# Architectural Patterns for Microservices: a Systematic Mapping Study

Davide Taibi[1] and Valentina Lenarduzzi [1] and Claus Pahl[2]
[1]*Tampere University of Technology, Tampere, Finland*
[2]*Free University of Bozen-Bolzano, Bozen-Bolzano, Italy*
{*davide.taibi,valentina.lenarduzzi*}*@tut.fi, claus.pahl@unibz.it*

Keywords: Microservices, Architectural Style, Architecture Pattern, Cloud Native, Cloud Migration, DevOps.

Abstract: Microservices is an architectural style increasing in popularity. However, there is still a lack of understanding how to adopt a microservice-based architectural style. We aim at characterizing different microservice architectural style patterns and the principles that guide their definition. We conducted a systematic mapping study in order to identify reported usage of microservices and based on these use cases extract common patterns and principles. We present two key contributions. Firstly, we identified several agreed microservice architecture patterns that seem widely adopted and reported in the case studies identified. Secondly, we presented these as a catalogue in a common template format including a summary of the advantages, disadvantages, and lessons learned for each pattern from the case studies. We can conclude that different architecture patterns emerge for different migration, orchestration, storage and deployment settings for a set of agreed principles.

## 1 Introduction

Microservices are increasing their popularity in industry, being adopted by several big players such as Netflix, Spotify, Amazon and many others and several companies are now following the trend, migrating their systems to microservices. However companies still have the issues of selecting the most appropriate architectural patterns, mainly because of the lack of knowledge about the available patterns (Taibi et al., 2017).

Microservices are small autonomous services deployed independently, with a single and clearly defined purpose (Lewis and Fowler, 2014). Their independent deployability is advantageous for continuous delivery. They can scale independently from other services, and they can be deployed on the hardware that best suits their needs. Moreover, because of their size, they are easier to maintain and more fault-tolerant since the failure of one service will not break the whole system, which could happen in a monolithic system (Lewis and Fowler, 2014). Microservices have emerged as a variant of service-oriented architecture (SOA). Their aim is to structure software systems as sets of small services that are deployable on a different platform and running in their own process while communicating with each other through lightweight mechanisms without a need for centralized control (Pahl et al., 2018). We consider an *architectural style* here as a set of *principles* and coarse-grained *patterns* that provide an abstract framework for a family of systems. An architectural style consists of a set of architectural principles and patterns that are aligned with each other to make designs recognizable and design activities repeatable: principles express architectural design intent; patterns adhere to the principles and are commonly occurring (proven) in practice (Zimmermann, 2009).

Microservices enable continuous development and delivery (Pahl et al., 2018). DevOps is the integration of *Dev*elopment and *Op*erations that include a set of continuous delivery practices aimed at decrease the delivery time, increasing the delivery efficiency and reducing time among releases while maintaining software quality. It combines software development, quality assurance, and operations (Bass et al., 2015).

Despite both microservices and DevOps being widely used, there are still challenges in understanding how to construct such kinds of architectures (Balalaie et al., 2016), (Taibi and Lenarduzzi, 2018) and developers often adopt the patterns where they can find more documentation online (Taibi et al., 2017), even if they are not the most appropriate ones. In order to help developers to identify the most appropriate patterns, we aim here to identify and characterize different microservice architecture patterns reported in the literature, be that as proposals and or as case studies with implementations. The identification process is supported by a systematic mapping study (Petersen et al., 2008). A previous systematic mapping (Pahl and Jamshidi, 2016) has aimed at classifying and comparing the existing research body on microservices including non peer-reviewed content, while a recent work with a similar goal has been published by (Malavolta and Lago, 2017). Our study differs in the following ways:

- *Focus*: We focus on suggested architectural style definitions, emerging patterns, while (Pahl and Jamshidi, 2016) initially characterized the body of research and (Malavolta and Lago, 2017) focused on the potential for industrial adoption. The central new aspect here is on an exploration of architecture patterns for a set of agreed principles.
- *Systematic Approach and Comprehensiveness*: We conducted a systematic mapping study implementing the protocol (Petersen et al., 2008), followed by a systematic snowballing process, including all references found in the papers. We included results from 8 bibliographic sources.

The contribution of our study is as follows: We identify and describe microservice architecture patterns as a pattern catalogue, thus analysing advantages and disadvantages of the different patterns based on their implementations. These patterns aim to support developers in finding suitable solution templates for architecting microservices-based software. As this is a still developing domain, we also discuss our findings in the context of wider research gaps and trends. Furthermore, we specifically address cloud computing as the key application environment.

The paper is structured as follows. Section 2 describes our pattern identification methodology. Section 3 shows the pattern catalogue, which is discussed in Section 4. Section 5 contains our conclusions.

## 2 Pattern Identification Process

We used a literature review protocol (Petersen et al., 2008) in combination with a systematic snowballing process to detemine the patterns from the published literature (Wohlin, 2014). We define our research goal as follows: *Analyze* the architectural style proposals *for the purpose of* comparing them and related implementations *with respect to* their advantages and disadvantages *in the context of* cloud-native software implementation. We derived two research questions:

- RQ1: Which are the different microservices-based architecture patterns?
- RQ2: Which advantages and disadvantages have been highlighted for these patterns?

We used eight **bibliographic sources**: ACM Digital Library, IEEE Xplore Digital Library, Science Direct, Scopus, Google Scholar, Citeeser library, Inspec and Springer Link. We defined the **search strings** based on the PICO terms of our questions (Kitchenham and Charters, 2007) using only the terms Population and Intervention. We applied the following query for the discovery of pattern descriptions:

(microservice* OR micro-service*) AND (architect* OR migrat* OR modern* OR reengineer* OR re-engineer* OR refactor* OR re-factor* OR rearchitect* OR re-architect* OR evol*)

The symbol * allow capturing possible variations in search terms such as plural and verb conjugation.

We **selected** considering the following **criteria**:

- *General Criteria:* We only included papers in English. However, we also considered non peer-reviewed contributions if their number of citations was higher than those of average peer-reviewed papers.
- *Selection by Title and Abstract:* We removed all papers that do not contain any references to microservices or that use the term microservices for different purposes or in different domains.
- *Selection by full papers:* We excluded papers that do not present any evidence related to our research questions or papers using microservices without any clear reference to the adopted architectural style, and microservices-based implementations that do not report any advantages and disadvantages of using microservices. We considered proposals of microservices-based architectural styles, implementations of microservices-based cloud systems, migrations of monolithic systems into cloud-native microservices-based systems, papers reporting their advantages and disadvantages.

We retrieved a total of 2754 unique papers applying the two queries to any fields. Based on the inclusion and exclusion criteria applied to title and abstract, we *selected 85 papers*. In order to validate the process, each author randomly and separately applied the inclusion and exclusion criteria to a set of 10 retrieved papers. Pairwise inter-rater reliability was measured across the three decision sets in order to get a fair/good agreement in the first process iteration. This process was carried out to clarify, where necessary, any incongruity among the authors. At this stage, we conducted a forward and backward systematic snowballing process among all papers referenced in the 85 papers elicited, so as to obtain a more comprehensive set of papers, as suggested by Wohlin (Wohlin, 2014). As for backward snowballing, we included all the references in the papers retrieved from bibliographic sources while for the forward snowballing we included all the papers that reference the retrieved papers, thereby obtaining 858 adidtional papers. After sifting these 858 papers based on title and abstract, we included resulting in 12 additional papers, obtaining a total of 97 papers. After reading the 97 papers, the process resulted in 40 peer-reviewed papers and 2 non peer-reviewed ones. The two works by Lewis and Fowler ([S1] and Richardson [S2]) added from the gray literature have a dramatically high number of citations compared to the remaining works. Note that the patterns presented at http://microservices.io/patterns/index.html are compiled by Richardson and were considered based on his comments in [S2] and selected study [S2].

The selection resulted in 42 accepted papers published up to end of 2016. 27 of these papers were published at conferences, while another 10 papers were accepted at workshops. Only 3 papers were published as journal articles, and 2 papers are non peer-reviewed websites (gray literature).

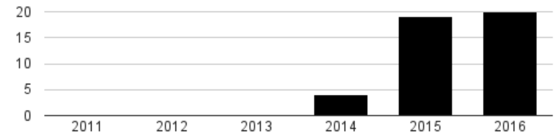


Figure 1: The chronological overview of the selected papers

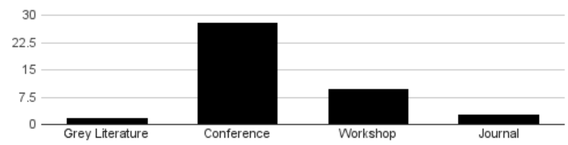


Figure 2: The publication types of the selected papers

# 3 The Pattern Catalogue

A key role in the microservices architectural style play the architecture patterns. Thus, we determine key patterns with their advantages and disadvantages. Three commonly used patterns emerge. In this classification, we attribute to the different patterns both the papers explicitly reporting the usage of a specific style and those where the adopted patterns can be clearly deduced from the description. Please note that we report on patterns in three subsections that categorize the emerging architecture patterns:

- *Orchestration and Coordination*-oriented architecture patterns that capture communication and coordination from a logical perspective.
- Patterns reflecting physical *Deployment* strategies for microservices on hosts through containers or virtual machines.
- Patterns that reflect on data management, specifically *Data Storage* options in addition to the orchestration and coordination patterns oriented at inter-component communication.

We will use a *Concept, Origin, Goal, Properties, Evolution, Reported Usage* and *Advantages, Disadvantages* template to discuss these microservices architecture patterns.

## 3.1 Orchestration and Coordination Architecture Patterns

### 3.1.1 Service Composition - The API-Gateway Pattern

*Concept:* Microservices can provide their functions to other services through an API. However, the creation of end-user applications based on the composition of different microservices requires a server-side aggregation mechanism. In the selected works, the API-Gateway emerged as a commonly recommended approach (Figure 3).

API Gateway is the entry point of the system that routes the requests to the appropriate microservices, also invoking multiple microservices and aggregating results. It provides a tailored API to each client to route requests, transform protocols, and implement shared logic like authentication and rate-limiters. Moreover, it can be responsible for different tasks such as authentication, monitoring, and static response handling. In some cases, the API Gateway can also serve as load balancer since it knows the location and the addresses of all services.

*Origin:* The API-Gateway is a pattern that resembles more SOA principles than REST ones, without an Enterprise Service Bus (ESB).

*Goal:* The main goal is to increase system performance and simplify interactions, thus reducing the number of requests per client. It acts as an entry point for the clients, routing their requests to the connected

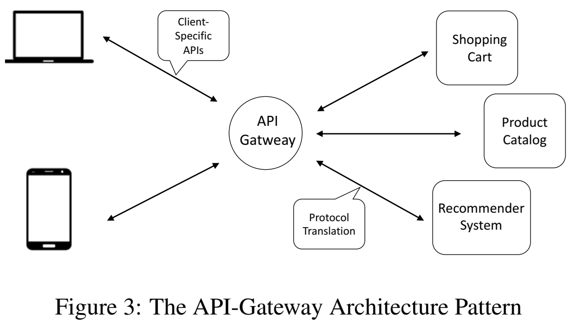


Figure 3: The API-Gateway Architecture Pattern

Table 1: The Papers Selection Process

| Selection Process | #considered papers | #rejected papers | Validation |
|---|---|---|---|
| Paper extracted from the bibliographic sources | 2754 | | 10 random papers independently classified by three researchers |
| Sift based on title and abstract | | 2669 | Good inter-rater agreement on first sift (K-statistic test) |
| Primary papers identified | 85 | | |
| Secondary papers inclusion | 858 | 855 | Systematic snowballing (Wohlin, 2014) including all the citations reported in the 85 primary papers and sifting them based on title and abstract |
| Full papers considered for review | 88 | | Each paper has been read completely by two researchers and 858 secondary papers were identified from references |
| Sift based on full reading | | 46 | Papers rejected based on inclusion and exclusion criteria |
| **Relevant papers included** | **42** | | |

services, aggregating the required contents, and serving them to the clients [S2].

*Properties:* The API-Gateway does not provide support for publishing, promoting, or administering services at any significant level. However, it is responsible for the generation of customized APIs for each platform and for optimizing communications between the clients and the application, encapsulating the microservices details. It allows microservices to evolve without influencing the clients. As an example, merging or partitioning two or more microservices only requires updating the API-Gateway to reflect the changes to any connected client. In Figure 3, the API-Gateway is responsible for communicating with the different front-ends, creating a custom API for each client for them to see only the features they need, which simplifies the creation of end-user applications without adding the complexity of exposing and parsing useless information.

*Evolution and Reported Usage:* This is the first architectural pattern reported in the considered works, and was named API-Gateway by Richardson [S2]. Ten works implemented cloud-native applications based on the pattern [S2], [S3], [S11], [S12], [S14], [S21], [S31], [S34], [S37] and [S39] reporting several API-Gateway specific **advantages**:

- *Ease of Extension*. Because the pattern can provide custom APIs, implementing new features is easier compared to other architectures [S14],[S3].
- *Market-centric Architecture*. Services can be easily modified, based on market needs, without the need to modify the whole system [S14].
- *Backward Compatibility*. The gateway guarantees for evolving services that existing clients are not hampered by interface endpoint changes [S34].

However, **disadvantages** have also been observed:

- *Potential Bottleneck.* The gateway layer is a single entry point for all requests. If not designed correctly, it could be a bottleneck [S14], [S39].
- *Implementation complexity.* The gateway layer increases the complexity since it requires implementation of several interfaces for each service [S14], [S34].
- API reuse must be considered carefully. Since each client can have a custom API, we must keep track of cases where different types of clients use the same API and modify both of them accordingly for changes to the API [S34].
- *Scalability.* When the number of microservices in a system explodes, a more efficient and scalable routing mechanism to route the traffic through the services APIs, and better configuration management to dynamically configurate and apply changes to the system will be needed [S37].

#### 3.1.2 Service Discovery Patterns

Multiple instances of the same microservice usually run in different virtualized containers/VMs. The communication among them must be dynamically defined and the clients must be able to efficiently communicate to the appropriate microservice that dynamically change instances. For this purpose, the service discovery dynamically supports the resolution of DNS address into IP addresses. Richardson proposes to differentiate between client-side and server-side patterns [S2].

**The Client-Side Discovery Pattern**. *Concept:* Here, clients query the Service Registry, select an available instance, and make a request directly (Fig. 4).

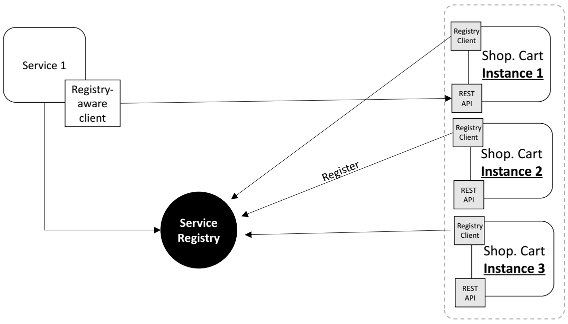


Figure 4: The Client-side Service Registry Pattern

*Goal:* With the client-side service discovery, the client is responsible for picking one of the available services instances to their network locations locations. Moreover, the service discovery can also serve as load balancer of the requests. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

*Evolution and Reported Usage:* According to Richardson, the main **advantage** of this pattern is connected to the *ease of development*. The clients are aware of the service instance locations and therefore can connect directly to them without adding the development complexity of the server-side discovery. The main reported **disadvantage** is the high coupling between the client and the service registry.

**The Server-Side Discovery Pattern**.

In this pattern, clients make requests via a load balancer, which queries the registry and forwards the request to an available instance.

*Goal:* Unlike the API-Gateway pattern, this pattern allows clients and microservices to talk to each other directly. It relies on a Service Registry, see Figure 5, acting in a similar manner as a DNS server.

*Properties:* The Service Registry knows the dynamic location of each microservice instance. When a client requests access to a specific service, it first asks the registry for the service location; the registry contacts the microservice to ensure its availability and forwards the location (usually IP address or DNS name and port) to the calling client. Finally, unlike in the API-Gateway, clients communicate directly with the required services and access all available APIs exposed by the service, without any filter or service interface translation.

*Evolution and Reported Usage:* Eleven papers implemented this pattern. Ten fully use it [S25], [S13], [S9], [S10], [S24], [S26], [S16], [S30] and [S38] while [S23] proposes a variant, implementing the registry through a NoSQL database. [S36] report a partial migration where a legacy SOA system provided some services in connection with new microservices. In this case, the legacy system was accessed like any other microservice. The Service Registry contained the addresses of all microservices and services provided by the legacy system.

This architectural pattern has several **advantages**:

- *Increased Maintainability.* All papers reported an increased maintainability.
- *Ease of Communication.* Services can communicate with each others directly, without interpretation [S25], [S36].
- *Health Management.* Resilience and scalability mechanisms provide "health management" and an out-scaling for atomic/composed service [S7].
- *Failure Safety.* In the case of failure, microservices can be easily restarted, due to their stateless properties [S7].
- *Software Understandability.* Better understandibility through small services is reported [S1], [S2].
- *Ease of Development.* Equally, easier development results from smaller services [S1], [S2].
- *Ease of Migration.* Existing services can be reimplemented with microservices, replacing the legacy service by changing its location in the Service Registry that will start to dynamically serve all microservices instances instead of statically pointing to the legacy system [S36], (Jamshidi et al., 2017).

Several papers also identified pattern **disadvantages**:

- *Interface Design Must be Fixed.* During maintenance, individual services may change internally, but there could be a need to also update the interface, requiring adaptation of all connected ser-

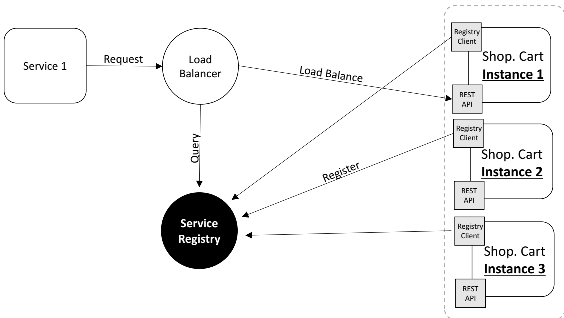


Figure 5: The Server-side Service Discovery Pattern

vices. Keeping the interface definition stable minimizes the impact of changes [S38].

- *Service Registry Complexity.* The registry layer increases the complexity as it requires the implementation of several interfaces per service [S16].
- *Reuse.* If not designed correctly, the registry could be the main bottleneck of the system [S25].
- *Distributed System Complexity*. Direct communication among services increases several aspects: *Communication among Services* [S2], *Distributed Transaction Complexity* [S2], *Testing* of distributed systems, including shared services among different teams can be tricky [S2].

#### 3.1.3 Hyrbid Patterns

**The Hybrid Pattern**. *Concept and Origin:* This pattern combines Service Registry and API-Gateway, replacing the API-gateway with a message bus.

*Goal and Properties:* Clients communicate with the message bus, which acts as a registry and routes requests to the requested microservices. Microservices communicate with each other through a message bus, similar to an Enterprise Service Bus used in SOA architectures.

*Evolution and Reported Usage:* Six works implemented this pattern [S27], [S33], [S32], [S35], [S4] and [S3] reporting the following **advantages**:

- *Ease of Migration.* This pattern eases the migration of existing SOA applications, since the ESB can be used a communication layer for the microservices that gradually replace legacy services.
- *Learning Curve.* Those familiar with SOA can easily implement this pattern with little training.

and a **disadvantage**:

- *SOA Issues.* The pattern does benefit from the IDEAL properties of microservices (isolated state, distribution, elasticity, automated management, loose coupling) and from the possibility to independently develop different services with different teams. However it still communicate thru the ESB as other SOA services.

### 3.2 Deployment Strategies & Patterns

We now look at architecture patterns from a microservice deployment strategy perspective (deployment pattern) that emerged from our mapping study.

**The Multiple Service per Host Pattern.** *Principle:* Multiple services run on the same host (node).

*Reported Usage:* Four of the selected works implemented this approach [S19], [S7], [S30], and [S33], without specifying whether they deployed the services into containers or VMs. Fifteen works adopted the pattern by deploying each service into a container [S25], [S10], [S35], [S8], [S9], [S11], [S32], [S34], [S36], [S37], [S38], [S40], [S41], [S16] and [S22]. Richardson refers to this sub-pattern as "service instance per container pattern" [S2]. Two works implemented this pattern deploying each microservice into a dedicated virtual machine [S27], [S31], called "service instance per virtual machine" by Richardson.

Despite adoption reports, only a few discuss their **advantages**:

- *Scalability*. Systems can easily scale to deploy multiple service instances at the same or in host.
- *Performance*. Multiple containers allow rapid deployment of new services compared to VMs [S40], [S34], [S10].

In principle, a **Single Service per Host Pattern** could be identified. Here [S2] mentions that every service is deployed in its own host. The main advantage of this approach is the complete isolation of services, reducing the possibility of conflicting resources. However, this dramatically reduces performance and scalability. In practice, dedicating a single node for a microservice is counterproductive, violating the basic idea of microservices.

### 3.3 Data Storage Patterns

Like any service, microservices need to store data. Sixteen implementations reported on the data storage pattern that they adopted. Among these papers, we identified three different data storage patterns that are also described by [S1], [S3] and [S24]. Although it is recommended to adopt Object Relational Mapping approaches with NoSQL databases [S1], the patterns identified are also applicable for relational databases.

**The Database-per-Service Pattern**. *Principle and Properties:* In this pattern, each microservice accesses its private database. This is the easiest approach for implementing microservices-based systems, and is often used to migrate existing monoliths with existing databases.

*Reported Usage:* In the selected works, six adopted this pattern [S12], [S23], [S36], [S11], [S24] and [S26]. This pattern has several **advantages**:

- *Scalability.* The database can be easily scaled in a database cluster within a second moment [S24], in case the service need to be scaled.
- *Independent Development.* Teams can work independently on a service, without influencing the

work of others in case of DB schema changes.

- *Security Mechanism.* Other microservices accessing and corrupting data not needed is avoided since only one microservice can access a schema.

**The Database Cluster Pattern**. *Principle and Properties:* The second storage pattern, described by Richardson [S2], proposes storing data on a database cluster. This improves scalability, allowing to move the databases to dedicated hardware. In order to preserve data consistency, microservices have a sub-set of database tables that can be accessed only from a single microservice; in other cases, each microservice may have a private database schema. From the microservices point of view, this pattern is identical to the "Shared Database Server" pattern since it in both cases the database is accessed in the same way. The only difference is internally in the adopted database. In both cases the database is seen from the microservice-side as a single database.

*Reported Usage:* The pattern was implemented by [S27], [S6], and [S15] by using a separated DB schema for each service. [S15] also proposed it for replicating the data across the DBs of each service.

This pattern has the **advantage** of improving scalability. It is recommended for implementations with huge data traffic. **Disadvantages** include:

- *Increased Complexity* due to cluster architecture.
- *Risk of Failure* increases because of the introduction of another component and the distributed mechanism.

**The Shared Database Server Pattern**. *Principle and Properties:* This pattern is similar to the Database Cluster Pattern, but, instead of using a database cluster, all microservices access a single shared database.

*Reported Usage:* Six implementations adopted this pattern [S13], [S39], [S25], [S18], [S30], and [S16]. All these implementations access to data concurrently, without any data isolation approach.

The main **advantage** reported is the simplicity of the migration from monolithic applications since existing schema can be reused without any changes. Moreover, the existing code base can be migrated without the need to make important changes (e.g., the data access layer remains identical).

## 3.4 Guiding Principles of a Microservices Architectural Style

In order to answer to RQ2, we report on microservices principles not related to a specific pattern, but mentioned in the papers as being important and serve as overarching principles that guide the pattern definition. A summary of generic and pattern-related advantages and disadvantages is reported in Table 2.

The most common reported advantages of microservice architectures are:

- *Increased maintainability* This is the key characteristic of microservices-based implementations reported by all the papers.
- *Possibility to write code in different languages*. Highlights the benefit of using different languages, in contrast to monolithic applications [S13], [S34], [S11].
- *Flexibility*. Every team can choose their own technology based on their needs [S30], [S14], [S38]
- *Reuse*. The maintenance of a single microservice will reflect on any connected project, reducing the effort overhead by applying the same changes to the same component used in different projects [S34], [S12].
- *Ease of Deployment.* Each microservice can be deployed independently, without the need to recompile and redeploy the whole application [S14], [S30]
- *Physical Isolation*. This is the key for scaling, thanks to the microservices architectural style [S3] and [S38].
- *Self-Healing*. Failed services can be easily restarted or replaced by previous safer versions [S7], [S30].
- *Application Complexity*. Since the application is decomposed into several components, these are less complex and easier to manage [S30].
- *Design for Failure*. Better support for continuous delivery of many small changes can help developers in changing one thing at a time [S37].
- *Observability*. A microservices architecture helps to visualize the "health status" of every services in the system in order to quickly locate and respond to any problem that occurs [S37].
- *Unlimited Application size.* In monolithic applications the application size is limited by the hardware and by web container specifications, while with microservices we could build a system with, in theory, no size limits. [S13].

As agreed advantages, these can be considered to form the **principles of the architectural style**.

However, several papers identified potential **disadvantages** that need to be taken into account:

- *Testing Complexity*. More components and collaborations among them increases testing complexity [S37], [S21], [S31], [S24], [S26], [S28].

- *Implementation Effort.* It is reported that implementing microservices requires more effort than monolithic systems [S30], [S38], [S28]

- *Network related issues.* Since endpoints are connected via a network, the network should be reliable [S14], [S41]. *Latency:* network latency can increase the communication time among microservices [S14], [S11], [S9]. *Bandwidth:* as service communication often relies on a network, bandwidth during normal and high peak operation needs to be considered. In cloud environments and other distributed systems settings, unreliability is given and failure is always possible. Microservices can also fail independently of each other, rather than together on a single node.

- *User Authorization.* The API exposed by the microservices need to be protected with a shared user-authentication mechanism, which is often much more complex to implement than monolithic solutions [S14].

- *Complexity.* In the case of applications with a relatively small number of users (hundreds or thousands), the monolith could be a faster approach to start and could possibly be refactored into a microservices-based architectural style once the user-base grows [S11].

- *Automation Requirement.* The explosion of the number of services and relationships among them requires a way to automate things. DevOps could be a good solution for this issue [S37].

- *Increased Dependence.* Microservices are meant to be decoupled, making it critical to preserve independence and independent deployability.

- *Development Complexity.* The learning curve is not very steep, but requires an experienced developer, at least for setting-up the basic architecture when compared to monolithic systems [S30].

# 4 Pattern Catalogue Discussion

Most of the implementations reported relate to research prototypes, with the goal of validating the proposed approaches (Table 3). Only six papers report on implementations in an industrial context. Regarding the size of the systems implemented, all the implementations are related to small-sized applications, except [S38]. Only four implementations report on the development language used ([S11], [S32] Java/NodeJS, [S34] php/NodeJS/Python, [S13] php).

## 4.1 Architecture and Deployment Pattern Applications

Several patterns for microservice-based systems have emerged from existing implementations (Table 2). We can associate some patterns with specific application settings such as a monolith-to-microservice or SOA-to-microservice migration.

*Migration:* Several implementations report the usage of hybrid systems, aimed at migrating existing SOA-based applications to microservices. Maintenance, and specially independent deployment and the possibility to develop different services with different non-interacting teams, are considered to be the main reasons for migrating monoliths to microservices. The flexibility to write in different languages and to deploy the services on the most suitable hardware is also considered a very important reason for the migration.

Reported migrations from monolithic systems tend to be architected with an API-Gateway architecture, probably due to the fact that, since the systems need to be completely re-developed and re-architected, this was done directly with this approach. Migrations from SOA-based systems, on the other hand, tend to have a hybrid pattern, keeping the Enterprise Service Bus as a communication layer between microservices and existing SOA services. Based on this, the Enterprise Service Bus could re-emerge in future evolutions of microservices.

*Deployment:* Another interesting outcome is that deployment of microservices is still not clear. As reported for some implementations, microservices are deployed in a private VM, requiring complete startup of the whole machine during the deployment, thus defeating the possibility of quick deployment and decreasing system maintainability due to the need for maintaining a dedicated operating system, service container, and all VM-related tasks.

## 4.2 Trends and Open Issues

We have compiled a catalogue of microservices architectural patterns, that together with the principles we identified form an architectural style (Malavolta and Lago, 2017). Some researchers also propose a new variant of such a microservice architectural style ([S24] and [S26]), applying a database approach for microservice orchestration. However, because they have just been published, no implementations have adopted these practices yet. Also in this case, we believe that an empirical validation and a set of benchmarks (Aderaldo et al., 2017) comparing this new style with existing one could be highly beneficial for

Table 2: Classification of Advantages and Disadvantages of the Identified Patterns

| | Pattern | Advantages | Disadvantages |
|---|---|---|---|
| Orchestration & Coordination | General | -Increased maintainability<br>-Can use different languages<br>-Flexibility<br>-Reuse<br>-Physical isolation<br>-Self-healing | -Development/testing complexity<br>-Implementation effort<br>-Network-related issue |
| | API Gateway | -Extension easiness<br>-Market-centric Architecture<br>-Backward compatibility | -Potential bottleneck<br>-Development complexity<br>-Scalability |
| | Service Registry | -Increased maintainability<br>-Communic., developm., migration<br>-Software understandability<br>-Failure safety | -Interface design must be fixed<br>-Service registry complexity<br>-Reuse<br>-Distributed system complexity |
| | Hybrid | -Migration easiness<br>-Learning curve | -SOA/ESB integration issues |
| Deploy-ment | Multiple service per host | -Scalability<br>-Performance | |
| | Single service per host | -Service isolation | -Scalability<br>-Performance |
| Data Storage | DB per Service | -Scalability<br>-Independent development<br>-Security mechanism | -Data needs to be splitted<br>-Data consistency |
| | DB Cluster | -Scalability<br>-Implementation easiness | -Increase complexity<br>-Failure risks |
| | Shared DB server | -Migration easiness<br>-Data consistency | -Lack of data isolation<br>-Scalability |

Table 3: The Implementations Reported in the Selected Works

| | Research Prototype / System | Validation-specific Implementations | Industrial Implementations |
|---|---|---|---|
| Websites | [S11], [S39] | [S15], [S24], [S26], [S31] | [S13], [S32] |
| Services & APIs | IOT Integration [S33] | [S9], [S10], [S14], [S16], [S23], [S37], [S36] | [S21], [S34] |
| Others | Enterprise Measurement [S4]<br>IP Multimedia [S25] | Benchmark/Test [S35], [S41], [S42]<br>Business Process Modeling [S12] | Mobile Dev Platform [S38]<br>Deployment Platform [S30] |

researchers and practitioners.

Microservices move us towards continuous development and delivery (Pahl et al., 2018). DevOps is the integration of *Dev*elopment and *Op*erations. An important observation is that microservices require full application stacks. That means that their infrastructure resources like data stores and networks have to be managed properly. DevOps in this way connects software development with concerns of technology operations and quality management. In this context, container technology and cloud-native services can provide the required cloud-based virtualization and implementation support for microservices (Pahl et al., 2018).

Resulting from our study, but also discussions in (Malavolta and Lago, 2017) or (Pahl et al., 2018), we can identify the following **emerging issues**:

- *Comparison of SOA and Microservices.* The differences have not been thoroughly investigated. There is a lack of comparison from different points of view (e.g., performance, development effort, maintenance).
- *Microservices Explosion.* What happens once a growing system has thousands, or millions of microservices is still not clear. Will all the mentioned microservices qualities degrade (Kratzke and Quint,2017)?
- *Negative Results.* In which contexts do microservices turn out to be counterproductive? Are there anti-patterns (Kratzke and Quint,2017)?

All require more experience reports and empirical investigations.

# 5 Conclusion

We presented principles and a catalogue of patterns of a microservices-based architectural style. We have determined these patterns and principles by conducting a systematic mapping study.

We have used the concept of architecture patterns to extract some common structural properties of microservice architectures. Three orchestration and data-storage patterns emerged that appear to be widely adopted when implementing microservices-based systems. Although some patterns were clearly used for migrating existing monolithic applications (using service registry pattern) and others for migrating existing SOA applications (using hybrid pattern), adopting the API-Gateway pattern in the orchestration layer in order to benefit from microservice architectures without needing to refactor a second time emerges as a key recommendation. Overall, a 3-layered catalogue of patterns emerges with patterns

- for *orchestration/coordination* and *storage* as classical *structure-oriented architectural pattern* groups that address component-level interaction and data management concerns, resp., as traditional software architecture concerns,
- for *deployment* alternatives that link microservices to their deployment in the form of containers or VMs, linking microservices inherently to their deployment strategies in host environments.

We have embedded these in a discussion of principle benefits and disadvantages of a microservices style overall.

Independent deployability, being based on strong isolation, and easing the deployment and self-management activities such as scaling and self-healing, and also maintainability and reuse as classical architecture concerns are the most widely agreed beneficial principles. The emergence of deployment patterns here is a first step towards a deeper coverage of continuous development, integration and deployment in a DevOps style framework. The patterns confirm the need to address microservices in conjunction with their deployment through containers or VMs.

A further observation concerns the notion of an architecture style itself in the presence of continuous architecting. The latter becomes an integral element of software architecture these days. Correspondingly, an architectural style needs to cover continuous architecting activities as well in addition to purely development stage concerns such as system design usually focused on in architectural styles.

# REFERENCES


Aderaldo, C.M., Mendonca, N.C., Pahl, C. and Jamshidi, P. (2017). Benchmark requirements for microservices architecture research. Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering.

Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. IEEE Software. Volume: 33, Issue: 3, May-June

Bass, L., Weber, I. and Zhu, L. (2015). DevOps: A Software Architects Perspective. (1st ed.). Addison-Wesley Professional. ISBN 0134049845 9780134049847.

Di Cosmo, R., Eiche, A., Mauro, J., Zacchiroli, S., Zavattaro, G. and Zwolakowski, J. (2015). Automatic Deployment of Services in the Cloud with Aeolus Blender. 13th International Conference on Service-Oriented Computing, ICSOC.

Jamshidi, P., Pahl, C. and Mendonca, N.C. (2017). Pattern-based multicloud architecture migration. Software: Practice and Experience 47 (9), 1159-1184.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering.

Kratzke, K. and Quint, P.C. (2017). Understanding Cloud-Native Applications after 10 Years of Cloud Computing - A systematic mapping study. Journal of Systems and Software, 1-16.

Lewis, J. and Fowler, M. (2014). MicroServices. www.martinfowler.com/articles/microservices.html.

Malavolta, P.D., Lago, P. 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In ICSA 2017.

Pahl, C. and Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. In International Conference on Cloud Computing and Services Science.

Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing.

Pahl, C., Jamshidi, P. and Zimmermann, O. (2018). Architectural principles for cloud software. ACM Transaction on Internet Technology.

Petersen, K., Feldt, R., Mujtaba, S. and Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering. In EASE 2008.

Richardson, C. (2014). Microservices: Decomposing Applications for Deployability and Scalability. (2014) https://www.infoq.com/articles/microservices-intro.

Taibi, D. and Lenarduzzi, V. (2018). On the Definition of Microservices Bad Architectural Smells. IEEE Software. In press.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing, vol 4, issue 5, pp.22-32.

Wohlin, C. (2014). Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In EASE

Zimmermann, O. (2009). An architectural decision modeling framework for service oriented architecture design. Ph.D. Dissertation. University of Stuttgart.


# A The Selected Studies


[S1] Lewis, J. and Fowler, M. 2014. Microservices". http://martinfowler.com/articles/microservices.html.

[S2] Richardson, C. 2014. Microservice Architecture" http://microservices.io.

[S3] Namiot, D. and Sneps-Sneppe, M. 2014. On microservices architecture. International Journal of Open Information Technologies V.2(9).

[S4] Vianden, M., Lichter, H. and Steffens, A. 2014. Experience on a Microservice-Based Reference Architecture for Measurement Systems. Asia-Pacific Software Engineering Conference.

[S5] Anwar, A., Sailer, A., Kochut, A., Butt, A. 2015. Anatomy of Cloud Monitoring and Metering: A Case Study and Open Problems. Asia-Pacific Workshop on Systems.

[S6] Patanjali, S., Truninger, B., Harsh, P. and Bohnert, T. M. 2015. CYCLOPS: A micro service based approach for dynamic rating, charging & billing for cloud. Conference on Telecommunications.

[S7] Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds. A. 2015. Architecture for Self-managing Microservices. Int. Workshop on Automated Incident Mgmt in Cloud.

[S8] Chen, H.M., Kazman, R., Haziyev, S.,Kropov, V. and Chtchourov, D. 2015. Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform. Symposium on Reliable Distr Syst Workshop.

[S9] Stubbs, J., Moreira, W. and Dooley, R. 2015. Distributed Systems of Microservices Using Docker and Serfnode. Int. Workshop on Science Gateways.

[S10] Abdelbaky, M., Diaz-Montes, J., Parashar, M., Unuvar, M. and Steinder, M. 2015. Docker containers across multiple clouds and data center. Utility and Cloud Computing Conference.

[S11] Villamizar, M., Garcas, O., Castro, H. et al. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. Computing Colombian Conference

[S12] Alpers, S., Becker, C., Oberweis, A. and Schuster, T. 2015. Microservice Based Tool Support for Business Process Modelling. Enterprise Distributed Object Computing Workshop.

[S13] Le, V.D., Neff, M.M., Stewart,R.V., Kelley, R., Fritzinger, E., Dascalu, S.M. and Harris, F.C. 2015. Microservice-based architecture for the NRDC. Industrial Informatics Conference.

[S14] Malavalli, D. and Sathappan, S. 2015. Scalable Microservice Based Architecture for Enabling DMTF Profiles. Int. Conf. on Network and Service Management.

[S15] Viennot, N., Mathias, M. Lécuyer, Bell, J., Geambasu, R. and Nieh, J. 2015. Synapse: A Microservices Architecture for Heterogeneous-database Web Applications. European Conf. on Computer Systems.

[S16] Versteden, A., Pauwels, E. and Papantoniou, A. 2015. An ecosystem of user-facing microservices supported by semantic models. International USEWOD Workshop.

[S17] Rahman, M. and Gao, J. 2015. A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development. Service-Oriented System Engineering Symposium.

[S18] Rahman, M., Chen, Z. and Gao, J. 2015. A Service Framework for Parallel Test Execution on a Developer's Local Development Workstation. Service-Oriented System Engineering Symposium.

[S19] Rajagopalan, S. and Jamjoom, H. 2015. App-Bisect: Autonomous Healing for Microservice-based Apps. USENIX Conference on Hot Topics in Cloud Computing.

[S20] Meink, K. and Nycander, P. 2015. Learning-based testing of distributed microservice architectures: Correctness and fault injection". Software Engineering and Formal Methods workshop.

[S21] Bak, P., Melamed, R., Moshkovich, D., Nardi, Y., Ship, H., Yaeli, A. 2015. Location and Context-Based Microservices for Mobile and Internet of Things Workloads. Conference Mobile Services.

[S22] Savchenko, D. and Rodchenko, G. 2015. Microservices validation: Methodology and implementation". Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists.

[S23] Gadea, C., Trifan, M., Ionescu, D., Ionescu, B. 2016. A Reference Architecture for Real-time Microservice API Consumption. Workshop on CrossCloud Infrastructures & Platforms.

[S24] Messina, A., Rizzo, R., Storniolo, P., Urso, A. 2016. A Simplified Database Pattern for the Microservice Architecture. Advances in Databases, Knowledge, and Data Applications conference.

[S25] Potvin, P., Nabaee, M., Labeau, F., Nguyen, K. and Cheriet, M. 2016. Micro service cloud computing pattern for next generation networks. EAI International Summit.

[S26] Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M. and Urso, A. 2016. The database-is-the-service pattern for microservice architectures. Information Technologies in Bio-and Medical Informatics conference.

[S27] Leymann, F., Fehling, C., Wagner, S., Wettinger, J. 2016. Native cloud applications why virtual machines, images and containers miss the point. Cloud Comp and Service Science conference.

[S28] Killalea, T. 2016. The Hidden Dividends of Microservices. Communications of the ACM. V.59(8), pp. 42-45.

[S29] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. "Service cutter: A systematic approach to service decomposition". European Confeence on Service-Oriented and Cloud Computing.

[S30] Guo, D., Wang, W., Zeng,G. and Wei, Z. 2016. Microservices architecture based cloudware deployment platform for service computing. Symposyum on Service-Oriented System Engineering. 2016.

[S31] Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J. and Montesi, F. 2016. Self-Reconfiguring Microservices". Theory and Practice of Formal Methods.

[S32] Gadea, M., Trifan, D. Ionescu, et al. 2016. A microservices architecture for collaborative document editing enhanced with face recognition. Symposyum on Applied Computing.

[S33] Vresk, T. and Cavrak, I. 2016. Architecture of an interoperable IoT platform based on microservices. Information and Communication Technology, Electronics and Microelectronics Conference.

[S34] Scarborough, W., Arnold, C. and Dahan, M. 2016. Case Study: Microservice Evolution and Software Lifecycle of the XSEDE User Portal API. Conference on Diversity, Big Data & Science at Scale.

[S35] Kewley, R., Keste, N. and McDonnell, J. 2016. DEVS Distributed Modeling Framework: A Parallel DEVS Implementation via Microservices. Symposium on Theory of Modeling & Simulation.

[S36] O'Connor, R., Elger, P., Clarke, P., Paul, M. 2016. Exploring the impact of situational context - A case study of a software development process for a microservices architecture. International Conference on Software and System Processes.

[S37] Jaramillo, D., Nguyen, D. V. and Smart, R. 2016. Leveraging microservices architecture by using Docker technology SoutheastCon.

[S38] Balalaie, A., Heydarnoori, A. and Jamshidi, P. 2015. Migrating to Cloud-Native architectures using microservices: An experience report. European Conference on Service-Oriented and Cloud Computing.

[S39] Lin, J. Lin, L.C. and Huang, S. 2016. Migrating web applications to clouds with microservice architectures. International Conference on Applied System Innovation.

[S40] Xu,C., Zhu, H., Bayley, I., Lightfoot, D., Green, M. and Marshall P. 2016. CAOPLE: A programming language for microservices SaaS. Symp. on Service-Oriented System Engineering.

[S41] Amaral, M., Polo, J., Carrera, D., et al. 2015. Performance evaluation of microservices architectures using containers. Internatinoal Symposium on Network Computing and Applications.

[S42] Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K., and Sekar, V. 2016. Gremlin: Systematic Resilience Testing of Microservices. International Conference on Distributed Computing Systems.