2-5-2025

ALO

From COPILOT (entre 14-4-2025 y 28-040-2025)

what are the steps to putting to work cluster autoscaler in azure kubernetes service (aks)?

To enable the **Cluster Autoscaler** in **Azure Kubernetes Service (AKS)**, follow these steps:

**1. Create a Resource Group**

Use the Azure CLI to create a resource group:

sh

az group create --name myResourceGroup --location eastus

**2. Create an AKS Cluster with Autoscaler Enabled**

Run the following command to create an AKS cluster and enable the **Cluster Autoscaler**:

sh

az aks create \

--resource-group myResourceGroup \

--name myAKSCluster \

--node-count 1 \

--vm-set-type VirtualMachineScaleSets \

--load-balancer-sku standard \

--enable-cluster-autoscaler \

--min-count 1 \

--max-count 3 \

--generate-ssh-keys

This sets up a cluster with **autoscaling enabled**, allowing it to scale between **1 and 3 nodes**.

**3. Enable Autoscaler on an Existing Cluster**

If you already have an AKS cluster, you can enable the **Cluster Autoscaler** using:

sh

az aks update \

--resource-group myResourceGroup \

--name myAKSCluster \

--enable-cluster-autoscaler \

--min-count 1 \

--max-count 3

**4. Verify Autoscaler is Working**

Check the status of your nodes:

sh

kubectl get nodes

You can also monitor autoscaling events using:

sh

kubectl describe node <node-name>

**5. Scale Your Application**

Deploy an application and scale it up to **20 pods** to observe autoscaling in action:

sh

kubectl scale --replicas=20 deploy my-app

kubectl get pods

kubectl get nodes

The **Cluster Autoscaler** will automatically **add or remove nodes** based on workload demands. You can find more details in the official Microsoft documentation.

Let me know if you need further clarification! 🚀

[1learn.microsoft.com](https://learn.microsoft.com/en-us/azure/aks/cluster-autoscaler)[2github.com](https://github.com/MicrosoftDocs/azure-docs.zh-tw/tree/95ea496f903c08c2771c22661213113174735397/articles%2Faks%2Fcluster-autoscaler.md)

What are some best practices for configuring the Cluster Autoscaler?

Here are some **best practices** for configuring the **Cluster Autoscaler** in **Azure Kubernetes Service (AKS)**:

**1. Use Separate Node Pools for Availability Zones**

* If using **availability zones**, configure **one node pool per zone** to ensure balanced scaling.
* Set --balance-similar-node-groups to True to maintain an even distribution of nodes across zones.

**2. Optimize Scaling for Large Clusters**

* For clusters with **more than 400 nodes**, use **Azure CNI or Azure CNI Overlay** for better networking performance.
* Avoid excessive scaling operations that could impact cluster stability.

**3. Configure Priority Expanders**

* If running workloads on **both Spot and Fixed node pools**, use **priority expanders** to ensure critical workloads get scheduled first.

**4. Manage Resource Requests Carefully**

* Be cautious when assigning **CPU and memory requests** to pods, as excessive requests can prevent efficient autoscaling.
* Use **Horizontal Pod Autoscaler (HPA)** alongside Cluster Autoscaler for better workload management.

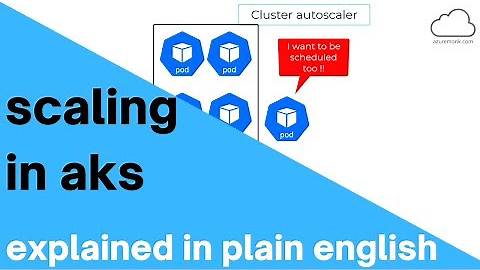
**5. Monitor and Adjust Scaling Parameters**

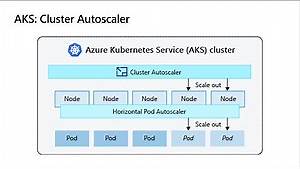
* Regularly review **autoscaler logs** and adjust **min/max node counts** based on workload patterns.
* Ensure that **manual scaling** is disabled when using Cluster Autoscaler to prevent conflicts.

For more details, check out the official Microsoft documentation. Let me know if you need further clarification! 🚀

Editar esta respuesta

My aks cluster autoscaling is not working. I have tested different virtual machine sizes (standard\_B2S and standard\_D2S-V2 for contoso-website) and also I have reduced request and limits in my deployment trying to increase the number of nodes, but I have not got cluster autoscaling with more nodes in response to an increased demand and resources and I desire a configuration for automatic kubernetes cluster scaling in the number of nodes with and increase in nodes with high demand of resources and a decrease of nodes with low demand of resources. At the beginning I would like to see an increase in the number of nodes in aks for a high demand in resources but I don't know what is wrong with my aks configuration?





-------------------------------------------------------------------------------------------------------------------------------------------------------------------

2 vídeos

# Scaling an AKS (Azure Kubernetes Services) cluster explained in plain English

<https://www.youtube.com/watch?v=HGUQW8fCBz8&t=8s>

# Scale applications in Azure Kubernetes Service (AKS)

<https://www.youtube.com/watch?v=5w4W8KNaGSI&t=7s>

-------------------------------------------------------------------------------------------------------------------------------------------------------------------

It sounds like you've done quite a bit of troubleshooting already! Here are some possible reasons why your **AKS Cluster Autoscaler** isn't scaling up nodes as expected:

**1. Check Cluster Autoscaler Logs**

Run the following command to check if the **Cluster Autoscaler** is detecting the need for scaling:

sh

kubectl logs -n kube-system deployment/cluster-autoscaler

Look for messages indicating **scale-up triggers** or errors preventing scaling.

**2. Verify VM Size Compatibility**

* **Standard\_B2S** and **Standard\_D2S-V2** may have limitations in availability or quotas.
* Check if your **Azure subscription** has enough **quota** for additional VMs:

sh

az vm list-usage --location eastus

* If quota limits are reached, request an increase via the **Azure portal**.

**3. Ensure Node Pool Configuration Supports Autoscaling**

Run:

sh

az aks show --resource-group myResourceGroup --name myAKSCluster --query "agentPoolProfiles"

Verify that:

* enableAutoScaling is **true**.
* minCount and maxCount are correctly set.
* The node pool is using **VirtualMachineScaleSets**.

**4. Check Pending Pods**

If pods are stuck in **Pending** state, it may indicate insufficient resources:

sh

kubectl get pods --all-namespaces --field-selector=status.phase=Pending

If pods are pending due to **Insufficient CPU/Memory**, Cluster Autoscaler should trigger a scale-up.

**5. Review Cluster Autoscaler Events**

Run:

sh

kubectl describe node <node-name>

Look for **Cluster Autoscaler events** indicating why scaling isn't happening.

**6. Ensure No Manual Scaling Conflicts**

If you've manually scaled nodes, **Cluster Autoscaler** may not override manual settings. Run:

sh

**az aks update --resource-group myResourceGroup --name myAKSCluster --enable-cluster-autoscaler --min-count 1 --max-count 3**

**This ensures autoscaling is properly configured.**

**7. Check for Known Issues**

There are reports of **Cluster Autoscaler failing to scale** due to VMSS issues. You can also review Microsoft's troubleshooting guide for common autoscaler failures.

**8. Try a Different VM Size**

If **Standard\_B2S** and **Standard\_D2S-V2** aren't scaling, try a different VM size like **Standard\_DS3\_v2** or **Standard\_F4s\_v2**.

**9. Watch a Tutorial**

You might find this video helpful—it explains AKS scaling in detail.

Let me know what you find! 🚀

Editar esta respuesta

what are the steps to putting to work cluster autoscaler in azure kubernetes service (aks)?

Below is a step‐by‐step guide to putting the Cluster Autoscaler into work on an AKS cluster. This setup ensures that as your workloads demand more resources, the cluster will automatically add nodes—and when demand drops, it will remove underutilized nodes.

**1. Prepare Your Environment**

* **Install & Update Azure CLI:** Ensure you have Azure CLI version 2.0.76 or later. You can verify by running:

sh

az --version

* **Log in to Azure:**

sh

az login

* **Create a Resource Group:** Choose a region (e.g., *eastus*) and create a resource group:

sh

az group create --name myResourceGroup --location eastus

**2. Create a New AKS Cluster with Autoscaler Enabled**

When creating your cluster, enable the autoscaler on the node pool by specifying the parameters:

* --enable-cluster-autoscaler tells AKS to manage automatic scaling.
* --min-count and --max-count define the lower and upper limit for node count.

Run:

sh

az aks create \

--resource-group myResourceGroup \

--name myAKSCluster \

--node-count 1 \

--vm-set-type VirtualMachineScaleSets \

--load-balancer-sku standard \

--enable-cluster-autoscaler \

--min-count 1 \

--max-count 3 \

--generate-ssh-keys

This command creates a cluster using VM scale sets (which are required for autoscaling) and sets the node pool to scale between 1 and 3 nodes.

**3. Enable Autoscaler on an Existing Cluster**

If you already have an AKS cluster and you’d like to enable the Cluster Autoscaler, update the existing cluster:

sh

az aks update \

--resource-group myResourceGroup \

--name myAKSCluster \

--enable-cluster-autoscaler \

--min-count 1 \

--max-count 3

This ensures that your agent pool is now configured for automatic scaling.

**4. Verify the Autoscaler Setup**

* **Check Node Pool Configuration:** Confirm that your node pool has autoscaling enabled and see the current parameters:

sh

az aks show --resource-group myResourceGroup --name myAKSCluster --query "agentPoolProfiles"

* **Inspect the Autoscaler Logs:** The autoscaler runs as a pod in the *kube-system* namespace. View its logs for any scaling events or errors:

sh

kubectl logs -n kube-system deployment/cluster-autoscaler

* **List Nodes to See Current Status:**

sh

kubectl get nodes

**5. Test the Autoscaler Under Demand**

To confirm that the autoscaler responds to resource pressure:

* **Deploy a Test Application:** Deploy an application that requires resources.
* **Scale the Application:** Increase the number of replicas so that some pods remain unscheduled because of resource constraints:

sh

kubectl scale --replicas=20 deploy my-app

* **Monitor Scaling:** Check pending pods with:

sh

kubectl get pods --all-namespaces --field-selector=status.phase=Pending

Then, watch how new nodes are added to accommodate the extra pods:

sh

kubectl get nodes -w

The Cluster Autoscaler should detect that pods are waiting for resources, triggering a scale-up of nodes until the pending pods can be scheduled.

**6. Additional Considerations**

* **Resource Requests/Overrides:** Ensure your pod specifications include realistic CPU and memory requests. Overestimated values might prevent scheduled placement even if excess capacity exists.
* **Avoid Conflicts with Manual Scaling:** When autoscaling is enabled, avoid manually changing the node count, as this might conflict with the autoscaler’s decisions.
* **Review Documentation:** For further troubleshooting and advanced configuration (like using node pools in availability zones), consult the official Microsoft documentation.

These steps establish a configuration where your AKS cluster adapts dynamically—scaling out during high demand and scaling in when the workload decreases, ensuring efficient resource use. If you’d like to explore additional scenarios such as mixed workload environments or multiple node pools, let’s continue the discussion

Editar esta respuesta

How can I effectively test the autoscaler after setup?

Here’s a detailed approach to effectively test your AKS Cluster Autoscaler after setup:

**1. Simulate a Scale-Up Scenario**

The autoscaler reacts to unschedulable pods due to resource constraints. To trigger this:

* **Deploy a Resource-Intensive Application:** Create a deployment with significant resource requests. For example, define a manifest for an application that requires a considerable amount of CPU or memory. This can force pods into a pending state if the current nodes are full.

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: load-generator

spec:

replicas: 10

selector:

matchLabels:

app: load-generator

template:

metadata:

labels:

app: load-generator

spec:

containers:

- name: load

image: busybox

command: ["sh", "-c", "sleep 300"]

resources:

requests:

cpu: "500m"

memory: "512Mi"

Apply this deployment:

sh

kubectl apply -f load-generator.yaml

* **Scale the Deployment Further:** Increase the number of replicas so the resource demand exceeds your current nodes’ capacity:

sh

kubectl scale deployment load-generator --replicas=50

This should create pending pods because the nodes don’t have enough available resources.

* **Observe Pending Pods:** Check for pending pods:

sh

kubectl get pods --all-namespaces --field-selector=status.phase=Pending

Seeing pending pods confirms that there is pressure on the cluster, which is a key trigger for a scale-up.

**2. Monitor Autoscaler Logs and Node Changes**

* **Check Autoscaler Logs:** The Cluster Autoscaler runs as a deployment in the kube-system namespace. View its logs to see decisions regarding scale-up events:

sh

kubectl logs -n kube-system deployment/cluster-autoscaler

Look for logs indicating “scale up” actions. The logs will report when the autoscaler has recognized pending pods and decided to add nodes.

* **Watch Node Count:** In a separate terminal, monitor the node count in real time:

sh

kubectl get nodes -w

Once the autoscaler kicks in, you should see new nodes being added to the cluster. You can also use the Azure CLI:

sh

az aks show --resource-group myResourceGroup --name myAKSCluster --query "agentPoolProfiles[\*].count"

**3. Test a Scale-Down Scenario**

After testing scale-up, you’ll want to see scale down in action:

* **Reduce the Workload:** Once the demand has subsided (for instance, by scaling down your deployment):

sh

kubectl scale deployment load-generator --replicas=0

* **Wait for the Cool-Down Period:** The Cluster Autoscaler uses a grace period to prevent jitter and flapping, so give it a few minutes after the workload drops.
* **Verify Node Reduction:** Monitor the node count again:

sh

kubectl get nodes -w

The nodes that become idle should be removed as the autoscaler detects underutilization.

**4. Additional Considerations**

* **Combine with Pod Autoscaling:** Although testing the autoscaler focuses on node-level scaling, you can gain insights by also verifying that your Horizontal Pod Autoscaler (HPA) is correctly adjusting pod counts based on metrics.
* **Use Monitoring Tools:** Leverage Azure Monitor or Prometheus (if set up) to get dashboards on node utilization and autoscaling events.
* **Testing Environment:** Always perform these tests in a non-production environment to avoid any disruption to critical workloads.

By following these steps, you’ll create controlled conditions where pending pods trigger a scale-up and subsequent idle states trigger a scale-down in your AKS cluster. This not only confirms that the autoscaler is configured correctly but also helps you understand its behavior under varying load conditions.

Would you like to dive into more advanced testing scenarios or tools for better observability?

What common pitfalls should I avoid when using the autoscaler?

Here are some common pitfalls to avoid when implementing and using the Cluster Autoscaler in AKS:

**1. Inadequate or Misconfigured Pod Resource Requests**

* **Pitfall:** If your pods don’t define realistic CPU and memory requests (or define them too high), the autoscaler may not detect pending unschedulable pods correctly. Without proper resource requests, the autoscaler might not trigger a scale-up or could even trigger it unnecessarily.
* **Tip:** Always define realistic and accurate resource requests for your application workloads. This helps the autoscaler understand when the current nodes are under pressure and need additional capacity.

**2. Node Pool Setup and Configuration Issues**

* **Pitfall:** The autoscaler in AKS requires that your node pools use Virtual Machine Scale Sets (VMSS). If you’re using node pools based on availability sets or misconfigured scaling parameters (e.g., narrow min/max ranges), the autoscaler might not work as expected.
* **Tip:** Verify that your node pools support autoscaling by using VMSS and ensuring that autoscaling is properly enabled with sensible minimum and maximum node counts. You can check this configuration with:

sh

az aks show --resource-group myResourceGroup --name myAKSCluster --query "agentPoolProfiles"

**3. Quota and Resource Exhaustion**

* **Pitfall:** Running into Azure subscription limits—such as VM quotas or IP address exhaustion in your virtual network—can prevent new nodes from being added even when scaling is triggered.
* **Tip:** Always check and monitor your subscription quotas and subnet capacities. Use commands like:

sh

az vm list-usage --location your-region

to ensure you have sufficient quota and IP space. 2

**4. Manual Scaling Conflicts**

* **Pitfall:** Manually changing the node count via the Azure CLI or portal while the autoscaler is enabled can conflict with autoscaling decisions. This may lead to unexpected behavior or even prevent the autoscaler from acting as intended.
* **Tip:** When autoscaling is enabled, avoid manual changes to the node count. Rely on autoscaler logs and events to ensure it’s making the right decisions based on workload changes.

**5. Overly Aggressive or Conservative Scaling Parameters**

* **Pitfall:** Setting the minimum or maximum node count too high (or too low) can result in inefficient resource utilization. A too-conservative range may not leave enough headroom during demand spikes, whereas too-high settings might lead to unnecessary costs.
* **Tip:** Choose a scaling range that reflects your actual workload requirements. It’s often useful to monitor your cluster over time and adjust these values as you better understand usage patterns.

**6. Ignoring Node Affinity, Taints, and Tolerations**

* **Pitfall:** If pods have strict node affinity rules or if taints are applied without appropriate tolerations, even scaled-up nodes may not schedule pending pods. The autoscaler might detect unschedulable pods but then not be able to place them on any node because of these constraints.
* **Tip:** Ensure that your pod scheduling constraints (affinity, taints, and tolerations) are properly configured to allow new nodes to accept the workloads that triggered the scale-up.

**7. Lack of Monitoring and Logging**

* **Pitfall:** Without monitoring the autoscaler’s behavior through logs and cluster events, you might overlook errors or misconfigurations that prevent proper scaling.
* **Tip:** Regularly check the autoscaler logs in the kube-system namespace:

sh

kubectl logs -n kube-system deployment/cluster-autoscaler

and use Azure Monitor or similar tools to gain insight into scaling events and resource utilization.

**8. Conflicts with Other Autoscaling Mechanisms**

* **Pitfall:** When using the Cluster Autoscaler in conjunction with the Horizontal Pod Autoscaler (HPA) or the Vertical Pod Autoscaler (VPA), there can be conflicts if they are not tuned to work together.
* **Tip:** Make sure that the combined autoscaling strategy is well thought-out. For instance, allow HPA some buffer so that it doesn’t demand more pods than the cluster can immediately provision, and adjust scaling profiles as necessary.

By avoiding these pitfalls, you can ensure that your AKS Cluster Autoscaler functions smoothly, responding accurately to workload demands while balancing cost and performance.

Would you like additional insights on monitoring these components or integrating other scaling solutions such as HPA or VPA for a more comprehensive autoscaling strategy?

Editar esta respuesta

are affinity, taints, and tolerations in pods configuration required to have an aks cluster autoscaler working properly with scale up and down in the number of nodes automatically?

Affinity, taints, and tolerations are **not required** for the Cluster Autoscaler to function. The core job of the autoscaler is to monitor unschedulable pods—typically those waiting because of insufficient CPU, memory, or other resource constraints—and then adjust the number of nodes accordingly.

That said, here’s how these scheduling features interact with autoscaling:

1. **Affinity:** Pod affinity (and anti-affinity) rules help control where pods are scheduled based on labels and other node or pod attributes. They are useful for dictating workload placement but are not needed for triggering node scale-up or scale-down events. However, if set too restrictively, they might cause pods to remain unschedulable, which in turn would trigger unnecessary scale-up events. It’s important to strike a balance so that these rules fit well with your autoscaling strategy.
2. **Taints and Tolerations:** Taints are used to repel pods from nodes unless the pods have a matching toleration. While they ensure that specific nodes host only particular kinds of workloads, they do not play a central role in the resource-based decisions the autoscaler makes. If configured correctly, they will not interfere with scaling operations. But if used incorrectly, they might lead to situations where pods remain pending—even though new nodes are added—because the new nodes might be tainted in a way that the pending pods can't tolerate. This situation can delay both scale-up (if pods are constantly unschedulable) and scale-down (if nodes host pods that shouldn’t be evicted).
3. **Practical Implications:**
   * **Basic Autoscaling:** For routine autoscaling based on resource needs, you can operate without any affinity, taints, or tolerations. The Cluster Autoscaler simply looks for pods waiting for resources and scales the cluster size accordingly.
   * **Advanced Scheduling:** When you add these configurations, ensure your rules are intentionally designed so that new nodes can accommodate the pending pods. Misconfigured rules can lead to scheduling conflicts that might undermine the autoscaler’s effectiveness.

In summary, while you don’t **need** affinity, taints, or tolerations for the autoscaler to work, **if you use them, ensure they are configured in a way that complements your autoscaling goals**. This will help avoid unintended scheduling issues that could otherwise lead to either excessive scaling (or not scaling enough) based on unschedulable pods that aren’t actually blocked by resource shortages.

Would you like to explore some practical examples or advanced configurations that combine autoscaling with these scheduling decisions?