Microservices as described by Sam Newman in his book building microservices [1] are basically small independent services, that work together. From this definition, the basic requirements of microservices can be derived.

* Small
* Autonomous
* Has an Interface

Moreover, in [1][5][29] the description continues and more requirements can also be derived

* Resilience
* Scalable
* Easy to deploy
* Reusable
* Automation
* Replicability
* Low influence if the interface changes

The following pages will go in details about each one of the requirements.

**Small**

The idea of microservices architecture is that the application will be composed of small services. In order to get the most out of microservices, each service should be doing one task. Such focus is tidy to the functionality requirements of the business. With having each microservice handling only one task, developers can be sure that their application will respect other requirements of microservices, such as resilience and scalability.

Each microservice is supposed to be small, the size of each service should be scaled down until it can’t be scaled anymore. Such approach will help to magnify the gains but also adds more overhead. Once each service is very small then it can easily be replaced, isolated, updated or deleted while the rest of the system is still running. On the other hand, having many small dynamic parts in the system will make it harder to manage and could add extra complexity.

**Autonomous**

Each service should have the possibility of getting deployed independently. Such feature gives applications more flexibility. It’ll help to make the application more loosely coupled. When one or more microservices need updates or changes. Hence the application can benefit from the concept of microservices. Moreover, the isolation of each microservice is important to make the whole system scalable [30].

Furthermore, each microservice is independent enough to allow developers to develop each microservice independently. This includes giving each team the freedom of choosing which toolsets to be used in the development of this particular microservice. Such freedom will help developers to choose the most suitable tool for each service depending on the service itself and regardless of other parts of the application.

**Has an Interface**

Since the application consists of many small independent parts that work together. This means that each part of the system should be able to provide some form of communication channels to other parts in order to be able to work with other parts and not to be isolated. Hence each microservice should provide an Application Programming Interface (API) that enables other microservices to talk to it and exchange data with it. Such API will enable microservices to support the principle of encapsulation. Where each microservice will have the freedom to hide its internal implementation and expose only a channel of communication.

Furthermore, having each microservice offering an API will help to make the system adheres more the principles of microservices. Hence allowing each microservice to be updated or changed without affecting the rest of the system as long as it keeps respecting its old API.

**Resilience**

One of the features offered by microservices architecture over monolithic architecture is having more resistance to system failures. When one part of the system, in monolithic application, fails the whole system will suffer from this failure. Such situation could easily deteriorate and in the end might lead to the complete failure of the system. On the other hand, when a service fails in microservices architecture, the system will be able to handle this failure much better.

Having each microservice as an independent entity means that each service can be isolated from the rest of the system once it shows signs of failures. Hence the system will be able to keep running and offers its services to clients except for the microservice that failed and currently isolated. This division of functionality leads to even more benefits, as in this case, the system will keep running while developers are working on fixing the failing part.

**Scalable**

One of the key features of microservice architecture is having systems that can be scaled easily. When the system needs new features in monolithic architecture, developers could end up scaling the whole system [5]. In order for it to handle the new additions to support the new features, where it is difficult to scale an application under monolithic architecture [30]. Such case means developers will spend more time worrying about how the system can be modified to support the new features. Thus more time and resources are spent. On the other hand, adding new features for a microservices-based application is much easier.

It could simply mean adding more services to the system. And so, other parts of the system will not be affected and will not need any intervention from the developers. Such case, means developers don’t have to spend time thinking about how the system can accept the new additions. Hence saving more resources.

**Easy to deploy**

Developers under monolithic architecture have to redeploy the whole system when they add new features of make changes. Meaning they would need to release new versions of the system with each update. And in case they try to gather each few updates together before releasing a new version then there are more risks in having errors in the released versions. But having the system composed of small independent parts means that adding or modifying is easier. Such modifications will only concern certain parts of the system. Hence developers will be able to have continuous deployment of the system without the need to release new versions each time.

Moreover, when something goes wrong after a new modification, it’ll be easier to isolate the new changes and fix the problem. Or even rollback to the old service while fixing the mistakes in the newly updated parts.

**Reusable**

Since an application built using microservices architecture will consists of many different small services, then other system might share some of the same functionality as another one. Hence developers could be able to use some of the services that’s built for one system in another one. This of course means saving more resources.

Furthermore, some services can be reused on different platforms. After building a web application using microservices, the same services can be used to help power a mobile application. Since microservices will have an interface thus making it easy to communicate with each service as long as the communication respects the provided interface. Such feature will help developers release more combatable applications on different platform, faster than ever.

**Automation**

Since services are independent entities that can be deployed on separate machines, developers need to be able to deploy the service and all its runtime environment automatically. It would make sense to deploy the services and their runtime environment manually when the number of services is still relatively small [5]. But when the number of services increases over the development time then developers should be able to deploy services and their runtime environments quickly.

**Replicability**

As mentioned in [29] “A microservice should be as easy to replace as possible.”

**Low influence if the interface changes**

Since microservices only method of communication is through API then system developers should strive to keep each interface unchanged. In case a change is absolutely necessary, the influence of such change must be keep minimal [30].

Since micro frontends are basically microservices architecture applied to the frontend part of the application. Then same requirements should be respected when developing frontends as a group of small independent micro frontends.

Each micro frontend should be small handling one task. Independent of other micro frontends thus any changes in one micro frontend should not affect other micro frontends. Hence, each micro frontend should be isolated from other micro frontends. But at the same time offers an interface for easy communication and data exchange with other micro frontends.

The final system should be able to handle failures gracefully. So a failure in one micro frontend should not lead to failure of the whole system. Other parts of the frontend should still be able to render normally.

The system should also be

Literature review

**Microservices in literature**

Microservices architecture as defined in [30] is a way to develop an application that’s composed of a group of small independent services. Similar definition is given in [33] where it says: “Microservices is an architecture style, in which large complex software applications are composed of one or more services”. Furthermore, microservices are also referred to as small independent services that work together [1]. The definition given in [5] also agrees with the above mentioned definitions, it states: “Microservices are relatively small, autonomous services that work collaboratively together”. Writers in [32] go on explaining that microservices is a product of Service Oriented architecture (SOA). The same is also mentioned in [31]

The size of each microservice has also gained a lot of attention when discussing microservices. “A microservice is a separate entity” that can’t get any smaller in size [1]. While [5] mentions that each microservice must implement only one business requirement. And it’s been argued that no rules have been given to how small each service should be. [33] Also agrees with [5] where each microservice should try to represent one business functionality.

Moreover how small each service is can be different from one system to another [5]. Other researchers also suggest that counting how many lines of code each service is should give developers a good measure on how big each microservice is [29]. It is also simply mentioned in [29] that services should be small, “The name “microservices” conveys the fact that the size of the service matters; obviously, microservices are supposed to be small”. Furthermore, counting the number of days each service takes to be developed is also another measure. In [1] it is advised that each microservice should not take more than two weeks to be developed. Just like other researchers [32] describes applications built with microservices architecture as a composition of small services. [33] Mentions that each microservice should only be concerned with implementing one task.

The literature also discuss how microservices should communicate between each other. At first it is mentioned in the definition of microservices architecture, that microservices collaborate with each other [1] [5]. This implies that microservices should find a method or mechanism to turn such collaboration into an actual exchange of data. [33] Mentions that each microservice should adopt an API where “They communicate with each other using language-neutral application programming interfaces (APIs)”. It is also mentioned that microservices only communicate with each other using network calls [1]. Moreover, [31] agrees that microservices should implement APIs interfaces to help communicate with each other, it states: “They are characterized by well-defined and explicitly published interfaces”. However. [29] Doesn’t go in details about how each microservice can communicate with other microservices, it is only mentioned that “Communication between microservices is distributed communication”. Researchers in [30] agree that microservices should implement API and that “Services can communicate with some lightweight mechanisms”.

The characteristics of microservices are also discussed by many writers. Many of them agree that each service should be small as in [1] [5] [29] [31] [32] [33]. The same agreement happens when talking about the independence of each microservice. For example, regarding the independence of each microservice [30] explains: “each of the services is running in its own independent process”. The same thought is mentioned in [1] and [31] where both of them mention the word autonomous to describe the independence of each microservice. [31] “Each service is fully autonomous”. While [1] says: “Microservices are small, autonomous services”. Same with [5] where the word “autonomous” is also mentioned, it is stated: “Microservices are relatively small, autonomous services”. Also [33] agree that microservices should be independent from each other.

The relationships and the effects each microservice has on other microservices is also discussed. [31] Mentions that when changing the implementation of a microservice other microservices should not be affected. [33] Agrees that microservices should not affect each other, the term “loosely coupled” is mentioned to describe the nature of the relationship between microservices. The word “isolation” is mentioned in [1] to describe how microservices should not affect each other when changes happen. On the other hand, such isolation could introduce “overhead” [1]. The write goes on and describe that microservices should be able to change independently from each other. It is stated: “The golden rule: can you make a change to a service and deploy it by itself without changing anything else?” [1]. Researchers in [5] agree with the concept of having microservices independent from each other, they say: “Loose coupling is critical to a microservices-based system”.

Furthermore, some researchers [29] suggest that microservices are supposed to be easily-replaced components. The same is suggested in [1] where microservices should have the possibility of being isolated from the rest of the system or even getting replaced completely. Such replacement and changes should not create complications for the system [1].

Moreover the term “bounded context” has been mentioned. [33] Explains it as microservices doesn’t need to know anything about how each microservice was developed. A similar idea is also mentioned by other researchers. [30] Mentions that microservices can communicate using their API and the way each microservice is implemented should not have an impact on the communication. Furthermore, they explains that this property gives more freedom to developers to use different tools for different microservice. This same concept is explained in [1] where the write states that such freedom in choosing different tools for different microservices could help developers in choosing the right tool for the right task.

Many researchers also agree that if one service fails, the system should still be able to operate normally [1] [29] [32]. Moreover since microservices support the promise of loosely coupling, at the times of failures, the failed service can be isolated and fixed while the rest of the system is still operating [1] [30].

**Microservices vs monolithic**

This section will give a comparison overview of microservices vs monolithic architecture in literature. The comparison will focus on the following:

* Size
* Scalability
* Loose coupling
* Maintainability

Size

Microservices architecture is composed of different services each service is collaborating with the one or more other services. This whole dynamic will come together to form the final system. One of the key characteristics of microservices architecture is the size of each service. Almost every paper suggests that the size of the service should be relatively small [1][5][29][34]. Even some researchers suggested counting lines of code as a measure to decide on the size of the service [29]. While many other researchers suggested that each service should be concerned in handling one task [33]. This task should be derived from the business requirements.

Such small service size helps managers to assign a small team of developers to each service hence making the development faster and more efficient. Moreover, each team can decide to use different tools for different services, depending on which toolset is the best for each service.

On the other hand, monolithic applications are divided into three layers: Backend, logic layer and frontend [34]. Each layer is covering many functionalities of the system. Hence the size of each layer could end up getting bigger and bigger with more business requirements. Furthermore, assigning small team of developers will be harder and each team needs to collaborate with other teams during the development. Moreover, usually, teams don’t have the freedom of choosing their own toolsets for development. They are bounded to what other teams are using and whether their toolset is compatible with the rest or not.

Scalability

Applications based on microservices have the chance to grow when there is a need for new features in the application. Adding such new features means adding new services to the system. And adding new services to the system is usually an easy task.

On the other hand, at some point monolithic applications grow to the point where they are hard to scale. The codebase becomes very big and complicated and each additions of new features require a good amount of work to allow the application to accept the new features.

In the book The Art of Scalability [35] the scale cube is introduced. It can be seen in figure 1. The Scale Cube has 3 axes: X-axis, Y-axis, and Z-axis [35]

* Horizontal Duplication and Cloning (X-Axis)
* Functional Decomposition and Segmentation (Y-Axis)
* Horizontal Data Partitioning - Shards (Z-Axis)

Commonly, monolithic-based application can scale only on one axis, that’s (X-Axis). On the other hand, a microservice-based application have the ability to scale over all three axes [36].

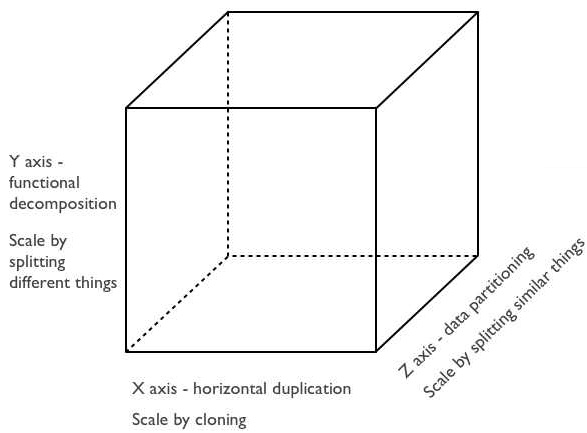


Figure 1. Scale cube [35]

Loose coupling

One of the important characteristics of microservices architecture is that services should be isolated from each other [33]. The connection between services is only achieved via a well-defined interfaces. And each service can be modified as long as it is still respecting its interface. Hence making the application components loosely coupled. On the other hand, monolithic applications developers have to take extra measures to make sure that the parts of their applications don’t overlap which costs them more time and work. The importance of loose coupling is stated clearly in [5] as “Loose coupling is critical to a microservices-based system”.

Maintainability

Since microservices are independent entities, then isolating each service in case of a failure is possible while the rest of the system continues operating [1]. However, the same can’t be true for monolithic-based applications and in worst cases one failure in the system could cascade to stop the whole system from operating [1].

In [34] researchers use different points to compare monolithic-based applications to microservices-based application. Table 1 presents the comparison. They conclude that both architecture styles have positive and negative points. In general microservice architecture is more suitable for projects with big codebase. But once the project is small building it with microservices architecture could bring an additional overhead [34].

|  |  |  |
| --- | --- | --- |
| Category | Monolith | Microservices |
| Time to market | Fast in the beginning, slower  Later as codebase grows. | Slower in the beginning because  of the technical challenges that microservices have. Faster later |
| Refactoring | Hard to do, as changes can affect multiple places. | Easier and safe because changes are contained inside the microservice. |
| Deployment | The whole monolith has to be deployed always. | Can be deployed in small parts, only one service at a time. |
| Coding language | Hard to change. As codebase is large. Requires big rewriting. | Language and tools can be selected per service. Services are small so changing is easy. |
| Scaling | Scaling means deploying the whole monolith. | Scaling can be done per service. |
| DevOps skills | Doesn’t require much as the number of technologies is limited. | Multiple different technologies a lot of DevOps skills required. |
| Understandability | Hard to understand as complexity is high. A lot of moving parts. | Easy to understand as codebase  is strictly modular and services use SRP. |
| Performance | No communicational overhead.  Technology stack might not support performance. | Communication adds overhead.  Possible performance gains because of technology choices |

Table 1. Comparing monolith and microservices [34]