**What Are AKS Virtual Nodes?**

**Virtual Nodes** in AKS provides **scale your workloads rapidly and elastically**

Using **Azure Container Instances (ACI)** as temporary "worker nodes" without actually managing VMs.

Think of them as:

⚡ "Just-in-time, serverless VMs" that run pods instantly when your cluster nodes are full.

**Scenario:**

You run a shopping website hosted on AKS.

* On regular days, you run **3 nodes** in your node pool.
* On a big sale day, traffic **spikes massively**, and your 3 nodes can't handle it.

Normally, you’d autoscale by adding VMs — but that takes 5–10 minutes per VM.

👉 Instead of waiting, **AKS Virtual Nodes** use **ACI** to instantly launch pods **outside your node pool**.

**🔧 How It Works (Technically)**

1. **You enable Virtual Nodes** in your AKS cluster (requires Azure CNI and a subnet)
2. AKS creates a special **“virtual-kubelet” node**
3. When your cluster is full (node autoscaler kicks in), extra pods are **scheduled onto the virtual node**
4. These pods are actually **running in Azure Container Instances (ACI)**
5. **✅ Benefits of Virtual Nodes**

| **Feature** | **Benefit** |
| --- | --- |
| ⚡ Fast | Pods start in **seconds**, not minutes |
| 💵 Cost-efficient | Pay only for the **seconds used** in ACI |
| 🧠 No VM management | No need to manage or patch extra nodes |
| 🔄 Auto-scaling | Integrates with HPA/KEDA easily |
| 💼 Great for burst workloads | Ideal for unpredictable traffic/jobs |

A screenshot of a computer

AI-generated content may be incorrect.

**Example Real Case in Fintech**

A payments company has batch jobs that run at 6 PM daily (100+ pods).  
Instead of scaling up 20 VMs (and paying for idle time), they:

* Keep just 5 base VMs
* Use Virtual Nodes to run the 100 pods on-demand
* Finish jobs in minutes and shut down automatically

**Virtual Kubelet + Azure Container Instances (ACI) = Azure Virtual Nodes in AKS**

**🔍 What Each Component Is**

**🔹 Virtual Kubelet**

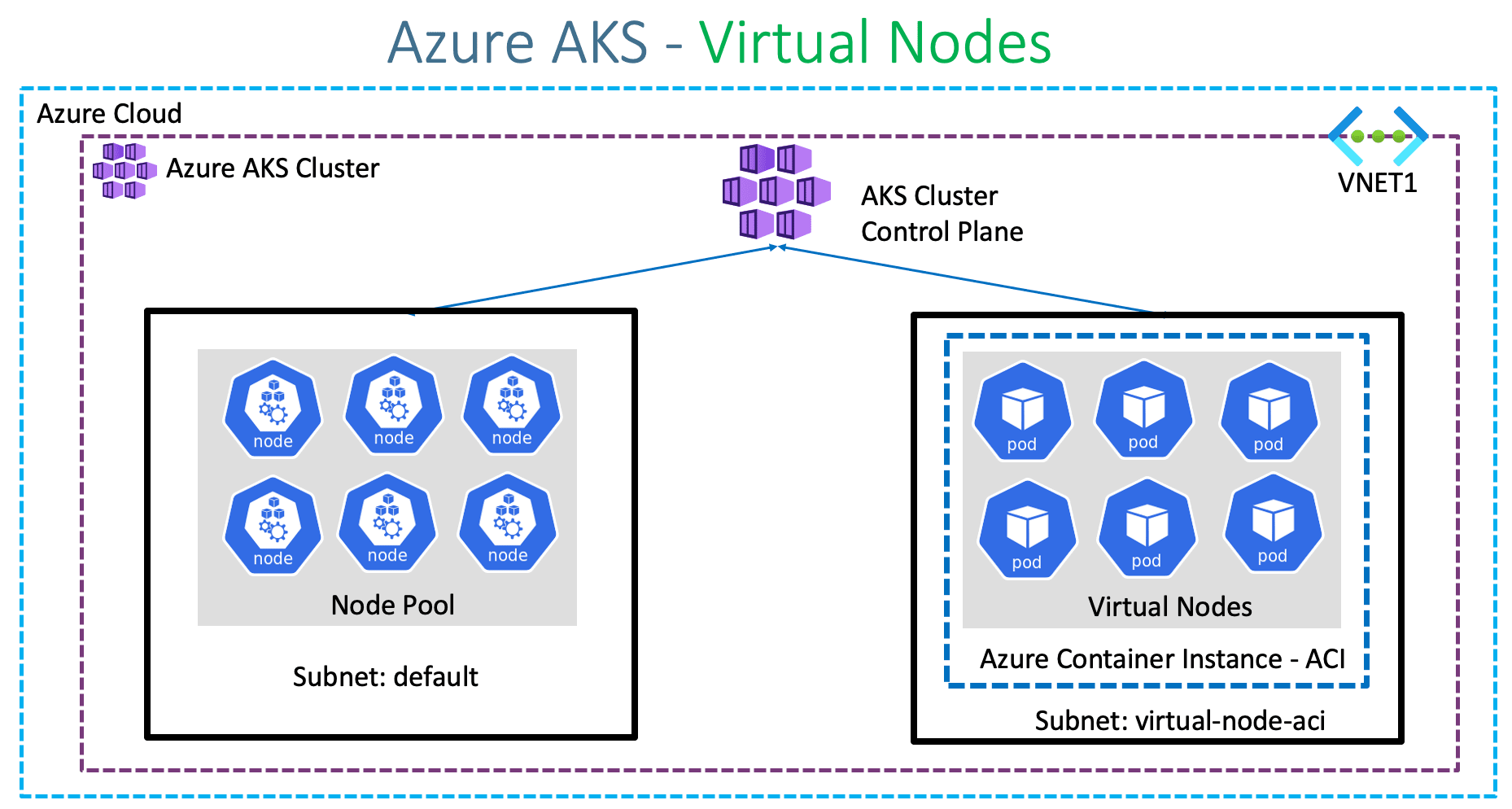
* A **Kubernetes component** that **pretends to be a node**
* It **forwards pod scheduling requests** to an external container runtime (like ACI)
* Think of it as a **bridge between Kubernetes and ACI**

It's not a real VM — just an interface that makes ACI look like a node to Kubernetes

**Azure Container Instances (ACI)**

* A **serverless container runtime** in Azure
* Lets you run containers without managing VMs
* Charges **per second** for container execution time

Think of it like a "one-shot container runner" in the cloud — no VM required



**🔐 For Production (Secure) Setup**

Use **Managed Identity + ACR RBAC** for secure long-term access by:

1. Creating a **User Assigned Managed Identity (UAMI)**
2. Assigning AcrPull role on ACR
3. Enabling **Virtual Node with that UAMI**

You're trying to schedule a **pod on a Virtual Node (ACI)**, but you're **missing a toleration** for the **taint** applied to that node:

kubectl get nodes -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints

virtual-kubelet.io/provider=azure:NoSchedule

You must ensure AKS is attached to ACR. Run:

az aks update \

--resource-group internal-training \

--name aksdemo \

--attach-acr skrisacr

**✅ Step 3: Deploy a Pod to Virtual Node**

You **must**:

* Add a nodeSelector for virtual-node-aci-linux
* Add a **toleration** for the taint virtual-kubelet.io/provider=azure:NoSchedule

az aks update \

--name aksdemo \

--resource-group internal-training \

--attach-acr skrisacr

az acr update -n skrisacr --admin-enabled true

az acr credential show --name skrisacr

{

"username": "skrisacr",

"passwords": [

{

"name": "password",

"value": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"

},

{

"name": "password2",

"value": "YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY"

}

]

}

kubectl create secret docker-registry acr-secret \

--docker-server=skrisacr.azurecr.io \

--docker-username=skrisacr \

--docker-password="1m5WN+4zmvP4F0PvV/F9ERaC7kI/R+ZNZHpSFQbuEn+ACRCOH558" \

[--docker-email=admin@skrisacr.com](mailto:--docker-email=admin@skrisacr.com)

✅ **AKS node → gets credentials** via Managed Identity.

❌ But **Virtual Nodes = Azure Container Instances**, which are **not part of the AKS node pool**. They **do not inherit** that access.

💡 ACI must **authenticate separately** — either via:

* 🔑 imagePullSecrets (short-term)
* 🔐 **Managed Identity + RBAC** (secure production method)

apiVersion: apps/v1

kind: Deployment

metadata:

name: springboot-on-aci

labels:

app: springboot-app

spec:

replicas: 1

selector:

matchLabels:

app: springboot-app

template:

metadata:

labels:

app: springboot-app

spec:

nodeSelector:

kubernetes.io/hostname: virtual-node-aci-linux

tolerations:

- key: "virtual-kubelet.io/provider"

operator: "Equal"

value: "azure"

effect: "NoSchedule"

imagePullSecrets:

- name: acr-secret

containers:

- name: springboot-app

image: skrisacr.azurecr.io/springbootapp:v1

ports:

- containerPort: 8881

---

apiVersion: v1

kind: Service

metadata:

name: springboot-lb-service

spec:

selector:

app: springboot-app

ports:

- protocol: TCP

port: 80

targetPort: 8881

type: LoadBalancer

^Croot@vm1:/home/azureuser/virtualnodes# kubectl get pods -o wide -w

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

springboot-on-aci-77879dd486-5lb82 1/1 Running 0 92s 10.239.0.13 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-d92b6 1/1 Running 0 16m 10.239.0.4 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-dcm9h 1/1 Running 0 91s 10.239.0.11 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-dg4kj 1/1 Running 0 92s 10.239.0.10 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-glr8r 1/1 Running 0 91s 10.239.0.12 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-jz26g 1/1 Running 0 92s 10.239.0.8 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-lpc9k 1/1 Running 0 92s 10.239.0.5 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-ncsm6 1/1 Running 0 92s 10.239.0.7 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-tmf4z 1/1 Running 0 92s 10.239.0.9 virtual-node-aci-linux <none> <none>

springboot-on-aci-77879dd486-wfkmp 1/1 Running 0 92s 10.239.0.6 virtual-node-aci-linux <none> <none>

**⏱️ Why Did It Take ~90 Seconds to Start Pods on Virtual Node (ACI)?**

While **Azure Container Instances (ACI)** is considered fast *compared to provisioning new VMs*, it’s **not fast compared to regular Kubernetes pods** on native AKS nodes.

**Here's what's happening behind the scenes per pod:**

1. 🧱 ACI allocates new compute container group
2. 🐳 Pulls your private image from ACR
3. 🧠 Initializes DNS/network for the container group
4. ✅ Connects it back to the AKS virtual kubelet endpoint

All this **adds cold-start latency** — typically **60–90 seconds** per pod for *non-cached private images*.

**Burst-to-ACI pattern**, where your app runs on regular **AKS node pool**, but when traffic (CPU) increases, **Horizontal Pod Autoscaler (HPA)** scales out into **Virtual Nodes (ACI)**.

**🧠 How It Works**

* ✅ Pods normally run on AKS nodes
* ✅ HPA increases replicas based on CPU (or memory)
* ✅ Once AKS nodes are full, extra pods are scheduled to **Virtual Node**
* ✅ This requires:
  + **Tolerations** (to allow ACI scheduling)
  + **Pod anti-affinity** or **no hard nodeSelector**
  + Optionally, **soft preferred nodeAffinity** to prefer AKS first

**✅ Objective Recap:**

You want:

* 🟢 Pods to **run on AKS node pool by default**
* 🚀 Pods to **fall back to virtual node (ACI)** **only** when AKS node(s) are full

**Use case 2: Run in both Worker node and Virtual node**

Choose **only the non-virtual nodes**, and label them: Apply the taint fo System node pool to avoid app pods into system nodepool.

kubectl taint nodes aks-agentpool-22770300-vmss000000 CriticalAddonsOnly=true:NoSchedule

kubectl taint nodes aks-agentpool-22770300-vmss000001 CriticalAddonsOnly=true:NoSchedule

Now the label the userpool with below label

kubectl label node aks-agentpool-22770300-vmss000000 node-type=aks --overwrite

kubectl label node aks-agentpool-22770300-vmss000001 node-type=aks –overwrite

kubectl get nodes --show-labels | grep node-type

**Update the manifest**

apiVersion: apps/v1

kind: Deployment

metadata:

name: springboot-on-aks

spec:

replicas: 2

selector:

matchLabels:

app: springboot-app

template:

metadata:

labels:

app: springboot-app

spec:

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 100

preference:

matchExpressions:

- key: node-type

operator: In

values:

- aks

tolerations:

- key: "virtual-kubelet.io/provider"

operator: "Equal"

value: "azure"

effect: "NoSchedule"

containers:

- name: springboot-app

image: skrisacr.azurecr.io/springbootapp:v1

ports:

- containerPort: 8881

imagePullSecrets:

- name: acr-secret

This tells Kubernetes:

* ✅ **Prefer nodes with node-type=aks**
* ✅ Only schedule on **ACI (virtual node)** if **AKS nodes are full**
  + (Because it tolerates the NoSchedule taint on the virtual node)

**✅ Step 4: (Optional) Simulate AKS Full Capacity -Option A: Cordon AKS nodes**

To test fallback to virtual node:

* Temporarily **cordon** the AKS nodes:

🔒 "Mark the node as *unschedulable*, so that **no new pods** can be scheduled on it — but **existing pods are not affected**."

This is often used for:

* 🛠️ Maintenance
* 📦 Forcing pods to schedule elsewhere (like ACI virtual node)
* 🔄 Preparing for node drain (eviction)

✅ Stops new pods from being scheduled   
🚫 Does **not** evict current pods

kubectl cordon aks-apppool-30803212-vmss000000

kubectl cordon aks-apppool-30803212-vmss000001

Then increase replicas to 5:

kubectl scale deployment springboot-on-aks --replicas=5

kubectl get pods -o wide

**✅ Step 5: Clean Up (Uncordon)**

kubectl uncordon aks-apppool-30803212-vmss000000

kubectl uncordon aks-apppool-30803212-vmss000001

Scale down to 2 :

kubectl scale deployment springboot-on-aks --replicas=2