



Microservices

.NET CORE

The Microservice architectural style is an approach to developing a single application as a suite of small services. Each service runs in its own process and usually communicates with HTTP resource API.

[HTTPS://MARTINFOWLER.COM/MICROSERVICES/](https://martinfowler.com/microservices/)

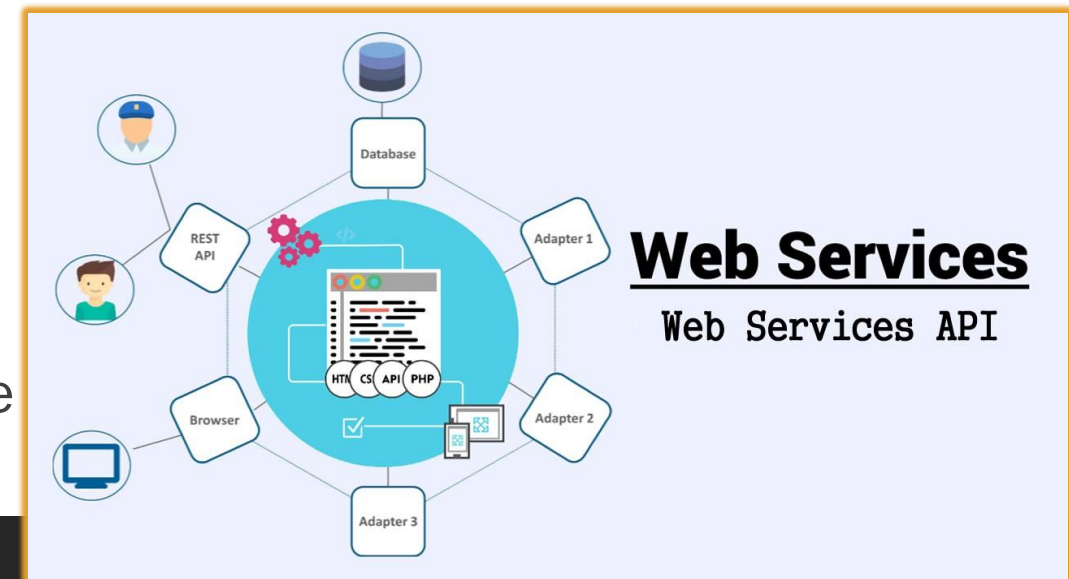
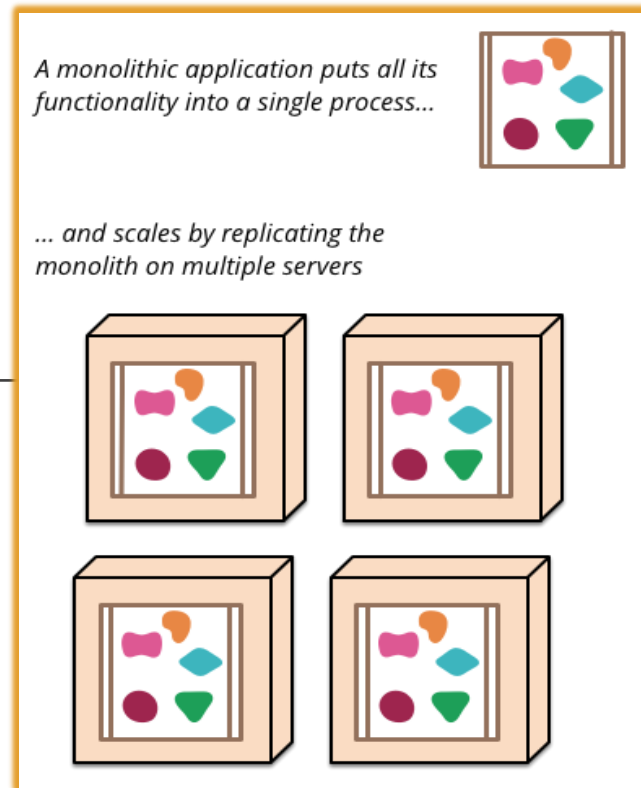
Web Services Applications Review

<https://martinfowler.com/articles/microservices.html>

Applications are often built as a “monolith”. All the code (except DB and UI) is compiled together and deployed together.

What’s the problem with this approach?

- One small change forces you to rebuild and redeploy the whole application as a new version.
- It’s hard to keep the code well organized with strong abstractions.
- If one part of the app is a bottleneck the whole app is affected.

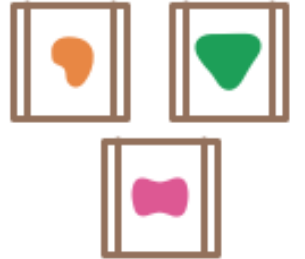


Microservices Architecture (MSA) – Overview

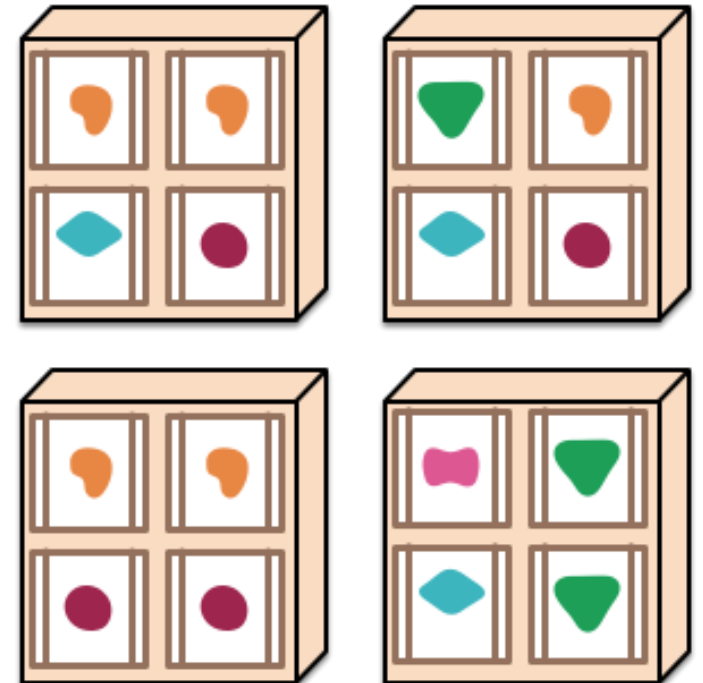
<https://martinfowler.com/articles/microservices.html>

The *Microservice architectural style* means developing a single application as a suite of smaller services. **MSA's** are built around business needs and each service (component) is independently deployable by fully automated deployment machinery. There is very little centralized management of the services and they are loosely coupled so they may be written in different programming languages with different data storage technologies.

A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.

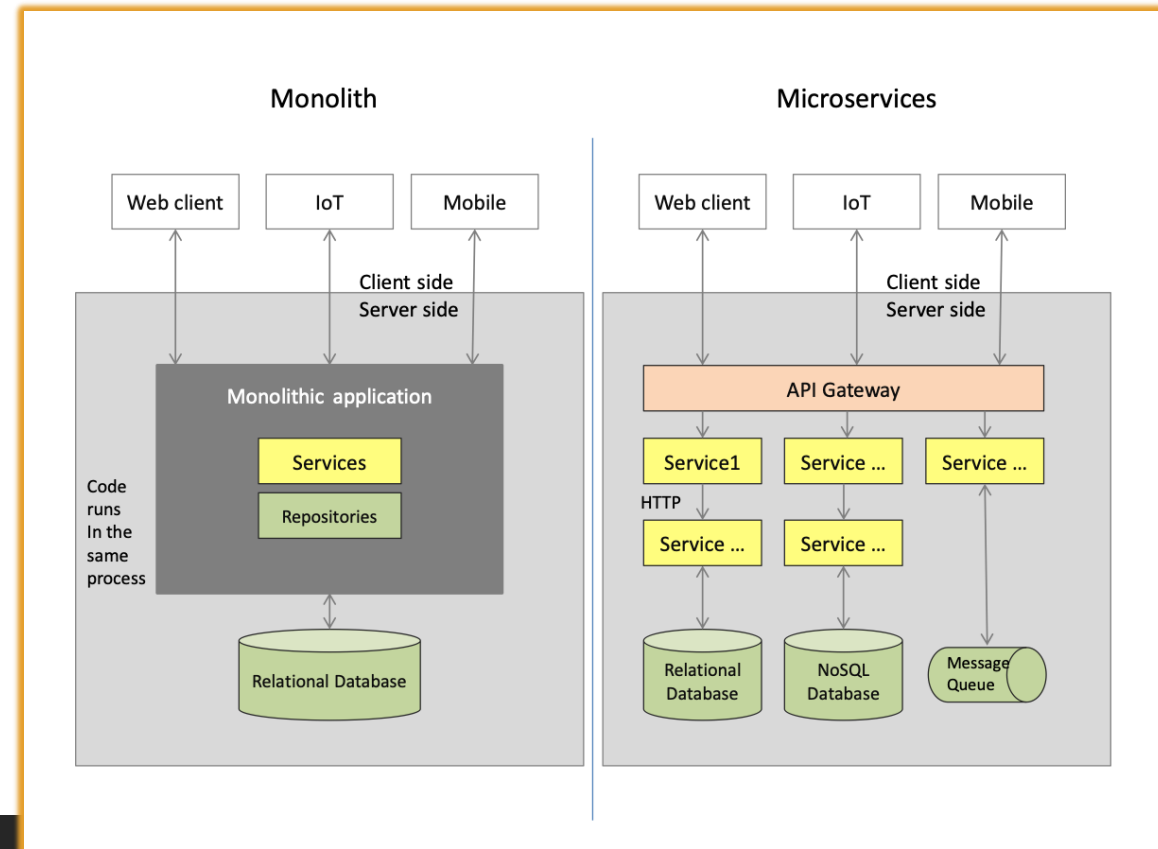


Microservices Architecture (MSA) – Overview

<https://martinfowler.com/articles/microservices.html>

The basic characteristics of Microservices Architecture are:

- Application divided into components (services).
- Avoids [Conway's Law](#).
- Products, not projects
 - Developers are responsible for the service for its entire lifetime.
- Smart endpoints and dumb pipes
 - Use HTTP to receive requests and respond, staying as decoupled as possible.
 - Use a lightweight message bus that acts as a message router only and don't do much more than provide a reliable asynchronous fabric
- CI/CD

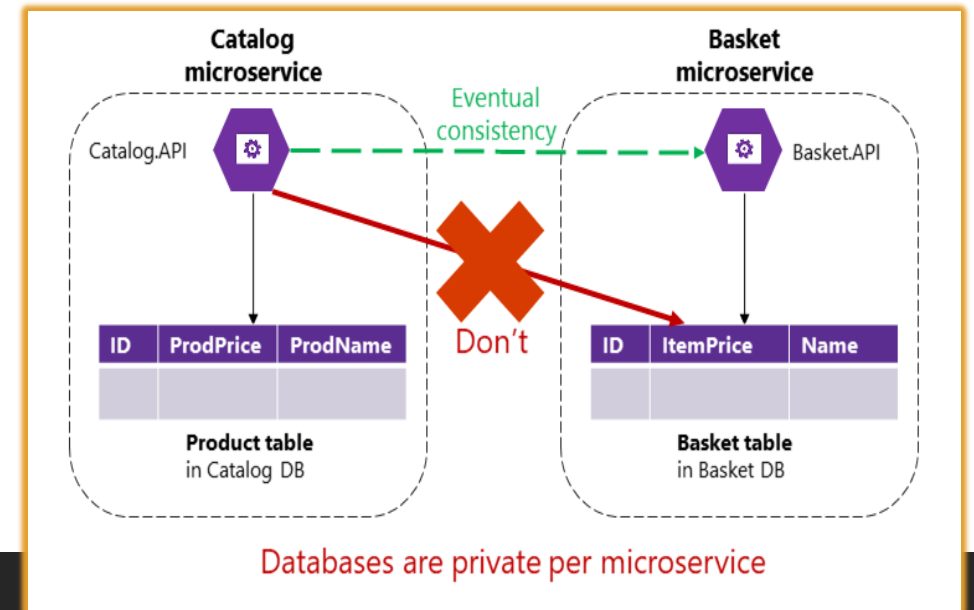
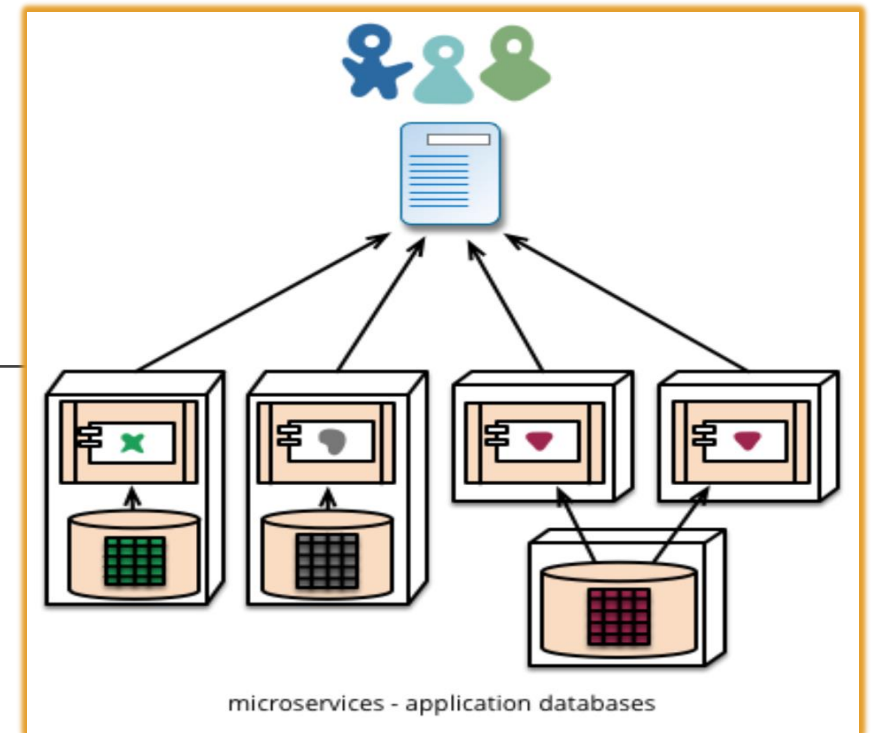


MSA Components – Overview

<https://martinfowler.com/articles/microservices.html>

The basic characteristics of Microservices Components (services) are:

- Each service implements one business capability.
- Services are developed, deployed, and scaled independently.
- Services control their own logic.
- Services manage/persist their own data
- Each services is replaceable and upgradable.
- Services communicate with *RPC's*

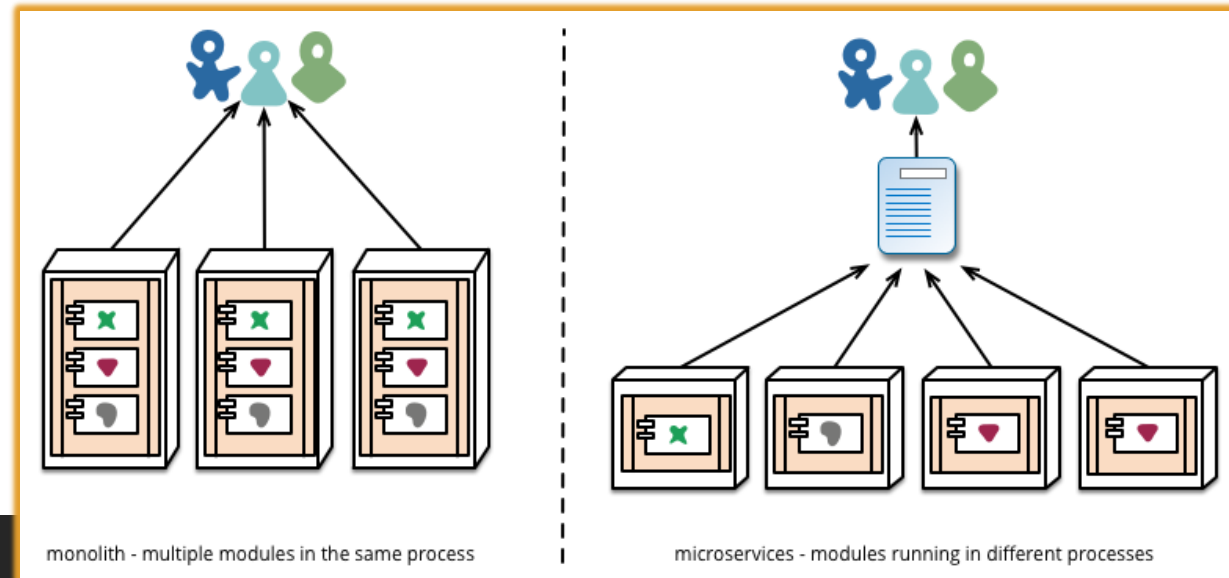
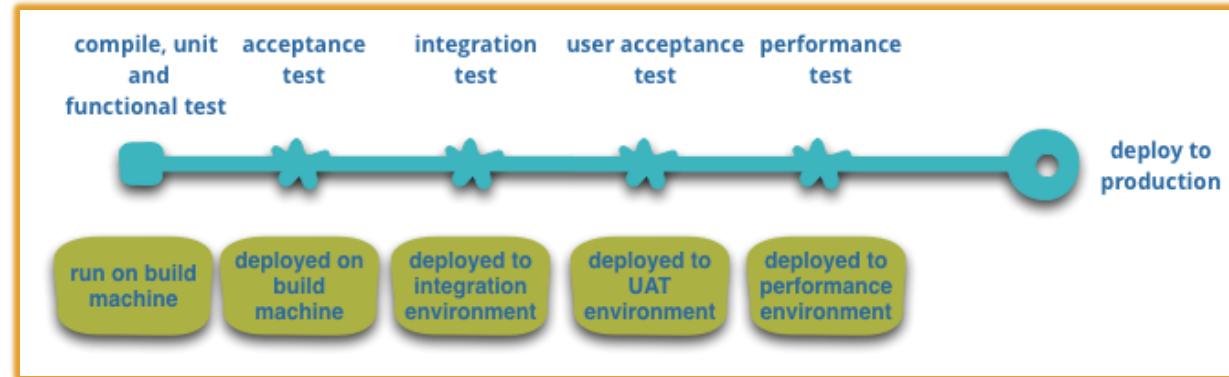


MSA and CI/CD

<https://martinfowler.com/articles/microservices.html>

The evolution of “the cloud” has reduced the operational complexity of building, deploying and operating microservices. Teams using CI/CD make extensive use of infrastructure automation techniques.

As long as deployment is “boring” there isn't much difference between monoliths and microservices. Although the operational landscape for each can be very different.



Pros of MSA

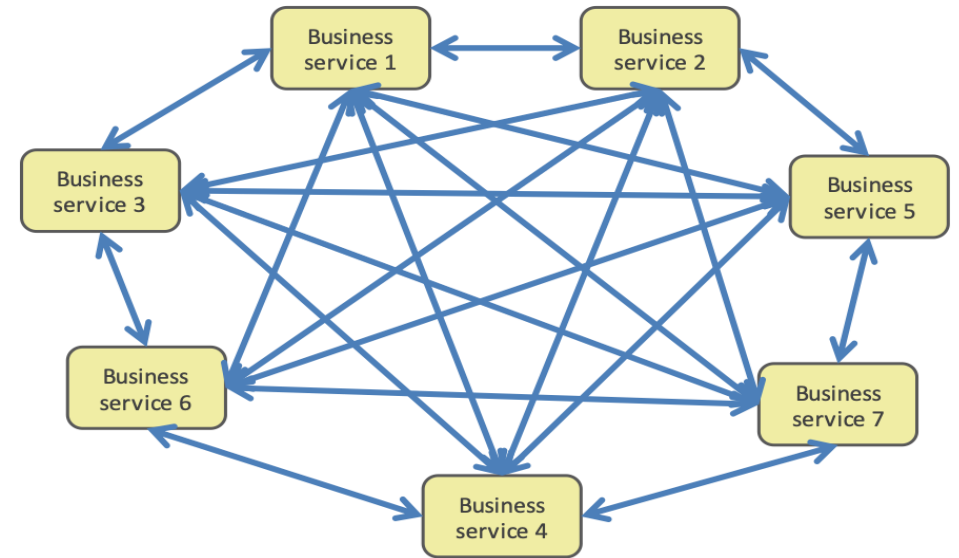
<https://developer.ibm.com/technologies/microservices/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/>

- Long-term flexibility when new technologies are developed.
- Higher ROI and better TCO (Total Cost of Ownership) with faster, cheaper development using simpler, cheaper hardware.
- Easier fault isolation and bug fixing leads to higher resiliency.
- Loose coupling is enforced by the architecture
- Time-to-market: Smaller, easier-to-understand services help to quickly get new features live.
- Easily scalable with increasing load requirements. Just add another server!
- Implement individual services in most appropriate technology

Cons of MSA

<https://developer.ibm.com/technologies/microservices/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/>

- If the service relies on a relational DB, they may be difficult to scale and complex to manage. ACID transactions increase overhead.
- There are many more moving parts that can break so there must be more error handling and resiliency built into the system.
- Different technologies used for each service can lead to difficulties:
 - when team members transition from one team to another.
 - when maintaining a very diverse technology group, more personnel are needed.
- High dependency between varying services can lead to a “microservices death star”. Many services needing to be altered for certain adjustments made to one service.
- A complex and changing communication system between services can be hard to understand. IP addresses and ports can get out of sync when updating.
- Harder to implement integration testing when each team only deals with their own microservice.

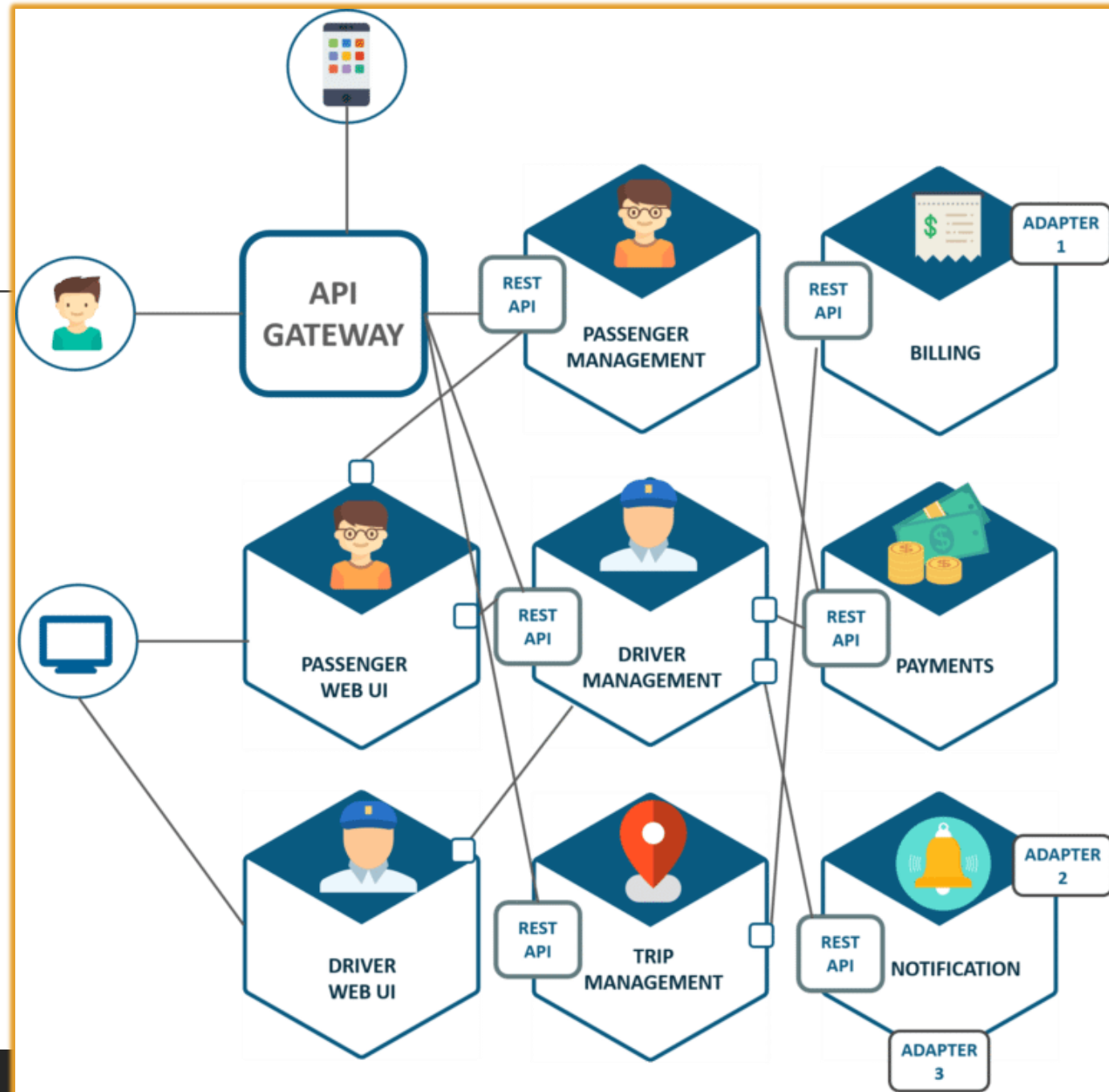


MSA Example and Requirements

<https://martinfowler.com/bliki/MicroservicePrerequisites.html>

You must have certain capabilities in place before starting a MDA application.

- Quick server creation – you must be able to automate provisioning to respond quickly to outages or fluctuating demand.
- Accurate Monitoring – detect problems quickly to be able to respond appropriately to arising problems.
- Fast deployment – preferably use a fully automated deployment pipeline to rapidly respond to developing needs.
- Employ Product-centered teams that develop and maintain the same product for the lifetime of the product.



When is MSA Appropriate?

<https://martinfowler.com/bliki/MicroservicePremium.html>

The decision of whether or not to use microservices depends on the complexity of the system you're contemplating.

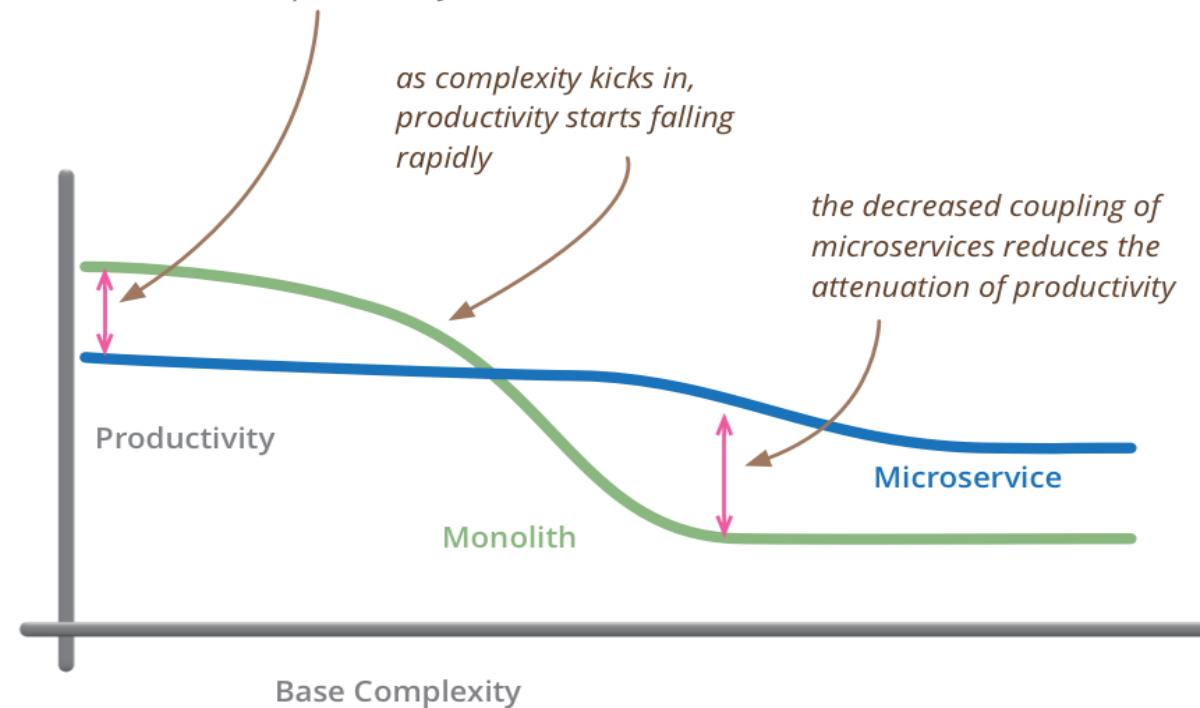
The MSA approach introduces its own set of complexities, like:

- automated deployment and monitoring,
- dealing with failure,
- eventual consistency

The primary guideline would be:

don't even consider microservices unless you have a system that's too complex to manage as a monolith.

for less-complex systems, the extra baggage required to manage microservices reduces productivity



but remember the skill of the team will outweigh any monolith/microservice choice

MSA and Containerization

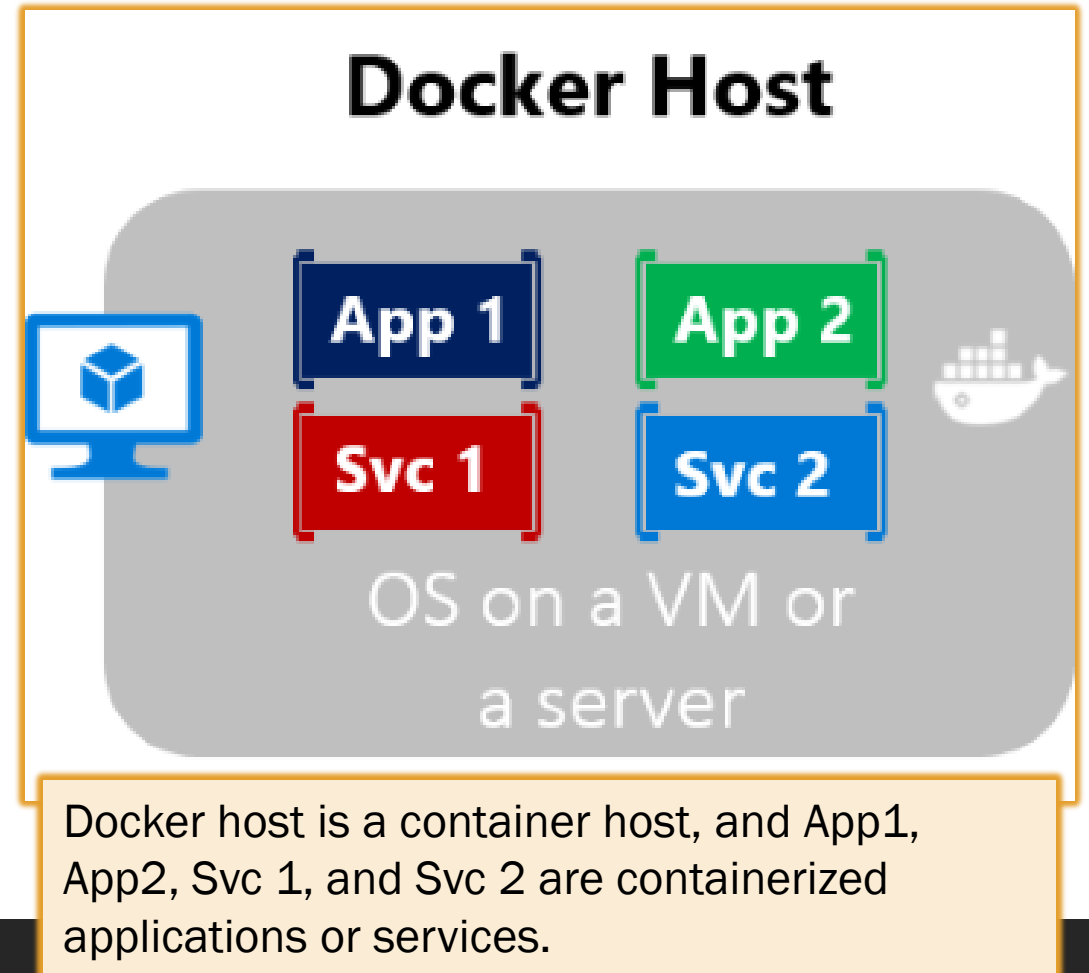
<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/>

Containerization is when an application (or service), its dependencies, and its configuration are packaged together as a container image. The containerized application can be tested as a unit and deployed as a container image instance to the host operating system (OS).

Software containers act as a standard unit of software deployment that can contain different code and dependencies.

Each container can run a whole web application or a service.

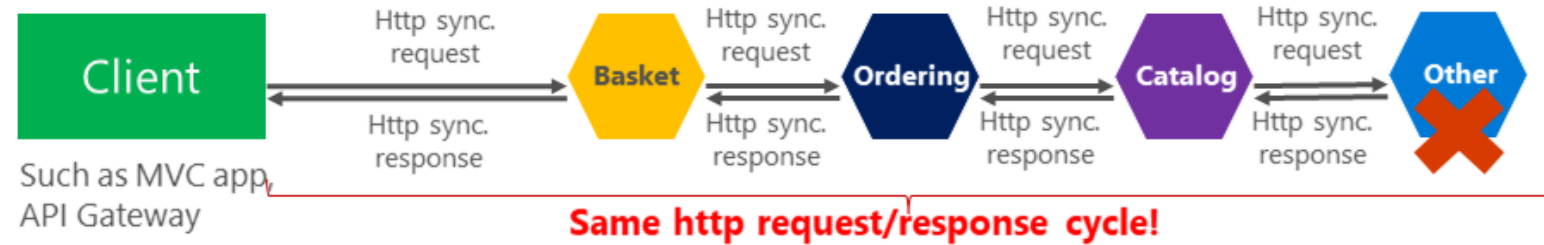
Containers offer the benefits of isolation, portability, agility, scalability, and control. The most important benefit is the environment's isolation provided between Dev and Ops.



Synchronous vs. async communication across microservices

Anti-pattern

Synchronous
all request/response cycle



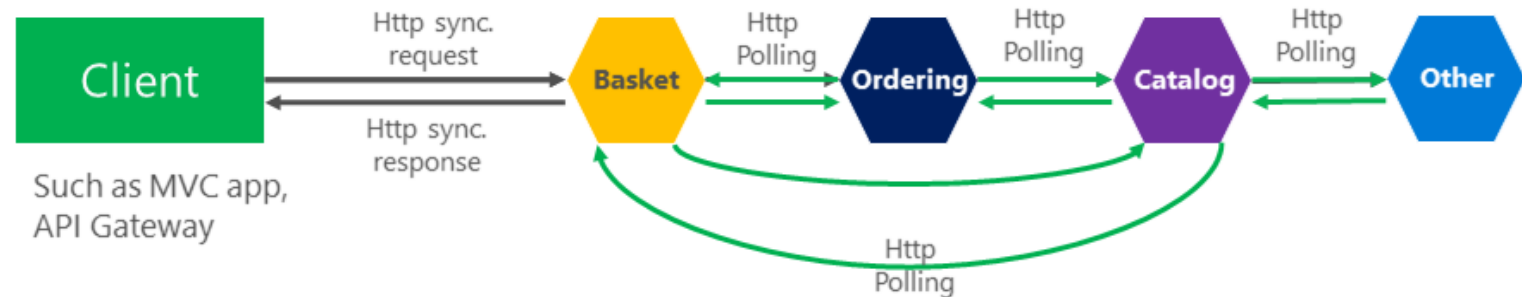
Asynchronous

Comm. across internal microservices
(EventBus: like **AMQP**)



"Asynchronous"

Comm. across internal microservices
(Polling: **Http**)

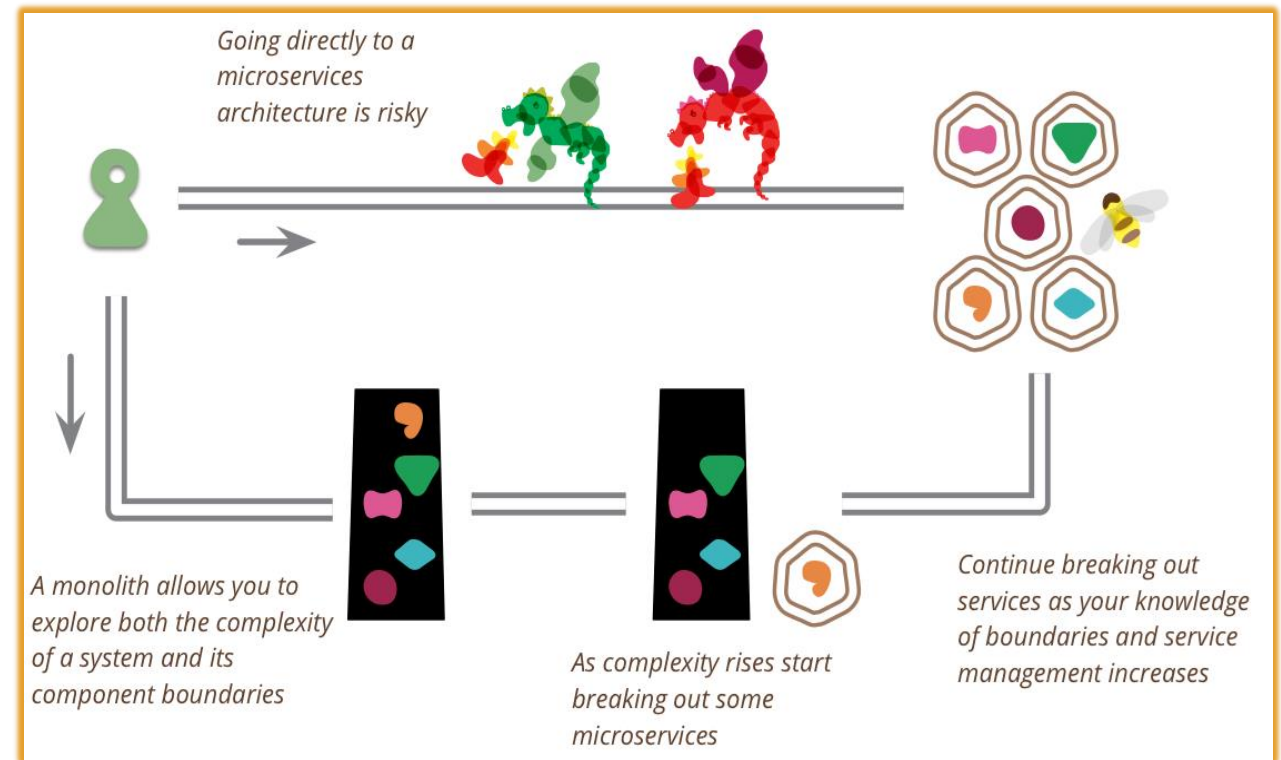


Monolith + MSA?

Some say, start with a monolith and evolve it to MSA if and when needed

Pros:

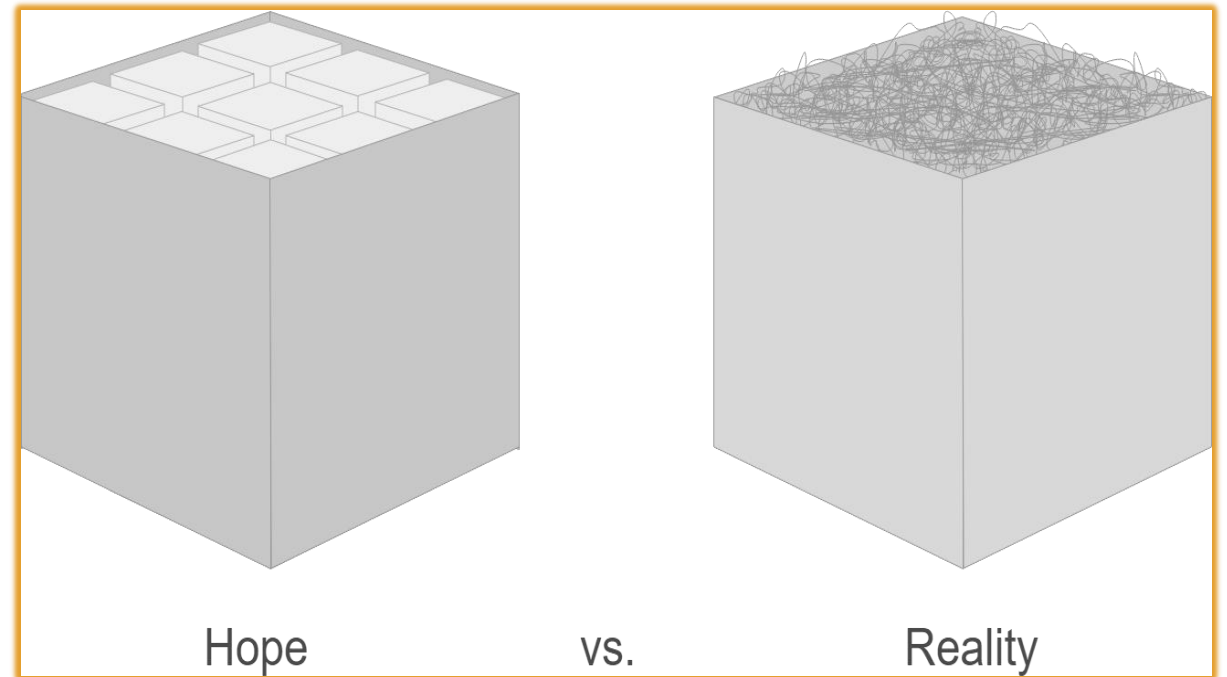
- That's what most MSA success stories did
- Do we *really* know where to draw all the service boundaries before we have an MVP?



Convert a Monolith to MSA?

Cons:

- The monolith's parts will inevitably be more tightly coupled to each other
- Good module separation in a monolith might not be the same as good service boundaries



Review – Things that go well with MSA

- Agile
- DevOps, CI/CD
- Containers/Docker
- Orchestration/Kubernetes
- Automated testing
- REST

Microservices Tutorial

<https://dotnet.microsoft.com/learn/aspnet/microservice-tutorial/intro>
