Microservices

**What are microservices?**

Microservices offer a way to build web-scale applications by breaking a large application down into small, independent services.

Microservices are small, autonomous services that work together. In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Microservices are really nothing more than another architectural solution for designing complex – mostly web-based – applications. Microservices have gained prominence as an evolution from SOA (Service Oriented Architecture), an approach that was designed to overcome the disadvantages of traditional monolithic architectures. In this article, we’ll try to explore the evolution of development from monolithic architectures toward microservices and its underlying justifications, as well as the pros and cons of microservices.

The main goal of this article is to explain the major ideas and principles of microservices, pros and cons for this pattern.

We will try to find the answer for the following questions regarding that microservices can be useful or a real pain for developing applications:

1. Are microservices appropriate/needed for the business' goals?
2. Is the effort needed to implement microservices worth spending?
3. Do we need scalability?
4. Do we need isolation?
5. Are we prone to being affected by disadvantages?
6. Both from a business and technical point of view - are those aspects also clear for our client?
7. Did we take a decision together, looking at pros and cons?

**1. Are microservices appropriate/needed for the business' goals?**

We will try to find the balance between advantages (scalability, isolation, reliable processes, quick time to market etc.) and disadvantages (integration/collaboration, general higher complexity, culture incompatibility etc.) –

Microservices are introducing an example of the modular architectural style, based on the idea of breaking large software projects into smaller, independent, and loosely coupled parts, which has gained prominence among developers for its dynamic and agile qualities in API management and execution of highly defined and discrete tasks.

They enable IT organizations to be more agile and reduce costs by taking advantage of the granularity and reuse of microservices. Yet, like other new architectural paradigms, they introduce challenges as well.

Microservices are focused on providing one capability. “Micro” doesn’t necessarily mean that it’s small, although it often is. It’s just singularly focused. It provides one piece of functionality very well. An ideal microservice also owns it data and data model, and is not dependent on any other microservice or service for it.

As with any definition that outlines common characteristics, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics.

Microservice architectures are introducing some concepts as it follows:

* 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

**Advantages of microservices**

The advantages of microservices seem strong enough to have convinced some big enterprise to adopt the methodology.

Compared to more monolithic design structures, microservices offer:

* **Scalability**: Since the services are separate, the most needed ones at the appropriate times can be more easily scaled, as opposed to the whole application. When done correctly, this can impact cost savings.
* **Isolation**: By physically separating components, we have an opportunity to introduce more isolation between application components, which allows us to reduce the coupling between components and potentially increase each components scalability. When working with microservices, we focus on four dimensions of isolation: **State**, **Space**, **Time**, and **Failure**.
  + **Isolation of state**

State refers specifically to a microservice's persisted data, which the service is wholly responsible for managing its state. Any access to this state from another service is made exclusively through the service's API. This approach is intended to avoid any direct access to a service's database across services.

* + **Isolation of space**

Space refers to the location in which the service is deployed. In monolithic applications, all components are deployed together and execute as a single process. The deployment strategy changes radically with microservice architectures. In a microservice architecture, services are deployed independently and execute within a separate process.

* + **Isolation of failure**

In a monolithic architecture, it is quite common for a single failing component to bring down an entire application. A single uncaught exception can crash the monolithic application's single OS process, taking the application offline. Larger applications can remain mostly unaffected by the failure of a single module.

* + **Isolation of time**

Monolithic architectures execute all of the application's constituent components within the same OS process. Since these components are co-located, they are naturally able to invoke each other directly and generally follow a synchronous call pattern. Microservices generally limit synchronous calls to services that respond quickly. Long-running service calls should be made asynchronously. By making long-running calls asynchronously, the caller no longer waits for the long-running service's response avoiding idle wait-time in the caller. By eliminating the idle time wasted waiting, we improve resource utilization in the caller.

* **Reliable processes:** 
  + **Eliminate vendor or technology lock-in**: Microservices provide the flexibility to try out a new technology stack on an individual service as needed. There won’t be as many dependency concerns and rolling back changes becomes much easier. With less code in play, there is more flexibility.
  + **Ease of understanding:**With added simplicity, developers can better understand the functionality of a service.
  + **Smaller and faster deployments**: Smaller codebases and scope = quicker deployments, which also allow you to start to explore the benefits of Continuous Deployment.
* **Faster Time to Market**: A microservice architecture presents flexibility. It allows developers to easily update a microservice’s behavior without having to rewrite a lot of code. This is one of the main benefits of decoupling your services. Also, new functionality can be developed quickly and deployed within this architecture.

**Disadvantages of microservices**

Microservices may be a hot trend, but the architecture does have drawbacks. In general, the main negative of microservices is the complexity that any distributed system has.

Here’s a list of some potential pain areas and other cons associated with microservices designs:

* **Communication between services is complex**: Since everything is now an independent service, you have to carefully handle requests traveling between your modules. In one such scenario, developers may be forced to write extra code to avoid disruption. Over time, complications will arise when remote calls experience latency.
* **Communication** between microservices can mean poorer performance, as sending messages back and forth comes with a certain overhead. And, while teams can choose which programming language and platform they want to use, they also need to collaborate much better. After all, they need to manage the whole lifecycle of the microservice, from start to end.
* **Collaboration** : Needs more collaboration (each team has to cover the whole microservice lifecycle)
* **Integration**: Harder to integrate and test and also to monitor of services because of the complexity of the architecture.
* **More services equals more resources**: Multiple databases and transaction management can be painful.
* **Global testing is difficult**: Testing a microservices-based application can be cumbersome. In a monolithic approach, we would just need to launch our WAR on an application server and ensure its connectivity with the underlying database. With microservices, each dependent service needs to be confirmed before testing can occur.
* **Debugging problems can be harder**: Each service has its own set of logs to go through. Log, logs, and more logs.
* **Deployment challengers**: The product may need coordination among multiple services, which may not be as straightforward as deploying a WAR in a container.
* **Large vs small product companies**: Microservices are great for large companies, but can be slower to implement and too complicated for small companies who need to create and iterate quickly, and don’t want to get bogged down in complex orchestration.

Therefore, with the right kind of automation and tools and the properly trained staff, all the above drawbacks can be addressed.

Microservices might also be understood by what they are not. The two comparisons drawn most frequently with microservices architecture are monolithic architecture and service-oriented architecture (SOA).

The difference between microservices and monolithic architecture is that microservices compose a single application from many smaller, loosely coupled services as opposed to the monolithic approach of a large, tightly coupled application.

1. **Is the effort needed to implement microservices worth spending?**

When considering any IT infrastructure changes, the most critical consideration is “Do the benefits and value of a new technology solution override its costs?”

In the case of adopting microservices, the answer is definitely yes, especially as they accelerate time to market of new applications for demanding customers while leveraging the stability of legacy systems.

For those who want to better enable legacy applications to be assets  — rather than speed bumps — in your digital journey, the idea of adopting microservices can be exciting and sobering simultaneously. Adopting a microservices architecture is similar to embracing any other new technology or software discipline: you need the appropriate environment and staff. Adopting microservices does present some unique cost-benefit considerations. Here are six to keep in mind:

## The Cost of Getting Started

Here are a few of the “getting started” expenses that might be incurred:

* **Personnel:** Not all developers will be familiar with microservices architecture.
* **Organizational expenses:** Microservices architecture benefits are best realized when administered by small, cross-functional teams.
* **Tools:** Containerization and other supporting technologies.

**Rule of thumb:** Depending on where you are starting from, the “Getting Started” costs might be a budget concern, but the investment is negligible when compared to the downstream benefits, which will be significant.

## The Cost of Maintenance

Let’s assume that you’re running application monoliths, as opposed to green-field development. Maintaining these applications takes time, because they have interlocking dependencies.

Imagine the login manager fails in a monolithic application. Every other part of the application relies on the login manager, so when it is down, it’s all down. You cannot support a growing number of customers with an application that crashes frequently; while there are workarounds — such as failover services and instancing — they tend to be expensive.

As for adopting microservices, the first part of value generation comes in the form of maintenance advantages.

**Rule of thumb:** A microservices architecture, with fewer application dependencies and simple APIs, will immediately reduce the time and money spent on application maintenance. Savings on application maintenance have proven to be more than enough to cover the “getting started” costs within a few years.

## The Cost of Marrying Quality and Speed

The dependencies inherent in any monolithic application will inhibit innovation. Application monoliths don’t tend to play well with newer development techniques — such as Agile and DevOps — that emphasize speed. Any update that’s made to one part of the application affects other parts, so any update will need to be tested thoroughly.

**Rule of thumb:** Microservices help developers increase the speed of their development without sacrificing quality. This results in a competitive advantage: They will be able to refine their applications faster than those who haven’t yet adopted a microservices strategy. External customers and vendors will build up loyalty to these applications, while internal end-users will become more productive.

## The Cost of Quality

**Here’s how it works:** DevOps, Agile, and other modern development practices rely heavily on automated testing. The idea is to give developers or QA personnel the ability to set up several test environments in just a few clicks and let an automated testing program handle most of the effort. DevOps practices expect to rebuild and test quickly, without hassle.

Thereby reducing the need for continuous automated testing of the legacy application and the significant time it takes to rebuild the monolith”

Done correctly, microservices should require zero change to your legacy applications, thereby reducing both the need for continuous automated testing of the legacy application and the significant time it takes to rebuild a monolith.

**Rule of thumb:** Microservices make for a much cleaner testing process. They’re built simpler, so it’s easier to review their code. As a result, it’s also simpler to perform unit tests. By definition, microservices are small, simple and quick and easy to write; therefore they are equally easy to test.

## The Cost of Speed

The value of speed is different for every organization, but one can easily appreciate the benefit of a 90% increase in delivered services per year or being able to push out 20 new services every five weeks.

**Rule of thumb:** Speed of development + Quality of Development = Competitive Advantage.

For example:

* An insurance organization that leverages microservices to compete in the large insurance quote comparison engines can now be part of a fast-growing digital channel used by millions of shoppers.
* A bank that offers mobile bill-pay and mobile deposits as a result of microservices can now capture younger generations of new banking clients who can provide lifelong value and add millions in deposits.

## The Total Cost of Ownership

If you use software that creates microservice directly from the legacy or on-premise system, you can bypass complex ESB/SOA layers that were previously used for legacy integrations. Depending on your architectural strategy and if you are focusing on a few microservices or an entire microservice architecture, your savings can be quite significant — easily in the millions for large organizations.

**Rule of thumb:** When you decommission ESB/SOA and replace it with independent, directly connected microservice-based APIs, you will always achieve exponential savings by comparison. Since microservice-based APIs are direct, they can perform five times faster; therefore, you may be able to free up hardware processing power as well.

## Considering Adopting Microservices? Find a Partner

Find an innovative company that will work with you on a pressing and compelling business use case, where a microservice architecture can bring immediate value. Define the success criteria of a low-risk proof of concept that will give you the ability to envision “what is possible” and the data points to confidently assess the potential and cost/value benchmarks necessary for you to begin your digital journey.

1. **Do we need scalability?**

Vertical scaling isn’t a sustainable or scalable way to architect microservices. It may appear to work out all right in situations where each microservice has dedicated hardware, but it will not work well with the new hardware abstraction and isolation technologies that are used in the tech world today, like Docker and Apache Mesos. Always optimize for concurrency and partitioning if you want to build a scalable application.

Other resource bottlenecks are not as obvious, and the best way to discover them is to run extensive load testing on the service.

When dealing with incredibly complex dependency chains, making sure that all microservice teams tie the scalability of their services to high-level business metrics (using the qualitative growth scale) can make sure that all services are properly prepared for expected growth, even when cross-team communication becomes difficult.

The problem of dependency scaling is an especially strong argument for the implementation of scalability and performance standards across every part of the microservice ecosystem. Most microservices do not live in isolation. Nearly every single microservice is a small part of large, intertwined, intricate dependency chains. In most cases, scaling the entire overall product, the organization, and the ecosystem effectively requires that each piece of the system scales together with the rest. Having a small number of super efficient, performant, and scalable microservices in a system where the rest of the services aren’t held to (and don’t meet) the same standards renders the efficiency of the standardized services completely moot.

Aside from standardization across the ecosystem, and holding each microservice development team to high scalability standards, it’s important that development teams work together across microservice boundaries to ensure that each dependency chain will scale together. The development teams responsible for any dependencies of a microservice need to be alerted when increases in traffic are expected. Cross-team communication and collaboration are essential here: regularly communicating with clients and dependencies about a service’s scalability requirements, status, and any bottlenecks can help to guarantee that any services that rely on each other are prepared for growth and aware of any potential scalability bottlenecks. A strategy that I’ve used to help teams accomplish this is by holding architecture and scalability overview meetings with teams whose services rely on one another. In these meetings, we cover the architecture of each service and its scalability limitations, then discuss together what needs to be done to scale the entire set of services.

Now that you have a better understanding of scalability and performance, use the following list of questions to assess the production-readiness of your microservice(s) and microservice ecosystem. The questions are organized by topic, and correspond to the sections within this chapter.

## Knowing the Growth Scale

* What is this microservice’s qualitative growth scale?
* What is this microservice’s quantitative growth scale?

## Efficient Use of Resources

* Is the microservice running on dedicated or shared hardware?
* Are any resource abstraction and allocation technologies being used?

## Resource Awareness

* What are the microservice’s resource requirements (CPU, RAM, etc.)?
* How much traffic can one instance of the microservice handle?
* How much CPU does one instance of the microservice require?
* How much memory does one instance of the microservice require?
* Are there any other resource requirements that are specific to this microservice?
* What are the resource bottlenecks of this microservice?
* Does this microservice need to be scaled vertically, horizontally, or both?

## Capacity Planning

* Is capacity planning performed on a scheduled basis?
* What is the lead time for new hardware?
* How often are hardware requests made?
* Are any microservices given priority when hardware requests are made?
* Is capacity planning automated, or is it manual?

## Dependency Scaling

* What are this microservice’s dependencies?
* Are the dependencies scalable and performant?
* Will the dependencies scale with this microservice’s expected growth?
* Are dependency owners prepared for this microservice’s expected growth?

## Traffic Management

* Are the microservice’s traffic patterns well understood?
* Are changes to the service scheduled around traffic patterns?
* Are drastic changes in traffic patterns (especially bursts of traffic) handled carefully and appropriately?
* Can traffic be automatically routed to other datacenters in case of failure?

## Task Handling and Processing

* Is the microservice written in a programming language that will allow the service to be scalable and performant?
* Are there any scalability or performance limitations in the way the microservice handles requests?
* Are there any scalability or performance limitations in the way the microservice processes tasks?
* Do developers on the microservice team understand how their service processes tasks, how efficiently it processes those tasks, and how the service will perform as the number of tasks and requests increases?

## Scalable Data Storage

* Does this microservice handle data in a scalable and performant way?
* What type of data does this microservice need to store?
* What is the schema needed for its data?
* How many transactions are needed and/or made per second?
* Does this microservice need higher read or write performance?
* Is it read-heavy, write-heavy, or both?
* Is this service’s database scaled horizontally or vertically? Is it replicated or partitioned?
* Is this microservice using a dedicated or shared database?
* How does the service handle and/or store test data?

1. **Do we need isolation?**

Different microservices can require different extents of isolation. Deploying multiple microservices in an environment that does not meet all of their isolation requirements can result in a number of problems associated with security, regulation, auditing, licensing, availability, and performance.

When building microservices, one of the primary goals is to achieve to a higher degree of isolation between components within our application.

Formal isolation levels are defined and assigned to microservices. The microservices are then deployed in environments that leverage logical and physical isolation mechanisms to support the required isolation levels.

## ISOLATION DIMENSIONALITY

As we move from monolithic to microservice-based architectures, we decompose the application into multiple, independently executing services. By physically separating components, we have an opportunity to introduce more isolation between application components, which allows us to reduce the coupling between components and potentially increase each components scalability.

So, what do we mean by isolation? When working with microservices, we focus on four dimensions of isolation: **State**, **Space**, **Time**, and **Failure**.

**Isolation of state**

One of the primary characteristics of a microservice is its state. State refers specifically to a microservice's persisted data, which the service is wholly responsible for managing its state. Any access to this state from another service is made exclusively through the service's API. This approach is intended to avoid any direct access to a service's database across services.

## ISOLATION OF SPACE

Another important dimension of isolation is space. Space refers to the location in which the service is deployed. In monolithic applications, all components are deployed together and execute as a single process. The deployment strategy changes radically with microservice architectures. In a microservice architecture, services are deployed independently and execute within a separate process.

## ISOLATION OF FAILURE

In a monolithic architecture, it is quite common for a single failing component to bring down an entire application. A single uncaught exception can crash the monolithic application's single OS process, taking the application offline.

Since each service in a microservice architecture execute independently, the same uncaught exception will still cause the service to crash and become unavailable to it's consuming services, but it no longer crashes the entire application. When a service fails, the application continues to run (in a degraded state).

## ISOLATION OF TIME

As we have already mentioned, monolithic architectures execute all of the application's constituent components within the same OS process. Since these components are co-located, they are naturally able to invoke each other directly and generally follow a synchronous call pattern. While this simplifies the implementation, it forces each caller to wait until the call has completed before returning control to the caller.

Microservices generally limit synchronous calls to services that respond quickly. Long-running service calls should be made asynchronously. By making long-running calls asynchronously, the caller no longer waits for the long-running service's response avoiding idle wait-time in the caller. By eliminating the idle time wasted waiting, we improve resource utilization in the caller.

Isolating applications in space allows us to redistribute an application's components dynamically to manage resource utilization. Isolating the application in time using asynchronous service calls eliminates the need for resources to wait for long-running requests to finish and facilitates better distribution of load across the application. Finally, by isolating the application components from failure, we can prevent a single failing component from crashing the entire application.

In conclusion each isolation level is important and should be taken into account if the decision is taken to break the monolithic architecture into microservices.

1. **Are we prone to being affected by disadvantages?**

Clearly there are many advantages and disadvantages of microservices architecture to consider — but it’s important to consider the organizational culture and goals in this equation, too.

Enterprises more suited to microservices architecture are those that have an organizational culture comfortable with distributing work among small development teams. Also, microservices work best for organizations that need to innovate rapidly, and that have larger or more diverse user bases.

Microservices architecture advantages and disadvantages differ from traditional monolithic architecture, and this model isn’t ideal for every organization. However, the big shift to this modular architectural style is happening for a reason — more enterprises are realizing the need for faster, easier, more agile application development, and microservices enable this in ways monolithic architecture simply cannot.

Before implementing a microservices-based application development strategy, you must first evaluate your business readiness and make the necessary business-level decisions to ensure the long-term success of your project. Compared to the traditional monolithic approach, a microservices strategy involves several different forms of business investment, including financial investments, an investment in the culture of your workplace, and an investment in new development and operations.

If, for instance, you have a small application of low complexity that is developed and maintained by a small team and does not change frequently, then it might be hard to justify the cost imposed by microservices, and a monolithic style might suffice. By contrast, however, large and complex applications that are developed and maintained by a large (and likely churning) team, and which need to change frequently, become increasingly difficult to productively maintain when using a monolithic style. This is where microservices deliver their value.

Knowing the business domain inside out and the experience with the domain-driven design is crucial to identify the bounded context of each service. Since we allocate teams per each Microservice and allow them to work with minimal interference, getting the bounded context wrong would increase the communication overhead and inter-team dependencies, impacting the overall development speed. So for a project starting from scratch, selecting Microservices is a risky move.

1. **Both from a business and technical point of view - are those aspects also clear for our client?**

The decision should be taken together with the client and the things should be considered to be transparent and directly involving his knowledge.

Also it might be useful to know the answer for the following questions for both sides:

#### Are You In Familiar Territory?

Maybe traveling down an unknown path on the other hand, a monolith may have actually been the safer option, otherwise it’s quite clear scaling is going to be a primary requirement, especially in infrastructure based services like cloud log management.

#### Is the Team Prepared?

Does your team have experience with microservices? What if you quadruple the size of your team within the next year, are microservices ideal for that situation? Evaluating these dimensions of your team is crucial to the success of your project.

The experience of the team is needed and should be taken into account when the plan is to create/update the application to the

#### How’s Your Infrastructure?

In reality, in many cases you’re going to need cloud-based infrastructure to make microservices work for your project.

#### Evaluate The Business Risk

You may think microservices is the “right” way to go as a tech-savvy startup with high ambitions. But microservices pose a business risk so the client should be also aware of this aspect.

1. **Did we take a decision together, looking at pros and cons?**

Definitely, the decision should be taken for both sides after some technical discussions, management should be involved also for both sides and for the other hand the risk should be assumed.

Using the wisdom distilled from the answers to the above questions and by applying them to the team’s current situation the best solution should be considered and applied accordingly.