Contents

[What are microservices ? 2](#_Toc57449691)

[Small, and Focused on Doing One Thing Well 2](#_Toc57449692)

[Autonomous 2](#_Toc57449693)

[Microservices vs SOA 3](#_Toc57449694)

[Service-Oriented Architecture 3](#_Toc57449695)

[Microservices 4](#_Toc57449696)

[Major Differences 4](#_Toc57449697)

[Key Benefits 5](#_Toc57449698)

[Technology Heterogeneity 5](#_Toc57449699)

[Resilience 5](#_Toc57449700)

[Scaling 6](#_Toc57449701)

[Ease of Deployment 6](#_Toc57449702)

[Organizational Alignment 6](#_Toc57449703)

[Composability 6](#_Toc57449704)

[Optimizing for Replaceability 6](#_Toc57449705)

[What About Service-Oriented Architecture ? 7](#_Toc57449706)

[Other Decompositional Techniques 8](#_Toc57449707)

[Shared Libraries 8](#_Toc57449708)

[Modules 8](#_Toc57449709)

[No Silver Bullet 8](#_Toc57449710)

Microservices

# What are microservices ?

Microservices are small, autonomous services that work together.

## Small, and Focused on Doing One Thing Well

Cohesion – the drive to have related code grouped together – is an important concept when we think about microservices.

“Gather together those things that change for the same reason, and separate those things that change for different reasons.” – Robert C. Martin’s definition of Single Responsibility Principle.

We focus our service boundaries on business boundaries, making it obvious where code lives for a given piece of functionality.

Jon Eaves at *RealState.com.au* in Australia characterizes a microservice as something that could be rewritten in two weeks, a rule of thumb.

“the smaller the service, the more you maximize the benefits and downsides of microservices architecture.”

As you get smaller, the benefits around interdependence increase.

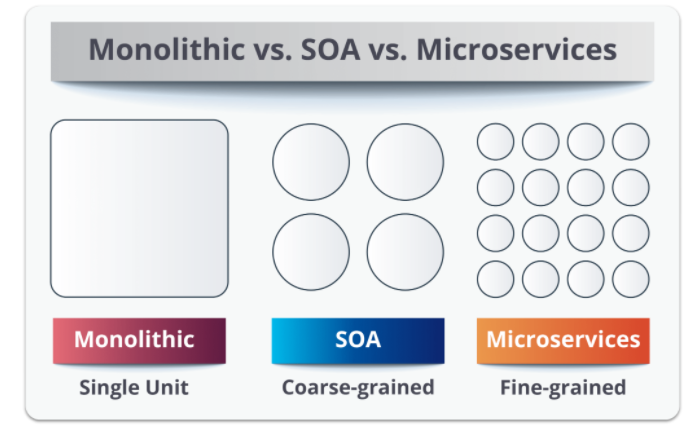
## Autonomous

Our microservice is a separate entity. It might be deployed as an isolated service on platform as a service (PAAS), it might be its own operating system process.

We try to avoid packing multiple services onto the same machine, although this isolation can add some overhead, as IPC will be over network to enforce separation between the services and avoid the perils of tight coupling.  
If there is too much sharing, our consuming services become coupled to our internal representations.   
This decreases our autonomy, as it requires additional coordination with consumers when making changes.

The golden rule: “Can you make a change to a service and deploy it by itself without changing anything else?”

# Microservices vs SOA

In layman’s terms, a **monolith** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.

## Service-Oriented Architecture

**Service-oriented architecture** is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or some activity coordination between two or more services. Some means of connecting services to each other is needed.

**SOA** defines 4 four basic service types:

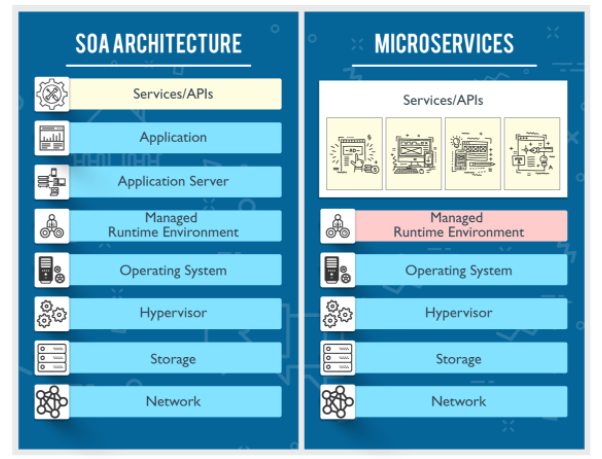
1. Functional/Business Services:
   1. This is related to *business users*.
   2. Coarse-grained services that define core business operations.
   3. Represented through XML, BPEL (Business Process Execution Language) and others.
2. Enterprise Services:
   1. This is related to *shared services team*.
   2. Implement the functionality defined by business services.
   3. Mainly rely on application services and infrastructure services to fulfil business requests.
3. Application Services:
   1. This is related to *application development team*.
   2. Fine-grained services that are confined to a specific application context.
   3. A dedicated user interface can directly invoke the services.
4. Infrastructure Services:
   1. This is related to *infrastructure services team*.
   2. Implement non-functional tasks such as authentication, auditing, security, and logging.
   3. Can be invoked from either application services or enterprise services.

## Microservices

**Microservices**, aka *microservice architecture*, is an architectural style that structures an application as a collection of small autonomous services modelled around a business domain.

Microservices have limited service taxonomy. They consist of two service types:

1. *Functional services* support specific business operations.
2. Accessing of services is done externally and these services are not shared with other services.
3. As in SOA, *infrastructure services* implement tasks such as authentication, auditing, security, logging.
4. In this, the services are not unveiled to the outside world.



## Major Differences

|  |  |  |
| --- | --- | --- |
| SOA | MSA |  |
| Follows “share-as-much-as-possible” architecture approach | Follows “share-as-little-as-possible” architecture approach | Sharing Granularity |
| Importance is on business functionality reuse | Importance is on the concept of “bounded context” | Component Sharing |
| They have common governance and standards | They focus on people, collaboration and freedom of other options |
| Uses Enterprise Service bus (ESB) for communication | Simple messaging system | Middleware vs API Layer |
| They support multiple message protocols | They use lightweight protocols such as HTTP/REST etc. | Remote Services |
| Multi-threaded with more overheads to handle I/O | Single-threaded usually with the use of Event Loop features for non-locking I/O handling |  |
| Maximizes application service reusability | Focuses on decoupling | Heterogenous Interoperability |
| Traditional Relational Databases are more often used | Modern Relational Databases are more often used |  |
| A systematic change requires modifying the monolith | A systematic change is to create a new service |  |
| DevOps / Continuous Delivery is becoming popular, but not yet mainstream | Strong focus on DevOps / Continuous Delivery |  |

# Key Benefits

Many of the benefits from microservices can be laid at the door of any distributed system. Microservices, however, tend to achieve these benefits to a greater degree primarily due to how far take the concepts behind the distributed systems and service-oriented architecture.

## Technology Heterogeneity

If one part of our system needs to improve its performance, we might decide to use a different technology stack that is better able to achieve the performance levels required.

We could also decide that how we store our data needs to change for different parts of our system.

With a monolithic application, if we want to try a new programming language, or framework, any change will impact a large amount of my system.

## Resilience

A key concept in resilience engineering is the bulkhead.

If one component of a system fails, but that failure doesn’t cascade, the problem can be isolated and the rest of the system can carry on working.

Service boundaries become obvious bulkheads.

## Scaling

With a large, monolithic service, we have to scale everything together. One small part of our overall system is constrained in performance, but if that behaviour is locked up in a giant monolithic application, we have to handle scaling everything as a piece. With small services, we can just scale those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware.

## Ease of Deployment

A one-line change to a million-line-long monolithic application requires the whole application to be deployed in order to release the change.

That could be a large-impact, high-risk deployment.

“the bigger the delta between the releases,   
the higher the risk that something will go wrong”

With microservices, a change to a single service can be made and deployed independent to the rest of the system. This will allow us to get our code deployed faster.

If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve; update rollout becomes faster as well.

## Organizational Alignment

Microservices allow use to better align our architecture to our organization, helping us:

1. Minimizing the number of people working on any one codebase to hit the sweet spot of team size and productivity.
2. Shift ownership of services between teams to try to keep people working on one service collocated.

## Composability

With microservices, we allow for our functionality to be consumed in different ways for different purposes.

## Optimizing for Replaceability

With our individual services being small in size, the cost to replace them with a better implementation, or even delete them altogether, is much easier to manage.

# What About Service-Oriented Architecture ?

Service-oriented architecture (SOA) is a design approach where multiple services collaborate to provide some end set of capabilities.

A service here typically means a completely separate OS process.

Communication between these services occurs via calls across a network rather than method calls within a process boundary.

It aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service don’t change too much.

Problems with SOA:

1. Communication protocols,
2. Vendor middleware,
3. A lack of guidance about service granularity, or
4. Wrong guidance or picking places to split your system.

## Other Decompositional Techniques

### Shared Libraries

A very standard Decompositional technique that is built into virtually any language is breaking down a codebase into multiple libraries. These libraries may be provided by third parties or created in one’s own organization.

Libraries give you a way to share functionality between teams and services.

But these have some drawbacks:

1. You lose technology heterogeneity; platform would become the dependent on the library-language.
2. The ease with which you can scale parts of your system independently from each other is truncated.
3. Unless the libraries are dynamically linked, a new library cannot be deployed without redeploying the entire process 🡺 Ability to deploy changes in isolation is reduced.

Services can and should make heavy use of third-party libraries to reuse common code. But they don’t get us all the way here.

## Modules

Open Source Gateway Initiate (OSGI) is worth calling out as one technology-specific approach to modular decomposition.

MEF exists for .NET.

The problem with OSGi is that it is trying to enforce things like module lifecycle management with enough support in the language itself.

Within a process boundary, it is also much easier to fall into the trap of making modules overly coupled to each other, causing all sorts of problems.

### Erlang Approach

Modules are baked into the language runtime, which can be stopped, restarted and upgraded without issue. It even supports running more than one version of the module at a given time, allowing for more graceful module upgrading.

Microservices - Not Silver Bullet