MDPI

Article

Feature-Driven Characterization of Microservice Architectures: A Survey of the State of the Practice

Mehmet Söylemez [1], Bedir Tekinerdogan [2,*] and Ayça Kolukısa Tarhan [1]

[1] Department of Computer Engineering, Hacettepe University, Ankara 06800, Turkey; mehmetsoylemez@hacettepe.edu.tr (M.S.); atarhan@hacettepe.edu.tr (A.K.T.)
[2] Information Technology Group, Wageningen University & Research, 6708 PB Wageningen, The Netherlands
* Correspondence: bedir.tekinerdogan@wur.nl

Abstract: With the need for increased modularity and flexible configuration of software modules, microservice architecture (MSA) has gained interest and momentum in the last 7 years. As a result, MSA has been widely addressed in the literature and discussed from various perspectives. In addition, several vendors have provided their specific solutions in the state of the practice, each with its challenges and benefits. Yet, selecting and implementing a particular approach is not trivial and requires a broader overview and guidance for selecting the proper solution for the given situation. Unfortunately, no study has been provided that reflects on and synthesizes the key features and challenges of the current MSA solutions in the state of the practice. To this end, this article presents a feature-driven characterization of micro-service architectures that identifies and synthesizes the key features of current MSA solutions as provided by the key vendors. A domain-driven approach is adopted in which a feature model is presented defining the common and variant features of the MSA solutions. Further, a comparative analysis of the solution approaches is provided based on the proposed feature model.

Keywords: microservice architecture; micro-service; architecture; survey

Citation: Söylemez, M.; Tekinerdogan, B.; Kolukısa Tarhan, A. Feature-Driven Characterization of Microservice Architectures: A Survey of the State of the Practice. *Appl. Sci.* **2022**, *12*, 4424. https://doi.org/10.3390/app12094424

Academic Editors: Paula Fraga-Lamas and Vito Conforti

Received: 22 March 2022
Accepted: 25 April 2022
Published: 27 April 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Traditionally, software systems were for a long time developed as monolithic systems with a monolithic architecture. As a result, software systems were built as a single unit in which different functional aspects, such as data input and output, data processing, business logic, error handling, and user interface, are interconnected and interdependent rather than loosely coupled [1]. In this tightly coupled monolithic system, each component and its associated components must be present for code to be executed or compiled. Provided that the system is not too large, this architectural style can be applied for small teams and small applications and can be easy to develop, debug, and test. However, with the increased size and complexity, monolithic architectures have to cope with serious disadvantages [2]. The start-up time of larger monolithic applications may slow down. Small changes in a single function can require compiling and testing the whole system, and thus maintenance and evolution of the system will become difficult. For each update, a redeployment of the entire application is necessary, and continuous deployment is difficult.

To cope with the obstacles of this architectural style, the notion of service-oriented architecture (SOA) has been proposed [3]. SOA services are provided to the other components by application components through a communication protocol that describes how they pass and parse messages using description metadata. The metadata provide the functional characteristics of the service, as well as the quality-of-service characteristics. Software components are made reusable via service interfaces, which utilize common communication standards and protocols over a network. A service is defined as a discrete unit of functionality that is made accessible by developers, and which can be remotely

accessed and acted upon by other users [3,4]. Examples of services are checking inventory, retrieving a credit card statement online, and payment. A large software application can be provided by using and composing different services.

MSA can be considered as a variant of the SOA, which includes a collection of loosely coupled services. In MSA, services are small in size, autonomously developed, independently deployable, and decentralized, while the protocols are lightweight. An MSA is assumed to have the following properties: (1) it lends itself to a continuous delivery software development process. This implies that a change to a small part of the application only requires rebuilding and redeploying only one or a small number of services. (2) It adheres to principles such as fine-grained interfaces (to independently deployable services), business-driven development (e.g., Domain-Driven Design (DDD)). The goal is that teams can bring their services to life independent of others. Loose coupling reduces all types of dependencies and complexities, as service developers do not need to care about the users of the service and they do not force their changes onto users of the service. MSA gives importance to autonomous and lightweight services [4]. MSA has been in more demand because it minimizes the disadvantages that come with SOA and it has a lightweight architecture. MSA can be deployed, developed, tested, and operated independently.

MSA has been widely addressed in the literature and discussed from various perspectives [5]. In addition, several vendors have provided their specific solutions in the state of the practice, each with its challenges and benefits. Yet, selecting and implementing a particular approach is not trivial and requires a broader overview and guidance for selecting the proper solution for the given situation. Unfortunately, no study has been provided that reflects on and synthesizes the key features and challenges of the current MSA solutions in the state of the practice. The main contribution of this article is a general framework for characterizing MSA platforms. To this end, this article first presents a feature-driven characterization of micro-service architectures that identifies and synthesizes the key features of current MSA solutions as provided by the key vendors. A domain-driven approach is adopted in which a feature model is presented, defining the common and variant features of the MSA solutions. Further, a comparative analysis of the solution approaches is provided based on the proposed feature model. For this, we have identified the three key cloud platforms, including Amazon AWS, Google Cloud Platform, and Microsoft Azure, and characterize these with the presented characterization framework.

The remainder of the paper is organized as follows. In Section 2, we present the background on MSA. Section 3 explains the research methodology. Section 4 introduces the characterization framework that is used to characterize the MSA solutions. Section 5 presents the survey of the selected MSA technologies using the characterization framework. Section 6 focuses on the analysis of existing cloud providers' technologies according to our identified classification framework. Section 7 presents related works and their research directions. Section 8 provides the discussion, including the threats to validity and the lessons learned. Finally, Section 9 concludes the paper.

2. Microservice Architecture

MSA is the latest trend while designing, developing, deploying, and delivering distributed applications [6]. MSA aims to accelerate software development by ensuring continuous delivery and deployment. There are many definitions of microservices. Sam Newman defines microservices as: "small autonomous services that work together, modelled around a business domain" [7], and the most used definition of the MSA is the one defined by Martin Fowler and James Lewis. They define MSA as "an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API".

It will be useful to compare the features and development stages of the microservices with the monolithic structure before moving on. Monolithic applications are developed and deployed as a single unit. Any error on this single unit reveals the necessity of rebuilding the whole application and obtaining the new version. In monolithic applications, all needs

are put in a single unit. This structure can be successful in some cases, but the MSA may be a better alternative as needs are shaped. Because, with this structure, you cannot separate the deployment cycle of each part of the application, maintaining modularity can become increasingly difficult, and it can also be difficult to track changes and their effects. In addition, the scale remains the size of the whole application and you cannot do more resource allocation for a specific part of the application.

Microservice-based development is a result of the evolutionary process. It completely changes the way our applications are designed. In real life, many applications need to be easily reconfigured, modified, and scaled as scenarios evolve. Since every microservice is a small business process and represents a small aspect of business functionality, it is easy to change the workflow [8]. The process starts with using DevOps applications and shaping the organization accordingly. The next step is to have a self-service, on-demand, and elastic infrastructure. Following this step, continuous integration (CI) with automation and the establishment of a continuous distribution (CD) pipeline are important. With such an infrastructure, advanced distribution techniques can be established and the company becomes ready to use microservices [4,9].

The main purpose of MSA is to design, develop, and maintain services that can be delivered faster, improving scalability and autonomous services [4]. Providing autonomous services significantly affects the success of the other two goals, so it can be considered as the primary goal. Service boundaries and communication interfaces with the outside world must be well defined for the fast delivery of services. At this point, domain-driven design can be used to divide the services into subdomains and determine bounded contexts. Autonomous services should be developed, deployed, tested, scaled, versioned, and operated independently. It also allows decentralization of governance and data management services [10]. Another important point here is that the CI and CD infrastructure should be robust and automated so that manual effort is minimized, and speed is gained in order to build, deploy, and operate microservices.

Having improved scalability is the ability to scale our services independently from each other. The importance of determining how the services should be separated from each other and from their boundaries emerges once again. Otherwise, we obtain services with interdependent scaling rules with designs that are made wrong, which is definitely not what is desired [4]. Other advantages that come with MSA are high availability, fault-tolerant infrastructure, and the elimination of long-time commitments to the technology stack. It can be evaluated as a result of the autonomous development of systems. In addition, when we think that each service developed is smaller and more understandable and changeable, it is easier to change the workflows of the services. Another advantage is that the teams become more autonomous and cross-functional. This provides a high level of agility to the teams. Moreover, teams can start developing microservices much faster.

It is not an easy task to develop and maintain the MSA due to its distributed nature. It contains many challenging points and a lot of processes from the distributed architecture that must be managed. It is a matter of expertise to implement a distributed application with this architecture, which includes dealing with basic challenges, such as service discovery, communication, integration, data integrity, fault tolerance, service orchestration, load balancing, data consistency, transaction management, and unavailability. Besides that, there are many complex challenges, such as profit optimization, elastic scheduling, intelligent autoscaling, anomaly detection, etc. [11–14]. However, thanks to both the advanced infrastructure of cloud providers and the community that recommends many common solutions and patterns, practitioners can use MSA easily.

Many companies, such as Amazon, Netflix, LinkedIn, and Spotify, have started to use MSA in their projects [10,15,16]. All of these companies follow the basic model for MSA, as shown Figure 1. This model is structured by some crucial building blocks, such as main business services, infrastructural services, discovery mechanisms, and communication infrastructure. Each block must be isolated from other blocks and communicate with them

using a lightweight protocol. Therefore, it is easy for them to evolve over time according to the needs of the business or technology.

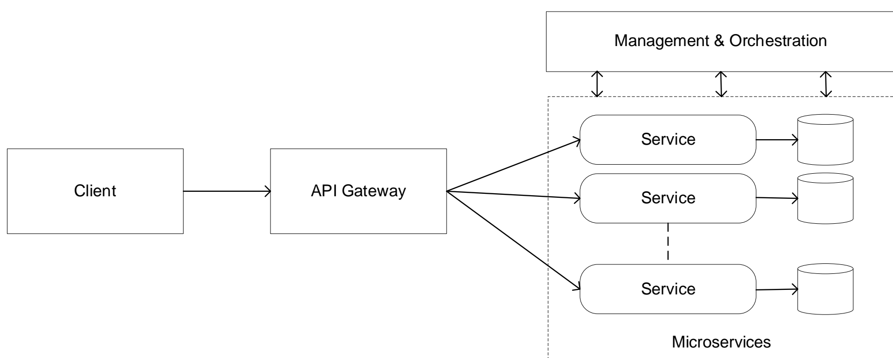

Figure 1. Reference model for MSA. Source: adapted from [17].

3. Research Methodology

This study aims to facilitate the design and development stages of applications to be developed with MSA and to serve as a guide. For this purpose, it is intended to identify the features in the MSA and to classify the technologies according to these features. In this way, the decision-making process will be accelerated, and it will be determined which factors the technology choice depends on. In order to achieve this goal, we have determined the following research questions:

RQ1—What are the current key MSA approaches in the state of the art?
RQ2—What are the key features of these MSA approaches in RQ1?
RQ3—What are the current implementation approaches for the MSA features in RQ2?
RQ4—What are the common and different features of the selected MSA vendor's approaches?

We developed and applied a research methodology shown in Figure 2 to reliably analyze all of the published work involved in this study. This protocol starts by performing domain analysis for MSAs and components; then, a characterization framework is developed according to this domain analysis. Domain analysis is the systematic process for analyzing and modeling the corresponding domain knowledge necessary for the engineering process. Domain analysis includes two key sub-steps of domain scoping and domain modeling. In the domain scoping process, the scope of the investigated domain is defined. In the domain modeling step, the domain knowledge is modeled for further reuse [18]. In this article, we use feature diagrams, which is one of the approaches for domain modeling [19]. Feature diagrams represent the common and variant features of a domain or system.

This process is followed iteratively because, in the meantime, the missing points in the characterization framework can be completed by returning to domain analysis again. Then, to validate our characterization framework, the studies that suggest technology and patterns from both the key providers and MSA area are handled separately and the related technologies are structured according to the characterization framework developed. While selecting and evaluating related MSA technologies and key vendors' infrastructure, the characterization framework can be updated again by going back to the domain analysis phase. Finally, we will eventually present a general evaluation of the work done.

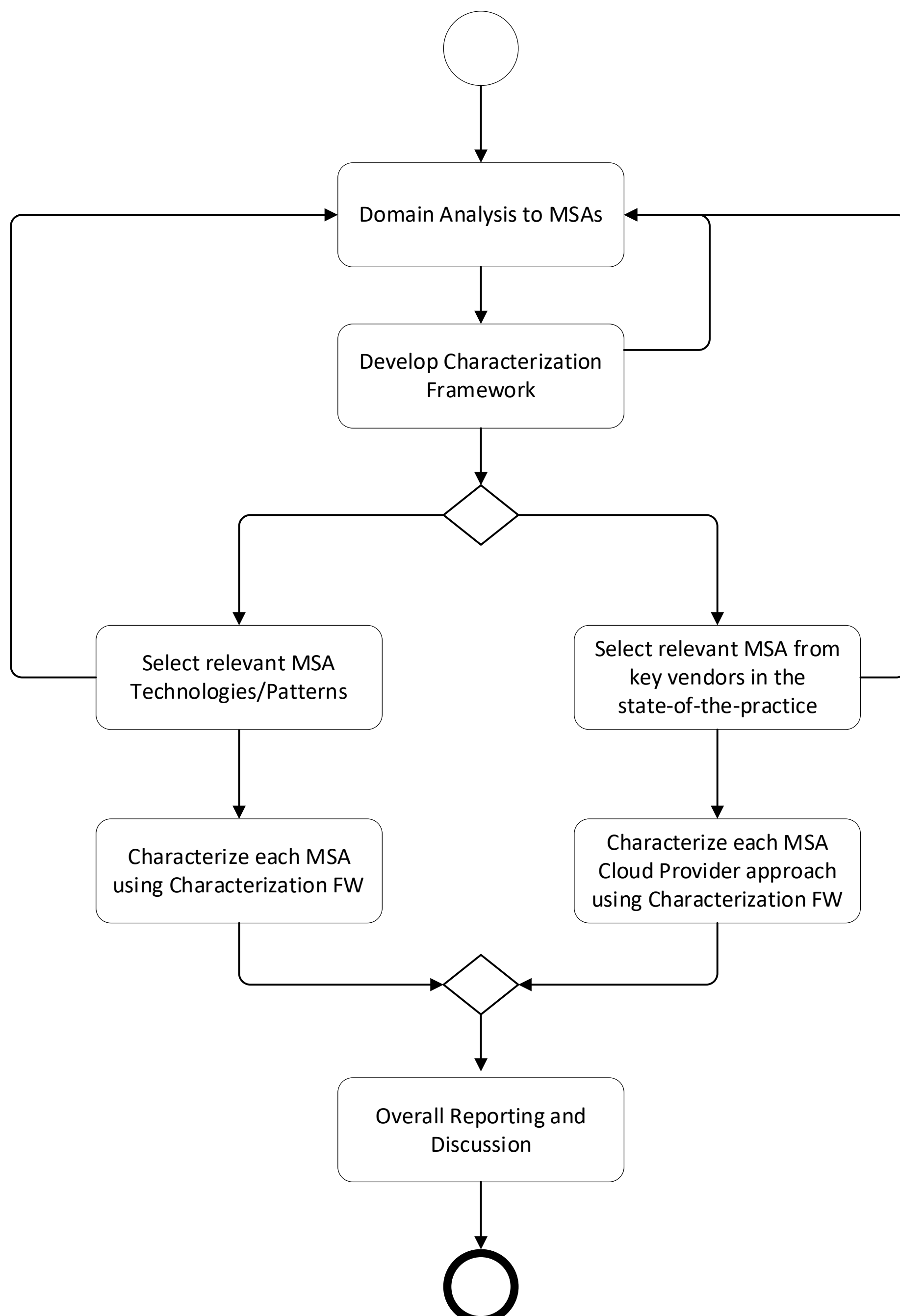

Figure 2. Research methodology.

4. Characterization Framework

We followed a bottom-up approach to classify studies on the MSA. As a result of this process, the characterization framework emerged. Figure 3 introduces the feature diagram of MSA, which represents the common and variant features as provided by the solutions. Table 1 defines the features of MSA. It has many features, with sub-elements being optional, obligatory, or having AND/OR and XOR relationships. Each top-level feature, together with the sub-elements, will be evaluated and discussed in detail in the following sub-section.

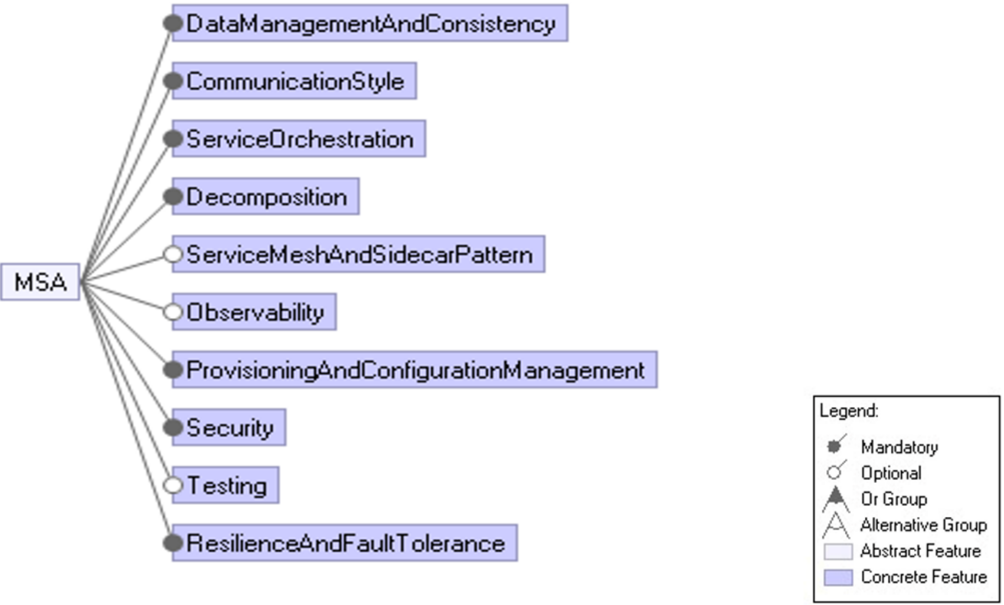

Figure 3. Top-level feature diagram of MSA.

Table 1. Description of the top-level features of the feature diagram for MSA.

1	Data Management and Consistency	Data Management and Consistency highlights ensuring the quality of the distributed data management and consistency between microservices. Moreover, it tries to answer what kind of techniques exist to tackle data management and consistency.
2	Communication Style	Communication Style pays attention to the importance of communication style because it is one of the most complicated parts of microservices. So, it is crucial to find out what kind of communication method exists to provide a stable communication channel between microservices and outside.
3	Service Orchestration	Service Orchestration is the most comprehensive one, addressing lots of critical concerns, such as auto-scaling, service discovery, resource management, load balancing, container availability, and deployment. It focuses on the methods and concepts to handle all of these concerns.
4	Decomposition	Decomposition is the most basic stage of the design of microservices. It directly affects the further detail designs and development activities. It is concerned with which practices we can use while dividing our domain model to microservices.
5	Service Mesh and Sidecar Pattern	Sidecar Pattern is a preceded pattern for the service mesh, and the Service Mesh is usually built on this pattern. Sidecar pattern and service mesh infrastructure is a dedicated infrastructure layer for communication among services and providing resiliency and fault tolerance.

Table 1. *Cont.*

6	Observability	Observability is an important item that ensures the sustainability of the system. In distributed systems, it is critical to obtain information about the general performance of the system and the status of each block of the system and to take appropriate action according to this information or to avoid problems that will force the system.
7	Provisioning and Configuration Management	Provisioning is the process of setting up the system infrastructure. In this process, the necessary resources for the system and users must be managed. These management operations can be achieved with various specialized tools. Configuration Management, on the other hand, is a process that takes charge after provisioning and is used to ensure that our system remains in the desired and consistent state.
8	Security	Security stands on two headings, which are authentication and authorization. In microservice-based systems, since a system consists of many small parts, it must be designed very differently from the one that is designed for monolithic application. Being authenticated and being authorized for many services are the main topics for this feature.
9	Testing	In the MSA, although the fact that a system consists of smaller services increases the testability and maintenance capability of the system to a great extent, it is necessary to develop structures suitable for the distributed architecture in order to test use cases that spread on many services.
10	Resilience and Fault Tolerance	Resilience and Fault Management is the concept for admitting that failures always happen and the system is designed for failures.

4.1. Data Management and Consistency

The relationship between the data layer and services creates different alternative situations in a distributed architecture because the design is shaped according to preferences. When a monolithic application is allocated to microservices, it begins to separate in transactions, which means that local transactions, which were previously in the monolithic, are now being handled as distributed between services. There are different approaches here.

The first and more primitive of these is to manage distributed transactions with a shared database. Each service can access data owned by other services using local Atomicity, Consistency, Isolation, Durability (ACID) transactions. While this situation enables distributed transactions to be handled more easily and to make queries that require joining from different tables more easy, it causes many disadvantages. These are coupling creation in run time, different services needing different requirements from the same database during development, and changes to affect all services [13].

Another method is to have a database for each service. There are many advantages over a shared database in this more common alternative, where microservices are literally decoupled. Each service uses the database that best suits its needs and, since the dependency between services is removed, loosely coupled services are obtained, and this situation makes deployment activities more independent.

As shown in Figure 4, there are some operations that need to be handled if a database per service pattern is selected. First of all, the business transactions spanning multiple services need to be managed and data consistency must be provided. In this case, since it is important to have a highly available system, one needs to choose availability, as specified in the CAP theorem [20], and consider the consistency eventually. This situation corresponds with the Base Availability, Soft State, Eventually Consistency (BASE) database types, which

is proposed by eBay for supporting faster reaction to possible inconsistencies by dismissing synchronization [21]. It is a database design methodology which favors availability over consistency of operations [22].

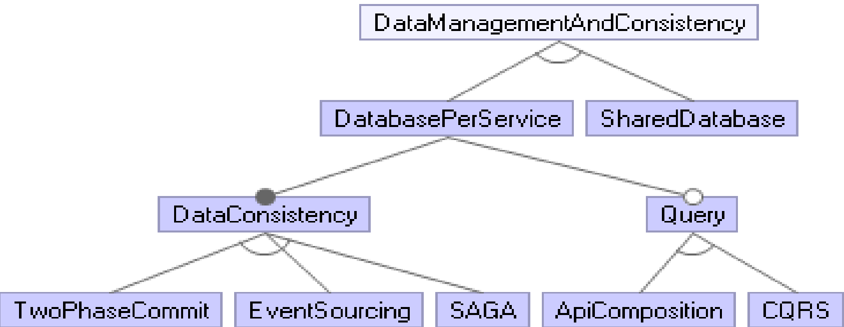

Figure 4. Feature diagram of data management and consistency.

For data consistency, there are three alternatives. Two-phase commit (2PC) pattern is the traditional and only synchronous solution recommended for the management of distributed transactions. This pattern consists of prepare and commit phases and ensures that the data in the entire service are consistent at any given time. According to its setup, in case of failure, writing operations are blocked and availability is compromised. *SAGA* is another alternative for distributed transaction management. It is asynchronous and is used to ensure eventual data consistency without ACID operations when spanning multiple services [23]. When necessary, SAGA carries out compensatory actions at different stages to take it back when any business rule is violated. Each local transaction causes the start of the new local transaction by publishing a new domain event and, at the same time, compensatory functions are executed to undo local transactions as needed [24]. SAGA is difficult to implement due to the reasons of implementing these compensatory actions, the developers implementing these compensatory actions, and the difficulty of managing and debugging these processes. Furthermore, a microservice needs to update its business entity and transmit the message atomically to avoid data integrity and possible bugs. This situation is possible with some improvement of the solution brought by SAGA. With the event sourcing pattern, the atomicity problem is avoided. It stores all states of the business entity in order in the event store. State changes and message delivery are performed atomically on the business entity and a new state event store is added for each state change. In this way, a more reliable transaction infrastructure is provided. Moreover, the status of the business entity at any time can be determined by queries made over the event store [25].

Queries requiring different microservices have become difficult with the existence of distributed transactions. Because most queries are obtained by joining operations over data of more than one service, to overcome this situation, a structure that makes separate queries from each service and combines them can be considered. The API composer pattern recommends this. The results of the original query are calculated by firstly dividing the queries into the required services and then composing the results from each service. However, this situation often causes in-memory problems due to the excess of in-memory joins [23]. Another solution is the command query responsibility segregation (CQRS) pattern. With the CQRS pattern, queries are made over a view database that is registered to domain events and shaped according to the type of queries, thus making handling of complex queries easier [26,27].

4.2. Communication Style

Communication in a microservices architecture is one of the most challenging points due to its distributed nature. It directly affects the availability and resiliency of the systems. In the MSA, we can examine the communication in two headings, intra-microservice communication and inter-microservice communication.

As shown in Figure 5, the first and most complex of the two is communication between services. Services can communicate with each other through a sync communication infrastructure, but, with this architecture, both the client and the server must be available to sustain the communication. Moreover, there is a tight runtime coupling among services. Communication can be sustained without the need for any message brokers, but services need to know each other's locations, which brings extra complexity. Furthermore, an external request generally needs collaboration between services, which might cause blocking of the system for a long time and some problems in availability and resource usage of the system. However, these concerns can be eliminated with async messaging. Availability and resource management improves, and runtime coupling becomes loose. The presence of a message broker can be counted as a challenging point. In addition, communication management is more complex, too. Sometimes, domain-specific protocol can be used in the communication between services; although this type of usage is limited, it can be preferred in an appropriate use case, such as SMTP or IMAP [23].

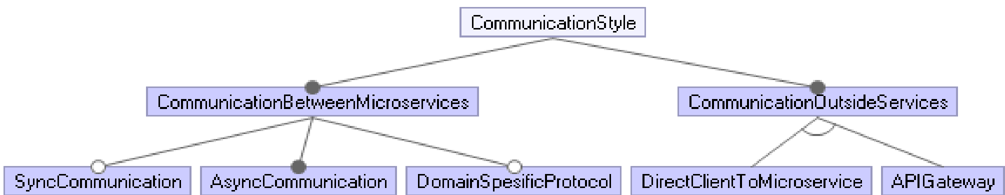

Figure 5. Feature diagram of communication style.

As shown in Figure 5, some patterns are recommended to make the communication of microservices outside of them healthier. For example, with the API Gateway pattern, all requests coming from outside are transferred to the appropriate services inside through this structure, and the services respond to this request by communicating with each other [28]. Different APIs can be created for each type of client. It is called Backend for Frontend (BFF) by SoundCloud [29]. Moreover, it can translate external requests into protocols used across microservices. Since the location information of the services changes dynamically, the outside world does not need to know this location information thanks to API Gateway. This structure can be thought of as the only door opening to the outside world and isolates the system inside. Security concerns can be addressed here. For example, in a scenario where HTTPS is used when talking to the outside world, it will be sufficient for the services inside to talk with the HTTP protocol because the inside can be considered safe after the API Gateway. Some cross-cutting concerns, such as SSL, could be handled in API Gateway so internal microservices are lightweight and simplified [30]. Another solution is that each client communicates directly with microservices, but this method is a primitive method and its usage area is very limited. None of the benefits that come with API Gateway can be achieved with this pattern.

4.3. Service Orchestration

This concept, which can be referred to as service orchestration or container orchestration, automates the management, scaling, deployment, and networking of microservices. The application provides great assistance in deploying to different environments, without the need for a new design, to orchestrate the services. In this context, service orchestration is a concept that addresses many different concerns.

As shown in Figure 6, auto-scaling is one of them, and, by monitoring our application, it automatically adjusts the capacity according to the incoming load and keeps the system highly available and steady [31,32]. Within the auto-scaling configurations, the system can scale horizontally or vertically. It also provides a manual scaling feature to be used in some cases.

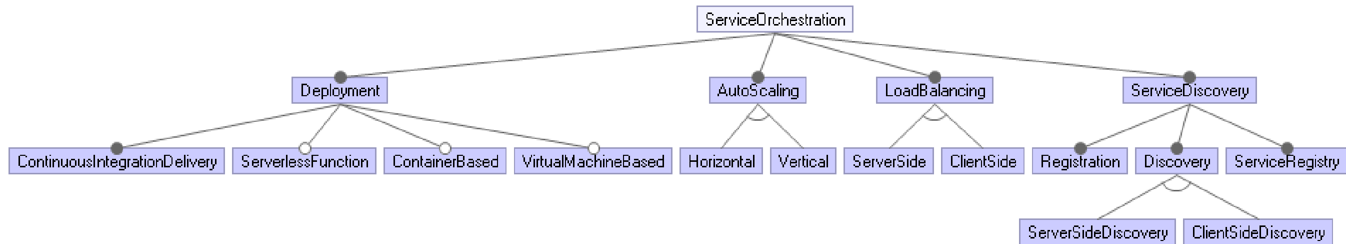

Figure 6. Feature diagram of service orchestration.

Another concern is load balancing. It is used to distribute the traffic coming to the system efficiently. It also provides high availability and reliability by sending incoming requests only to the servers that are standing [33]. It works in harmony with the new server, adding and removing operations when necessary. In this way, the system will be more scalable and flexible. Different variations depend on where the load balancing setup is carried out. For example, in server-side load balancing, the client does not interfere with the load balancing process and its request is distributed efficiently to the appropriate servers on the server side; but, in client-side load balancing, the client takes over the load balancing job. After querying which servers are suitable or not from a structure, such as a service registry, it distributes the load effectively.

Service discovery, on the other hand, is an indispensable structure in the distributed architecture. Thanks to this structure, the changing services, whose location information is dynamic, become able to discover after they complete service-registration. Here, similar to the client- and server-side load balancer distinction, there is either a registry-aware client mechanism or a structure that requires the request from the client to be directed to our services via a registry-aware router.

Independent deployment is one of the most important skills aimed at and acquired by MSA. In this way, the CI and CD pipelines of our services are separated. An advanced deployment setup is created with automated infrastructure. In this way, fast delivery is ensured. While deploying, one can prepare an application for deployment with the help of containers. Thus, the containers are isolated from each other and encapsulated in the technology stack used while the services are built. Moreover, the services can be easily scaled up and down. Another method is to deploy the services using virtual machine (VM). Compared to containers, resource usage is high in VM. The container-based method has become a de facto for deploying the services at the moment and it is a lot more portable. Another deployment method is serverless deployment. It emerged as a result of the spread of microservices and cloud environments. With this deployment method, the user simply writes the code and uploads a provider that provides a serverless infrastructure. After that, it is completely up to the provider. Many headings, such as scalability, deployment, and operating system, are completely managed by the provider. Moreover, serverless is the deployment and development method, which is developed to implement the Function as a Service (FaaS) category of cloud computing services.

4.4. Decomposition

One can develop systems, which are large in terms of business rule and domain, with MSA. Hence, the aim is to develop the system in smaller applications and achieve continuous delivery and deployment. In addition, each microservice is developed faster and more easily. However, determining the boundaries of these small applications is not an easy task and needs to be carried out carefully. Moreover, the aim is to create loosely coupled, highly cohesive, and autonomous services. In addition, tools can be more cross-functional in this way.

The method used to design an application as smaller services are either decompose by business capability or decompose by subdomain, as shown in Figure 7. In decomposition by business capability, services are concentrated around business capability; while

using DDD [34] principles in the decomposition by subdomain, they are concentrated on subdomains and use cases related to these subdomains.

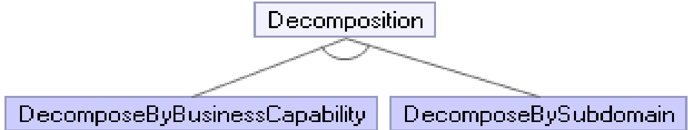

Figure 7. Feature diagram of decomposition.

4.5. Service Mesh and Sidecar Pattern

Before the service mesh ecosystem was introduced, sidecar proxies had emerged and started to be used. Sidecar proxies encapsulate service discovery, communication protocols, load balancing, and fault tolerance mechanism to abstract them from the developer [8,27]. With the service mesh structure built on the sidecar proxy pattern, a fully integrated service-to-service communication infrastructure is provided and the security, reliability, and observability features are managed by the platform layer [9,28].

4.6. Observability

The large and complex nature of modern systems, dynamic infrastructure, and monitoring the health of these systems and taking the necessary actions as a result of this monitoring reveal the importance of observability.

As shown in Figure 8, monitoring collects information about a system by communicating with services. In order for this need to continue uninterruptedly, the system must have a scalable infrastructure and it must be easy to query the collected information. Monitoring focuses on runtime metrics created by the applications themselves and related measurements, such as CPU, memory, I/O, etc., which are the infrastructural metrics of the system. Distributed tracing in a system, on the other hand, is where requests are spread over multiple services and each service responds to this request by communicating with different layers. It follows the behavior of the application while responding to this request and whether it is experiencing any problems by assigning an external request ID to each request and recording it. In log aggregation, it is ensured that the logs coming from all these services are collected in a central service and can be queried and analyzed from there. In addition, by creating alerts for specific logs to be examined, developers are notified when such logs occur. Exception tracking, on the other hand, concentrates on exceptions and records the exceptions that occur in the system. With the help of the recorded data, various inquiries and informing the developers using alerts are provided when necessary. In this way, with a central exception tracking infrastructure, developers are prevented from working continuously with the same error because historic data are provided for the relevant error type and the user knows that the error has been solved before. Audit logging, on the other hand, records the information that the system users performed on the system and the stages they went through.

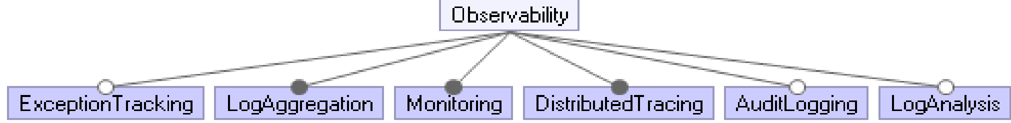

Figure 8. Feature diagram of observability.

4.7. Provisioning and Configuration Management

Provisioning and configuration management has become a hot topic with the increasing interest in distributed systems and MSA. As shown in Figure 9, they should be established in each mature MSA.

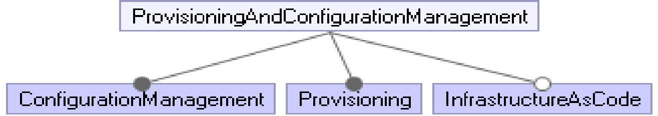

Figure 9. Feature diagram of provisioning and configuration management.

Within the scope of provisioning, operations such as introducing the information technology (IT) infrastructure and then managing the resources needed by this infrastructure are handled. In addition, providing this infrastructure to the service of the system and users is also one of the provisioning activities. There are four subtypes: service, user, server, and network, coming after provisioning. It is a process to maintain systems and software, which ensures that systems remain in the desired state. With any configuration management tool, we can separate and manage the system into related groups or modify the basic configuration center, prioritize some actions, and automate processes, such as updating the system and expanding new settings [35]. Infrastructure as code can be considered as the next step. With this feature, one can program infrastructure by writing code and configure it the way it is wanted. In other words, one writes code to automate the infrastructure and run it. The idea behind this approach is that the systems and devices used to run the software themselves can be treated like software [36].

4.8. Security

In an MSA, security is actually gathered under two main headings as in all other systems, as shown in Figure 10. These are authorization and authentication. As shown in Figure 10, both are concepts to be addressed. However, handling these processes in MSA can create a more complex structure compared to the monolithic architecture. There are some best practices and patterns for this.

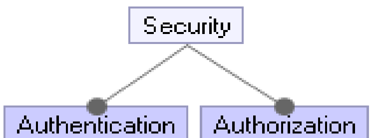

Figure 10. Feature diagram of security.

For authentication, a token is usually given to the user by performing an identity check through a structure that is developed into a communication task between the external world and the internal world, such as the API Gateway. This token contains the information that the user has authenticated to the system and what his/her permissions are. Thanks to this structure, the user can authenticate from a single point to a structure with many services. This eliminates the disadvantage that many services have relations with the outside world. Moreover, in this way, the services inside will have the convenience of talking to each other with HTTP instead of HTTPS as an example. In this case, however, it should not be forgotten that the API Gateway is a centralized failure point and the design for failure should be carried out accordingly. Another option, for each microservice to run, is having its own authentication and authorization processes locally, as opposed to global management. This situation requires the requests to be authenticated separately for each microservice and the complexity increases. However, each service can use different authentication and authorization methods according to preference and, at this point, a more fine-grained mechanism should be designed.

4.9. Testing

With the spread of software development with MSA, the need for revising some approaches used in monolithic applications and adding new approaches has arisen, because, now, this is an environment where each microservice can be deployed individually and perhaps developed by different teams.

As shown in Figure 11, there are various types of test able to be used in MSA. We can test whether the system shows the expected behavior from end to end, with the end-to-end test, as in monolith applications. However, this approach is very difficult to manage because the test boundaries are too large and tests are very fragile. We can test smaller parts with integration tests. For example, it can be detected with this approach whether there is an error in the communication interfaces between different layers, such as the data layer and service layer. With the consumer-driven contract test, we concentrate on communication between services and, in the communication of the two services, it is tested whether the waiting of the service that will consult a message can be met by the service that will produce the message. In the service component test, which is another type of test, the components to be tested are isolated from the remaining parts of the system by using test doubles and are tested by manipulating through internal interfaces. This enables each tested item to be tested in more detail. Finally, in chaos engineering, to ensure the stability of the system under all kinds of conditions, the system's responses are examined by leaving the system to deal with various failure conditions in the production environment, and thus the reliability of the system is tested. To sum up, it is of great importance to use the mentioned test approaches together and in harmony for the systems to reach high test coverage.

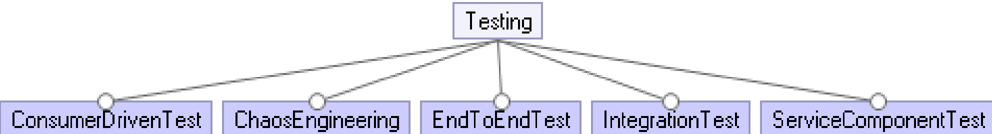

Figure 11. Feature diagram of testing.

4.10. Resilience and Fault Tolerance

In monolithic applications, any failure had the potential to completely down the application. However, in case of failure that can be experienced with the MSA, it provides an opportunity to compensate for this situation without affecting the overall application. It is necessary to admit that there will always be failures in the system, and designs that address failure situations should be made to quickly avoid such failures or to reduce the number of failures. For this, failure scenarios should be determined as much as possible, locations that may cause a single point of failure should be identified, and our designs should be arranged to avoid cascading failure in case of a failure. As shown in Figure 12, there are many ways to ensure resiliency. It is highly recommended to use as many patterns as possible.

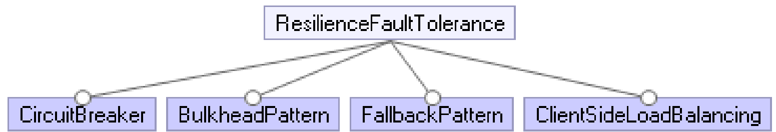

Figure 12. Feature diagram of resilience and fault tolerance.

Client-side load balancing is often used in some scenarios as it eliminates a single point of failure and distributes the responsibility for load balancing and is easier to scale than server-side load balancing. Service instances query and cache information of health services from service discovery. In calls to be made by the service to other services, service information is received from the service discovery and communication is provided in a way that the load will be distributed equally. If one of the services responds late or gives an error, the load balancing mechanism detects this problem and removes it from the service repository and prevents cascading failures and system downtime. If service discovery does not respond, client services can use the information in their own cache copy. In circuit breaker pattern, if a client faces several problems over the call to another service, it stops communicating with the relevant service, and thus prevents cascading failure in the system. Fallback pattern is another approach for handling failure cases. In this pattern, as a result

of the detection of a problematic request, it prevents the occurrence of a large problem that will affect the system in general by giving an alternative response to the client. In bulkhead pattern, by separating and isolating both suitable components and data from each other, it is ensured that the problems encountered in any group will not affect those in the other group.

5. Survey of MSA

With the development of microservices and distributed architecture, the need for practitioners to develop common solutions to problems arose. Due to these needs, many products have been or are still being developed by various companies or communities. Knowing what purpose these developed solutions serve and where they are located in the MSA enable us to design the architecture more comfortably and to make our architecture more robust. Therefore, we think that showing the feature set we have determined in the field of microservices to match the relevant technologies will benefit practitioners greatly.

As is seen in Table 2, common solutions have been developed by various companies or open-source communities for many features. We observe that no technology has been developed for some feature sets and they are more design-oriented feature sets. In other words, for these feature sets, it is recommended to apply the design decisions specified in the feature instead of a solution. For example, database per service or shared database, which are two different patterns in data management and consistency, is entirely a design decision about how you will position the data layer.

Table 2. Mapping features with MSA technologies.

Feature	Technology/Product/Service
Testing/Chaos Engineering	Chaos Monkey Chaos Toolkit Simian Army
Testing/Service Component Test	Spring Cloud Contract Test
Resilience and Fault Tolerance/Circuit Breaker	Netflix Hystrix Resilience4j
Communication Style/API Gateway	Nginx Netflix/Zuul Spring Cloud Gateway
Communication Style/Domain Spesific Protokol	SMTP IMAP
Communication Style/Async Communication	Apache Kafka Rabbit MQ Active MQ
Observability/Log Analysis	Kibana Datadog LogDNA
Observability/Distributed Tracing	Zipkin Datadog OpenCensus Sentry LogDNA
Observability/Monitoring	Prometheus Graphite Grafana InfluxDB Zabbix

Table 2. *Cont.*

Feature	Technology/Product/Service
Observability/Log Aggregation	Kibana Datadog LogDNA
Observaility/Exception Tracking	Sentry
Provisioning and Configuration Management	Ansible Chef Puppet SaltStack
Provisioning and Configuration Management/Infrastructure as Code	Terraform
Security/Authentication	CAS Spring Security SSO
Security/Authorization	JWT Spring Security
Decomposition/Decompose by Subdomain	Domain Driven Design
Service Orchestration	Kubernetes Apache Mesos + Marathon Docker Swarm
Service Mesh and Sidecar Pattern	Istio Linkerd Envoy Redhat Openshift
Deployment/CI & CD	Jenkins CircleCI Travis DroneCI Gitlab CI Bamboo
Deployment/Container	Docker LXC
Deployment/Virtual Machine	VMWare VirtualBox
Load Balancing/Server-side	Nginx Zuul Eureka
Load Balancing/Client-side	Ribbon Client
Service Discovery/Service Registry	Eureka Zuul Consul Apache Zookeeper
Service Discovery/Server-side	Eureka Zuul Consul Apache Zookeeper
Service Discovery/Client-side	Ribbon Client

We also observe that some features are taken together, and solutions are proposed accordingly. Such solutions suggest a more comprehensive solution for one or more features. For example, provisioning and configuration management are two separate activities that can be considered as a continuation of each other. One cannot be thought of

without the other. Therefore, it makes sense to propose a solution that handles these two features together while recommending a solution. Instead of learning and using multiple technologies and integrating them, it is a more preferred way for developers to use the ready-made solution. In such cases, if these solutions are suitable for all sub-features of the relevant parent feature, these solutions are shown at the parent level. If it does not fit all children, these technologies are shown in the feature diagram for each feature. It is also observed that some solutions may be not only for siblings, but also for features in different feature families. For example, the solutions suggested in load balancing and service discovery are taken together for both features, and solutions addressing these two features are produced.

6. Analysis of Existing Key Cloud Providers

In parallel with the development of the distributed architecture and MSA, cloud computing is also improving. Cloud providers develop managed services for the difficulties brought by the distributed architecture and make them ready to be used in related architectures. Users configure and use the relevant services and are not interested in many quality indicators because the services give warranties on many basic quality indicators and developers focus more on domains and business rules. However, it should be decided by considering the cost of the usage of cloud services.

In this section, the services provided by Amazon AWS, Google Cloud Platform, and Microsoft Azure, which are the three most preferred cloud providers today, have been examined and which solutions are available for which functions are presented.

As is seen in Table 3, services are offered by cloud providers for many features. These services are provided as managed by the cloud provider, so you can start using the services by making the necessary configurations. The services have the ability to work in harmony with each other and can be easily configured as interoperable. In this respect, using these products will give us speed. Another important point is the fact that services are developed parallel to each other by all three providers. You can find the equivalent of each service in another provider. Here, which one will be selected can be concluded by making some more detailed evaluations within the scope of performance, usability, use cases, and cost analysis. The absence of a direct solution for some features indicates that it must be a feature that needs to be programed and designed, that is, it cannot be fully managed by the cloud provider. They are more conceptual and design-oriented features. However, by bringing together more than one service, the design criteria recommended within the scope of these features can be met.

Table 3. Feature-based service comparison among AWS, Google Cloud, and Microsoft Azure.

Feature		AWS	Google Cloud	Microsoft Azure
Communication Style	Async Communication	AWS MQ AWS SNS AWS SQS AWS Kinesis	Google Dataflow Google Pub/Sub	Azure Queue Storage Azure Service Bus Azure Event Grid Azure Event Hubs
	API Gateway	AWS API Gateway	Google Apigee	Azure API Management
Service Orchestration		AWS ECS AWS EB AWS EKS	Google Cloud Run Google App Engine Google Kubernete Engine	Azure Container Instances Azure App Service Azure EKS
Service Orchestration	Deployment/CI and CD	AWS CodeDeploy AWS CodeBuild AWS CodePipeline	Google Cloud Build	Azure Devops
	Deployment/Serverless Function	AWS Lambda AWS Step Function	Google Cloud Function Google Cloud Composes	Azure Durable Azure Functions
	Deployment/Container	AWS Fargate AWS EKS	Google Cloud Run Google Kubernete Engine	Azure Container Instance Azure Kubernete Service

Table 3. *Cont.*

Feature		AWS	Google Cloud	Microsoft Azure
	Deployment/VM	AWS EC2	Google Compute Engine	Azure VM
	Auto-scaling	AWS EC2 Auto-scaling	Google Computer Engine Auto-scaling	Azure Virtual Machine Scale Set
	Load Balancing/Server-side	AWS ELB	Google Cloud Engine Load Balancing	Azure Load Balancing
	Service Discovery/Server-side	AWS Route 53 AWS Cloud Map AWS ELB	Google Cloud DNS Google Cloud Engine Load Balancing	Azure DNS Azure Load Balancing
Service Mesh and Sidecar Pattern		AWS AppMesh	Google Anthos Service Mesh	Azure Service Fabric Mesh
Observability	Log Analysis	AWS Elasticsearch Service AWS Redshift AWS Quicksight AWS Athena	Google Elasticsearch Service Google BigQuery	Azure Elasticsearch Service Azure PowerBI Azure Data Lake Analytics
	Exception Tracking	AWS CloudWatch	Google Cloud Debugger Google Cloud Trace	Azure Application Insights Azure Monitor
	Log Aggregation	AWS CloudWatch	Google Cloud Logging	Azure Application Insights Azure Monitor
	Audit Logging	AWS CloudTrail AWS Config	Google Audit Logs Google Cloud Asset Inventory	Azure Monitor
	Distributed Tracing	AWS X-Ray	Google Cloud Debugger Google CloudTrace	Azure Monitor
	Monitoring	AWS CloudWatch	Google Cloud Monitoring	Azure Monitor
Provisioning and Configuration Management		AWS CloudFormation	Google Cloud Deployment Manager	Azure Resource Manager Azure VM extensions Azure Automation
Security		AWS Cognito	Google Firebase Authentication	Azure Active Directory B2C

7. Related Work

There is a limited amount of work focusing on the comprehensive approach to help practitioners to identify and choose features they need during the designing phase. A few studies on MSA [2,37–41] present an overview of academic and grey literature. Pahl et al. [37] discover practical motivations behind using MSA and different types of MSA. In addition, existing research issues and potential future research items have been identified in this study. Soldani et al. [2] explore the potential challenges and obstacles. Furthermore, the advantages of MSA are explained by reviewing more than 50 studies. Alshuqayran et al. [38] identify the quality attributes and architectural diagrams in MSA, as well as architectural challenges from 33 studies. Vural et al. [40] focus on the practical motivations in MSA-related studies and provide overview of emerging standards and tools in MSA development. Francesco et al. [39] present publication trends and industrial adoptions of MSA with the help of more than 100 studies. Bushong et al. [41] study more than 50 studies and extract some useful information, such as current issues, opportunities, and

potential future research items. Moreover, methods and techniques in MSA are examined. Finally, Shanshan et al. [42] aim to identify quality attributes of MSA by focusing from a QA point of view.

Some studies do not give a complete overview of MSA. They just focus on one aspect or feature of MSA. By the way of illustration, Berardi et al. [43] and Pereira et al. [44] focus on the microservice security, Karabey et al. [45] address the deployment and communication patterns in MSA, whereas Fredy et al. conduct a systematic literature review to reveal the approaches to be used to define MSA granularity and identify the metric to be used to evaluate MSA granularity.

According to our observations, there is only one study to classify some technologies and patterns. Jamshidi et al. [4] present MSA evolution from scratch. Progression within MSA evolution is also detailed with the technology used in this period. In addition, tools and technologies in MSA are also provided regarding the predetermined categories.

Our approach differs from studies in this section by addressing many aspects and also making characterization with a well-defined comprehensive characterization framework.

8. Discussion

In this study, we have proposed a characterization framework by performing domain-driven analysis and a feature modeling approach. The approach provides a comprehensive analysis of MSA features. It is stated that there are many critical features in the MSA and for what purposes these features are used and what kind of contributions they make in terms of the integrity of the system. Knowing the sub-features of these features and their relationships is critical to designing a strong and highly maintainable architecture. The teams that design the architecture and develop it later need information on what solutions have been developed, both in the academic field and on the industry side, and what purposes they serve. From this point of view, a feature diagram was proposed for MSA.

With the help of the proposed framework, we have also identified and described the existing platforms without the focus on qualitative comparisons. This study may work as a well-defined guide for both software architects and developers. However, it is observed that there are not enough solutions in some features, such as testing, or some features, such as data management and consistency, remain more in the form of design level or pattern. Features such as these remain more conceptual, so it is recommended that related components should be designed according to the proposed design or pattern-based features.

It has been shown that the technologies used within the scope of MSA are either based on the cloud provider or developed as separate technology by a community or a company. As can be seen in Table 2, the services developed by the cloud providers based on features are very diverse and rich. On the other hand, de facto technologies, such as Docker and Kubernetes, in this area are among the technologies developed by the companies or communities. A company that uses MSA can also use the solutions in both in a hybrid way.

As a result of researching our solutions on a feature basis, it has been evaluated that there are solutions for lots of features and the teams can easily adapt them. However, the lack of sufficient solutions for data management and consistency and testing shows that these features are areas open for further solution proposals. When we develop our software system as multiple microservices, transaction integrity becomes critical. In a process that concerns different microservices, data management should be ensured while maintaining transaction integrity. Similarly, it is necessary to test whether these microservices have a problem in their communication with each other before going to the real environment. Testing the communication of the microservices is seen as a challenging step at this point.

In future work, we plan to enhance our characterization framework by adding a quantitative comparison module for practitioners to select suitable platforms comfortably and easily. We consider this study to be a good basis for such a study to be evolved.

9. Conclusions

MSA has gained interest and momentum in the last years and has ben discussed in both the literature and applied in practice. For practitioners, it is not easy to select the proper vendor solution and apply the MSA properly. To this end, this article has presented first a characterization framework that captures the common and variant features of MSAs. The framework has been developed after a thorough domain analysis process following our research questions, which resulted in a family feature model for the MSA domain. The second part of the study focused on the illustration and validation of the framework. For this, we have selected three key vendor solutions and were able to characterize these with the characterization framework. The framework does not only cover the three MSAs, but can be also used for other vendor solutions. The framework can be used for both researchers and practitioners. Researchers can obtain a broad insight into the common and variant features and this way will pave the way for further research. Practitioners can use the result of the study to guide their activities in selecting and applying MSAs. In our future work, we will apply the framework for other vendor solutions and aim to provide further support for architecture design solutions of applying the MSA.

Author Contributions: Conceptualization, M.S., B.T. and A.K.T.; methodology, M.S., B.T. and A.K.T.; software, M.S.; validation, M.S., B.T. and A.K.T.; writing-review and editing, M.S., B.T. and A.K.T.; supervision, B.T. and A.K.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Valdivia, J.A.; Lora-González, A.; Limón, X.; Cortes-Verdin, K.; Ocharán-Hernández, J.O. Patterns Related to Microservice Architecture: A Multivocal Literature Review. *Program. Comput. Softw.* **2020**, *46*, 594–608. [CrossRef]
2. Soldani, J.; Tamburri, D.A.; van den Heuvel, W.J. The Pains and Gains of Microservices: A Systematic Grey Literature Review. *J. Syst. Softw.* **2018**, *146*, 215–232. [CrossRef]
3. Josuttis, N. *Soa in Practice: The Art of Distributed System Design*; O'Reilly Media, Inc.: Newton, MA, USA, 2007; ISBN 0596529554.
4. Jamshidi, P.; Pahl, C.; Mendonca, N.C.; Lewis, J.; Tilkov, S. Microservices: The Journey so Far and Challenges Ahead. *IEEE Softw.* **2018**, *35*, 24–35. [CrossRef]
5. Thönes, J. Microservices. *IEEE Softw.* **2015**, *32*, 116. [CrossRef]
6. Zimmermann, O. Microservices Tenets. *Comput. Sci.* **2017**, *32*, 301–310. [CrossRef]
7. Newman, S. *Building Microservices*, 1st ed.; O'Reilly Media, Inc.: Newton, MA, USA, 2015; ISBN 1491950358.
8. Shadija, D.; Rezai, M.; Hill, R. Towards an Understanding of Microservices. In Proceedings of the ICAC 2017—2017 23rd IEEE International Conference on Automation and Computing: Addressing Global Challenges through Automation and Computing, Huddersfield, UK, 7–8 September 2017. [CrossRef]
9. Benevides, R. Istio on Kubernetes. Available online: http://bit.ly/istio-kubernetes%0A (accessed on 13 March 2022).
10. Fowler, M.; Lewis, J. Microservices. Available online: https://martinfowler.com/articles/microservices.html (accessed on 13 March 2022).
11. Liu, C.; Li, K.; Li, K.; Buyya, R. A New Service Mechanism for Profit Optimizations of a Cloud Provider and Its Users. *IEEE Trans. Cloud Comput.* **2021**, *9*, 14–26. [CrossRef]
12. Wang, S.; Ding, Z.; Jiang, C. Elastic Scheduling for Microservice Applications in Clouds. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 98–115. [CrossRef]
13. Khaleq, A.A.; Ra, I. Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications. *IEEE Access* **2021**, *9*, 35464–35476. [CrossRef]
14. Jin, M.; Lv, A.; Zhu, Y.; Wen, Z.; Zhong, Y.; Zhao, Z.; Wu, J.; Li, H.; He, H.; Chen, F. An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis. *IEEE Access* **2020**, *8*, 226397–226408. [CrossRef]
15. Villamizar, M.; Garcés, O.; Castro, H.; Verano, M.; Salamanca, L.; Casallas, R.; Gil, S. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. In Proceedings of the 2015 10th Computing Colombian Conference (10CCC), Bogota, Colombia, 21–25 September 2015; pp. 583–590.

16. Yahia, E.B.H.; Réveillère, L.; Bromberg, Y.-D.; Chevalier, R.; Cadot, A. Medley: An Event-Driven Lightweight Platform for Service Composition. *Lect. Notes Comput. Sci.* **2016**, *9671*, 3–20. [CrossRef]
17. O'Connor, R.V.; Elger, P.; Clarke, P.M. Continuous Software Engineering-A Microservices Architecture Perspective. *J. Softw. Evol. Process* **2017**, *29*, e1866. [CrossRef]
18. Tekinerdogan, B.; Aksit, M. Classifying and Evaluating Architecture Design Methods. *Softw. Archit. Compon. Technol.* **2002**, 3–27. [CrossRef]
19. Tekinerdogan, B.; Öztürk, K. Feature-Driven Design of SaaS Architectures. In *Software Engineering Frameworks for the Cloud Computing Paradigm*; Springer: London, UK, 2013; pp. 189–212. [CrossRef]
20. Brewer, E. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* **2012**, *45*, 23–29. [CrossRef]
21. DeLoatch, C.; Blindt, S. *NoSQL Databases: Scalable Cloud and Enterprise Solutions*; University of Illinois at Urbana Champaign: Champaign, IL, USA, 2012.
22. Ganesh Chandra, D. BASE Analysis of NoSQL Database. *Future Gener. Comput. Syst.* **2015**, *52*, 13–21. [CrossRef]
23. Richardson, C. *Microservices Patterns: With Examples in JAVA*; Simon and Schuster: New York, NY, USA, 2019.
24. Limon, X.; Guerra-Hernandez, A.; Sanchez-Garcia, A.J.; Perez Arriaga, J.C. SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture. In Proceedings of the 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), San Luis Potosi, Mexico, 24–26 October 2018; pp. 50–58.
25. Event Sourcing. Available online: https://martinfowler.com/eaaDev/EventSourcing.html (accessed on 13 March 2022).
26. Command Query Responsibility Segregation (CQRS). Available online: https://microservices.io/patterns/data/cqrs.html (accessed on 13 March 2022).
27. CQRS. Available online: https://martinfowler.com/bliki/CQRS.html (accessed on 13 March 2022).
28. What Is an API Gateway? NGINX Learning. Available online: https://www.nginx.com/learn/api-gateway/ (accessed on 13 March 2022).
29. BFF@SoundCloud | ThoughtWorks. Available online: https://www.thoughtworks.com/insights/blog/bff-soundcloud (accessed on 13 March 2022).
30. The API Gateway Pattern versus the Direct Client-to-Microservice Communication | Microsoft Docs. Available online: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern (accessed on 13 March 2022).
31. Microservices on AWS. Available online: https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices.html (accessed on 13 March 2022).
32. AWS Auto Scaling. Available online: https://aws.amazon.com/autoscaling/ (accessed on 13 March 2022).
33. Liu, C.; Li, K.; Li, K. A Game Approach to Multi-Servers Load Balancing with Load-Dependent Server Availability Consideration. *IEEE Trans. Cloud Comput.* **2021**, *9*, 1–13. [CrossRef]
34. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley: Boston, MA, USA, 2004.
35. What is Configuration Management? Available online: https://www.redhat.com/en/topics/automation/what-is-configuration-management (accessed on 13 March 2022).
36. Infrastructure as Code: A Reason to Smile | ThoughtWorks. Available online: https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile (accessed on 13 March 2022).
37. Pahl, C.; Jamshidi, P. Microservices: A Systematic Mapping Study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science, Rome Italy, 23–25 April 2016; pp. 137–146. [CrossRef]
38. Alshuqayran, N.; Ali, N.; Evans, R. A Systematic Mapping Study in Microservice Architecture. In Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), Macau, China, 4–6 November 2016; pp. 44–51.
39. Francesco, P.D.; Lago, P.; Malavolta, I. Architecting with Microservices: A Systematic Mapping Study. *J. Syst. Softw.* **2019**, *150*, 77–97. [CrossRef]
40. Vural, H.; Koyuncu, M.; Guney, S. *A Systematic Literature Review on Microservices*; Springer: Cham, Switzerland, 2017.
41. Bushong, V.; Abdelfattah, A.S.; Maruf, A.A.; Das, D.; Lehman, A.; Jaroszewski, E.; Coffey, M.; Cerny, T.; Frajtak, K.; Tisnovsky, P.; et al. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. *Appl. Sci.* **2021**, *11*, 7856. [CrossRef]
42. Li, S.; Zhang, H.; Jia, Z.; Zhong, C.; Zhang, C.; Shan, Z.; Shen, J.; Babar, M.A. Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review. *Inf. Softw. Technol.* **2021**, *131*, 106449. [CrossRef]
43. Berardi, D.; Giallorenzo, S.; Melis, A.; Prandini, M.; Mauro, J.; Montesi, F. Microservice Security: A Systematic Literature Review. *PeerJ Comput. Sci.* **2022**, *7*, e779. [CrossRef]
44. Pereira-Vale, A.; Fernandez, E.B.; Monge, R.; Astudillo, H.; Márquez, G. Security in Microservice-Based Systems: A Multivocal Literature Review. *Comput. Secur.* **2021**, *103*, 102200. [CrossRef]
45. Karabey Aksakalli, I.; Çelik, T.; Can, A.B.; Tekinerdogan, B. Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review. *J. Syst. Softw.* **2021**, *180*, 111014. [CrossRef]