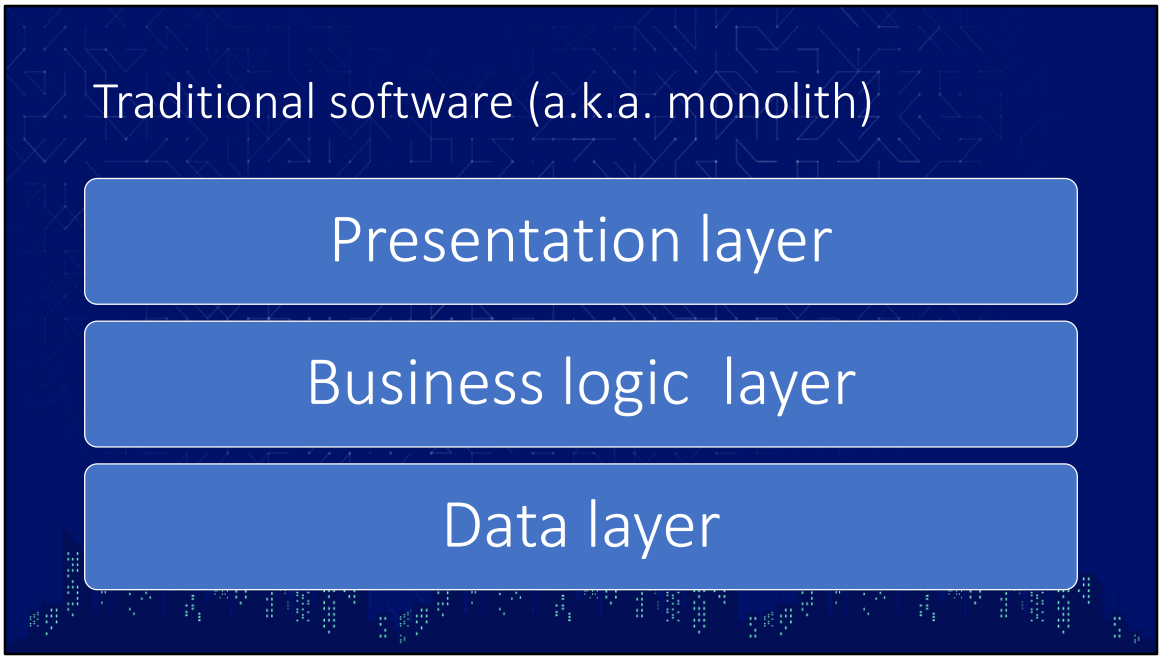
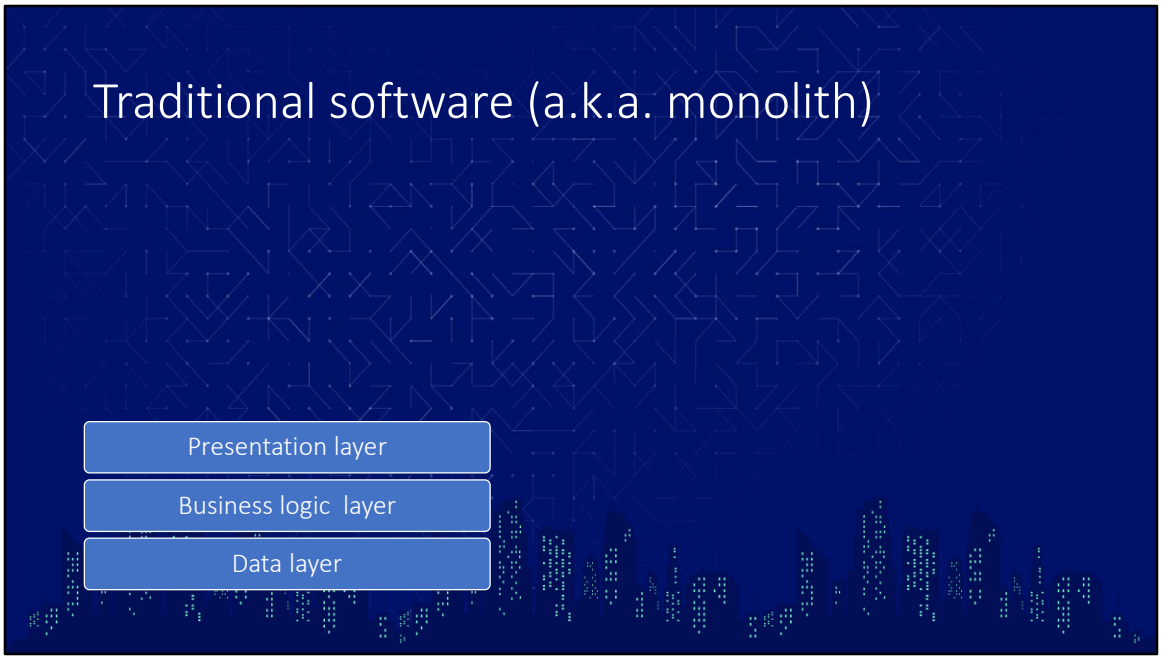
I'm Confused: Should I Orchestrate My Containers on Service Fabric or AKS?

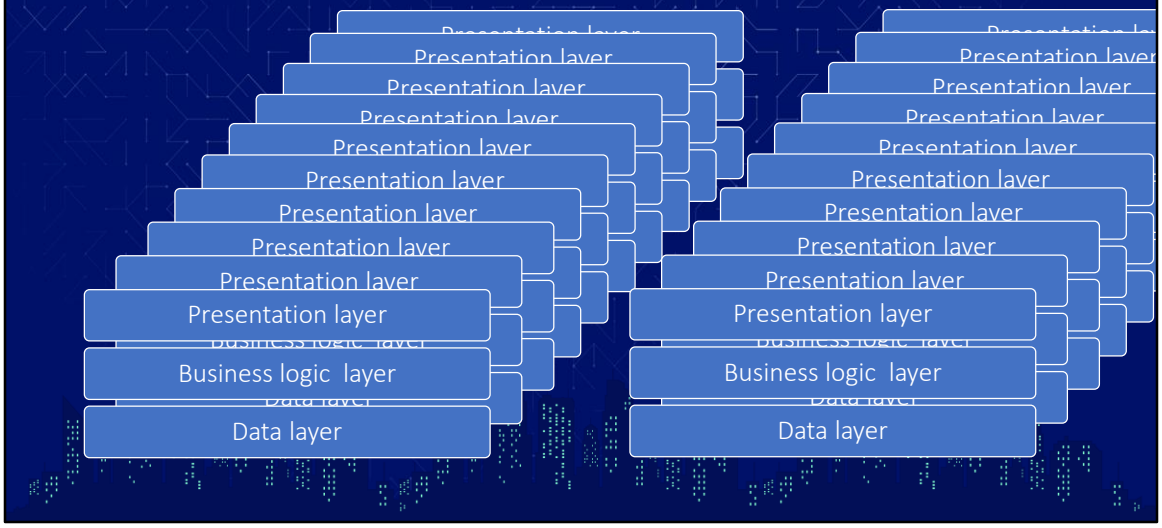
Alex Mang


Traditional software (a.k.a. monolith)
Presentation layer
Business logic layer
Data layer

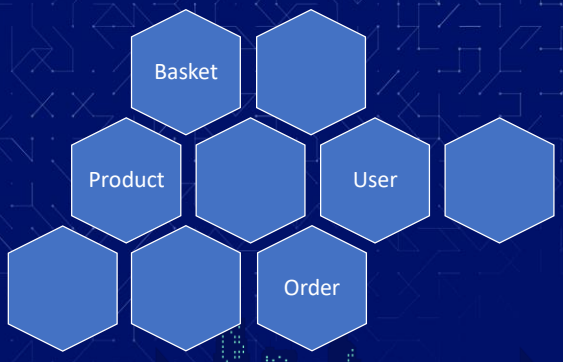
Traditional software (a.k.a. monolith)



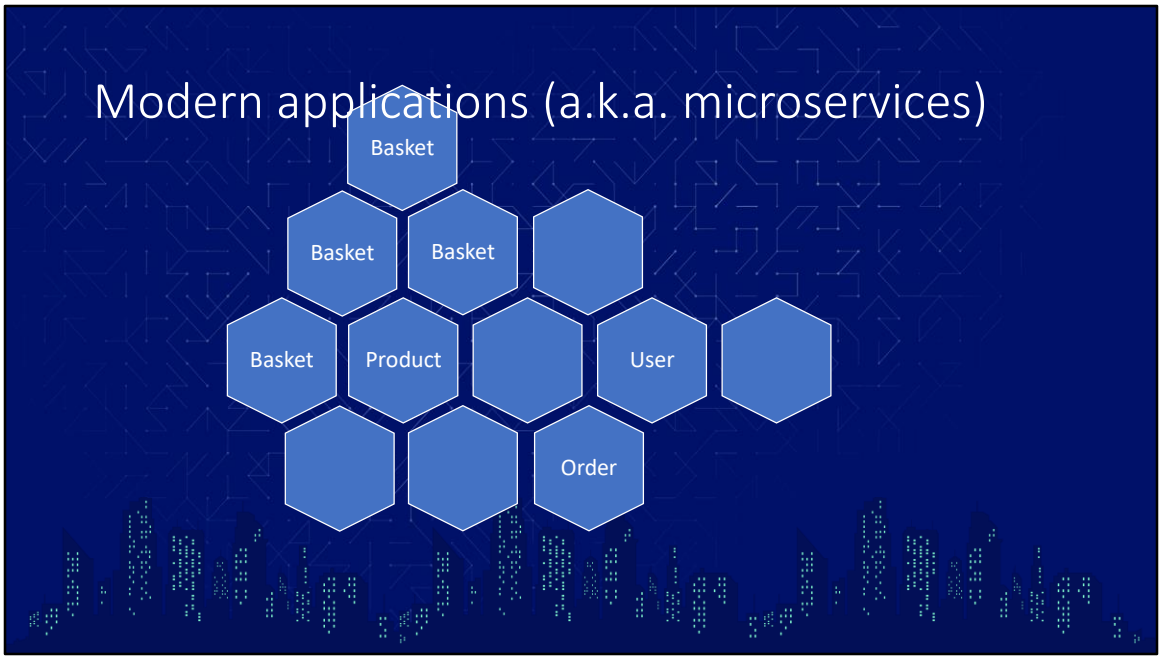
Traditional software (a.k.a. monolith)



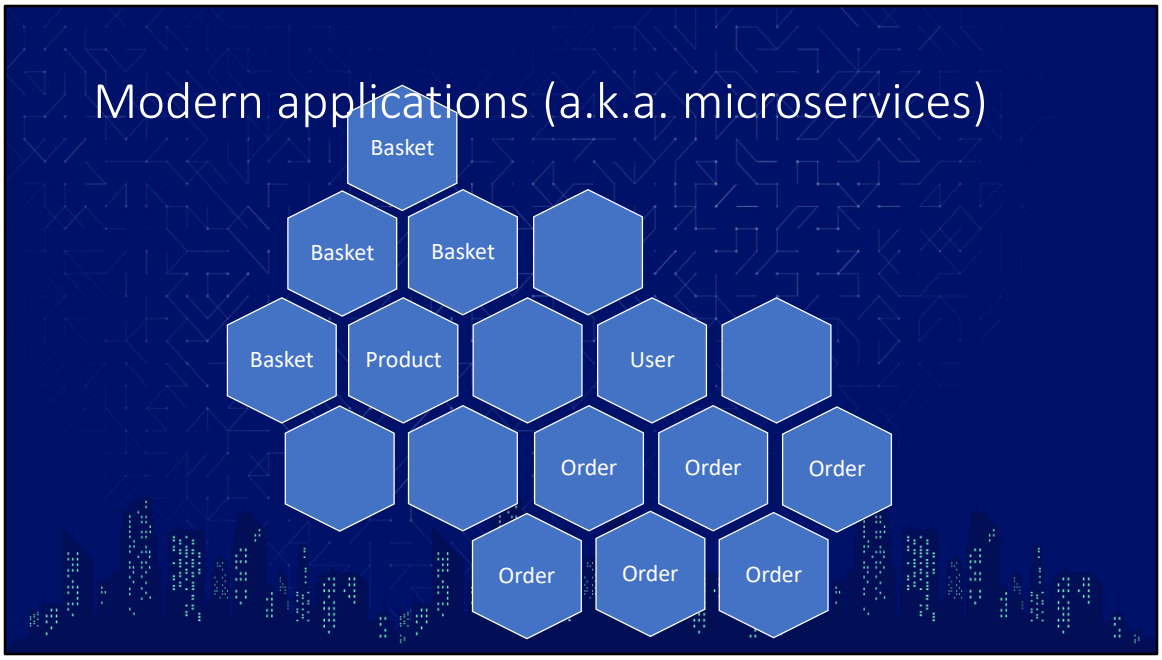
Modern applications (a.k.a. microservices)



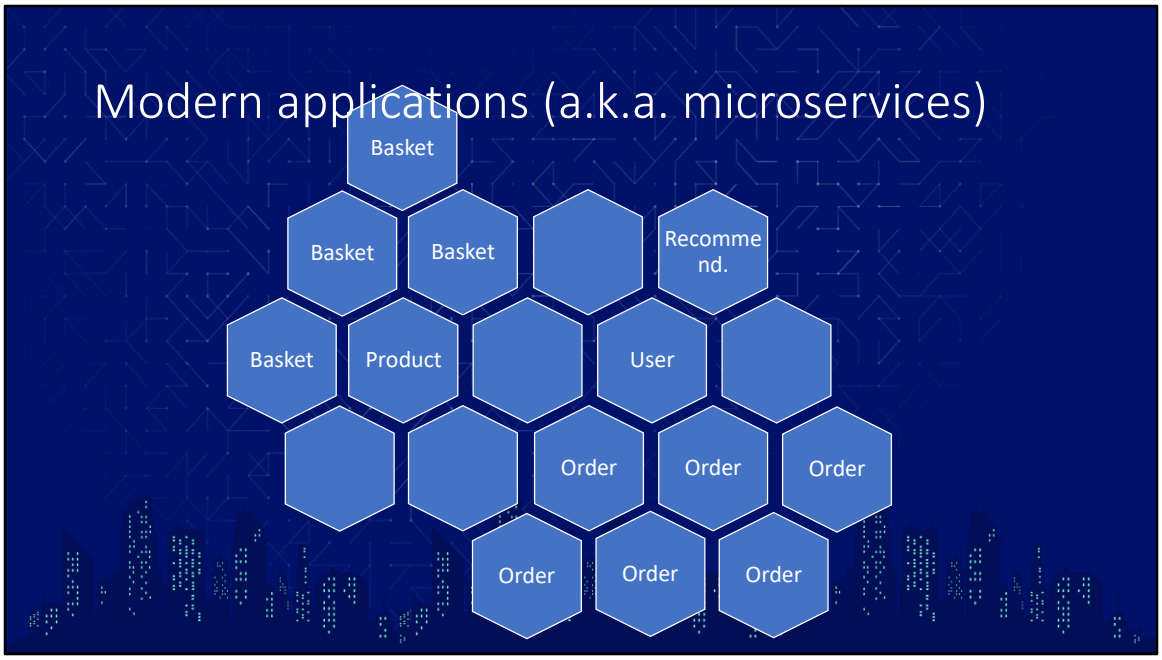
Modern applications (a.k.a. microservices)



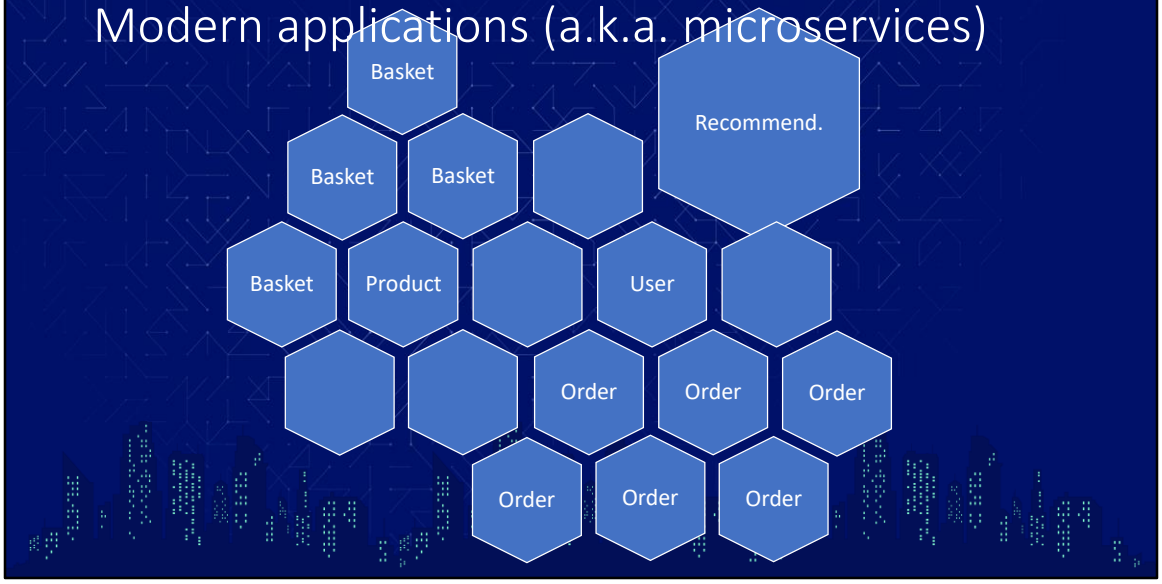
Modern applications (a.k.a. microservices)



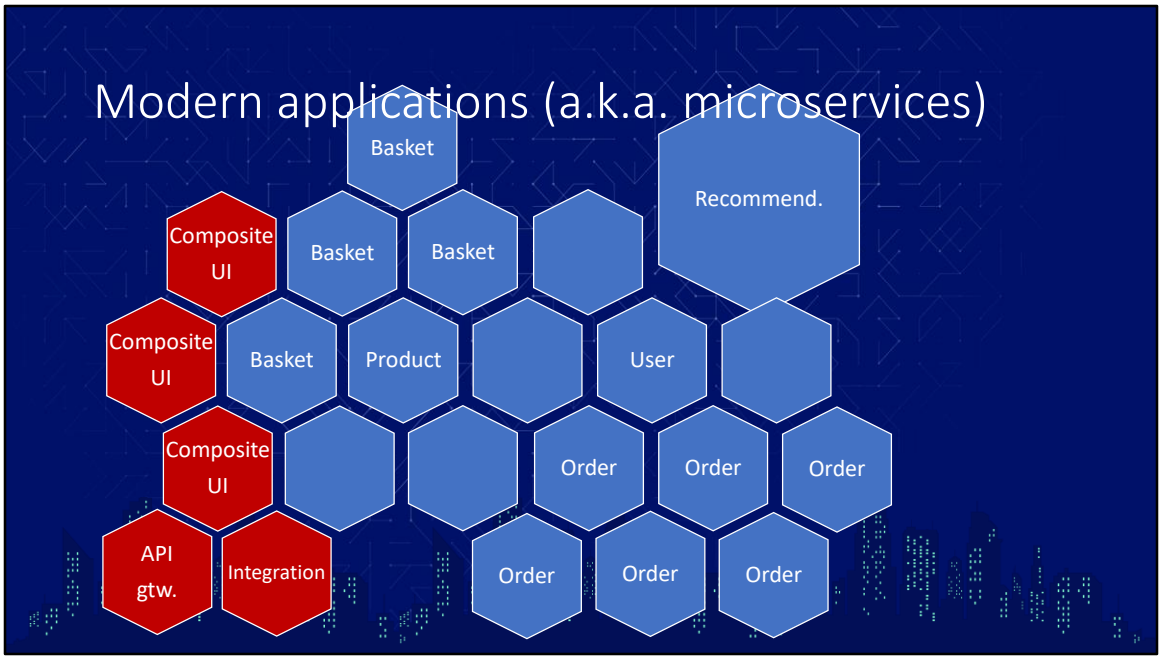
Modern applications (a.k.a. microservices)

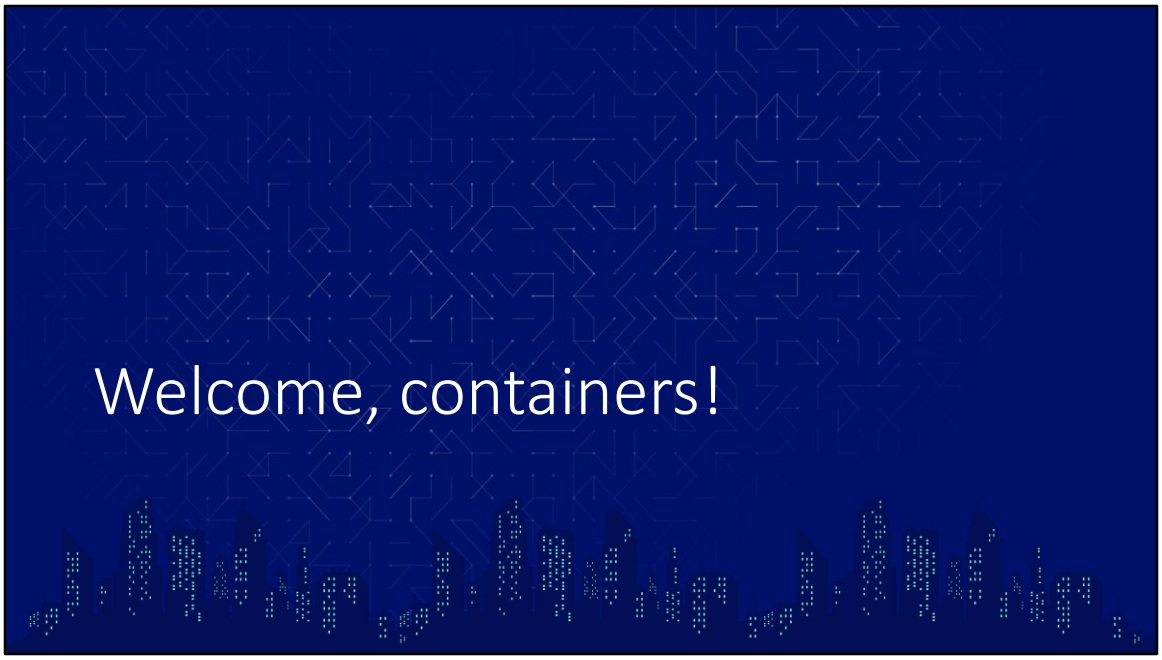


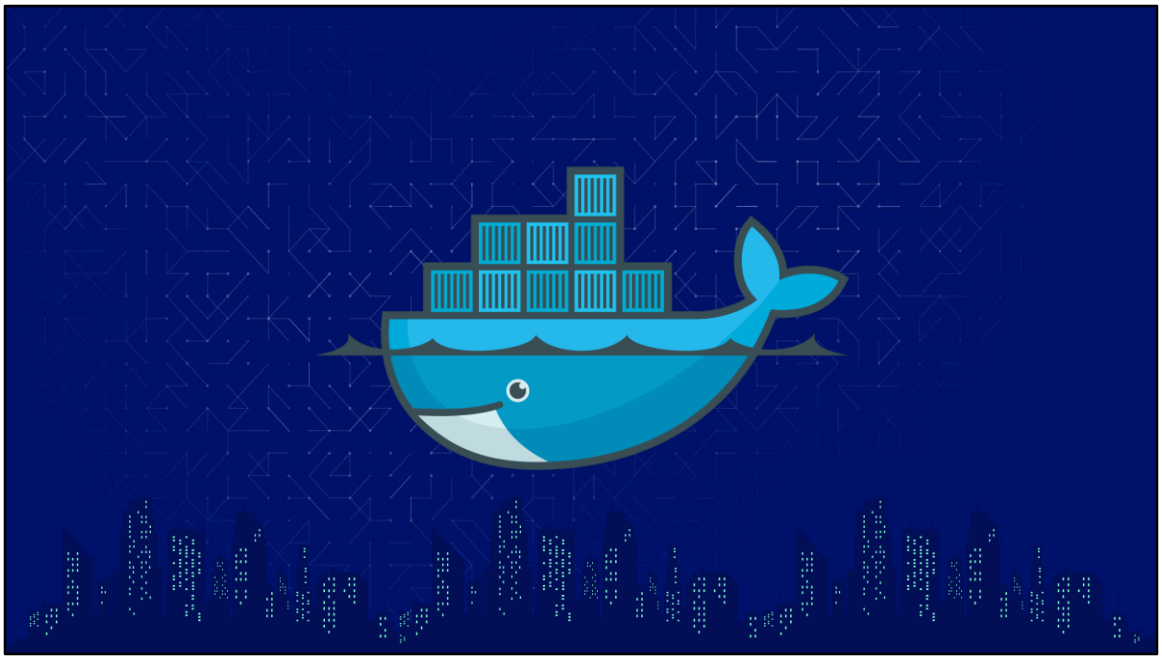
Modern applications (a.k.a. microservices)

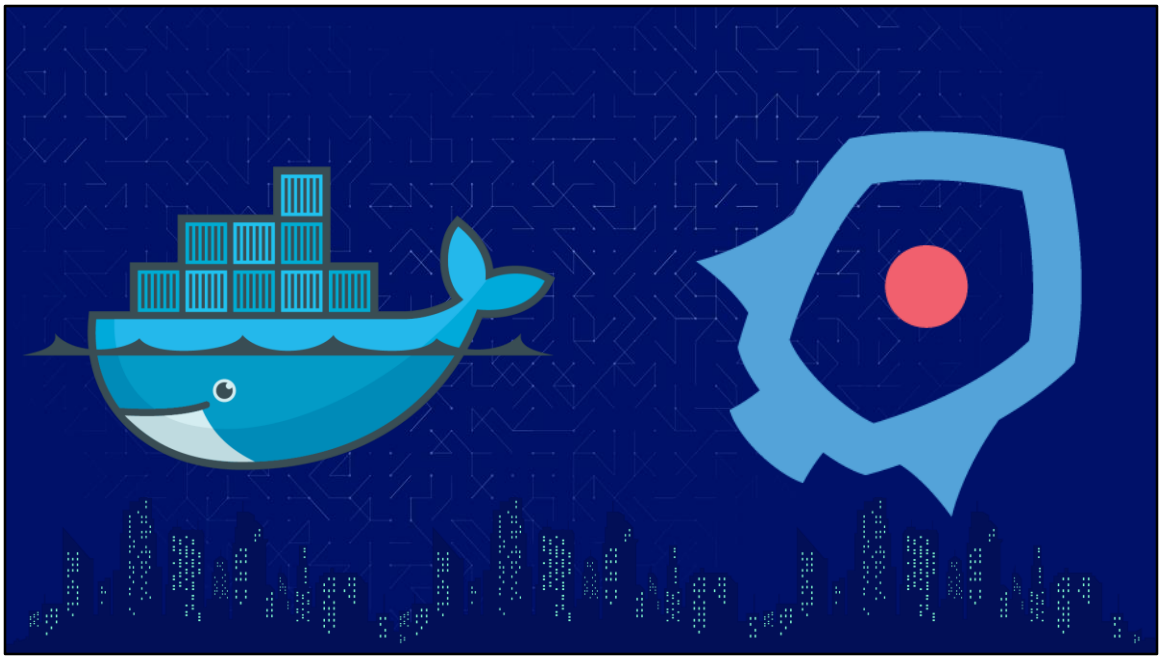


Modern applications (a.k.a. microservices)




Welcome, containers!





Both are at the end of the day platforms for packaging, distributing and running applications.

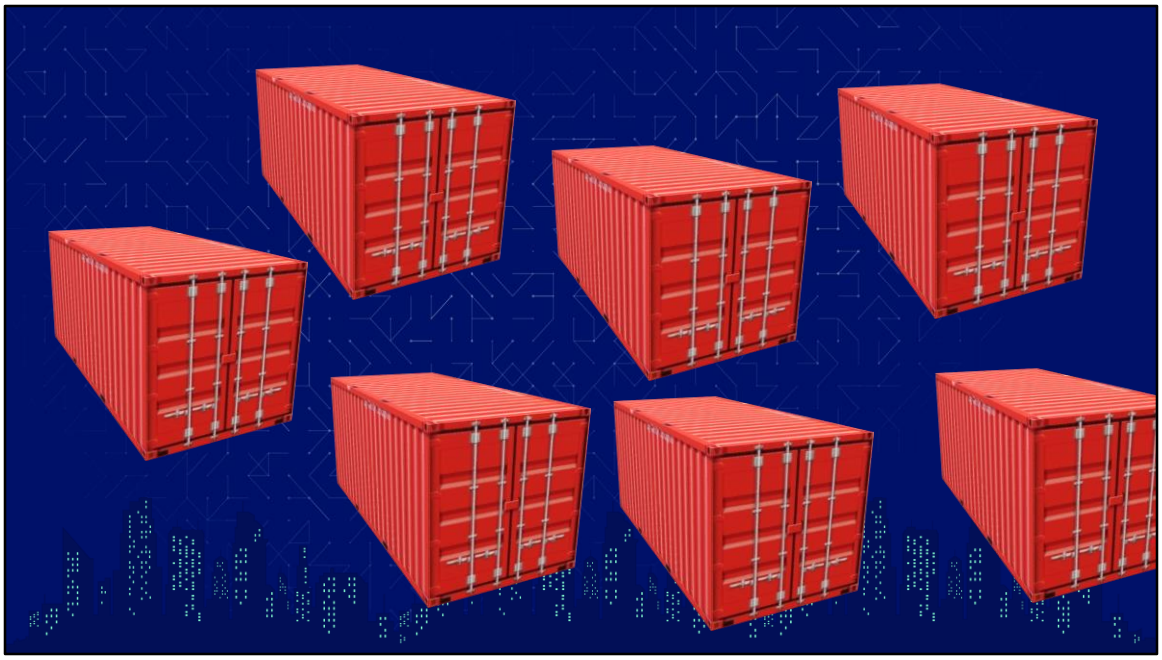

Dockerfile


ACME


ACME
ACME
ACME
ACME
ACME
ACME
ACME
ACME
ACME
ACME

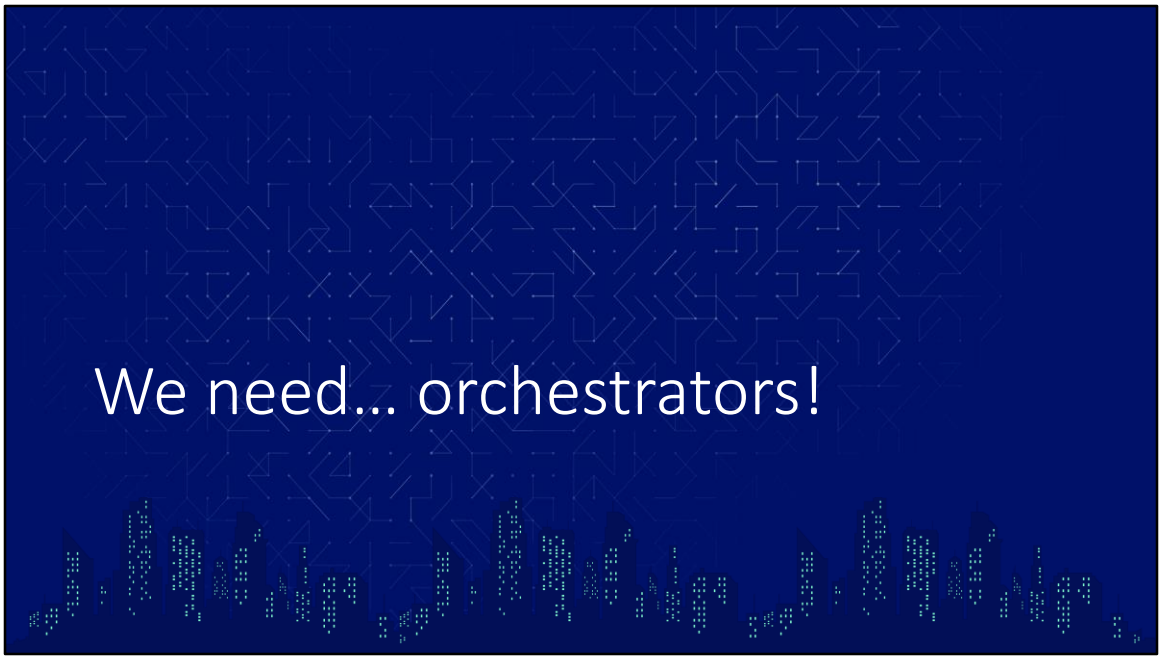

ACME
ACME
ACME
ACME
ACME
ACME
ACME
ACME

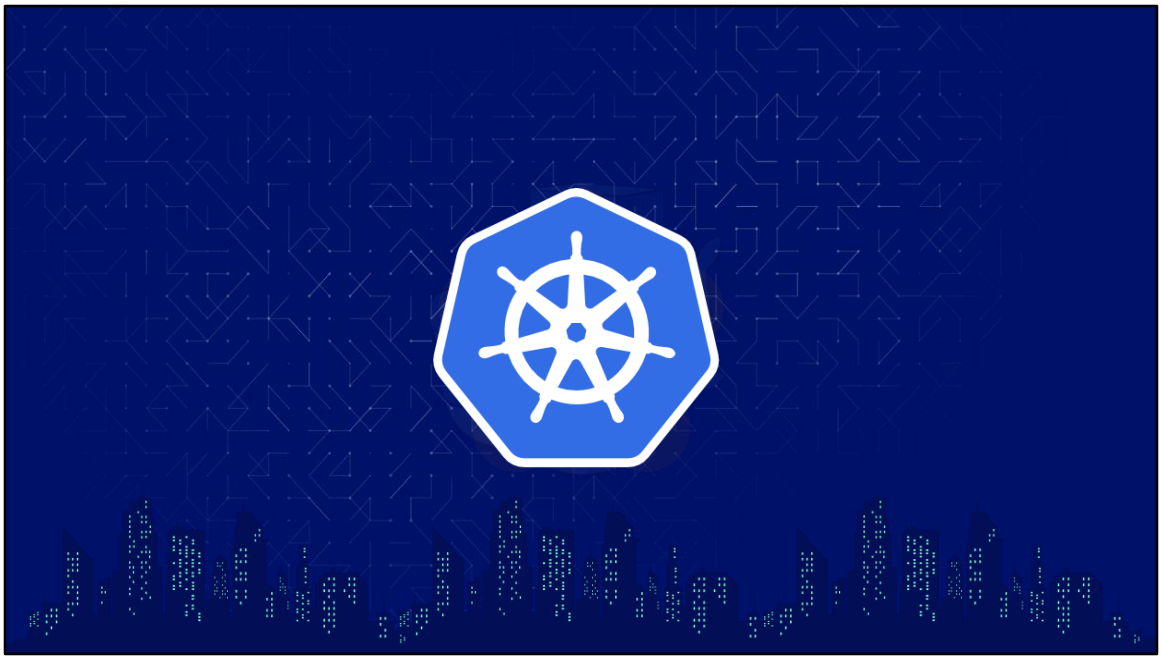

ACME
ACME
ACME
ACME
ACME
ACME
ACME
ACME



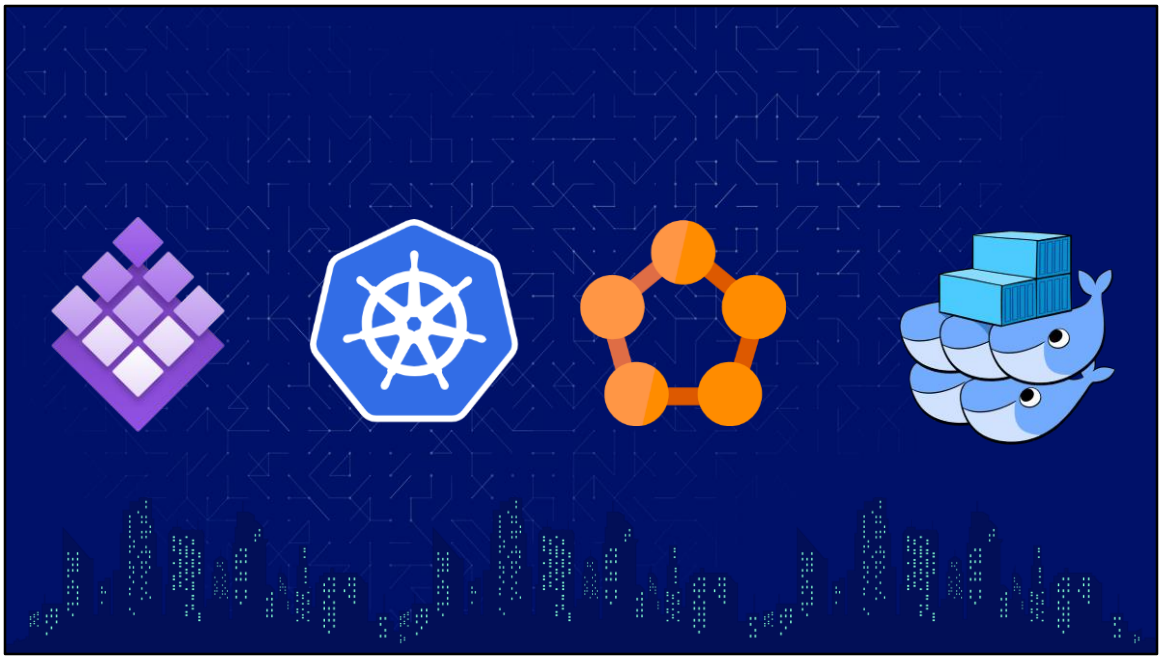
Once you have many containers…

- …many components, many moving parts
- …difficult inter-process communication
- …manual management can be difficult

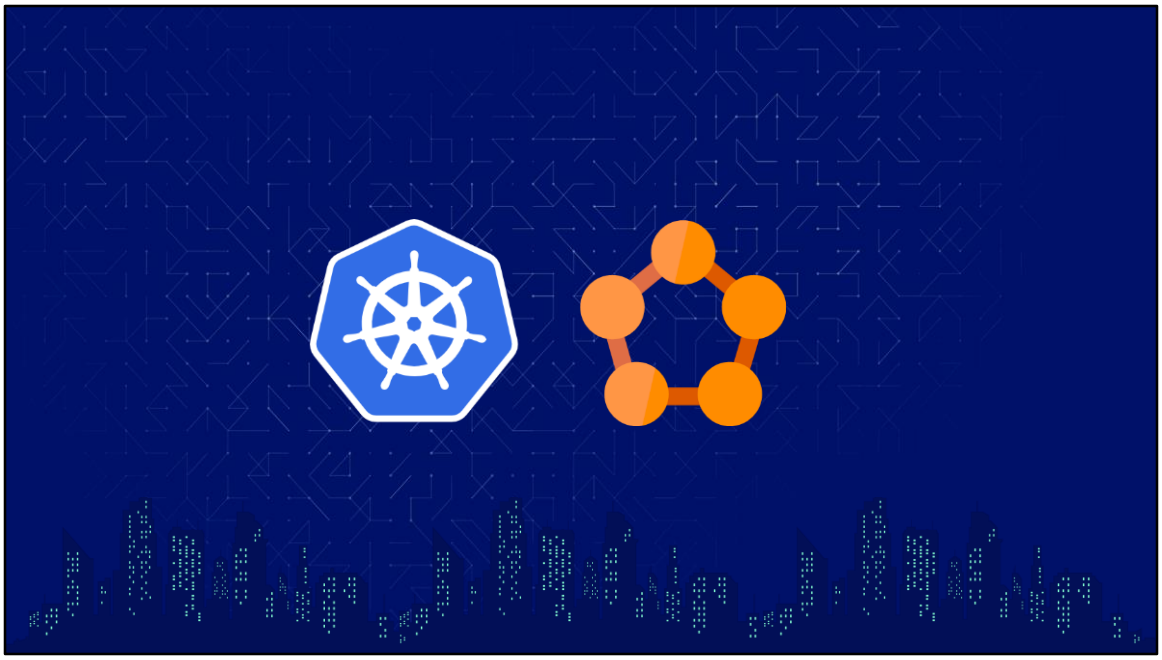

We need… orchestrators!



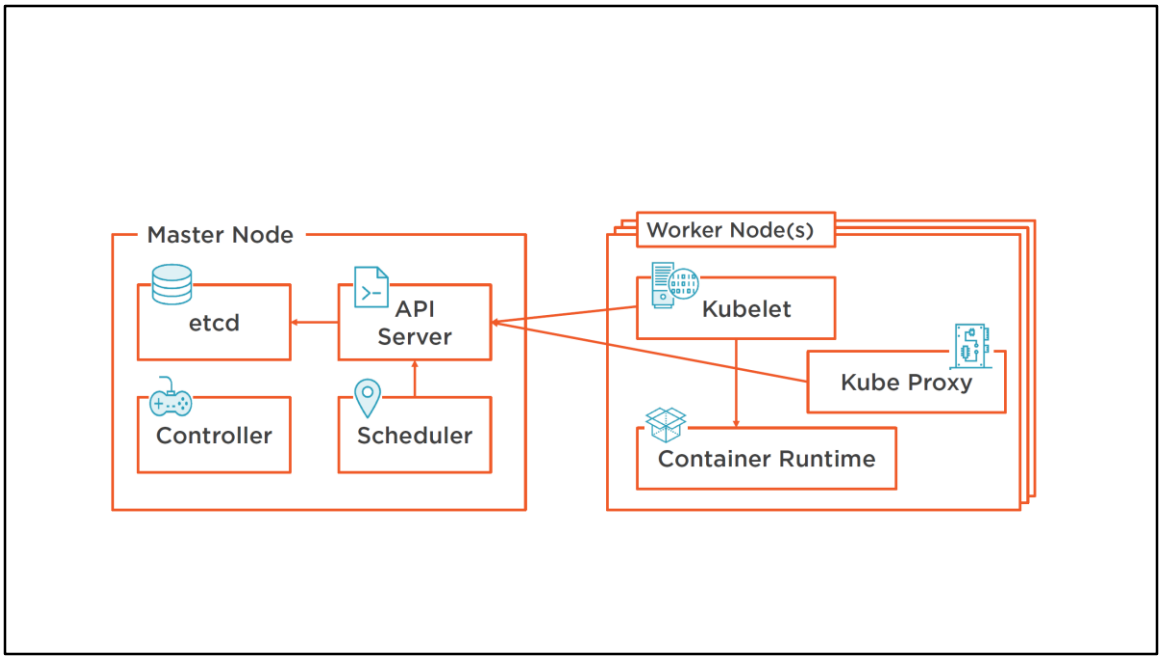
- Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it.
- It enables you to run your software applications on thousands of computer nodes as if all these nodes were a single anonymous computer. It abstracts away the underlying infrastructure and doesn't care whether it contains physical machines or virtual machines.
- When you deploy your applications through Kubernetes, the process is always the same whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply translate to additional amount of resources available to your deployed applications.



- Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it.
- It enables you to run your software applications on thousands of computer nodes as if all these nodes were a single anonymous computer. It abstracts away the underlying infrastructure and doesn't care whether it contains physical machines or virtual machines.
- When you deploy your applications through Kubernetes, the process is always the same whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply translate to additional amount of resources available to your deployed applications.

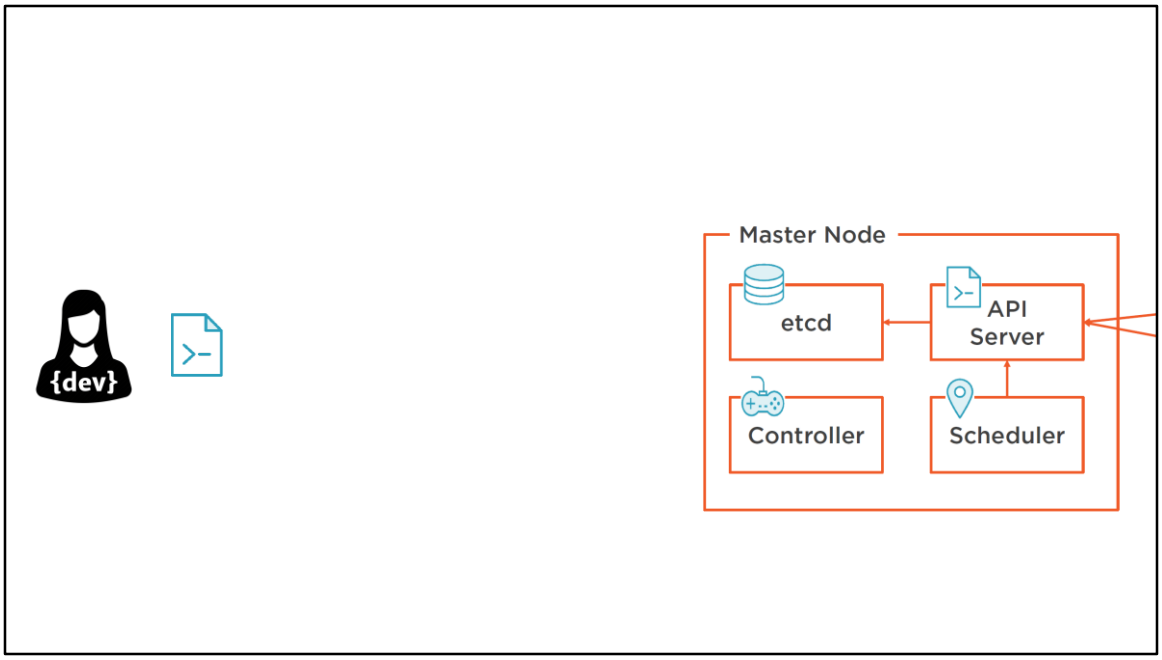


- Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it.
- It enables you to run your software applications on thousands of computer nodes as if all these nodes were a single anonymous computer. It abstracts away the underlying infrastructure and doesn't care whether it contains physical machines or virtual machines.
- When you deploy your applications through Kubernetes, the process is always the same whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply translate to additional amount of resources available to your deployed applications.



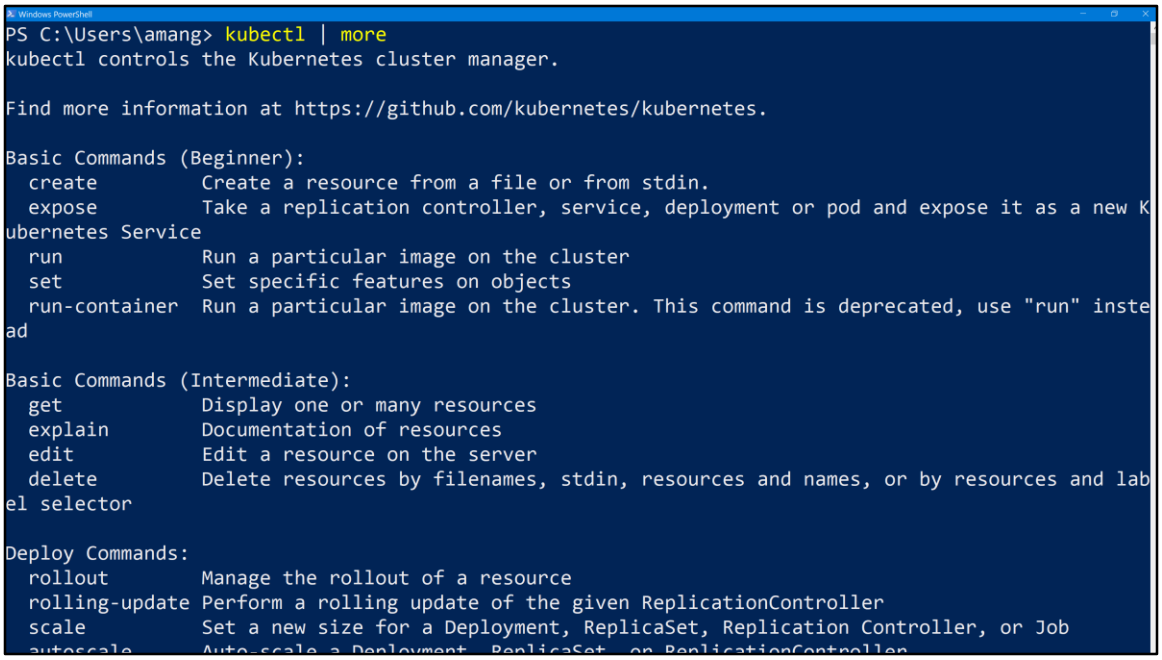
The Kubernetes **API server**, which you and the other control plane components communicate with;
the scheduler, which schedules your applications, basically assigning a worker node to each deployable component of your application;
the controller manager, which performs the cluster-level functions, such as replicating component, keeping a track of worker nodes, handling node failures, and so on;
etcd, which is a reliable distributed data store that persistently stores the cluster configuration.
Note that these components of the control plane hold and control the state of the cluster, but they don't run your applications.

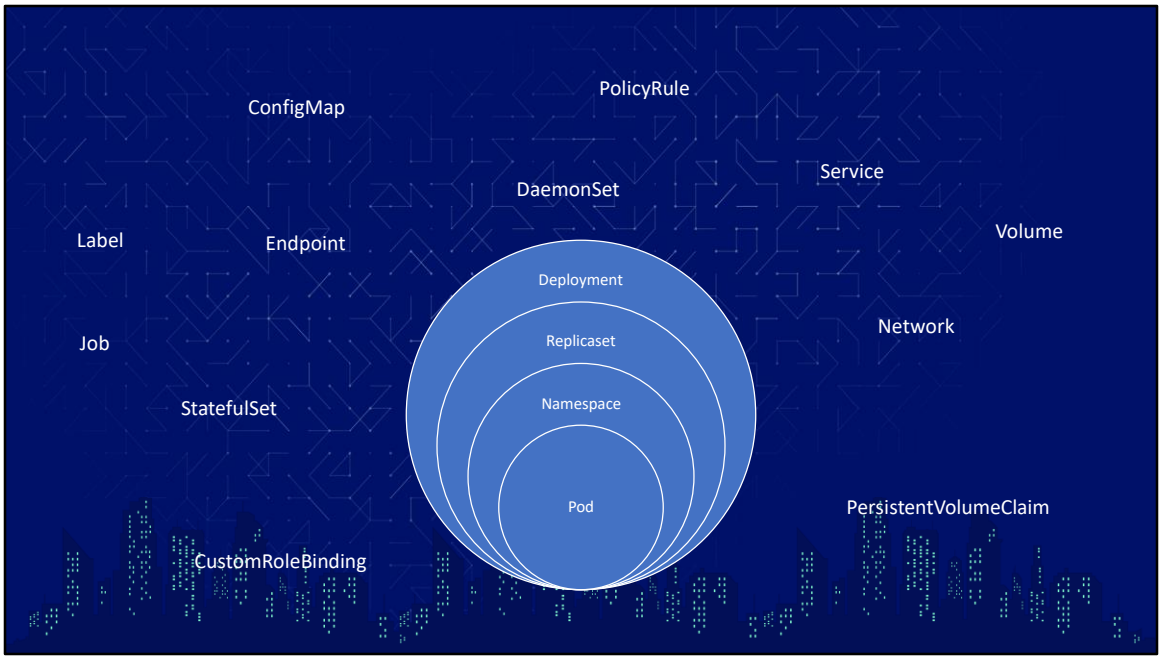
This is done by the worker nodes. The worker nodes are the machines that run your containerized applications. It consists of three important components. The **container runtime**, which can be Docker or rkt;
the kubelet, which talks to the API server and manages containers on its node; and the **Kubernetes service proxy**, which load balances network traffic between the application components



The Kubernetes **API server**, which you and the other control plane components communicate with;
the scheduler, which schedules your applications, basically assigning a worker node to each deployable component of your application;
the controller manager, which performs the cluster-level functions, such as replicating component, keeping a track of worker nodes, handling node failures, and so on;
etcd, which is a reliable distributed data store that persistently stores the cluster configuration.
Note that these components of the control plane hold and control the state of the cluster, but they don't run your applications.

This is done by the worker nodes. The worker nodes are the machines that run your containerized applications. It consists of three important components. The **container runtime**, which can be Docker or rkt;
the kubelet, which talks to the API server and manages containers on its node; and the **Kubernetes service proxy**, which load balances network traffic between the application components


Windows PowerShell
PS C:\Users\amang> kubectl | more
kubectl controls the Kubernetes cluster manager.
Find more information at https://github.com/kubernetes/kubernetes.
Basic Commands (Beginner):
create Create a resource from a file or from stdin.
expose Take a replication controller, service, deployment or pod and expose it as a new K
ubernetes Service
run Run a particular image on the cluster
set Set specific features on objects
run-container Run a particular image on the cluster. This command is deprecated, use "run" inste
ad
Basic Commands (Intermediate):
get Display one or many resources
explain Documentation of resources
edit Edit a resource on the server
delete Delete resources by filenames, stdin, resources and names, or by resources and lab
el selector
Deploy Commands:
rollout Manage the rollout of a resource
rolling-update Perform a rolling update of the given ReplicationController
scale Set a new size for a Deployment, ReplicaSet, Replication Controller, or Job



Pod

A pod is the smallest unit that Kubernetes manages and is the fundamental that the rest of the Kubernetes system is built on.

It is made of one or more containers and the information associated with those containers. Containers within a pod can share files through volumes attached to the containers. A pod has an explicit lifecycle and will always remain on the node in which it was started.

All the containers for a pod will be run on the same node. Any container running within a pod will share the node's network and any other containers in the same pod. Containers within a pod can share files through volumes attached to the containers. A pod has an explicit lifecycle and will always remain on the node in which it was started.

Namespaces

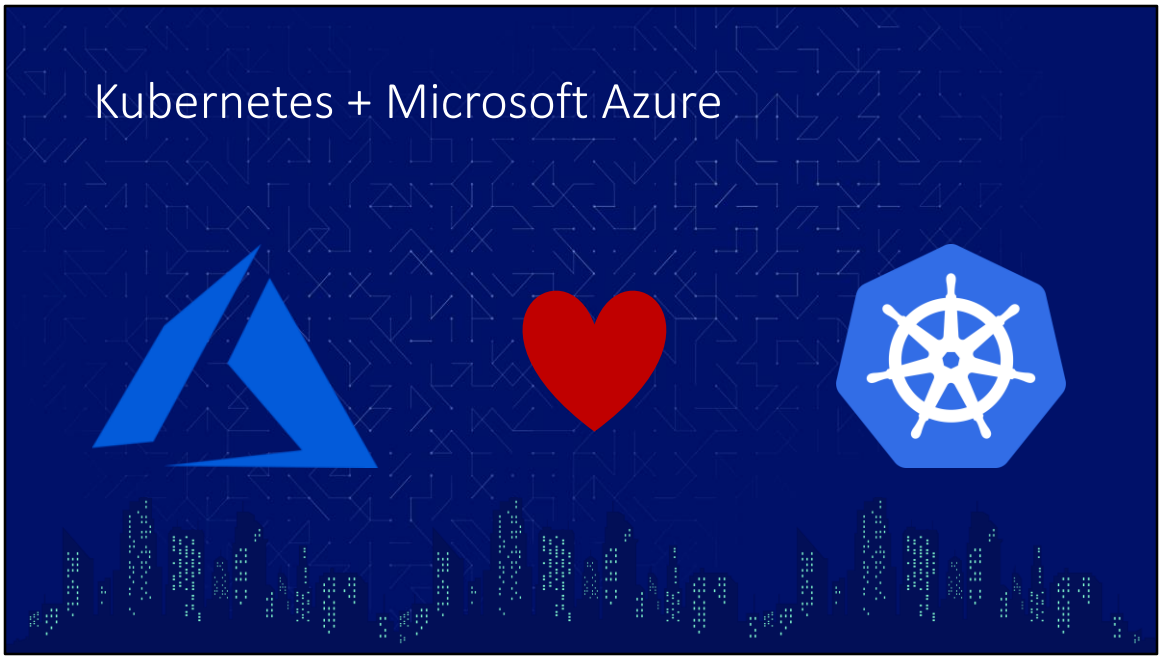
Pods are collected into namespaces, which are used to group pods together for a variety of purposes. For example, to see the status of all the pods in the cluster, you

can run kubectl get pods with the --all-namespaces option.

Namespaces can be used to provide quotas and limits around the resource usage, have an impact on the DNS names that Kubernetes creates internal to the cluster, and in future, even impact access control policies.

ReplicaSet
A ReplicaSet is associated with a pod and indicates how many instances of that pod should be running within that cluster. A ReplicaSet also implies that Kubernetes has a controller that watches the ongoing state and knows how many of your pod to keep running.

This is where Kubernetes is really starting to do the work for you. If you specify three pods in a ReplicaSet and one failed, Kubernetes will automatically schedule and run another for you. A ReplicaSet is commonly wrapped in turn by a deployment



Kubernetes requires unnecessary underlying plumbing

➔Azure Kubernetes Services does that for you

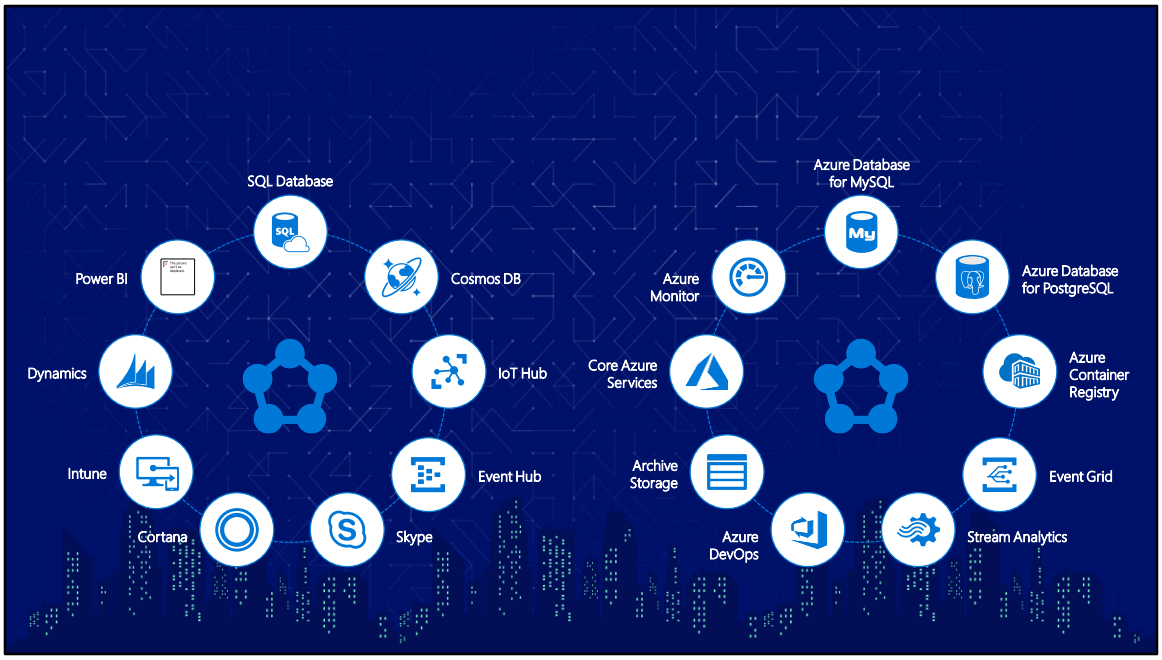
The master node is fully managed by Microsoft!

Why AKS?

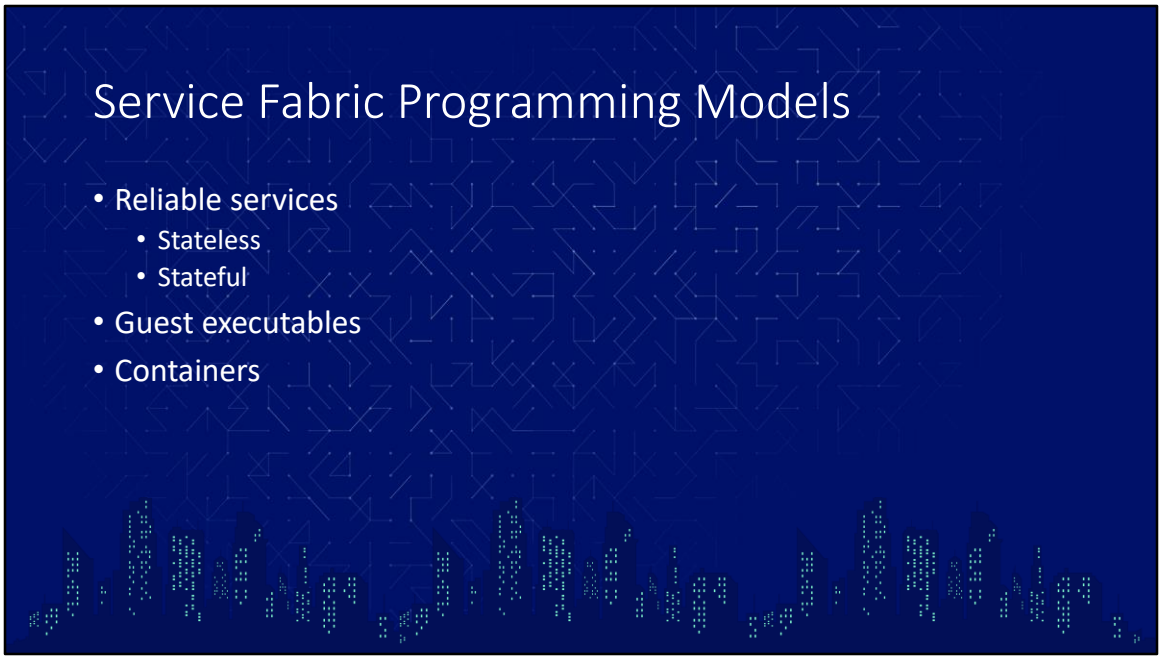
Simplifies deployment, management and operations of Kubernetes

Eliminates the burden of ongoing operations and maintenance by provisioning, upgrading and scaling resources on demand


But… what about Service Fabric?


SQL Database
Power BI
Cosmos DB
Dynamics
IoT Hub
Intune
Event Hub
Cortana
Skype
Azure Database for MySQL
Azure Monitor
Azure Database for PostgreSQL
Core Azure Services
Azure Container Registry
Archive Storage
Event Grid
Azure DevOps
Stream Analytics


Dev machine
Any cloud
On-premises
Service Fabric
Standalone
Bring your own infrastructure
Azure
Service Fabric Cluster
Dedicated Azure clusters
Azure
Service Fabric Mesh
Fully managed
microservices platform

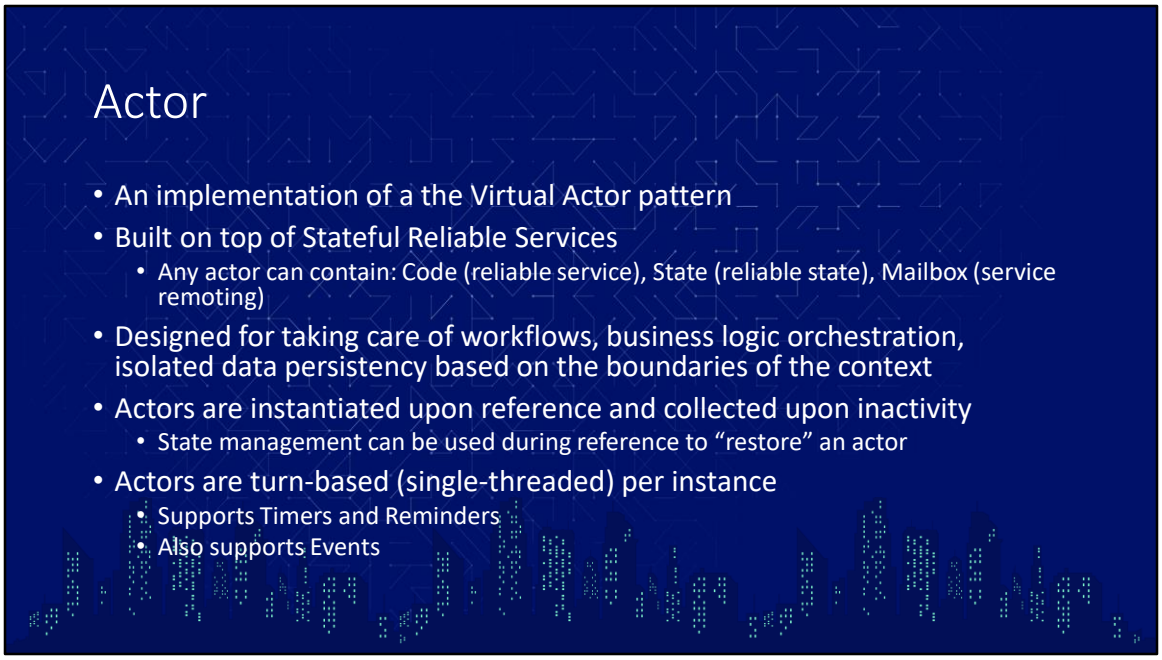


Reliable Service & Reliable Actor Services are built on top of .NET SDK frameworks provided by Azure Service Fabric. As such, they have programmatic access to service and service-fabric functionality. This includes accessing the Service Fabric Naming Service (resolve service instance URL by name) and integration with the instance lifecycle events, among others. Additionally, service instances are create as objects under the management of Service Fabric, instead of individual processes, which allows for higher density within the host cluster. These models will be discussed more in detail shortly

Guest Executable Services allow you to create a service that hosts an arbitrary executable. This executable can be written in any language you choose, and Service Fabric will take care of execution management tasks like ensuring the application is running and other orchestration tasks. Because guest executables are not built on top of the Service Fabric APIs, they have limited access internally to the Service Fabric functionality.

Container Services can be thought of as an application-within-an-application, in that multiple services can be placed within a container. This includes supporting Reliable Services which include Stateless Services within a container (Linux only), Stateful

Services within a container (Windows only), or arbitrary guest executables within a container.



An Actor is an isolated Entity and is defined as a fundamental unit of computation, which embodies three essential parts, processing, storage, and communications. It sounds a bit confusing, abstract, and not very clear how it solves multithreading issues, so let's go over this. **Objective:** To introduce the Reliable Actors framework available in Azure Service Fabric development

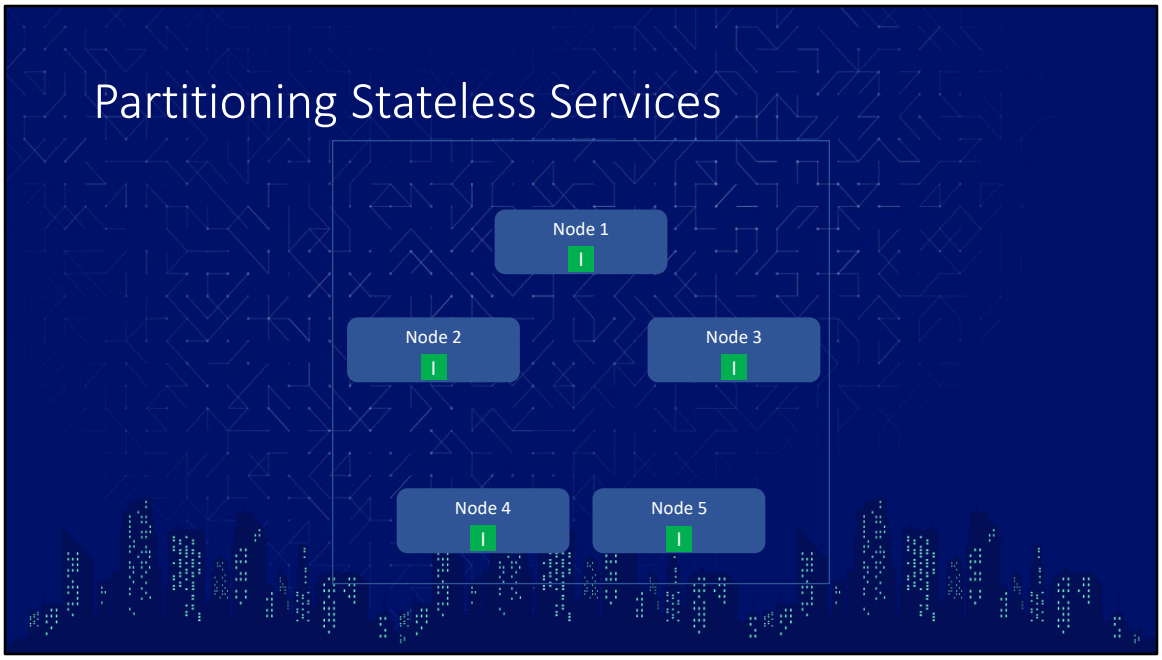
Notes: The Reliable Actors framework is a Virtual Actor implementation that is built on top of stateful Reliable Services.
In a Virtual Actor implementation, an Actor is a unit of both logic and state that is managed by the framework. Client applications reference an actor by a unique ID. The framework manages the allocation and lifetime of individual actor instances, including “resurrecting” an actor instance which has gone dormant due to a period of inactivity.

An Actor is an abstract entity, which contains code to execute, this is the same code you would normally write using any methodology, language, or libraries you're used to. It has a persistent state, the state can only be accessed by the Actor and no one else. It's not accessible even by the Actor of the same type, but only by different instances of the Actor. An Actor has a mailbox, which means it's discoverable by other

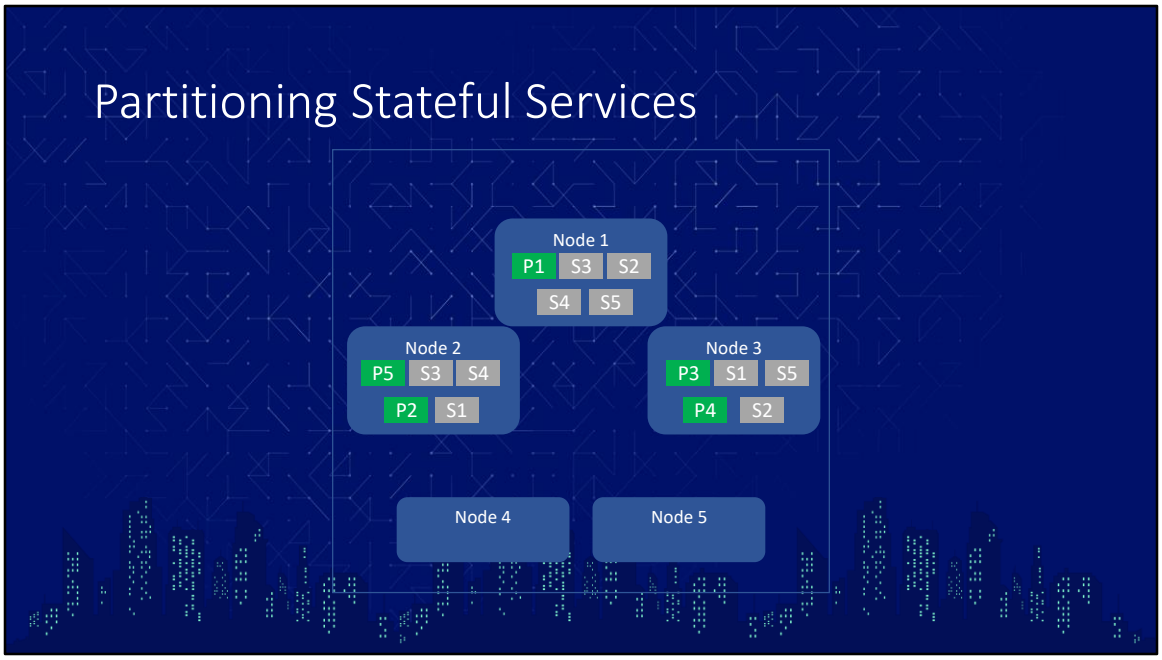
Actors or different parts of the application like services. The mailbox stores messages for the Actor, which it will process at some point. And the last thing an Actor can do is send messages to other Actors, including itself.

Actor instances provide a turn-based, single-threaded access model, making it possible to not have to deal with concurrency issues in a distributed programming model. With this single-threaded behavior, Timers and Reminders both offer support for running code after a function call has completed. Timers and reminders differ mainly in that Timers cannot “re-awaken” a time-expired actor, whereas Reminders will continue to execute, “waking” a dormant actor instance as necessary, until the actor unregisters the reminder or the actor is explicitly deleted.

Finally, actors also support raising events back to the calling application. Because of the nature of distributed programming, these events are not strictly guaranteed to reach their listeners (if reliable messaging is required, a queue-based or similar solution is recommended.)



In a stateless environment, scalability and availability generally achieved by adding instances
Place an instance of the service in each Node
Add Nodes to scale out



Providing scalability and reliability is a little trickier in a Stateful Service. In order to do so, we need to maintain distributed replicas of the state in each partition, keeping track of a primary Partition and a number of Secondary Partitions.
Reliability exists because if Service Fabric is aware of a single node going down, it can rebalance using the Secondary partition replicas located on other nodes, promoting them to Primary nodes as necessary and creating new additional Secondary replicas.
Scalability is present as long as the number of Nodes approaches the number of Partitions (when scaling up.) Once you have more Nodes than Partitions, the additional Nodes do not generally contribute to expanded capacity in the system.

Here we have a Stateful Service running in 5 Partitions with 3 Replicas per partition, distributed across 3 Nodes. (Animation) If we add two more Nodes to the service, Service Fabric rebalances the existing partitions, easing the load on the original 3 Nodes.

Collections

e.g. List, Dictionary, Queue, SortedList, Stack

Concurrent Collections

Multi-threaded

e.g. ConcurrentQueue, ConcurrentStack

Reliable Collections can be thought of as the natural evolution of the System.Collections classes: a new set of collections that are designed for the cloud and multi-computer applications without increasing complexity for the developer. As such, Reliable Collections are:

Replicated: State changes are replicated for high availability.

Persisted: Data is persisted to disk for durability against large-scale outages (for example, a datacenter power outage).

Asynchronous: APIs are asynchronous to ensure that threads are not blocked when incurring IO.

Transactional: APIs utilize the abstraction of transactions so you can manage

multiple Reliable Collections within a service easily.

Today, Microsoft.ServiceFabric.Data.Collections contains three collections:
Reliable Dictionary: Represents a replicated, transactional, and asynchronous collection of key/value pairs. Similar to ConcurrentDictionary, both the key and the value can be of any type.
Reliable Queue: Represents a replicated, transactional, and asynchronous strict first-in, first-out (FIFO) queue. Similar to ConcurrentQueue, the value can be of any type.
Reliable Concurrent Queue: Represents a replicated, transactional, and asynchronous best effort ordering queue for high throughput. Similar to the ConcurrentQueue, the value can be of any type.

https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-replica-lifecycle

Application Modernization Strategy


Let’s compare apples to apples

Kubernetes' truth matrix

Good stuff

- Everything is customizable
- Stable, well documented, amazing community support

Less good stuff

- Very infra-savvy
- Very steep learning curve
- High complexity for a developer

Service Fabric's truth matrix

Good stuff

- Out-of-the-box support for full .NET Framework
- Supports lift & shift migrations for legacy .NET stack

Less good stuff

- Extremely complex at runtime (as in there's a lot of boiler-plate code within your app)
- Containers are second-class citizens

K8s

Joe Beda, Brendan Burns and Craig McLuckie

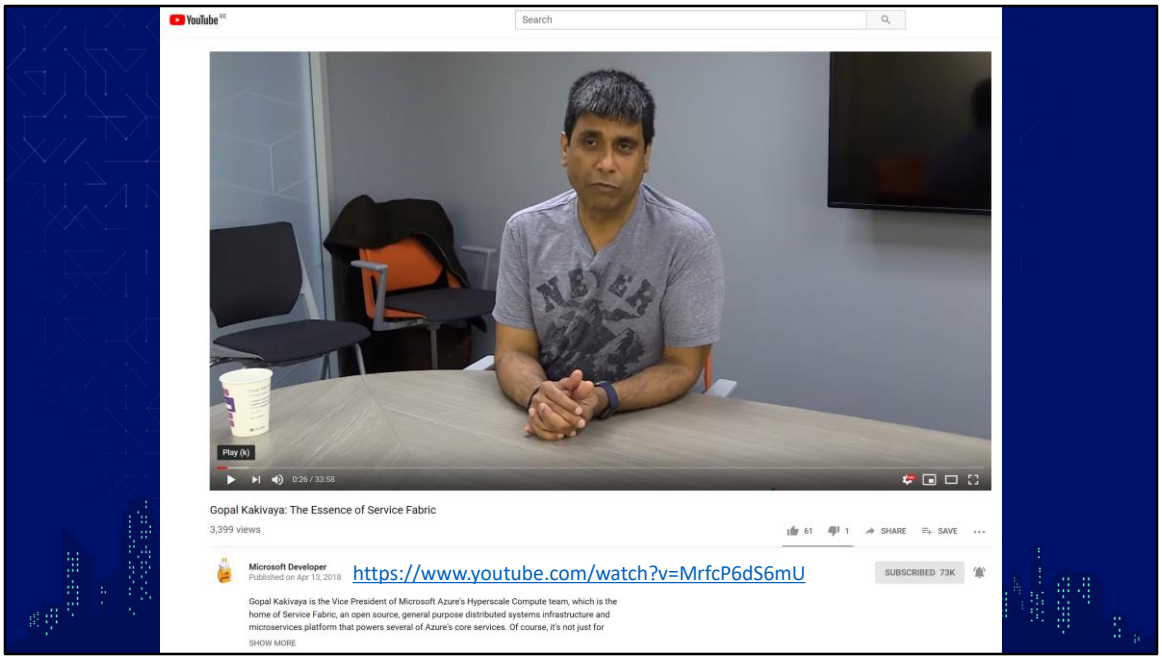
Borg is focused on power-users

Borg orchestrates plain old processes running huge, statically linked executables

Kubernetes orchestrates containers

Kubernetes API server is designed to be much more general and microservices based

Service Fabric


YouTube
Search
Play (k)
0:26 / 33:58
Gopal Kakivaya: The Essence of Service Fabric
3,399 views
61
1
SHARE
SAVE
Microsoft Developer
Published on Apr 13, 2018
https://www.youtube.com/watch?v=MrfcP6dS6mU
SUBSCRIBED 73K
Gopal Kakivaya is the Vice President of Microsoft Azure's Hyperscale Compute team, which is the home of Service Fabric, an open source, general purpose distributed systems infrastructure and microservices platform that powers several of Azure's core services. Of course, it's not just for
SHOW MORE


PDC2008
BB03
SQL Data Services
Under The Hood
Gopal Kakivaya
Partner Architect
Microsoft Corporation
https://channel9.msdn.com/Blogs/pdc2008/BB03

K8s

Joe Beda, Brendan Burns and Craig McLuckie

Borg is focused on power-users

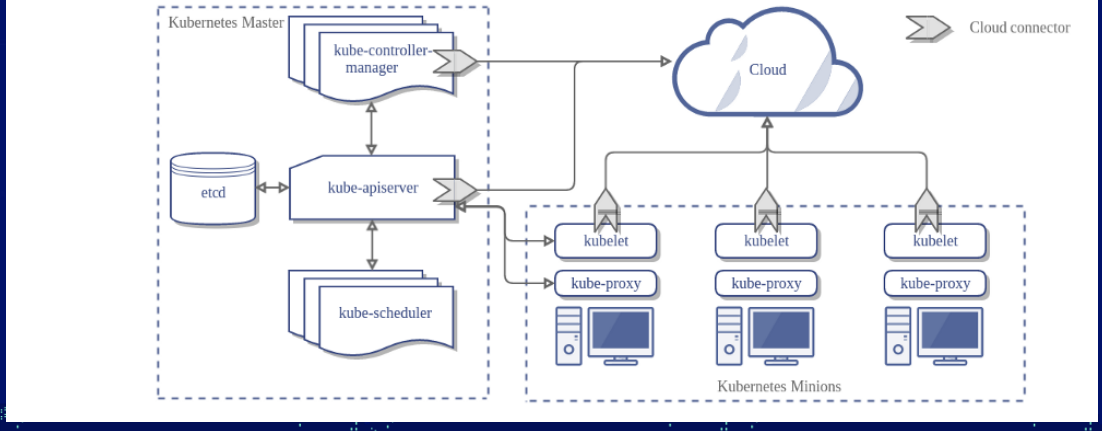
Borg orchestrates plain old processes running huge, statically linked executables

Kubernetes orchestrates containers

Kubernetes API server is designed to be much more general and microservices based

Service Fabric

Architectural design – K8s

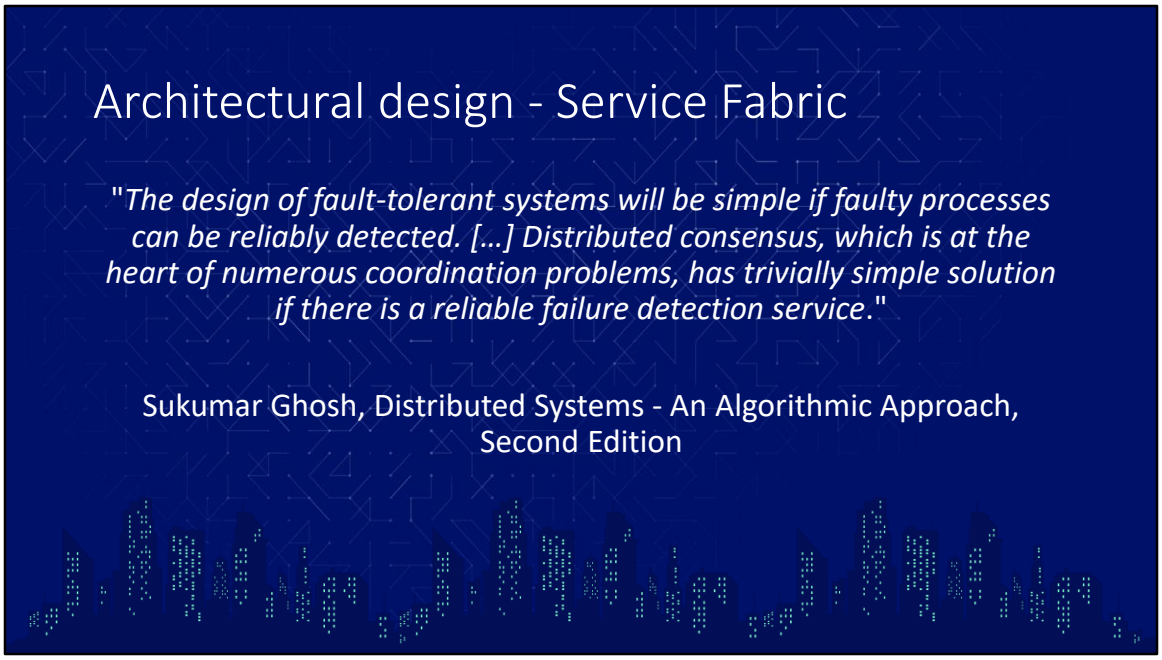


Paradoxically, Kubernetes is a heavily centralized system. It has an API Server in the middle, and agents called Kubelets installed on all worker nodes. All Kubelets communicate with the API Server, which keeps the state of the cluster persisted in a centralized repository: the etcd cluster. The etcd cluster is a fairly simple Raft-backed distributed KV-store. Most production configurations use between 3 and 7 nodes for etcd clusters.

To maintain the cluster membership information, all the Kubelets are required to maintain a connection with the API Server and send heartbeats every period of time (for example 10 seconds).

Such an approach can have negative effects on the scalability.

https://github.com/kubernetes/community/blob/master/keps/sig-node/0009-node-heartbeat.md



The authors of Service Fabric, instead of using a distributed consensus protocol like raft or paxos (which does not scale well with growing number of nodes), or using a centralized store for cluster state (which might use distributed consensus underneath, and which represents a separate point of failure), have focused on designing a fully distributed system. Within Service Fabric's architecture, this is referred to as the Federation Subsystem.

The Federation Subsystem is responsible for giving a consistent answer to the core question: is the specific node part of the system? In the centralized approach discussed earlier, this answer was given by the API Server based on the heartbeats received from all the nodes. In Service Fabric, the nodes are organized in a ring and heartbeats are only sent to a small subset of nodes called the neighborhood.

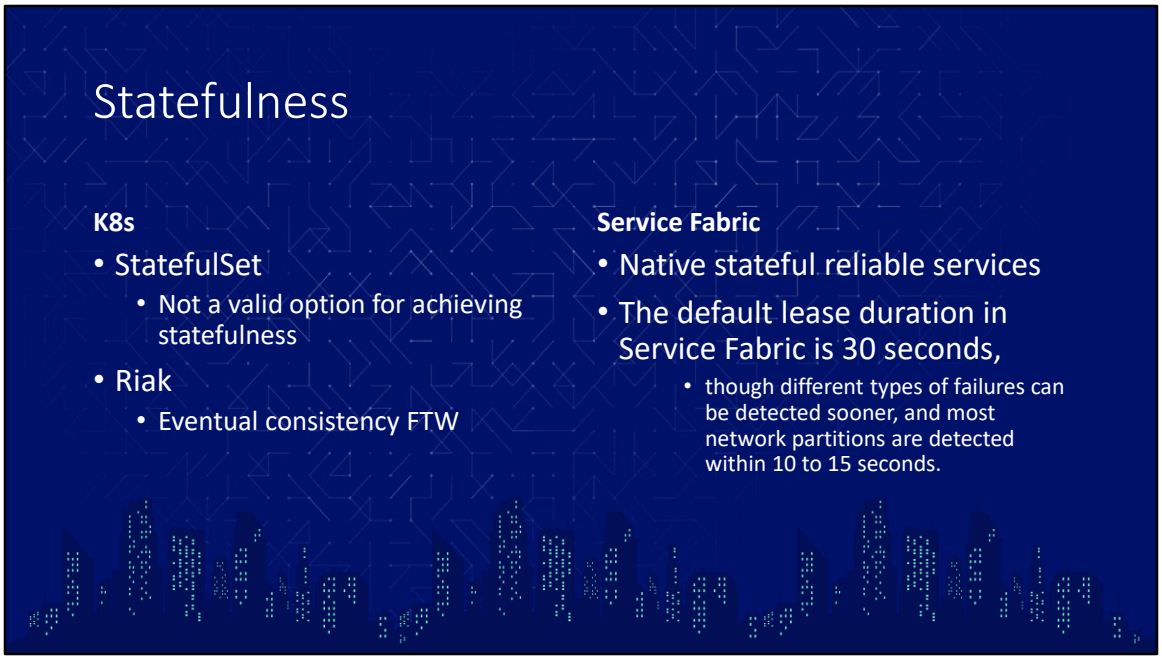
Core infrastructure services

K8s

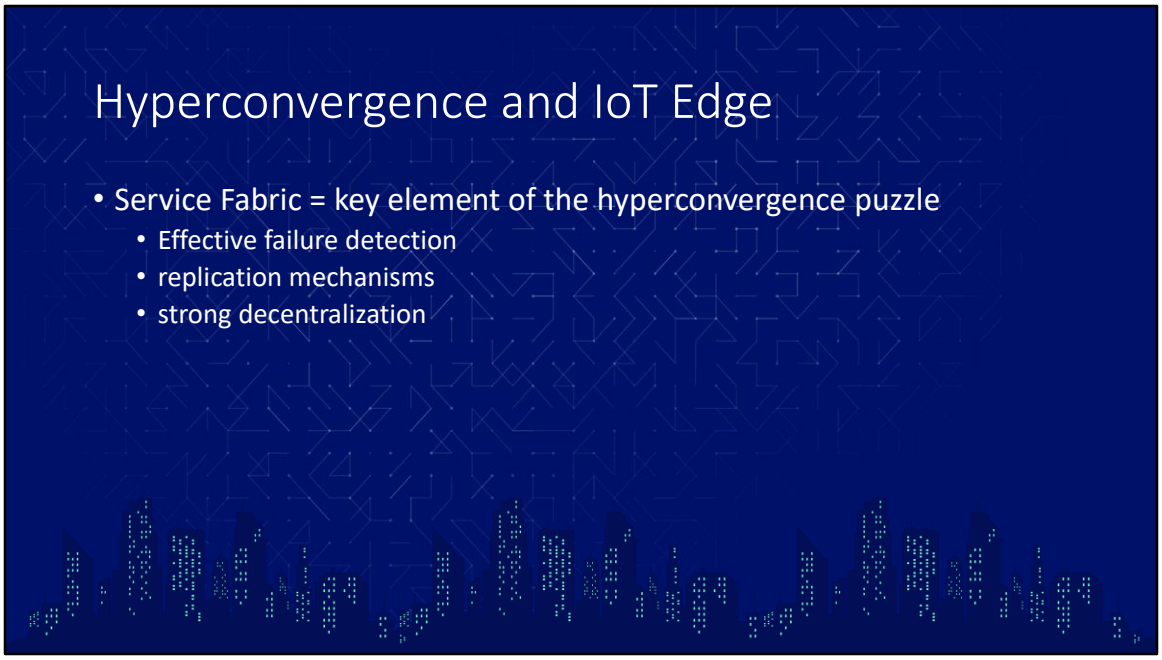
- Everything goes inside the master

Service Fabric

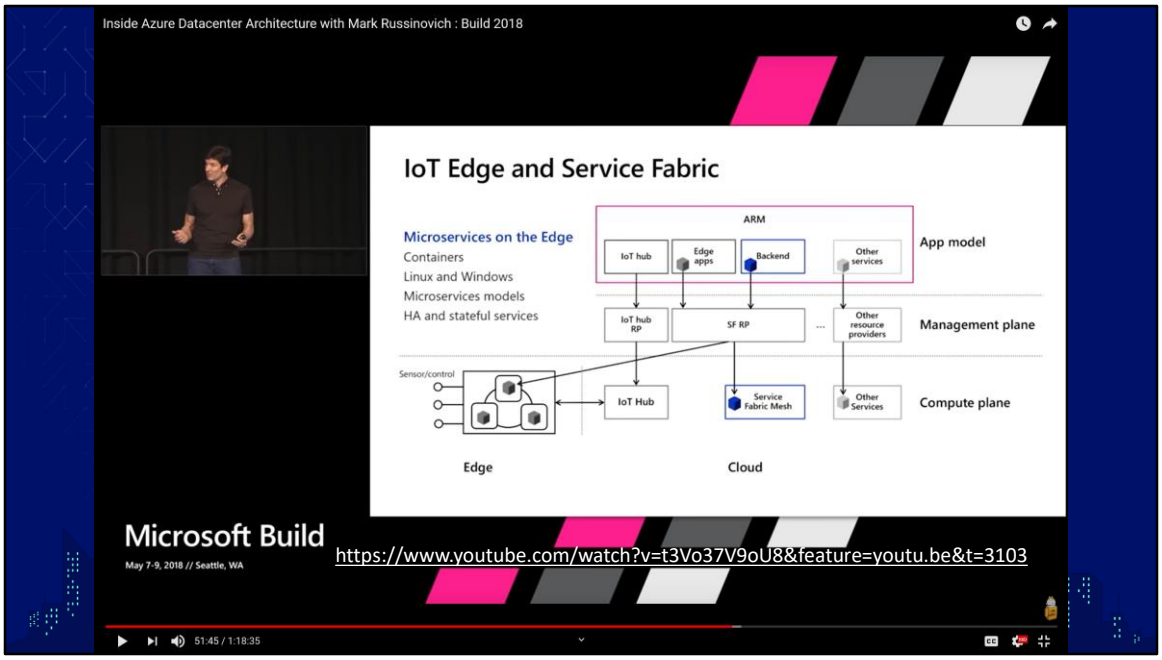
- A system service is exactly like any application service
  - Distributed across the cluster
  - Respect placement constraints



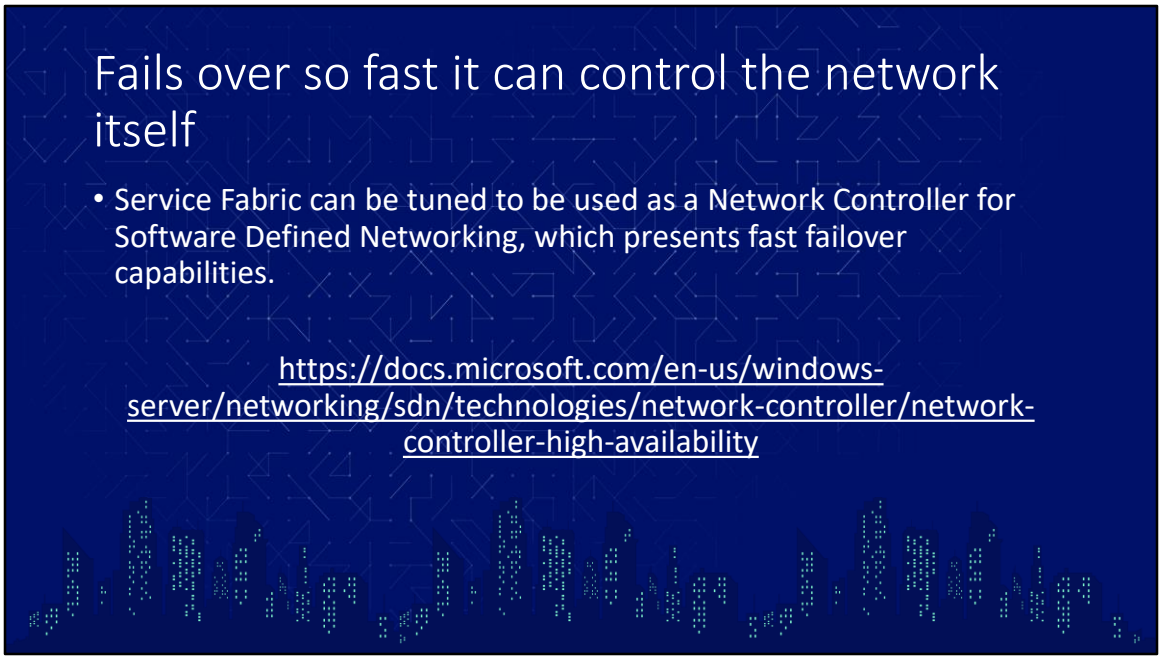
Service Fabric has no competition when it comes to orchestrating mission-critical stateful workloads. It is a strongly consistent platform that is able to provide guarantees like RTO < 30 seconds in case of single rack/availability zone failure, or RTO < 60 seconds in case of entire region outage, with RPO = 0 guarantee, while maintaining high performance and extremely low latency from extensive use of RAM and ephemeral SSDs.
Because Service Fabric was designed to run both stateless and stateful workloads, it tries to detect dead nodes ASAP. Its design allows sending heartbeats much more often in order to detect and correct failure. 5 minutes of unavailability is unacceptable for a stateful service in many scenarios. The default lease duration in Service Fabric is 30 seconds, though different types of failures can be detected sooner, and most network partitions are detected within 10 to 15 seconds.



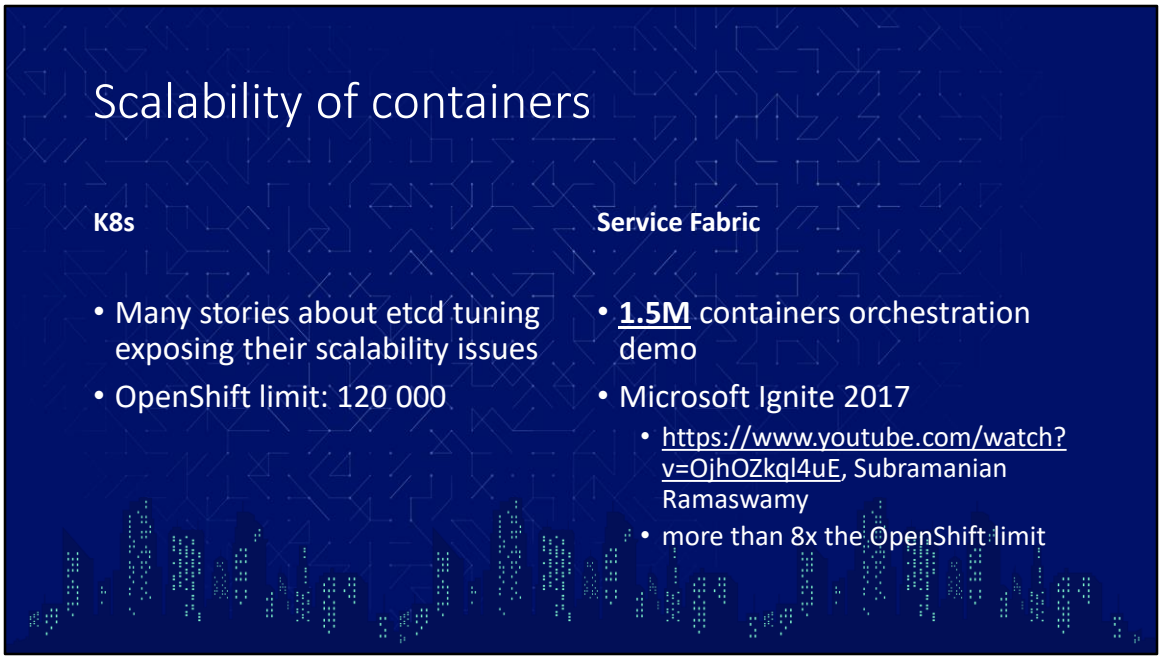
Effective failure detection and replication mechanisms combined together with strong decentralization enable efficient use of ephemeral storage, networking resources, and computing resources, making Service Fabric a key element of the hyperconvergence puzzle. Service Fabric can efficiently run on the commodity hardware or even can form clusters using the small IoT Edge devices



. This capability was recently demonstrated in Mark Russinovich's "Die Hard" demo.



On the other hand, Service Fabric can be tuned to be used as a Network Controller for Software Defined Networking, which presents fast failover capabilities.



- If you browse the web, there are many stories about etcd tuning exposing their scalability issues.
- Meanwhile, Subramanian Ramaswamy ran the following demo during the Ignite conference – orchestrating 1 million containers, which is more than 8x the OpenShift limit (120 000).

Geo-distribution support

K8s

- etcd scalability issues
- cross-regional strong consistency configs unfeasible

Service Fabric (Cluster)

- Living examples: Cortana
- Fault domain redesign
  - e.g. fd:/westeurope/1
  - Intra/inter-region failure detection
- Requires at least 3 regions

Other Azure services for container needs

Find the Azure service for your container needs

YOU WANT TO:	USE THIS:
Scale and orchestrate Linux containers using Kubernetes	Azure Kubernetes Service (AKS)
Deploy web apps or APIs using Linux containers in a PaaS environment	Azure App Service
Elastically burst from your Azure Kubernetes Service (AKS) cluster	Azure Container Instances
Run repetitive compute jobs using containers	Azure Batch
Lift, shift, and modernize .NET applications to microservices using Windows Server containers	Azure Service Fabric
Store and manage container images across all types of Azure deployments	Azure Container Registry

https://azure.microsoft.com/en-us/overview/containers/

Should I Orchestrate My Containers on Service Fabric or AKS?

If you

- only have a handful of services…
- don't need A/B testing, canary releases, ring-based deployments…
- don't care about money…

…don't use containers

Should I Orchestrate My Containers on Service Fabric or AKS?

If you really want containers…

- Kubernetes clearly wins Mr./Mrs. Popularity prize
- Limitations pick the “winner”
- Consider whether you want to learn Linux infra.

Belgian-beer discussion facts

- Service Fabric:
  - First class citizen for Azure
- AKS:
  - Much larger community
- Service Fabric:
  - Great Visual Studio tooling
- AKS:
  - Ops-specific experience

Alex’s favorite?