Kubernetes on Azure 101 Workshop

Rui Félix Pereira

2020.05.05

Containers & Kubernetes workshops

Containers 101

- Intro to Containers
- Pack your app in a custom Docker image
- Run a Docker container locally
- Using a private container registry

Kubernetes 101

- Intro to K8S
- K8S on Azure
- Create your K8S cluster
- Pods & Deployments
- Deploy & scale your app
- Organize with Namespaces
- Expose your app using Services
- Traffic rules with Ingress controllers

Kubernetes 201

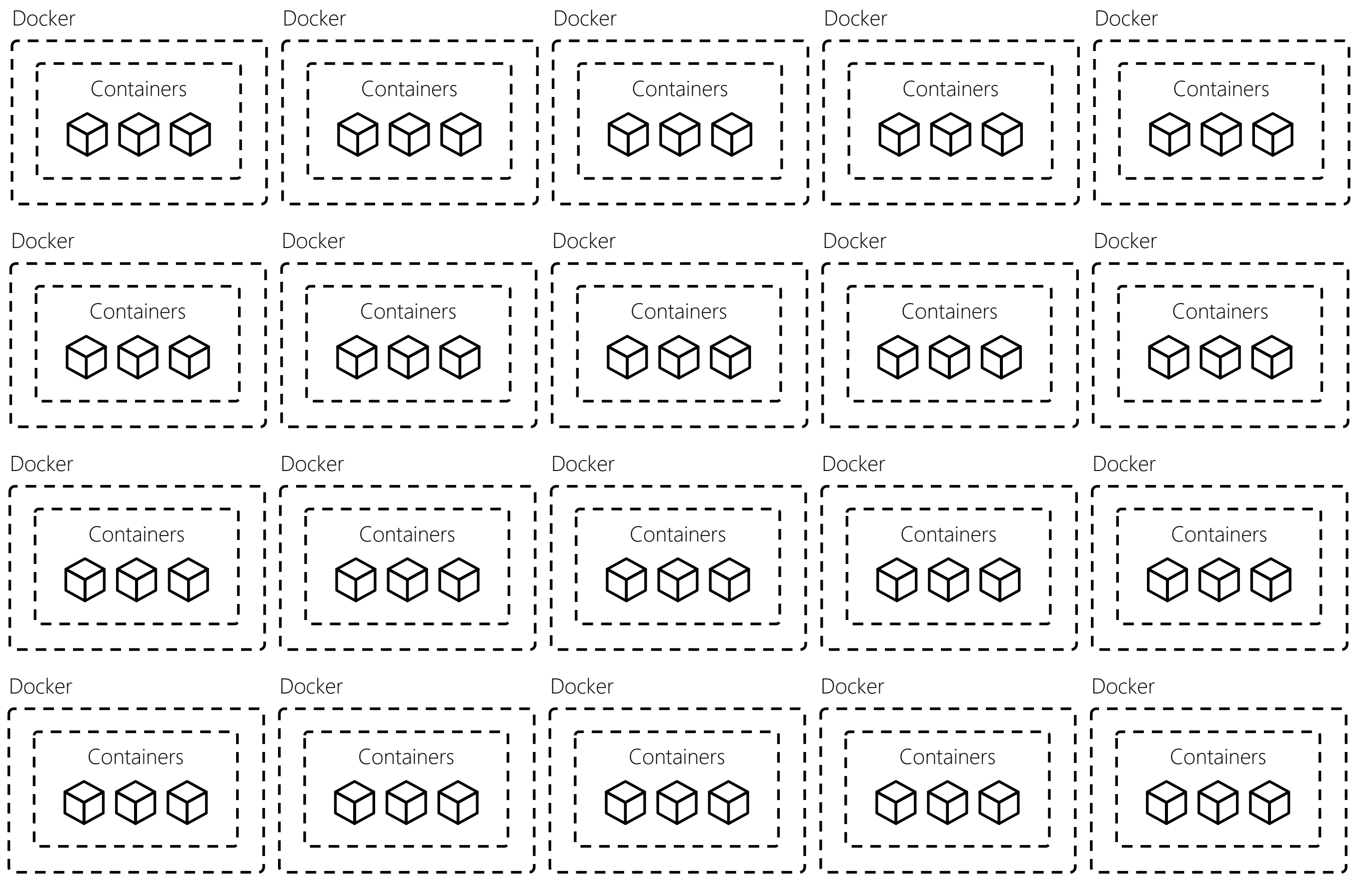
- Using Persistent volumes
- Role based access control (RBAC)
- Manage configurations
- Manage Secrets
- K8S Operators
- Package apps with Helm

Introduction to Kubernetes

Containers are great but "at scale" become very challenging

- Standardize Build
- DevOps CI / CD
- Run Anywhere
- Rapid deployment
- Compute Density & Scale

Deploy, manage and update 1000s of containers in production



Container management & orchestration “at scale”

Scheduling

Affinity
anti-affinity

Health
monitoring

Failover

Scaling

Networking

Service
discovery

Coordinated
app upgrades

Kubernetes: the industry leading orchestrator

Portable

Public, private, hybrid, multi-cloud

Extensible

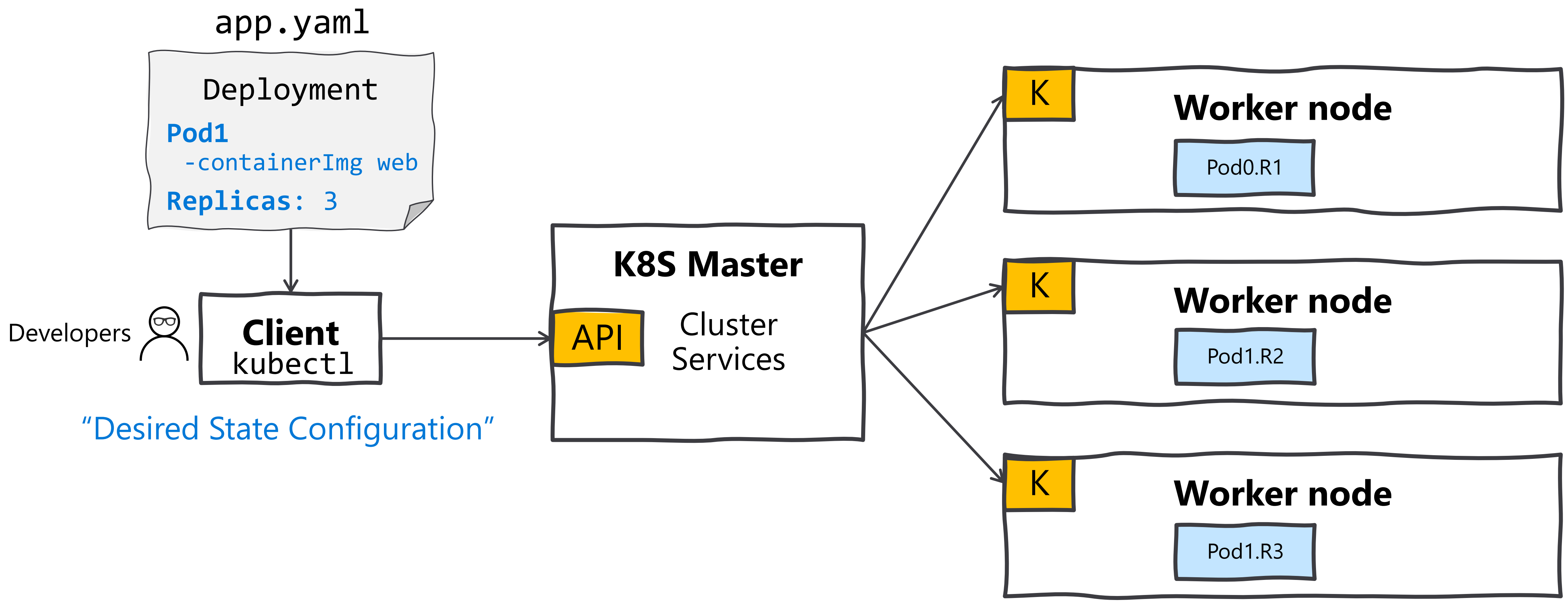
Modular, pluggable, hookable, composable

Self-healing

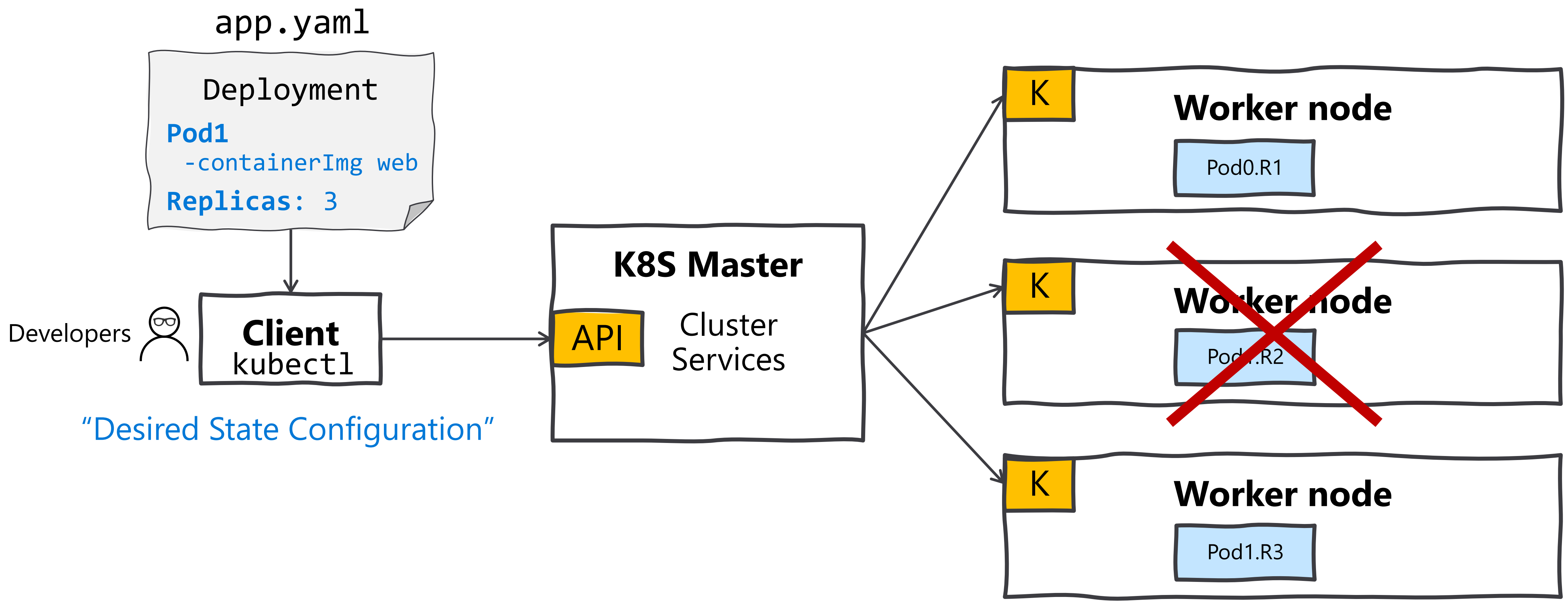
Auto-placement, auto-restart, auto-replication, auto-scaling

CLOUD NATIVE COMPUTING FOUNDATION

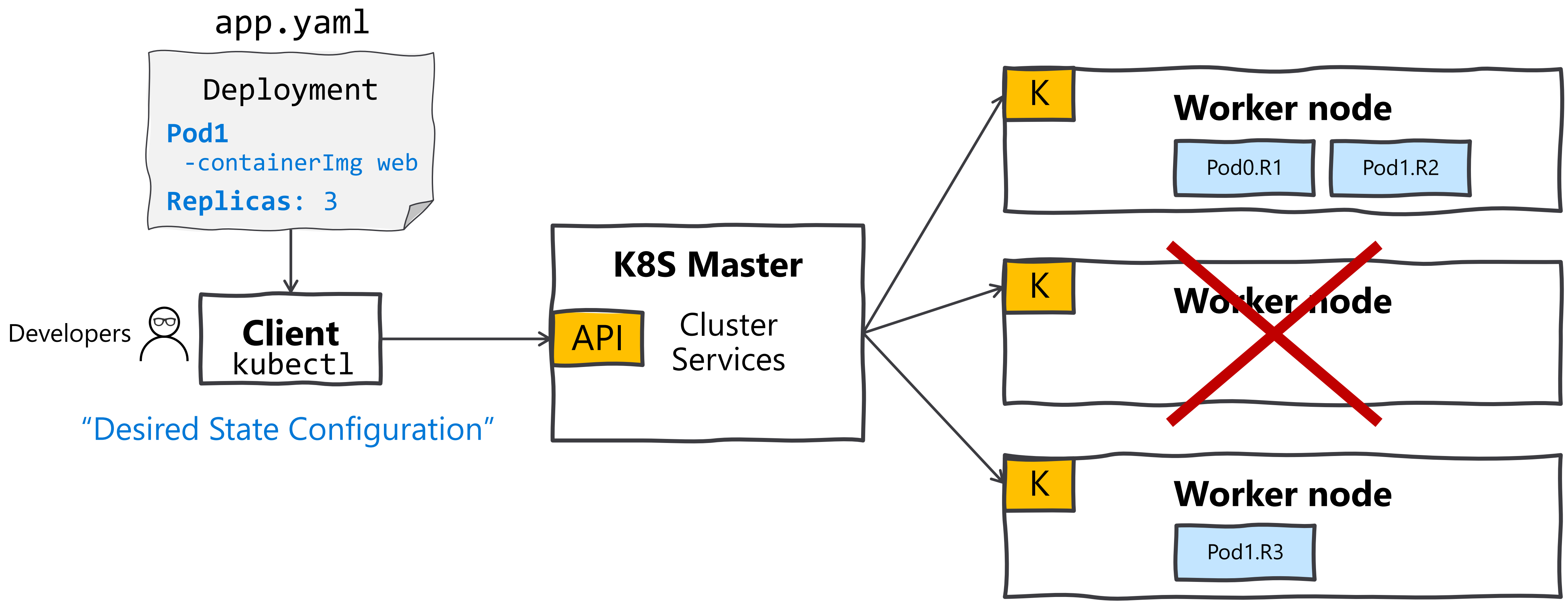
Kubernetes 101



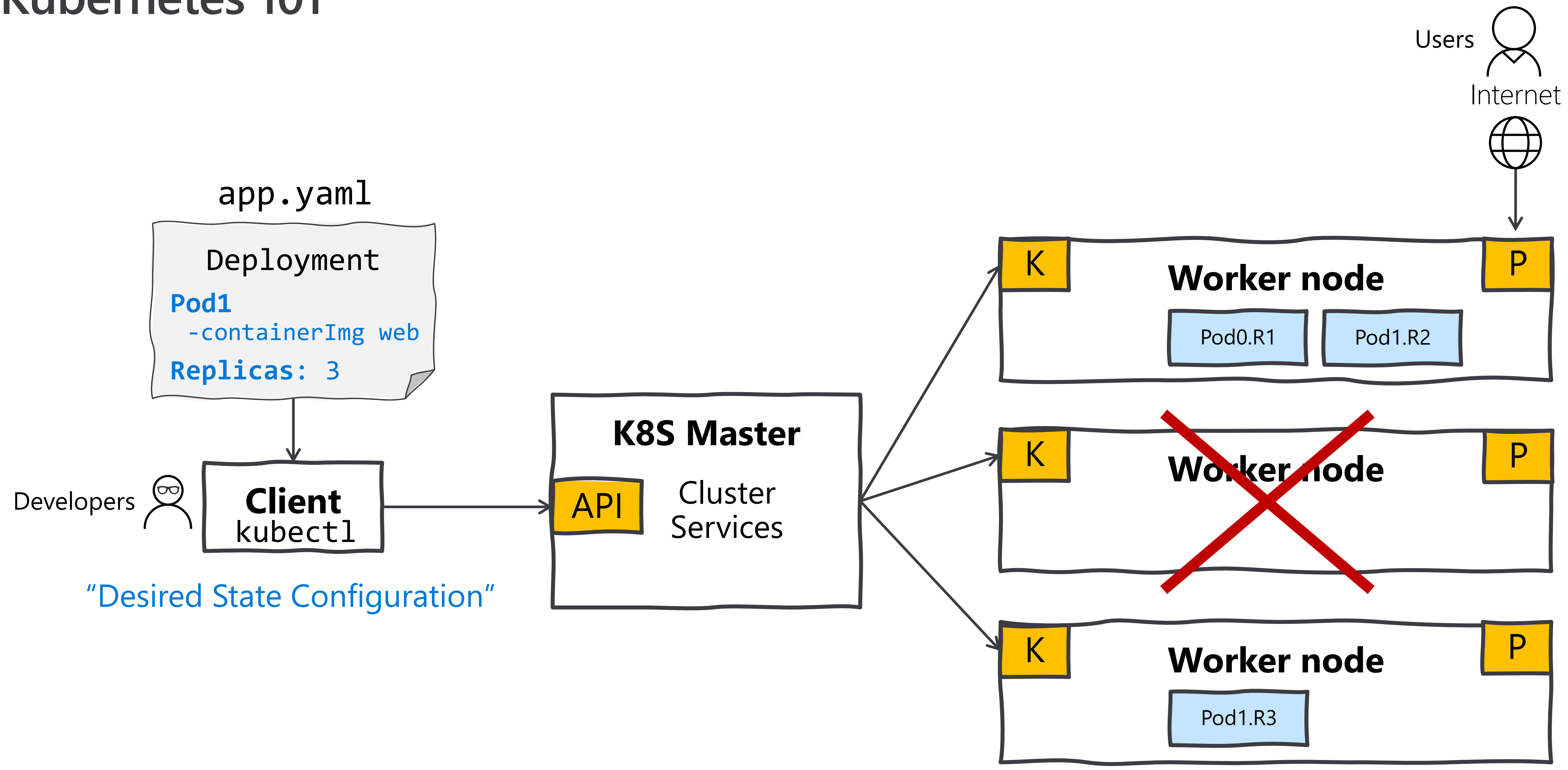
Kubernetes 101



Kubernetes 101



Kubernetes 101



Kubernetes on Azure

Azure Kubernetes Service (AKS)

Simplify deployment, management and operations of Kubernetes

Manage Kubernetes with ease

Accelerate containerized development

Build on an enterprise-grade, secure foundation

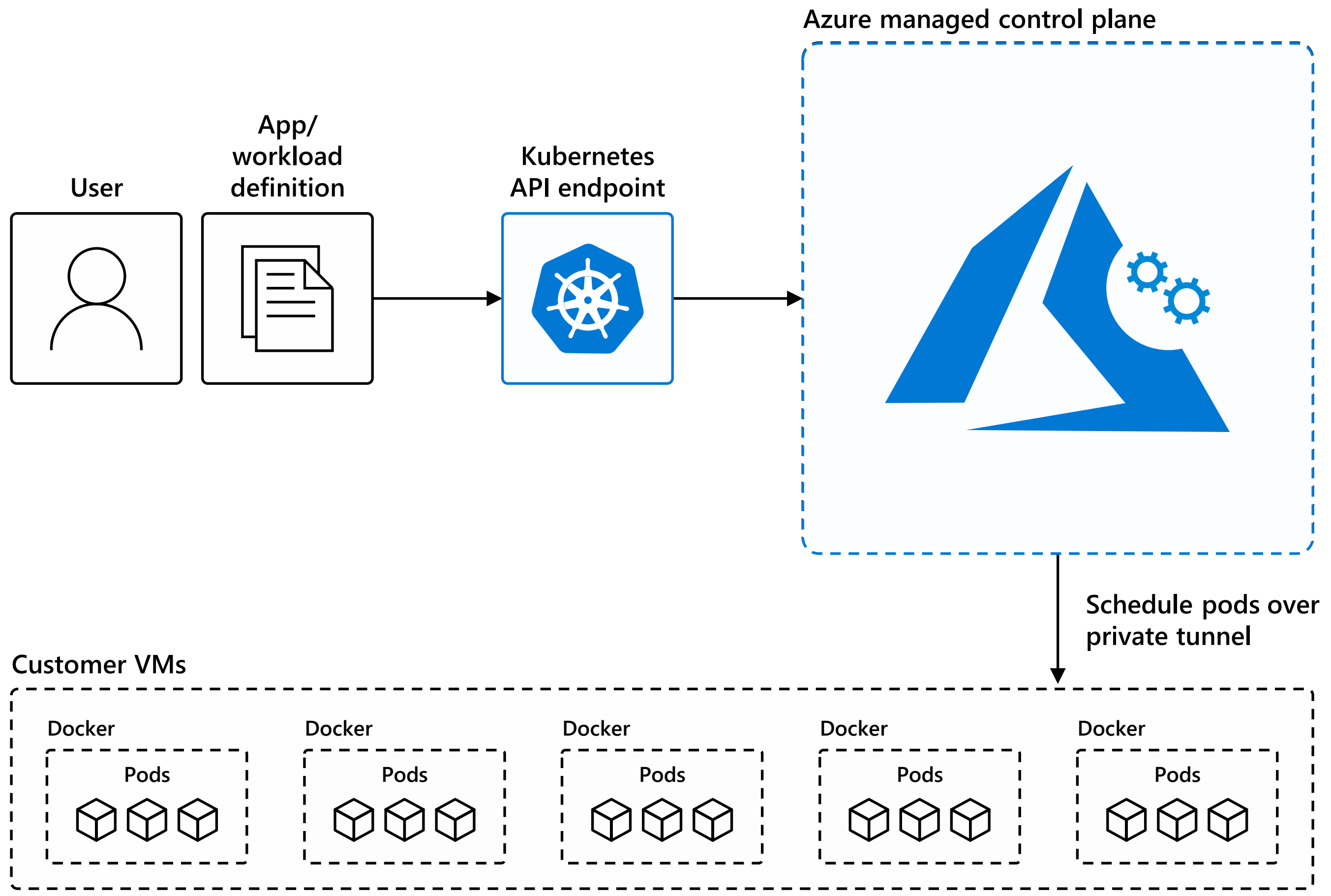
Run anything, anywhere

Increase operational efficiency

Focus on your containers and code, not the plumbing of them

Responsibilities	DIY with Kubernetes	Managed Kubernetes on Azure
Containerization	⬜	◪
Application iteration, debugging	⬜	◪
CI/CD	⬜	◪
Provisioning, upgrades, patches	⬜	🟦
Reliability availability	⬜	🟦
Scaling	⬜	🟦
Monitoring and logging	⬜	🟦

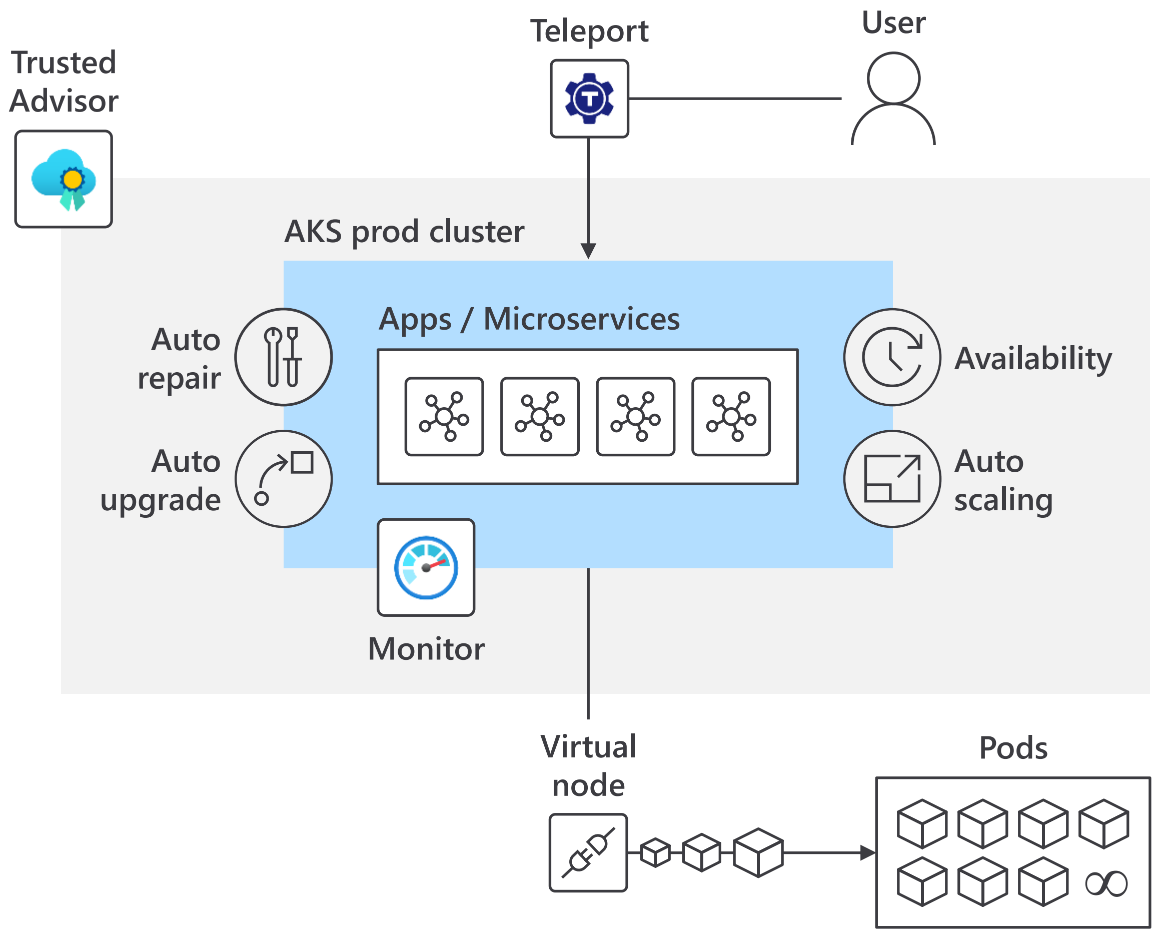
⬜ Customer 🟦 Microsoft



Increased operational efficiency

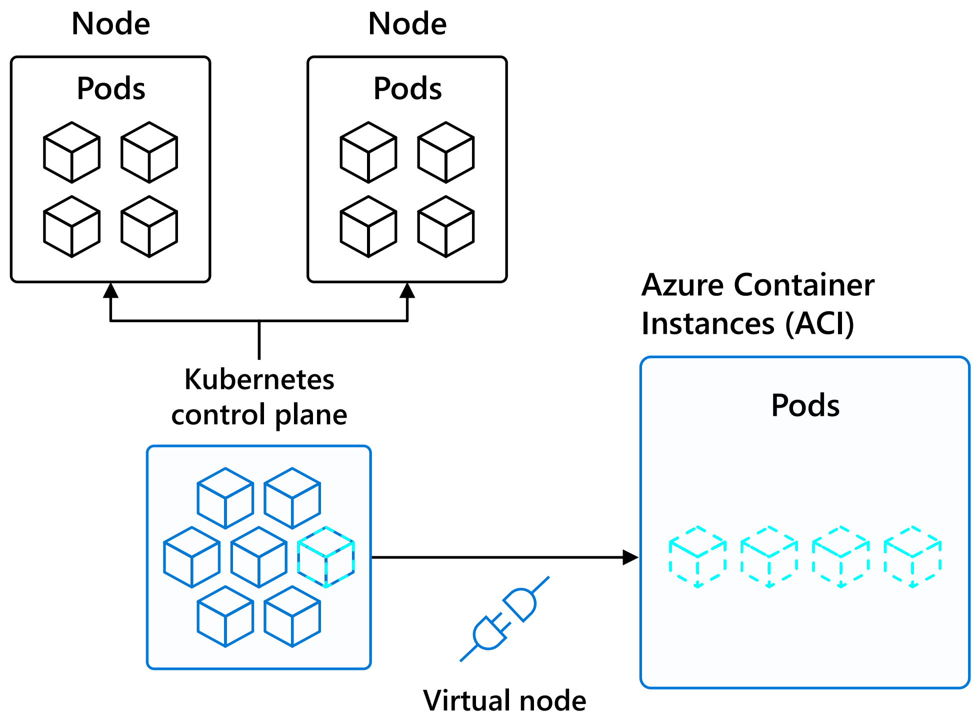
Highly available, reliable service with serverless scaling

- Easily provision fully managed clusters with automatically configured monitoring capabilities based on Prometheus
- Real-time personalized recommendations to optimize your AKS deployments with Azure Advisor integration
- Elastically add compute capacity with serverless Kubernetes in seconds without worrying about managing the infrastructure.
- Higher availability using redundancies across availability zones, protecting applications from datacenter failures



Serverless Kubernetes using AKS virtual nodes

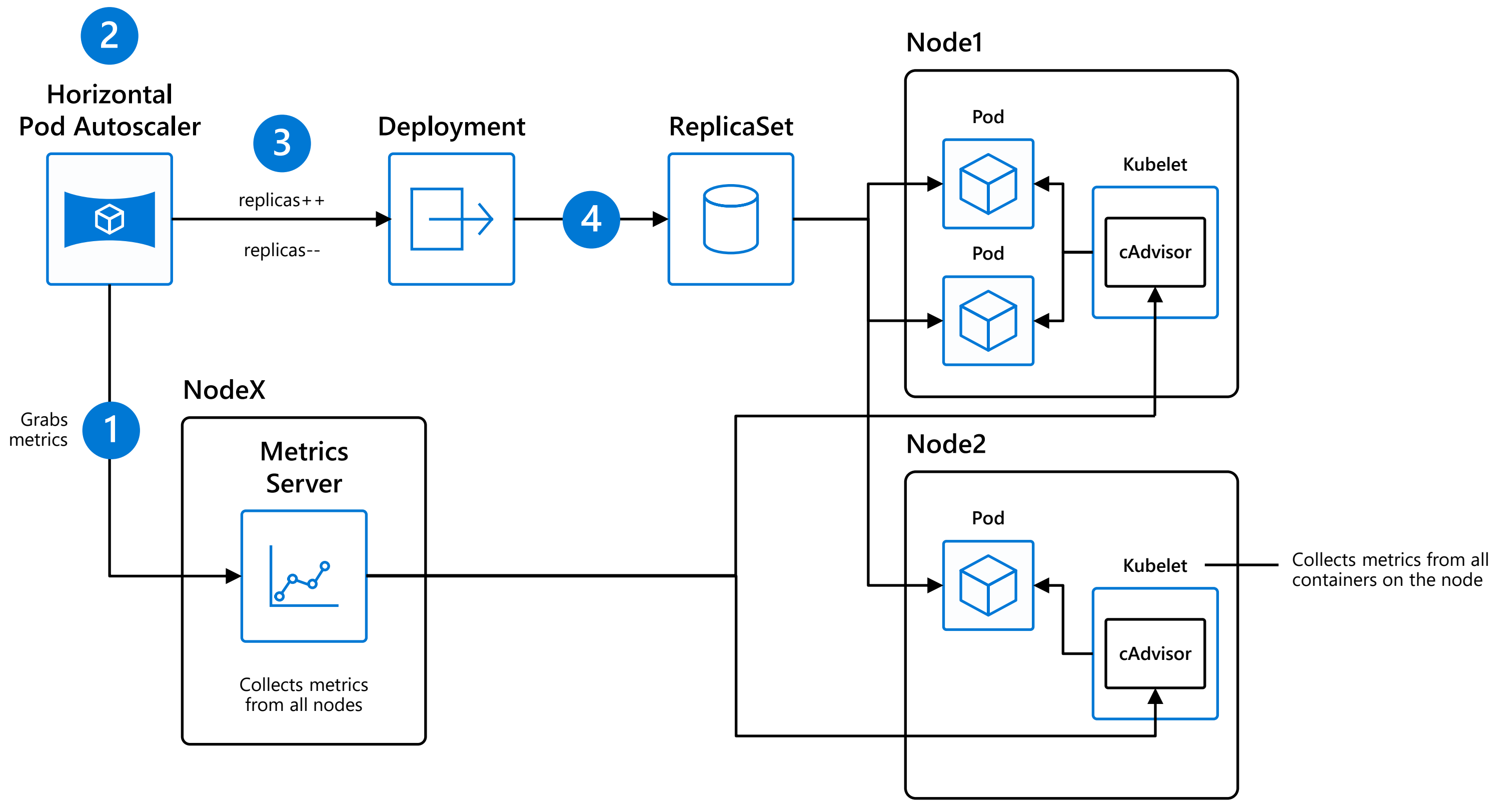
- Elastically provision compute capacity in seconds
- No infrastructure to manage
- Built on open sourced Virtual Kubelet technology, donated to the Cloud Native Computing Foundation (CNCF)



Horizontal Pod Autoscaler

The horizontal pod autoscaler (HPA) uses the Metrics Server in a Kubernetes cluster to monitor the resource demand of pods. If a service needs more resources, the number of pods is automatically increased to meet the demand.

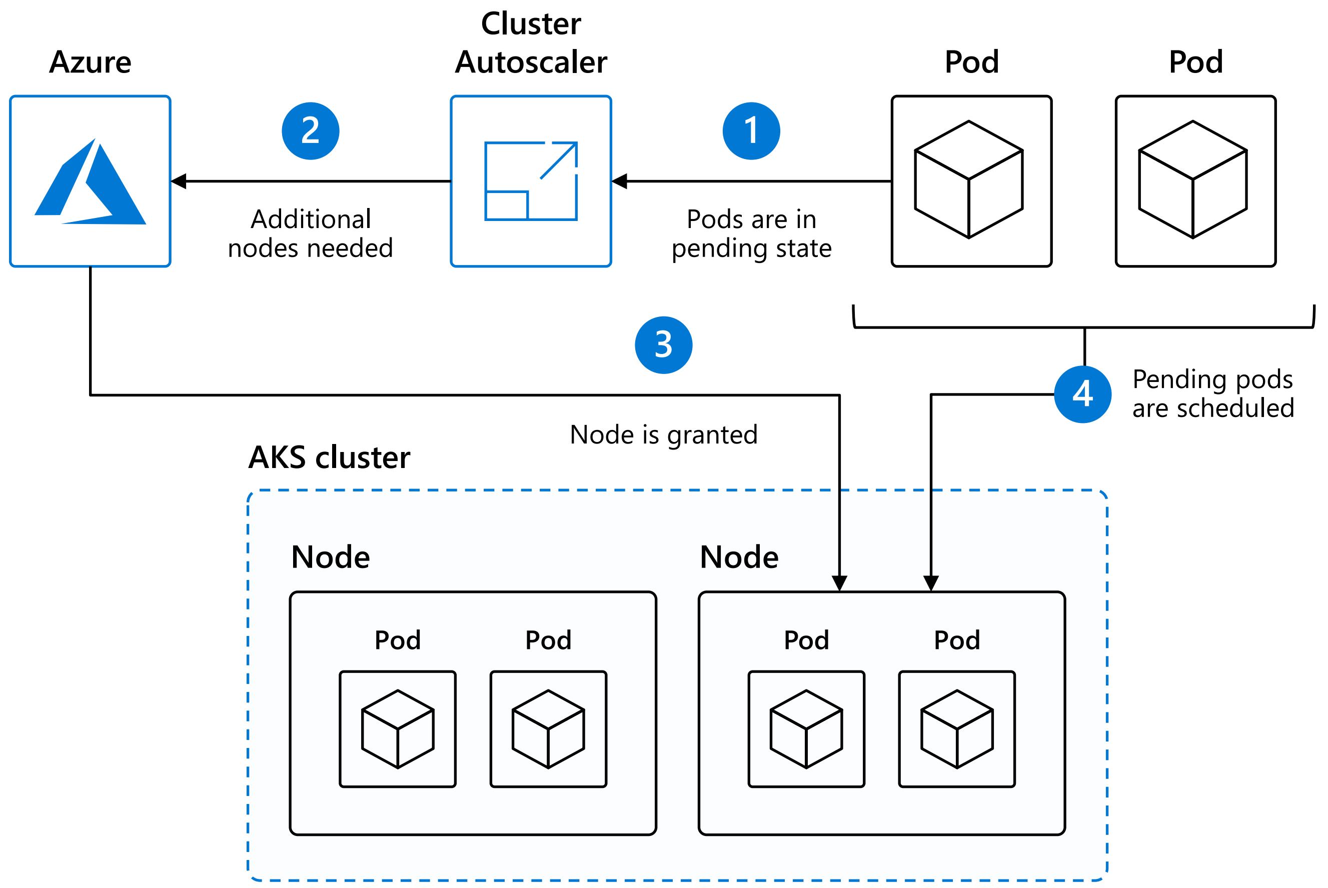
1. HPA obtains resource metrics and compares them to user-specified threshold
2. HPA evaluates whether user specified threshold is met or not
3. HPA increases/decreases the replicas based on the specified threshold
4. The Deployment controller adjusts the deployment based on increase/decrease in replicas



Cluster Autoscaler

The cluster autoscaler watches for pods that can't be scheduled on nodes because of resource constraints. The cluster then automatically increases the number of nodes.

1. HPA obtains resource metrics and compares them to user-specified threshold
2. HPA evaluates whether user specified threshold is met or not
3. HPA increases/decreases the replicas based on the specified threshold
4. The Deployment controller adjusts the deployment based on increase/decrease in replicas



Build on an enterprise-grade, secure platform

Control access through AAD and RBAC

Get runtime vulnerability scanning and auditing through Azure Security Center

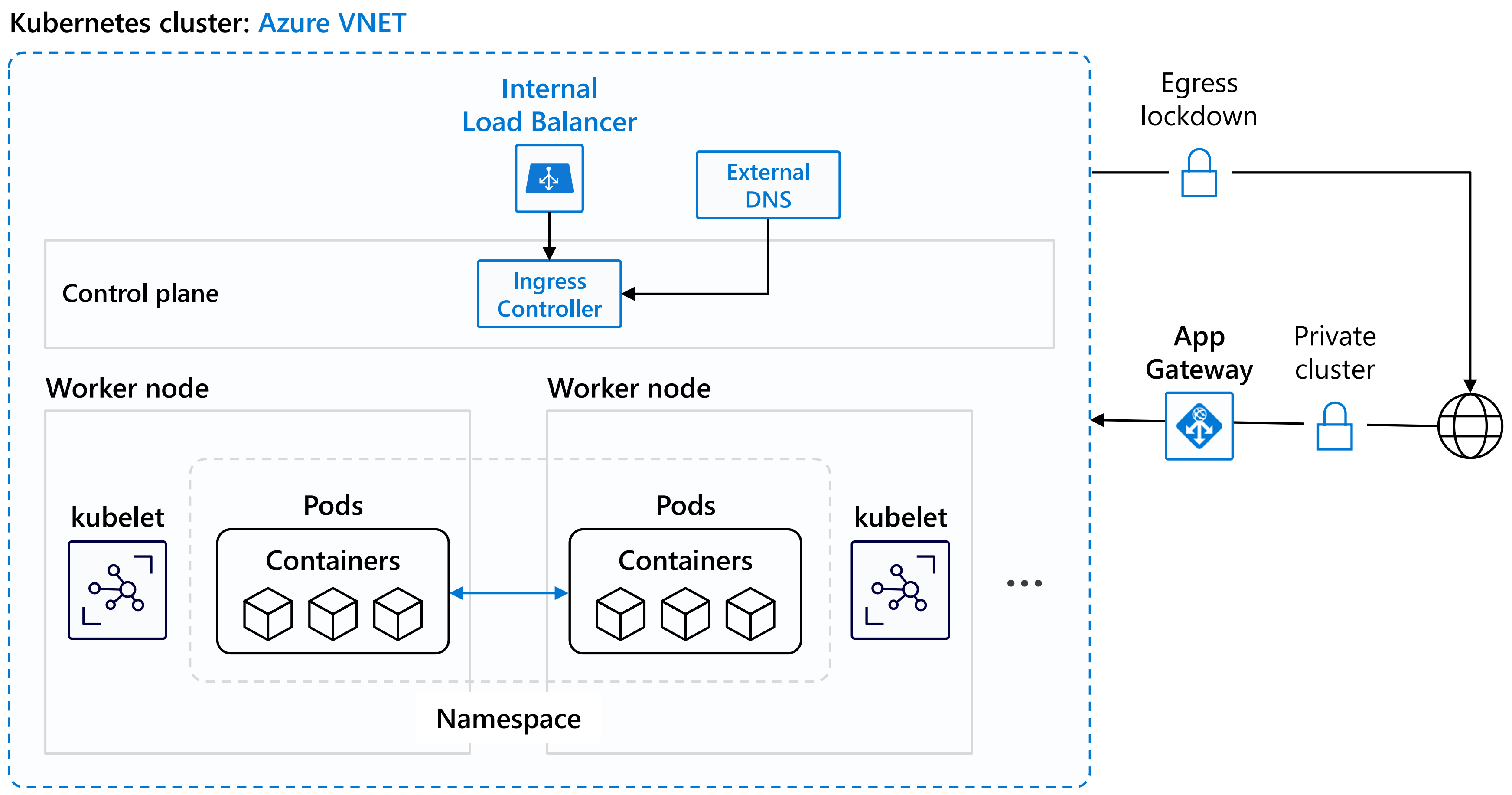
Put guardrails in your development process with Azure Policy

Secure network communications with VNET and network policy

Gain automated threat protection and best practice recommendations for Kubernetes clusters

Networking

Secure your Kubernetes workloads with virtual network and policy-driven communication paths between resources



Basic networking

Uses kubenet network plugin and has the following features

Nodes and Pods are placed on different IP subnets

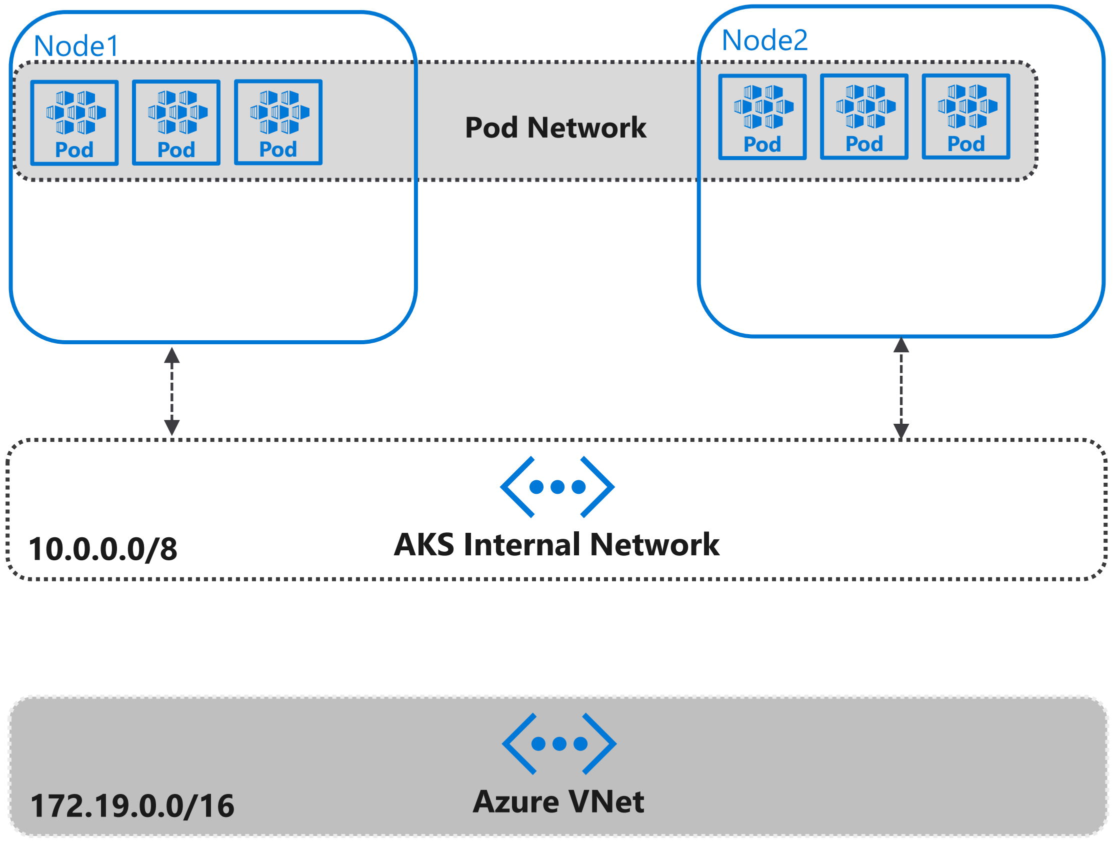
User Defined Routing and IP Forwarding is for connectivity between Pods across Nodes.

Drawbacks

2 different IP CIDRs to manage

Performance impact

Peering or On-Premise connectivity is hard to achieve



Advanced networking

Uses the Azure CNI (Container Networking Interface)

CNI is a vendor-neutral protocol, used by container runtimes to make requests to Networking Providers

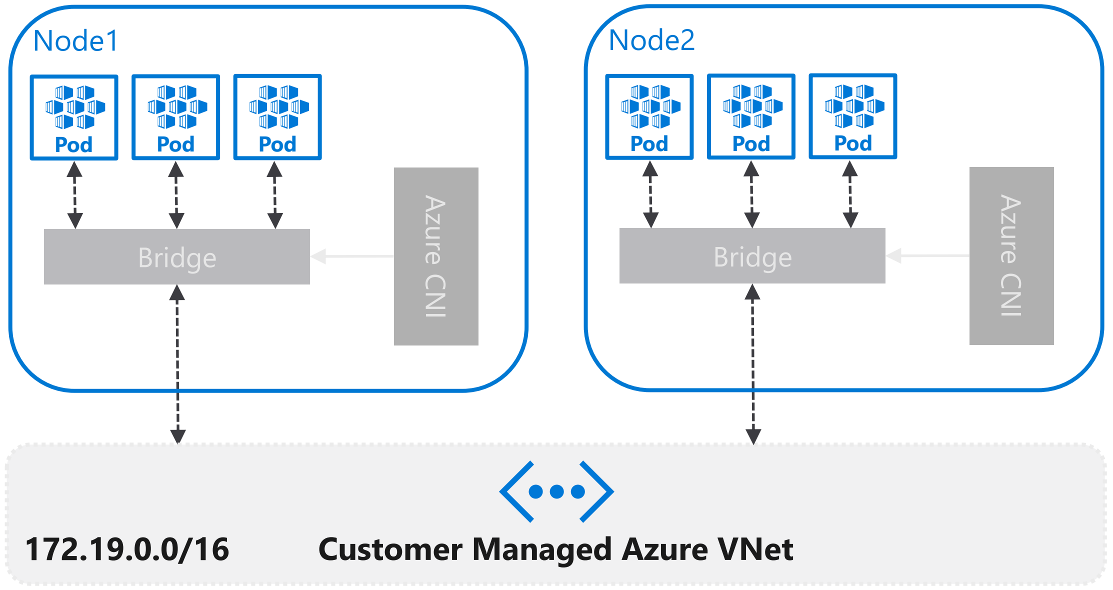
Azure CNI is an implementation which allows you to integrate Kubernetes with your VNET

Advantages

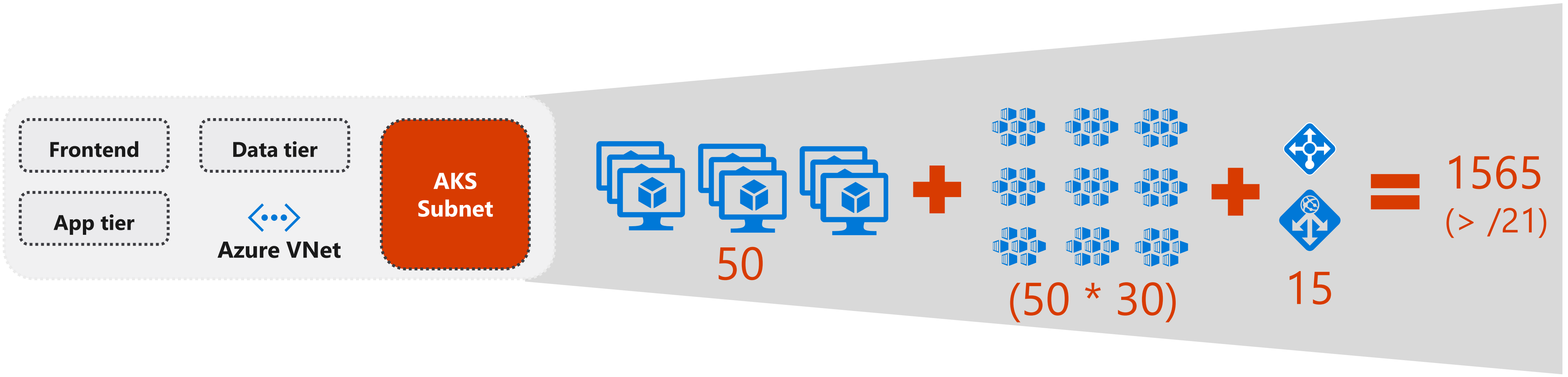
Single IP CIDR to manage

Better Performance

Peering and On-Premise connectivity is out of the box



Advanced networking: Planning IP addressing for your cluster



Additional subnets

Kubernetes service address range

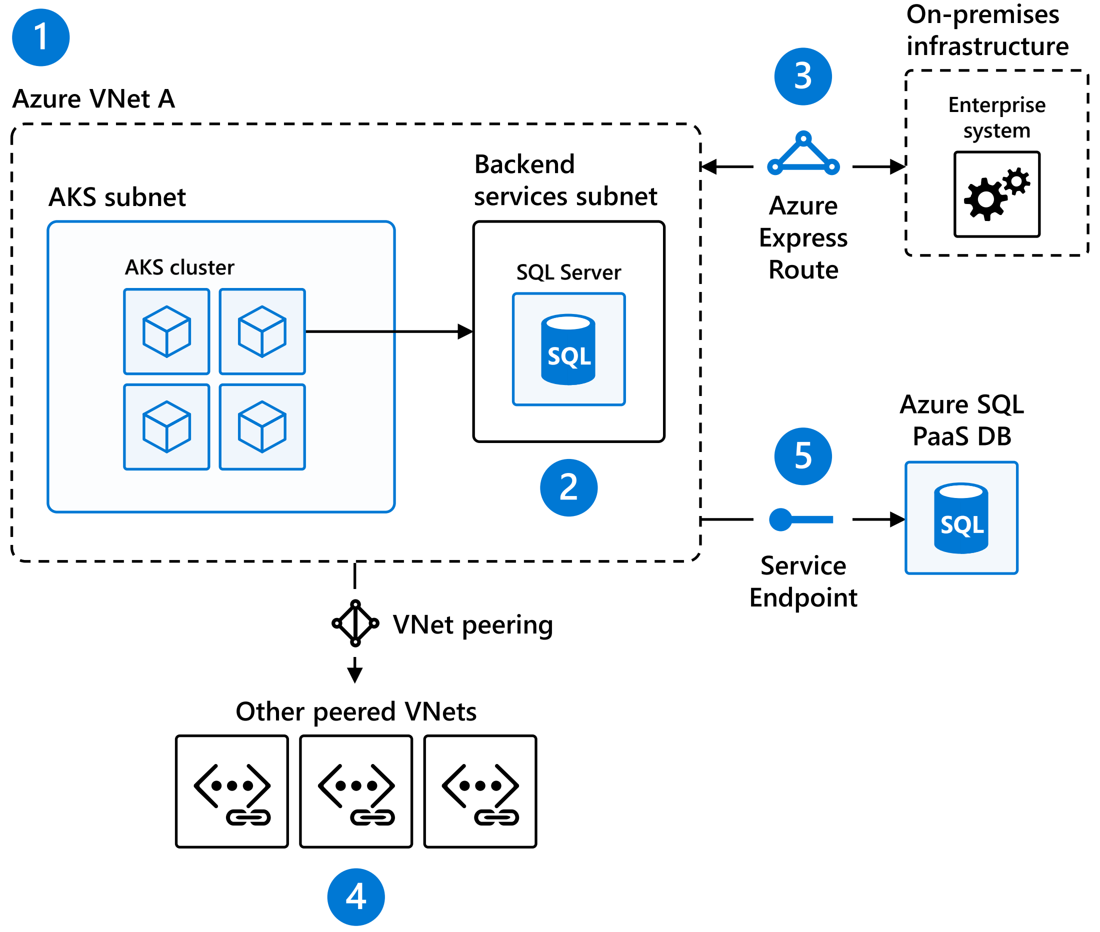
- Non-overlapping with other subnets in your network (does not need to be routable) and not 169.254.0.0/16, 172.30.0.0/16 and 172.31.0.0/16.
- Smaller than /12

Docker bridge address

- Non-overlapping with AKS Subnet

Secure network communications with VNET and CNI

1. Uses Azure subnet for both your containers and cluster VMs
2. Allows for connectivity to existing Azure services in the same VNet
3. Use Express Route to connect to on-premises infrastructure
4. Use VNet peering to connect to other VNets
5. Connect AKS cluster securely and privately to other Azure resources using VNet endpoints

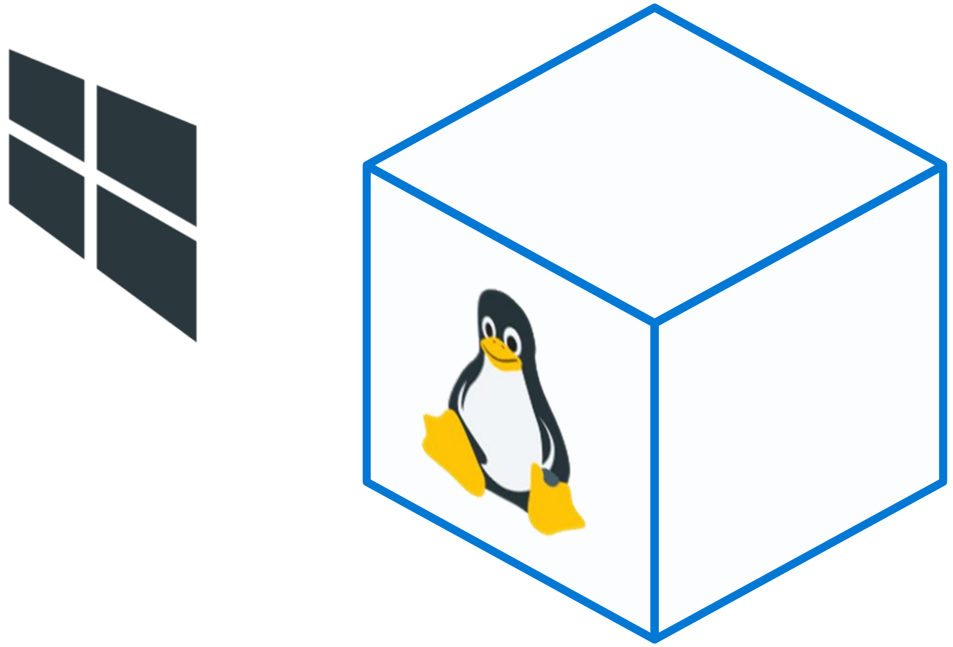


AKS VNet integration works seamlessly with your existing network infrastructure

Azure Kubernetes Service (AKS) support for Windows Server Containers

Now you can get the best of managed Kubernetes for all your workloads whether they’re in Windows, Linux, or both

- Lift and shift Windows applications to run on AKS
- Seamlessly manage Windows and Linux applications through a single unified API
- Mix Windows and Linux applications in the same Kubernetes cluster—with consistent monitoring experience and deployment pipelines



Kubernetes on Azure | Enterprise-grade by design

Development tools

- Visual Studio Code
- GitHub
- Azure Dev Spaces
- Azure Container Registry
- Azure Monitor

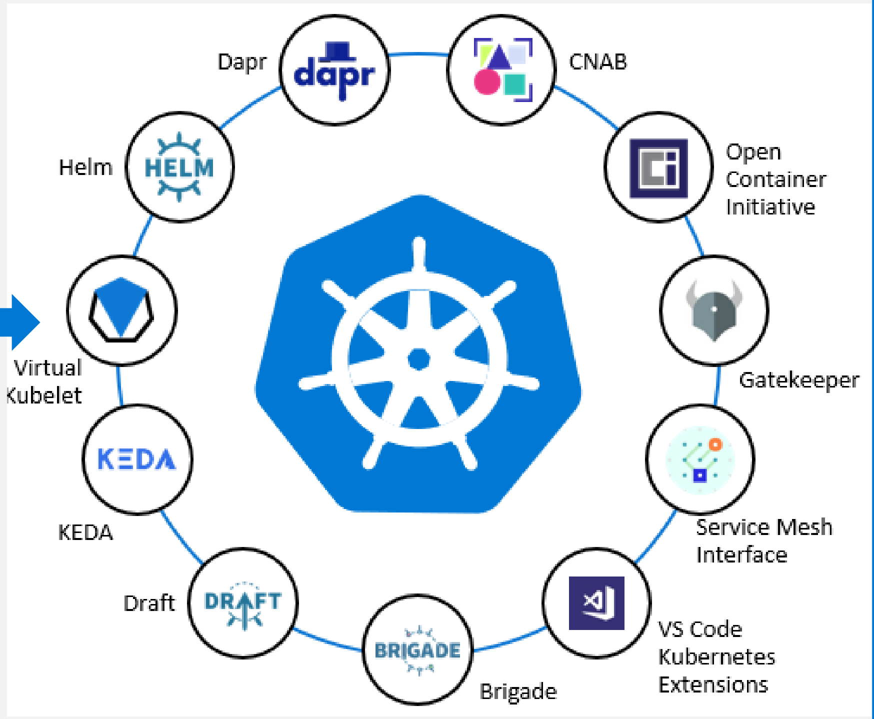
Platform

- Active Directory
- Azure Policy
- Security Center
- Key Vault
- Azure Advisor

- Azure Kubernetes Service
- Azure Container Instances
- Azure Red Hat OpenShift

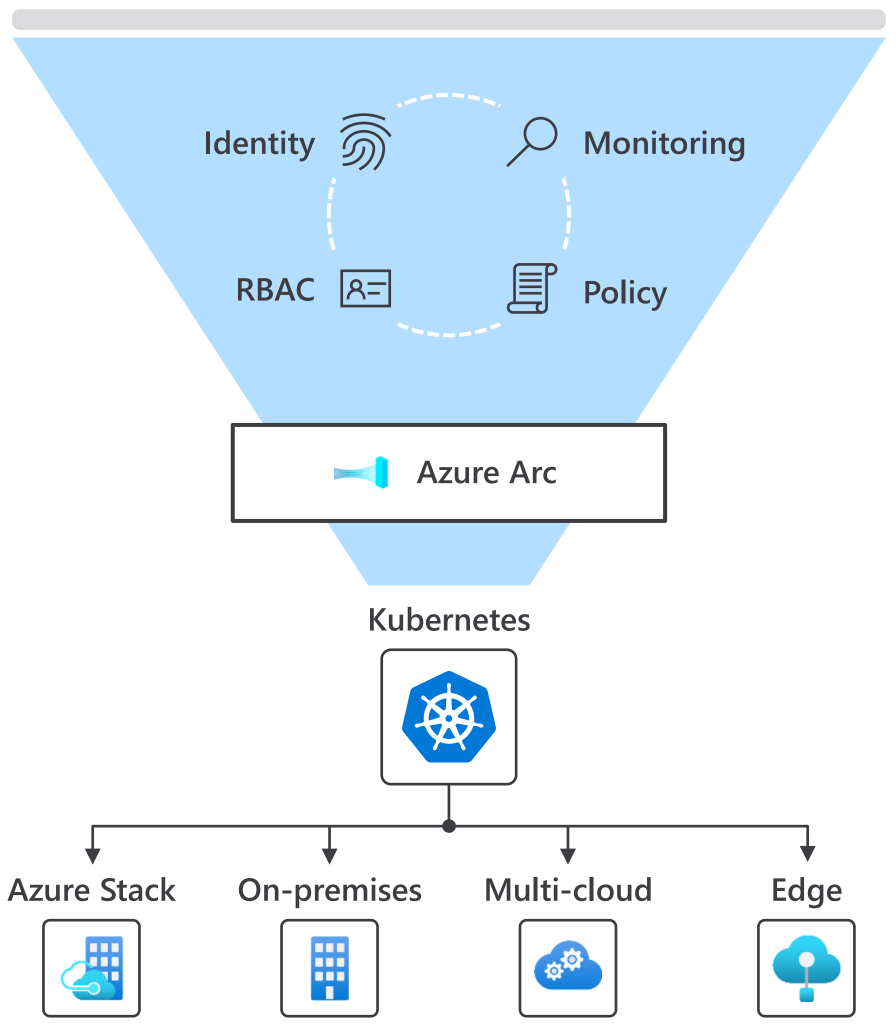
Azure Arc
Management across environments

Community



Unified management

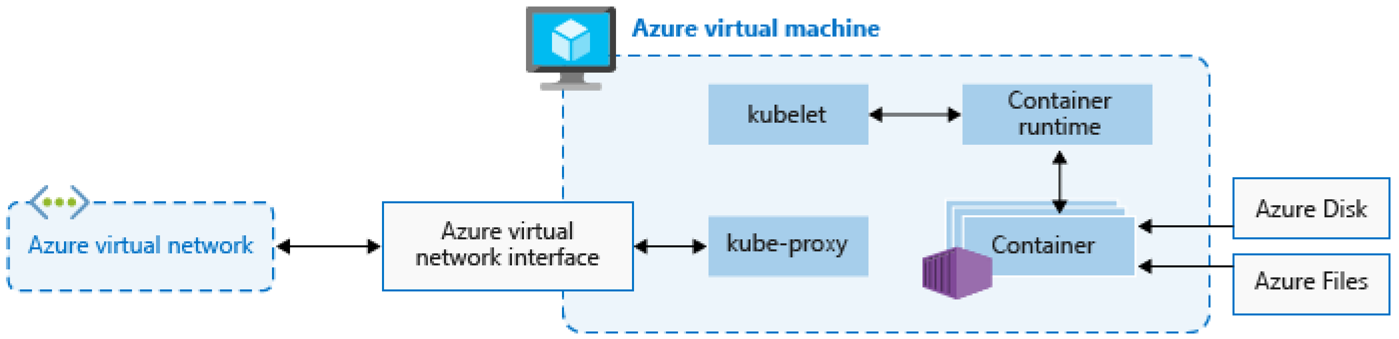
- Central inventory and monitoring of the sprawling assets running anywhere from on-premises to edge
- Consistently apply policies, role-based-access-controls (RBAC) for at-scale governance
- Deploy Kubernetes resources to all clusters using a GitOps-based workflow



Create your first Kubernetes cluster

AKS: Nodes and node pools

- AKS can have multiple node pools (Linux and Windows)
- Each node is an Azure VM
- Azure VM size for your nodes defines how many CPUs, how much memory, and the size and type of storage available (such as high-performance SSD or regular HDD)



Lab 0: Setup

Check proper access and customize your environment

Task	With Azure
Have access to Azure Subscription for AKS deployment	Check if you have access to an Azure Subscription for deploying your own AKS managed cluster. Test access to Azure Cloud Shell.
Create or use a Service Principal	Create a new Service Principal or get one, namely the servicel principal application client ID and the corresponding client secret. This will be used for AKS deployment.
Copy .env file	`mv .env.customize .env`

Customize .env	Edit your .env file and customize the following properties:
	
	`LOCATION=northeurope`
	`APP_NAME=azure-vote`
	`RESOURCE_GROUP=k8s101`
	`CLUSTER_NAME=k8s101`
	`SUBSCRIPTION=<your subscription id>`
	`SERVICE_PRINCIPAL_APP_ID=<your service principal application client id>`
	`SERVICE_PRINCIPAL_SECRET=<your service principal client secret>`

Lab 1: Nodes and node pools

Create a managed Azure Kubernetes Service cluster

Task	With Azure

Create a cluster	`az group create --name $(RESOURCE_GROUP) --location $(LOCATION) --output table`
	`az aks create \`
	`        --resource-group $(RESOURCE_GROUP) \`
	`        --name $(CLUSTER_NAME) \`
	`        --node-count 2 \`
	`        --node-vm-size Standard_D4s_v3 \`
	`        --generate-ssh-keys`
	`        --no-wait`

Validate cluster creation	`sudo az aks install-cli        # not needed if running on Azure cloud shell`
	`az aks show \`
	`        --resource-group $(RESOURCE_GROUP) \`
	`        --name $(CLUSTER_NAME)`
	`az aks get-credentials \`
	`        --resource-group $(RESOURCE_GROUP) \`
	`        --name $(CLUSTER_NAME)`
	`kubectl get nodes`
	`kubectl cluster-info`

Lab 1b: Kubectl alias

Just a suggestion

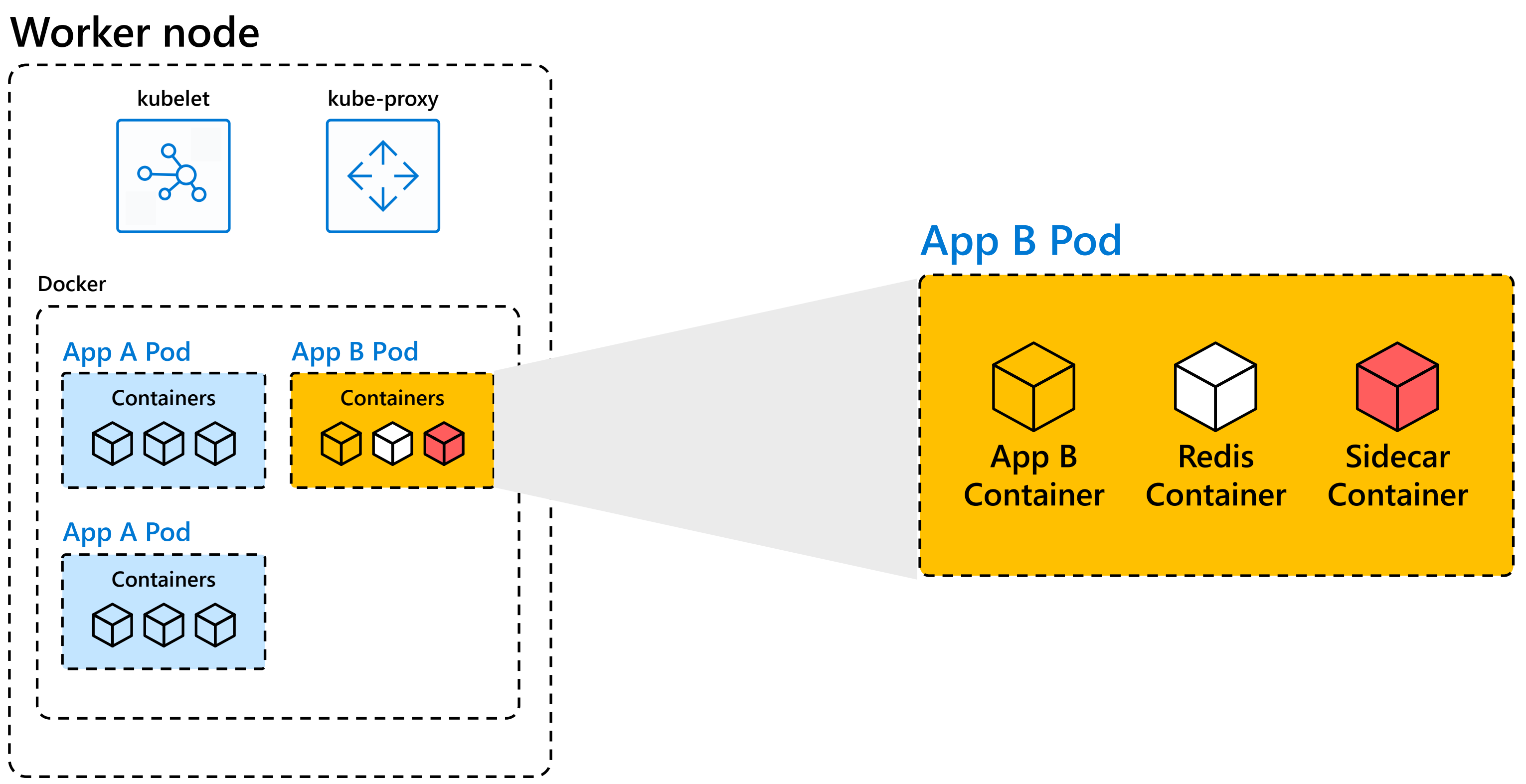
Task	With Azure
Create kubectl alias	see commands below

```
# Edit your .bashrc and add this line:
alias k='kubectl’

# Apply the new settings
source .bashrc

# Test it
k get nodes
```

Pods & Deployments core concepts

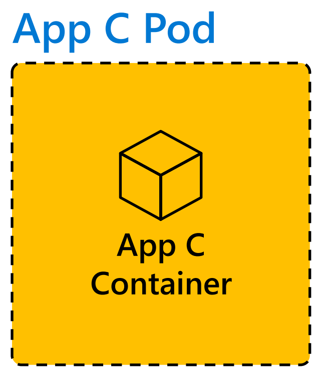
Pods: The building block for your applications in K8S

- Encapsulates an application container (or multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run
- Pods group containers that work together
- Represents a unit of deployment and scaling: a single instance of an application in Kubernetes



Pods - Single container deployment

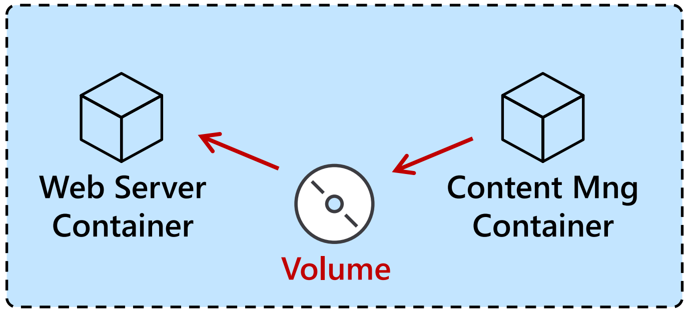
- Pods that run a single container.
- The "one-container-per-Pod" model is the most common Kubernetes use case
- You can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: helloWeb
spec:
  containers:
    image: nginx
    ports:
    - containerPort: 80
```

Pods - Multiple containers deployment

Web App Pod



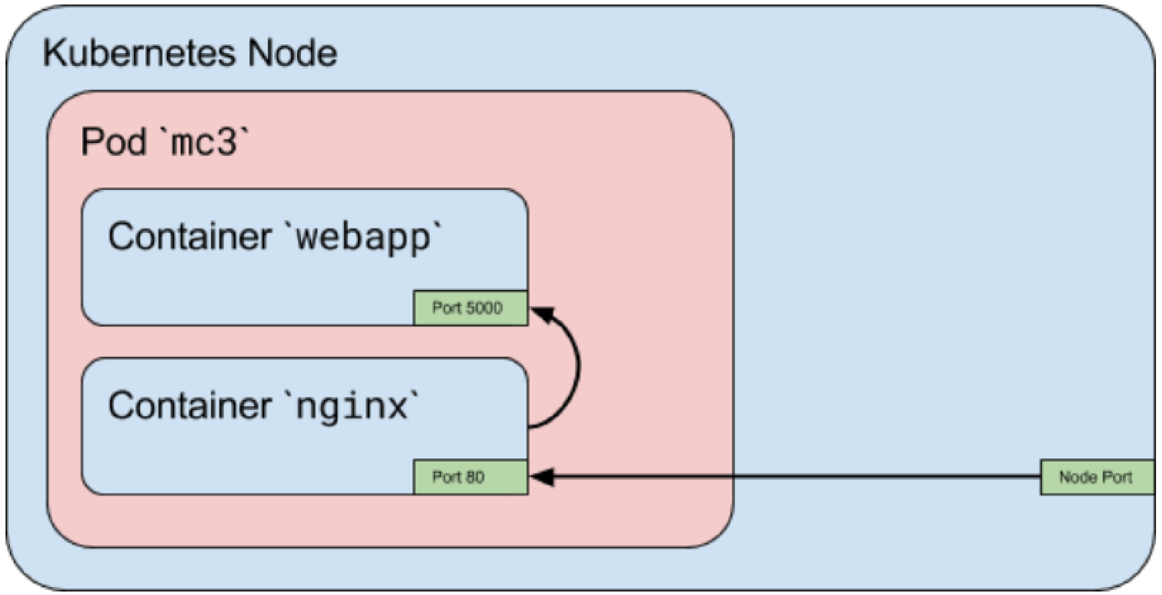
- Pods that run multiple containers that need to work together

- A Pod might encapsulate an application composed of **multiple co-located containers** that are tightly coupled and **need to share resources**

```
apiVersion: v1
kind: Pod
spec:
  volumes:
  - name: html
    emptyDir: {}
  containers:
  - name: webserver
    image: nginx
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: content
    image: debian
    volumeMounts:
    - name: html
      mountPath: /html
```

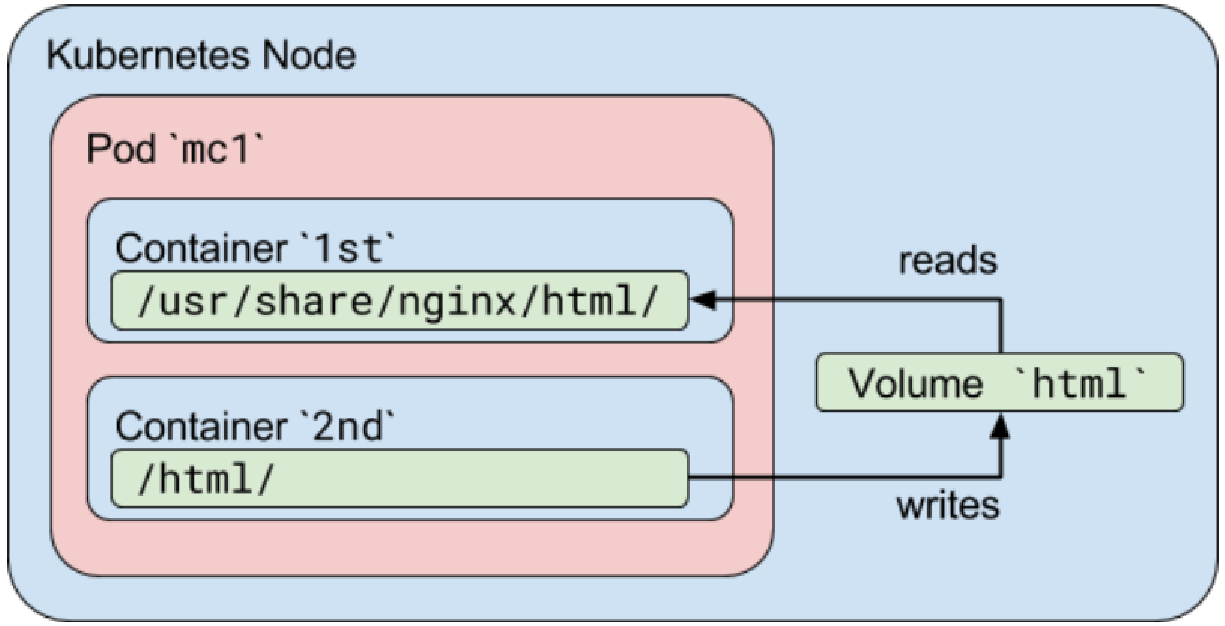
Pods - Network

- Each Pod is assigned a unique IP address
- Every container in a Pod shares the network namespace, including the IP address and network ports
- Containers inside a Pod can communicate with one another using localhost
- When containers in a Pod communicate with entities outside the Pod, they must coordinate how they use the shared network resources (such as ports)



Pods - Storage

- A Pod can specify a set of shared storage volumes
- All containers in the Pod can access the shared volumes, allowing those containers to share data
- Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted



Pods – Resource limits

- When Containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on

- When Containers have their limits specified, contention for resources on a node can be handled in a specified manner

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
```

Deployments for fault tolerance and scale

- A **Deployment** controller provides declarative updates for **Pods** and **ReplicaSets**

- You describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate

- **How it works:**
  - Deployment controller creates a ReplicaSet
  - Deployment controller scales the ReplicaSet to the desired count
  - ReplicaSet creates the desired number of Pods

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world-deployment
  labels:
    app: hello-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-app
    spec:
      containers:
      - name: hello-world
        image: hello
```

Deploy & scale your app

Lab 2: Deployments

Deploy sample application and scale it

Task	With Azure
Deploy sample app	`kubectl create -f azure-vote.yaml`

Validate new app	`# get-deployments`
	`kubectl get deployments`
	
	`# describe-deployments`
	`kubectl describe deployments`
	
	`# get-replicasets`
	`kubectl get rs`
	
	`# get-pods`
	`kubectl get pods --show-labels`

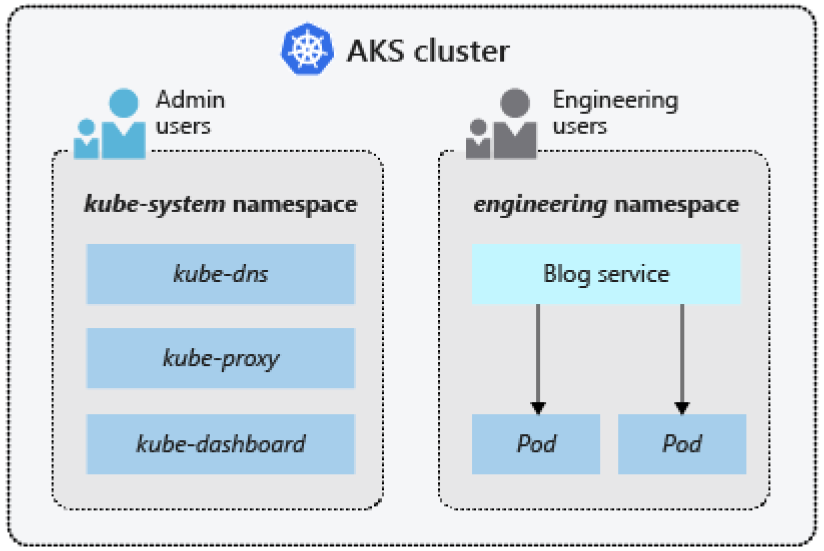
Scale your app	`kubectl scale deployment azure-vote-front --replicas=2`

Organize with Namespaces

Namespaces

- Namespaces provide a scope for names. Names of resources need to be unique within a namespace
- Namespaces are a way to divide cluster resources between multiple users (via resource quota)
- A service DNS entry is of the form <service-name>.<namespace-name>.svc.cluster.local, which means that if a container just uses <service-name>, it will resolve to the service which is local to a namespace

```
$ kubectl --namespace=<namespace-name> get pods

$ kubectl config set-context $(kubectl config current-
context) --namespace=<namespace-name>
```



```
apiVersion: v1
kind: Namespace
metadata:
  name: engineering
```

Lab 3: Namespaces

Create new namespace

Task	With Azure
Check existing namespaces	`kubectl get namespaces`
Create namespace	`kubectl create namespace hellons`

Check new namespace	`kubectl get namespaces`
	
	`# get pods in the new namespace`
	`kubectl --namespace=hellons get pods`

Expose your app using Services

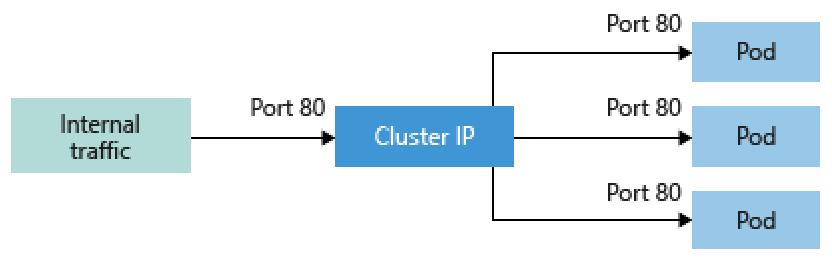
Services to expose and load balance applications

- Services logically group a set of pods together and provide network connectivity
- Allow to expose endpoints (public or private)
- Load balance requests between app replicas
- Several Service types are available

```
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: azure-vote-front
```

Services – Cluster IP

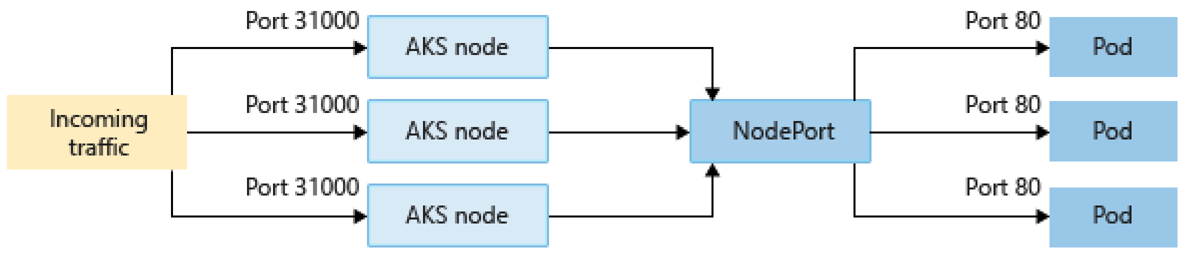
- Creates an internal IP address for use within the AKS cluster
- Good for internal-only applications that support other workloads within the cluster



```yaml
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
  - port: 80
  selector:
    app: azure-vote-back
```

Services – NodePort

- Creates a port mapping on the underlying nodes that allows the application to be accessed directly with the node IP address and port

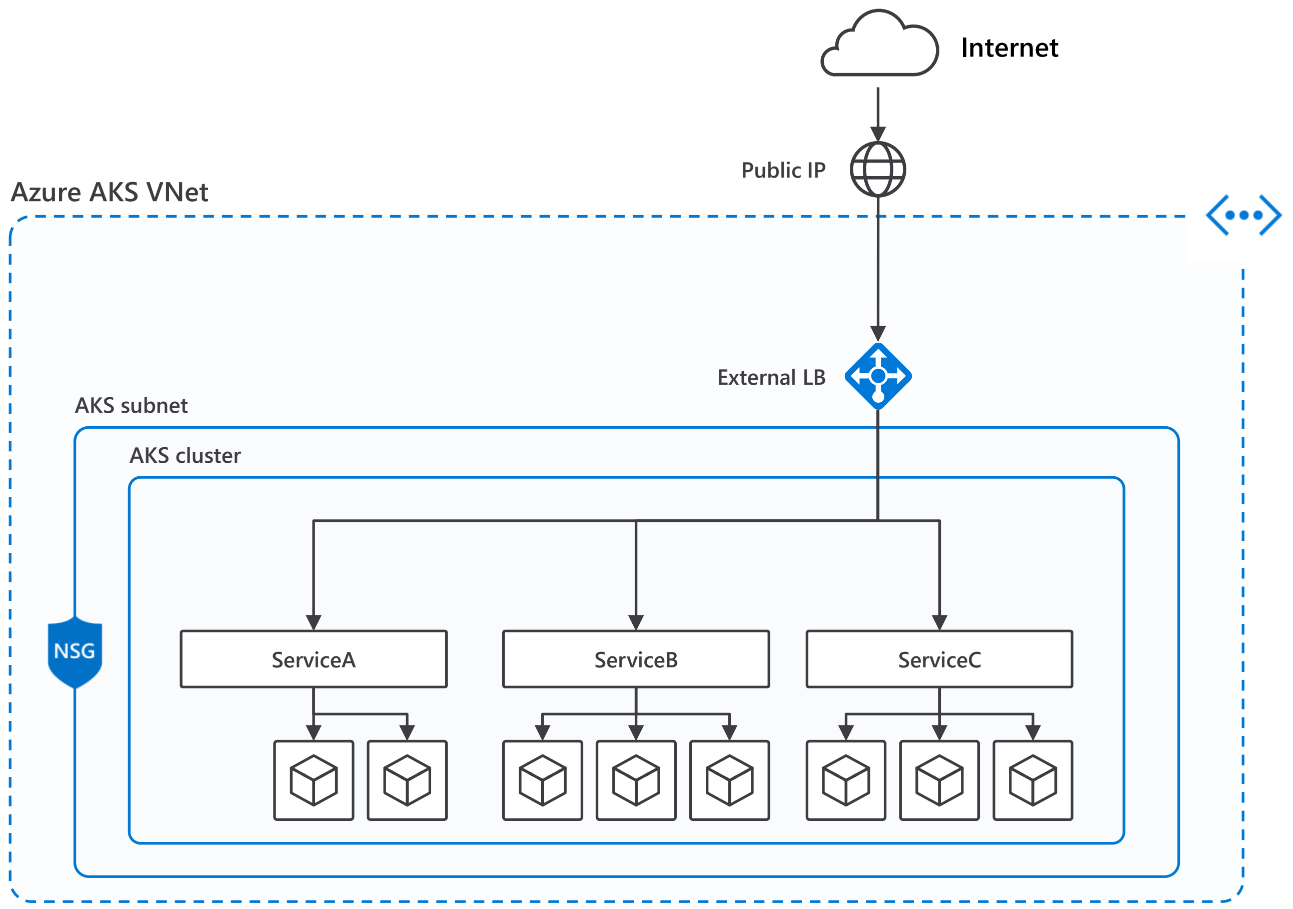


```
kind: Service
metadata:
  name: azure-vote-back
spec:
  type: NodePort
  ports:
  - port: 31000
  selector:
    app: azure-vote-back
```

Services – Load balancer (external)

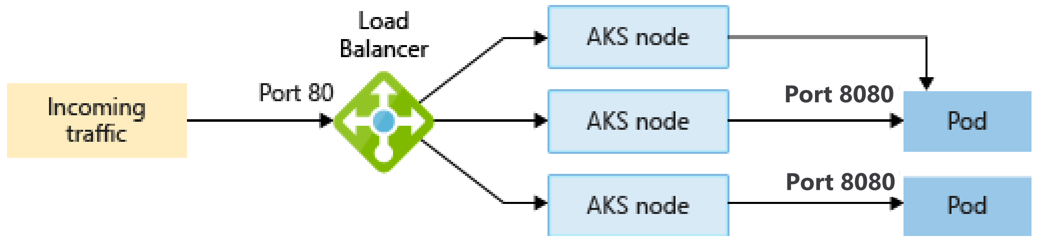
- Creates an Azure load balancer resource, configures a Public external IP address, and connects the requested pods to the load balancer backend pool

- To allow customers traffic to reach the application, load balancing rules are created on the desired ports



Services – Load balancer (external)

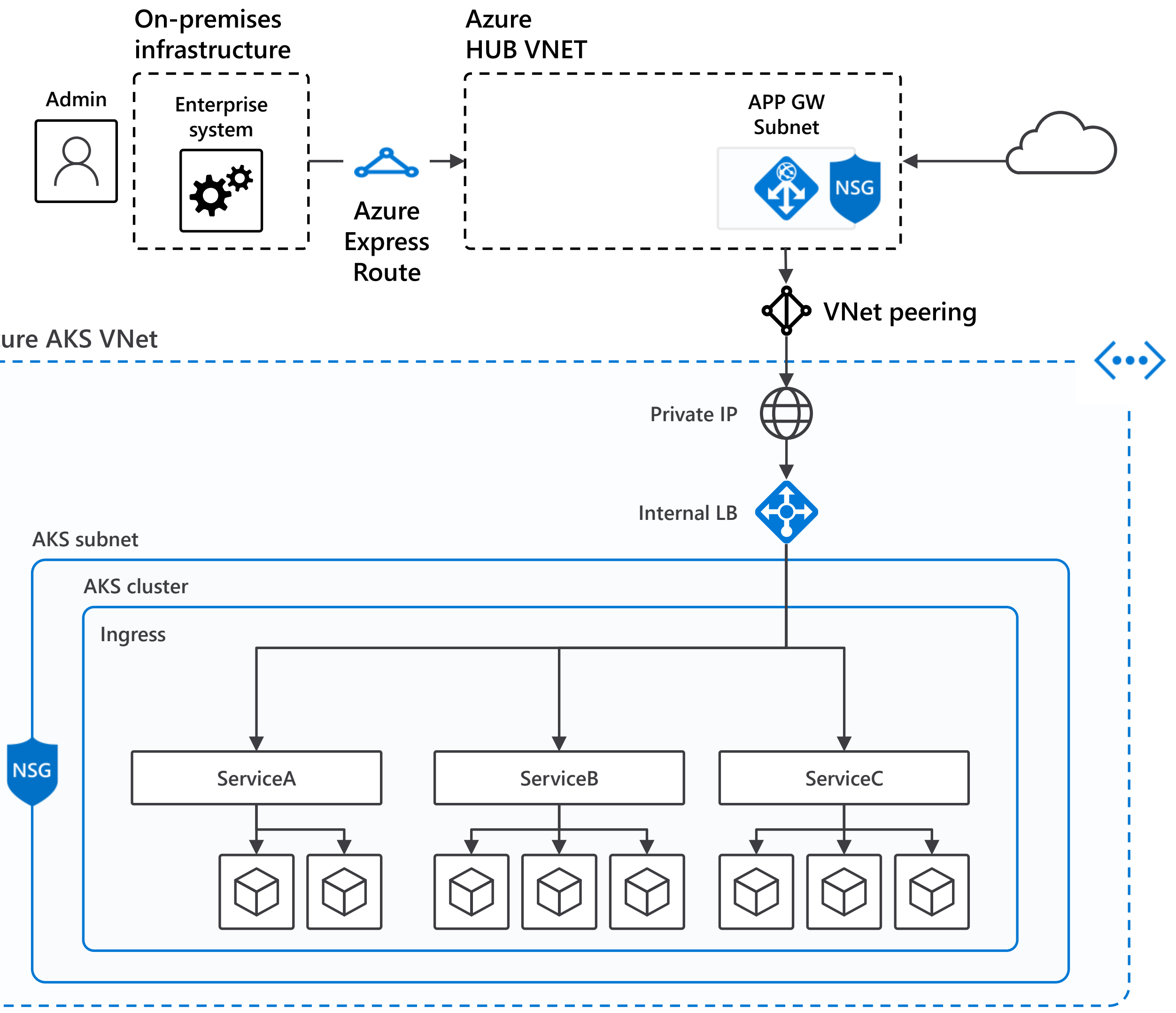
- Creates an Azure load balancer resource, configures a Public external IP address, and connects the requested pods to the load balancer backend pool
- To allow customers traffic to reach the application, load balancing rules are created on the desired ports



```yaml
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: azure-vote-front
```

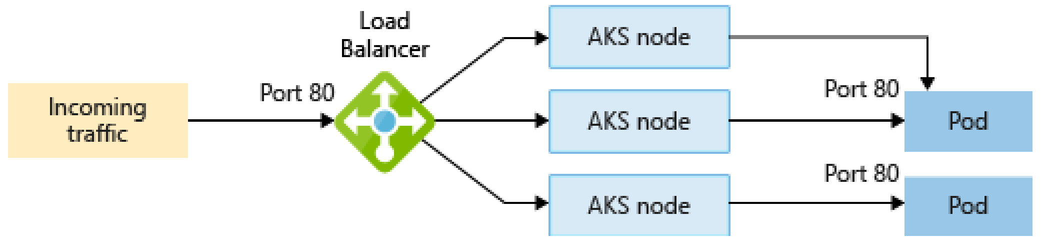
Services – Load balancer (internal)

- Creates an Azure load balancer resource, configures a VNET external IP address, and connects the requested pods to the load balancer backend pool
- To allow customers traffic to reach the application, load balancing rules are created on the desired ports



Services – Load balancer (internal)

- Creates an Azure load balancer resource, configures a VNET external IP address, and connects the requested pods to the load balancer backend pool
- To allow customers traffic to reach the application, load balancing rules are created on the desired ports



```yaml
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front-internal
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Lab 4: Services

Expose different types of services

Task	With Azure

Expose public service	`kubectl create -f azure-vote-service-public.yaml`
	
	`# get public IP`
	`kubectl get service azure-vote-front –watch`

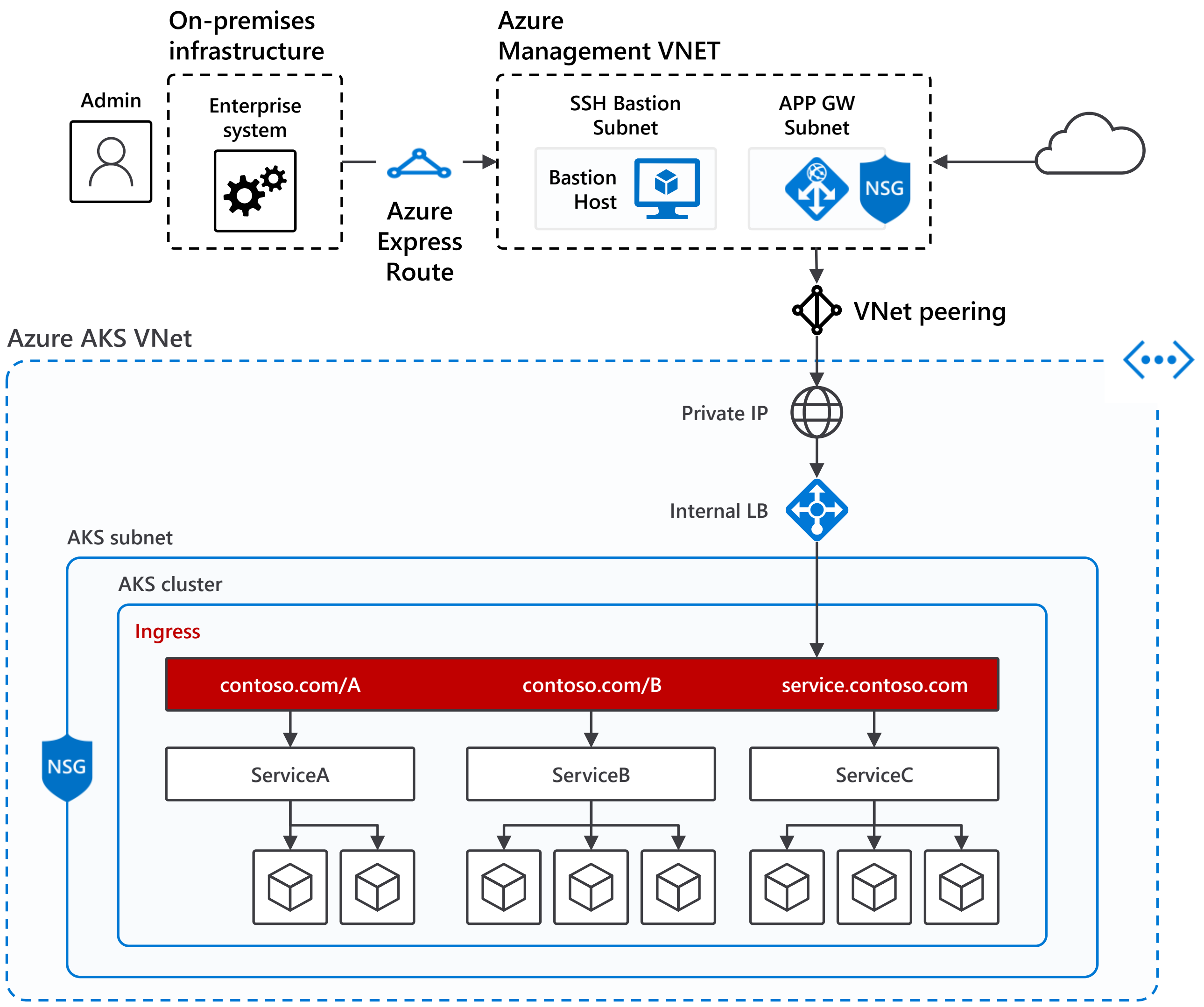
Expose private (Cluster IP) service	`kubectl create -f azure-vote-service-private.yaml`
	
	`# get IP`
	`kubectl get services`
	
	`# run test pod to for curl`
	`kubectl run -i --tty --rm debug --image=radial/busyboxplus --restart=Never -- sh`

Expose NodePort service	`kubectl create -f nodeport-service.yaml`
	
	`# get IP`
	`kubectl describe service azure-vote-front-node`

Intelligent traffic distribution with Ingress Controllers

Ingress controllers

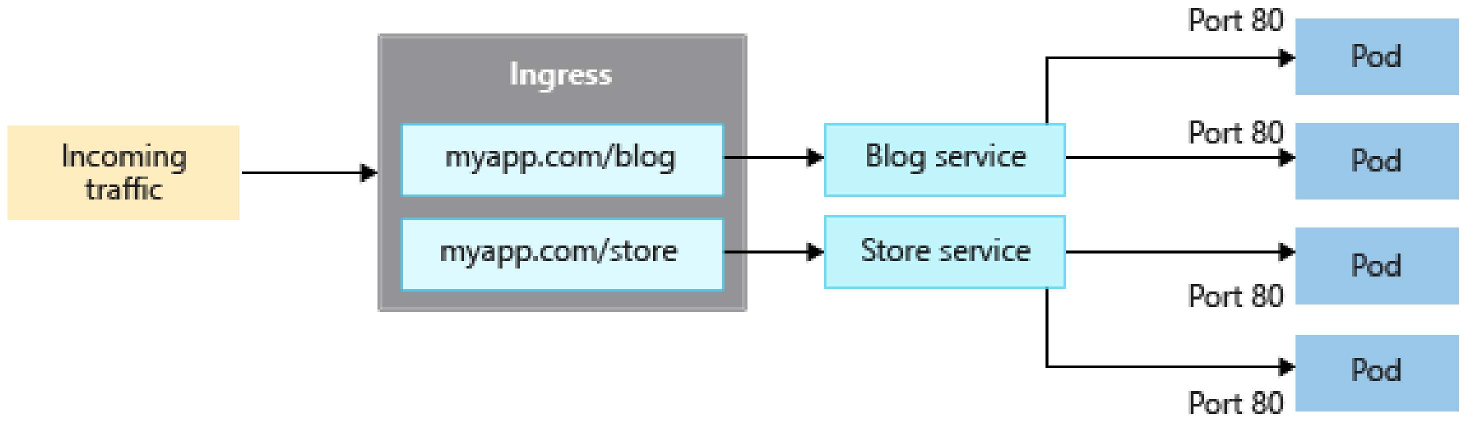
- Ingress is an object that allows access to your Kubernetes services from outside the Kubernetes cluster. You configure access by creating a collection of rules that define which inbound connections reach which services.
- Ingress controllers work at layer 7 and can use more intelligent rules to distribute application traffic (e.g., route HTTP traffic to different applications based on the inbound URL).
- Typically, ingress controllers provide load balancing, SSL termination and name-based virtual hosting



Ingress controllers

- A LoadBalancer type Service rely on an underlying Azure load balancer. The load balancer is configured to distribute traffic to the pods in your Service on a given port. The LoadBalancer only works at layer 4 - the Service is unaware of the actual applications and can't make any additional routing considerations.

- Ingress controllers work at layer 7 and can use more intelligent rules to distribute application traffic. A common use of an Ingress controller is to route HTTP traffic to different applications based on the inbound URL.



Lab 5a: Setup Ingress Controller

Create basic ingress controller using nginx

Task	With Azure
Setup nginx as an ingress controller	See commands below
Get ingress controller IP	See commands below

Setup nginx as an ingress controller

```
# Create a namespace for your ingress resources
kubectl create namespace ingress-basic

# Add the official stable repo
helm repo add stable https://kubernetes-charts.storage.googleapis.com/

# Use Helm to deploy an NGINX ingress controller
helm install nginx stable/nginx-ingress \
    --namespace ingress-basic \
    --set controller.replicaCount=2 \
    --set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
    --set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux
```

Get ingress controller IP

```
# check-ingress-controller
kubectl get service -l app=nginx-ingress --namespace ingress-basic
```

Lab 5b: Use Ingress Controllers

Expose APIs using ingress controllers

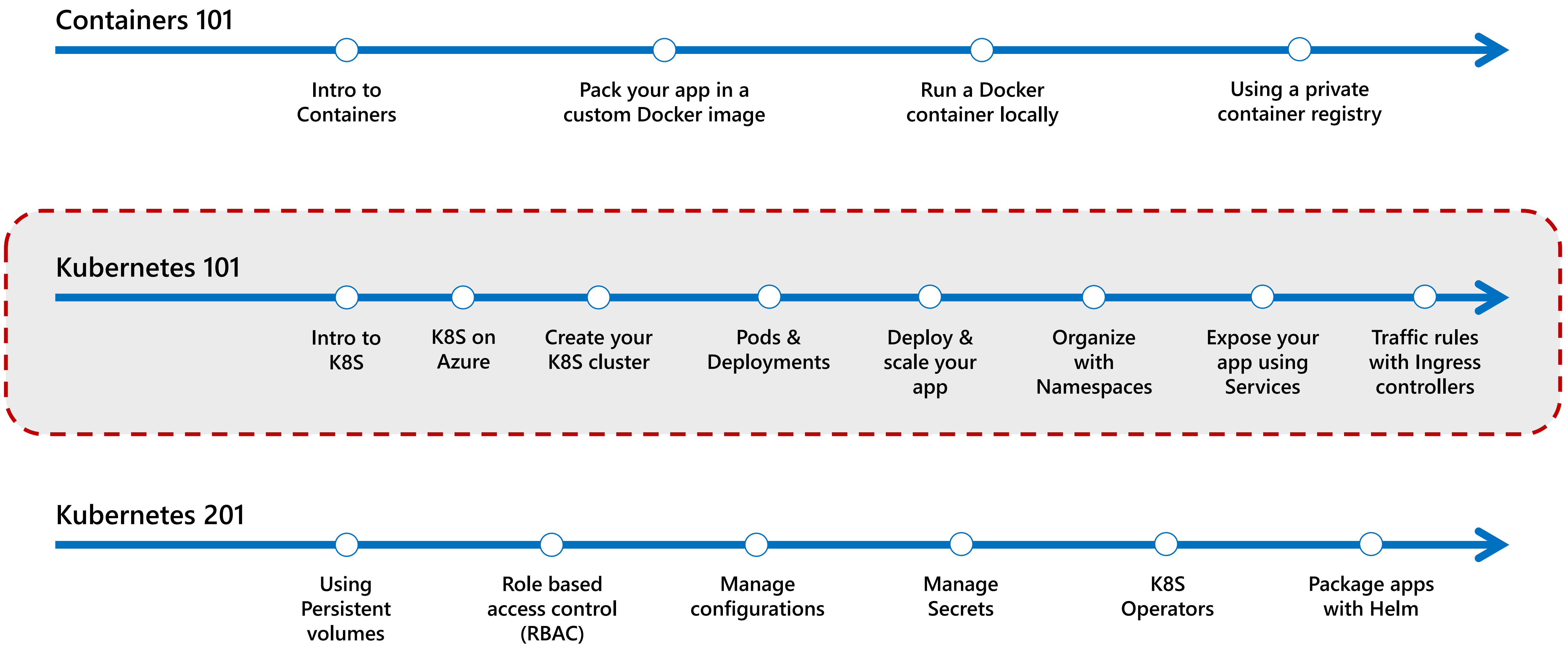
Task	With Azure

Create sample apps	`kubectl create -f ingress/apple.yaml`
	`kubectl create -f ingress/banana.yaml`

Define ingress rules	`kubectl create -f ingress/app-ingress.yaml`
Get ingress controller IP	`kubectl get service -l app=nginx-ingress --namespace ingress-basic`

Test it in your browser	`Open your browser with the above IP and test:`
	
	`http://<your-ingress-external-ip>/apple`
	
	`http://<your-ingress-external-ip>/banana`

Containers & Kubernetes workshops



Thank you