Received 14 July 2023, accepted 3 August 2023, date of publication 15 August 2023, date of current version 23 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3305687

RESEARCH ARTICLE

A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges

VICTOR VELEPUCHA AND PAMELA FLORES
Departamento de Informática y Ciencias de la Computación, Escuela Politécnica Nacional, Quito 170517, Ecuador
Corresponding author: Victor Velepucha (victor.velepucha@epn.edu.ec)

ABSTRACT Microservices architecture is a new trend embraced by many organizations as a way to modernize their legacy applications. However, although there is work related to the migration process, there is a gap in the body of knowledge related to the principles they should adopt when implementing a microservices architecture. This work presents a comprehensive survey, gathering literature that explores the fundamental principles underlying the object-oriented approach and how these concepts are related to monolithic and microservices architectures. In addition, our research encompasses both monolithic architectures and microservices, along with an investigation into the design patterns and principles utilized within microservices. Our contribution is present a list of patterns used in microservices architecture, the comparation between the principles expounded by the experts in the decomposition of microservices architectures, Martin Fowler and Sam Neuman, and the forerunner of the Principle of Information Hiding, David Parnas, who discusses modularization as a mechanism to improve flexibility and understanding of a system. Additionally, we expose the advantages and disadvantages of monolithic and microservices architectures obtained from the literature review carried out in summary form, which can help as a reference for researchers from academia and industry and finally reveal the trends of microservices architectures today.

INDEX TERMS Microservices, monolithic, decomposition, principles, patterns, migration.

I. INTRODUCTION

An application with monolithic architecture is one that, although it can be composed of several modules, has a single executable. On the other hand, an application with microservices architecture is a distributed application where all its modules or elements are microservices and can be run independently [1], [2]. Organizations can make the decision to maintain their monolithic applications by only updating the SDK and/or programming language version. However, another path followed by many companies is to undergo a migration process towards microservices architectures [3], [4], [5]. This migration process can have several factors that can influence the migration to microservices. For example, the complexity of the monolithic architecture, complexity of the data repository, organizational culture of the company, experience of the software development team in creating microservices, and the ability to make a proper division of the monolithic architecture to create the microservices. Then, migrating monolithic applications to a microservices architecture can be a challenging process. There is some work done in this regard; however, to date, there is no consensus on how to conduct this process successfully [6]. Some works recommend migrating a monolithic application to microservices only if it has many modules and its architecture is complex [6], [7], [8]. Migration case studies are reported, where a refactoring of a monolith system has been carried out, organizing it by business functionality, there are also recommendations towards reorganizing those monolithic applications that are small into components since they do not consider it convenient to migrate it towards a microservices architecture [9]. From the literature review conducted, few studies make migrations based on theoretical guidelines that guide towards a microservices architecture [10], [11]. Based on this research gap, we set the following research questions:

- RQ1. Are there principles that support the decomposition of monolithic applications towards microservices?

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

- RQ2. What are the advantages and disadvantages of monolithic architectures and microservices?

To answer the first research question RQ1, it is proposed to correlate the principles put forward by authors Martin Fowler and Sam Neuman, scholars of microservices architecture, and the forerunner of the Principle of Information Hiding, David Parnas. Our contribution is comparing and correlate these principles proposed by each author and the expected benefits. To answer the second research question RQ2, we perform some literature review regarding about the advantages and disadvantages of monolithic architectures and microservices and we our contribution is provide a summary list with an analysis of each item. Finally, case studies and success stories of using microservices exposed in the literature by authors from academia and industry are presented. The article proceeds as follows. Sections II and III present concepts and foundations of monolith, microservice, and different patterns of architecture and design are indicated. In Section IV the fundamentals of software engineering are exposed; in Section V the principles that guide the development of microservices are studied. In Section VI, a correlation is made between the principles for the decomposition of systems into modules proposed by Parnas and the principles that guide the development of microservices indicated by Fowler and Newman. In Section VII a study of the advantages and disadvantages of developing microservices in a monolithic and microservices architecture is carried out. In addition, Section VIII indicates case studies and success stories in migrating monolithic applications to microservices. Finally, in Section IX the conclusions and future work are presented.

II. BACKGROUND

Software Engineering is the study of methodologies and principles used for the development of computer systems [12]. It is a discipline formed by a set of methods, techniques, and tools. It is essential to use software engineering when creating computer systems, since hand in hand with the software development life cycle, the needs that a client has, the elicitation of requirements, analysis, design, construction of the software, functional and non-functional tests to finally deploy the system in a productive environment to be used by the end-user and continue with the incorporation of improvements and maintenance of the software. Without Software Engineering, creating applications would be a messy process, having a high probability of failures and failed attempts [13]. Currently, most software products have been developed mainly in two paradigms, the structured and the object-oriented. According to a study conducted by Ponce et al. [14], 90% of monolithic applications being migrated to a microservices architecture have traditionally been written using object-oriented programming languages, due to the fact that objects allow for greater malleability, and probably when migrating to microservices architectures, they will continue under the same object-oriented paradigm [15]. Therefore, we are going to indicate some concepts used in object-oriented approach and their relationship with monolithic and microservice architectures.

A. FOUNDATIONS OF THE OBJECT-ORIENTED APPROACH

There are several foundations found in the literature that guides the object-oriented approach, some state as principles the concepts of encapsulation, abstraction, polymorphism, and inheritance [16], while other authors extend these concepts to decomposition and information hiding [17], [18]. The foundations mentioned above are used both in monolithic and microservices architectures. In the next subsection we explain in detail each one of these concepts:

1) DECOMPOSITION

Decomposition in software consists of dividing a complex system into parts that are easier to understand, program, and maintain over time. In structured programming, algorithmic decomposition breaks a process into well-defined stages. In object-oriented programming, decomposition consists of dividing a large system into small classes that are responsible for solving a particular business rule or problem [19]. A decomposition paradigm in computer programming is a strategy for organizing a program as several pieces, and it usually involves a specific way of organizing a program text. Generally, using a decomposition paradigm is to optimize some metrics related to the complexity of the program, for example, the modularity of the program or its maintenance. Most decomposition paradigms suggest breaking a program into parts to minimize static dependencies between parts and maximize the cohesion of each part. Some popular decomposition paradigms are modules, procedural and object-oriented abstract data types. [18], [19] The functional, object-oriented, and database-level decompositions are described in detail below:

The functional or structured decomposition consists of creating functions and sub-functions that reduce the complexity of software. These subfunctions can be decomposed into smaller ones until they reach the degree of primitive functions, and this process is known as refinement. This paradigm is based on the idea of "divide and conquers" [20]. In structured decomposition, the decomposition of the system into small functions, according to steps or functionalities, can be generated with tools obtaining a structured diagram that shows the relationships between various functional elements of an application [18]. A schematic of a functional decomposition can be seen in Figure 1. Some use cases where functional decomposition can be applied are in software development to divide large systems in small modules or functions. Functional decomposition is also an effective approach for analyzing and designing complex systems, breaking down the system into functional components, it becomes easier to understand.

Oriented Decomposition refers to decomposing based on the domain of a problem by identifying objects and their relationships (messages) between them. By having objects

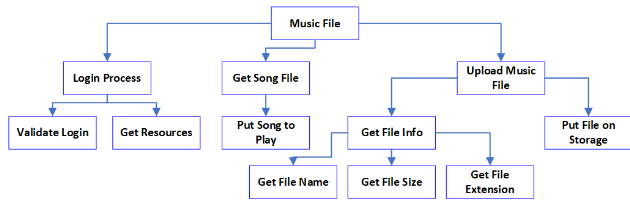

FIGURE 1. Functional decomposition example.

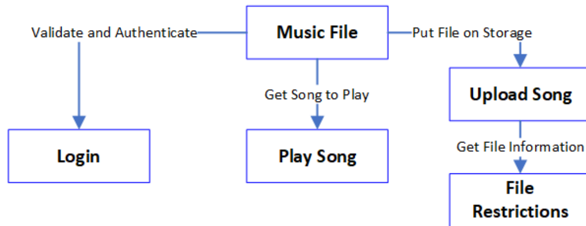

FIGURE 2. Object-Oriented decomposition example.

with unique behaviors and capable of solving a specific problem, this form of decomposition for many is simpler, both to understand and to use since the way to interact is by sending messages between objects [18]. Several tools on the market can generate a class diagram and their relationships from a conceptual model; some of these tools also serve to obtain these diagrams through reverse engineering, one of the best-known being Power Designer. An example of object-oriented decomposition can be seen in Figure 2.

According to experiences, Functional vs Object Oriented Decomposition are valid ways of decomposing, however when complex systems are built, object-oriented decomposition has advantages over functional decomposition, since it reduces complexity, which helps to understand it, an object can be defined through its attributes and behavior, which allows it to evolve and make changes to it [18]. Both functional and object-oriented programming, we can create microservices. For example, if we program with JavaScript using functions, along with Node.js and the Express package, we can create a microservice that performs CRUD operations. Alternatively, we can use ASP.NET Core and create a microservice using classes and methods, with both approaches being completely valid paths.

2) ABSTRACTION

Another important foundation considered by several authors is that of abstraction. Abstraction is a mechanism that allows us to represent a complex reality in terms of a simplified model to suppress irrelevant or secondary details to improve understanding [21]. According to Kramer [22], formal modeling and progressive analysis have shown effective methods in terms of practicing and developing abstract thinking and consolidating related cognitive ability to apply abstraction.

In terms of Software Engineering, Booch [23] mentions that abstraction focuses on the external view of an object, focusing only on the similarities of certain objects and discarding their differences, and therefore serves to separate the essential behavior of an object of its implementation. On the other hand, Wing [24] cites abstraction as the extraction of important information and discarding irrelevant data from complex systems to generate patterns and find common ground between different representations.

The human being, due to his learning and memory limitations, and for a better understanding of complex systems, has learned to fragment to abstract and then understand a particular problem. Through abstraction, a piece of information is taken to analyze and understand it better.

Object-oriented programming makes it easy to apply abstraction by creating real-world entities that represent a dense and cohesive grouping of information [18]. The definition of abstraction for Software Engineering and particularly in Software Design is evidenced through manifestations such as the abstract type of data and object-oriented programming.

3) INHERITANCE

Inheritance is a term proper to the object-oriented approach. In object-oriented programming, inheritance occurs when an object or class is based on another parent object or class, using the same implementation or behavior, achieving reusability and extensibility. Through it, designers can create new classes starting from a class or a pre-existing class hierarchy, thus avoiding redesign, modification, and verification of the part already implemented. Inheritance makes it easy to create objects from existing ones and implies that a subclass gets all the behavior (methods) and attributes (variables) of its parent class. Inheritance is one of the mechanisms used by object-oriented programming languages, through which one class is derived from another in a way that extends its functionality. The class from which it is inherited is often called the base class, parent, superclass, or ancestor class. In programming languages with a strong typing system, inheritance is usually a fundamental requirement to be able to perform Polymorphism [18]. To validate if we are making correct use of inheritance, we can check it with the Liskov principle, which tells us that if in some part of our code we are using a class, and this class is extended, we have to be able to use any of the child classes and that the program remains valid, which forces us to make sure that when we extend a class, we are not altering the behavior of the parent.

4) POLYMORPHISM

In object-oriented programming, polymorphism refers to the possibility of defining different classes that have identically named methods, but that behave differently in execution; in this way, it is possible to send syntactically the same messages to objects of different types, obtaining different results. Some object-oriented programming languages allow two objects of different class hierarchies to respond to the same methods through so-called interfaces. Two objects that implement the

same interface can be treated identically as the same type of object defined by the interface. Thus, different objects can be exchanged at run time as long as they are of the same type. When speaking of ''inheritance'', this concept is different from ''interface polymorphism'' because a class that implements an interface only obtains its data type and the obligation to implement all its methods; it does not copy behavior or attributes. However, one or more additional types can be added to the implemented class, which, together with the composition, avoids the need for multiple inheritances and favors a broader use of polymorphism [25].

5) INFORMATION HIDING

Parnas [17] introduced ''The Information Hiding Principle'' in 1972. It is a principle that consists of: a) ''Allow modules to be reassembled and replaced without reassembly of the whole system'', which refers to create small modules and that by design, they are displayed and exposed independently in such a way that they are like a black box, without having to know what language or technology they are created with, and b) ''Allow one module to be written with little knowledge'', allows internal changes to be made in each module without the user of the module having to worry about how the internal implementation is done. Parnas, when recommending decomposing systems into quasi-decomposable modules (dividing systems into modules) that have the least dependence on each other, indicates that these benefits can be obtained: a) Short development times, b) Flexibility in the development of a system, c) Understandability of a system. Short development times are achieved when it is possible to separate an application into modules that allow group work to be carried out with as little communication as possible. The flexibility is that drastic or high-impact changes can be made to modules without affecting other modules. Understandability refers to the fact that you can understand one module at a time without having to understand the entire system. This can benefit a development team being specialized to maintain and add new functionality in a specific module [17], [26].

6) ENCAPSULATION

Encapsulation is the mechanism for hiding the data implementation of an object by restricting its access through public methods. The packaging of the variables of an object with the protection of its methods is called encapsulation. Typically, encapsulation is used to hide unimportant implementation details from other objects, and implementation details can change at any time without affecting other parts of the program. In OOP (Object-Oriented Programming), instance variables are kept private, and access methods are made public. The benefit you get is that when you use the class, you can bypass the implementation of the methods and properties to focus on how to use them. On the other hand, the user is prevented from changing his state in unexpected and uncontrolled ways. [18].

B. MONOLITHIC AND MICROSERVICES WHAT ARE THEY?

Monolithic and microservice architectures are two different approaches to designing and building software systems. The choice between monolithic and microservice architecture depends on several factors, and there is no definitive answer to which one is the best. Both architectures have their own strengths and weaknesses, and the decision should be based on the specific requirements and constraints of the project.

1) MONOLITHIC ARCHITECTURE

A monolithic application is one that has a single executable, although it can be made up of one or more modules, with the restriction that these modules cannot be run independently [1]. An application can be divided into several logical layers; however, the deployment of all its layers that contain its functionality is carried out jointly as if it were a single block, while at the same time a single process is created unique [27]. Generally, a monolithic application or monolithic systems have the characteristic of using a single code base to expose their services or functionalities. From object-oriented programming, through application development based on an SOA (Service Oriented Architecture) architecture, creating monolithic applications has been the alternative followed by most organizations; however, as the complexity of these applications increases due to the rise in business functionalities, maintaining this type of architecture has become unsustainable, and it is also difficult to incorporate new cutting-edge technologies in an application that carries an architecture that is too rigid and standardized for the use of a single technology.

A reference architecture for monolithic applications is that of an application that has divisions in several logical layers. For example, in the 3-layer pattern like the one shown in Figure 3, a web application can be split into multiple logical layers in the following order: *Presentation*, *Business* and *Data Access* layer. In the presentation layer, the logic is placed to respond to the interactions that the user makes with the application through a web browser, in the business layer, the rules and validations of business functionalities are placed, and in the data access layer has all the mechanisms to make calls either to databases or external services that the application needs.

Let's consider a concrete example where we are responsible for maintaining a monolithic music streaming application similar to Spotify. In a requirement, we are asked to scale only the functionality of the music player as it is the most used by users. Additionally, we are asked to make a functionality change to include a new field, which would be the number of songs plays. In this case, any small change made in the monolithic application and poorly tested in functional tests could cause the entire system to be unavailable for minutes or hours since the monolithic application is deployed as a single block. Scaling only the music player functionality would not be possible as scaling is done for the entire monolithic application rather than individual parts. The deficiencies in

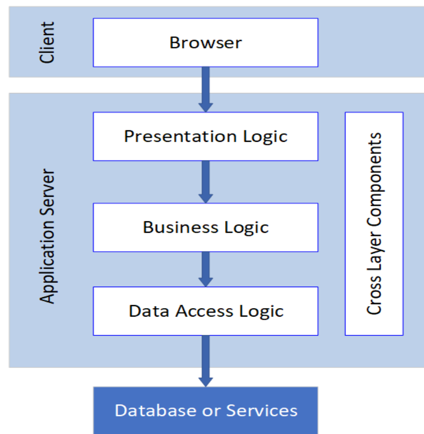

FIGURE 3. Monolith architecture reference.

terms of quality attributes for this example would include difficulties in scalability, maintenance problems, limited flexibility in making changes, compromised availability and reliability, and longer development times [4], [28].

In the case of web applications, depending on the number of concurrent users that can access the application, the application can be deployed on a web server and configured so that it can scale horizontally to what is known as *webgarden*, with which more than one running process is created that will allow to attend more requests from users. Another alternative to meet a high demand from concurrent users is to deploy the application on several servers, known as a distributed monolith or vertical scale, and its call is exposed through a load balancer [27].

The applications after their construction and delivery for their operation continue to evolve due to requests for changes, the incorporation of new requirements, or solving bugs. When an estimate is required to determine the effort of a change in a monolith application, it is a challenging task since any change, no matter how small, can affect the entire system's behavior, especially in a large and complex monolith application, this being its main disadvantage. What demands more development effort and cost in a monolith application created is its maintenance. Figure 4 shows an example of monolithic architecture and microservice architecture, where it is observed that each microservice can be deployed independently; in addition, some microservices make calls to their own database.

2) MICROSERVICE ARCHITECTURE

On the other hand, a microservice is an element of an application that can be executed independently, unlike a monolith application, where each microservice is developed to fulfill a specific business functionality and is not developed based on logical layers such as a monolith application, has a flexible coupling and high functional cohesion; they can interact with each other through a messaging system [1]. These small services can be developed independently and can

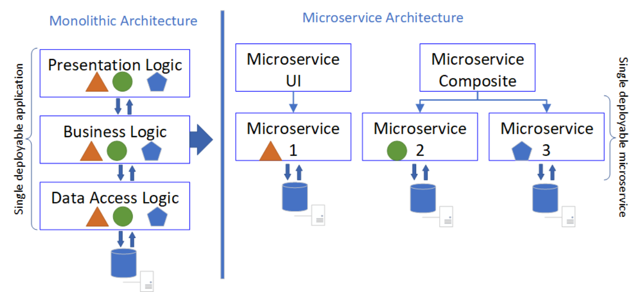

FIGURE 4. Monolithic vs Microservice achitecture.

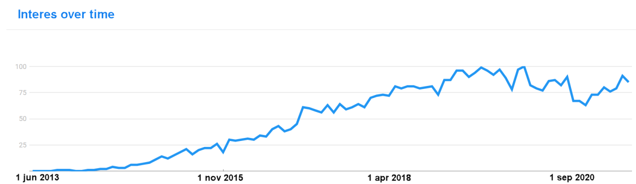

FIGURE 5. Microservice search term trends over time.

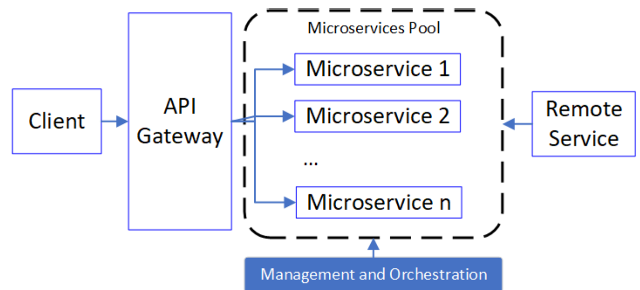

FIGURE 6. Microservice reference architecture.

be operated in the same way as in a monolith application [10]. Currently, the use of microservices has become popular due to the ease of development where each microservice fulfills a specific business functionality, the comfort of scaling based on the use of containers, and its ease of deployment in cloud computing environments. However, challenges such as managing and monitoring these microservices must be considered. In the case of a migration from a monolith application to microservices, the main objective is to decompose an application into small autonomous services that function independently.

The term microservice was recently mentioned in 2014 by Martín Fowler [2], and since that year, interest in this term has grown, as can be seen in Figure 5.

Microservices architecture is a distributed application where all its modules or elements are microservices, and these can run independently [1], [2]. A reference architecture of microservices can be seen in Figure 6.

The current trends are focused on the migration of monolithic applications to microservices and their deployment in cloud computing [4]. There are also works that mention automation in the compilation and deployment of migrated

applications using DevOps tools [7]. Currently, the number of proposals for adopting microservices architectures has been increasing over the years, reducing the complexity of systems, and achieving greater flexibility than monolithic architectures [29]. There are studies in the literature that demonstrate that many companies are migrating their monolithic applications to a microservices architecture [1], [30]. Security in microservices is a crucial aspect that is also gaining momentum, such as employing Mutual Authentication of Services Using Mutual Transport Layer Security (MTLS), guidelines, and best practices in designing security in microservices, utilizing secure protocols, and using security tokens for authentication [31]. Serverless and Function-as-a-Service (FaaS) are concepts that are also trending alongside microservices, as they allow for the development of stateless functions that can easily be deployed in the cloud computing environment. These functions automatically scale up or down based on demand [32].

III. PATTERNS, PRACTICES, AND PRINCIPLES FOR MICROSERVICES

Each programmer can carry out the implementation of software differently, having different approaches and points of view to solve a need or problem posed. However, it is essential to standardize the design and construction of the software, with the aim that the application built is scalable, easy to test, and promote the reuse of components, for which in the design and construction of the software it is necessary the use of patterns. It is essential and important from the beginning of development to establish correct design patterns. Some of the most important patterns used when designing microservices are explained below, especially when migrating from a monolithic application to microservices.

- Domain-Driven Design

Domain-Driven Design (DDD) is a set of tools that help design and implement quality and high-value software both in the strategic and in the tactical part, helping in the design of software and its integration with the business [33]. The Domain-Driven Design pattern is used to develop a vision of the system to be developed, generating a business domain model from where the knowledge of the business functionalities is extracted, generating for example, "smart cases", in such a way that it can build a common language between software developers, architects, and domain experts. Generating diagrams brings benefits such as identifying subdomains, identifying functionalities that are closely related to other functionalities, and identifying functionalities that are priority or important for the company, dependencies between functionalities, calls to other external systems or third-party services, all of which they must be included in the domain model. By decomposing a system into so many small microservices that perform a single business functionality with the support of the DDD to delimit the limits that each microservice must-have, it is possible to obtain microservices with high cohesion [34].

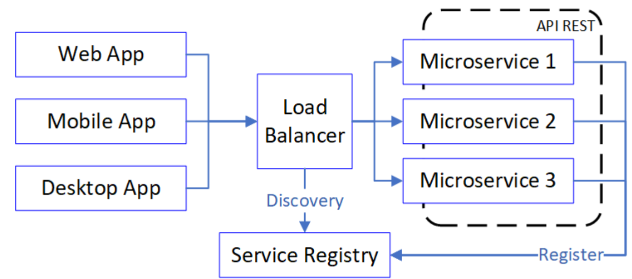

FIGURE 7. Service discovery pattern.

- Service Discovery Pattern

When microservices are designed and created, an API is generally used, which offers the necessary services so that the application can interact with the backend. Microservices require knowing where these services are hosted. With the Service Discovery Pattern, the idea is that the services themselves, when they are started, are registered in an entity called Service Registry, which takes control of all active services in such a way that when you want to consume a service, you search the Service Registry available instances. Each service, when starting, indicates its name and the address where the service is located; in this way, the Service Registry has a record of all the services that are available. In addition to the initial registration, the services have to send signs of life from time to time so that the Service Registry knows that the service is still available; this is known as heartbeat. If a service is not reported, the Service Registry will know that there is something wrong with that service and will assume that it is not available so that all requests will be redirected to the other instances. This pattern allows us to run a specific service without knowing the physical address of the service; for this, a load balancer is used that will rely on the Service Registry to know the real location of the service.

An example of this architecture pattern can be seen in Figure 7.

- Data-Driven Design

Data is one of the most critical assets for any organization; therefore, being able to collect and manage it correctly gives a competitive advantage over the competition. Data-driven design (D-DD) are designs that have been created using the data collected from the analytics of an application, with which you can find out information that is important or relevant from the users who use the application, rethink the visual design to facilitate the usability and accessibility of the application, reorder and adapt the content to suit the user's needs, thereby being able for example to maximize sales [35]. If an application has been made using D-DD, there is better transparency when migrating to microservices; since the business functionalities are already separated, there is a loose coupling. Microservices are inspired by the proposals given in the domain-driven design, where you must first

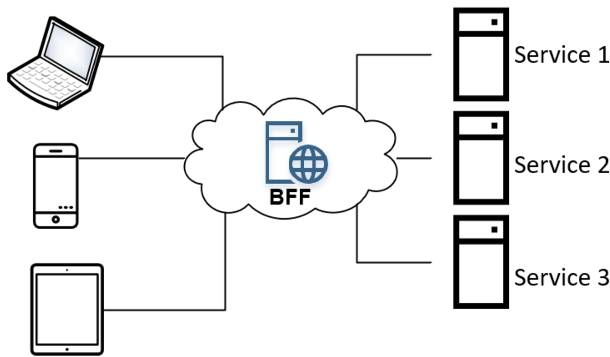

FIGURE 8. BFF pattern example.

define a delimited context to a particular business problem, which perfectly matches the principle of single responsibility that is used when creating a microservice.

- Backend for Frontend Pattern

The Backend for Frontend (BFF) pattern is an architectural software pattern that improves the way in which data is obtained between clients, be it a browser, mobile application, or any device connected to the Internet and servers hosted on-premises or in Cloud. The goal of this pattern is to decouple the frontend applications from the backend architecture. When using this pattern, if a new service is added, there is no need to make any changes between the frontend applications and the backend, only to adapt the calls to the BFF service. Its main benefit is allowing you to isolate the backend from the frontend. An advantage of this pattern is code reuse because all clients have the ability to get data from the same BFF server, making it fewer complex [36]. When creating microservices, an example of this BFF architecture pattern can be seen in Figure 8.

In this example shown in Figure 8, there are three types of clients: web application, mobile application, and third-party external application. You can create one or three different API Gateways.

- Adapter Microservices Pattern

The Microservices Adapter pattern adapts, as needed, between a business-oriented API built using light or RESTful messaging techniques, with the same domain-driven techniques as a traditional microservice, and a legacy API or traditional SOAP-based service. Improved Webservices (WS- *). This adaptation is necessary, for example, when a development team does not have decentralized control over an application's data source. An adapter microservice wraps and translates existing services into an entity-based REST interface. This type of microservice treats each new entity interface as a microservice and builds, manages, and scales it independently. In many cases, converting a function-based interface (for example, one built with SOAP) into a business-concept-based interface is easy; it's like going from a verb-based (functional) approach to a noun-based (entity) approach [37].

- Strangler Application Pattern

The strangler pattern helps us manage the refactoring of a monolithic application in stages. The idea is that it uses the structure of a web application, the fact that they are built from individual URIs (Uniform Resource Identifiers) that are functionally assigned to different aspects of a business domain, to divide an application into different functional domains and replace those domains with a new microservices-based implementation done one domain at a time. These two aspects form separate applications that live together in the same URI space. Over time, the new refactored application replaces the original application until you can finally shut down the monolithic application. The strangler pattern includes these steps: *Transform*, which consists of creating a new parallel site and transforming it incrementally *Coexist*, where you have two sites, the original one and the migrated one where it is for a while. It will gradually redirect from the existing site to the new site for the newly implemented functionality *Remove*, where the old functionality of the existing site is removed, and traffic is redirected away from that part of the old site. The best way to apply this pattern is to create an incremental approach to microservices adoption, one in which, if you find that the method doesn't work in your environment for some reason, you have a simple way to change direction if necessary. There are two ways to apply the Strangler pattern: Refactoring your back-end for microservices design (the inner part) or Refactoring your front-end to accommodate the microservices and also to make any new functional changes (the outer part) [38].

- Shared Data Microservice Design Pattern

The pattern indicates that in a migration process from a legacy application to microservices, in the transition phase of the migration, consider temporarily using the same database repository so that the migration process is smoother. For this, the limits of the business functionality must have been clearly defined previously in such a way that microservices with loose coupling and high cohesion are created; for this, it relies on DDD. It must be taken into account that if at the end of the migration process, a single database repository is maintained, sharing data in a single repository can be considered an anti-pattern [39]. An example of this pattern can be seen in Figure 9.

- Aggregator Microservice Design Pattern

An entity is an object that is distinguished primarily by its identity. Entities are the objects in the modeling process that have unique identifiers. Entity objects need well-defined object lifecycles and a good definition of what the root identity relationship is. A good entity-relationship model has well-defined entities, and each entity has a well-known specific identifier, but they may never live independently. A combination of entities is an aggregate. In cases where you have a group of entities that must be kept consistent, you can refer to those entities as dependent entities, and you must make sure you know what the root of the aggregate is

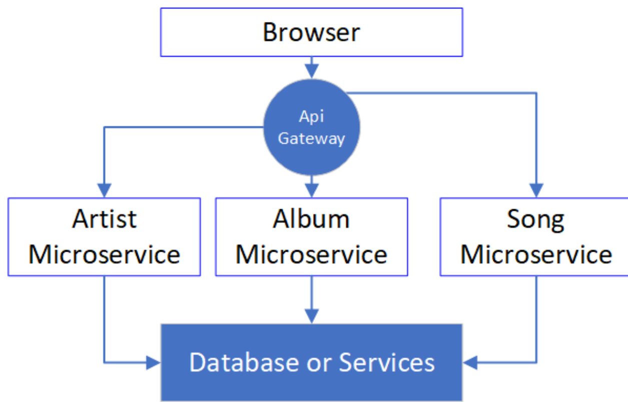

FIGURE 9. Shared Data Microservice Design Pattern Example.

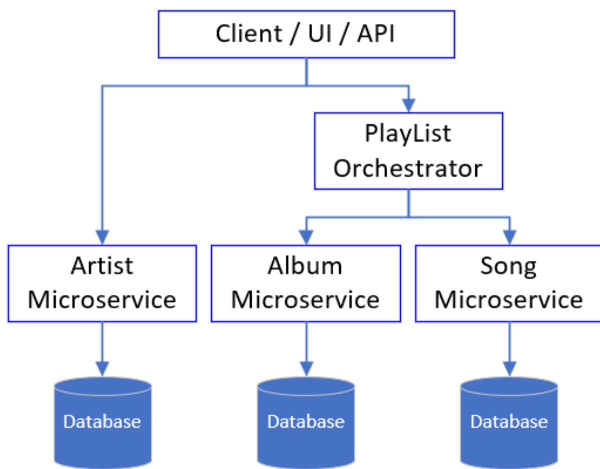

FIGURE 10. Aggregator microservice design pattern example.

since the root defines the life cycle of the entities dependents. For development teams, entity and aggregate patterns are useful for identifying specific business concepts directly mapped into microservices, performing end-to-end business functions. Business microservices tend to be stateful and tend to own their own data in a database managed by [40]. The Figure 10 show an example of this pattern.

- Summarizing the methodologies and patterns

The different methodologies and patterns mentioned are summarized in the Table 1, according to their purpose, usage scenarios, and key benefits.

A. PRINCIPLES THAT GUIDE THE CREATION OF MICROSERVICES

Multiple principles can be used as a reference when creating microservices, which have emerged over the years; among the most important are those proposed by microservices experts Martin Fowler and Sam Newman [10]. On the one hand, Fowler exposes nine principles [2] where it indicates the characteristics that microservices should have when they are built, emphasizing that not necessarily every microservice should have or comply with all these principles. On the other hand, Newman, in his book "Building Microservices",

TABLE 1. Summarized methodologies and patterns for microservices.

Name	Intent	Usage Scenario	Key Benefits
Domain-Driven Design	Design and develop software systems based on the domain model	Complex domains with rich and evolving business logic	Clear alignment with the domain, encapsulation of domain knowledge, modular and maintainable design, improved collaboration between teams.
Service Discovery Pattern	Facilitate dynamic service discovery and communication	Microservices architectures with service interaction	Dynamic and automatic service registration, decoupling between services, load balancing, fault tolerance
Data-Driven Design	Design software systems around the behavior and structure of data	Applications where data is a key component	Focus on data-driven decision-making, improved data integrity, optimized data access and manipulation, alignment with business objectives
Backend for Frontend Pattern	Develop back-end services tailored to specific frontend clients	Applications with multiple frontend clients	Customized and optimized APIs for each frontend client, improved frontend-backend decoupling, better user experience
Adapter Microservices Pattern	Adapt and transform data or functionality between microservices	Microservices architectures with different interfaces	Enable seamless communication between microservices with varying protocols or data formats, promote interoperability
Strangler Application Pattern	Incrementally migrate from a monolithic system to microservices	Legacy monolithic applications	Gradual migration without disrupting the existing system, reduced risk, improved maintainability and scalability
Shared Data Microservice Design Pattern	Manage and share data between microservices	Microservices architectures with shared data requirements	Centralized data management, data consistency, reduced data duplication and redundancy, improved data integrity
Aggregator Microservice Design Pattern	Aggregate data or functionality from multiple microservices	Applications requiring consolidated information	Centralized data aggregation, reduced client complexity, improved performance, reduced network overhead

TABLE 2. Principles comparative by authors.

Concept	Fowler [2]	Newman [36]	Parnas [17]
Modularity	Componentization via services	Hide internal implementation details	Allow modules to be reassembled and replaced without reassembly of the whole system
DDD	Organized around business capabilities	Model around business concepts	Allow one module to be written with little knowledge
Decentralize	Decentralized governance	Decentralize all the things	—
Automation	Infrastructure automation	Adopt a culture of automation	—
Failure	Design for failure	Isolate failure	—
Products	Products not projects	—	—
Endpoints	Smart endpoints and dumb pipes	—	—
Management	Decentralized data management	—	—
Design	Evolutionary design	—	—
Deploy	—	Independently deployable	—
Observability	—	Highly observable	—

highlights seven principles [36], in which he includes guidelines at both a technological and organizational level on how to create good microservices. When carrying out an analysis of these principles, we realize that there are points in common between the principles proposed by these authors.

Considering that microservices are a way of decomposing architectures, forms of decomposition have been sought in the literature, one of the most relevant being the decomposition exposed by Parnas in 1972, in which two principles are exposed [17] to decompose systems into modules, which we see that despite having been in effect for many years, we can also contrast them with the principles set out by Fowler and Newman. Next, in Table 2 we show a comparative list of these principles with an explanation of them. A detailed analysis of these microservices principles is indicated below, taking into account that the principles presented by Martin Fowler will be taken as a pivot:

1) Componentization via services

This principle was formulated by Fowler et al. [2] and can be mapped with the principle of Newman et al. [36] which indicates ''Hide Internal Implementation Details'', and with Parnas [17] ''Allow modules to be reassembled and replaced without reassembly of the whole system'' which refers to creating small microservices and that by their design, these microservices are independently deployable and exposed in such a way that they are like a black box, without the need to know what language or technology they are created with.

2) Organized around business capabilities

This principle was formulated by Fowler et al. [2] and can be mapped with the principle of Newman et al. [36] Model Around Business Concepts, and Parnas ''Allow one module to be written with little knowledge''. It refers to the fact that the modular division of microservices should be based on business capabilities and not on technical layers such as presentation, business, and data access layers.

3) Decentralized governance

This principle [2] is related to ''Decentralize All the Things'' [36] and refers to the fact that microservices must be autonomous, with autonomous work teams, where each team is responsible for the group of microservices that correspond to a product. You can have a shared governance model that can serve as a reference for different members of the work teams.

4) Infrastructure automation

This principle [2] is related to ''Adopt a Culture of Automation'' proposed by Newman et al. [36] and refers to using DevOps tools to automate the merge of the source code, thus having Continuous Integration in an automated way, as well as using tools for continuous Deployment for test, pre-production and production environments with which it can be reached to a Continuous Delivery, getting to release new versions in an automated, controlled and frequent way. Automated testing should also include automated functional and technical testing.

5) Design for failure

Design for Failure [2] relates to the ''Isolate Failure'' proposed by Newman et al. [36]. It refers to the fact that given the design of a microservice to work independently and autonomously, in the event of a failure, only this microservice will not be available from the entire swarm, the rest of the microservices being able to continue their operation, thus having tolerance to failures. By incorporating mechanisms such as microservices deployments in containers such as Docker, monitoring tools such as Grafanam Heapster, Prometheus can be had, with which it can be automated that in the event of a microservice crash, through configurations, the fallen microservices can be automatically raised.

6) Products not projects

Products not Projects [2]. It refers to creating multifunctional teams (full-stack developers) with a wide range of skills that not only build the software but also do the testing, deployment, monitoring, and solve application issues in a productive environment. An example of a monolithic organization and another that works with microservices can be seen in Figure 14.

7) Smart endpoints and dumb pipes

Smart Endpoints and Dumb Pipes [2] refers to avoiding creating microservices that are too heavy, since a microservice has to do data transformation, message routing, and possibly apply business rules, being a recommendation, that communication mechanisms be light using HTTP messaging with REST.

8) Decentralized data management

Decentralized Data Management [2]. This principle matches the principle ''Hide Internal Implementation Details [36]

TABLE 3. Principles benefits by authors.

Fowler [2]	Newman [36]	Parnas [17]
Strong Module Boundaries	Organizational Alignment	Managerial (Ease of work distribution)
Technology Diversity	Technology Heterogeneity	Comprehensibility
Independent Deployment	Ease of Deployment, Scaling, Resilience, Composability, Optimizing for Replaceability	Product flexibility

proposed by Newman et al. [36], and also by Parnas [17]. It refers to the fact that each microservice must manage or administer its own data, which is to say, that in the design of a microservice, it must connect to its own database or storage. This makes a great difference with a monolithic architecture, where there is a single database repository for the entire application. Figure 4 shows an example of monolithic architecture and microservice architecture, where it can be seen that each microservice has a database repository.

9) Evolutionary design

Evolutionary Design [2] refers to the fact that given the design of a microservice that is autonomous and independent, its evolution can also occur independently or, if required, a microservice can be replaced by a new implementation without affecting service to consumers. It also refers to the fact that if a microservice is no longer used, it can be safely terminated.

10) Independently deployable

Independently Deployable [2] refers to the fact that due to any change either due to code maintenance or evolution of a microservice, this microservice is capable of being deployed independently, without affecting other microservices and without affecting the entire application. Likewise, mechanisms with Docker Swarm allow new versions to be created as instances, while old versions of microservices live until they finish serving requests.

11) Highly observable

Reference architecture of microservices indicates that there must be monitoring tools, so in the design of a microservices architecture, it must be decided which monitoring tools can be used; there are various tools on the market such as Prometheus, Spigo, Heapser, Grafana, Trace, Sensu.

B. EVALUATION AND DISCUSSION OF PRINCIPLES

If the principles mentioned by the authors are maintained, benefits are obtained, which are mentioned in Table 3 and detailed below.

In ''Strong Module Boundaries'', Martin Fowler [2] indicates that microservices reinforce doing work in a modular way, which is essential when working with large development teams. On the other hand, Newman [36] emphasizes the fact that having small development teams that work on specific microservices tend to be more productive, managing to better align with what the organization needs, which helps us minimize the number of people working in a particular code reaching the sweet spot of team size and productivity in an organization. In *Managerial*, Parnas [17] refers to being able to manage development teams that work on specific modules independently, and that over time these developers acquire more expertise and at the same time become more autonomous in the work they do.

For ''Technology Diversity'', Fowler [2] states that each microservice is an independently deployable unit, having the freedom to code a microservice in different programming languages or use any databases, allowing the developer team to choose the most suitable tool, language, and libraries to solve specific problems. Newman [36] also agrees on this benefit, since having a system made up of multiple collaborative microservices, you have the flexibility of being able to decide to use different technologies for each one according to the work they are going to do, instead of having to select a more standardized approach. For example, if an application of the social network type, the interactions of the users can be stored in a graph-oriented database to reflect the highly interconnected nature of a social graph, and for the publications that users would be stored in a document-oriented data warehouse, resulting in a heterogeneous architecture. For *Comprehensibility* Parnas [17], the benefit is to be able to study a system module by module without having to understand the whole system from the beginning as it can be a complex task. This can benefit that a development team may be specialized to maintain or add new functionality in a specific module. Also, understand each module and create a microservice for each business logic in any programming language.

Independent Deployment [2] means that microservices, being modular and simple, are easier to implement, and since they are autonomous, they are less likely to cause system-wide failures. In microservices, deployment is carried out in an agile way, relying on continuous delivery, reducing the time between the cycle between an idea and the running software, with the benefit that organizations can respond quickly to market changes and introduce new features faster than your competition. On the other hand, Newman [36] indicates that a change in a line of code in a monolithic application necessarily implies releasing a new complete version of the application, having a high impact and risk, but with microservices, a change can be made in a single service and deployed independently from the rest of the system, should a problem occur, this individual service can be quickly isolated, making fast rollback easy to achieve. It also means that new functionality can be delivered to customers faster, which is why many organizations such as Amazon and Netflix have chosen to use these architectures. Identifying reusable microservices allows them to be reused by different consumers, for example, web applications, desktop applications, mobile applications. In the case of resilience, by configuring monitoring tools, if one microservice fails, the rest of the system can continue to function. In contrast, in a monolithic system, if one service fails, everything

stops working. Regarding scalability, since microservices are small, it allows scaling only those services that are most in-demand, being able to optimize the use of hardware. Finally, since microservices are small, the cost of replacing them with a better implementation or eliminating them is much easier. Development teams using microservices approaches are more comfortable with either completely rewriting a microservice or simply removing it when it is no longer needed, and the cost of replacing it is quite small. Correlating with what Parnas indicates [17], it refers to having the flexibility of being able to make changes in a specific module without causing it to affect other modules. After the analysis presented, the first research question is answered *Are there principles that support the decomposition of monolithic applications towards microservices?*, We confirmed that principles were found that lead to the decomposition of microservices and the decomposition of architectures in general.

On the one hand, David Parnas, is based on the principles: ‘‘Allow modules to be reassembled and replaced without reassembling the whole system’’ and ‘‘Allow one module to be written with little knowledge’’. On the other hand, Martin Fowler proposes nine principles: Componentization via Services, Organized around Business Capabilities, Products not Projects, Smart endpoints and dumb pipes, Decentralized Governance, Decentralized Data Management, Infrastructure Automation, Design for failure, Evolutionary Design. Finally, Sam Newman proposes seven principles: Model Around Business Concepts, Adopt a Culture of Automation, Hide Internal Implementation Details, Decentralize All the Things, Independently Deployable, Isolate Failure, Highly Observable.

It was possible to show that the principles proposed by Fowler and Newman are correlated with the decomposition proposed by David Parnas when he introduced the Principle of Information Hiding. At the same time, there is a correlation between the principles proposed by Fowler and Newman.

The principle proposed by Parnas ‘‘Allow modules to be reassembled and replaced without reassembly of the whole system’’ correlates with Fowler’s principle ‘‘Componentization via Services’’ and Newman’s ‘‘Hide Internal Implementation Details’’, refers to creating small and that by design, these microservices are independently and easy to replace, they are exposed in such a way that there is no need to know what technology they are created with.

Another correlation exists between the principle proposed by Parnas ‘‘Allow one module to be written with little knowledge’’, the principle ‘‘Organized around Business Capabilities’’ proposed by Fowler, and the principle ‘‘Model Around Business Concepts’’ indicated by Newman. This principle refers to creating microservices-based on business capabilities, which leads us to create small microservices that fulfill a single functionality.

Between Fowler and Newman, we find the following correlations. On the one hand, Fowler proposes ‘‘Decentralized Governance’’ while Newman is more radical and proposes ‘‘Decentralize All the Things’’. These principles refer to the fact that when it comes to creating microservices, the hierarchical structure of an organization must be changed, where responsibility is transferred to work teams. These work teams in turn become autonomous, and each team is responsible for a group of microservices, which over time generates advantages such as experience in the business functionality of the microservices created.

On the other hand, Fowler proposes ‘‘Infrastructure Automation’’ while Newman indicates ‘‘Adopt a Culture of Automation’’. Both authors refer to using DevOps tools to automate the compilation, deployment, and delivery of software. In addition, it is recommended to use tools to monitor the correct execution of the microservices. In the event of a microservice crash, a new one can be automatically created again.

The last correlation between the principles proposed by Fowler and Newman is that Fowler proposes ‘‘Design for Failure’’ and Newman ‘‘Isolate Failure’’ and refers to that when a microservice is designed, it is considered to have loose coupling, of such that it works independently. In the event of a failure, only this microservice will not be available, and there will be no impact on the entire system.

The purpose of this survey is to evaluate the principles indicated by experts in microservices such as Martin Fowler, Sam Newman and contrast them with what David Parnas indicates. However, to mention some other renowned microservices experts we have Chris Richardson, who authored the book *Microservices Patterns*, that serves as a practical guide for designing, implementing, and deploying microservices-based systems [37]. Adrian Cockcroft is another prominent figure in the software industry, particularly known for his advocacy and promotion of microservices architecture, who played a significant role in popularizing and advancing the adoption of microservices, especially during his tenure at Netflix, he was instrumental in transforming the company’s monolithic architecture into a highly scalable and resilient microservices-based architecture. He emphasizes the importance of building loosely coupled and independently deployable services that align with the organization’s business domains [41]. Another expert is Eberhard Wolff, who write the book *Microservices: Flexible Software Architecture*, that provides valuable insights and practical guidance for designing, implementing, and maintaining microservices-based systems, introducing the fundamentals of microservices architecture and explaining the motivations behind adopting this architectural style, emphasizing in the benefits of scalability, flexibility, and autonomy that microservices offer compared to traditional monolithic architectures [42].

IV. ADVANTAGES AND DISADVANTAGES OF MONOLITHS AND MICROSERVICES

Monolithic applications, having all their components together and are easy to develop. However, as new functionality is added, their degree of coupling increases and becomes complex, making them more challenging to maintain since any

change, no matter how small it is, involves the deployment of the entire application. Microservices have emerged as a new alternative to handle monolithic application problems. Microservices consist of decomposing a monolithic application into various small services that are responsible for business functionality. It could be said that microservice architecture is an evolution of SOA (Service Oriented Architecture) since each microservice can evolve independently, which benefits since any change in a microservice does not affect the other microservices. However, there is complexity when adopting this new microservices architecture, such as the difficulty of understanding the business logic of a legacy application for the migration to microservices and exist a lack of reporting of migration experience, which makes it necessary to carry out this process be challenging. Here are a number of advantages and disadvantages of a monolithic and microservices architecture.

A. ADVANTAGES WITH MONOLITHS

It cannot be said that an application with monolithic architecture is bad. Instead, experts recommend sticking with this architecture if the application is small [43]. Next, we are going to expose a series of advantages that a monolith application has.

1) APPLICATION AS A SINGLE BLOCK

When a monolithic application is small, its development and deployment are simpler because it is treated as a single block. In addition, a developer can track source code end-to-end more quickly, which is useful when working in small development teams. Monitoring can be a simpler task [27].

2) EASE OF CODE REUSE

Another advantage is being able to reuse a specific part of the code; only a portion is copied within the monolith application, having control over the changes made, which is simpler than making changes in distributed systems where it could affect other consumers [27].

B. ADVANTAGES WITH MICROSERVICES

Through the literature review carried out, many advantages are found when using a microservices architecture; below, a list of the advantages found with their explanation is exposed.

1) ALWAYS ONLINE

When releasing new versions of a microservices, it is not necessary to decommission the entire system or restart the service by having a good architecture of microservices, deployed in containers and by using DevOps with Continuous Integration and Continuous Delivery/Deployment can be carried out hot, in such a way that new requests from new clients will automatically call the new microservices, while the existing microservices that are running when their requests end or reach zero will disappear (shutdown) [44]. Likewise, in case of failures, only the specific microservice will be affected, not the entire system, so the rest of the modules can continue working without problem [1]. Instead of using virtual machines, using containers to place microservices has excellent advantages, such as scaling them when there is more demand for specific microservices. Docker is the most popular platform used by many companies to create containers and deploy our microservices, to have efficiency and speed of provisioning, and together with other tools, it is possible to obtain metrics of the health of the microservices. A challenge is for companies to train their employees so that they can automate most processes with DevOps tools to perform, for example, Continuous Integration, Continuous Deployment, shortening development, testing, and delivery times to different environments such as pre-production and production [36].

2) CHANGES WITHOUT AFFECTING OTHER COMPONENTS

Since a microservice is a small component that can run independently, it is easy to maintain it, while in case of a bug, it is easier to navigate through the source code, find the bug and fix it, thus facilitating its maintainability, as well as ease of releasing new versions [1].

3) AUDIT AND MONITORING

Having individual microservices allows us to place security on specific microservices so that they can be audited, monitored. Making changes to specific microservices to incorporate security mechanisms makes it possible to optimize costs [36].

4) EASE OF FUNCTIONAL TESTING AND ERROR IDENTIFICATION

Because microservices are independent, they do not impact each other when they are built with a good microservices architecture. This means that if one part fails, the entire application does not crash [45]. In addition to using DevOps practices, they have focused on improving resilience through automated testing. By making tests part of the build process, tests are constantly run against code uploaded to a repository, increasing the chances of finding and fixing bugs in the code. Some attempts to improve resilience are to deploy new versions in a small group of users and, if everything goes well for a period of time, the new version is deployed to all users; otherwise, it will return to the previous version until the problems are resolved.

5) SELF-MANAGED TEAMS

Separating a monolith application into microservices brings us advantages such as creating work teams that take one or more microservices and take charge of them (take full ownership), which gives us benefits such as modifications or evolution of microservices independently and autonomously, work teams can be in different geographic locations and can work independently on these microservices. Having

small rather than large work teams also brings advantages such as faster and more frequent development and delivery times because work teams specialize in knowing in detail a specific business functionality that they are implementing in microservices and with each iteration development, deepen their knowledge further, in addition to the experience acquired, these small work teams can give more accurate development estimates, in addition to being able to perform faster developments [36]. Continuous integration tools bring us advantages such as obtaining metrics on how the quality of our code is, shortening delivery times, merging our code with the rest of the development team's code in a controlled way, having version control, traceability of the artifacts delivered [36].

A microservice being independent, the software development team can develop a specific microservice in another programming language to take advantage of algorithms or libraries that are incorporated, thus having other alternatives to innovate, optimize costs and development times.

A microservices architecture supports project managers, architects, and programmers to correctly build microservices, have the proper guidelines, carry out a correct design, coding [1]. The key is to have small and independent services [2]

C. DISADVANTAGES WITH MONOLITHS

Several disadvantages related to a monolithic architecture have been identified, such as operating costs, complexity and scalability costs, difficulty incorporating modern technologies or using outdated technology, difficulty in hiring adequate personnel to maintain these monolithic applications. Below we expose the disadvantages found in more detail.

1) HIGH MAINTENANCE COST

A problem with monolithic applications is the excessive cost of source code maintenance, especially when an application is large and complex. In monolithic systems, given their design, it can be challenging to maintain. It is difficult to identify an error in monolith applications, correct it, and publish the solution in a test and production environment.

Among the disadvantages identified is how difficult it is to maintain a monolith, since, being a single block, in case an internal component is modified and a failure occurs in this maintenance process, the entire monolith application can reach fail, you also have the problem of introducing new functionality at the same time you can incorporate new errors (bugs), and at the same time, it is required to test all the functionality of the application again [46]. Any change in a module in the monolith application requires a complete restart of the application, which requires time to perform the deployment of the updated version of the application [1].

2) SCALABILITY

Another drawback is in the scalability of a monolithic application, since while it can be scaled through virtual servers and a load balancer, this demands more hardware and software resources, licensing costs, and time, therefore an alternative to using containers with microservices is a good alternative [47]. The disadvantage of a monolith application is that, for example, if a module is the most used, it is not possible to scale only this module, but the entire monolith must be scaled, consequently increasing costs [1].

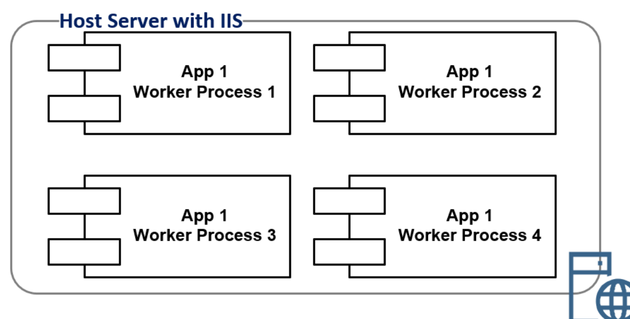

FIGURE 11. Monolithic scalability duplicating worker process.

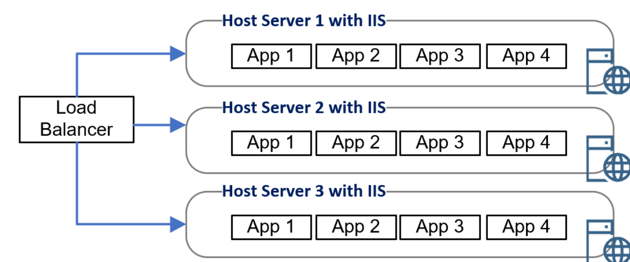

FIGURE 12. Monolithic scalability in servers.

The scalability in applications at the physical or virtual server level with technology can be done in two ways:

- Via WebGarden
- Via WebFarm

For example, in web applications with Microsoft technologies, which are deployed in the Internet Information Server (IIS), the webserver and its application configuration files can be configured so that several processes of the same application are created. This is known as WebGarden or horizontal scaling. An example of this is shown in Figure 11.

Another alternative is to deploy a web application on a virtual or physical server farm. In order to have high availability, a load balancer can be configured so that the application is always available at all times. This scaling is known as Web Farm or vertical scaling. An example of how this scheme works is indicated in the following Figure 12.

Finally, given the technological advances, a current trend is scalability using containers with Docker technology, as shown in the following Figure 13.

This scheme has the advantage that new instances of the monolith application can be created in a matter of seconds. In addition to having advantages of an application in a container, given that by the nature of a container,

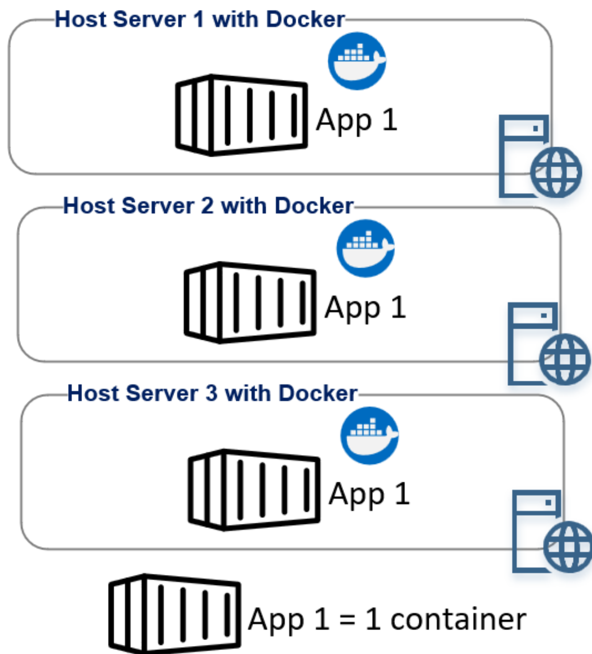

FIGURE 13. Monolithic Scalability in containers.

it is immutable, isolated, and does not depend on any configuration of the operating system. The use of containers brings multiple advantages such as provisioning in seconds, automated service monitoring and restoration, version updates in seconds, all these using container management tools that exist in the market. The disadvantage of scaling a monolith is that the entire monolith application is scaled, regardless of when only a part of the functionality is the most used.

3) LOK-IN LANGUAGE

Another drawback of monolithic applications is that they are generally locked (lock-in) to be programmed with a single programming language [1] while the flexibility of microservices is that each microservice can be programmed with a different programming language [2]. However, in large software factories, despite having this flexibility of each microservice, program it in any programming language, due to standardization, training for programmers, learning curve, for ease of maintenance to correct errors or add new functionality, it is recommended to standardize and use a single programming language in the development of large and complex software [1].

D. DISADVANTAGE WITH MICROSERVICES

When creating an application under the microservices architecture, a drawback is the high learning curve of this new architecture, which goes hand in hand with an organizational change to form autonomous teams that create the new microservices, possibly in a new programming language already while using new tools to automate compilation, deployment, and monitoring, and there is a high possibility that training will be needed to understand how to deploy containerized microservices to cloud computing. However, once this learning curve is overcome, developing microservices will be much faster than working in a monolithic application.

1) LACK OF EXPERIENCE FROM THE DEVELOPMENT TEAM

Another disadvantage is that an organization when it begins the incorporation of a new architecture, such as microservices, does not know if they will be successful due to factors such as the limited experience of the development team in working with this new architecture, where for example Buchgeher et al. [10] in a Case Study indicates that the migration of a project took eight months to overcome the learning curve to work with microservices. For a small company it is easier to organize and form work teams based on products, not roles, while in large organizations due to the size of their development team and the silos that are formed between areas and their billing scheme for their services provided organizational change is more complex.

2) NETWORK OVERHEAD

Microservices are characterized by making more calls to implemented functions, causing communication delays between microservices and network overheads, which leads to analyzing the network architecture and possibly implementing changes at the architecture level. Additionally, optimizations should be considered at the programming level to be able to make asynchronous calls, parallel processing to balance the network overheads.

The increase in microservices, even though they are in containers, can lead to an increase in infrastructure given the rise in the number of microservices, which is costly to maintain.

3) DATA DUPLICATION

Since a microservices architecture recommends that each microservice have its own database, it will be necessary to duplicate data in the different new databases created, which will require additional programming mechanisms to guarantee the consistency of the data, increasing the complexity due to some services will be more difficult to orchestrate, given the increasing number of microservices [48]. New services should be developed as compensatory (reverse) mechanisms since the use of a distributed transaction system is not recommended.

Since each microservice isolates unique business functionality, testing an entire application increases complexity, and it will take longer to find and fix bugs with the new microservices architecture. Working with microservices without automation would be a severe mistake, so using DevOps tools to automate code integration, compilation, and deployments is mandatory.

4) INCREASE IN COMPLEXITY

Having many microservices and consuming them from a Front End when trying to identify an error through traceability can be complex and require many hours of searching for the bug, which is why it can also be expensive [6], [45].

a: SUMMARY

The following Table 4 shows a summary of the advantages and disadvantages in the monolithic and microservice architectures found.

b: DISCUSSION

Once the advantages and disadvantages of monolithic architectures and microservices were exposed, the second research question was answered *What are the advantages and disadvantages of monolithic architectures and microservices?*, Obtaining a list of advantages, disadvantages that they are in monolithic and microservice architectures. From this study, it can be found that monolithic architecture presents several problems that can be covered by microservice architectures, presenting greater advantages compared to monolithic architectures. Among the main advantages and disadvantages that both architectures have, we can highlight that a monolithic architecture is not a bad choice when you have a small application. However, if an application over the years has grown in complexity and the development team spend more time fixing bugs than incorporating new business functionality, it is necessary to make a change towards a microservices architecture. Making this architecture change not only involves breaking an application into small components but rather relying on the principles of software engineering and the principles and patterns related to microservices to perform a good decomposition of a monolithic application towards a microservices architecture. Evolving to this architecture will bring advantages in the short and long term, such as being able to use state-of-the-art technology, make use of cloud computing with optimized costs, make deliveries in shorter times, evolving and replacement of microservices with minimal impact on the entire application.

V. MICROSERVICES NOWADAYS

Organizations are showing interest in modernizing their applications towards microservices to have the benefits that this architecture offers: freedom and flexibility since they can be written in any programming language and positively influencing the performance of an application easy to develop to meet business needs, scalability on demand. Many organizations are currently migrating their applications to microservices, while others have migrations to microservices in their roadmap in the short term.

Francesco et al. [29] has carried out a study where it mentions that in 2015 show 23 publications related to microservices were identified, while in 2016, this number doubled, reaching 41, and to date, this number continues to increase. This report also indicates that the studies carried out by the academy regarding microservices represent four times more than the studies carried out by the industry. While organizations have increased experiences of migrations to microservices, it is expected to have more reference information on how to carry out this process successfully, as there are still many challenges to address.

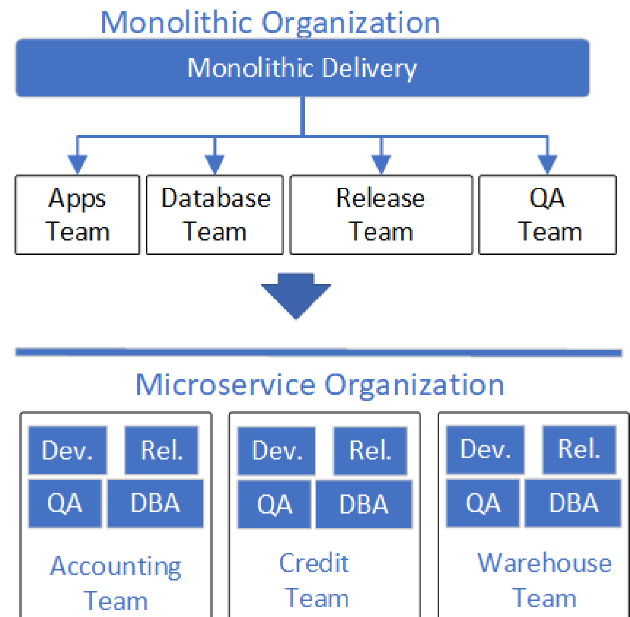

FIGURE 14. Monolithic organization vs Microservices organization.

In the case of small organizations, there is a detailed study carried out by Buchgeher [10] who indicates a Case Study in a small software development company and shows the experience of migrating a monolithic sample analysis application from labs to a microservices architecture. This study is relevant because they carry out the migration based on a list of microservices principles and report their experience. It should be noted that in order to be successful in this migration, they had to make adaptations in their organizational structure and assume the high initial person-hour costs in creating the first microservices due to the learning of this new microservices architecture, in addition to learning to incorporate new technologies such as the use of DevOps automation tools and microservices monitoring tools. An example of how a small organization working with a monolithic architecture style would be restructured towards microservices is seen in Figure 14.

In [9] they report another experience of migration of a monolithic application called SSaaS towards a microservice architecture and its deployment of cloud computing. In this case, the motivator for carrying out this migration was that when they tried to incorporate a new chat module in the application and they realized that this service had to be reusable, with automated deployment and with scalability capabilities. They realized that performing a refactoring of your current application to a microservices architecture could

TABLE 4. Comparative table of advantages and disadvantages between monoliths and microservices architectures.

Architecture	Advantages	Disadvantages
Monolith	Recommended for small applications [43]	Difficult to maintain [1], [43]
Monolith	It is easy to develop	Expensive to maintain [43]
Monolith	Ease of deployment	Difficulty understanding the business logic to make changes
Monolith	Flexibility to work alone or on small projects	Increased complexity by adding new features [1]
Monolith	Easier to debug	Costly when scaling in infrastructure [1]
Monolith		Source code developed with a single technology / language [1]
Monolith		In case of an error or bug, the entire application crashes [43]
Monolith		It is more difficult to perform functionality tests [1], [43]
Monolith		In case of any changes, it is required to test all the functionality of the application again [1], [49]
Microservice	Microservices are the solutions to managing complex software systems [43]	Initially, little knowledge and experience of the development team to create microservices [49]
Microservice	Understandability [1], [49]	Problems trying to split a monolith application [49]
Microservice	Easy to deploy (using DevOps tools) [1], [43]	More chatty interactions between function calls between microservices which causes communication delay between microservices [1], [27], [43]
Microservice	High Availability	Infrastructure increase [1]
Microservice	Independent scaling using containers [1], [27], [43]	Data duplication due to microservices architecture [48]
Microservice	Ease of testing for business functionalities [27], [43], [49]	Difficulty in testing and debugging a distributed application [43]
Microservice	Maintainable [1]	Expensive to maintain [43]
Microservice	Fault tolerance and better fault isolation	Difficult to orchestrate, given the incremental number of microservices [1], [43]
Microservice	Independent deployment DURS (Deployed, Updated, Replaced and Scaled) [1], [27], [49]	Complexity in the use of different technologies [1]
Microservice	Resilient [45]	Microservices are exposed to more potential attacks [1]
Microservice	Save cost to enterprise [43]	Difficult to debug and perform a full test [49]
Microservice	Application Security [1]	
Microservice	Rapid and Shorter deployment cycles, speed of development / Teamwork /work in Small Modules [1], [43], [49]	
Microservice	Easier innovation and replacement of microservices [1], [43]	
Microservice	Small, independent services [1]	
Microservice	Reuse of microservices [43]	

meet this requirement. As a report of their experience, they highlight that it is advisable to carry out an incremental migration, rely on mechanisms to automate the delivery process using continuous integration, use Containers for deployment. In addition, in [9] they indicate that the migration from a monolithic architecture to A microservices architecture entails new complexities such as handling new technology, standardizing the use of different programming languages, reusing components such as Netflix OSS (Open-Source Software) that already exist so as not to start from scratch.

On the other hand, Bucciarone et al. [50] reports the experience of the migration of a banking application that had a monolithic architecture towards a microservices architecture, where they indicate that in this migration process, they did it by implementing one functionality at a time. The identification of the business functionalities was carried out through conversations with the business experts; in this way, the business functionalities that should be isolated in each service were easily identified, although at the beginning, some of the microservices they created had more than one functionality, however as more knowledge of the business was acquired they were able to divide and isolate each business functionality in each microservice. In this experience, he also mentions the use of recent technologies such as the use of containers, automation tools for integration, and continuous deployment, so when there is a more significant number of microservices, the time that it takes to compile is transparent and faster. In this paper, they report that having a monolithic application with large

components, it is difficult to separate into small modules due to high coupling between components and overlapping responsibilities. They realized that the first microservices were well defined when they realized that they had loose coupling. Likewise, the migration result revealed that the architecture was simplified since each component had a clear and well-defined responsibility.

There is another experience report [51] of the migration of monolithic applications to microservices carried out in 3 public institutions. This report indicates the benefits of migrating a monolithic application to microservices. Among the motivators that led to the migration to microservices is the lack of freedom of development teams when incorporating new technologies, duplication of code by copying rigid components. To carry out this process, they first carried out a self-observation and then a survey with closed questions whose purpose was to understand the consequences of the migration to microservices. Among the advantages obtained, the development teams have now adopted DevOps for continuous delivery, carrying out several deployments in one day from what they previously carried out one deployment per week. Another advantage obtained was the improvement in the motivation of development teams since, with this adoption of microservices, development teams could now organize themselves into groups to do independent development of groups of microservices, have the freedom to use new tools, and technologies. The deployment could also be done independently for each microservice. Scalability on demand independently was another benefit, and finally, another benefit was being able to implement and deliver small pieces of software (microservices) in short periods of time.

A. SUCCESSFUL MICROSERVICES IMPLEMENTATIONS

The success of a microservices architecture is reflected in the implementations carried out by large companies such as Amazon, Netflix, LinkedIn, SoundCloud, Spotify [3], [4], [5].

For example, the company Spotify, in a conference in 2005, indicated how they create microservices and demonstrate the benefits of using microservices instead of monolithic applications. With the microservices architecture style, they have also managed to have fewer dependencies between the components of their product. In addition, they mention that in their organization, they have formed several autonomous teams which are free to create and maintain microservices; each team owns the microservice [52].

Netflix, another pioneer in adopting microservices, which was initially a monolithic application for DVD movie rental, reinvented itself to form small work teams responsible for creating specific microservices. As Netflix engineers transformed their monolithic application towards a microservices architecture, they established several best practices for designing and implementing microservices, such as that each service should have its own database repository for which they had to denormalize its database. If a microservice was too large, it had to be divided into several microservices and use containers. Finally, he released many of these tools created to create microservices as Open Source Software [53].

In the case of Amazon, it was in 2002 that it reinvented itself because by mandate, all work teams had to expose their data and business functionality through services, and this was the only form of communication, reaching an SOA architecture and giving freedom to the tools with which they would create the services. It can be said that later these services continued their evolution towards microservices. The transformation of Amazon was that from what it was a platform to sell and send books to become one of the first platforms for the sale of any type of article on the internet, in addition to offering an entire infrastructure in the cloud so that any company can reuse its infrastructure to deploy microservices in the Amazon cloud [54].

SoundCloud is another success story. SoundCloud is a product by which artists can share their work, collaborate on the creation of tracks, and be discovered by the industry. This application was born with a Monolithic architecture. However, they realized some drawbacks such as scalability problems and that they were spending too many time-solving errors instead of incorporating new functionality, so they decided to adopt a microservices architecture, following an incremental transformation process, that is, as they needed to refactor or add new functionality, instead of doing it in the monolith system, they created a new microservice, and thus incrementally eliminated functions from the main monolith application. The main benefit they noticed from using a microservices architecture is that they were able to create and release new functionality in short cycles [5].

B. CHALLENGES

The crucial part of migrating monolithic applications to microservices is understanding the challenges that arise. The most relevant are detailed below.

- Understanding of business logic
- Understanding the current (as-is) and future architecture (to-be)
- Understanding only the required business functionality
- Cost optimization
- Rapid reaction to business demand (elasticity)

Some companies, as part of the evolution of their monolithic applications, are working on migrating them towards an SOA architecture and others towards a microservices architecture; however, one of the biggest and most complex challenges that programmers face is breaking down a monolith system into small ones. and independent modules [6], [28].

However, decomposing a monolith system into microservices also brings performance-related challenges, given the inter-microservice calls [28]. Another challenge is that there are still no tools that allow decomposition, and this task can be done manually [6], [55]. The tools that exist on the market

only allow for static code analysis and dependency scanning, so proposals for decomposition process frameworks have emerged in the literature proposals [28], [56].

One of the challenges for which many companies adopt the new microservices architecture is to generate Fast Time-to-Marquet; that is, it is possible to incorporate new functionalities requested by clients in the applications and deploy them in a production environment as soon as possible [10]. In the literature, there are challenges that new developers must face since the high learning curve must be solved at the beginning of the adoption of this new architecture [10]. Another challenge is creating applications that are multitenancy. Multitenancy is used, for example, in an application like SaaS (Software as a Service) when the application-level model allows multiple instances of an application to be shared by multiple tenants through configurations, and only the tenant has access to their only information [7]. Through our research, we provide some advices and perspectives to consider when undertaking migrations to microservices:

- Using a domain-driven design is crucial in order to define the boundaries and responsibilities of each created microservice.
- The use of microservices is ideal when a high level of scalability is required, especially in IoT systems where scalability is needed to handle a large number of interconnected devices, and also to mitigate security risks and attacks associated with IoT data.
- As the number of microservices increases, it is necessary to consider implementing a robust testing process supported by automated testing tools.
- Finally, it is highly recommended to have a solid monitoring process for deployed services, allowing operators to oversee the health status of the microservices.

C. FUTURE DIRECTIONS

Microservices are now a popular architectural style for building any kind on applications in different programming languages. They offer a number of benefits, such as scalability, flexibility, and maintainability. Due to the great current technological changes, we can predict what or what is coming in the future about microservices:

- Micro frontends: Micro frontends represent the future of frontend development and in the next years, more developers will use micro frontends. This means that more and more companies are adopting this architecture because off as microservices architectures become more complex, micro frontends can help to improve the maintainability and scalability of these applications. By dividing large-scale web applications into smaller, independent frontends now is called micro frontends. Each micro frontend, which may be designed and deployed independently, is in charge of a certain aspect of the user interface. Because individual micro frontends may be changed without affecting the entire application, this makes it simpler to manage and expand large-scale online applications, obtaining the same benefits of microservices like improved scalability, maintainability, flexibility. Micro frontends can be integrated with other technologies, such as microservices, serverless computing, and artificial intelligence. This will allow developers to build even more powerful and sophisticated applications [57], [58], [59].
- Containerization evolution: Microservices can run in lightweight, isolated environments thanks to containers. For managing and scaling microservices installations, container orchestration solutions like Kubernetes have seen a considerable increase in popularity. The deployment and maintenance of microservices should become easier in the future because to developments in containerization technology and orchestration tools [30], [60].
- Serverless Computing and Cloud-native architecture: Serverless computing and microservices are likely to be progressively integrated, opening the door to deployment approaches that are more effective and scalable. Developers may concentrate entirely on building code for individual microservices thanks to serverless architectures, which abstract away infrastructure-related concerns. The future of microservices with cloud-native architecture appear to have a highly promising future. Both microservices and cloud-native architecture are growing in popularity, and they offer a variety of advantages that can help organizations to build more scalable, flexible, and maintainable applications [30], [61].
- IoT and Edge Computing: IoT and microservices both have a bright future. Microservices will become a more crucial architectural pattern for developing IoT applications as the IoT expands. Scalable, adaptable, and secure IoT applications can be created using microservices. Edge computing will be essential in processing data closer to the source as IoT devices keep multiplying. By deploying microservices at the edge, it is possible to provide distributed processing and lower latency. With this strategy, reliance on centralized cloud resources can be reduced and reaction times can be increased [62], [63], [64].
- AI-Enabled Microservices: The future of microservices in relation to AI (Artificial Intelligence) holds significant potential. Microservices that are AI-enabled can use AI's capabilities to improve their functionality. To enable intelligent decision-making, predictive analytics, natural language processing, computer vision, and other AI-driven tasks, AI algorithms can be integrated into microservices. Microservices may have access to cutting-edge capabilities through this integration, enabling them to offer services that are intelligent and adaptive [65], [66], [67].

VI. CONCLUSION AND FUTURE WORK

This survey provides a detailed explanation of monolithic and microservice architectures, as well as clarifies the different

foundations on which software engineering is based, highlighting important concepts such as abstraction, inheritance, polymorphism, encapsulation, information hiding and how these concepts are related to monolithic and microservice architectures. We also show that there are patterns that can be used in the design and development of microservices.

We conclude that it is important that microservices are created using domain-driven design, so that can be easily replaceable, that hide the details of their implementation and their use is only through smart endpoints, that each microservice has one and only one business responsibility, they are formed Autonomous development teams, which also involves a change in the organizational structure in companies, it is also recommended that both the compilation and deployment of the microservices created to be done in an automated way and also when the microservices are created, they must have logs, in such a way that through monitoring tools the correct operation of the microservices can be reviewed.

This work can be helpful to those researchers and professionals looking to carry out migration processes from architecture from monoliths to microservices since the theoretical foundations presented here can guide them to carry out successful migrations. Finally, as future work, it is proposed to experiment with the migration of a monolith application towards a microservices architecture, applying the principles and patterns identified here.

REFERENCES

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017, pp. 195–216.

[2] J. Lewis and M. Fowler, "Microservices: A definition of this new architectural term," MartinFowler.com, 2014, vol. 25, nos. 14–26, p. 12.

[3] J. Thönes, "Microservices," *IEEE Softw.*, vol. 32, no. 1, p. 116, Jan. 2015.

[4] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. 10th Comput. Colombian Conf. (CCC)*, Sep. 2015, pp. 583–590.

[5] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Proc. Int. Conf. Web Eng.* Cham, Switzerland: Springer, 2017, pp. 32–47.

[6] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, Sep. 2017.

[7] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency," *IEEE Softw.*, vol. 35, no. 3, pp. 63–72, May 2018.

[8] A. Carrasco, B. V. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proc. 2nd Int. Workshop Refactoring*, Sep. 2018, pp. 1–6.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: An experience report," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.* Cham, Switzerland: Springer, 2015, pp. 201–215.

[10] G. Buchgeher, M. Winterer, R. Weinreich, J. Luger, R. Wingelhofer, and M. Aistleitner, "Microservices in a small development organization," in *Proc. Eur. Conf. Softw. Archit.* Cham, Switzerland: Springer, 2017, pp. 208–215.

[11] O. Zimmermann, "Microservices tenets," *Comput. Sci. Res. Develop.*, vol. 32, nos. 3–4, pp. 301–310, Jul. 2017.

[12] R. S. Pressman, *Software Engineering: A Practitioner'S Approach*. London, U.K.: Palgrave Macmillan, 2005.

[13] I. Sommerville, *Software Engineering*, 9th ed. India: Pearson Education, 2011.

[14] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *Proc. 38th Int. Conf. Chilean Comput. Sci. Soc. (SCCC)*, Nov. 2019, pp. 1–7.

[15] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," in *Proc. 23rd Int. Conf. Autom. Comput. (ICAC)*, Sep. 2017, pp. 1–6.

[16] D. Kasture and R. C. Jaiswal, "Pillars of object oriented system," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 7, no. 12, pp. 589–590, 2019.

[17] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[18] G. Booch, R. A. Maksimchuk, and M. W. Engle, "Object-oriented analysis and design with applications," *ACM SIGSOFT Softw. Eng. Notes*, vol. 33, no. 5, p. 29, 2008.

[19] K. Kwon and J. Cheon, "Exploring problem decomposition and program development through block-based programs," *Int. J. Comput. Sci. Educ. Schools*, vol. 3, no. 1, pp. 3–16, Apr. 2019.

[20] H. Van Vliet, H. Van Vliet, and J. Van Vliet, *Software Engineering: Principles and Practice*, vol. 13. Princeton, NJ, USA: Citeseer, 2008.

[21] I. Hadar and E. Hadar, "An iterative methodology for teaching object oriented concepts," *Informat. Educ.*, vol. 6, no. 1, pp. 67–80, Apr. 2007.

[22] J. Kramer, "Is abstraction the key to computing?" *Commun. ACM*, vol. 50, no. 4, pp. 36–42, Apr. 2007.

[23] G. Booch, *Object Oriented Analysis & Design With Application*. India: Pearson Education, 2006.

[24] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[25] S. Kendal, *Object Oriented Programming Using C*. London, U.K.: BookBoon, 2011.

[26] V. Velepucha, P. Flores, and J. Torres, "MOMMIV: Modelo para descomposición de una arquitectura monolítica hacia una arquitectura de microservicios bajo el principio de ocultación de información," *Revista Ibérica de Sistemas e Tecnologias de Informação*, vol. 1, no. E17, pp. 1000–1009, 2019.

[27] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Sebastopol, CA, USA: O'Reilly Media, 2019.

[28] D. Taibi and K. Systä, "A decomposition and metric-based evaluation framework for microservices," 2019, *arXiv:1908.08513*.

[29] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2017, pp. 21–30.

[30] F. Wang and J. Zhang, "Research on the current situation and future trend of microservice technology development," in *Proc. IEEE 6th Adv. Inf. Technol., Electron. Autom. Control Conf. (IAEAC)*, Nov. 2022, pp. 44–54.

[31] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," in *Proc. IEEE Symp. Service-Oriented Syst. Eng. (SOSE)*, Mar. 2018, pp. 11–20.

[32] U. Zdun, E. Wittern, and P. Leitner, "Emerging trends, challenges, and experiences in DevOps and microservice APIs," *IEEE Softw.*, vol. 37, no. 1, pp. 87–91, Jan. 2020.

[33] V. Vernon, *Domain-Driven Design Distilled*. Reading, MA, USA: Addison-Wesley Professional, 2016.

[34] M. Richards, *Fundamentals of Software Architecture*. Sebastopol, CA, USA: O'Reilly Media, 2020.

[35] A. Kirk, *Data Visualisation: A Handbook for Data Driven Design*. Newbury Park, CA, USA: Sage, 2016.

[36] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[37] C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY, USA: Manning Publications, 2019.

[38] S. D. Santis, L. Florez, D. V. Nguyen, and E. Rosa, *Evolve the Monolith to Microservices With Java and Node*. Indianapolis, IN, USA: IBM Redbooks, 2016.

[39] V. F. Pacheco, *Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Birmingham, U.K.: Packt, 2018.

[40] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Reading, MA, USA: Addison-Wesley Professional, 2004.

[41] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May 2018.

[42] E. Wolff, *Microservices: Flexible Software Architecture*. Reading, MA, USA: Addison-Wesley Professional, 2016.

[43] A. Singleton, "The economics of microservices," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 16–20, Sep. 2016.
[44] L. Chen, "Microservices: Architecting for continuous delivery and DevOps," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2018, pp. 39–397.
[45] T. Prasandy, D. F. Murad, and T. Darwis, "Migrating application from monolith to microservices," in *Proc. Int. Conf. Inf. Manage. Technol. (ICIMTech)*, Aug. 2020, pp. 726–731.
[46] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018.
[47] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *Int. J. Open Inf. Technol.*, vol. 2, no. 9, pp. 24–27, 2014.
[48] W. Fan, Z. Han, Y. Zhang, and R. Wang, "Method of maintaining data consistency in microservice architecture," in *Proc. IEEE IEEE 4th Int. Conf. Big Data Secur. Cloud (BigDataSecurity) Int. Conf. High Perform. Smart Comput., (HPSC) IEEE Int. Conf. Intell. Data Secur. (IDS)*, May 2018, pp. 47–50.
[49] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, "Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages," in *Proc. XP Scientific Workshops*, May 2017, p. 23.
[50] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Softw.*, vol. 35, no. 3, pp. 50–55, May 2018.
[51] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, "An experience report on the adoption of microservices in three Brazilian government institutions," in *Proc. 32nd Brazilian Symp. Softw. Eng.*, Sep. 2018, pp. 32–41.
[52] B. Linders, "Microservices at spotify," 2015, vol. 11, p. 18. [Online]. Available: https://www.infoq.com/news/2015/12/microservices-spotify/
[53] T. Mauro. (2015). *Adopting Microservices at Netflix: Lessons for Architectural Design*. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices
[54] B. Smith and G. Linden, "Two decades of recommender systems at amazon.com," *IEEE Internet Comput.*, vol. 21, no. 3, pp. 12–18, 2017.
[55] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," in *Proc. 8th Int. Conf. Cloud Comput. Services Sci.*, 2018, pp. 221–232.
[56] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," in *Proc. 9th Int. Conf. Cloud Comput. Services Sci.*, 2019, pp. 1–12.
[57] M. Geers, *Micro Frontends in Sction*. New York, NY, USA: Simon & Schuster, 2020.
[58] L. Mezzalira, *Building Micro-Frontends*. Sebastopol, CA, USA: O'Reilly Media, 2021.
[59] P. Y. Tilak, V. Yadav, S. D. Dharmendra, and N. Bolloju, "A platform for enhancing application developer productivity using microservices and micro-frontends," in *Proc. IEEE-HYDCON*, Sep. 2020, pp. 1–4.
[60] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, "Microservices: Architecture, container, and challenges," in *Proc. IEEE 20th Int. Conf. Softw. Quality Rel. Secur. Companion (QRS-C)*, Dec. 2020, pp. 629–635.
[61] P. Raj, S. Vanga, and A. Chaudhary, *Cloud-Native Computing: How to Design, Develop, and Secure Microservices and Event-Driven Applications*. Hoboken, NJ, USA: Wiley, 2022.
[62] A. Razzaq, "A systematic review on software architectures for IoT systems and future direction to the adoption of microservices architecture," *Social Netw. Comput. Sci.*, vol. 1, no. 6, p. 350, Nov. 2020.
[63] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based iot applications scheduling in edge and fog computing: A taxonomy and future directions," 2022, *arXiv:2207.05399*.
[64] S. Pallewatta, V. Kostakos, and R. Buyya, "Placement of microservices-based IoT applications in fog computing: A taxonomy and future directions," *ACM Comput. Surveys*, vol. 55, no. 14s, pp. 1–43, Dec. 2023.
[65] G. M. Lee, T.-W. Um, and J. K. Choi, "AI as a microservice (AIMS) over 5G networks," in *Proc. ITU Kaleidoscope, Mach. Learn. 5G Future (ITU K)*, Nov. 2018, pp. 1–7.
[66] S. Alrubei, E. Ball, and J. Rigelsford, "A secure distributed blockchain platform for use in AI-enabled IoT applications," in *Proc. IEEE Cloud Summit*, Oct. 2020, pp. 85–90.
[67] F. Al-Doghman, N. Moustafa, I. Khalil, N. Sohrabi, Z. Tari, and A. Y. Zomaya, "AI-enabled secure microservices in edge computing: Opportunities and challenges," *IEEE Trans. Services Comput.*, vol. 16, no. 2, pp. 1485–1504, Mar. 2023.

VICTOR VELEPUCHA was born in Ecuador, in 1976. He received the degree in computer systems from Escuela Politécnica Nacional (EPN), Quito, Ecuador, in 2003, and the master's degree in project management from Universidad de las Fuerzas Armadas (ESPE), in 2012. He is currently pursuing the Ph.D. degree in informatic systems.

Since 2003, he has been working in the systems engineering area as a programmer, a software architect, and a project manager, mainly in the banking and finance enterprises. His research interests include software development with different technologies, study about different architectures styles, cloud computing, continuous delivery, DevOps, and microservices.

PAMELA FLORES received the Engineering degree in computer systems from Escuela Politécnica Nacional (EPN), in 2005, and the master's degree in information technologies and the Ph.D. degree in software and systems from Universidad Politécnica de Madrid (UPM), in 2011 and 2016, respectively. She is currently a Professor with EPN. She also coordinated the Ph.D. degree in informatics for three years, and she also coordinates the master's degree in software with EPN. Her research area is related with object-oriented approach; she has also worked on qualitative research in computer science.