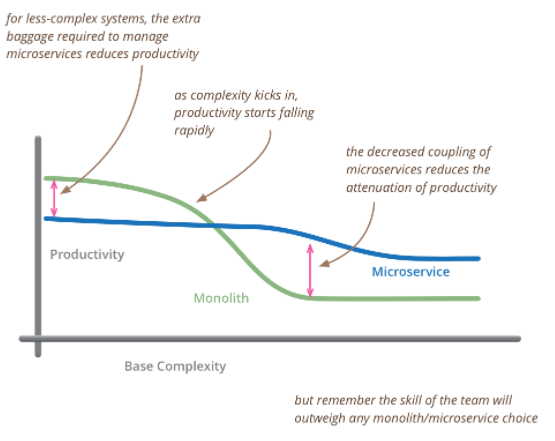
While this hype around microservices is annoying, I do think it's a useful bit of terminology for a style of architecture which has been around for a while, but needed a name to make it easier to talk about. The important thing here is not how annoyed you feel about the hype, but the architectural question it raises: **is a microservice architecture a good choice for the system you're working on?**

"It depends" must start my answer, but then I must shift the focus to what factors it depends *on*. The fulcrum of whether or not to use microservices is the complexity of the system you're contemplating. The microservices approach is all about handling a complex system, but in order to do so the approach introduces its own set of complexities. When you use microservices you have to work on automated deployment, monitoring, dealing with failure, eventual consistency, and other factors that a distributed system introduces. There are well-known ways to cope with all this, but it's extra effort, and nobody I know in software development seems to have acres of free time.



So my primary guideline would be **don't even consider microservices unless you have a system that's too complex to manage as a monolith**. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.

The complexity that drives us to microservices can come from many sources including dealing with large teams [2], multi-tenancy, supporting many user interaction models, allowing different business functions to evolve independently, and scaling. But the biggest factor is that of sheer size - people finding they have a monolith that's too big to modify and deploy.

At this point I feel a certain frustration. Many of the problems ascribed to monoliths aren't essential to that style. I've heard people say that you need to use microservices because it's impossible to do ContinuousDelivery with monoliths - yet there are plenty of organizations that succeed with a cookie-cutter deployment approach: Facebook and Etsy are two well-known examples.

I've also heard arguments that say that as a system increases in size, you have to use microservices in order to have parts that are easy to modify and replace. Yet there's no reason why you can't make a single monolith with well defined module boundaries. At least there's no reason in theory, in practice it seems too easy for module boundaries to be breached and monoliths to get tangled as well as large.

We should also remember that there's a substantial variation in service-size between different microservice systems. I've seen microservice systems vary from a team of 60 with 20 services to a team of 4 with 200 services.

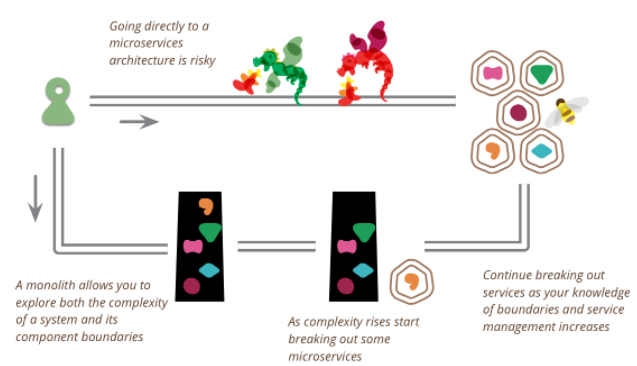
As size and other complexity boosters kick into a project I've seen many teams find that microservices are a better place to be. But unless you're faced with that complexity, remember that the microservices approach brings a high premium, one that can slow down your development considerably. So if you can keep your system simple enough to avoid the need for microservices: do.

**Monolith First**

As I hear stories about teams using a [microservices architecture](https://martinfowler.com/articles/microservices.html), I've noticed a common pattern.

1. Almost all the successful microservice stories have started with a monolith that got too big and was broken up
2. Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

This pattern has led many of my colleagues to argue that **you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile.**.



**The first reason** for this is classic [Yagni](https://martinfowler.com/bliki/Yagni.html). When you begin a new application, how sure are you that it will be useful to your users? It may be hard to scale a poorly designed but successful software system, but that's still a better place to be than its inverse. As we're now recognizing, often the best way to find out if a software idea is useful is to build a simplistic version of it and see how well it works out. During this first phase you need to prioritize speed (and thus cycle time for feedback), so the premium of microservices is a drag you should do without.

**The second issue** with starting with microservices is that they only work well if you come up with good, stable boundaries between the services - which is essentially the task of drawing up the right set of [BoundedContexts](https://martinfowler.com/bliki/BoundedContext.html). Any refactoring of functionality between services is much harder than it is in a monolith. But even experienced architects working in familiar domains have great difficulty getting boundaries right at the beginning. By building a monolith first, you can figure out what the right boundaries are, before a microservices design brushes a layer of treacle over them. It also gives you time to develop the [MicroservicePrerequisites](https://martinfowler.com/bliki/MicroservicePrerequisites.html) you need for finer-grained services.

I've heard different ways to execute a monolith-first strategy. The logical way is to design a monolith carefully, paying attention to modularity within the software, both at the API boundaries and how the data is stored. Do this well, and it's a relatively simple matter to make the shift to microservices. However I'd feel much more comfortable with this approach if I'd heard a decent number of stories where it worked out that way

A more common approach is to start with a monolith and gradually peel off microservices at the edges. Such an approach can leave a substantial monolith at the heart of the microservices architecture, but with most new development occurring in the microservices while the monolith is relatively quiescent.

Another common approach is to just replace the monolith entirely. Few people look at this as an approach to be proud of, yet there are advantages to building a monolith as a [SacrificialArchitecture](https://martinfowler.com/bliki/SacrificialArchitecture.html). Don't be afraid of building a monolith that you will discard, particularly if a monolith can get you to market quickly.

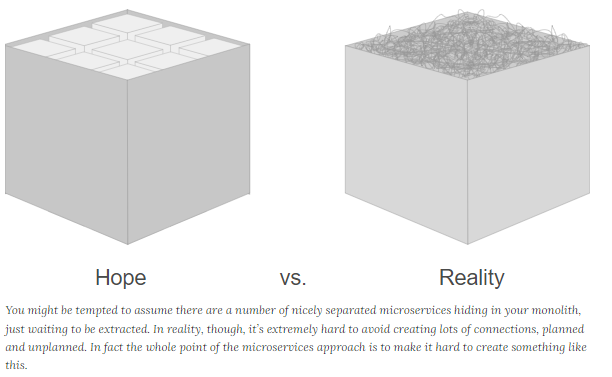
**Don’t start with a monolith** … when your goal is a microservices architecture

Starting to build a new system is exactly the time when you should be thinking about carving it up into pieces.

In the majority of cases, it will be awfully hard, if not outright impossible, to cut up an existing monolith this way. There is some common ground in that I agree you should know the domain you’re building a system for very well before trying to partition it, though: In my view, the ideal scenario is one where you’re building a *second version* of an existing system.

If you are actually able to build a well-structured monolith, you probably don’t need microservices in the first place. Which is OK! You shouldn’t introduce the complexity of additional distribution into your system if you don’t have a very good reason for doing so.

(So what would be a good reason? There are many, but to me the most important one is to allow for fast, independent delivery of individual parts within a larger system. Microservices’ main benefit, in my view, is enabling parallel development by establishing a hard-to-cross boundary between different parts of your system. By doing this, you make it hard – or at least harder – to do the wrong thing: Namely, connecting parts that shouldn’t be connected, and coupling those that need to be connected too tightly. In theory, you don’t need microservices for this if you simply have the discipline to follow clear rules and establish clear boundaries within your monolithic application; in practice, I’ve found this to be the case only very rarely.)



But if you start with a monolith, the parts will become extremely tightly coupled to each other. *That’s the very definition of a monolith*. The parts will rely on features of the platform they all use. They’ll communicate based on abstractions that are shared because they all use the same libraries. They’ll communicate using means that are only available when they are hosted in the same process. And these are only the technical aspects! Far worse than that, the parts will (almost) freely share domain objects, rely on the same, shared persistence model, assume database transactions are readily available so that there’s no need for compensation … Even the very fact that it’s easy to refactor things and move them around – all in the convenience of your IDE’s view of a single project – is what makes it extremely hard to cut things apart again. It’s *extremely* hard to split up an existing monolith into separate pieces.

I strongly believe – and experience from a number of our recent projects confirms this – that when you start out, you should think about the subsystems you build, *and build them as independently of each other as possible*. Of course you should only do this if you believe your system is large enough to warrant this. If it’s just you and one of your co-workers building something over the course of a few weeks, it’s entirely possible that you don’t.

But starting with an approach where you carve up your system into smaller parts, and treat each of them as a clearly separated, individual system with its own development, deployment, and delivery cycle, and (the possibility of) its own internal architecture, is a very powerful concept that can help you to deliver a system in the first place.

Beware of architectural recipes that are too simple and too obvious. This one – start by carving up your domain into separate, independent parts – is no exception. Sometimes a monolith *is* preferable, sometime it’s not. If you decide to build things using a microservices approach, you need to be aware that while it will be a lot easier to make localized decisions in each individual part, it will be much harder to change the very boundaries that enable this. Refactoring in the small becomes easier, refactoring in the large becomes much harder.

