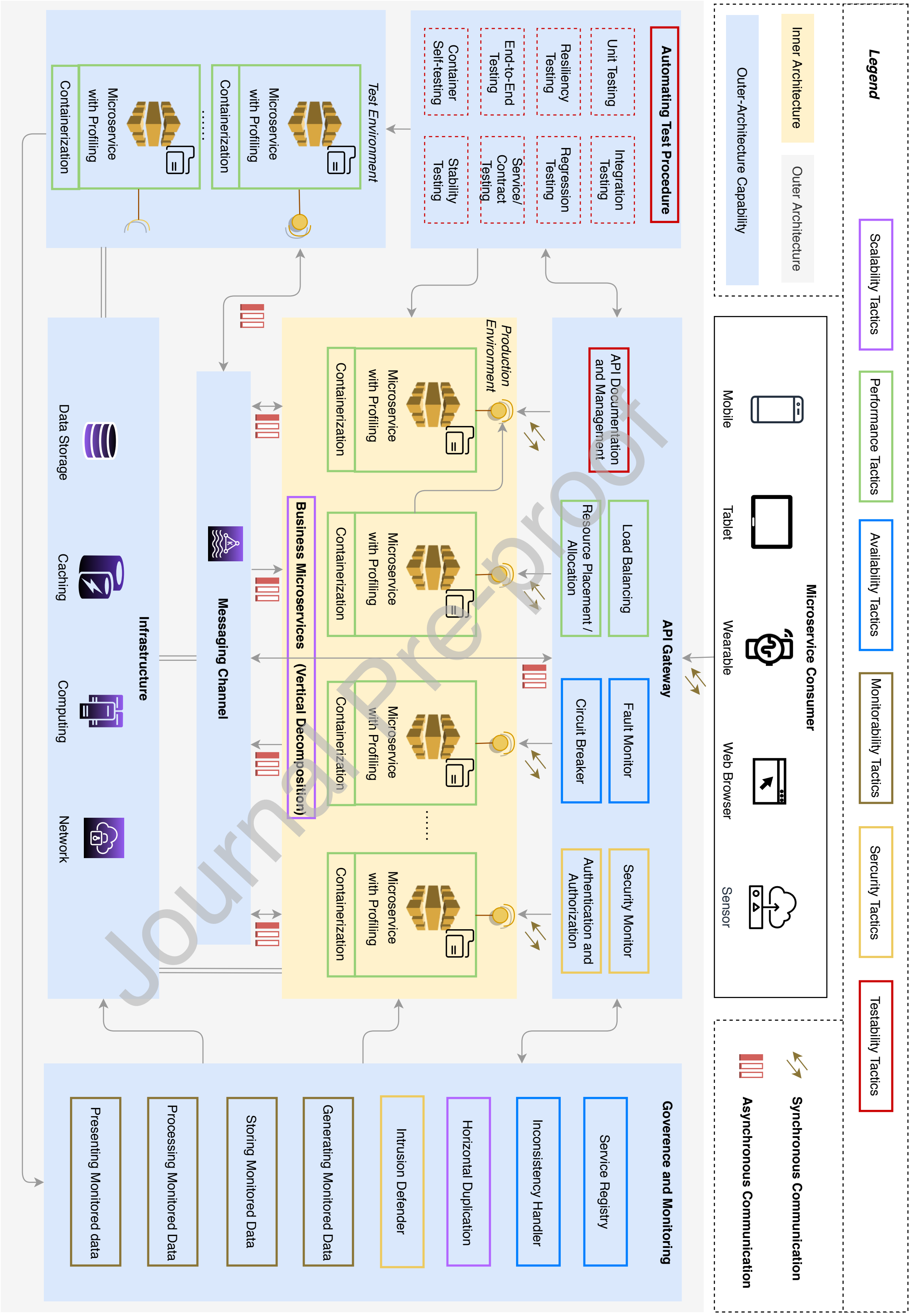


Figure 5: Reference architecture of MSA-based systems involving tactics identified.

6. Discussion

In this section we provide a summary of the main findings which indicate the potential implications for both practitioners and researchers based on the findings in Section 4 and threats to validity of this SLR.

6.1. Implications for practitioners

This SLR observes from primary studies that adopting the microservice architecture may help to reduce internal complexities of a system with monolithic architecture and also brought them many potential benefits, e.g., independent and continuous deployment, scalability, and technology heterogeneity. However, this new architectural style is not a free lunch, since we confirmed that it may bring additional pains and complexities to tackle specific QAs using a series of tactics (cf. Figure 5). The findings in Section 4 reflect that the inside chaos of monoliths were transformed into some other external and visible complications after the separation and componentization.

MSA should be embraced with caution and decisions should be carefully made by practitioners when considering the migration from monolithic systems. Firstly, practitioners may need to decide whether or not they must migrate to MSA through evaluating the return on investment (ROI). The return may refer to some intolerable pains' in their legacy systems that could be addressed by MSA, while the investment means the additional efforts and cost to implement the tactics for specific QAs of the identified set in this SLR or other important ones we do not obtain due to the restricted time span. Secondly, the complex relationships among QAs require to be considered during the migration to MSA, e.g, dependency or trade-off. On the one hand, the improvement of one QA may have positive influence (dependency) on other QAs. For example, tactics for *monitorability* would also offer data support for *performance* optimization and capacity planning. On the other hand, addressing one QA may negatively impact certain QAs. In other words, there may be trade-off relationships between two different QAs. Taking the *security* as an example, its Security Monitor tactic should provide concerned nonfunctional requirements, such as minimal overhead and tamper resistance from the perspective of *performance* [Sun 15]. Another example is that using the Vertical Decomposition tactic to decompose monolithic systems into microservice candidates may consider the optimal trade-off between *security* [Hass 17] or *performance* [Kloc 17] and *scalability*.

6.2. Implications for researchers

Two implications for researchers are reflected from the findings of this SLR, which are the simultaneous addressing experience of QAs and the need of empirical research and evidence.

6.2.1. Simultaneous addressing of QAs

From the reviewed studies we find that QAs in the context of MSA are not always impacted and addressed independently and 25% of the 72 studies only address more than a single quality attribute in their microservices architecture, as shown in Table 13. This table indicates that *scalability* and *monitorability* are the most frequently addressed combination of QAs. As mentioned in Section 5.1.1, the Reactive Auto-scaling approach *scalability* tightly depends on the tactics for *monitorability* (cf. Section 5.4) to provide necessary information for the decision of threshold configuration and the scaling operations. *Performance*, *testability*, *monitorability* and *security* were also taken into consideration when addressing *availability*, for example, the Fault Injection tactic for *testability* and the Data collection for *monitorability* can facilitate the fault detection for *availability*. Moreover, *availability* and *performance*, and *scalability* and *availability* were considered in pairs by one study respectively, since some tactics (Vertical Decomposition) may have positive or negative influence on *performance* or *availability*. *Testability* was respectively combined with *monitorability* and *security* by one study to address, for the Automating Test Procedure may rely on the data provided by the *monitorability* tactics and the *Testability* addressing may positively impact the *security*. It is worth mentioning that there are also two studies addressing three QAs at the same time.

Table 13
List of simultaneous QAs' addressing for MSA

QA sets	Studies	#
Scalability and Monitorability	[Lehv 17, Gan 18, Alip 17, Crec 18, Toff 17]	5
Availability and Performance	[Sing 17, Kang 16]	2
Availability and Testability	[Karg 18, Heor 16]	2
Availability and Monitorability	[Kita 17, Stub 15]	2
Scalability and Performance	[Karw 17]	1
Scalability and Availability	[Thal 17]	1
Availability and Security	[Otte 17]	1
Testability and Monitorability	[Ma 18]	1
Testability and Security	[Tork 17]	1
Performance, Scalability, and Availability	[Guer 18a, Toff 15]	2

The above results may have two more in-depth implications for researchers. One the one hand, it will be more valuable from both industry and academia side, if researchers of a primary study related to MSA can take the multiple relevant QAs together into consideration in their research or methods proposed. We believe this implication also applies to researchers studying the architecture design of applications in other domains. On the other hand, researchers may pay their attention to analyze the complex interaction and trade-off relationship of QAs in MSA, which is a possible direction for the future research work seldom studied at present.

6.2.2. The need of empirical research and evidence

Although we aim to offer a relatively comprehensive understanding of QAs related to MSA. It may lack some evidence for specific QAs. Some QAs, apart from the six ones identified in this study, turn to arouse the interest of researchers in MSA domain. Unfortunately, we could not extract evidence from the reviewed studies to answer our research questions. These QAs need further investigation from the aspects of tactics and measurements in future research. In addition to the QAs listed in Table 4, we also identify other three QAs but without essential evidence support, i.e. *interoperability*, *portability*, and *maintainability* [6]. They were only mentioned in the reviewed studies with no evidence for answering our research questions. *interoperability* and *portability* of MSA are claimed to be positively influenced in three and two papers respectively. On the contrary, three papers mentioned that maintaining multiple services with complex relationship is in the list of the challenging items while building MSA. More empirical research is needed to study these QAs, in particular for improving *maintainability* of MSA.

For those six QAs, we were not able to obtain any information related to the constraints of three tactics, e.g., the Inconsistency Handler tactic in Section 5.3.4, the Presenting Monitored data tactic in Section 5.4.4, and the API Documentation and Management in Section 5.7.2 tactic. At present, we also lack enough data to systematically assess the risks of introducing the corresponding tactics since relevant studies do not provide such information, which indicates that attention should be paid here to collect additional empirical evidence in the future.

6.3. Threats to Validity

The possible threats to the validity of this study are discussed in this section.

Construct Validity: The selection of the primary studies in this SLR may directly affect the quality of data for synthesis. Hence, we first defined a rigorous search strategy and revised it during the search; then a multi-step selection process was applied. The search string in the search process may include inadequate search terms related to the research topic. Therefore, to avoid missing any relevant studies, we used the search string in a generic form and removed any terms related to QA. Moreover, we applied the snowballing strategy in the search to ensure that we could search the relevant studies as many as possible. We used tailored search strings in each digital library to ensure the repeatability of the process.

Internal Validity: In case of missing relevant studies, we searched four primary databases: IEEE Xplore, ACM Digital Library, ScienceDirect and SpringerLink. We also used the snowballing method to ensure that we identify the relevant studies as much as possible. We adopted the QGS method, which systematically integrates the manual and automated search strategies and suggests a relatively rigorous approach to search *performance* evaluation in terms of sensitivity and precision.

Conclusion Validity: One possible threat to the conclusion of this SLR is that some studies we identified are not mature work and may impact the genaralizability of the conclusion. To mitigate this threat, we carefully selected the relevant studies, extracted the data and tried to synthesize data with enough evidence support. Moreover, to ensure the quality of the extracted data, a protocol was developed to define the data extraction strategy and format. The review protocol was mainly developed by the PhD student, and was then reviewed by the supervisors and the industrial expert. The cross-check was necessary among the reviewers, and we had at least two researchers extracting data from each paper independently to minimize the possible researcher bias introduced by individuals. The supervisor dealt with any divergences and disagreements during the process.

7. Conclusion

As an increasingly accepted and adopted architecture style, MSA overcomes the limitations of the traditional monolithic architecture. The QAs impacted by MSA have received significant attention both in academia and in industry. To the best of our knowledge, however, there is no comprehensive understanding of the state-of-the-art of QAs reported in particular to MSA. This paper reports an SLR that identifies 72 relevant studies, confirms six most critical QAs of MSA from those papers, and further codifies 19 tactics architecturally addressing the six QAs (using a template including introduction, motivation, description, constraint, and dependency).

The results and findings from this SLR offer implications for both practitioners and researchers. For practitioners, this paper provides a comprehensive understanding of the QAs impacted by MSA, which can serve as a useful design space for MSA-based systems. The quantitative demography and quantitative tactics are able to contribute for efficient decision making of software architects, considering non-functional requirements and the potential investment of migrating to MSA-based systems. For researchers, we identify the missing areas and research opportunities on this trending topic, and contribute to the progressive development of a broad body of knowledge on MSA with the latest available evidence, for example, the simultaneous addressing and trade-off analysis of QAs and the need of more empirical research and evidence for some other concerned QAs, e.g., *maintainability*.

The future work of this paper can be twofold: firstly, we plan to conduct primary empirical studies on the research opportunities identified in this SLR, e.g., exploration of more MSA-specific tactics or trade-off analysis among different QAs through conducting experiments; secondly, we will extend this review with a continuous focus on the latest knowledge supplements, i.e. emerging QAs, tactics, response measures and tools for relevant QAs.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2019YFE0105500)

References

[1] Abbott, M.L., Fisher, M.T., 2009. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education.

[2] Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture, in: Proceedings of the 2016 IEEE

9th International Conference on Service-Oriented Computing and Applications (SOCA), IEEE Press. pp. 44–51.

[3] Balalaie, A., Heydarnoori, A., Jamshidi, P., 2016. Microservices architecture enables devops: Migration to a cloud-native architecture. Ieee Software 33, 42–52.

[4] Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T., 2018. Microservices migration patterns. Software: Practice and Experience 48, 2019–2042.

[5] Bass, L., Clements, P., Kazman, R., 2012. Software architecture in practice .

[6] Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2018. Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service-and microservice-based systems, in: Proceedings of the 2018 International Conference on Technical Debt, ACM. pp. 125–133.

[7] Cruzes, D.S., Dyba, T., 2011. Recommended steps for thematic synthesis in software engineering, in: 2011 international symposium on empirical software engineering and measurement, IEEE. pp. 275–284.

[8] Dybå, T., Dingsøyr, T., 2008. Empirical studies of agile software development: A systematic review. Information and software technology 50, 833–859.

[9] Febrero, F., Calero, C., Moraga, M.Á., 2016. Software reliability modeling based on ISO/IEC SQuaRE. Information and Software Technology 70, 18–29.

[10] Fowler, M., Lewis, J., 2014. Microservices. URL: https://martinfowler.com/articles/microservices.html.

[11] Francesco, P.D., Malavolta, I., Lago, P., 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption, in: Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), IEEE Press. pp. 21–30.

[12] Garcés, L., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y., 2017. Quality attributes and quality models for ambient assisted living software systems: A systematic mapping. Information and Software Technology 82, 121–138.

[13] Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O., 2016. Service Cutter: A systematic approach to service decomposition, in: Proceedings of the European Conference on Service-Oriented and Cloud Computing, Springer. pp. 185–200.

[14] Hassan, S., Ali, N., Bahsoon, R., 2017. Microservice ambients: An architectural meta-modelling approach for microservice granularity, in: Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), IEEE Press. pp. 1–10.

[15] Hassan, S., Bahsoon, R., 2016. Microservices and their design tradeoffs: A self-adaptive roadmap, in: Proceedings of the 2016 IEEE International Conference on Services Computing (SCC), IEEE Press. pp. 813–818.

[16] Jayawardana, Y., Fernando, R., Jayawardena, G., Weerasooriya, D., Perera, I., 2018. A full stack microservices framework with business modelling, in: 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer), IEEE. pp. 78–85.

[17] Kecskemeti, G., Marosi, A.C., Kertesz, A., 2016. The ENTICE approach to decompose monolithic services into microservices, in: Proceedings of the 2016 International Conference on High Performance Computing Simulation (HPCS), IEEE Press. pp. 591–596.

[18] Kitchenham, B.A., Budgen, D., Brereton, P., 2015. Evidence-based software engineering and systematic reviews. volume 4. CRC press.

[19] Lin, J., Lin, L.C., Huang, S., 2016. Migrating web applications to clouds with microservice architectures, in: Proceedings of the 2016 International Conference on Applied System Innovation (ICASI), IEEE Press. pp. 1–4.

[20] Mahdavi-Hezavehi, S., Durelli, V.H., Weyns, D., Avgeriou, P., 2017. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. Information and Software Technology 90, 1–26.

[21] Mahdavi-Hezavehi, S., Galster, M., Avgeriou, P., 2013. Variability in quality attributes of service-based software systems: A systematic literature review. Information and Software Technology 55, 320–343.

[22] Newman, S., 2015. Building microservices. Second ed., O'Reilly Media, Sebastopol, California.

[23] Osborne, J.W., 2008. Best practices in quantitative methods. Sage.

[24] Pahl, C., Jamshidi, P., 2016. Microservices: A systematic mapping study, in: Proceedings of the 6th International Conference on Cloud Computing and Services Science, ACM. pp. 137–146.

[25] Perera, K., Perera, I., 2018. A rule-based system for automated generation of serverless-microservices architecture, in: 2018 IEEE International Systems Engineering Symposium (ISSE), IEEE. pp. 1–8.

[26] Soldani, J., Tamburri, D.A., Van Den Heuvel, W.J., 2018. The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software 146, 215–232.

[27] Ullah, F., Babar, M.A., 2019. Architectural tactics for big data cybersecurity analytics systems: A review. Journal of Systems and Software 151, 81–118.

[28] Zhang, H., Ali Babar, M., Tell, P., 2011. Identifying relevant studies in software engineering. Information and Software Technology 53, 625–637.

[29] Zimmermann, O., 2017. Microservices tenets: Agile approach toservice development and deployment. Computer Science-Research and Development 32, 301–310.

Appendix A. Quality assessment results

Appendix B. The Reviewed Studies

References

[Abad 18] C. L. Abad, E. F. Boza, and E. Van Eyk. "Package-Aware Scheduling of FaaS Functions". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 101–106, ACM, 2018.

[Ahma 16] M. Ahmadvand and A. Ibrahim. "Requirements reconciliation for scalable and secure microservice (de) composition". In: *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, pp. 68–73, IEEE, 2016.

[Alip 17] H. Alipour and Y. Liu. "Online machine learning for cloud resource provisioning of microservice backend systems". In: *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2433–2441, IEEE, 2017.

[Amar 15] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. "Performance evaluation of microservices architectures using containers". In: *2015 IEEE 14th International Symposium on Network Computing and Applications*, pp. 27–34, IEEE, 2015.

[Arcu 17] A. Arcuri. "RESTful API automated test case generation". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 9–20, IEEE, 2017.

[Bare 17] L. Baresi, M. Garriga, and A. De Renzis. "Microservices identification through interface analysis". In: *European Conference on Service-Oriented and Cloud Computing*, pp. 19–33, Springer, 2017.

[Bham 17] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan. "Multi-objective scheduling of micro-services for optimal service function chains". In: *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2017.

[Cele 17] A. Celesti, L. Carnevale, A. Galletta, M. Fazio, and M. Villari. "A watchdog service making container-based micro-services reliable in IoT clouds". In: *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 372–378, IEEE, 2017.

[Chen 17] R. Chen, S. Li, and Z. Li. "From monolith to microservices: a dataflow-driven approach". In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–475, IEEE, 2017.

[Crec 18] C.-C. Crecana and F. Pop. "Monitoring-based auto-scalability across hybrid clouds". In: *Proceedings of the 33rd An-*

Table 14
Quality score of the reviewed studies

Studies	Q1	Q2	Q3	Q4	Q5	Total	Studies	Q1	Q2	Q3	Q4	Q5	Total
[Kook 16]	1	1	1	1	0.5	4.5	[Maye 17]	1	0.5	0.5	0.5	0.5	3
[Kita 17]	0.5	1	1	1	0.5	4	[John 17]	1	0.5	0.5	0.5	1	3.5
[Maye 18]	1	0.5	1	1	1	4.5	[Phip 18]	1	0.5	1	1	0.5	4
[Khaz 16]	1	1	1	1	0	4	[Guer 18a]	1	1	1	1	0.5	4.5
[Ruse 16]	1	0.5	1	1	0	3.5	[Sing 17]	1	1	1	1	0	4
[Krat 16]	0.5	0.5	1	1	0	3	[Mei 18]	1	1	1	1	0	4
[Kang 16]	1	1	1	1	1	5	[Kloc 17]	1	1	1	1	0.5	4.5
[Bham 17]	1	0.5	1	1	0.5	4	[Abad 18]	1	1	0.5	0.5	1	4
[Zhen 17]	1	0.5	0.5	1	0.5	3.5	[Amar 15]	1	0.5	1	1	0.5	4
[Guer 18b]	1	1	0.5	1	0.5	4	[Lloy 18]	1	1	1	1	0	4
[Grib 18]	0.5	1	1	1	0.5	4	[Pard 18]	0.5	0.5	0.5	1	0	2.5
[Kalr 18]	1	1	0	0.5	0.5	3	[Fili 18]	1	1	1	1	0.5	4.5
[Nico 18]	1	1	0.5	0.5	0.5	3.5	[Liu 18b]	1	0.5	1	0.5	0	3
[Gan 18]	1	0.5	1	0.5	0	3	[Cele 17]	1	1	1	1	0.5	4.5
[Karg 18]	1	1	0.5	0.5	0	3	[Kaki 18]	1	1	1	1	0.5	4.5
[Toff 15]	1	0.5	0	0.5	1	3	[Lope 17]	1	1	1	1	1	5
[Lehv 17]	0.5	0.5	0.5	1	1	3.5	[Alip 17]	1	0.5	1	1	1	4.5
[Karw 17]	1	0.5	1	1	0.5	4	[Crec 18]	1	1	1	1	0	4
[Klin 18]	1	0.5	1	1	0.5	4	[Goti 18a]	1	1	1	1	1	5
[Prac 18]	0.5	0.5	1	0.5	0.5	3	[Sun 15]	1	1	1	1	0	4
[Otte 17]	1	0.5	0.5	0.5	0	2.5	[Fetz 17]	1	0.5	1	1	0.5	4
[Wals 17]	0.5	0.5	1	0.5	0	2.5	[Furd 18]	0.5	0.5	0.5	0.5	0	2
[Tork 17]	1	1	0.5	0.5	1	4	[Jand 18]	0.5	0.5	0.5	0.5	0.5	2.5
[Yary 18b]	1	0.5	0.5	0.5	0.5	3	[Heor 16]	1	1	1	1	1	5
[Arcu 17]	1	1	1	1	0.5	4.5	[Savc 15b]	1	0.5	0.5	0.5	0.5	3
[Mein 15]	1	0.5	1	1	0.5	4	[Savc 15a]	1	1	0.5	0.5	0	3
[Ma 18]	1	1	1	1	0.5	4.5	[Mazl 17]	1	1	1	1	1	5
[Thal 17]	1	1	1	1	1	5	[Bare 17]	1	1	1	1	0.5	4.5
[Ahma 16]	1	1	1	0.5	0.5	4	[Hass 17]	0.5	0.5	0.5	0.5	0	2
[Gyse 16]	1	1	1	1	1	5	[Kryl 15]	1	0.5	0.5	1	0.5	3.5
[Chen 17]	1	1	1	1	1	5	[Toff 17]	1	0.5	1	0.5	0.5	3.5
[Hase 17a]	1	1	0.5	0.5	0	3	[Stub 15]	1	1	1	0.5	1	4.5
[Eski 18]	0.5	1	1	1	0.5	4	[Goti 18b]	1	1	1	0.5	1	4.5
[Seli 17]	1	0.5	1	0.5	1	4	[Hase 17b]	0.5	1	0.5	1	1	4
[Mont 18]	1	0.5	0.5	0.5	0	2.5	[Liu 18a]	1	1	1	0.5	1	4.5
[Yary 18a]	1	1	1	0.5	1	4.5	[Mora 17]	1	1	1	1	1	5

nual ACM Symposium on Applied Computing, pp. 1087–1094, ACM, 2018.

[Eski 18] S. Eski and F. Buzluca. "An automatic extraction approach: Transition to microservices architecture from monolithic application". In: *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pp. 1–6, 2018.

[Fetz 17] C. Fetzer, G. Mazzeo, J. Oliver, L. Romano, and M. Verburg. "Integrating Reactive Cloud Applications in SERECA". In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*, p. 39, ACM, 2017.

[Fili 18] I.-D. Filip, F. Pop, C. Serbanescu, and C. Choi. "Microservices scheduling model over heterogeneous cloud-edge environments as support for iot applications". *IEEE Internet of Things Journal*, Vol. 5, No. 4, pp. 2672–2681, 2018.

[Furd 18] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros. "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency". *IEEE Software*, Vol. 35, No. 3, pp. 63–72, 2018.

[Gan 18] Y. Gan and C. Delimitrou. "The Architectural Implications of Cloud Microservices". *IEEE Computer Architecture Letters*, Vol. 17, No. 2, pp. 155–158, 2018.

[Goti 18a] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner. "Investigating performance metrics for scaling microservices in cloudiot-environments". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 157–167, ACM, 2018.

[Goti 18b] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner. "Investigating performance metrics for scaling microservices in cloudiot-environments". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 157–167, 2018.

[Grib 18] M. Gribaudo, M. Iacono, and D. Manini. "Performance evaluation of replication policies in microservice based architectures". In: *Ninth International Workshop on the Practical Application of Stochastic Modelling (PASM)*, pp. 45–65, 2018.

[Guer 18a] C. Guerrero, I. Lera, and C. Juiz. "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture". *Journal of Grid Computing*, Vol. 16, No. 1, pp. 113–135, 2018.

[Guer 18b] C. Guerrero, I. Lera, and C. Juiz. "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications". *The Journal of Supercomputing*, Vol. 74, No. 7, pp. 2956–2983, 2018.

[Gyse 16] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. "Service cutter: A systematic approach to service decomposition". In: *European Conference on Service-Oriented and Cloud Computing*, pp. 185–200, Springer, 2016.

[Hase 17a] S. Haselböck and R. Weinreich. "Decision guidance models for microservice monitoring". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 54–61, IEEE, 2017.

[Hase 17b] S. Haselböck, R. Weinreich, and G. Buchgeher. "Decision guidance models for microservices: service discovery and fault tolerance". In: *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*, pp. 1–

10, 2017.

[Hass 17] W. Hasselbring and G. Steinacker. “Microservice architectures for scalability, agility and reliability in e-commerce”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246, IEEE, 2017.

[Heor 16] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. “Gremlin: Systematic resilience testing of microservices”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 57–66, IEEE, 2016.

[Jand 18] K. Jander, L. Braubach, and A. Pokahr. “Defense-in-depth and Role Authentication for Microservice Systems”. *Procedia computer science*, Vol. 130, No. C, pp. 456–463, 2018.

[John 17] W. John, F. Moradi, B. Pechenot, and P. Sköldström. “Meeting the observability challenges for VNFs in 5G systems”. In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1127–1130, IEEE, 2017.

[Kaki 18] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, *et al.* “Service fabric: a distributed platform for building microservices in the cloud”. In: *Proceedings of the Thirteenth EuroSys Conference*, p. 33, ACM, 2018.

[Kalr 18] S. Kalra and T. Prabhakar. “Towards Dynamic Tenant Management for Microservice based Multi-Tenant SaaS Applications”. In: *Proceedings of the 11th Innovations in Software Engineering Conference*, p. 12, ACM, 2018.

[Kang 16] H. Kang, M. Le, and S. Tao. “Container and microservice driven design for cloud infrastructure devops”. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 202–211, IEEE, 2016.

[Karg 18] M. J. Kargar and A. Hanifizade. “Automation of regression test in microservice architecture”. In: *2018 4th International Conference on Web Research (ICWR)*, pp. 133–137, IEEE, 2018.

[Karw 17] W. Karwowski, M. Rusek, G. Dwornicki, and A. Orłowski. “Swarm based system for management of containerized microservices in a Cloud consisting of heterogeneous servers”. In: *International Conference on Information Systems Architecture and Technology*, pp. 262–271, Springer, 2017.

[Khaz 16] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. “Efficiency analysis of provisioning microservices”. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 261–268, IEEE, 2016.

[Kita 17] S. Kitajima and N. Matsuoka. “Inferring Calling Relationship Based on External Observation for Microservice Architecture”. In: *International Conference on Service-Oriented Computing*, pp. 229–237, Springer, 2017.

[Klin 18] F. Klinaku, M. Frank, and S. Becker. “CAUS: An Elasticity Controller for a Containerized Microservice”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 93–98, ACM, 2018.

[Kloc 17] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, and S. Jansen. “Workload-based clustering of coherent feature sets in microservice architectures”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 11–20, IEEE, 2017.

[Kook 16] P. Kookarinrat and Y. Temtanapat. “Design and implementation of a decentralized message bus for microservices”. In: *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–6, IEEE, 2016.

[Krat 16] N. Kratzke and P.-C. Quint. “Investigation of impacts on network performance in the advance of a microservice design”. In: *International Conference on Cloud Computing and Services Science*, pp. 187–208, Springer, 2016.

[Kryl 15] A. Krylovskiy, M. Jahn, and E. Patti. “Designing a smart city internet of things platform with microservice architecture”. In: *2015 3rd International Conference on Future Internet of Things and Cloud*, pp. 25–30, IEEE, 2015.

[Lehv 17] J. Lehvä, N. Mäkitalo, and T. Mikkonen. “Case study: building a serverless messenger chatbot”. In: *International Conference on Web Engineering*, pp. 75–86, Springer, 2017.

[Liu 18a] H. Liu, Z. Cao, and X. Zhang. “An efficient algorithm of context-clustered microservice discovery”. In: *Proceedings of the 2nd International Conference on Computer Science and Application Engineering*, pp. 1–6, 2018.

[Liu 18b] X. Liu, Y. Jin, Y. Song, and X. Zhao. “Queue-Waiting-Time Based Load Balancing Algorithm for Fine-Grain Microservices”. In: *International Conference on Services Computing*, pp. 176–191, Springer, 2018.

[Lloy 18] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. “Serverless computing: An investigation of factors influencing microservice performance”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 159–169, IEEE, 2018.

[Lope 17] M. R. López and J. Spillner. “Towards quantifiable boundaries for elastic horizontal scaling of microservices”. In: *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*, pp. 35–40, ACM, 2017.

[Ma 18] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh. “Using Service Dependency Graph to Analyze and Test Microservices”. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 81–86, IEEE, 2018.

[Maye 17] B. Mayer and R. Weinreich. “A dashboard for microservice monitoring and management”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 66–69, IEEE, 2017.

[Maye 18] B. Mayer and R. Weinreich. “An Approach to Extract the Architecture of Microservice-Based Software Systems”. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 21–30, IEEE, 2018.

[Mazl 17] G. Mazlami, J. Cito, and P. Leitner. “Extraction of microservices from monolithic software architectures”. In: *2017 IEEE International Conference on Web Services (ICWS)*, pp. 524–531, IEEE, 2017.

[Mei 18] X. Mei and X. Tan. “Research on health monitoring mechanism based on service invocation”. In: *2018 5th International Conference on Industrial Engineering and Applications (ICIEA)*, pp. 79–83, IEEE, 2018.

[Mein 15] K. Meinke and P. Nycander. “Learning-based testing of distributed microservice architectures: Correctness and fault injection”. In: *SEFM 2015 Collocated Workshops*, pp. 3–10, Springer, 2015.

[Mont 18] F. Montesi and J. Weber. “From the decorator pattern to circuit breakers in microservices”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pp. 1733–1735, 2018.

[Mora 17] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu. “ConMon: An automated container based network performance monitoring system”. In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 54–62, IEEE, 2017.

[Nico 18] J. Nicol, C. Li, P. Chen, T. Feng, and H. Ramachandra. “Odp: an infrastructure for on-demand service profiling”. In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 139–144, ACM, 2018.

[Otte 17] C. Otterstad and T. Yarygina. “Low-level exploitation mitigation by diverse microservices”. In: *European Conference on Service-Oriented and Cloud Computing*, pp. 49–56, Springer, 2017.

[Pard 18] G. Pardon and C. Pautasso. “Consistent Disaster Recovery for Microservices: the CAB Theorem”. *IEEE Cloud Computing*, 2018.

[Phip 18] C. Phipathananunth and P. Bunyakiati. “Synthetic Runtime Monitoring of Microservices Software Architecture”. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 448–453, IEEE, 2018.

[Prac 18] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang. “Auto-scaling microservices on IaaS under SLA with cost-effective framework". In: *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pp. 583–588, IEEE, 2018.

[Ruse 16] M. Rusek, G. Dwornicki, and A. Orłowski. "A decentralized system for load balancing of containerized microservices in the cloud". In: *International Conference on Systems Science*, pp. 142–152, Springer, 2016.

[Savc 15a] D. Savchenko and G. Radchenko. "Microservices validation: Methodology and implementation". In: *CEUR Workshop Proceedings. Vol. 1513: Proceedings of the 1st Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2015).Yekaterinburg*, 2015.

[Savc 15b] D. I. Savchenko, G. I. Radchenko, and O. Taipale. "Microservices validation: Mjolnirr platform case study". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 235–240, IEEE, 2015.

[Seli 17] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga. "Practical service placement approach for microservices architecture". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 401–410, IEEE, 2017.

[Sing 17] V. Singh and S. K. Peddoju. "Container-based microservice architecture for cloud applications". In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 847–852, IEEE, 2017.

[Stub 15] J. Stubbs, W. Moreira, and R. Dooley. "Distributed systems of microservices using docker and serfnode". In: *2015 7th International Workshop on Science Gateways*, pp. 34–39, IEEE, 2015.

[Sun 15] Y. Sun, S. Nanda, and T. Jaeger. "Security-as-a-service for microservices-based cloud applications". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 50–57, IEEE, 2015.

[Thal 17] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. "Sieve: actionable insights from monitored metrics in distributed systems". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pp. 14–27, ACM, 2017.

[Toff 15] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds. "An architecture for self-managing microservices". In: *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pp. 19–24, ACM, 2015.

[Toff 17] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert. "Self-managing cloud-native applications: Design, implementation, and experience". *Future Generation Computer Systems*, Vol. 72, pp. 165–179, 2017.

[Tork 17] K. A. Torkura, M. I. Sukmana, and C. Meinel. "Integrating continuous security assessments in microservices and cloud native applications". In: *Proceedings of the10th International Conference on Utility and Cloud Computing*, pp. 171–180, ACM, 2017.

[Wals 17] K. Walsh and J. Manferdelli. "Mechanisms for Mutual Attested Microservice Communication". In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, pp. 59–64, ACM, 2017.

[Yary 18a] T. Yarygina and A. H. Bagge. "Overcoming security challenges in microservice architectures". In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 11–20, IEEE, 2018.

[Yary 18b] T. Yarygina and C. Otterstad. "A Game of Microservices: Automated Intrusion Response". In: *IFIP International Conference on Distributed Applications and Interoperable Systems*, pp. 169–177, Springer, 2018.

[Zhen 17] T. Zheng, Y. Zhang, X. Zheng, M. Fu, and X. Liu. "Bigvm: A multi-layer-microservice-based platform for deploying saas". In: *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pp. 45–50, IEEE, 2017.