THE DEATH (OR NOT) OF MICROSERVICES

A form of software development that embraces small, independent components, microservices have a controversial reputation as both the next best thing and something that's so passé, it might already be dead.

Proponents of microservices are quick to cite the successes of Netflix and Amazon, who both work within some form of DevOps culture while embracing a microservice approach in their software development. Successfully deploying microservices can give the impression that your system is highly resilient and fault-tolerant. Plus, one school of thought links microservices with a well-functioning DevOps culture, rightly or wrongly.

Other say that microservices are just the latest buzzword for a been-there-done-that approach to software development, but one that's getting a revamp thanks to rapid development in container and orchestration technology.

Whether you're for or against microservices, there's something to be said for changing how we approach software development. Technology changes so rapidly in a couple years, let alone an entire generation, so we should consider changes to how we approach software development. Perhaps microservices is one approach.

The appeal of microservices

Microservices refer to a software development technique that's a variation on service-oriented

architecture (SOA). Developing microservices means you are designing software as a suite of independently deployable services. The ideal environment for microservices is very granular, featuring lightweight protocols and functionality.

Indeed, independence is key to microservices. While there is no specific architecture for microservices, a microservice environment often has common characteristics such as organization around business capability, automated deployment, decentralized control of languages and data, and endpoint intelligence.

Microservices are easier to understand in contrast to a monolithic system. A traditional monolithic app is usually built as a single unit that has three main parts:

- Server-side application, that handles HTTP requests, executes domain logic, and calls up data from associated database(s)
- Databases and external libraries
- Client-side user interface

In such a monolith system, any changes to the software requires building, testing, and deploying a whole new version of the server-side app, with little to no downtime or decrease in services on the client-side UX.

In everyone's favorite example of microservices, Netflix started with a single large unit of software: a video streaming platform. As worldwide use and their content library expanded, their functionality took a hit. Their solution? To break their bulky single unit into several smaller units of service – microservices. Each microservice has a specific, individual role: transcoding, uploading, streaming, downloading, recommending, and managing subscriptions.

Microservices: the pros and the cons

It's often tempting to replace the old with the new. Developers may think they can do it better, or at least approach a software problem with a language he's more familiar with in an attempt to make it less bloated.

The idea of breaking one massive system into smaller independent functionalities seems beneficial at first glance. Indeed, this independence can improve some dev cultures. Here are some common benefits to microservices:

- **Independent development**. If your small team is focused on one or two functionalities, you can learn and produce at a much quicker pace, especially if you don't need to interfere with – let alone know about – what other dev teams are working on.
- **Independent deployment**. Because components in one microservice aren't related to components in another (at least in theory), you can deploy faster with reduced risk to other services.
- **Independent scalability**. Depending on your product, you may have busy times that require scaling up. But if you only need to scale up part of your product, microservices allow that. For instance, Netflix may want to improve their streaming microservice around a big new release (like, say, Stranger Things Season 3). With microservices, they can improve just that one microservice, which is easier, less risky, and more affordable than retooling the entire system.
- **Reduced costs**. Developing for a single function instead of an entire platform helps keep resources in check.

- **Reusability**. Because microservices are so specific to a function, the function can be adapted for reuse reused in a whole different product, with just a few tweaks and without bringing the whole monolith environment along.

Of course, we've been experimenting with independent functions for some time. Only with the development of container and orchestration technology has it actually become truly feasible and more cost effective to implement a microservices approach.

But, let's tread lightly here. Just because we can all switch to microservices doesn't mean we should. Microservices tend to masquerade as a simple solution to a complicated problem. Unfortunately, the truth eventually comes out: microservices will reveal their true complexities, something that can be frustrating and difficult to solve.

Experienced developers point out that independent functionality is often a false idol: in the real world, it's unlikely that many components or functionalities can, actually, work so independently. Boundaries between components are a lot muddier than we think, relying on other pieces, needing to communicate among various functionalities. Unless you can very clearly define boundaries and separate pieces accordingly, you'll actually be working with interdependencies – something that your monolith system is already doing.

Then there's the issue of more is more: the more you compartmentalize functions, the more expertise is needed. That's because lots of things get more complex: running every service in a feature instead of executing a single program, operating and maintaining a lot more services, trying to ensure smooth communication, especially as communication paths multiply). With more comes more risk for failure, and in more places.

For non-DevOps companies, the maintenance team will likely have to manage dozens or hundreds of microservices, instead of just a single monolith or two. And if you are working in a DevOps culture, you'll need a person who understands and stays up to date with the nuances and rapid changes in container orchestration systems – which ensure all your microservices are up and running. (Not to mention how elusive that expert person may be.)

In a nutshell, microservices seem to simplify things, but consider all the underlying pieces that, in all likelihood, will become more complicated. Without serious expertise, componentization can just be messy. A good way to gauge your team's expertise: if a single monolith system isn't running smoothly, breaking them into smaller services probably won't make it any better.

Hybrid approach to software development

The current trend is to trade monolithic legacy systems for a bunch of independent but flexible microservices, which is certainly appealing. But, let's take the long view. Legacy systems seem to have garnered a bad reputation, albeit unwittingly. Legacy systems are so named at least in part because they lasted, they were part of a legacy, a positive thing.

Today, monolith systems remain the most widely used of various software architecture for good reason: they are easiest to develop. While they come with inherent issues, particularly the longer they're in use and the more developers who touch them, they are surprisingly simple to develop in the earlier stages of product. Quick development means faster time to market, something that start-ups and SMBs can get on board with them.

Instead of choosing a singular approach to software development, perhaps the answer is to

approach microservices as one tool in your developers' toolkits. Along with monolith systems and serverless apps, microservices are useful and appropriate for particular types of tasks and apps.

A future for microservices?

We're not ready to herald microservices as the silver bullet for software development, but we're equally unprepared to sound the death knell. Just as some monoliths age poorly, it's likely that some microservices will do the same. It's too soon to tell.

The bottom line is that breaking a system down into independent components is always risky because it requires very clear boundaries that are often difficult to discern in the real world. If you can't find this balance, you're simply relocating the complexity. And, experts caution, avoid change for the sake of change. Trends come and go, so if your processes are already working smoothly, opt to invest time and money into providing value where it is truly needed.

The likeliest bet for microservices? Turn to your team, their skills, and the product at hand. Good teams build good products with good tools, and the reverse can be said about bad teams.