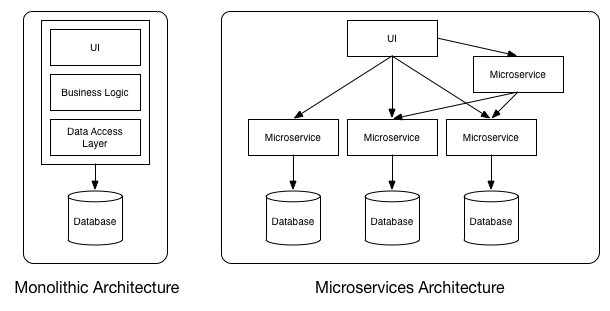
**1. Definition of Microservices**

Microservices is **a service-oriented architecture pattern wherein applications are built as a collection of various smallest independent service units**. It is a software engineering approach that focuses on decomposing an application into single-function modules with well-defined interfaces.

Today, microservices are one of the increasingly popular architecture patterns next to SOA ([Services Oriented Architecture](https://en.wikipedia.org/wiki/Service-oriented_architecture)). If you are following industry trends, then you realize that today business houses are no longer interested in developing large applications to manage their end-to-end business functions as they did a few years ago, rather they opt for quick and agile applications which cost them less money as well.

Microservices help in breaking the boundaries of large applications and build logically independent smaller systems inside the system. E.g. using [Amazon AWS](https://aws.amazon.com/) you can build a cloud application with minimum effort. It’s a good example of what microservices can do.



Monolithic vs MicroServices Architecture

As you can see in the above diagram, each microservice has it’s own business layer and database. By doing so, changes to one microservice do not impact others.

In general, **microservices communicate with each other using widely adopted lightweight protocols**, such as HTTP and REST, or messaging protocols, such as [JMS](https://howtodoinjava.com/jms/jms-java-message-service-tutorial/) or AMQP. In specific scenarios, they can go for more specialized protocols as well.

**service-oriented architecture (SOA) has an enterprise scope, while the microservices architecture has an application scope**

## Principles of Microservices

Now let’s examine the “must-have” principles of a microservice.

#### Single responsibility principle

The single responsibility principle is one of the principles defined as part of the [SOLID design pattern](https://howtodoinjava.com/best-practices/solid-principles/#SRP). It implies that a unit, either a class, a function, or a microservice, should have one and only one responsibility.

At no point in time, one microservice should have more than one responsibility.

#### Built around business capabilities

**Microservices should focus on certain business functions** and ensure that it helps in getting things done. A microservice shall never restrict itself from adopting appropriate technology stack or backend database storage which is most suitable for solving the business purpose.

This is often the constraint when we design monolithic applications where we try to solve multiple business solutions with some compromises in some areas. Microservices enable you to choose what’s best for the problem at hand.

#### You build it, you own it!

Another important aspect of such design is related to responsibilities pre-and-post development. In a large organization, usually one team develops the app location, and after some knowledge transfer sessions it hand over the project to the maintenance team. In microservices, the team which builds the service – owns it, and is responsible for maintaining it in the future.

[You build it, you own it !!](https://aronatkins.github.io/2014/12/23/you-build-it-you-own-it.html)

This ownership brings developers into contact with the day-to-day operation of their software and they better understand how their built product is used by customers in the real world.

#### Infrastructure Automation

Preparing and building infrastructure for microservices is another very important need. **A service shall be independently deployable** and shall bundle all dependencies, including library dependencies, and even execution environments such as web servers and containers or virtual machines that abstract physical resources.

One of the major **differences between microservices and SOA** is in their level of autonomy. While most SOA implementations provide service-level abstraction, microservices go further and abstract the realization and execution environment.

In traditional application developments, we build a WAR or an EAR, then deploy it into a JEE application server, such as with JBoss, WebLogic, WebSphere, and so on. We may deploy multiple applications into the same JEE container. In an ideal scenario, in the microservices approach, each microservice will be built as a [fat Jar](https://howtodoinjava.com/maven/maven-shade-plugin-create-uberfat-jar-example/), embedding all dependencies and run as a standalone Java process.

#### Design for Failure

A microservice shall be designed with failure cases in mind. What if the service fails, or go down for some time. These are very important questions and must be solved before actual coding starts – to clearly estimate **how service failures will affect the user experience**.

Fail fast is another concept used to build fault-tolerant, resilient systems. This philosophy advocates systems that expect failures versus building systems that never fail. Since services can fail at any time, it’s important to be able to detect the failures quickly and, if possible, automatically restore service.

Microservice applications put a lot of **emphasis on real-time monitoring** of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received). Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.

**DDD:**

Domain-Driven Development allows us to plan a microservice architecture by decomposing the larger system into self-contained units, understanding the responsibilities of each, and identifying their relationships

DDD patterns **help you understand the complexity in the domain**. For the domain model for each Bounded Context, you identify and define the entities, value objects, and aggregates that model your domain. You build and refine a domain model that is contained within a boundary that defines your context

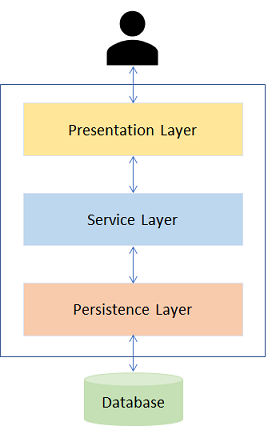
**Example:**

Consider an online shopping application with the following functionalities:

1. Search products
2. Place order
3. View all orders

The most common approach to develop this application is to implement all these functionalities in a single application. This application is then deployed as one application for all different types of users. This type of architecture where all the functionalities are implemented in one single application is called as **Monolithic Architecture**. In this architecture the application is divided into different layers. Some common layers in this architecture are as follows:

* **Presentation Layer** — It responsible for handling HTTP requests and sending respose.
* **Service Layer** — In this layer business logic of the application is implemented.
* **Persistence layer** — In this layer logic for accessing the database is implemented.



The applications developed using this architecture are easy to develop, test and deploy. However, there are certain issues that tag along with monolithic applications. As the application becomes larger and complex, we will face difficulties in the following aspects:

1. Deployment will take a long time.
2. Scalability is an issue as it is not possible to scale any single functionality alone. For example, the 'Search' functionality maybe used more often than the 'Book' functionality. Hence, we would like to scale the 'Search' functionality and not the 'Book' functionality. This is not possible when we go with the monolithic approach.
3. Failure of a single functionality will lead to failure of entire application in monolithic architecture.
4. New technologies or frameworks cannot be used in the existing application. If new technology is needed, then complete re-write must be done.
5. It is not very reliable as a single bug in any module can bring down the entire application.

How do we resolve these issues?

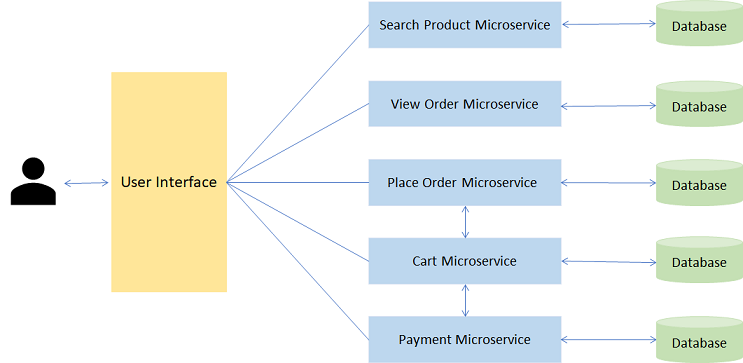
The answer is **Microservices**. It is another architecture of application development. Now let us explore it in detail.

We have learned that in monolithic architecure all the fuctionalities are implemented in one single application and then this application is deployed. But it has some drawbacks which can be addressed if we develop the application using another architecure called as **microservices**.  In this architecture we split the application into a set of smaller services instead of building a single monolithic application. Each of these services implements specific business requirement and can communicate with each other through well defined interfaces. These services have their own database and can be deployed independently.

Consider the online shopping application which we have developed using monolithic architecure. Now if you have do develop same application using microservices, then you need to split the business requirements or functonalities into several services communicating with each other and having its own database. You can have following services for this application:

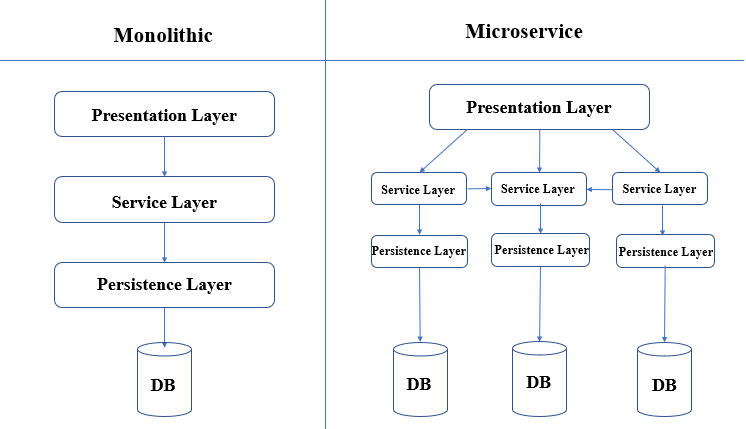
* Search Product Microservice - Responsible for seaching a product.
* View Order Microservice - View orders placed by customer.
* Place Order Microservice - Takes an order and process it.
* Cart Microservice - Manage user cart, this service can utilize Catalog service as a data source.
* Payment Microservice - Manage payments.

Each of these microservices also interact with each other to transfer data which is required by another microservice. For example, Place Order microservice will interact with Card and Order Microservice.



Now that we have seen a real-life example of Microservice, let us understand why we should choose Microservices for developing enterprise applications.

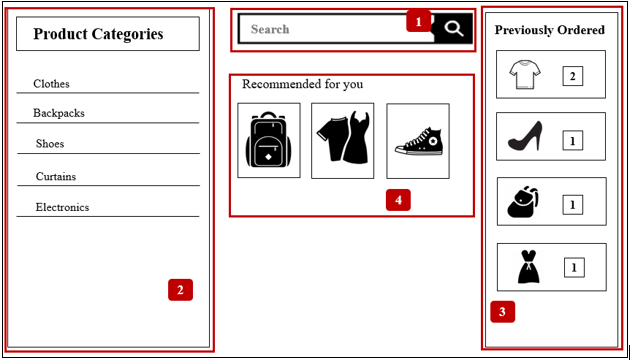
You have learnt the architectural differences between Monolithic Applications and Microservices. Now let us see how functionally wise they are different.



|  |  |  |
| --- | --- | --- |
| **Parameter** | **Monolithic** | **Microservice** |
| Development | Needs to be developed in a single language | Each service can be written in different languages |
| Testing | For any change, the entire application needs to be tested again | Only the modified service needs to be tested again |
| Runtime | Monolithic application runs as a single process | Each service runs its own process |
| Packaging | Packaged as a single JAR/WAR file | Each service is packaged as single JAR/WAR file |
| Scalability | Entire application needs to be scaled or replicated on multiple servers | Only the service that requires scaling, can be scaled |
| Minor Changes | For any modification. entire application needs to be re-built and re-deployed | Only the modified service needs to be re-built and re-deployed |

Now we will understand more about Microservices with an example.

An online shopping application can be divided into a number of microservices based on the functionality and business requirements.



This application can be divided into the following microservices:

1. **Product Microservice**: This microservice can be used to help the User search for any specific product that they are looking for.
2. **Product Categories Microservice**: This microservice shows all product categories available to the User.
3. **Orders Microservice**: This Microservice is responsible for placing orders and the User can view the order history as well.
4. **Recommended Products Microservice**: This microservice shows a list of recommended products to the User based on purchase history.

Now that we have seen a real-life example of Microservice, let us understand why we should choose Microservices for developing enterprise applications.

Top reasons for choosing Microservice

The following are the top reasons for choosing Microservice architecture:

* **Easy deployment**: A large code base will make the IDE slow and build time increases. In Microservice architecture, each project is independent and small in size. So overall build and development time gets reduced.
* **Small teams**: Microservices have single-purpose design, which means they can be built and maintained by smaller teams. Moreover, each team can be cross-functional i.e each team can have their own developers, testers and operations team and can be responsible for a single microservice.
* **Loose coupling**: It extremely difficult to change technology or language or framework when everything is tightly coupled and dependent on each other. In microservices, it is easy to change technology or framework because every module and project is independent and loosely coupled.
* **Domain-driven design**: Although microservices are supposed to be small in size, it is not the only criteria. The services should have well-defined boundaries centered around business requirements. Domain-driven design, helps design software systems based on the underlying model of the business domain, which is the requirement of microservices.

As we have seen, there are multiple reasons for choosing Microservice architecture. The following are the scenarios in which microservices can be used.

1. Migrating a monolithic application due to improvements required in scalability, manageability and agility.
2. Rewriting a heavily used legacy application. (A legacy application is a software program that is outdated or obsolete.)
3. Highly agile applications due to domain-driven design of microservices.
4. Applications that demand speed of delivery as microservices are easy to build and maintain.
5. New product development where a new product is conceived and brought to market.

**Does this mean we should use microservice architecture always?**

 The answer is No.

If we look at the evolution of software, we started with unstructured code. Then we moved on to packages for codes, which slowly moved on to creating libraries for reuse. Then we had multiple applications and multiple libraries which had services added to them.

**So, when should we choose Microservices and when should we choose Monolithic applications**

We should choose Monolithic when:

1. We have little knowledge of the market
2. The application we are developing is small

We should choose Microservices when:

1. We have more knowledge of the market
2. The application being developed is a large enterprise application and needs to be highly scalable.

There are some advantages of using microservices.

1. **Scalability**: It is easier to scale only the required service based on demand.
2. **Fault Isolation**: Failure in one service does not affect the working of other services. So the entire application does not fail due to failure of single functionality.
3. **Speed of deployment**: As services are small, it can be built and deployed faster.
4. **Freedom of technology**: Each service can be written in different technology, using different frameworks based on requirement.
5. **Autonomous teams**: Microservices grant the developers more independence to work autonomously and make technical decisions quickly in smaller groups.

Although microservices have advantages, there are a few disadvantages as well. The following are the disadvantages of using microservices.

1. **Performance**: As we have a more distributed and complex application, performance could be affected.
2. **Maintenance**: An application can have hundreds of microservices and maintaining all of them can be a challenge. This can be overcome by de-centralized management.
3. **Infrastructure**: Initially, microservices require a huge infrastructure setup.
4. **Cost**: Due to huge infrastructure setup huge cost may also be incurred. Also, services need to communicate with each other. This can increase network latency and processing costs.

**Dcoker:**

Docker is an open platform for developing, shipping, and running applications. You can develop applications very fast and deploy them fast. **Using Docker, it is easy to create required services separately and manage them as microservices without affecting other services**.

With Docker, **you can make your application independent of the host environment**. Since you have microservices architecture, you can now encapsulate each of them in Docker containers. Docker containers are lightweight, resource isolated environments through whch you can build, maintain, ship and deploy your application.

# Docker & The Rise Of MicroServices

Are you a software engineer/developer/coder or maybe even a tech enthusiast who has been hearing of terms like Microservices and Docker but still clueless as to why there is so much hype around it?

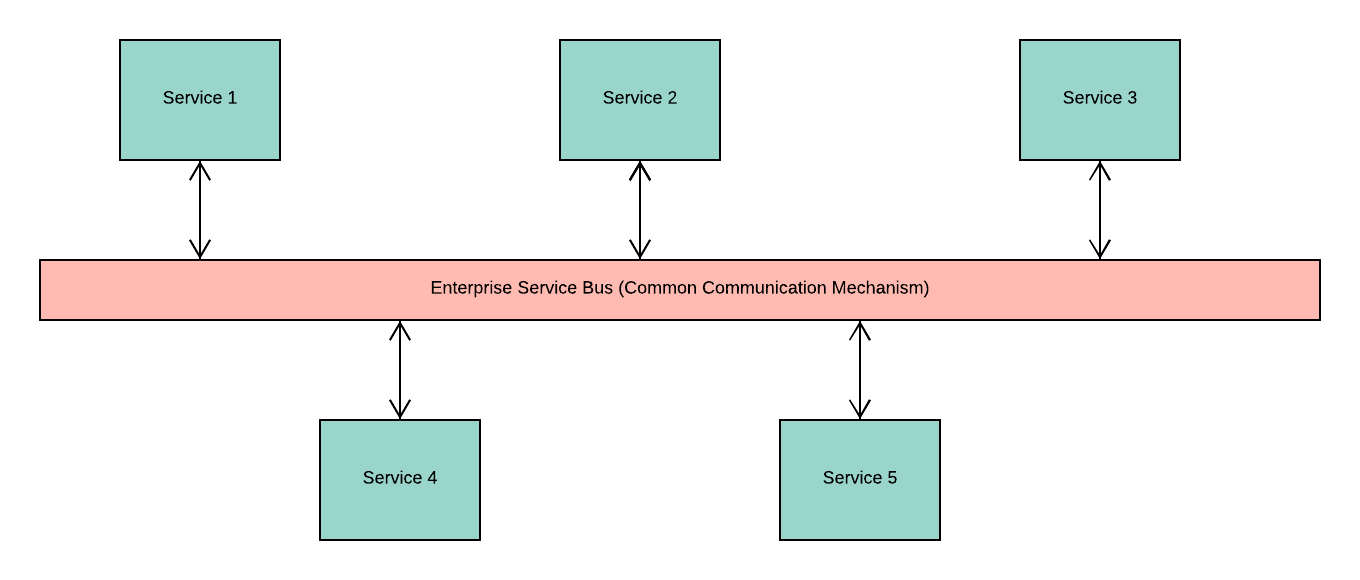
If yes, this is a must-read for you!

This is a guest post brought to you by your friends @ Timber. If you're interested in writing for us, reach out [on Twitter](https://twitter.com/timberdotio).

## [A Little Bit Of "History"](https://timber.io/blog/docker-and-the-rise-of-microservices/" \l "a-little-bit-of-history-)

In the early 2000's, we witnessed the rise of Service Oriented Architecture (SOA), a popular design paradigm for building software. In simple words, SOA is a software architecture pattern that allows us to construct large-scale enterprise applications that generally require us to integrate multiple services, each of which is made over different platforms and languages through a common communication mechanism.

Here is a simple pictorial representation of the Service Oriented Architecture (SOA):



**Key Points**

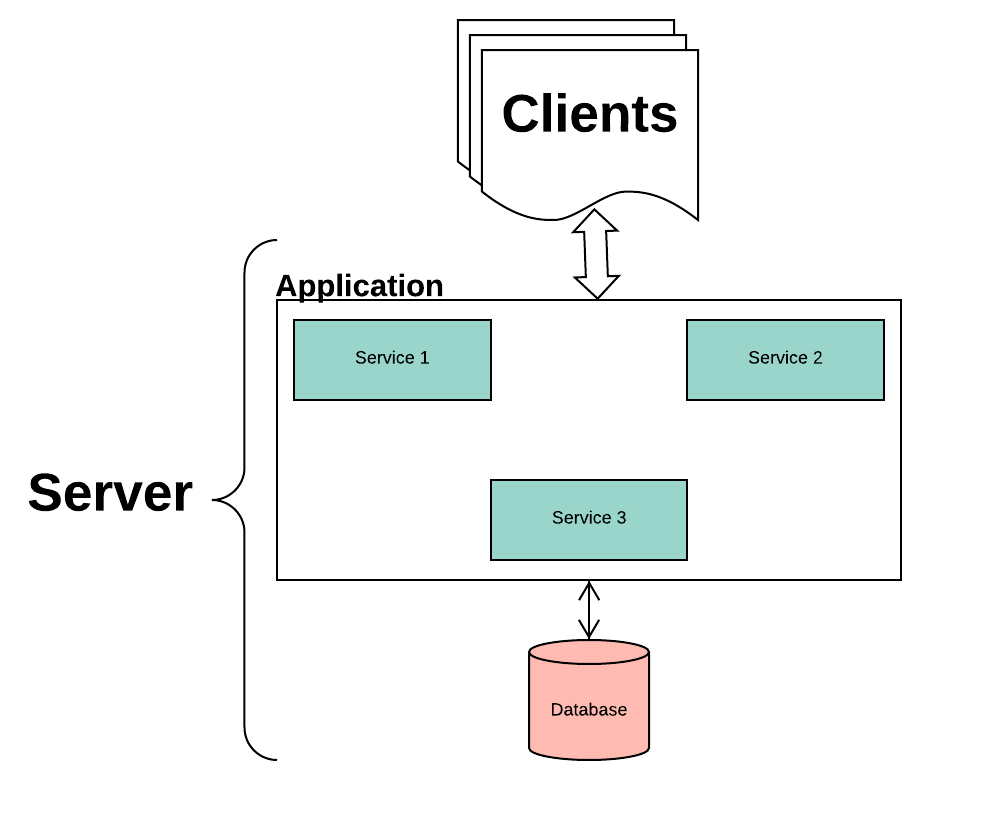
1. SOA is preferred for large-scale software products such as enterprise applications
2. SOA focuses on integrating multiple services in a single application rather than emphasizing on modularizing the application
3. The common communication mechanism used for interaction between various services in an SOA is referred to as [Enterprise Service Bus (ESB)](https://en.wikipedia.org/wiki/Enterprise_service_bus)
4. SOA based applications could be monolithic in nature. This means that you have single application layer which contains your user interface or presentation layer, business logic or application layer, and database layer all integrated into a single platform

## [Interesting, Tell Me More About "Monolithic Architecture"](https://timber.io/blog/docker-and-the-rise-of-microservices/" \l "interesting-tell-me-more-about-monolithic-architecture-)

Let's take a look at a case study: an ecommerce store. We know that multiple devices can access most ecommerce sites, so they have various user interfaces for laptop and mobile views.

We also know that multiple operations or services are running with each other to ensure the regular functioning of your ecommerce applications. Some of these services are account creation, displaying product catalog, building and validating your shopping cart, generating bill, order confirmation, payment mechanism and so on.

In a monolithic application, all these services run under a single application layer so the ecommerce software architecture would look like this:



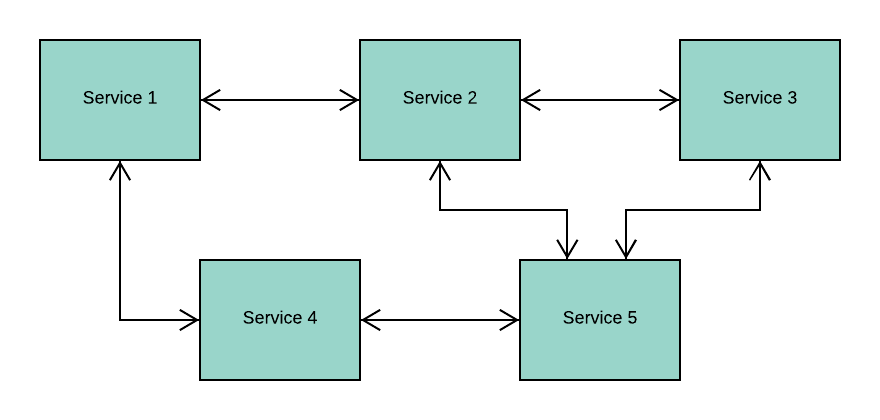
**Drawbacks**

1. It is evident that the application is going to grow in size with the increase in the number of services offered. This might become overwhelming for developers to build and maintain the application codebase
2. Not only is it difficult to update your current stack, but it is a nightmare to change something in that stack
3. Every change requires developers to rebuild the entirety of the application, which wastes resources.
4. With the increase in the customer base, we will have more requests to process, which will require more resources. Therefore, it is essential to build products that can scale. With monolithic applications, we can scale only in one direction, i.e., vertically but not horizontally. This means we can scale the application over a single machine by adding more hardware resources such as memory and computational power, but it is still going to be a challenge to ensure horizontal scaling, which is spread across multiple machines

## ["Microservices" To The Rescue!](https://timber.io/blog/docker-and-the-rise-of-microservices/" \l "-microservices-to-the-rescue-)

Microservice architecture can be considered to be a specialization of SOA and an alternative pattern that overcomes the drawbacks of a monolithic architecture.

In this architecture, we focus on modularizing the application by dividing it into smaller standalone services that can be built, deployed, scaled and even maintained independently of other existing services or the application itself as a whole. These independent services are called microservices, hence the name Microservice Architecture.



**Highlights**

1. Microservices Architecture and SOA are not the same, but they do hold some similarities. Microservice Architecture is referred to as a variant of SOA or even a specialization of SOA. In other words, SOA can be considered to be a superset of Microservices Architecture
2. The main reason why people find similarity between these architectures is because both of them focus on building loosely coupled services for an application. These services have clear boundaries and separate, well-defined functionalities set for each one of them
3. The difference lies in the fact that SOA can mean a lot of other things. For instance, SOA can be applicable over a Monolithic Architecture as well where the focus is to integrate systems together in an application and ensure code reusability. This does not hold true for a Microservice Architecture, where the focus is to modularize the application by building independent services and ensuring scalability of the product

**Advantages**

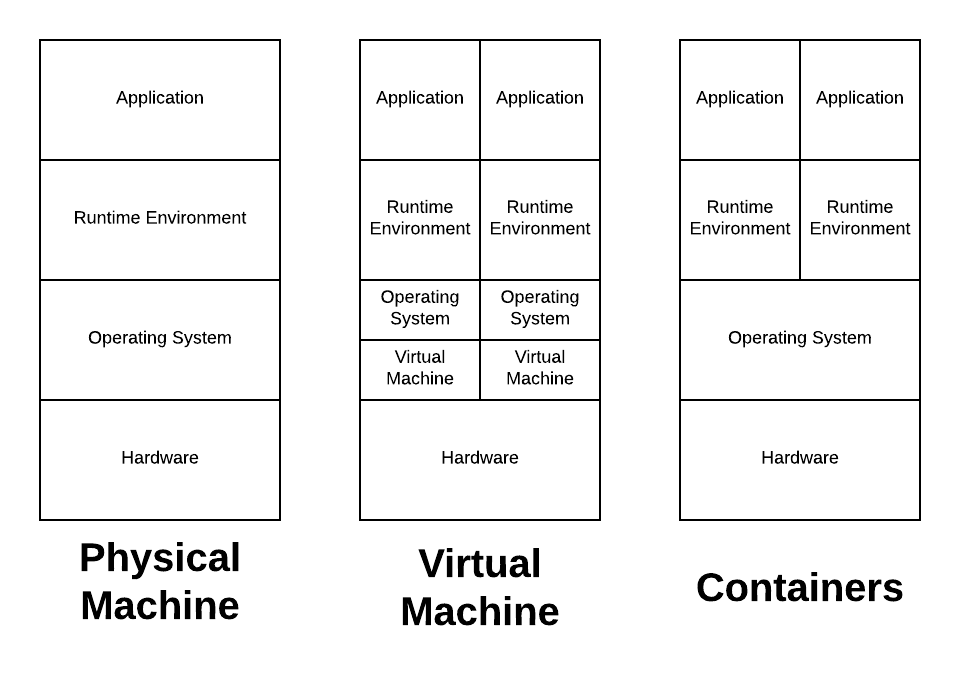
1. Introduces the philosophy of Separation of Concerns and ensures Agile Development of software applications in both simple and complex domains
2. The standalone ability or independent nature of microservices open doors for following benefits:
   * Reduces complexity by allowing developers to break into small teams, each of which builds/maintains one or more services
   * Reduces risk by allowing deployment in chunks rather than rebuilding the whole application for every change
   * Easy maintenance by allowing flexibility to incrementally update/upgrade the technology stack for one or more services, rather than the entire application in a single point in time
   * In addition to giving you the flexibility to build services in any language, thereby making it language independent, it also allows you to maintain separate data models of each of the given services
3. You can build a fully automated deployment mechanism for ensuring individual service deployments, service management and autoscaling of the application

## [Evolution Of Technologies](https://timber.io/blog/docker-and-the-rise-of-microservices/" \l "evolution-of-technologies)

Alongside, the evolution of software architectural patterns, we have also seen an emergence of some new technologies such as Docker and Kubernetes for supporting our software infrastructures and ensuring efficient management of our scalable products and services. We have evolved from using hardware virtualization to containerization.

Wait, you might be thinking, what does this even mean?

Now, to understand how we have evolved in the IT infrastructure space, let us take the help of the following diagram:



The first picture shows a physical machine or a hardware server. Typically, when we build applications, we use the resources provided by our host OS and the same pattern used to be followed when deploying the application. But what if you want to scale the application? At some point, you might want another hardware server and as the number keeps increasing, so does your cost and other resources like hardware and energy consumptions.

Also, you might be thinking if you require all the resources of your hardware and host OS at the same time for running your application. Not really, so then why such a massive infrastructure?

This has led to the evolution of hardware virtualization for optimizing IT infrastructure setups through what we call today as Virtual Machines (VMs). As you see in the second diagram, VMs have their guest OS which is run over a single physical machine or host OS. This enables us to run multiple applications without needing installing numerous physical machines. The host OS can ensure that there are systematic resource distribution and load balancing between the different VMs running on it.

Although VMs made software more accessible to maintain and drastically reduced costs, more optimization was still possible. For instance, not all applications would behave as expected in a guest OS environment. Additionally, the guest OS would require a lot of resources for even running simple processes.

These problems led to the next innovation: containerization. Unlike virtual machines which were more operating system specific, containers are application specific, making them far lighter. Furthermore, VMs can run multiple processes whereas a container runs as a single process. This leads us to two things:

1. You can run multiple containers on a physical machine, or you can even think of running it on a single VM. In either case, it solves your application related problems
2. Containerization is not in competition with Virtualization, but rather a complementary factor to further optimize your IT software infrastructure

## [Docker](https://timber.io/blog/docker-and-the-rise-of-microservices/" \l "docker)

Now that we understand the evolution of IT software infrastructure, we might want to know how we can achieve things like Microservices Architecture and Containerization that we discussed earlier? A simple answer to this could be, Docker.

[Docker](https://www.docker.com/) is the world's leading software containerization platform. It encapsulates your microservice into what we call as Docker container which can then be independently maintained and deployed. Each of these containers will be responsible for one specific business functionality.

To understand Docker in a little more depth, let us take the same example of an ecommerce website that we discussed earlier. We know that it will have multiple operations and services such as account creation, displaying product catalog, building and validation shopping cart and so on. In a microservice architecture, all these can be treated as microservices and encapsulated in a Docker container. But why would you do that?

One of the reasons would be to ensure consistency between development and production environment. For instance, consider that three developers are working on this application. Each of them having their environments. So one developer might be running Windows OS on his machine, while the second developer could be running Mac OS and the third developer would prefer a Linux based OS. Each of them would take hours of efforts to install the application in their respective development environments and an additional set of efforts to later deploy the same app on the cloud. This is not smooth, and there is a lot of friction to port such applications to cloud infrastructure.

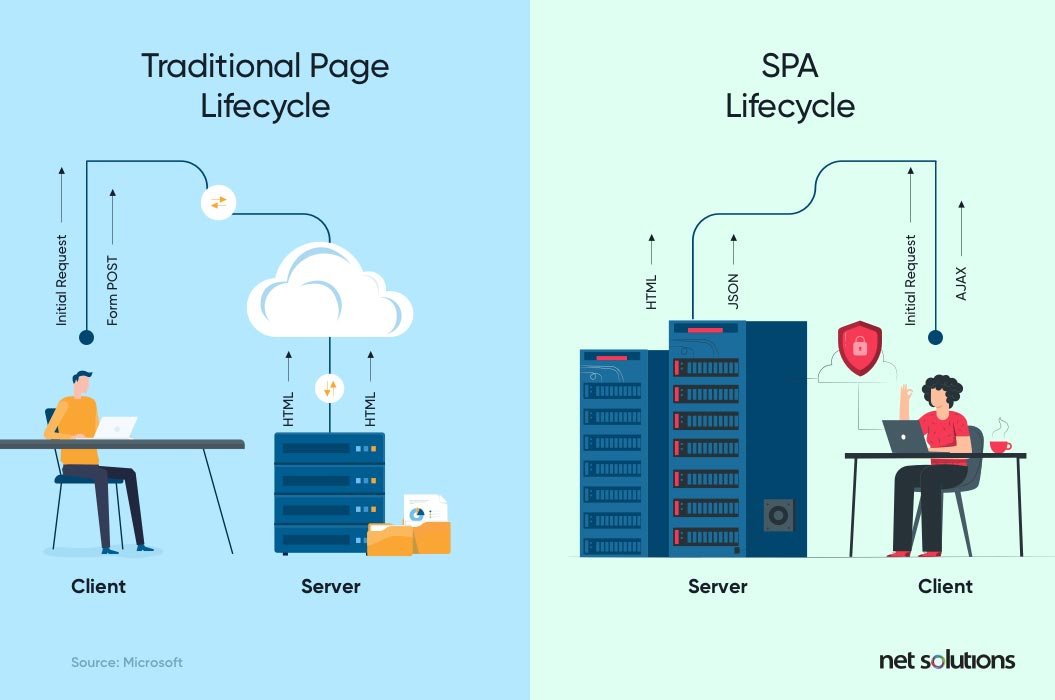
With Docker, you can make your application independent of the host environment. Since you have microservices architecture, you can now encapsulate each of them in Docker containers. Docker containers are lightweight, resource isolated environments through which you can build, maintain, ship and deploy your application.

**Advantages**

1. Docker is a popular evolving software with excellent community support and built for microservices
2. It is lightweight when compared to VMs making it cost and resource effective
3. It provides uniformity across development and production environments making it a suitable fit for building cloud-native applications
4. It provides facilities for continuous integration and deployment
5. Docker is not going anywhere; it provides integration with popular tools and services such as AWS, Microsoft Azure, Ansible, Kubernetes, Istio and lot more

SPA:

A single-page application is an app that doesn't need to reload the page during its use and works within a browser. Think of the apps you use daily: **Facebook, Google Maps, Gmail, Twitter, Google Drive, or even GitHub**. All these are examples of a SPA



## Single Page Application Pros and Cons

Just like most other technologies, SPAs have their own advantages and disadvantages. Knowing about each one of them will bring you closer to deciding whether the SPA framework fits your app idea or not.

### Single Page Application Pros

Single-page applications are fast as most of the resources, including HTML, CSS, and Scripts, are loaded once, and only data is transmitted back and forth. Here are some of the business benefits of building single-page applications:

#### 1. Quick Loading Time

A page taking over [200 milliseconds to load](https://ai.googleblog.com/2009/06/speed-matters.html) can significantly affect your online business and, eventually, sales.

With the SPA approach, your full page loads quicker than traditional web applications, as it only has to load a page at the first request. On the other hand, traditional web apps have to load pages at every request, taking more time.

#### 2. Seamless User Experience

SPAs deliver an experience like a desktop or mobile app. Users do not have to watch a new page load, as only the content changes and not the page, making the experience an enjoyable one.

#### 3. Ease in Building Feature-rich Apps

SPA application makes it easy to add advanced features to a web application. For example, it is easier to build a content editing web app with real-time analysis using SPA development. Doing this with a traditional web app requires a total page reload to perform content analysis.

#### 4. Uses Less Bandwidth

It is no surprise that SPAs consume less bandwidth since they only load web pages once. Besides that, they can also do well in areas with a slow internet connection. Hence, it is convenient for everyone to use, regardless of internet speed.

### Single Page Application Cons

Single page application architecture is best for developing high-performing SAAS platforms and social networks. However, this approach has some disadvantages that make it unsuitable for developing highly secure, and SEO optimized websites.

#### 1. Doesn’t Perform Well With SEO

One of the metrics that search engines use is the number of pages a website has. However, since SPAs only load a single page, it serves as a disadvantage when[ranking on search engines](https://searchenginewatch.com/2018/04/09/an-seos-survival-guide-to-single-page-applications-spas/)

#### 2. Uses a Lot of Browser Resources

SPAs require many resources from the web browser since the browser is doing most of the tasks for the SPAs. Creating SPAs often need users to use the latest browsers with support for some modern features.

#### 3. Security Issues

As compared to multi-page apps, SPAs are more prone to cross-site scripting attacks. Using XSS, it becomes easy for hackers to introduce client-side scripts into a web app. Also, SPAs are more likely to expose sensitive data to all users.

Serverless and Event driven architecture.

An event-driven architecture **uses events to trigger and communicate between decoupled services** and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.

Is serverless good for microservices?

Serverless microservices are cloud-based services that use serverless functions to perform highly specific roles within an application. Serverless functions, which execute small segments of code in response to events, are modular and easily scalable, making them **well-suited for microservice-based architectures**.

What is a serverless architecture? A serverless architecture is **a way to build and run applications and services without having to manage infrastructure**. Your application still runs on servers, but all the server management is done by AWS.

An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Event-driven architectures have three key components: event producers, event routers, and event consumers. A producer publishes an event to the router, which filters and pushes the events to consumers. Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently.

[Event](https://pages.awscloud.com/AWS-Learning-Path-How-to-Use-Amazon-EventBridge-to-Build-Decoupled-Event-Driven-Architectures_2020_LP_0001-SRV.html?&trk=ps_a134p000003yBd8AAE&trkCampaign=FY20_2Q_eventbridge_learning_path&sc_channel=ps&sc_campaign=FY20_2Q_EDAPage_eventbridge_learning_path&sc_outcome=PaaS_Digital_Marketing&sc_publisher=Google) Driven Architecture:

Learn the basics of event-driven architectures and get started with EventBridge, including creating an event bus, setting up SaaS event sources, and more.

## Benefits of an event-driven architecture

### Scale and fail independently

By decoupling your services, they are only aware of the event router, not each other. This means that your services are interoperable, but if one service has a failure, the rest will keep running. The event router acts as an elastic buffer that will accommodate surges in workloads.

### Develop with agility

You no longer need to write custom code to poll, filter, and route events; the event router will automatically filter and push events to consumers. The router also removes the need for heavy coordination between producer and consumer services, speeding up your development process.

### Audit with ease

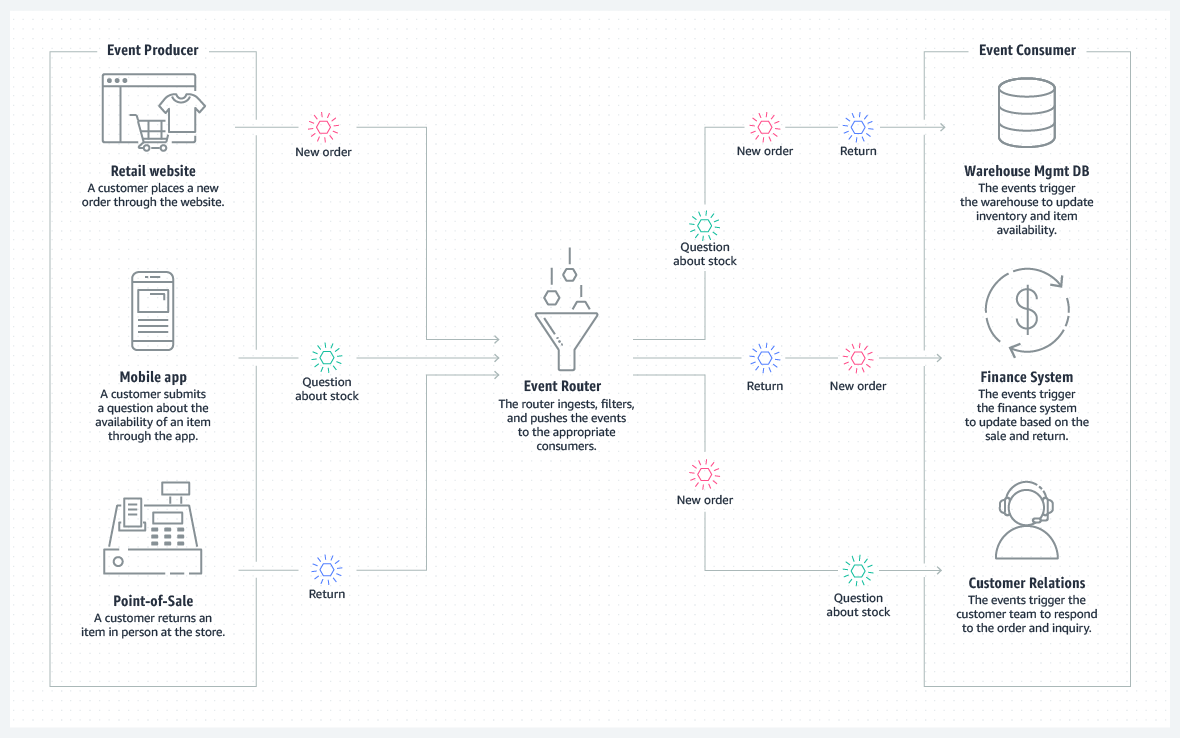
An event router acts as a centralized location to audit your application and define policies. These policies can restrict who can publish and subscribe to a router and control which users and resources have permission to access your data. You can also encrypt your events both in transit and at rest.

### Cut costs

Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router. This way, you’re not paying for continuous polling to check for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and less SSL/TLS handshakes.

## How it works: example architecture

Here's an example of an event-driven architecture for an e-commerce site. This architecture enables the site to react to changes from a variety of sources during times of peak demand, without crashing the application or over-provisioning resources.



## When to use this architecture

### Cross-account, cross-region data replication

You can use an event-driven architecture to coordinate systems between teams operating in and deploying across different regions and accounts. By using an event router to transfer data between systems, you can develop, scale, and deploy services independently from other teams.

### Resource state monitoring and alerting

Rather than continuously checking on your resources, you can use an event-driven architecture to monitor and receive alerts on any anomalies, changes, and updates. These resources can include storage buckets, database tables, serverless functions, compute nodes, and more.

### Fanout and parallel processing

If you have a lot of systems that need to operate in response to an event, you can use an event-driven architecture to fanout the event without having to write custom code to push to each consumer. The router will push the event to the systems, each of which can process the event in parallel with a different purpose.

### Integration of heterogeneous systems

If you have systems running on different stacks, you can use an event-driven architecture to share information between them without coupling. The event router establishes indirection and interoperability among the systems, so they can exchange messages and data while remaining agnostic.

### Should you use an event-driven architecture?

Event-driven architectures are ideal for improving agility and moving quickly. They’re commonly found in modern applications that use microservices, or any application that has decoupled components. When adopting an event-driven architecture, you may need to rethink the way you view your application design. To set yourself up for success, consider the following:

• The durability of your event source. Your event source should be reliable and guarantee delivery if you need to process every single event.

• Your performance control requirements. Your application should be able to handle the asynchronous nature of event routers.

• Your event flow tracking. The indirection introduced by an event-driven architecture allows for dynamic tracking via monitoring services, but not static tracking via code analysis.

• The data in your event source. If you need to rebuild state, your event source should be deduplicated and ordered.