ISSN 0361-7688, Programming and Computer Software, 2020, Vol. 46, No. 8, pp. 594–608. © Pleiades Publishing, Ltd., 2020.

Patterns Related to Microservice Architecture: a Multivocal Literature Review

J. A. Valdivia[a,*], A. Lora-González[a,], X. Limón[a,***], K. Cortes-Verdin[a,****], and J.O. Ocharán-Hernández[a]**

[a]*School of Statistics and Informatics, Universidad Veracruzana Av. Xalapa, Obrero Campesino, Xalapa-Enríquez, 91020 Ver. Mexico*

**e-mail: zs15011636@estudiantes.uv.mx*

***e-mail: zs15011639@estudiantes.uv.mx*

****e-mail: hlimon@uv.mx*

*****e-mail: kcortes, jocharan@uv.mx*

Received April 9, 2020; revised May 15, 2020; accepted July 16, 2020

Abstract—A Microservice Architecture enables the development of distributed systems using a set of highly cohesive, independent, and collaborative services, ready for current cloud computing demands. Each microservice can be implemented in different technologies, sharing common communication channels, which results in heterogeneous distributed systems that exhibit high scalability, maintainability, performance, and interoperability. Currently, there are many options to build microservices; some of them led by patterns that establish common structures to solve recurrent problems. Nevertheless, as microservices are an emerging trend, the relationship between quality attributes, metrics, and patterns is not clearly defined, which is a concern from a software engineering point of view, since such understanding is fundamental to correctly design systems using this architecture. This paper aims to extend the knowledge on the design of microservices-based systems by presenting a multivocal systematic literature review for microservices related patterns, tying them together with quality attributes and metrics, as can be found in academic and industry research.

DOI: 10.1134/S0361768820080253

1 INTRODUCTION

The microservice architecture (from now on MSA) has arisen as a response to fast technological changes, extensibility and scalability concerns, and to the necessity of shorter software delivery cycles. Such concerns are the motivation of several technological trends based on cloud computing [1–4]. MSA can be understood as a set of small, heterogeneous, highly cohesive, and distributed services that use common communication channels to interact with each other in order to achieve business goals. The successful MSA systems of Netflix, Amazon, and SoundCloud [5–7] set the basis for the increasing interest in this architectural style, positioning it as an alternative to Service Oriented Architecture (SOA), from which MSA has originated [8]. Furthermore, the microservices implementation offers services with high granularity, supports heterogeneous technologies, and allows high failure tolerance [9]; such features can be translated to quality attributes such as maintainability, portability, and reliability.

Quality attributes are paramount in software development since their degree of satisfaction has a fundamental impact on the overall success of a software project. The choice of architectural style, along with the architectural decisions made during the design process, enables the fulfillment of quality attributes [10]. A quality attribute is a measurable and verifiable property of the system that helps to identify how well the system satisfies the needs of the stakeholders [11]. Decisions on architectural problems should consider trade-offs for each solution, the balance between the set of quality attributes specified for the system, and the benefits offered by the proposed solution [12].

Related to quality attributes are patterns, which entail a set of reusable structures for similar problems in different contexts and provide a unified vocabulary for the software design community [13]. Pattern structures represent an extension of the architectural elements used for constructing a system; in the same way, they ease the development of more complex systems, by means of combining multiple patterns [14]. Different abstraction levels are related to the pattern concept, architecture patterns, and design patterns that appeared in the literature almost at the same time [12], [13]. Regarding the description of both concepts, [13] defines architectural patterns as fundamental structures of the system which maintain responsibilities and rules. Such structures allow comprehensive solutions

to general problems that can be related to more than one quality attribute.

In contrast, design patterns are described in [12] as subsystems or components added to the system. Also, design patterns are described in more detail, and its inclusion should not affect the fundamental structure of the architecture. Furthermore, design patterns are usually associated with a specific problem and one quality attribute. In this work, the term pattern is used interchangeably to refer to design or architectural patterns.

The inclusion of a pattern seeks to solve a problem or to improve a quality attribute related to requirements [10]. One or more patterns can be included in the design to fulfill a required quality attribute; such combination may provide different levels of quality satisfaction, but identifying the correct combination can be a complex task. Metrics reduce the difficulty of quantifying a system property, providing a numeric value as a measure of how close an attribute is to the desired level of satisfaction [15]. The value obtained by a metric enables the comparison and selection of the correct pattern to satisfy the defined goals; it also reduces the subjectivity to estimate how this decision impacts on other quality attributes.

The emergence and adoption of patterns is the result of a high maturity degree in software development, derived from experience. For example, the widely known work of Buschmann et al. [13] was the result of the authors' experience when working at Siemens. They created a catalog of solutions to improve the architecture process definition. In the same way, the publication of [12] was based on the authors' previous experience, providing a collection of object-oriented patterns [16]. Concerning Microservice Architecture, to the best of our knowledge, the pattern collections are limited, being microservices an emerging trend. However, we can find studies like [17–19] with numerous architectural patterns, providing some range of alternatives. Among them, only the last study provides a mapping between quality attributes and patterns. Therefore, it is difficult to identify an appropriate pattern to satisfy a quality attribute. Besides, it is complicated to identify quality attributes affected by the selected pattern.

Patterns in microservices are essential elements for the construction of software since the inclusion of a pattern in the architecture provides a reliable solution to a problem, helps to satisfy one or more quality attributes, and it even improves communication across the development team. However, identifying available patterns for microservices is a difficult task since related information is scarce and spread across white and grey literature, i.e., academy and industry publications, respectively.

This work aims to present a current MSA pattern collection related to quality attributes, with associated metrics which, in turn, indicate the degree of fulfillment of a quality attribute by a specific pattern. Such patterns were recovered through a multivocal literature review, covering white and grey literature. This work benefits practitioners interested in understanding the impact of selecting a given MSA pattern, and researchers for further work in this area. This work does not intend to present a classification among the patterns found, but to associate patterns with related quality attributes and metrics.

This paper is an extension from previous work [20], which presented a pattern collection from 2014 to 2018 through a systematic literature review; that is, only white literature was considered. In this extension, the review covers patterns from 2019 as well, and integrates a grey literature revision from 2014 to 2019, making the required adjustments to the methodology for a multivocal literature review (from now on MLR).

An MLR combines academy and industry knowledge, providing a broad perspective of patterns from practitioners in MSA. The knowledge of the academy and the industry is not the same; some patterns have not been spread to academic literature and vice versa. For example, the *bulkhead* pattern [21] is not found in academic literature, while in the industry, it appeared in 2017. The patterns discussed from the industry and academy perspective will provide a more robust understanding of what patterns are used for, determining related quality attributes, and metrics used along with such patterns.

This paper is organized as follows. Section 2 presents related work, i.e., literature reviews of MSA patterns, and describes the differences between such studies and this one. Section 3 states the methodology used as a multivocal literature review, which encompasses research questions, limitations, and selection criteria. Section 4 introduces the findings derived from the application of the methodology, in order to answer research questions and present data interpretation. Section 5 summarizes the most relevant findings and describes future work.

2 RELATED WORK

Although the microservices trend has emerged in the industry since 2014, few secondary studies on MSA have been published. As far as we know, between 2014 and 2019 only [17, 22–24] have been published as secondary studies, either as systematic mapping studies or as systematic literature reviews. Systematic mapping studies and systematic literature reviews, identify and evaluate relevant information with respect to a research question or topic. However, only academic sources, also known as white literature, are included for this kind of secondary studies.

Concerning the MSA area, only [25] has performed an MLR and [26] a grey literature review. However, the previous studies [22, 23] have included in some manner grey literature. To summarize the

inclusion of grey literature in MSA, and the need for an MLR related to patterns, we will briefly discuss the previous four studies and their dissimilarity with the present work.

The first secondary study on MSA is [22], it was published in 2016 and interestingly it included grey literature. The aims of the study were to identify trends and define the research maturity in the area of microservices. Its contributions allowed researchers to detect areas of interest and understand how the information was published. On the other hand, the inclusion of grey literature helped the authors to extract the industry perspective of microservices principles. It should be noted that they only included blogs. The overview of 21 selected studies describes software engineering as the main computer field that contributed to MSA. Other highlights from the overview were: Most of the publications provided a solution or validation to the architecture or method; six design patterns were identified, and the most common publication format were magazine articles and thesis. In summary, [22] was merely an exploration of microservices literature, but it was a good first secondary study because it was not specific to one topic; it analyzed the beginnings of MSA in the literature and identified possible future research. However, only one section included grey literature, and for its inclusion, no guideline was followed; therefore, it cannot be considered as an MLR. The identified design patterns were only mentioned, and it did not provide any relation with quality attributes or metrics.

In 2018, [17] added more information about the principles of the microservices style from 42 primary studies, it detailed the advantages, and disadvantages of microservices, contributing to the principles identified in [22]. Authors of [17] also explored the area of architectural patterns. They presented a catalog of nine patterns, and each pattern was described by a template, which included: concept, origin, properties, evolution, reported usage, advantages, and disadvantages. The catalog was divided into three categories, and the patterns included were:

- Orchestration and coordination: Patterns for communication and coordination from a logical perspective.
  - API-gateway
  - Client-side discovery
  - Server-side discovery
  - Hybrid
- Deployment strategies: Physical strategies and virtual machines
  - Multiple service per host
  - Single service per host
- Data storage: Data management in addition to inter-component communication.
  - Database-per-service
  - Database cluster
  - Shared database server

This work also provides a pattern catalog with a high detail level for the nine patterns. However, it does not relate quality attributes and metrics to patterns. Besides, grey literature was incorporated by forward and backward snowballing and not by a web search engine. This could cause that only ~5% of the selected sources were from grey literature.

In the same year, [26] presented a grey literature review with respect to pains and gains of microservices, that is, technical and operational difficulties and benefits related to design, development, and operation. The analysis of 51 industrial studies provided a taxonomy of pains and gains, it also allowed a comparison between the existing concerns. The pain concerns were: architecture, security, microservices, storage, testing, management, monitoring, and resource consumption. Meanwhile, the gain concerns were: architecture, design patterns, security, microservices, storage, testing, deployment, and management. From the sources selected, 26% included some content regarding patterns but only five patterns were identified as design solutions. Interestingly, the most frequent pattern was database per service. However, no relationship between patterns and quality attributes or metrics was made. Another aspect to highlight is that snowballing method was not included, and quality assessment was partially involved, an important activity suggested by [27], for the inclusion of grey literature.

In 2019, [25] performed an MLR regarding architectural smells indicating design principles violations, and refactoring as a solution to mitigate them. The result was a taxonomy illustrating the relationship between seven architectural smells and 16 refactorings. The refactoring collection represents solutions with different abstraction degrees, some of these can be translated to patterns, i.e., API gateway and circuit breaker. Although this study was not related to metrics, it provides a measurement based on the usage frequency to discriminate when multiple refactorings are possible. Additionally, the snowballing method and quality assessment were not involved as suggested by [27, 28]; no synthesis method was incorporated for result analysis and the study goals were not related to patterns.

In summary, our study differs in conducting a multivocal literature review to identify patterns and with their respective associations with quality attributes and metrics, incorporating the backward snowballing method, a quality assessment, as suggested by [27], and reciprocal translation method for data synthesis.

3 REVIEW PROTOCOL

As we previously examined in section 2, to the best of our knowledge, there is not an MLR about patterns in the MSA area. Therefore, this work proposes an

MLR to add evidence from grey literature, supplying useful insights to researchers from the industrial community.

For this, we used the guidelines provided by [27] for including grey literature and conducting multivocal literature reviews in software engineering. The phases proposed are based in [28], which has the main guidelines for systematic literature reviews in software engineering. Planning the review is the first phase to identify the need for a review and to specify the research questions. The second phase conducting the review involves: establishing the search strategy and source selection criteria, source selection, study quality assessment for grey literature, data extraction, and data synthesis. Finally, in the third phase, the results are reported answering the research questions.

3.1. Planning the Review

The primary motivation to perform an MLR is to improve, from industry knowledge, the pattern information presented in our previous study, extending the relationship between patterns and quality attributes, and exploring metrics related to patterns in the industry. The review is guided by three research questions (RQ) about design and architectural patterns in microservices.

RQ1. What patterns are identified in microservices systems?

RQ2. Which quality attributes are related to the patterns used in microservices?

RQ3. What metrics are used to quantify quality attributes related to a microservices pattern?

RQ1 aims to identify microservices patterns directly. Answering **RQ2** is the main interest of this study, as it allows, from the characteristics that make each pattern different, the incorporation of detailed information about quality attributes, helping to clarify the inclusion of patterns for specific problems. **RQ3** explores metrics for measuring quality attribute satisfaction through a quantitative method, bolstering the understanding of the relation between MSA patterns and quality attributes.

4 CONDUCTING PROCESS

The conducting process includes a manual and automatic search process; source selection, including different sources for grey and white literature; a study of quality assessment for grey literature; data extraction; and data synthesis using the meta-ethnographic method reciprocal translation [29], which improves the interpretation of qualitative information.

4.1. Search Strategy and Source Selection Criteria

Automated search was used to collect relevant information in white literature in the following indexing services: Scopus, Science Direct, Springer Link, IEEE Xplore, and ACM. These academic sources were chosen due to their relevance in software engineering. Meanwhile, for grey literature, the Google search engine was used. In order to reduce the huge number of results (2.640.000), and based on [27] recommendation, we decided to use Google's PageRank algorithm and theoretical saturation as stopping criteria, which is the inclusion of more sources only when new concepts are found.

In order to limit the results, the following criteria were applied:

Inclusion criteria:

• Publications between 2014 and 2019. The initial date was determined by the year of the first microservices study known in the literature.

• Studies must be written in English.

• Publication type. For grey literature, blogs were included to improve source quality and to facilitate evaluation.

Exclusion criteria:

• Book or book chapter. Due to the limitations of resources and time, materials of this type were not considered.

For automated search in white literature, a single search query was used. The query was formed by three blocks. The first block was associated to **RQ1**, the second one to **RQ2**, and the third one to **RQ3**. Using this single query allowed the retrieval of studies associated with any of the three research questions.

Search query used for white literature:

((“Microservices” OR “Service-oriented Architecture” OR “Micro-service*” OR “Micro Service*” OR “Microservices Architecture”) AND (“Architectural Pattern*” OR “Pattern*” OR “Architectural Style” OR “Architectural Tactic”)

AND

(“Quality Attribute*” OR “Quality” OR “Architectural Driver*” OR “Quality Requirement*” OR “QA” OR “Quality Attribute Requirement*”)

AND

(“Metric*” OR “Response Measure” OR “Measure*” OR “Checklist” OR “Analytic Model” OR “Instrumentation” OR “Evaluation”))

For automated search in grey literature, a simplified query was used, only main keywords were considered. This helped to increase the number of results since in an initial search the same search query of the white literature was applied, but that query hardly returned results. Additionally, to obtain more relevant results and reduce the noise retrieved from the engine, Google's PageRank algorithm was used.

Simplified query associated with **RQ1**, **RQ2**, and **RQ3**:

(Implementing or Developing) microservices or microservices design patterns

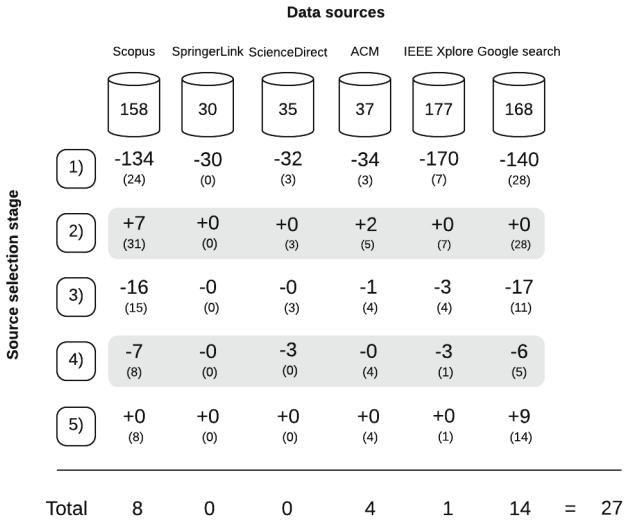

Fig. 1. Data sources frequency against source selection stages. Each intersection represents the result between source selection stage and data source, and the numbers enclosed in parentheses highlight the remaining studies after each selection stage.

In addition to the automated search, manual research was incorporated using the backward snowballing method [30]. The inclusion criteria for backward snowballing in white literature was to be referenced in at least two studies and fulfill all the previous requirements. Meanwhile, for grey literature, the minimum of two references was not requested, in order to have more possible publications. However, to improve the inclusion of quality sources, backward snowballing was added as a final step.

4.2. Source Selection

The selection process was made in five stages:

1 Exclusion by title.

2 Inclusion by backward snowballing for white literature.

3 Exclusion by checklist for white literature/Exclusion by quality assessment for grey literature.

4 Exclusion by the answer to research questions.

5 Inclusion by backward snowballing for grey literature.

Table 1. Stage 3 checklist

Concern
—Does the text potentially answer a research question?
—Does the text demonstrate clarity in the purpose of the investigation?
—Was the investigation validated?
—Does the study focus specifically on patterns in microservices?

In order to follow the guidelines for including grey literature [27], we adjusted the selection process. A quality assessment was performed for reducing bias, enforcing the validation of the source for grey literature.

In relation to the execution process, an automated search was carried out to the five information sources and search engine, retrieving 605 results matching the search strategy filters, as is shown in Fig. 1. After that, the exclusion for stage 1 discarded any document with titles not related to a research question, this reduced the number of publications to 65. Then, in stage 2 for white literature, we found nine studies from the 65 studies retrieved. Although due to the possibility of answering a research question, an exception to the inclusion criteria for backward snowballing was made for six of the nine studies found, these studies were cited by only one document. Table 1 illustrates the checklist used in stage 3, this checklist was used to identify the studies related to a research question and to evaluate the study quality, only studies with three or four points were selected, 20 studies were discarded. Meanwhile, the quality assessment for grey literature removed 17 entries. After the exclusion from the checklist and the quality assessment, the total number of studies was reduced to 37. Finally, in stage 4, a complete reading of the studies was carried out, and the answer to at least one research question was verified by another checklist, cutting the final list to 18. On the other hand, backward snowballing for grey literature incorporated nine publications, yielding a total of 27 sources for information extraction.

4.3. Quality Assessment

In grey literature, there is no quality control over the material that will be uploaded, and there are no institutions that review whether the content is valid and free of bias. Consequently, the materials found in this type of literature are diverse. They can be found in blogs of renowned companies (e.g., Amazon, Microsoft, Google), and in the same search blogs of people sharing their opinion. Therefore, researchers must ensure the quality of the data included. To perform this process, we applied the checklist suggested by [27], to each of the sources from grey literature.[1]

The checklist comprises 20 questions, which are organized in the following criteria: Authority of the producer, methodology, objectivity, date, position related sources, novelty, and impact. Each question is assigned a value depending on its answer (Yes = 1, Partly = 0.5, No = 0). To conclude the process, the values obtained are added, and the value is then normalized (sum of the obtained values/20). The normalized result determines the tier of control and credibility of the source, according to [27] there are three tiers:

[1] http://www.uv.mx/personal/jocharan/files/2020/05/Quality-assessment.pdf

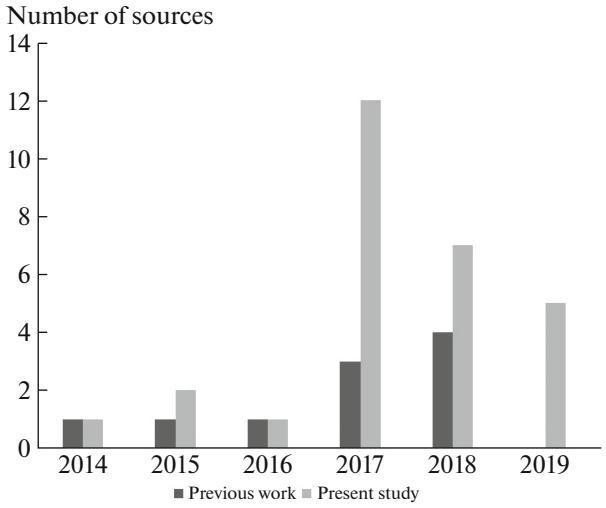

Fig. 2. Microservice patterns trend, showing also results from our previous study [20], which does not cover 2019 and does not include grey literature.

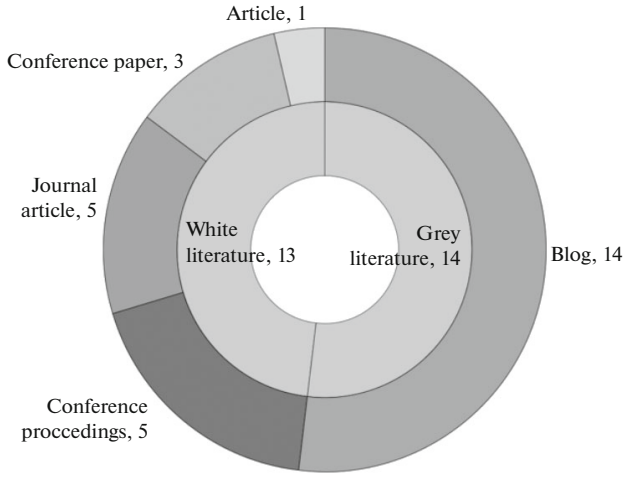

Fig. 3. Literature type identified in the sources analyzed.

• 1st tier (measure = 1): High outlet control/High credibility.

• 2nd tier GL (measure = 0.5): Moderate outlet control/Moderate credibility.

• 3rd tier GL (measure = 0): Low outlet control/Low credibility.

4.4. Data Extraction and Data Synthesis

Information extraction consisted of filling a table about the general paper information; for example, title, year, abstract, keywords, and personal notes. Besides, another special table for each answer related to a research question, giving a total of 82 tables with information. With respect to the synthesis process, we identified 11 concepts from the last 82 tables, grouping the information was by concept in ascending chronological order.

5 RESULTS AND DISCUSSION

After conducting the research, we can draw a perspective concerning the interest in microservices patterns from industry and academy. The frequency of the selected sources is illustrated in Fig. 2. Interestingly, in our previous work [20], only considering white literature, the highest frequency was in 2018, but including grey literature, the highest frequency was in 2017. Additionally, as shown in Fig. 3, grey literature inclusion represented almost half of the total frequency of the sources, even though only blogs were considered. Meanwhile, journal articles and conference proceedings were the most frequent formats in white literature.

From the final sources, 54 patterns were identified. Despite the number of patterns, it should be noted that some patterns represent variations or specialized implementations of other patterns. For example, *gateway aggregation* and *gateway offloading* are specialized implementations of *API gateway*, these patterns introduce a specialized endpoint, hiding the complexity from the client. On the other hand, some patterns have a simple implementation, and therefore their abstraction degree and scope could differ from what would be expected from a pattern. For example, *health check* helps to track microservices state, so compared to other patterns, its complexity is low. We also identified three redundant patterns: *Change code dependency to service call* with *adapter microservice*, *messaging* with *event-driven messaging*, and *load balancer* with *load-balancing*.

Regarding quality attributes, we used the software product quality model defined in ISO/IEC 25010 [31] as a reference. This model comprises eight quality characteristics. However, we only found six characteristics associated with microservice patterns. As we can see in Fig. 4, the quality characteristics associated with descending order were maintainability, reliability, security, performance efficiency, compatibility, and portability. No association with functional suitability and usability was found. Some of the patterns are related to more than one attribute.

*5.1 **RQ1** Analysis*

Concerning what patterns are identified in microservices systems, Table 2 presents the patterns grouped by benefits, each pattern is mentioned in chronologically ascending order. Most of the patterns have been identified in the literature for microservices, however, some patterns are also related to SOA, which could be considered as a more general implementation of a distributed service architecture [8]. For example, *health check, asynchronous messaging, service discovery, circuit breaker*, and *monitor*. This is not surprising since microservices can be understood as an evolution of

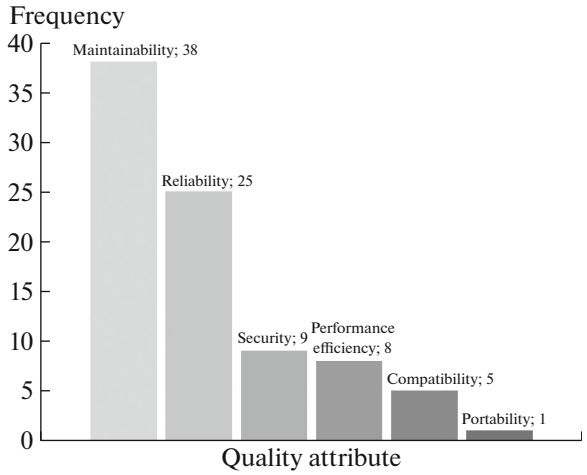

Fig. 4. Quality attributes frequency related to the identified patterns.

SOA. Being SOA a non-centralized architectural style, based on the communication between services, in order to offer functionality, including consuming third-party services [32]. However, SOA has challenges in communication, middleware, and granularity; microservices, as an alternative, have been conceived to mitigate these challenges [8].

The most frequently identified patterns are related to solving communication problems and coordination between services – for example, *asynchronous messaging, service registry*, and *request-reaction*. In the literature, the most frequent patterns found were *API gateway, backend for frontend*, and *service registry*. As the term design and architectural pattern are not clearly distinguishable among authors, some patterns were classified in both categories, and, in some cases, under the general pattern term. For example, the following terms were found under the pattern category: *Enable continuous integration* [19] and *container* [19, 33]. *Continuous integration* is a practice related to agile software development. This mitigates the so-called "big-bang integration" using tools for code integration, running test suites, and building deployment artifacts [34]. Therefore, it implies development activities not directly related to system structures. The abstraction degree of *container* might be limited by the technology since this is a specialized implementation from the architecture style virtualization [32].

During data synthesis, five main benefits of patterns were identified. These benefits allowed us to group each pattern into one of the following groups: Data persistence, communication, entry point, distribution, fault-tolerant, and supplementals. Each group represents the shared benefits among patterns and facilitates the interpretation of the results, but such groups do not intend to be a classification proposal. In regard to source frequency by literature type, Fig. 5 illustrates the frequency per group. White literature represents the main source in all groups, in some cases, it represents ~75% of the sources cited in the group. Despite the number, this only represents a higher number of patterns reported in white literature than in grey literature, since total source frequency is distributed almost equally.

The following subsections examine the five pattern groupings. Due to space limitations, we decided to focus on the research objective, tying patterns together with quality attributes and metrics, however, within these groups each pattern is briefly explained. Additionally, references for each pattern are included for more details, although not all patterns have a detailed description. The description of each grouping of patterns begins with their benefits and disadvantages, shared among the grouped patterns. Then, the dissimilation is drawn by comparing each pattern with one another. Each pattern is presented in chronologically ascending order, as previously presented in Table 2.

5.1.1. Data persistence. This group consists of patterns oriented to improve the management of data-

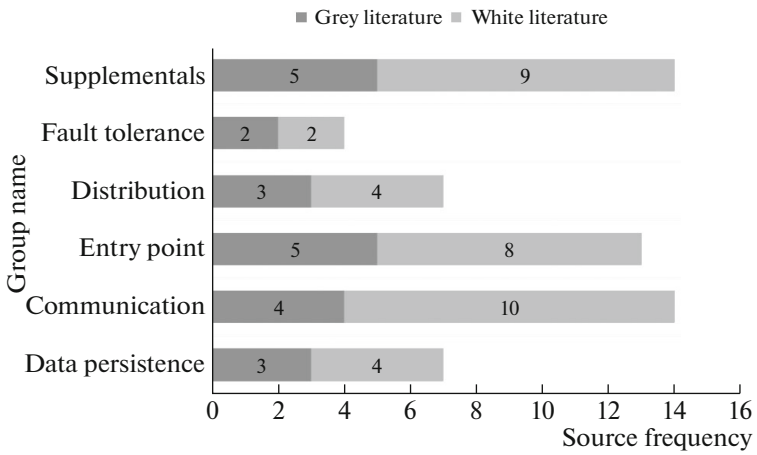

Fig. 5. Literature type related to patterns groups.

Table 2. Microservice patterns grouped by benefits and related quality attributes

Group name	*Quality attribute*
Data persistence	
Local database proxy	Maintainability and reliability
Local sharding-based router	Maintainability and performance efficiency
Command and Query Responsibility Separation	Maintainability, performance efficiency, reliability, and security
Event sourcing	Security* and performance efficiency*
Scalable store	Maintainability and reliability
Communication	
Secure channel	Security
Change code dependency to service call/adapter microservice	Compatibility
Service registry	Portability, maintainability, and reliability
Service discovery	Security and reliability
Service locator	Security and maintainability*
Event notification	Reliability
Service registry client	Maintainability and reliability
REST integration	Maintainability and compatibility*
Competing consumers	Maintainability
Pipes and filters	Reliability and maintainability
Asynchronous messaging	Reliability, maintainability, and performance efficiency
Request-reaction	Maintainability*
Entry point	
API gateway	Maintainability, reliability, performance efficiency, and security
Backend for frontend	Compatibility and maintainability
Auth-service	Reliability
Gatekeeper	Security
Distribution	
Strangler	Compatibility
Anti-corruption layer	Compatibility
Externalized configuration	Security
Self-containment of services	Maintainability*
Container	Maintainability and reliability
Deploy cluster and orchestrate containers	Reliability and maintainability
Microservices DevOps	Maintainability*
Database is the service	Maintainability and reliability
Enable continuous integration	Maintainability
Self-contained systems	Maintainability and reliability
Fault tolerance	
Bulkhead	Reliability
Circuit breaker	Maintainability and reliability
Supplementals	
Priority queue	Maintainability
Results cache	Performance efficiency
Page cache	Performance efficiency
Key-Value store	Security and reliability
Correlation ID	Maintainability
Log aggregator	Maintainability and performance efficiency

Table 2. (Contd.)

Group name	*Quality attribute*
Load balancer/load-balancing	Reliability and maintainability
Ambassador	Reliability and maintainability
Sidecar	Maintainability
Gateway aggregator	Maintainability
Gateway Offloading	Maintainability and reliability
Asynchronous query	Reliability
Asynchronous completion token	Reliability
Edge server	Maintainability and reliability
Internal load balancer	Maintainability and reliability
External load balancer	Maintainability and reliability
Health check	Maintainability
Monitor	Maintainability
Consumer-driven contracts	Maintainability*
Tolerant reader	Maintainability*
Aggregator	Maintainability

* = Not reported but inferred by the literature

bases, allowing information independence and scalability. The shared benefits are non-blocking conditions in case of failure, and dynamical instances lifecycle. However, these patterns can increase the complexity of the system or have limitations in complex queries.

In 2014, *local database proxy* and *local sharding-based router* appeared in [35]. The former is useful for applications that have heavy reading loads, but is limited for large numbers of write operations, whereas the latter enables a larger number of writing and reading operations. In 2018 more information was included in [33] about both patterns. The *proxy* pattern provides flexibility at execution time since its master and slave principle allows to add or remove instances. On the other hand, in the *router* pattern, the sharing logic can be achieved using multiple strategies; even the work can be balanced through shards. In 2018 *Command and Query Responsibility Separation (CQRS)* and *event sourcing* [36] were introduced. The *CQRS* pattern separates the problem of data consistency into two processes, one to read and another to write, this, in turn, provides two asynchronous actions. *Event sourcing* suggests recording the sequence of events in a transaction instead of saving the data directly, in order to improve consistency in distributed transactions. In the same year, *scalable store* was mentioned in [19]; this pattern improves scalability, distributing horizontally all states in stores. Finally, in 2019, two patterns were found again, *event sourcing* [37, 38] and *CQRS* [38, 39]. More information was added only for *CQRS*; read and write separation requires constant updating information to avoid reading stale information. Also, it was not recommended the inclusion of *CQRS* and *event sourcing* because it can lead to a more complex application.

5.1.2. Communication. The patterns in this group improve communication and coordination between services or provide a communication channel. This benefits services identification and reduces complexity in large systems. Nonetheless, these patterns may cause unnecessary complexity for small systems, or a single point of failure.

Patterns in this group started to appear in 2015. *Secure channel* was the first pattern found in [40], which generally allows a secure communication path for services. The next year, in [41], the patterns found were *adapter microservice* and *service registry*. The first one creates a service as an intermediary to incorporate segments that are not services yet, it can be a simple solution when a system is being migrated, or an external system must be integrated. The second pattern enables decoupling the current physical service address creating a unique service identifier. In 2017 *service discovery* was introduced in [42], which allows dynamic identification of services instances through metadata. Also, tactics can be incorporated to check the status of the registered services. In 2018, 10 patterns were identified. First, *service locator* and *event notification* were described in [43]. The first pattern is useful to improve security and the second one for reliability. In the same year, *service registry* was found again in [18, 19, 44]. A benefit of this pattern is that it pulls all services instances in one place. However, this could be interpreted as a single point of failure. Also, in [44] *service registry client* was included as a variant; this one implements tactics to improve service control. One tactic example is *heartbeat*, which confirms the

status of the service, and allows to decide if it is necessary to keep the service in the registry. In [18] *change code dependency to service call* can be identified as an adapter counterpart of *adapter microservice*, since it does the same but with a different objective; this pattern is described as a solution to avoid dependencies, isolating the code in a service. Additionally, in the same source, *REST integration* was mentioned as a solution to expose communication points between services. Again in 2018, *competing consumers* [33] enables the management of flexible workloads through the deployment and coordination of consumer instances, it guarantees that failure will not result in blocking a producer. In addition, the same source recommended the use of *pipes and filters* for the decomposition of complex processes, and the improvement of reliability, because if a part fails, it can be reprogrammed to complete the task. *Service discovery* was mentioned again in [19]; it specifies that each service must register itself during initiation. In [19, 36, 45] *asynchronous messaging* was described as an inter-service communication way for the asynchronous process.

In 2019, *asynchronous messaging* in [37, 38, 46] was identified again; unfortunately, the support to distributed communication was only mentioned. *Service registry* was mentioned again in [37, 46] no new information was added. Finally, a combined implementation of *request-reaction* and *asynchronous messaging* was suggested in [37].

5.1.3. Entry point. The patterns in this group are oriented to supply control access and entry point to services or types of backends. However, all patterns have the disadvantage of being a single point of failure and in some cases, they are inconvenient for high workloads.

In 2015 the first pattern was identified in [47]. *API gateway* is the most popular in this group; it enables a client to retrieve details of a service in a single query through an encapsulated structure; one disadvantage is the risk to become a bottleneck. Additionally, the source suggests a combined implementation with *service discovery*. In 2016, *Backend for frontend* was mentioned in [41]; this pattern creates a special backend for each client type. In this way, the clients can be grouped and redirected to a specific channel. Also, this pattern is useful as a verification point in the design stage, since all the services must be related to at least one consumer type. In 2017, four studies [42, 48–50] described the *API gateway* advantages: to provide an authentication solution, to hide the real instance from the client, to contain changes in the backend, to monitor the workload, and to support load balancing. In contrast, it must be easily scaled to handle high workloads, and it can be a single point of failure. [42, 51] confirmed the importance of *backend for frontend*, providing a smaller endpoint, with less complexity, being faster than a generic backend for all the interfaces; also, it increases the frontend team autonomy, having control of their own backend, and reduces the load of the *gateway*. In [48], *auth-service* was described. This pattern separates information from users with reduced functionality and improves the response time of the authorization process. In 2018, *backend for frontend* was identified again in [18, 19]. These sources strengthen the benefit of a detached frontend from the backend since a service can be easily replaced, reducing changes in the frontend. In [19, 36], *API gateway* was named again. In [33], *gatekeeper* was identified to support applications that handle sensitive information, since it is responsible for verifying identities and requests, and redirecting requests to reliable instances. Finally, in 2019, *backend for frontend* was identified again in [37]; its implementation is documented to prevent too many concurrent long-running HTTP requests. In [37, 38] *API gateway* was reported as a solution to aggregate the results back when multiple servers interact in the backend.

5.1.4. Distribution. Patterns in this group improve the distribution of services in a logical way or propose a particular organization of structures to integrate another part of the system. With some exceptions, it may be difficult to fully implement the patterns in this group.

In 2017, *strangler* was described in [52]. This pattern helps to reduce the migration risk and distributes the migration effort over time; this implementation can be a bottleneck or a single point of failure if the integration is exposed in only one interface. In [53] *anti-corruption layer* was presented to isolate subsystems by placing a layer between them; this layer provides a communication channel, avoiding design or implementation alterations on one side. Nevertheless, this implementation adds latency to calls between the subsystems and requires additional maintenance. In [42] the *externalized configuration* pattern was mentioned, it is used to maintain main configurations outside the services, and to allow only read requests. In 2018 in [18], four patterns were identified. *Self-containment of services* enhances autonomy by decoupling services and gathering the required libraries. Then, *container* was mentioned as a technological option for the compliance of the first pattern; in this way, the resources and deployment can be restricted; additionally, better testability and scalability were reported in [19]. Following this trend, *deploy cluster and orchestrate containers* continues the principles of the last patterns and constitutes the option for managing multiple instances. In the same source, *microservices DevOps* was described as a pattern for the identification and development of services. [19] also described *database is the service* and *enable continuous integration*. The former pattern suggests separating the database in small portions for each microservice. The later pattern introduces a series of activities to accelerate the integration: automate the process, support for the production of ready artifacts, supply a pipeline from each service repository for building and testing. However, most of the latest patterns maintain a granularity degree that could question their identification as pat-

terns. In 2019, *strangler* was mentioned again in [37, 38]. This pattern is identified for extending or migrating gradually an existing monolith to new services until its final replacement. Finally, in [37] *self-contained systems* pattern was described to achieve vertical isolation between subsystems, the general implementation of *self-containment of services* can be inferred.

5.1.5. Fault tolerance. In this group, patterns solve problems related to fault tolerance. According to [31], fault tolerance is the "degree to which a system, product, or component operates as intended despite the presence of hardware or software faults". Patterns within this group improve reliability by isolating component errors, but add some problems, for example, complexity and overhead to the system.

Only *bulkhead* and *circuit breaker* were included in this group. *Bulkhead* was mentioned in [21] in 2017. This pattern isolates consumers and services from cascading failures; one instance of this pattern implements a pool of services, so in case of failure, just one may fail, and functionality will be preserved. However, prior to development, the granularity level must be defined. In 2018 *circuit breaker* was identified in [19], it also appeared the next year in [38, 46]. This pattern is used to stop a faulty service, it checks the recent service responses and acts when the number of failures passes a predefined threshold. To restart the service, after a given timeout, it checks the service status, in case of a successful restart, the state will be changed. The stopping action isolates the failure, and the restart criteria prevent sending requests when the service is not available.

5.1.6. Supplementals. According to the literature, the following patterns can be used to complement and improve another pattern since their implementation is very specific and oriented to solve a particular problem. These patterns fall into 10 cases. Table 3 describes the relationship between patterns and their supplementals.

The rest of the patterns that conform to this group were not documented by supplementing another pattern, but these represent specific solutions to concrete problems. Also, given the low complexity of the implementation, it is possible that the inclusion of one of these patterns does not solve the problem completely. It should be noted that excessive use of these patterns can add unrequired complexity to the design.

In 2014, the first supplemental pattern was identified in [35]. The *priority queue* benefits improving asynchronous communication between the components when different types of messages are involved. Two years later in [41] four simple patterns were identified to improve the storage and retrieval of information. The first two patterns, *results cache* and *page cache* help to improve performance in retrieval of information. Although both patterns have weaknesses, *results cache* may not recover up to date information, while *page cache* is not appropriate for mobile environments given the long datasets retrieved. Regarding the last two patterns, it is possible to refine the service or request identification by implementing *key-value store* or *correlation ID*. Additionally, in the same source, the pattern *log aggregator* was described, this collects the log files of a single element allowing to keep track of the events in one place, this is helpful to check microservices health.

Table 3. Patterns supplemented with another specific pattern. For some patterns, the supplemental role is defined by the context of the problem, the relationship and role are not strict

Main pattern	Supplemental pattern
• Backends for frontend	• Correlation ID
• Change code dependency to service call	• Log aggregator • Key-value store
• Adapter microservice	• Page cache • Results cache
• Local database proxy	• Adapter microservice
• Local sharding-based router	
• Load balancer	• Circuit breaker • Health check
• Request-reaction	• Asynchronous messaging
• Command and Query Responsibility Separation	• Event sourcing
• API gateway	• Service discovery
• Ambassador	• Sidecar
• Bulkhead	• Circuit breaker
• Gateway aggregator	• Bulkhead • Circuit breaker
• Sidecar	• Container • Ambassador

During 2017, [42, 50] mentioned *load balancing* pattern. This pattern handles several requests with different time frames, distributing the workload. Also, it can be implemented with several algorithms, but round-robin is the most beneficial. In [54] *ambassador* was described. This pattern creates helper services that send network requests on behalf of another entity. This pattern is useful for integrating systems with limited network capabilities. In the same year, *sidecar* was found in [55]. This pattern separates secondary functions or components (*sidecar*) from the core functionality, but it is connected to the core for all the lifecycle. In this way, the *sidecar* can access the resources through an insignificant latency, providing isolation and encapsulation. In [56, 57] two *gateway* specific implementations were mentioned: *gateway aggregation* and *gateway offloading*. The first pattern aggregates multiple requests into one single request, reducing chattiness between the client. However, it may introduce a bottleneck and a single point of failure. The second pattern simplifies the development of services

by adding a shared proxy and allowing the implementation of specialized *gateways*.

In 2018 *asynchronous query* and *asynchronous completion token* were identified in [43]. These patterns seek to improve performance. The same year, *edge server* was described in [44]. This pattern obscures the server structure details to reduce complexity; in this way, dynamic routes can be established and the information can be tracked. In the same source, two variants of *load balancing* previously mentioned in 2017 were described. The *internal load balancer* controls the load of multiple service instances and holds an entry of them. In addition, local metrics, and algorithms can be implemented to define the best instance. On the other hand, *external load balancer* usually uses the *services registry* list because its scope is not specific to one kind of instance. Therefore, it can not consume local metrics, but algorithms can be implemented to improve its work. In the same way, in 2018, seven patterns were identified in [19]. The following five patterns were already mentioned: *Key-value store*, *load balancer*, *log aggregator*, *page cache*, and *result cache*. Regarding *page cache*, this source suggests implementing it on mobile applications, meanwhile in [41] does not recommend it. Also in the same source, *health check* and *monitor* were described. The first pattern tracks the service status implemented by some tactic, the information usually is exposed by an API. The second pattern gathers important information and sends it to a monitoring instance. In [33], *priority queue* was identified again. The importance of improving messages communication with different degrees of priority is reaffirmed.

In 2019, *consumer-driven contracts* and *tolerant reader* were mentioned in [37]. Both patterns provide a service interface to prepare consumers for future changes and support interface evolution. Finally in [38] *aggregator* was described, this pattern benefits combining data from multiple services. It can be considered as a general pattern for other specifics solutions, for example, *log aggregator*.

5.2 ***RQ2*** *Analysis*

As presented previously, RQ2 is concerned with quality attributes related to the patterns used in microservices. The quality attributes identified in the literature correspond to those knowingly related to MSA. Table 2 presents the quality attributes associated with each identified pattern. To perform the mapping process, we identified directly in the source the attribute related to the pattern. However, not all the patterns found had a documented association with a quality attribute; for those cases, we analyzed the characteristics and consulted a secondary researcher with microservices experience.

In relation to Fig. 4, the first two most frequent quality attributes represent almost 75% of the total, and the rest is distributed in four attributes. The high frequency of maintainability and reliability can be considered as a result of the fast and easy deployment principle of MSA, and the attention to deliver functionality satisfying quality attributes. However, not all MSA areas were related to patterns. For example, the maintainability characteristic was the most frequent quality attribute, but no pattern description mentioned the sub-characteristic testability; despite the fact that testing is one of the most important areas in MSA. Additionally, some patterns are related to three or four quality attributes; this might call into question how much each quality attribute is satisfied considering their trade-offs.

5.3 ***RQ3*** *Analysis*

Finally, RQ3 is about the metrics used to quantify quality attributes related to microservices patterns. The metrics identified in the literature associated with the evaluation of a quality attribute in a pattern were:

- Time (performance efficiency);
- Percentage of requests accepted (interoperability);
- Number of files to be modified (maintainability);
- Energy consumption (performance efficiency).

Three special aspects concerning time were identified: the response time to fulfill a number of requests per second, the interaction time in milliseconds (time before the user can interact with the GUI), and the response time in milliseconds. With respect to time in 2014, the patterns *local database proxy, local sharding-based router*, and *priority queue* were analyzed in [35]. The three patterns have a positive impact with respect to reading and writing requests. The first pattern is more suitable for reading requests, the second one for writing, and the last one has a moderately positive effect on both. It is also appropriate to combine the last pattern with one of the first two patterns. Besides, in these three patterns, the choice of the algorithms does not represent a big difference since the key point is to make the right pattern decision.

In 2017 *Self-containment* and *API gateway* patterns were evaluated for modifiability in [48]. The first pattern is better than the second one in terms of the number of files to modify and the percentage of acceptances. The *API gateway* has weaknesses when facing high workloads. Regarding the number of milliseconds with a low number of requests, both patterns obtained a similar time, but in cases with high, *API gateway* takes a little more time.

In 2018, *priority queue*, *local proxy database*, *local sharding-based router*, *pipes and filters*, *competing consumers*, and *gatekeeper* were evaluated in energy consumption in [33]. The energy consumption was measured at process-level using Power-API, an application programming interface (API) written in Java to monitor the energy consumed. In general, the five patterns improve the reduction of energy consumption

measured in kilojoules and time measured in milliseconds, improving performance efficiency in microservices compared to monolithic systems. *Pipes and Filters* improves the performance time and the reduction of energy consumption. Surprisingly, this pattern does not achieve the same benefits in monolithic systems. Likewise, these benefits can be achieved with *gatekeeper* and *competing consumers*, but *gatekeeper* isolated implementation can negatively affect response time and energy consumption. The combination of *local database proxy* and *competing consumers* results in a negative impact on response time, as does the combination of *local sharding-based router*, *competing consumers*, and *pipes and filters*.

Unfortunately, the analysis of information published in 2019, and the industry perspective inclusion did not provide any new metrics from what we reported in [20]. According to the results from grey literature, there may not be a quantitative measurement when choosing a pattern, and it is possible to question, that for now, the only experience is considered. In consequence, it can be inferred that the maturity level of MSA metrics is low in grey literature.

6 CONCLUSIONS

Conducting an MLR helped to expand the list of patterns presented in our previous work [20]. With the inclusion of grey literature publications 20 new patterns were found. In addition, certain sources from grey literature presented extra information about patterns, since authors are not limited in terms of number of pages or words. For instance, Microsoft details the following aspects of each pattern: context, problem, solution, issues, considerations and, in some cases, examples with code.

In total, 54 patterns reported in white and grey literature were identified. The most frequent patterns are oriented to solve communication problems and to improve performance, this can be related to well-known MSA quality attributes and the MSA's build and ready to deploy the microservice principle. Maintainability and reliability were the most frequent quality attributes related to patterns. Interestingly, functional suitability and usability are not associated with patterns, this suggests future work to identify what solutions are proposed in MSA to satisfy these attributes. Meanwhile, metrics was an area not reported in grey literature, however, it could be interesting to explore what other characteristics are used to select a solution in the industry.

Dealing with grey literature involves extra work, for example, the checklist proposed by [27] for quality assessment. This activity must be done with each of the selected publications from grey literature. Some of the questions on the checklist are subjective and others need extra searches. Another drawback is the huge number of search results in grey literature. In the first stage of searching 2.640.000 results were found, even though a search string was used. In order to analyze the most relevant results for the research, we used the Google ranking algorithm. However, irrelevant material was still found, for example, links to purchase books, courses, or workshops.

We realized that in white literature the explanations about patterns were brief and not direct, the authors do not usually explain the context and the method used to obtain the information. In contrast, in grey literature authors were more direct, if the topic needs some explanation of context, they attach links to other resources that explain it. Another point is that the abstraction degree of the term "pattern" may not be clear in the literature. Some of the patterns mentioned in the primary studies are not rigorously patterns, for example, *Enable continuous integration* [19] and *container* [19, 33].

In summary, the inclusion of grey literature brought benefits such as adding new patterns to the catalog, but extra work was also added for quality assurance. The presented list of patterns should not be considered as a final list, although the number of patterns increased. For future work the inclusion of more types of publications is expected; for example, books, conferences, and white papers, also exploring different ways to reduce the number of irrelevant publications for grey literature.

REFERENCES

1. Grushin, D.A. and Kuzyurin, N.N., On effective scheduling in computing clusters, *Program. Comput. Software,* 2019, vol. 45, no. 7, pp. 398–404.
2. Shabanov, B.M. and Samovarov, O.I., Building the software-defined data center, *Program. Comput. Software,* 2019, vol. 45, no. 8, pp. 458–466.
3. Voevodin, V.V. and Popova, N.N., Infrastructure of supercomputing technologies, *Program. Comput. Software,* 2019, vol. 45, no. 3, pp. 89–95.
4. Kryukov, A.P. and Demichev, A.P., Decentralized data storages: technologies of construction, *Program. Comput. Software,* 2018, vol. 44, no. 5, pp. 303–315.
5. Hoff, T., Amazon architecture, *High Scalability,* 2007. http://highscalability.com/blog/2007/9/18/amazonarchitecture.html. Accessed Nov. 7, 2018.
6. Netflix Conductor: a microservices orchestrator, *Netflix Techblog,* 2016. https://medium.com/netflix-techblog/netflix-conductor-amicroservices-orchestrator-2e8d4771bf40. Accessed Nov. 7, 2018.
7. Calçado, P., Building products at SoundCloud – part I: dealing with the monolith, *Developers Soundcloud,* 2014. https://developers.soundcloud.com/blog/building-productsat-soundcloud-part-1-dealing-with-the-monolith. Accessed Nov. 3, 2018.
8. Newman, S., *Building Microservices: Designing Fine-Grained Systems,* 1st ed., O'Reilly Media, 2015.
9. Lewis, J. and Fowler, M., Microservices – a definition of this new architectural term, *Martinfowler.Com,* 2014. http://martinfowler.com/articles/microservices.html.

10. Clements, P., Kazman, R., and Klein, M., *Evaluating Software Architectures: Methods and Case Studies,* Boston, MA: Addison-Wesley, 2001.
11. Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice,* 3rd ed., Addison-Wesley Professional, 2012.
12. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Boston, MA: Addison-Wesley Longman Publ. Co. Inc., 1995.
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern-Oriented Software Architecture,* vol. 1: *A System of Patterns,* Wiley, 1996.
14. Schmidt, D.C., Stal, M., Rohnert, H., and Buschmann, F., *Pattern-Oriented Software Architecture,* **vol. 2**: *Patterns for Concurrent and Networked Objects,* Chichester: Wiley, 2000.
15. Müller, K.H. and Paulish, D.J., *Software Metrics: a Practitioner's Guide to Improved Product Development,* IEEE Computer Society Press, 1993.
16. Garlan, D., et al., *Documenting Software Architectures: Views and Beyond,* 2nd ed., Addison-Wesley Professional, 2010.
17. Taibi, D., Lenarduzzi, V., and Pahl, C., Architectural patterns for microservices: a systematic mapping study, *Proc. 8th Int. Conf. on Cloud Computing and Services Science,* Funchal, 2018, pp. 221–232.
18. Osses, F., Márquez, G., and Astudillo, H., Poster: exploration of academic and industrial evidence about architectural tactics and patterns in microservices, *Proc. Int. Conf. on Software Engineering,* Göteborg, 2018, pp. 256–257.
19. Marquez, G. and Astudillo, H., Actual use of architectural patterns in microservices-based open source projects, *Proc. Asia-Pacific Software Engineering Conf. APSEC,* Nara, 2018, vol. 2018-Decem, pp. 31–40.
20. Valdivia, J.A., Limyn, X., and Cortes-Verdin, K., Quality attributes in patterns related to microservice architecture: a systematic literature review, *Proc. 7th Int. Conf. in Software Engineering Research and Innovation (CONISOFT),* Mexico, 2019, pp. 181–190.
21. Microsoft, Bulkhead pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead. Accessed Jan. 13, 2020.
22. Pahl, C. and Jamshidi, P., Microservices: a systematic mapping study, *Proc. 6th Int. Conf. on Cloud Computing and Services Science,* Rome, 2016, vol. 1, pp. 137–146.
23. Alshuqayran, N., Ali, N., and Evans, R., A systematic mapping study in microservice architecture, *Proc. 9th IEEE Int. Conf. on Service-Oriented Computing and Applications, SOCA 2016,* Macau, 2016, pp. 44–51.
24. Vural, H., Koyuncu, M., and Guney, S., A systematic literature review on microservices, in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics),* 2017, **vol. 10409** LNCS, pp. 203–217.
25. Neri, D., Soldani, J., Zimmermann, O., and Brogi, A., Design principles, architectural smells and refactorings for microservices: a multivocal review, *Software-Intensive Cyber-Phys. Syst.,* 2020, vol. **35**, pp. 3–15.
26. Soldani, J., Tamburri, D.A., and Van Den Heuvel, W.J., The pains and gains of microservices: a systematic grey literature review, *J. Syst. Software,* 2018, vol. 146, pp. 215–232.
27. Garousi, V., Felderer, M., and Mäntylä, M.V., Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, *Inf. Software Technol.,* 2019, vol. 106, pp. 101–121.
28. Kitchenham, B. and Charters, S., *Guidelines for performing Systematic Literature Reviews in Software Engineering,* 2007.
29. Noblit, G.W. and Hare, R.D., Meta-ethnography: synthesizing qualitative sudies, *Particularities: Collect. Essays Ethnography Edu.,* 1999, vol. 4, pp. 93–123.
30. Wohlin, C., Guidelines for snowballing in systematic literature studies and a replication in software engineering, *Proc. 18th Int. Conf. on Evaluation and Assessment in Software Engineering,* London, 2014, p. 38.
31. *ISO/IEC 25010: Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models,* 2011.
32. Taylor, R.N., Medvidovic, N., and Dashofy, E.M., *Software Architecture: Foundations, Theory, and Practice,* Wiley, 2009.
33. Khomh, F. and Abtahizadeh, S.A., Understanding the impact of cloud patterns on performance and energy consumption, *J. Syst. Software,* 2018, vol. 141, pp. 151–170.
34. Carneiro, C. and Schmelmer, T., *Microservices from Day One,* Berkeley: Apress, 2016.
35. Hecht, G., Jose-Scheidt, B., De Figueiredo, C., Moha, N., and Khomh, F., An empirical study of the impact of cloud patterns on Quality of Service (QoS), *Proc. 6th IEEE Int. Conf. on Cloud Computing Technology and Science,* Singapore, 2014, pp. 278–283.
36. Aram, K., A microservices implementation journey – part 1, *Medium Koukia blog,* 2018. https://koukia.ca/a-microservices-implementation-journey-part-1-9f6471fe917. Accessed Jan. 12, 2020.
37. Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A., Assuring the evolvability of microservices: insights into industry practices and challenges, *Proc. IEEE Int. Conf. on Software Software Maintenance and Evolution (ICSME),* Cleveland, 2019, pp. 546–556.
38. Sahiti, K., Everything you need to know about microservices design patterns, *Edureka blog,* 2019. https://www.edureka.co/blog/microservices-design-patterns. Accessed Jan. 12, 2020.
39. Microsoft, Responsibility Segregation (CQRS) pattern, *Azure Architecture Center,* 2019. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs. Accessed Jan. 13, 2020.
40. Dwivedi, A.K. and Rath, S.K., Incorporating security features in service-oriented architecture using security patterns, *ACM SIGSOFT Software Eng. Notes,* 2015, vol. 40, no. 1, pp. 1–6.
41. Brown, K. and Woolf, B., Implementation patterns for microservices architectures, *Proc. Conf. on Pattern Language of Programs,* Buenos Aires, 2016, p. 7.
42. Torkura, K.A., Sukmana, M.I.H., Cheng, F., and Meinel, C., Leveraging cloud native design patterns for security-as-a-service applications, *Proc. 2nd IEEE Int. Conf. on Smart Cloud, SmartCloud 2017,* New York, 2017, pp. 90–97.

43. Rodríguez, G., Díaz-Pace, J.A., andSoria, Á., A case-based reasoning approach to reuse quality-driven designs in service-oriented architectures, *Inf. Syst.,* 2018, vol. 77, pp. 167–189.
44. Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., and Lynn, T., Microservices migration patterns, *Software Pract. Exper.,* 2018, vol. 48, no. 11, pp. 2019–2042.
45. Microsoft, Designing a microservice-oriented application, *.NET Microservices: Architecture for Containerized. NET Applications,* 2018. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design. Accessed Jan. 12, 2020.
46. Márquez, G. and Astudillo, H., Identifying availability tactics to support security architectural design of microservice-based systems, *Proc. European Conf. on Software Architecture (ECSA),* Paris, 2019, pp. 123–129.
47. Richardson, C., Building microservices: using an API gateway, *Nginx blog,* 2015. https://www.nginx.com/blog/building-microservices-usingan-api-gateway/. Accessed Jan. 12, 2020.
48. Harms, H., Rogowski, C., and Lo Iacono, L., Guidelines for adopting frontend architectures and patterns in microservices-based systems, *Proc. ACM SIGSOFT Symp. on the Foundations of Software Engineering,* Paderborn, 2017, vol. Part F1301, pp. 902–907.
49. Microsoft, Gateway routing pattern, *Azure Architecture Center*, 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-routing. Accessed Jan. 13, 2020.
50. Müssig, D., Stricker, R., Lässig, J., and Heider, J., Highly scalable microservice-based enterprise architecture for smart ecosystems in hybrid cloud environments, *Proc. 19th Int. Conf. on Enterprise Information Systems,* Porto, 2017, vol. 3, pp. 454–459.
51. Microsoft, Backends for frontends pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends. Accessed Jan. 12, 2020.
52. Microsoft, Strangler pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler. Accessed Jan. 13, 2020.
53. Microsoft, Anti-corruption layer pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer. Accessed Jan. 12, 2020.
54. Microsoft, Ambassador pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador. Accessed Jan. 12, 2020.
55. Microsoft, Sidecar pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar. Accessed Jan. 13, 2020.
56. Microsoft, Gateway aggregation pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation. Accessed Jan. 13, 2020.
57. Microsoft, Gateway offloading pattern, *Azure Architecture Center,* 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading. Accessed Jan. 13, 2020.