### When you are building a modern software application:

* You want it to have a good performance.
* Resilient to errors
* Should have a flexible architecture which can adapt to new requirements
* To keep the data secure

Micro-services is an architectural style and it has the potential to be very flexible and maintainable in the modern distributed software applications.

**What are Micro-Services?**

It is an architectural style, where autonomous, independently deployable services collaborate together to form a software application or system.

The name “micro-services” also suggest that these services should be small, but there isn’t an official size limit. Some team try to keep them extremely small, may be just a couple of hundred lines of code but many of the benefits of micro-services still apply even if they’re a bit larger.

**Why do we need Micro-Services?**

To answer this question, we need to understand, what is a “Monoliths” Application?

A Monolith is a software application that typically has all of its code in a **single code** base and which all developers collaborate together on that same source code repository. The build artifact is actually a **single executable or process** that runs on a **single** **host** server or virtual machine and persist all of its data into a **single database**. And the development environment typically uses a single **consistent technology**, such as what programming language or SDK you’re using throughout the code base.

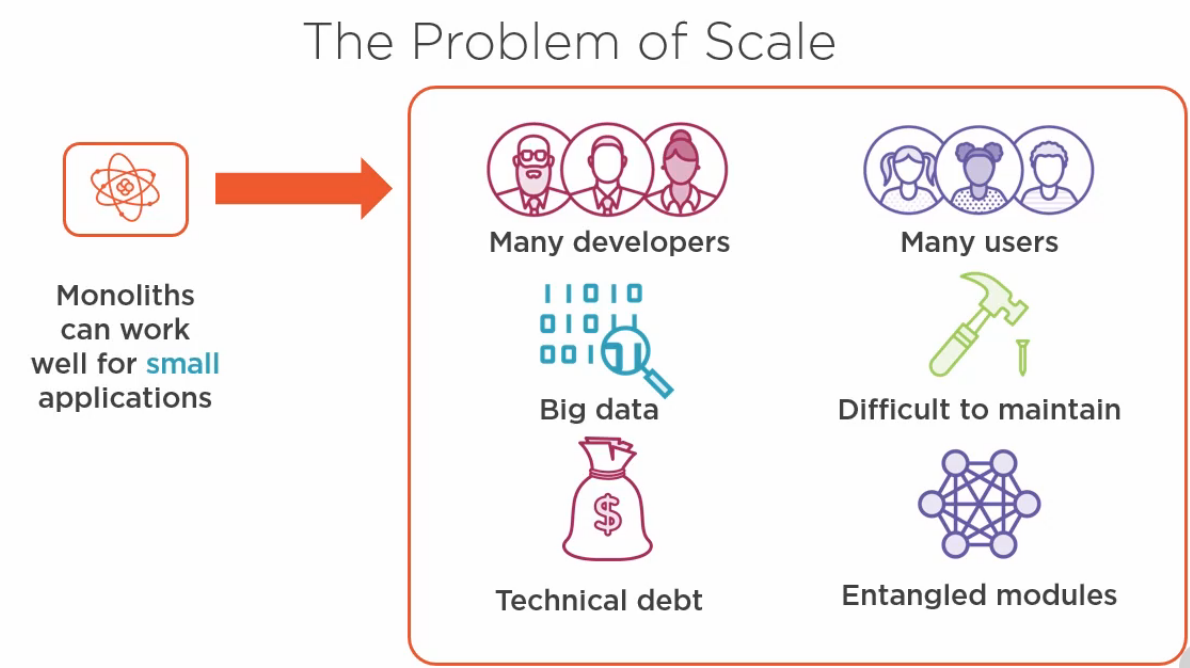
**The Major benefits of this approach is simplicity.** All the code is in one place, so it’s easy to find things. There is only one thing that you have to build and run, so it’s straight forward for a developer to work on. And when you come to deploy it to production, well, there’s just one application to update. And there’s nothing inherently wrong with this approach, and if it’s working well for you, that’s probably a sign that you don’t need micro-services. Or perhaps you don’t need micro-services yet.

Monolith Benefits:

* Simplicity
* One code base
  + Easy to find things
* Deployment
  + One application to replace
* Monoliths are not “wrong”

**The problem with monoliths is scaling.** The monolith model works well for single developer or a small team, or to handle a small website. But what happens when our project grows much larger? What if we have 20-30 developers? And they are working for a period of several years building a system that will have thousands of users and store massive amounts of data? When we get into this kind of situation, the monolith approach becomes a big problem.

For one thing, as a code base grows larger, there’s a tendency for it to become more difficult to maintain due to growing complexity and the accumulation of technical debt. And even if you’ve made a coordinated effort to keep your monolith modularized, often you’ll find that these modules end up becoming very entangled and inter-dependent

**Monolith Problems:**

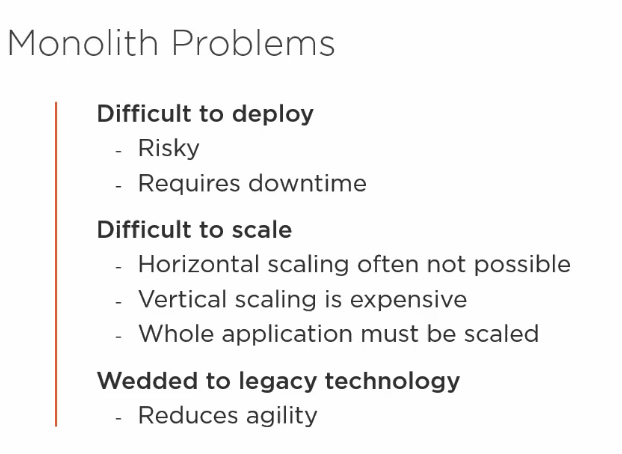
When it comes time to deploy your monolith, even a **single line code change requires the entire application to be deployed**, which is **risky** and usually **involves a period of downtime** (something that’s becoming less acceptable in the modern era of cloud services that are expected to maintain high availability).

Scaling a monolith to meet demands of increased users or large amounts of data is also very difficult. Unless great care has been taken to make your monolith stateless, you probably can’t scale it out horizontally. That’s where you add additional servers. And so you have only option to scale vertically where you provision much more powerful and expensive servers.

[Horizontal scaling means that you scale by adding more machines. Vertical scaling means that you scale by adding more power (CPU, RAM) to an existing machine]

And monolith require the entire application to be scaled out together rather than just scaling the individual components that require additional processing power.

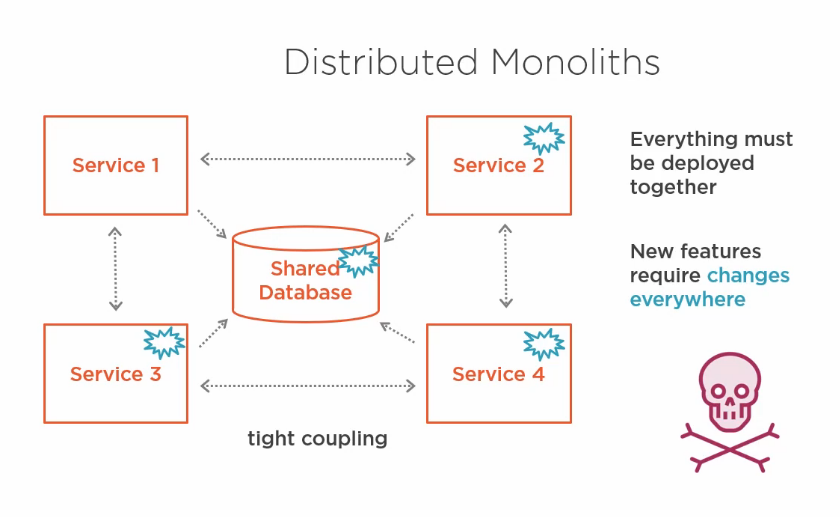
Finally, with monolith, you can very easily find yourself wedded to legacy technology. Whatever stack you built the original version with is going to be very hard to get away from as you have to upgrade the entire application to move to a new framework. Which reduces your agility to adopt newer patterns and practices or to take advantages of innovation such as new tools and services that would benefit your application.



May be you’ll think that your application is not a monolith. It’s made up of several services that can run on different hosts and talk to each other over the network. However, it’s very important to understand that just because you have an architecture based on services doesn’t mean you’re using micro-services.

It possible to create a system where you’ll have several services, but they all access data in shared database, and they’re all tightly coupled to each other in such a way that you have to deploy them all together. And any change you want to make to the system requires modifications to multiple services. Systems like this are sometimes referred to as **distributed monoliths**, and these things are big trouble. They combine all the downsides of monoliths with all the challenges of micro-services and offer very few of the benefits of either approach.

So, it is important to have a very clear understanding of micro-services before diving into creating them.

**The Benefits of Micro-Services**

By break our application into smaller pieces, each micro-service can be owned by a small development team and it’s much easier to understand and work on. A good rule of thumb for the size of micro-services is that it’s small enough to be throw away and rewritten if necessary.

Micro-services allow us the freedom to adopt new technologies without needing to upgrade everything at once. And the flexibility to choose the right tool for the job. For example one micro-service might use a RDBMS to store its data while another uses a document database.

One micro-services might be written in a functional programming language while another might use an object-oriented approach. Of course, there are still benefits in using standardized approaches where possible, but there’s no hard requirements. Individual micro-service development teams have the freedom to use the most appropriate technology.

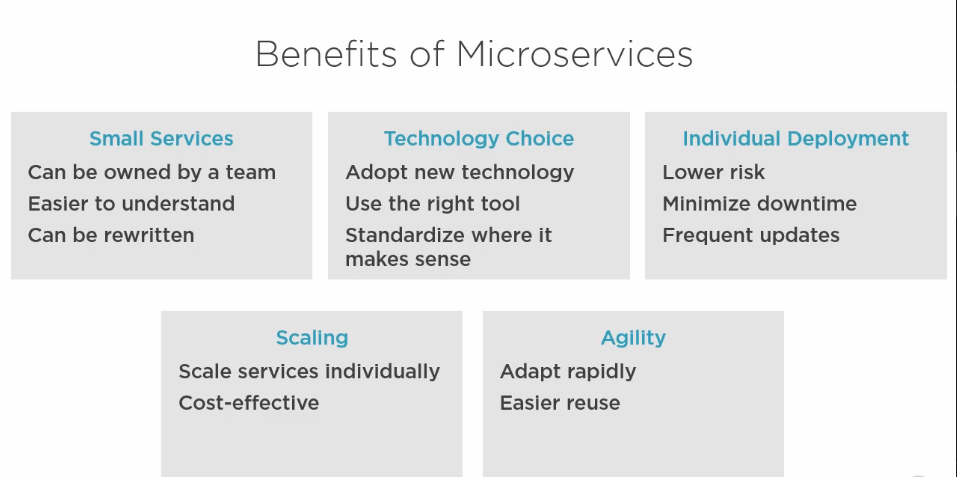
When we create micro-services, their loose coupling means we can deploy them individually, and this is a huge benefit since it’s much lower risk than upgrading everything in the entire system in one go, and it also helps us to achieve zero downtime deployments. And that’s because while we’re upgrading an individual micro-service, it’s quite possible for the rest of the system to carry on running.

Micro-service architectures enable teams to upgrade with much greater frequency. Rather than waiting months between upgrades, as is often common with monoliths, changes to individual micro-services can be pushed to production as soon as they’re ready, and it’s even possible to deploy micro-services multiple times a day.

Micro-services give us much more control over scalability. We can scale out each micro-service individually, which is much more cost-effective than scaling a monolith.

Micro-services makes us more agile. We can adapt more rapidly to changing business requirements. By keeping our micro-services lightweight and decoupled, they’re more easily adaptable to use in new scenarios.

Micro-Services are getting popular, particularly in large cloud deployed system.



**The Challenges of Micro-Services**

One of the first challenges you’ll face is:

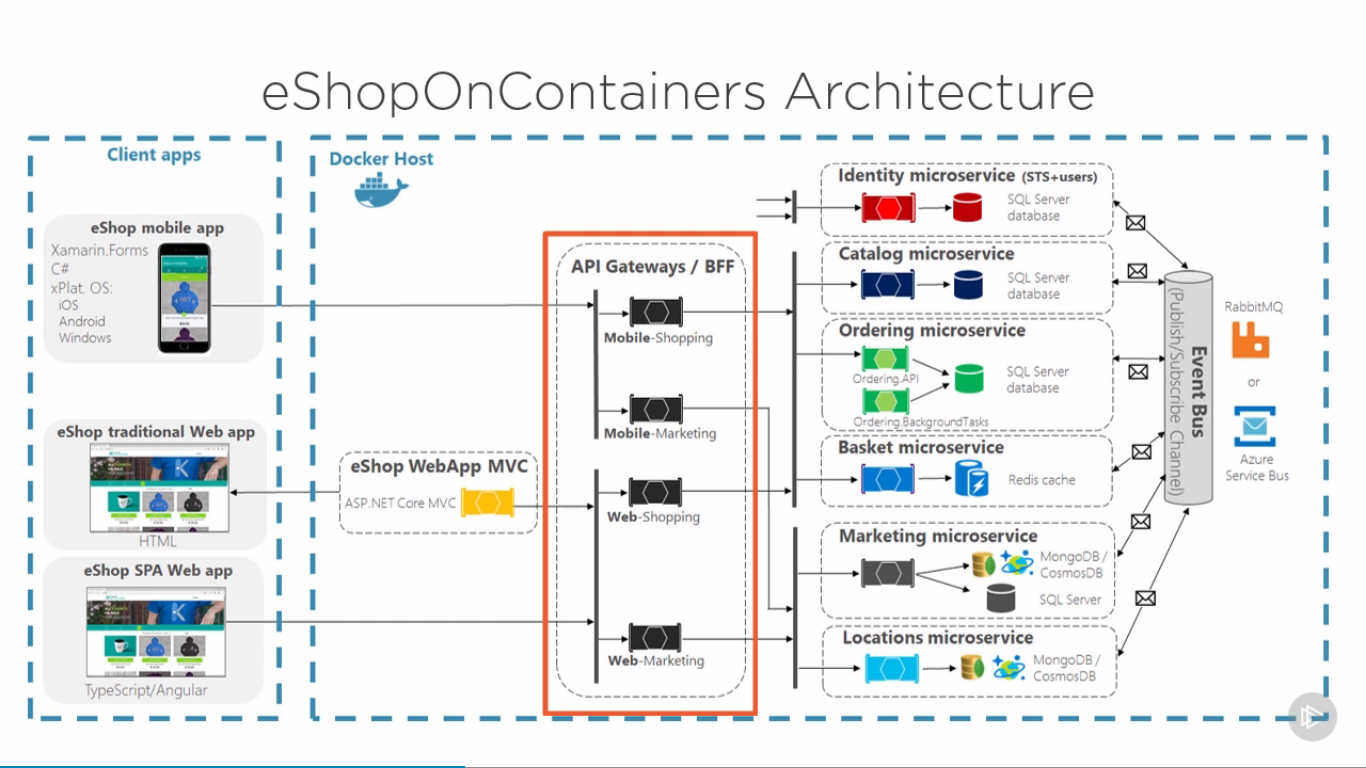
1- How can you ensure developers are **productive, working** on the system? Do they have to download the code for every single micro-service and run it individually and configure them all to talk to each other? That could be very error prone and time consuming. You also need to have a way of working that enables developers to easily work on an individual micro-service in isolation, as well as testing their micro-service in the context of the whole application.

2- If we break an application up into dozens of micro-services, then there’s potential for the interaction between those **micro-services to get very complex,** making it hard to understand the behavior of the system as a whole, and it can lead to performance issues. If you don’t take care, you can find that a single operation requires lots of inefficient, verbose communications between many micro-services.

3- While deploying an individual micro-service is a smaller and safer task than deploying a monolith, when you have many micro-services, automated **deployment becomes critical** because you’re going to have many more deployments to do.

4- If there’s a problem in production, we don’t want to have to connect individually to every single micro-services to examine its logs and work out which one is the cause of our trouble. So, it’s essential that you have a great **monitoring solution**, enabling you to view logs and telemetry(process of recording) for all of your micro-services in a centralized place.





**Micro-Services Gives You Choice:**

Micro-Service don’t dictate what programming language you use, what type of database you use, whether you communicate synchronously or asynchronously, or what authentication mechanism you’re going to use.

It means you are completely free to pick the ones that you’re most familiar with and meet your needs.

