



Microservices

.NET 5

MicroServices Architecture (MSA) style is an approach to developing an application as a suite of smaller 'services'.

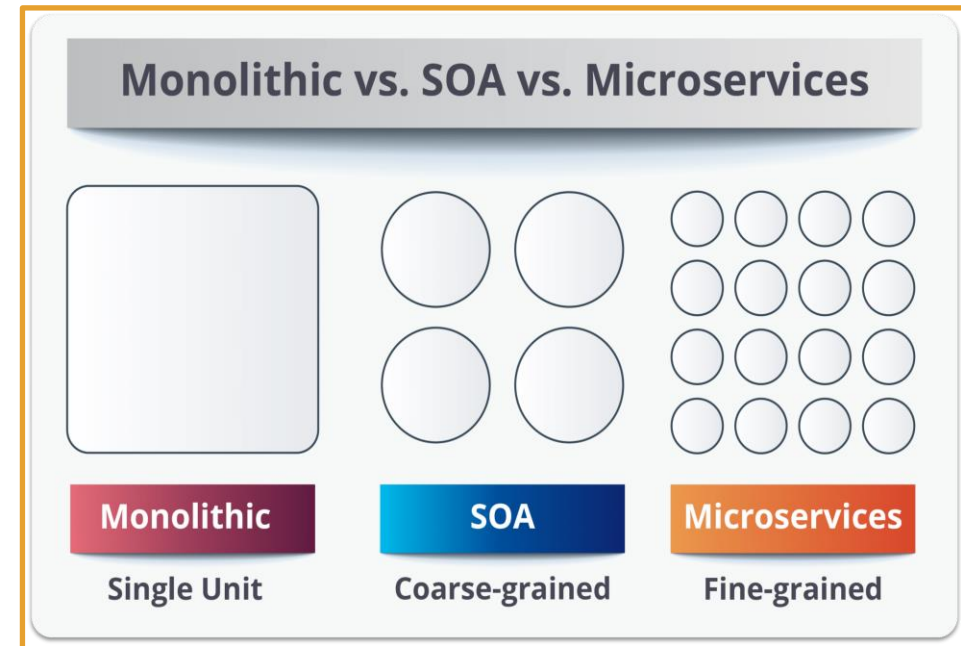
Each service runs within its own process and communicates through HTTP with other APIs. All the API's combine to form a complete application.

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

MSA vs. SOA

<https://dzone.com/articles/microservices-vs-soa-whats-the-difference>
<https://www.bmc.com/blogs/microservices-vs-soa-whats-difference/>
<https://www.guru99.com/microservices-tutorial.html>

Service Oriented Architecture	Microservices Architecture
Divisions based on business functionality	Divisions based on ' bounded context '
Often leverages a Service Bus for communication	Uses a simple messaging system (http)
Support for multiple messaging protocols	Uses lightweight protocols (HTTP/REST)
Multi-threaded	Single-threaded
Focus on app reusability	Focus on decoupling components
Systemic change means altering the monolith or service.	Systemic change means adding a new service
CI/CD is becoming more popular	CI/CD is integral to development.



Web Services Review

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

Applications have been traditionally built as “monoliths”. All the code (except DB and UI) is compiled together and deployed together. This approach presents certain problems.

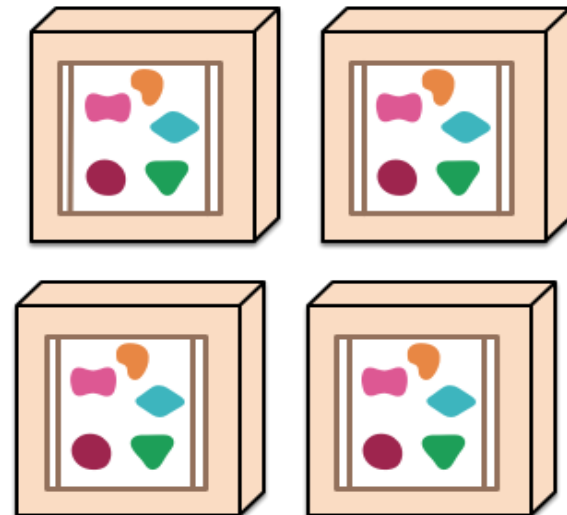
- One small change forces you to rebuild and redeploy the whole application as a new version.
- It's hard to keep the code well organized with its logical sections decoupled.
- If one part of the app is a bottleneck the whole app is affected.



A monolithic application puts all its functionality into a single process...



... and scales by replicating the monolith on multiple servers



Microservices Architecture – Overview

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

The **Microservice Architectural Style (MSA)** can be seen as a subset of SOA. MSA means developing a suite of small, highly focused services, then using them together to create a single application.

MSA's are built around business needs. Each service is independently deployable by fully automated deployment machinery.

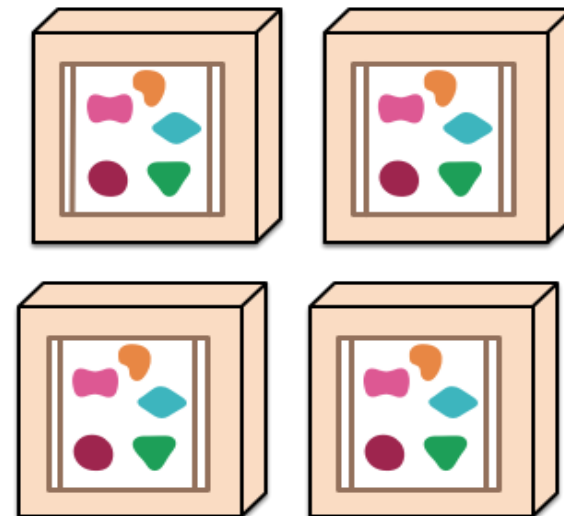
Individual services are loosely coupled with no central management. They may even be written in different programming languages with different data storage technologies.



A monolithic application puts all its functionality into a single process...



... and scales by replicating the monolith on multiple servers

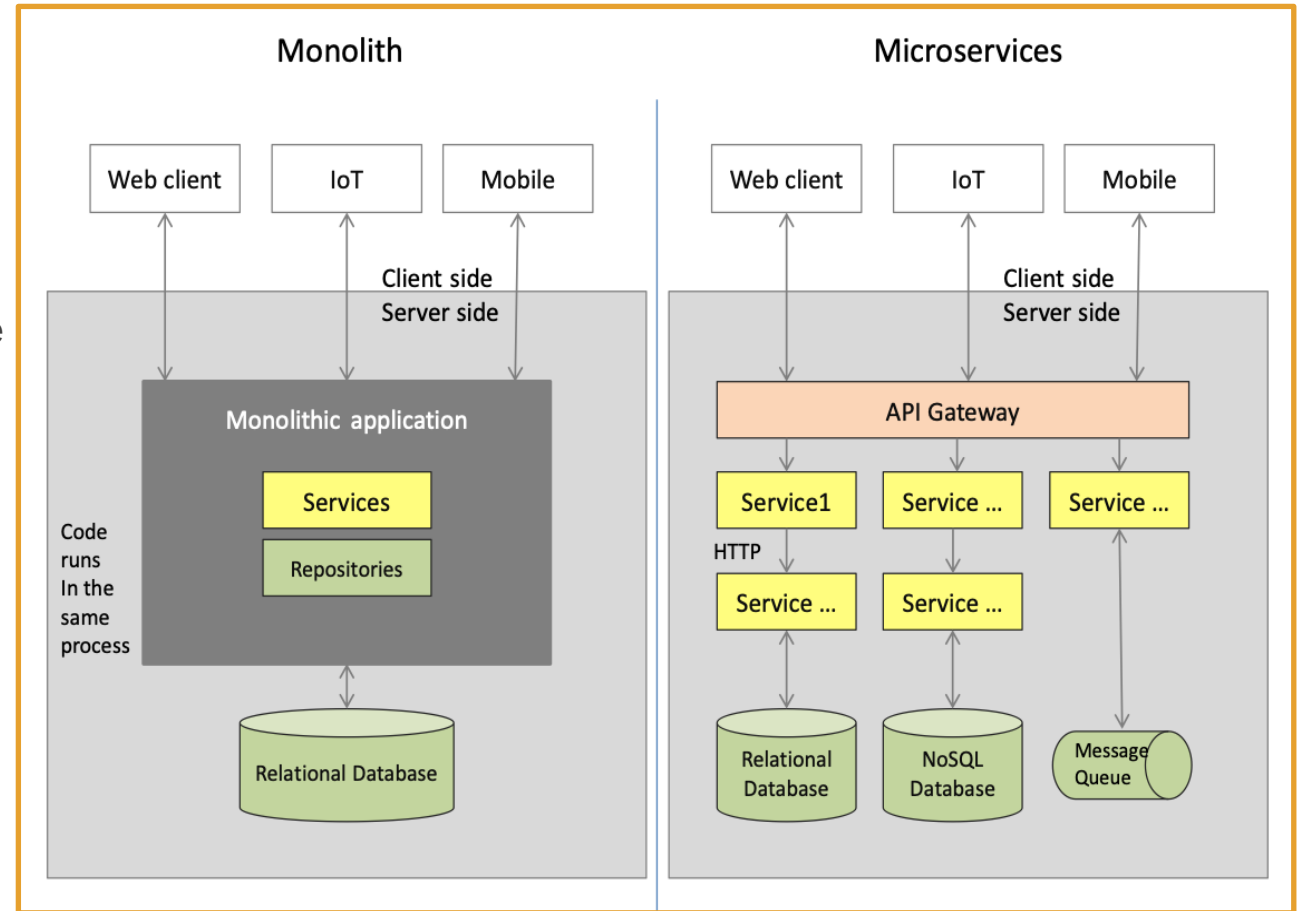


Microservices Architecture (MSA) – Overview

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

The basic characteristics of MSA are:

- Application has SOA.
- Avoids [Conway's Law](#).
- “Products, not projects”
- Developers are responsible for their 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 doesn't do much more than provide a reliable asynchronous fabric.
- CI/CD

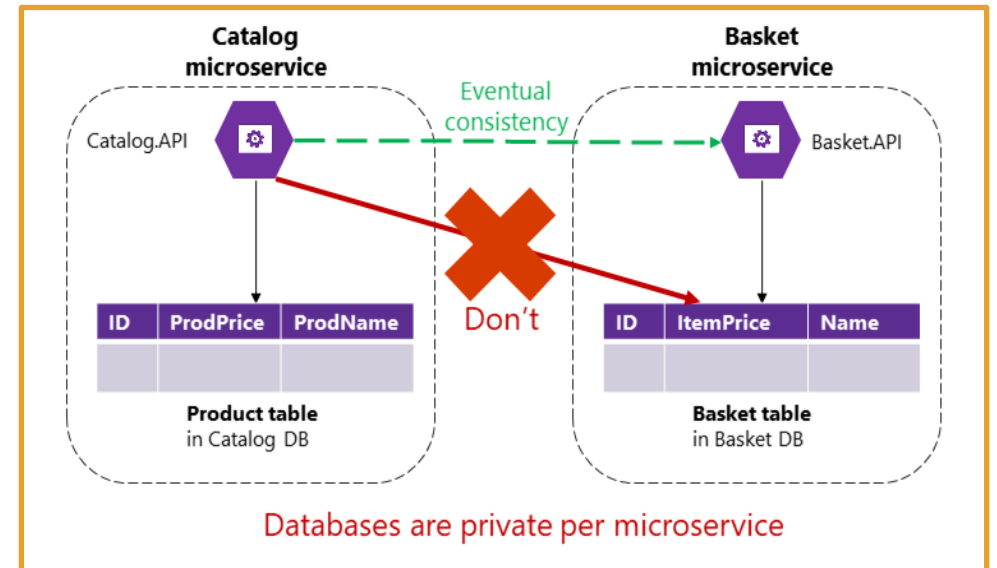
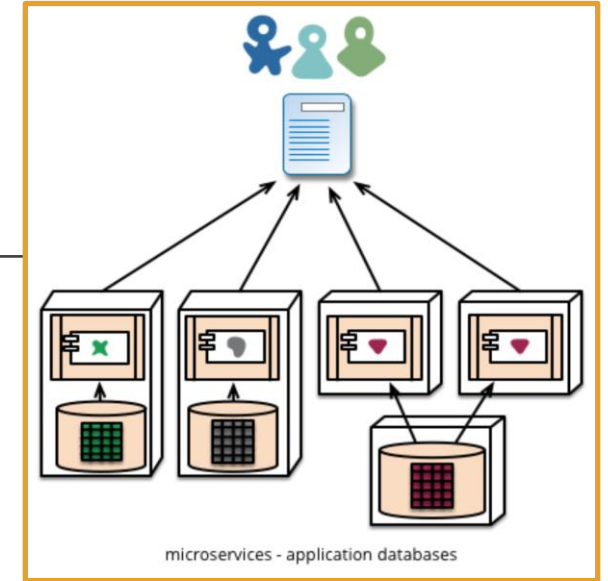


MSA Components – Overview

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

The basic characteristics of Microservices Components (services) are:

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



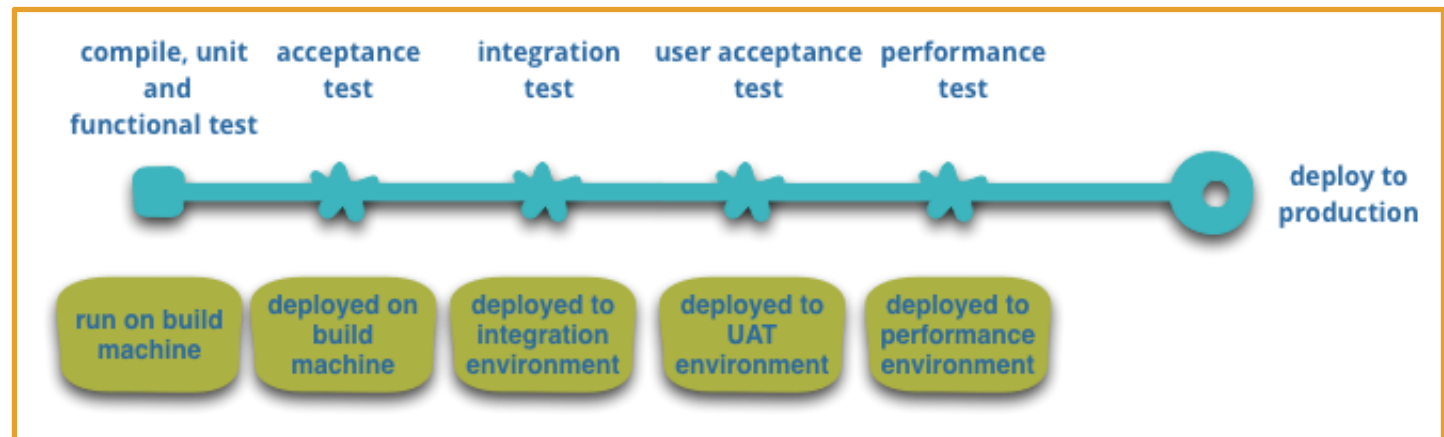
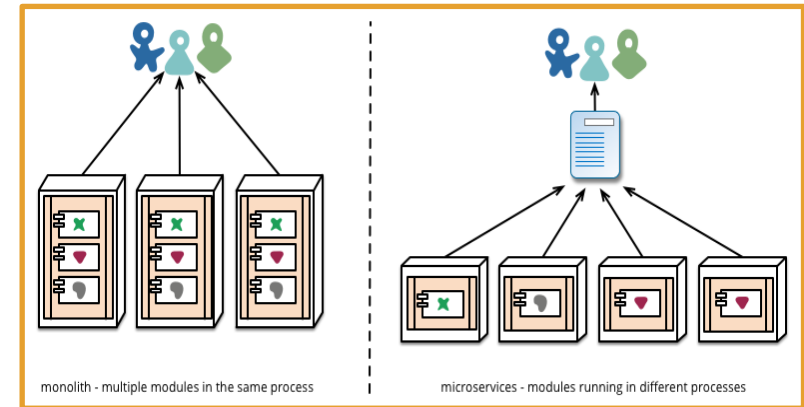
MSA and CI/CD

<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.

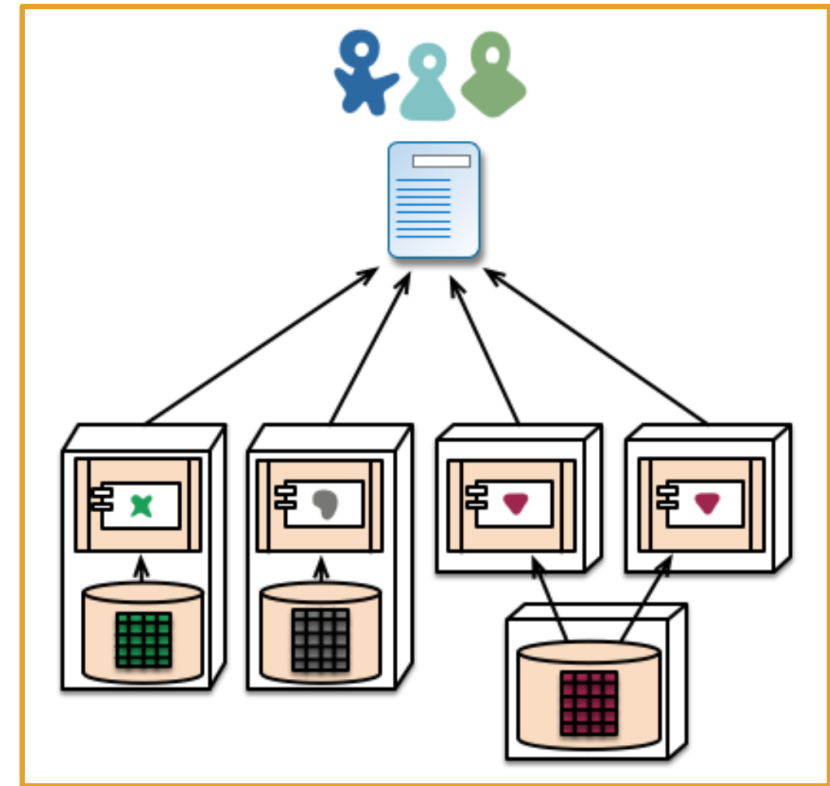
As long as deployment is “boring” there isn’t really a functional difference between monoliths and microservices.



Pros of MSA

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

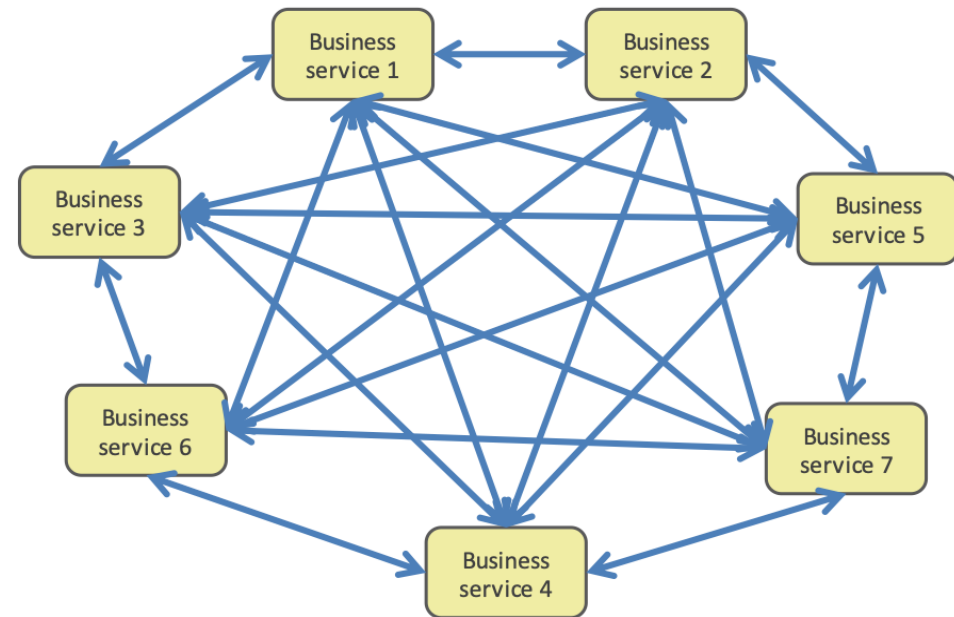
- Long-term flexibility when new technologies are developed.
- Higher ROI and lower TCO (Total Cost of Ownership) with faster, less expensive, development using simpler hardware.
- Easier fault isolation and bug fixing leads to higher resiliency.
- Loose coupling is enforced by the architecture.
- Smaller,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



Cons of MSA (1 / 3)

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

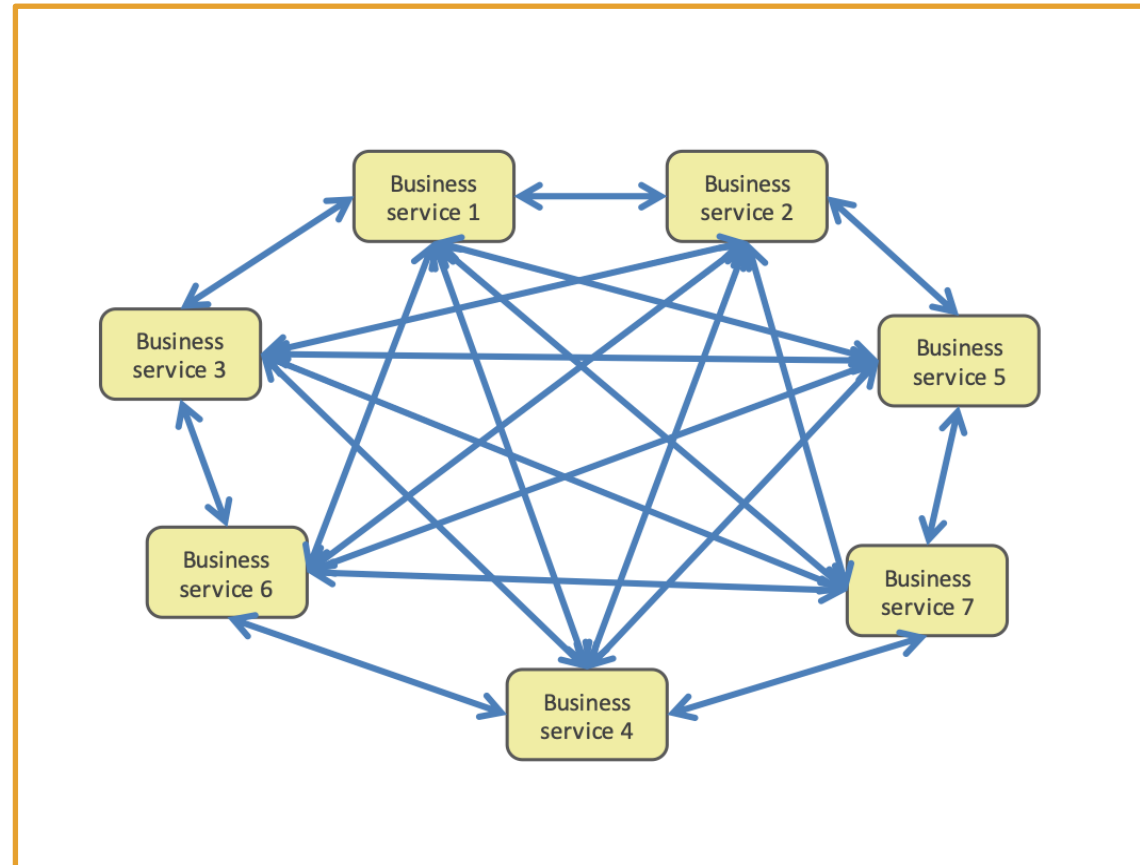
- Relational DB's may be difficult to scale and complex to manage.
- ACID transactions increase overhead.
- There are many more moving parts that can break requiring more error handling and resiliency built into the system.



Cons of MSA (2/3)

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

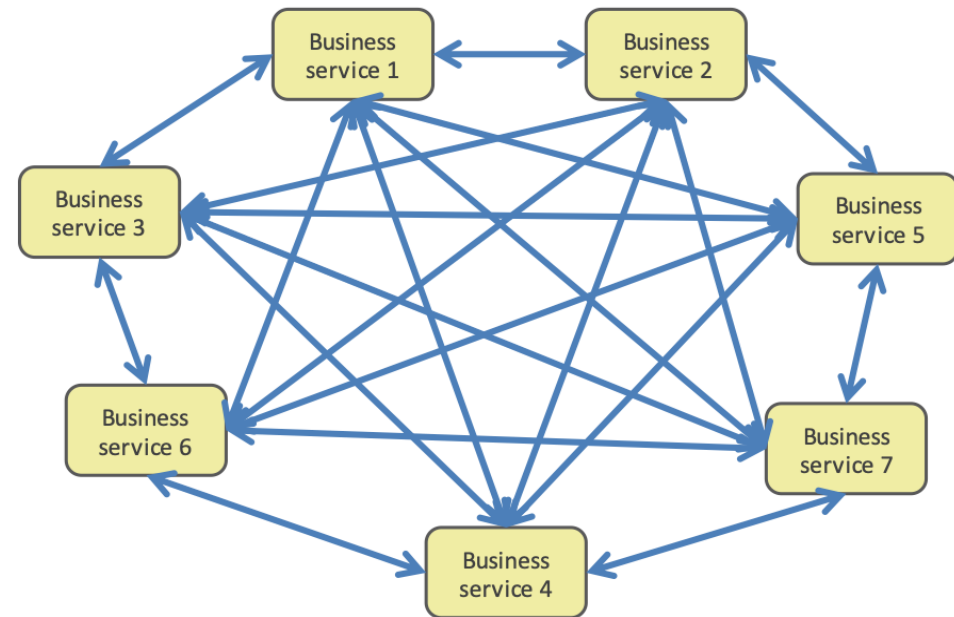
- Different technologies used for each service can lead to difficulties:
 - When team members transition from one team/technology to another.
 - A diverse technology group requires more personnel for maintenance.
- High dependency between services can lead to a “*microservices death star*”.
 - Adjustments to one service may require adjustments to many.



Cons of MSA (3/3)

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

- A complex and changing communication system between services is difficult to understand.
 - IP addresses and ports can get out of sync when updating.
- It's harder to implement integration testing when each team only deals with their own microservice.

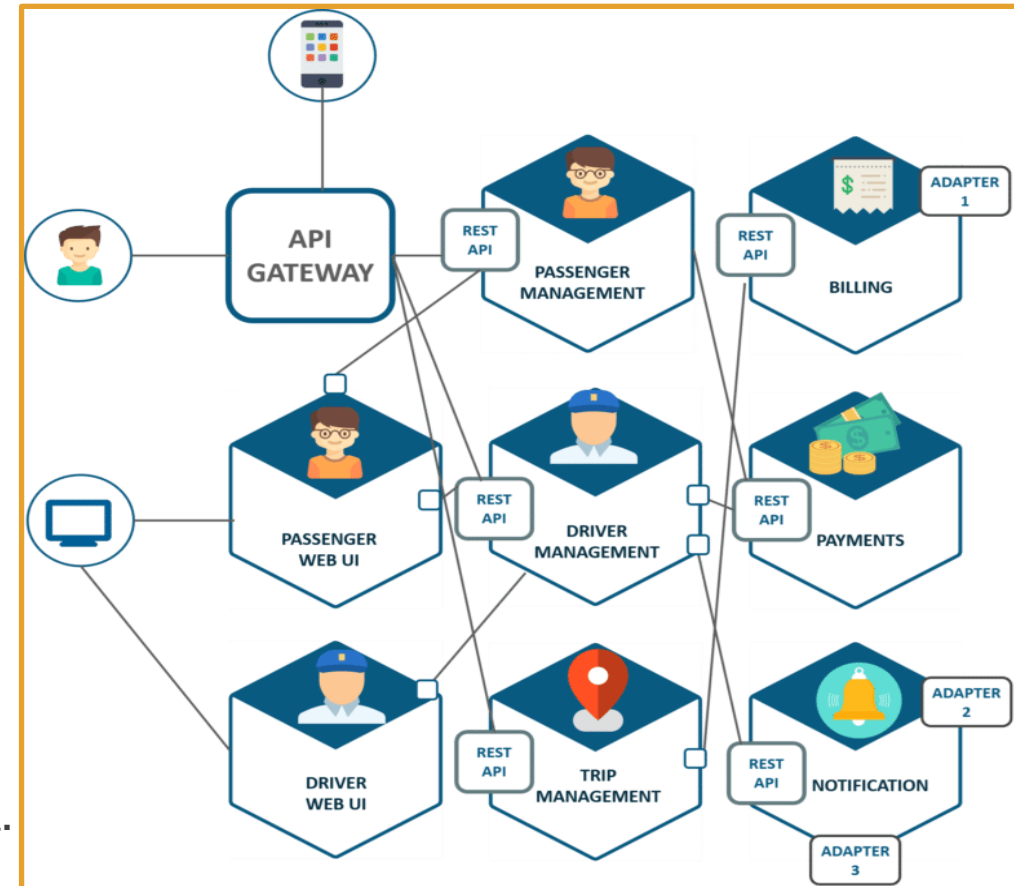


MSA Example and Requirements

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

Certain capabilities must be in place before starting a MSA application.

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



When is MSA Appropriate?

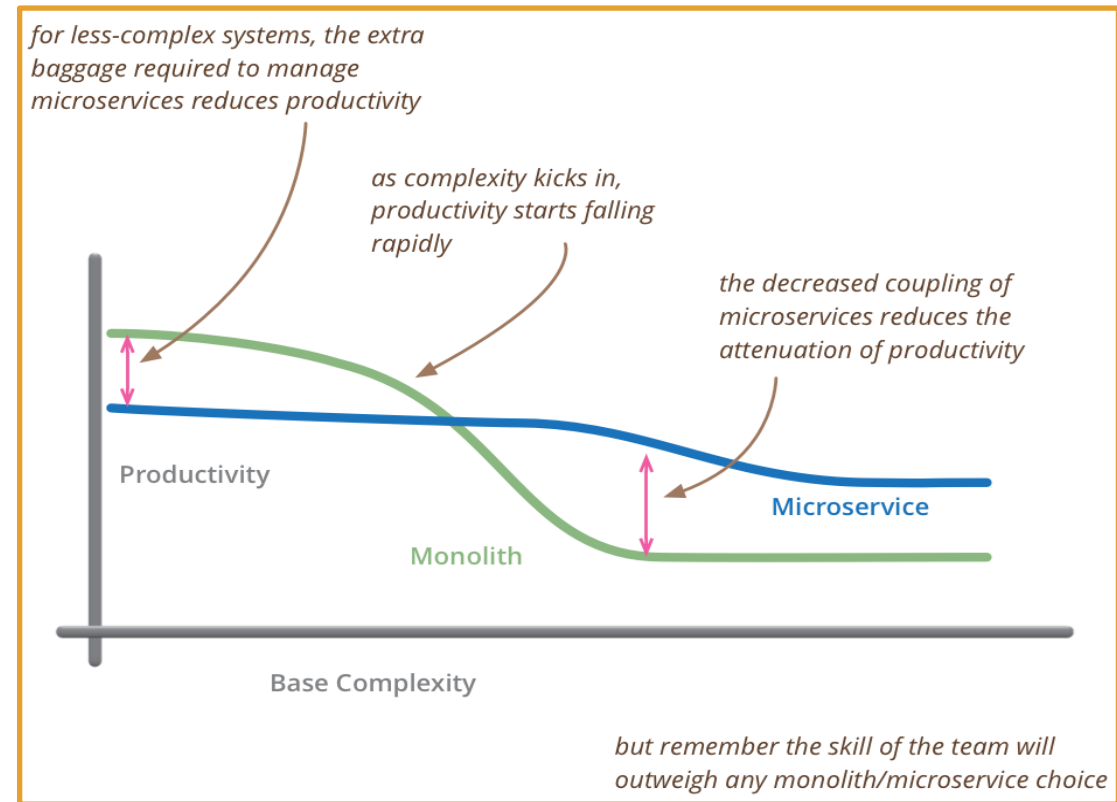
<https://martinfowler.com/bliki/MicroservicePremium.html>

The decision to use microservices depends on the complexity of the planned system.

The MSA approach introduces its own set of complexities, such as:

- automated deployment and monitoring.
- dealing with failure.
- gaining eventual consistency.

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



Migration from Monolith to MSA?

<https://martinfowler.com/articles/break-monolith-into-microservices.html>

Developers must decide what type of structure is appropriate for their application.

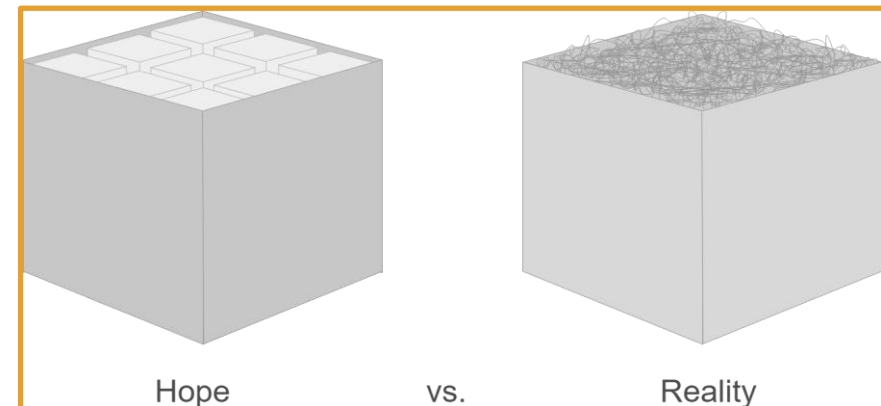
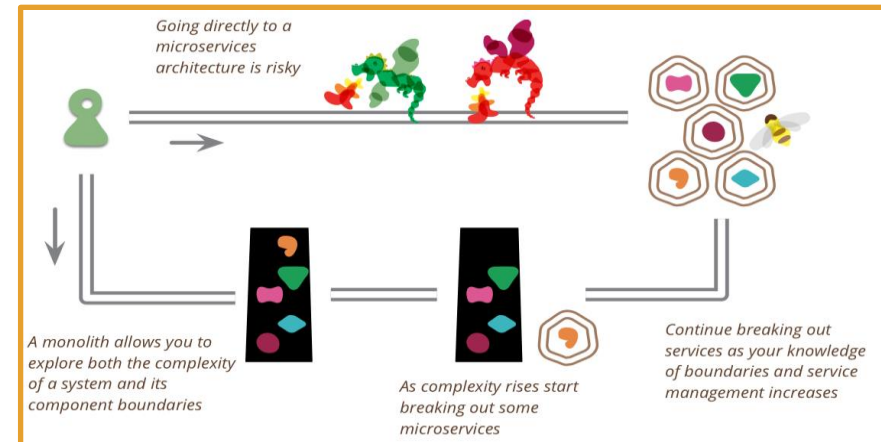
Should you start with a monolith and evolve it to MSA if needed?

Pros:

- It's what most MSA 'success stories' have done.
- Do we really know where to draw all the service boundaries before we have a Minimum Viable Product (MVP)?

Cons:

- The monolith's parts will inevitably be tightly coupled and difficult to decouple.
- Good module separation in a monolith might not be the same as good service boundaries.



MSA and Containerization

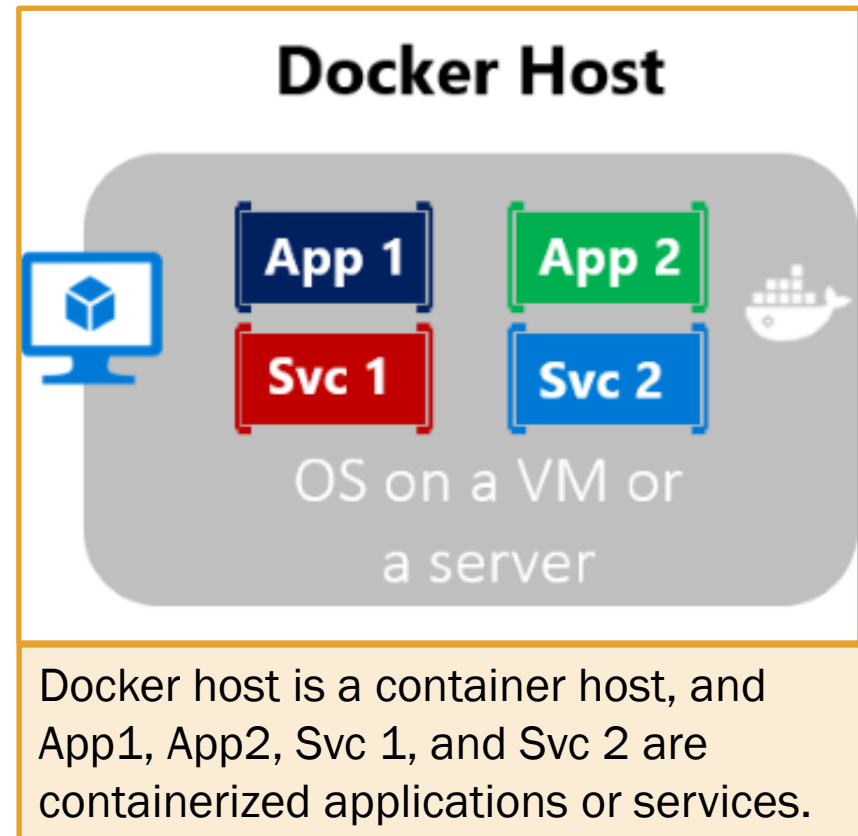
<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/>

An application, its dependencies, and its configuration are packaged together as a container image (containerized) and tested as a unit, then deployed as a container image instance to the host operating system.

Software containers act as standard units of software deployment and can therefore contain different code and dependencies.

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

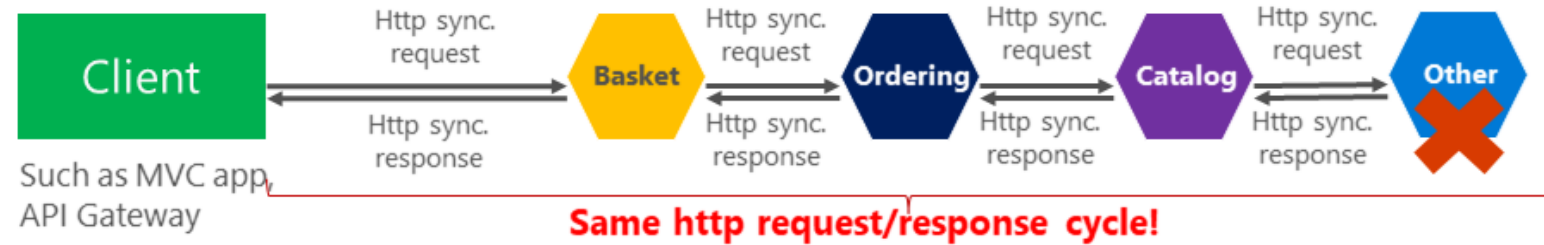
Containers offer the benefits of isolation, portability, agility, reliability, scalability, and control.



Synchronous vs. async communication across microservices

Anti-pattern

Synchronous
all request/response cycle



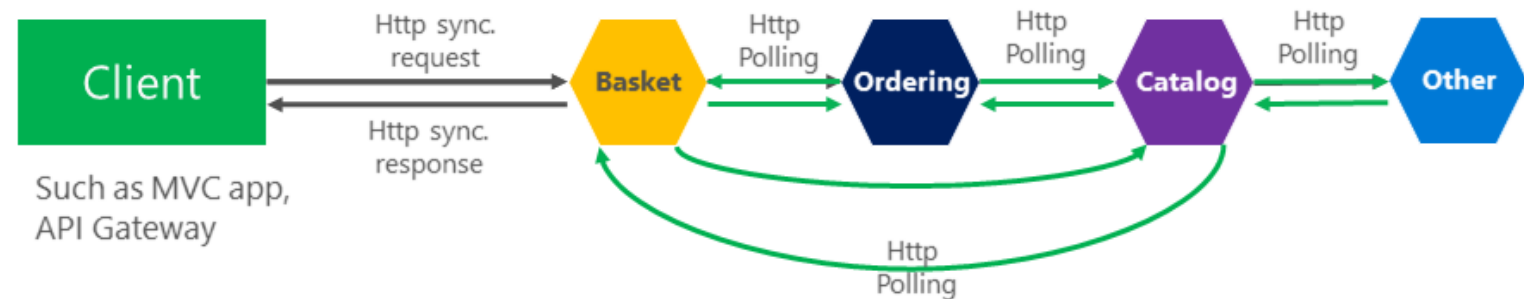
Asynchronous

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



"Asynchronous"

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



Microservices Tutorial (1/3)

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

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

1. Create a new api with `dotnet new webapi -o myMicroservice --no-https`. This creates the template WeatherForecast API.
2. `cd myMicroservice` into the new directory.
3. Run it with `dotnet run`.
4. Make sure you have Docker with `docker --version` or download Docker [here](#).
5. Create a **Dockerfile** with `vim dockerfile` (No suffix needed).
6. Add the text to the right to the Dockerfile.
7. Build the Docker Image with '`docker build -t mymicroservice .`'. The image is tagged as 'mymicroservice'.
8. Check that the image is created with `docker images ls`.
9. Run the service in the container with '`docker run -it --rm -p 3000:80 --name mymicroservicecontainer mymicroservice`'.
10. Verify that the container is running with `docker ps`.
11. Access the running app at <http://localhost:3000/WeatherForecast>.

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src
COPY myMicroservice.csproj .
RUN dotnet restore
COPY . .
RUN dotnet publish -c release -o /app

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
WORKDIR /app
COPY --from=build /app .
ENTRYPOINT ["dotnet", "myMicroservice.dll"]
```

Dockerfile text

Microservices Tutorial (2/3)

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

12. Make sure you are signed into DockerHub with **docker login** in your command line.
13. Upload the docker image with
 - **docker tag mymicroservice [YOUR DOCKER USERNAME]/mymicroservice**
 - **docker push [YOUR DOCKER USERNAME]/mymicroservice**
14. [Install Azure CLI](#) and sign in with **az login** in command line.
15. Install Azure Kubernetes Service with **az aks install-cli**. (ignore PATH variable config. options)
16. Create a resource group with:
 - **az group create --name MyMicroserviceResources --location westus**
17. Create an AKS cluster in the resource group with:
 - **az aks create --resource-group MyMicroserviceResources --name MyMicroserviceCluster --node-count 1 --enable-addons http_application_routing --generate-ssh-keys**
18. Download the credentials for the AKS Cluster with:
 - **az aks get-credentials --resource-group MyMicroserviceResources --name MyMicroserviceCluster**
19. cd back into the directory you created the service in. It was named 'MyMicroservice'.
20. Create a deployment **.yaml** file to hold the instructions for deployment with **start deploy.yaml**.
21. Copy the following text into **deploy.yaml**.

Microservices Tutorial (2/3)

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

22. Run the deployment with:

- `kubectl apply -f deploy.yaml`

23. See the details of the deployed service with:

- `kubectl get service mymicroservice --watch`

24. Look for the External IP address and see the deployed site with:

- `http://[EXTERNAL IP]/WeatherForecast`

25. To scale up your services to 2 (or more), use:

- `kubectl scale --replicas=2 deployment/mymicroservice`

26. Delete all created resources with:

- `az group delete -n MyMicroservice Resources`

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mymicroservice
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mymicroservice
    spec:
      containers:
      - name: mymicroservice
        image: [DOCKER ID]/mymicroservice:latest
        ports:
        - containerPort: 80
        env:
        - name: ASPNETCORE_URLS
          value: http://*:80
      selector:
        matchLabels:
          app: mymicroservice
---
apiVersion: v1
kind: Service
metadata:
  name: mymicroservice
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: mymicroservice
```

deploy.yaml

Replace [DOCKER ID] with your actual Docker ID.