Microservices or Not? Your Team Has Already Decided

Adrian Kosmaczewski

2022-08-05

(Originally published on the VSHN blog in November 2021.)

Let's take a somewhat tangential approach to the subject of the Microservices architecture. Most discussions about it are centered around technological aspects: which language to choose, how to create the most RESTful services, which service mesh is the most performant, etc.

However at VSHN we have long ago figured out that the most important factor of success for software projects is people. And the thesis of this article is that the choice of Microservices as an architectural pattern has more to do with the organizational structure of your organization, rather than the technological constraints and features of the final deliverable.

Or, put differently: your team has already chosen an architecture for your software, even if they are not fully aware of it. And lo and behold, Microservices might or might not be it.

Definition

First of all we must define the Microservices architecture. What is it?

> "Microservices" is an architectural pattern in which the functionality of the whole system is decomposed into completely independent components, communicating with each other over the network.

The counterpart of the Microservices architecture is what is commonly referred to as the "Monolith", which has been the most common approach for web applications and services in the past 25 years. In a Monolith, all functions of the application, from data management to the UI, are all contained within the same binary or package.

On the opposite side of the street we find the Microservices architecture, where each service is responsible of its own implementation and data storage.

By definition, Microservices are fine grained, and have a single purpose. They have strong cohesion, that is, the operations they encapsulate have a high degree of relatedness. They are an example of the "Single Responsibility Principle". They are also deployed separately, with visibly bounded contexts, and they require DevOps approaches to their deployment and management, such as automation and CI/CD pipelines.

Very importantly, the Microservices architecture is a "share nothing architecture", in which individual services never share common code through libraries, instead restricting all interaction and communication through the network connecting them.

And last but not least, Microservices (as the name implies) should be as small as possible, have low requirements of memory, and should be able to start and stop in seconds.

Given all of these characteristics, Microservices are, by far, the most complex architectural pattern ever created. It is difficult to plan, it can dramatically increase the complexity of projects, and for some teams, experience has shown that it was impossible to cope with.

History

This idea of "components sending messages to one another" is absolutely not new. Back in 1966, one of the greatest computer scientists of all time, Alan Kay, coined the term "Object Oriented Programming". The industry co-opted and deformed his original definition, which was the following:

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.
>
> Alan Kay, source.

This text is from 2003; the following is from 1998:

> The big idea is "messaging" — that is what the kernal of Smalltalk/Squeak is all about (…) The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
>
> Alan Kay, source.

Alan Kay designed in the 1970s the Smalltalk programming language, based on these concepts. And after reading the texts above, it becomes clear that to a large extent, the Microservices architecture is an implementation of Alan Kay's ideas of messaging, decoupling and abstraction, taken to the extreme.

To achieve the current state of Microservices, though, many other breakthroughs were required. During the 2000s, the emergence of XML, the SOAP protocol, and its related web services made the term "Service Oriented Architecture" a very common staple in architectural discussions. The rise of Agile methodologies made the industry pivot towards the REST approach instead of SOAP. During the last decade, the rise of DevOps and the rise of containerization through Docker and Kubernetes finally enabled teams to deploy thousands of containers as a microservice architecture, thanks to the whole catalog of Cloud Native technologies.

Pros and Cons

If Microservice architectures are so complex, why use them? It turns out, they can bring many benefits:

- Since each component can be completely isolated from the others, once teams agree on their interfaces they can be developed, documented, and tested thoroughly, 100% independently from others. This allows teams to move forward in parallel, implementing features that have 0% chance of collision with one another.
- Teams are also encouraged to choose the programming language that fits best the particular task that their microservice must fulfill.
- Since they are by definition "micro" services, they are designed to be quickly launched and dismissed, so that they only intervene when required, making the whole system more efficient and responsive.
- The size of microservices also allows for higher availability, since it is possible to have many of them behind a load balancer; should any instance of a microservice fail, it can be quickly dismissed and a new one instantiated in its place, without loss of availability.
- Systems can be updated progressively, with each team fixing bugs and adding functionality without disturbing the others. As long as interfaces are respected (and eventually versioned) there is no reason for the system to suffer from updates.

But there are many reasons **not** to choose the Microservices architectural approach; among the most important:

- Performance: a system built with Microservices must take into account the latency between those services; and in this respect, Monolithic applications are faster; a function call is always faster than a network call.
- Team maturity: Microservices demand a certain degree of experience and expertise from teams; those new to Microservices have a higher chance of project failures.
- Cost: creating a Microservices system will be costlier, if anything, because of the overhead created by each individual service.
- Feasibility: sometimes it is simply not possible to use Microservices, for example when dealing with legacy systems.
- Team structure: this is a decisive factor we will talk about extensively later.
- Complexity: it is not uncommon to end up with systems composed of thousands of concurrent services, and this creates challenges that we will also discuss later.

I would like to talk now about the last two points, which are in our experience the biggest issues in Microservice implementations: **team structure** and the **perception and management of complexity.**

Conway's Law

One of the decisive factors that constrain teams' ability to implement microservices is often invisible and overlooked: their own structure. Again, this phenomenon is not new. In 1968, Melvin A. Conway wrote an article for the Datamation magazine called "How Do Committees Invent?" in which the following idea stands out:

> The basic thesis of this article is that organizations which design systems (…) are constrained to produce designs which are copies of the communication structures of these organizations.
>
> Melvin Conway, source.

There is extensive evidence, both anecdotal and empirical, of this fact through research.

The corollary of this principle is the following: the choice of a Monolithic versus a Microservices architecture is already ingrained in the very hierarchical chart representing any organization.

One of the services we offer at VSHN tackles, precisely, this very issue. In our "DevOps enablement workshop" we evaluate the degree of agility of organizations, and the extent and improvements that DevOps could bring. Based on that information, we reverse engineer their structure through Conway's Law in order to find a starting point for their digital transformation.

Complex vs. Complicated

Another important point is the distinction of "Complex" versus "Complicated", as these two words can sometimes be confused with one another in everyday language, and to make things more difficult, the word "Simple" can be used as an antonym of both.

"Complex" is borrowed from the Latin *complexus,* meaning "made of intertwined elements". *Complexus* is itself derived from *plectere* ("bend, intertwine"). This word has been used since the XVI century to qualify that which is made of heterogenous elements. It shares the same root (*plectere*) with the medical term "plexus" meaning "interlacing" and used since the 16th century as a medical term for "network of nerves or blood vessels".

(Source: Dictionnaire historique de la langue française by Alan Rey)

On the other hand, "Complicated" has a similar origin but a different construction: it comes from the Latin *complicare*, literally meaning "to fold by rolling up". Figuratively speaking this was taken as close to the notion of embarrassment or awkwardness. The word is composed of the word plicare which means "to fold". Watches commonly known as "Complications" (such as the Patek Philippe Calibre 89, the Franck Muller Aeternitas Mega and the "Référence 57260" de Vacheron Constantin) are, well, complicated machines by definition.

In short, "Complex" and "Complicated" stem from slightly different roots: the Latin root *plectere* ("to intertwine") in the former, and *plicare* ("to fold") for the latter. Complex conveys the idea of a network of intertwined objects, whose state and behavior are continuously altered by the interaction with their peers in said network. The word complicated implies an intrinsic apparent "obscurity" through folding unto itself, inviting to an "unfolding" discovery process.

Or, put in another way: individual Microservices should not be complicated, but a Microservice architecture is complex by definition. Monoliths, on the other hand, tend to become very complicated as time passes. And of course, neither are simple.

History shows that software developers have a passionate relationship with complication; complicated systems are great to brag about on Hacker News, while maintainers also cry about them in private.

A "best practice" in this context has one and only one basic job: to help engineers translate the complicated into the complex. Most software-related disasters are caused by a simple fact: because of deadlines, organization, tooling, or just plain ignorance, software developers have a tendency to build complicated, instead of complex, systems.

This is another point we take care of in our DevOps Workshop, through the evaluation of the current assets (not only source code, but also current databases schemas, security requirements, network topologies, deployment procedures and rhythms, etc.)

Migrate or Rewrite? Equilibrium

The complication of Monoliths by itself is not problematic; it makes for tightly bound systems, which tend to be very fast, because as we said, a function call is faster than a network call. After all, we have been building very successful monoliths in the past. But experience shows that they tend to present problems regarding availability and scalability. Microservices represent a diametrally oposed approach, based on complexity rather than complication, but one that solves those issues.

There is a tension, then, between complexity and complication on one side, and organizational forces on the other. Put in other words, there is a tension between monolithic and microservices systems on one side, and more or less hierarchical structures on the other. Achieving equilibrium between these forces is, then, the engineering challenge faced by software architects these days.

Many teams face today the task, either requested internally (from their management) or externally (through customer demand or vendor requirements) of migrating their monoliths into Microservice-based architectures. Architects can thankfully apply a few techniques to find an equilibrium:

1. Start your migration path knowing that very often one does not need to migrate the whole application to Microservices. Some parts can and should remain monolithic, and in particular, proven older systems, even if written in COBOL or older technologies, can still deliver value, and can play a very important role in the success of the transition.
2. Identify components correctly, so that when isolated they will be neither only functionality-driven, nor only data-driven, nor only request-driven, but rather driven by these three factors (functionality, data, and request) at the same time. Pay attention to the organizational chart, and use that as a basis for the decomposition in microservices.
3. Remember that network bandwidth is not infinite. Some interfaces should be "chunky", while others should be "chatty". Plan for latency issues from the start.
4. Reduce inter-service communication as much as possible, which can be done in many ways:
    1. Consolidating services together
    2. Consolidating data domains (combining database schemas or using common caches)
    3. Using technologies such as GraphQL to reduce network bandwidth
    4. Using messaging queues, such as RabbitMQ.
5. Adopt Microservice-friendly technologies, such as Docker containers, Kubernetes, Knative, or Red Hat's OpenShift and Quarkus.
6. Implement an automated testing strategy for each and every microservice.
7. Standardize technology stacks around containers & Kubernetes, and create common ground for a true microservice ecosystem within organizations.
8. Automate all of your work as much as possible, knowing that the effort for automation (DevOps, CI/CD) can be amortized over many services, and becomes thus a net investment in the long run.

As mentioned previously, we regularly help organizations in their digital transformation towards microservices, Kubernetes, OpenShift, DevOps, CI/CD, GitLab, and DevOps in general, to support your teams with the tooling they will need in the future. Borrowing Henny Portman's Team Topologies concept, VSHN can support both as an "Enabling Team" (DevOps workshop, consulting) and a "Platform Team" (Kubernetes/OpenShift) to build microservices on, ensuring stability and peace of mind.

Conclusion

Going beyond the hype, Microservice architectures bring great benefits, but can become huge challenges to software teams.

The best way to tackle those challenges consists in reverse engineering Conway's Law, and start by the analysis of the human organization of your teams first. Make them independent, agile, and free to choose the best tools for their job. Encourage them to negotiate with one another the interfaces of their respective components.

Let us create and run complex, not complicated, systems. We cannot get away from complexity; that is our job as engineers. But we can get rid of the complicated part.

As a closing thought, I will quote my former colleague and lifetime friend Adam Jones, an independent IT consultant from Geneva: in order to achieve success with the Microservice architecture, **you must embed structure in your activities, and remove it from your hierarchy.** It is not about making the structure go away; but instead, moving it to where it does the most good.