Introduction

This document addresses troubleshooting scenarios for networking issues related to an AKS cluster. There are two main sections.

The first section **'Common Networking flows'** covers inbound and outbound networking scenarios, addressing problems that occur when traffic goes into or out of the AKS cluster.

For inbound scenarios this document provides step-by-step guidance on troubleshooting connectivity issues to applications that are hosted on the AKS cluster. This could include issues related to firewall rules, network security groups, or load balancers, verifying network connectivity, checking application logs, and examining network traffic to identify potential bottlenecks or other issues.

For outbound scenarios it covers troubleshooting issues related to traffic leaving the AKS cluster, such as connectivity issues to external resources like databases, APIs, or other services hosted outside of the AKS cluster.

The second section **'AKS Triage and Troubleshooting labs'** put the above section into practice using a series of labs that cover common field scenarios. Every lab has a common format of starting with a working use case, that has an issue introduced causing the solution to break. Participants will then use well-known tools to triage, troubleshoot and resolve issues.

This section provides step-by-step labs for resolving connectivity issues, DNS and external access failures, endpoint connectivity issues across virtual networks, web server failures, and failed applications. Participants will also learn how to enable AKS monitoring and logging to help you quickly diagnose and resolve issues.

Whether you are a beginner or experienced user, this document will help you develop the necessary skills required to troubleshoot networking scenarios in AKS and quickly identify and resolve common networking issues in your AKS deployments.

Common networking flows

Inbound traffic scenarios

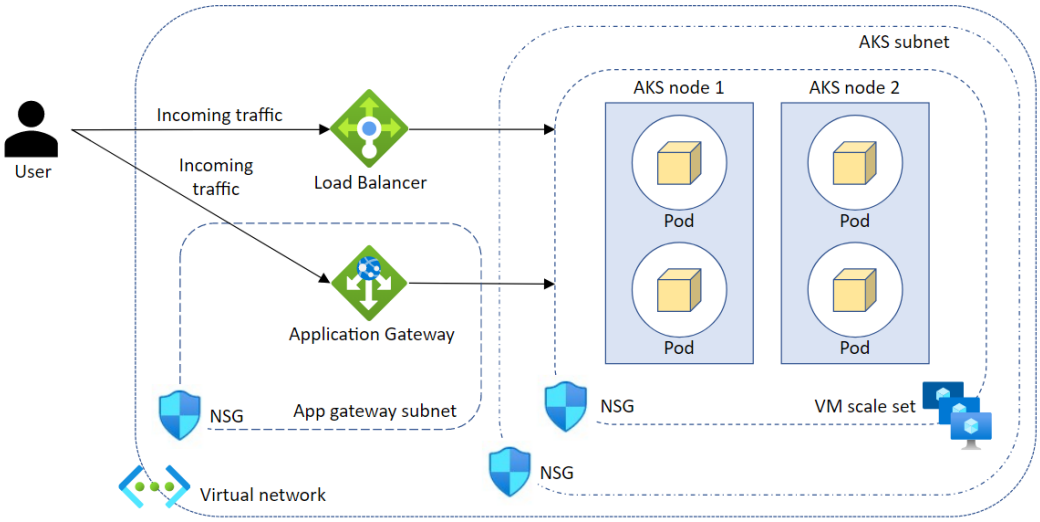
Inbound troubleshooting scenarios will involve 1) connection issues to applications hosted on AKS cluster or 2) API server related issues blocking connection to AKS cluster

Application related connectivity issues

In general, the request flow for accessing applications that are hosted on an AKS cluster is as follows.

```
Client >> DNS name >> AKS load balancer IP address >> AKS nodes >> Pods
```

The basic request flow is shown in the following diagram.



Step 1: Check whether the pod is running and the app or container inside the pod is responding correctly

Is the Pod running?

```
# List pods in the specified namespace.
kubectl get pods -n <namespace-name>

# List pods in all namespaces.
kubectl get pods -A
```

If not running, check Pod events and logs of containers

```
kubectl describe pod <pod-name> -n <namespace-name>
kubectl logs <pod-name> -n <namespace-name>

# Check logs for an individual container in a multicontainer pod.
kubectl logs <pod-name> -n <namespace-name> -c <container-name>

# Dump pod logs (stdout) for a previous container instance.
kubectl logs <pod-name> --previous

# Dump pod container logs (stdout, multicontainer case) for a previous container instance.
kubectl logs <pod-name> -c <container-name> --previous
```

If running, test connectivity using test pod in cluster to connect to suspect pod.

```
# Start a test pod in the cluster:
kubectl run -it --rm aks-ssh --image=debian:stable

# After the test pod is running, you will gain access to the pod.
# Then you can run the following commands:
apt-get update -y && apt-get install dnsutils -y && apt-get install curl -y
```

After the packages are installed, test the connectivity to the application pod:

```
curl -Iv http://<pod-ip-address>:<port>
```

Step 2: Check whether the application is reachable from the service

Is Pod exposed to service?

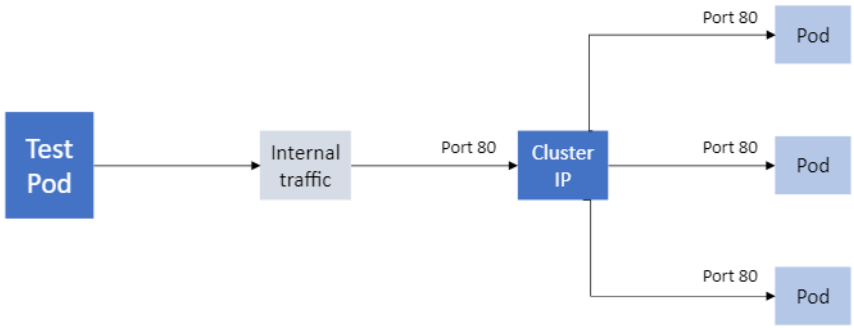
```
# Check the service details.
kubectl get svc -n <namespace-name>

# Describe the service.
kubectl describe svc <service-name> -n <namespace-name>
```

Is Pods IP present as Endpoint in the service?

```
kubectl get pods -o wide  # Check the pod's IP address.
kubectl describe service my-cluster-ip-service  # Check the endpoints in the service.
kubectl get endpoints  # Check the endpoints directly for verification.
```

Step 3: Can the SERVICE be reached through the ClusterIP?

Use a test pod to validate access to the Service IP

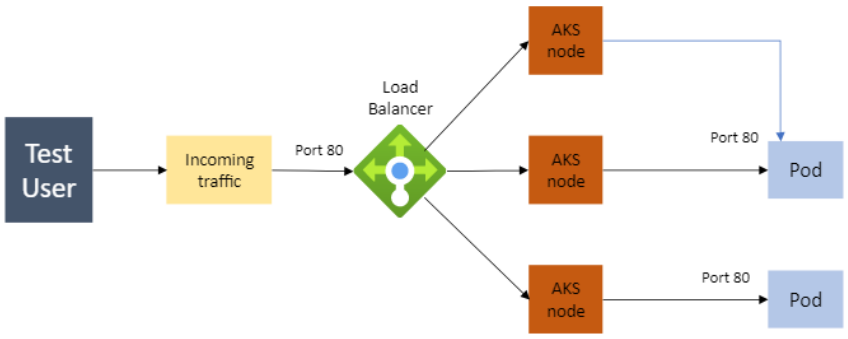



```
# Start a test pod in the cluster:
kubectl run -it --rm aks-ssh --image=debian:stable

# After the test pod is running, you will gain access to the pod.
# Then, you can run the following commands:
apt-get update -y && apt-get install dnsutils -y && apt-get install curl -y

# After the packages are installed, test the connectivity to the service:
kubectl exec -it aks-ssh -- curl -Iv http://<service-ip-address>:<port>
```

Step 4: Can the LoadBalancer be reached from outside the AKS cluster?

Use curl to verify

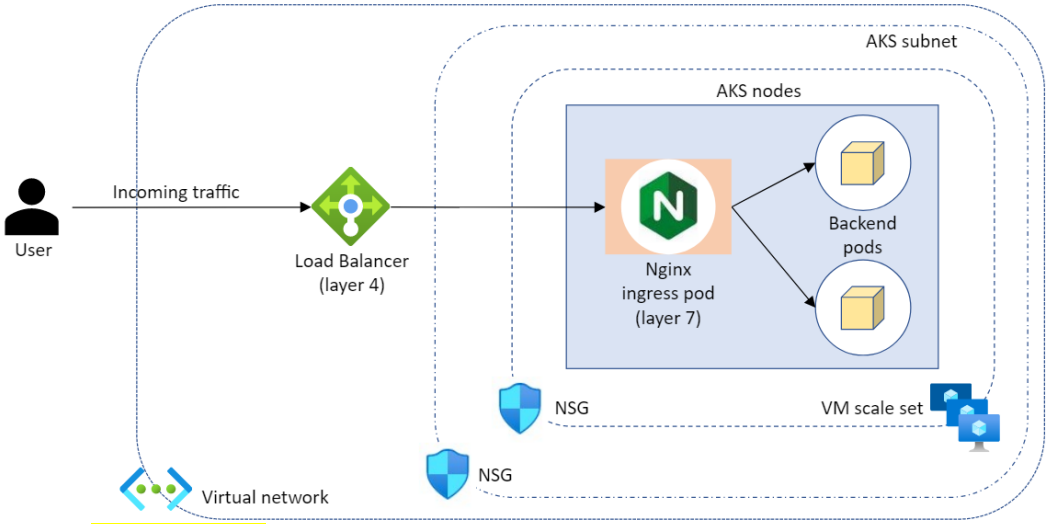
```
curl -Iv http://<service-ip-address>:<port>
```



Step 5: Is Ingress being used instead of a Service?

For scenarios in which the application is exposed by using an Ingress resource, the traffic flow resembles the following progression:

```
Client >> DNS name >> Load balancer or application gateway IP address >> Ingress pods inside the cluster >> Service or pods
```



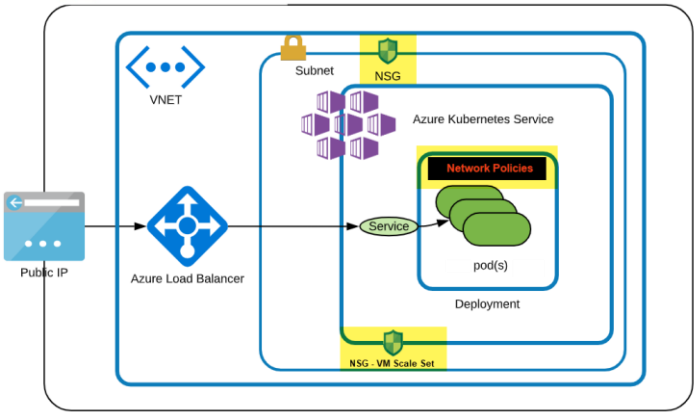
Apply the inside-out approach of troubleshooting here by checking the ingress and ingress controller details for more information.

```
# Check the ingress details and events.
$ kubectl get ing -n <namespace-of-ingress>

# Check Path and URL routing with backends look correct and Event show Normal
$ kubectl describe ing -n <namespace-of-ingress> <ingress-name from above>
```

```
# Verify that the back-end services are running
$ kubectl get svc -n <namespace-of-ingress>

Verify the logs for the ingress controller pods if there's an error:
# Get the ingress controller pods.
$ kubectl get pods -n <namespace-of-ingress>
# Check logs from the pods.
$ kubectl logs -n <ingress-ns> <ingress-name>
```

Q. Client requests don't have log entries in ingress controller pods
A. Requests might not be reaching the cluster, and the user might be receiving a Connection Timed Out error message.

Step 6: Is there an NSG blocking traffic?



Does client requests results in 'Connection Timed Out' error message?
Check the Network Security Group (NSG) that's associated with the AKS nodes. Also, check the AKS subnet. It could be blocking the traffic from the load balancer or application gateway to the AKS nodes.

```
# Check Pod events if Running with no Restarts and make your way out.
$ kubectl describe pod <pod-name>

# Launch the test pod to check access to the Application pod
$ kubectl run -it --rm aks-ssh --image=debian:stable

# Install packages inside the test pod.
apt-get update -y && apt-get install dnsutils -y && apt-get install curl -y

# Try to check access to the pod using the pod IP address from the "kubectl get" output.
curl -Iv http://<pod-ip>
.. HTTP/1.1 200 OK
```

The pod is accessible directly. Therefore, the application is running.

For LoadBalancer service type request flow from the end client to the pod will be as follows:

```
Client >> Load balancer >> AKS node >> Application pod
```

In this request flow, we can block the traffic through the following components:
1. Network policies in the cluster
2. Network Security Group (NSG) for the AKS subnet and AKS node

To check network policy use below to see any custom outside from default policy exists.

```
$ kubectl get networkpolicy --all-namespaces
```

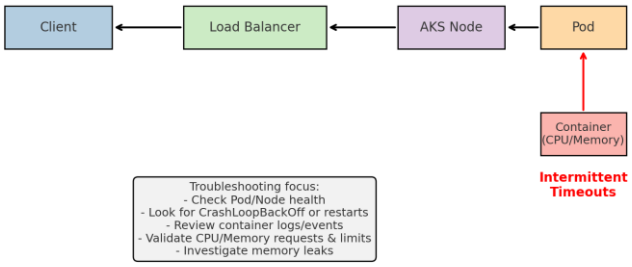
To check for NSG use the Portal and for selected NSG, validate Inbound and Outbound port rules tab have no blockers.

Remember AKS applies NSG rule only to its managed NSG's. It won't modify custom NSGs which will require manual intervention.

Solution:

If the application is enabled for access on a certain port, you must make sure that the custom NSG allows that port as an Inbound rule. After the appropriate rule is added in the custom NSG at the subnet level, the application is accessible.

Step 7: Intermittent timeouts on application access?



When you run a cURL command, you occasionally receive a "Timed out" error message.

One connection is successful, which results in a HTTP 200 response. Next connection gets Timed out.

```
$ curl -Iv http://<External-IP>
```

Cause:

Intermittent time-outs suggest component performance issues, as opposed to networking problems.

1. Check usage and health of the components.

```
# Check the health of the pods and the nodes.
$ kubectl top pods/nodes

# Check the state of the pod. No restarts should be seen that results in CrashLoopBackoff
$ kubectl get pods

# Check logs for the pod on per container basis
$ kubectl logs <pod> -c <container1.. containerN>

# Check any Pod events
$ kubectl describe pod <pod-name>
```

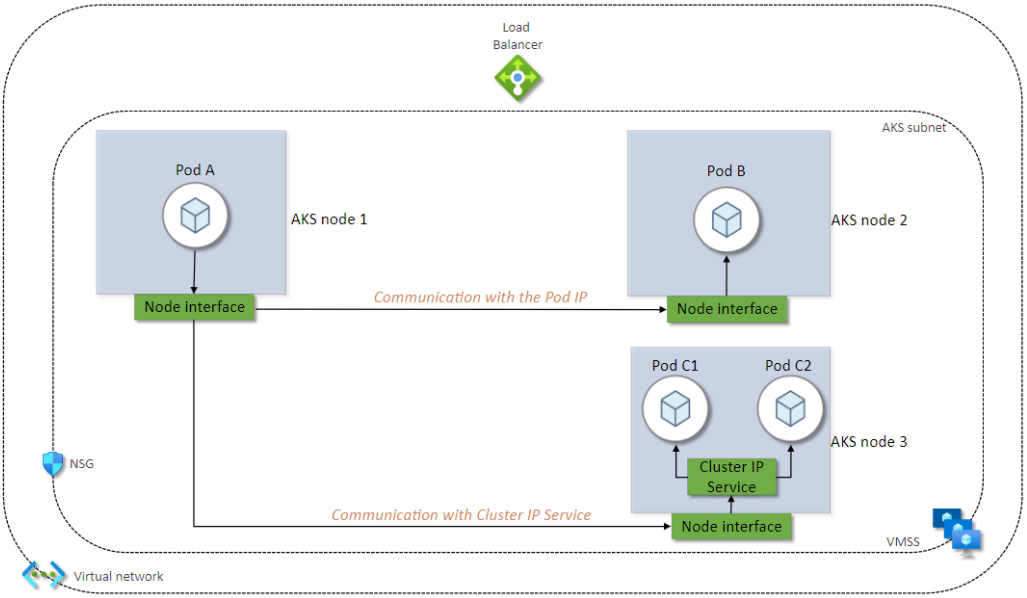
 # Check for Memory/CPU Limits and Requests

Remove the memory limit and monitor the application to determine how much memory it needs. After you learn the memory usage, you can update the memory limits on the container. If the memory usage continues to increase, determine whether there's a memory leak in the application.

Outbound traffic scenarios

Outbound AKS cluster traffic can be classified into 1) Traffic internal to the cluster (Pod-Pod, Pod-Service) or 2) Traffic leaving the cluster as external through Load Balancer or Firewall.

Internal traffic



Internal scenarios that could affect outbound connectivity cover below two scenarios:

- Traffic to a pod or service within the same cluster
- Traffic within virtual networks i.e., to a device or endpoint in the same virtual network or a different virtual network (that uses virtual network peering)

Traffic to a pod or service within the same cluster

Triage steps for traffic to a pod or service within the same cluster:

1. Check resource utilization: **kubectl top pod** and **kubectl top node**
2. Check the status of the kubernetes components: **kubectl get componentstatuses**
3. Check the events in the namespace: **kubectl describe pod <pod-name>** and **kubectl describe svc <service-name>**
4. Verify that there are no network policy restrictions: **kubectl get networkpolicy**

Troubleshooting steps for traffic to a pod or service within the same cluster:

1. Verify the Endpoints: The first step is to check if the endpoints for the service are available and accessible. Use the **kubectl get endpoints** command to view the endpoints associated with the service. If the endpoints are missing or not accessible, there may be a problem with the deployment, the pod, or the service definition.

2. Check Networking: Verify that the networking in the cluster is working correctly.
    1. Make sure that the pods can communicate with each other using the correct IP addresses and ports. You can use the **kubectl describe pod** command to see the IP address of a pod and verify that it is accessible from other pods.

2. Check the network connectivity between the pod and service: **kubectl exec <pod-name> -- nslookup <service-name>**

3. Check Pod Status: Check the status of the pod using the **kubectl get pods** command. If the pod is not running or is in a crash loop, there may be an issue with the pod specification or with the image used to run the pod.

4. Verify the Service: Ensure that the service is defined correctly and is accessible from the pods.
    1. Use **kubectl get svc** and **describe service** commands to see the details of the service and check that it is accessible from the pods.
    2. Verify that the service and pod are in the same namespace: **kubectl describe svc <service-name>** and **kubectl describe pod <pod-name>**. However, it's possible for them to talk if Service uses FQDN of Pod in another namespace.
    3. Verify that the service and pod have the correct labels: **kubectl describe svc <service-name>** and **kubectl describe pod <pod-name>**
    4. Verify that the service selector matches the pod labels: **kubectl describe svc <service-name>**
    5. Check the ClusterIP, External IP, and LoadBalancer IP of the service: **kubectl describe svc <service-name>**

5. Review Logs: Check the logs for the pod using **kubectl logs <pod-name>,** and for any related components such as the API server, controller-manager, and scheduler to see if there are any error messages that can provide further insight into the issue.

6. Use Network Tools: If all the above steps do not yield a resolution, you can use network debugging tools such as **tcpdump** and **nsenter** to capture network traffic and diagnose any issues with the communication between pods.

7. Check for Resource Constraints: Finally, ensure that the pods and services are not being constrained by resource limits such as CPU or memory. You can use the **kubectl describe pod** command to view the resource usage for a pod.

Traffic to a device or endpoint in the same virtual network

AKS triage and troubleshooting steps are listed below for traffic involving AKS device or endpoint within the same Virtual network:

1. Verify connectivity:
    - Check the network security group (NSG) rules: Ensure that the correct ports are open for communication between the AKS and the device/endpoint. Check NSG rules using: **az network nsg rule show --nsg-name <nsg_name> --name <rule_name> --resource-group <resource_group>**
    - Verify the subnet configurations: Ensure that both the AKS and the device/endpoint are in the same subnet and have correct IP addresses. Verify subnet configurations using: **az network vnet subnet show --name <subnet_name> --vnet-name <vnet_name> --resource-group <resource_group>**
    - Verify the Virtual Network peering: Ensure that the virtual networks are properly peered, and traffic is allowed between them. Verify peering using: **az network vnet peering show --name <peering_name> --resource-group <resource_group>**

2.Check network configuration:

1. Ensure that the AKS and the device/endpoint are in the same Virtual Network: Confirm that both the AKS and the device/endpoint are in the same Virtual Network, this is important for network traffic to be properly routed between them. Verify using: **az aks show --name <aks_name> --resource-group <resource_group>**
2. Check the subnet configuration: Ensure that both the AKS and the device/endpoint are in the same subnet, this will make sure that they are part of the same network segment and can communicate directly with each other. Verify using: **az network vnet subnet show --name <subnet_name> --vnet-name <vnet_name> --resource-group <resource_group>**

3. Monitor network traffic:
    1. Use network monitoring tools such as Azure Network Watcher or tcpdump: These tools can be used to monitor network traffic between the AKS and the device/endpoint, this provides insights into what traffic is being sent, received, and any potential issues.
    2. Azure Network Watcher can be used to monitor network traffic using : **az network watcher flow-log show --nsg-name <nsg_name> --resource-group <resource_group>**

4. Check endpoints:
    1. Verify the endpoints and their configurations: Ensure that the endpoints such as the Load Balancer and the Service are properly configured and available for communication. Endpoints configurations can be verified using:

        az network lb show --name <lb_name> --resource-group <resource_group>

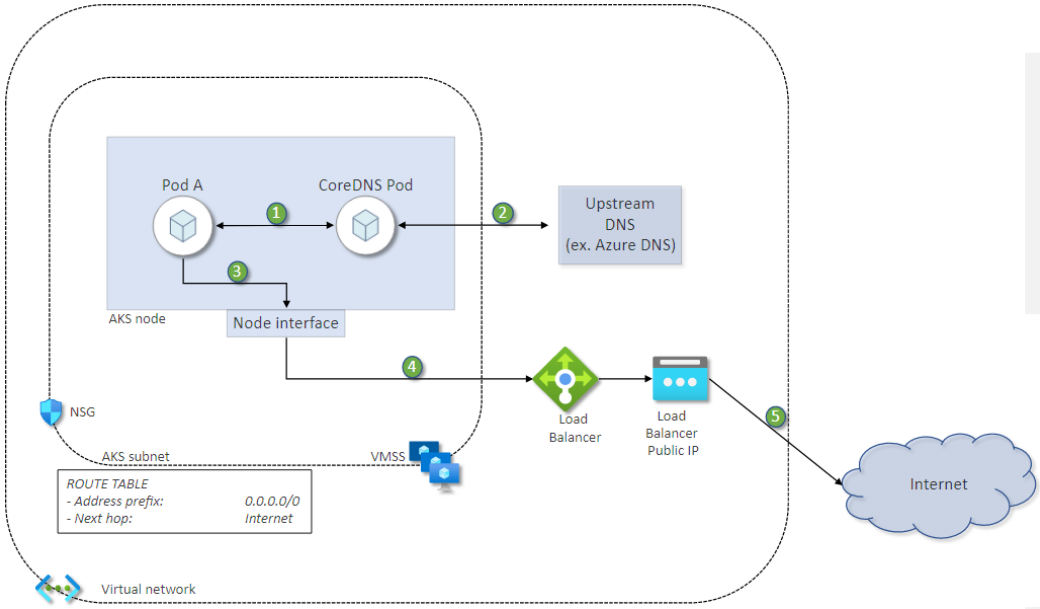
        az network service show --name <service_name> --resource-group <resource_group>
    2. Check Load Balancer configuration: Ensure that the Load Balancer is configured with the correct health probe and load balancing rules, this will make sure that traffic is properly distributed and that the endpoint is healthy.

        az network lb probe show --lb-name <lb_name> --name <probe_name> --resource-group <resource_group>

        az network lb rule show --lb-name <lb_name> --name <rule_name> --resource-group <resource_group>
5. Check logs of the AKS cluster and the device/endpoint: Review the logs for any error messages related to network connectivity, this will help identify if there are any issues with the network traffic.
6. Check DNS resolution:
    1. Ensure that the AKS and device/endpoint can resolve the hostname to the correct IP address: Make sure that the hostname can be resolved to the correct IP address, this is important for network traffic to be properly routed. Use: **nslookup <hostname>**
7. Restart the affected components:
    1. Try restarting the affected components such as the AKS nodes, or the device/endpoint: Restarting the components can sometimes resolve the issue and get the traffic flowing again.
        1. Restart node by using below command, where instance-id = last digit of the instance from ‘k get nodes’

            az vmss restart -g MC_aksrgnp_aksnp_eastus -n aks-nodepool1-58793749-vmss --instance-ids 0

External traffic through Azure Load Balancer (public outbound)



In the above flow:

1. Pod sends DNS Request to CoreDNS
2. CoreDNS realizes that the request is for an external domain and sends it to Upstream DNS Server.
3. Upstream DNS Server responds back, and the Pod receives the IP address.
4. Pod sends traffic to the endpoint, through the Node's Interface
5. The Node sends traffic to the Load Balancer
6. Load Balancer NATs the traffic and sends the packet to the destination using its own Public IP address.

Triage and troubleshooting steps for AKS traffic through Load Balancer (LB):

1. Verify the Load Balancer configuration. Validate using command:
   az network lb show --name <load_balancer_name> --resource-group <resource_group_name>

2. Verify the LB frontend IP address:
   az network lb frontend-ip list -g <resource_group_name> --lb-name <load_balancer_name>

3. Verify LB backend pool status:
   az network lb address-pool list -g <resource_group_name> --lb-name <load_balancer_name>

4. Verify the LB outbound rules to make sure they match the desired traffic routing.

az network lb outbound-rule list --lb-name <load_balancer_name> --resource-group <resource_group_name>
az network lb outbound-rule show --lb-name <load_balancer_name> --resource-group <resource_group_name> --name <from_above>

5. Check the network security groups and firewall rules to ensure the correct ports are open:

- Verify that the firewall allows incoming traffic to the desired port(s). If LB is expected to route traffic to port 80 on the AKS cluster, port 80 should be open in the firewall.
- Ensure the security group rules allow the desired traffic. For example, if the Load Balancer is expected to route traffic to port 80, use below to verify if the security group allows Inbound traffic to port.

  az network nsg rule list --resource-group <resource_group_name> --nsg-name <nsg-name> --query '[[].direction,[].destinationPortRange]'

6. Check the load balancer logs and metrics for any errors or dropped traffic:
   - Review the Azure Monitor logs for the Load Balancer to look for any error messages or dropped traffic.

     az monitor log-analytics query --analytics-query "AzureDiagnostics | where ResourceType == 'Microsoft.Network/loadBalancers' | where Category == 'LoadBalancerProbeHealthStatus' | where TimeGenerated > ago(1h)" --workspace-id <workspace-GUID>

   - Check the Load Balancer metrics for any unusual spikes or dips.

     az monitor metrics list -g <resource_group_name> --resource <load_balancer_name> --metric "PacketCount" --resource-type "Microsoft.Network/loadBalancers"

7. The Load Balancer routes traffic to specific endpoints and IP addresses. It's important to ensure that these endpoints and IP addresses are correct and up to date, otherwise, the Load Balancer may not be able to route traffic correctly.
   - Verify that the endpoints are correct and up to date using **'kubectl get endpoints'**
   - Check the IP addresses of the AKS cluster and make sure they match the expected addresses, using **'kubectl get nodes -o wide'**.

     If there is a mismatch between Internal-IPs and expected IP addresses this could indicate a problem with the AKS cluster network configuration. Check NSG, Route Tables and subnets to investigate further.

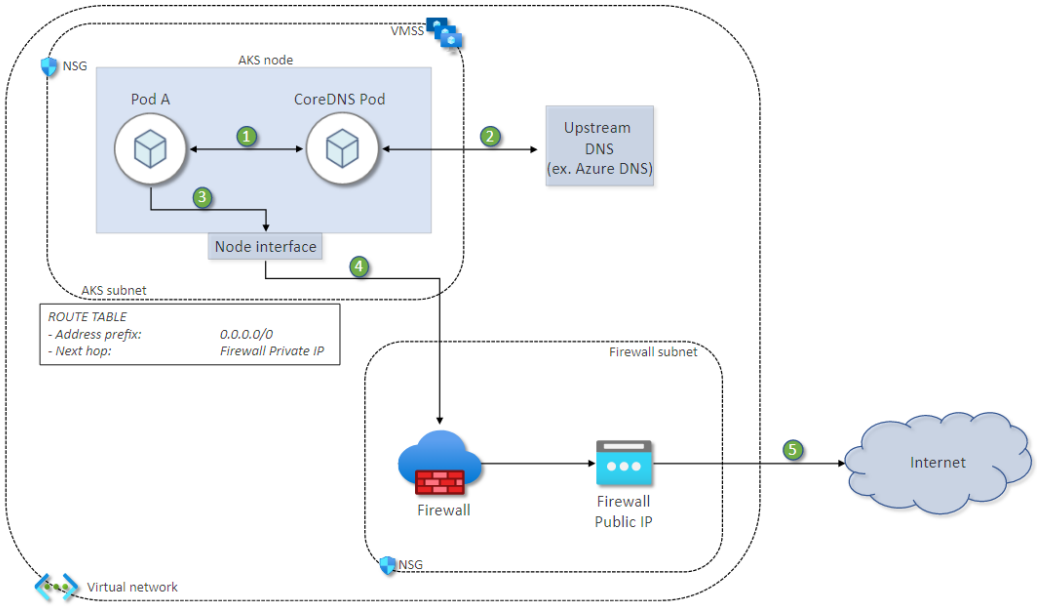
8. Validate the status of the AKS nodes, pods, and services to ensure they are healthy and available:
   - Check the status of the AKS nodes to make sure they are healthy and available. Use **"kubectl get nodes"** to view the node status.
   - Verify the status of the pods in the AKS cluster. Use **"kubectl get pods"** to view the pod status.
   - Check the status of the services in the AKS cluster. Use **"kubectl get services"** to view the service status.

9. Verify that the correct version of Kubernetes is installed and running:
   - Use the **"kubectl version"** command to check the version of Kubernetes running in the AKS cluster.
   - Ensure that the version of Kubernetes matches the expected version and is fully supported.

10. Check the resource utilization of the AKS cluster, including CPU and memory utilization, to ensure that it is not over-utilized:
    - Use the **"kubectl top"** command to view the CPU and memory utilization of the AKS cluster.

11. Use network tools, such as traceroute or tcpdump, to diagnose connectivity issues:
    - Use the **"traceroute"** command to diagnose routing issues between the Load Balancer and the AKS cluster.
    - Use the **"tcpdump"** command to capture network traffic and diagnose issues with the traffic flow.

External traffic through Azure Firewall or Proxy server (public outbound)



In the above flow:

1. Pod sends DNS Request to CoreDNS
2. CoreDNS realizes that the request is for an external domain and sends it to Upstream DNS Server.
3. Upstream DNS Server responds back, and the Pod receives the IP address.
4. Pod sends traffic to the endpoint, through the Node's Interface
5. The Node sends traffic to the Firewall's Private IP, following the Route Table rules
6. Firewall NATs the traffic and sends the packet to the destination using its own Public IP address.

Triage and troubleshooting steps for traffic flow from AKS cluster through the Azure Firewall to the internet:

1. Verify NSG does not block traffic from AKS cluster to Azure Firewall. Below command shows details of the specified NSG, including security rules. To get a specific NSG use list command.

az network nsg show -g <resource_group_name> --name <nsg_name>

Verify rules that ensure desired traffic is allowed to pass through NSG.

2. Verify Azure Firewall is properly configured using the command below. Ensure it's properly connected to the virtual network and subnet and that it has a public IP address.

az network firewall show -g <resource_group_name> --name <firewall_name>

3. Another factor that could block traffic from the AKS cluster to the internet through the Azure Firewall is the *network routes*. Command lists all the routes for the specified resource group and route table.

az network route-table list -g <resource_group_name> --route-table-name <route_table_name>

You can check the routes to ensure that the desired traffic is being directed to the Azure Firewall as specified in nextHopIpAddress and nextHopType.

AKS Triage and Troubleshooting Labs

AKS setup instructions

1. Use earlier lab instructions to setup AKS or reuse existing AKS

2. Setup credentials:

```
az aks get-credentials -g $resource_group -n $aks_name --overwrite-existing
```

3. Switch to the newly created namespace

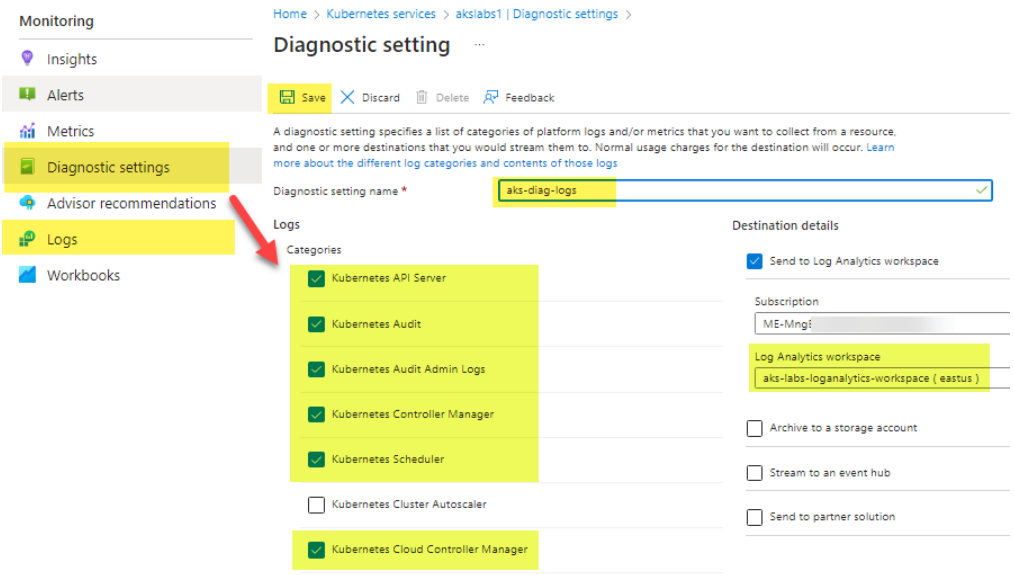
```
kubectl create ns student
kubectl config set-context --current --namespace=student
# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
# Confirm ability to view
kubectl get pods -A
```

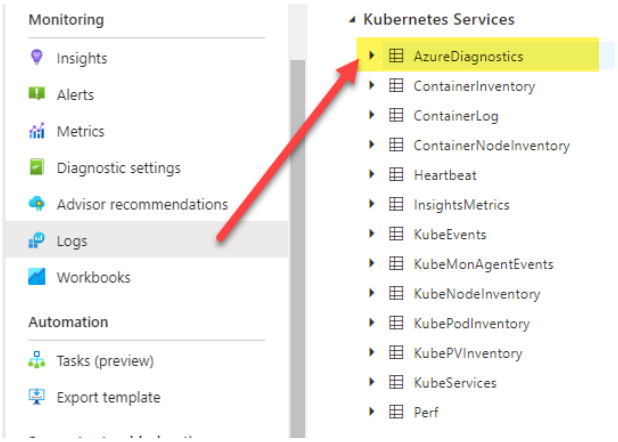
4. If jq and curl isn't installed, open a Powershell as Admin, and run below commands:

```
choco install jq -y
choco install curl -y
choco install grep -y
```

5. Enable AKS Diagnostics logging using Azure Portal > AKS as shown.

   This should be visible from Portal in AKS > Diagnostics settings. After some time, `AzureDiagnostics` should appear in Monitoring > Logs as shown.

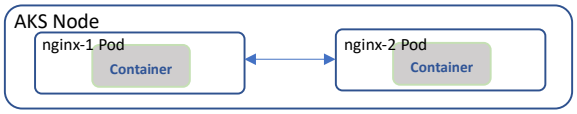



Monitoring
Insights
Alerts
Metrics
Diagnostic settings
Advisor recommendations
Logs
Workbooks
Automation
Tasks (preview)
Export template
Kubernetes Services
AzureDiagnostics
ContainerInventory
ContainerLog
ContainerNodeInventory
Heartbeat
InsightsMetrics
KubeEvents
KubeMonAgentEvents
KubeNodeInventory
KubePodInventory
KubePVInventory
KubeServices
Perf

Lab 1: Connectivity resolution to pods or services in same cluster.

Objective: The goal of this exercise is to troubleshoot and resolve connectivity between pods and services within the same Kubernetes cluster.

Layout: Simple cluster layout with 2 Pods created by their respective deployments and exposed using Cluster IP Service.



Step 1: Set up the environment

1. Setup up AKS as outlined in this section.

2. Create and switch to the newly created namespace

```
kubectl create ns student
kubectl config set-context --current --namespace=student
# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
```

Step 2: Create two deployments and respective services

1. Create a deployment nginx-1 with a simple nginx image:

```
kubectl create deployment nginx-1 --image=nginx
```

2. Expose the deployment as a ClusterIP service:

```
k expose deployment nginx-1 --name nginx-1-svc --port=80 --target-port=80 --type=ClusterIP
```

3. Repeat the above steps to create another deployment and a service:

```
kubectl create deployment nginx-2 --image=nginx
k expose deployment nginx-2 --name nginx-2-svc --port=80 --target-port=80 --type=ClusterIP
```

4. Confirm deployment and service functional. Pods should be running and services listening on Port 80.

```
kubectl get all
```

Step 3: Verify that you can access both services from within the cluster by using Cluster IP addresses

```
# Services returned: nginx-1-svc for pod/nginx-1, nginx-2-svc for pod/nginx-2
kubectl get svc

# Get the value of nginx-1-pod and nginx-2-pod
$nginx1pod = kubectl get pod -l app=nginx-1 -o jsonpath="{.items[0].metadata.name}"
$nginx2pod = kubectl get pod -l app=nginx-2 -o jsonpath="{.items[0].metadata.name}"

# below should present HTML page from nginx-2
kubectl exec -it $nginx1pod -- curl nginx-2-svc:80

# below should present HTML page from nginx-1
kubectl exec -it $nginx2pod -- curl nginx-1-svc:80


# check endpoints for the services
kubectl get ep
```

Step 4: Backup existing deployments

1. Backup the deployment associated with nginx-2 deployment:

```
kubectl get deployment.apps/nginx-2 -o yaml > nginx-2-dep.yaml
```

2. Backup the service associated with nginx-2 service:

```
kubectl get service/nginx-2-svc -o yaml > nginx-2-svc.yaml
```

Step 5: Simulate service down

1. Delete nginx-2 deployment

```
kubectl delete -f nginx-2-dep.yaml
```

2. Apply the broken.yaml deployment file found in Lab1 folder

```
kubectl apply -f https://raw.githubusercontent.com/jvargh/aks/refs/heads/main/day-3/broken.yaml
```

3. Confirm all pods are running

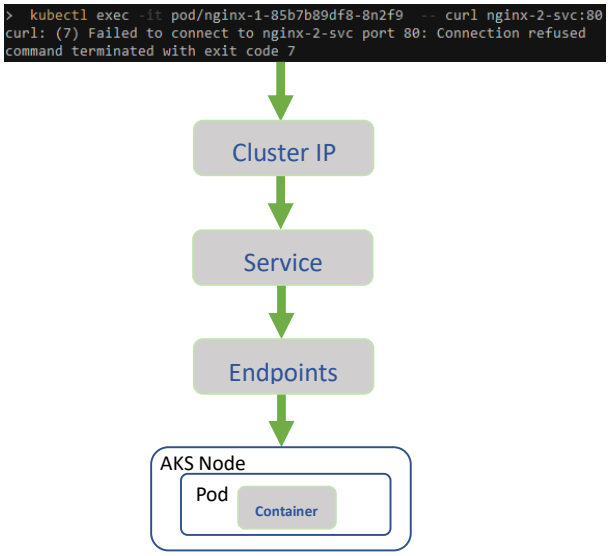
```
kubectl get all
```

4. Get the new values of the pods

```
# Get the value of nginx-1-pod and nginx-2-pod
$nginx1pod = kubectl get pod -l app=nginx-1 -o jsonpath="{.items[0].metadata.name}"
$nginx2pod = kubectl get pod -l app=nginx-02 -o jsonpath="{.items[0].metadata.name}"
```

Step 6: Troubleshoot the issue

Below is the flow. Confirm every step from top down.



1. Check the health of the nodes in the cluster to see if there is a node issue
   ```
   kubectl get nodes
   ```

2. Verify that you can no longer access **nginx-2-svc** from within the cluster
   ```
   kubectl exec -it $nginx1pod -- curl nginx-2-svc:80
   # msg Failed to connect to nginx-2-svc port 80: Connection refused
   ```

3. Verify that you can access **nginx-1-svc** from within the cluster
   ```
   kubectl exec -it $nginx1pod -- curl nginx-1-svc:80
   # displays HTML page
   ```

4. Verify that you can access **nginx-2** locally. This confirms no issue with the nginx-2 application.
   ```
   kubectl exec -it $nginx2pod -- curl localhost:80
   # displays HTML page
   ```

3. Check the Endpoints using below command and verify that the right Endpoints line up with their Services. There should be at least 1 Pod associated with a service, but none seem to exist for nginx-2 service but nginx-2 service/pod association is fine.

   ```
   kubectl get ep
   ```

   ```
   NAME          ENDPOINTS          AGE
   nginx-1-svc   10.244.0.18:80     149m
   nginx-2-svc   <none>             148m
   ```

4. Check label selector used by the Service experiencing issue, using below command:
   ```
   kubectl describe service <service-name>
   ```

   Ensure that it matches the label selector used by its corresponding Deployment using below command:
   ```
   kubectl describe deployment <deployment_name>
   ```

Use ‘k get svc’ and ‘k get deployment’ to get service and deployment names.

Do you notice any discrepancies?

5. Using the Service label selector from #3, check that the Pods selected by the Service match the Pods created by the Deployment using the following command

   kubectl get pods --selector=<selector_used_by_service>.

   If no results are returned then there must be a label selector mismatch.

   From below selector used by deployment returns pods but not selector used by service.

```
$ kubectl get pods --selector=app=nginx-02
NAME                       READY   STATUS    RESTARTS   AGE
nginx-2-7bc7c8f5ff-htnmw   1/1     Running   0          126m
$ kubectl get pods --selector=app=nginx-2
No resources found in student namespace.
```

6. Check service and pod logs and ensure HTTP traffic is seen. Compare nginx-1 pod and service logs with nginx-2. Latter seems empty suggesting no incoming traffic.

```
k logs pod/$nginx2pod
k logs pod/$nginx1pod

k logs svc/nginx-2-svc
k logs svc/nginx-1-svc
```

```
PS C:\Users\LabUser\Downloads\Labs\Lab1> kubectl logs pod/nginx-1-85b7b89df8-s22rf
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/04/10 16:51:05 [notice] 1#1: using the "epoll" event method
2023/04/10 16:51:05 [notice] 1#1: nginx/1.23.4
2023/04/10 16:51:05 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2023/04/10 16:51:05 [notice] 1#1: OS: Linux 5.4.0-1104-azure
2023/04/10 16:51:05 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/04/10 16:51:05 [notice] 1#1: start worker processes
2023/04/10 16:51:05 [notice] 1#1: start worker process 28
2023/04/10 16:51:05 [notice] 1#1: start worker process 29
10.244.0.15 - - [10/Apr/2023:16:52:38 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.74.0" "-"
10.244.0.16 - - [10/Apr/2023:17:11:19 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.74.0" "-"
10.244.0.16 - - [10/Apr/2023:17:11:22 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.74.0" "-"
```

Step 7: Restore connectivity

1. Check the label selector the Service is associated with and get associated pods:

```
# Get label
kubectl describe service nginx-2-svc

# use label from service to get pods
# indicates no resources found or no pods available
kubectl describe pods -l app=nginx-2
```

2. Update deployment and apply changes.

```
kubectl delete -f nginx-2-dep.yaml
```

Download file

```
Invoke-WebRequest `
  -Uri "https://raw.githubusercontent.com/jvargh/aks/refs/heads/main/day-3/broken.yaml" `
  -OutFile ".\broken.yaml"
```

In broken.yaml,

- Update labels > app: nginx-02, to app: nginx-2, as shown below


```
kubectl apply -f broken.yaml # or apply dep-nginx-2.yaml

# Get the value of nginx-1-pod and nginx-2-pod
$nginx1pod = kubectl get pod -l app=nginx-1 -o jsonpath="{.items[0].metadata.name}"
$nginx2pod = kubectl get pod -l app=nginx-2 -o jsonpath="{.items[0].metadata.name}"
$nginx1pod
$nginx2pod

k describe pod $nginx2pod
k describe pod $nginx1pod

k get ep # nginx-2 svc should have pods unlike before
```

3. Verify that you can now access the newly created service from within the cluster:

```
# Should return HTML page from nginx-2-svc and vice versa
kubectl exec -it $nginx1pod -- curl nginx-2-svc:80
kubectl exec -it $nginx2pod -- curl nginx-2-svc:80

# Confirm from logs
k logs pod/$nginx2pod
k logs pod/$nginx1pod
```

Step 8: Using Custom Domain Names

Currently Services in your namespace 'student' will resolve using `<service name>.<namespace>.svc.cluster.local`. Below command should return web page.

```
k exec -it <nginx-1 pod> -- curl nginx-2-svc.student.svc.cluster.local
```

1. Apply broken2.yaml in Lab1 folder and restart coredns

```
kubectl apply -f broken2.yaml
kubectl delete pods -l=k8s-app=kube-dns -n kube-system
# Monitor to ensure pods are running
kubectl get pods -l=k8s-app=kube-dns -n kube-system
```

2. Validate if DNS resolution works and it should fail wit 'curl: (6) Could not resolve host:'

```
k exec -it <nginx-1 pod> -- curl nginx-2-svc.student.svc.cluster.local
k exec -it <nginx-1 pod> -- curl nginx-2-svc
```

3. Check the DNS configuration files in kube-system which shows the configmap's, as below.

```
k get cm -A -n kube-system | grep dns
```

4. Describe each of the ones found above and look for inconsistencies

```
k describe cm coredns -n kube-system
k describe cm coredns-autoscaler -n kube-system
```

```
k describe cm coredns-custom -n kube-system
```

5. Since the custom DNS file holds the breaking changes, either edit coredns-custom (**to remove data section**) or delete ConfigMap `coredns-custom`. Restart DNS which should re-create `coredns-custom`.

```
k delete cm coredns-custom -n kube-system
kubectl delete pods -l=k8s-app=kube-dns -n kube-system
# Monitor to ensure pods are running
kubectl get pods -l=k8s-app=kube-dns -n kube-system
```

6. Confirm DNS resolution now works as before.

```
k exec -it <nginx-1 pod> -- curl nginx-2-svc.student.svc.cluster.local
```

Challenge lab: Resolve aks.com as below

```
k exec -it <nginx-1 pod> -- curl nginx-2-svc.aks.com
```

 # Solution

```
k apply -f working2.yaml
kubectl delete pods -l=k8s-app=kube-dns -n kube-system
# Monitor to ensure pods are running
kubectl get pods -l=k8s-app=kube-dns -n kube-system

# Confirm working > k exec -it <nginx-1 pod> -- curl nginx-2-svc.aks.com
# Bring back to default
k delete cm coredns-custom -n kube-system
kubectl delete pods -l=k8s-app=kube-dns -n kube-system
# Monitor to ensure pods are running
kubectl get pods -l=k8s-app=kube-dns -n kube-system
```

Step 9: What was in the broken files

In broken.yaml deployment labels didn't match up with the service i.e., it should have been nginx-2

In broken2.yaml changes were made to resolve '`student.svc.cluster.local`' to a custom FQDN like '`bad.cluster.local`', which caused DNS to break.

```
$kubectl_apply=@"
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom
  namespace: kube-system
data:
  internal-custom.override: | # any name with .server extension
    rewrite stop {
      name regex (.*)\.svc\.cluster\.local {1}.bad.cluster.local.
      answer name (.*)\.bad\.cluster\.local {1}.svc.cluster.local.
```

```
    }
"@
$kubectl_apply | kubectl apply -f -
```

Step 10: Cleanup

```
k delete deployment/nginx-1 deployment/nginx-2 service/nginx-1-svc service/nginx-2-svc
```

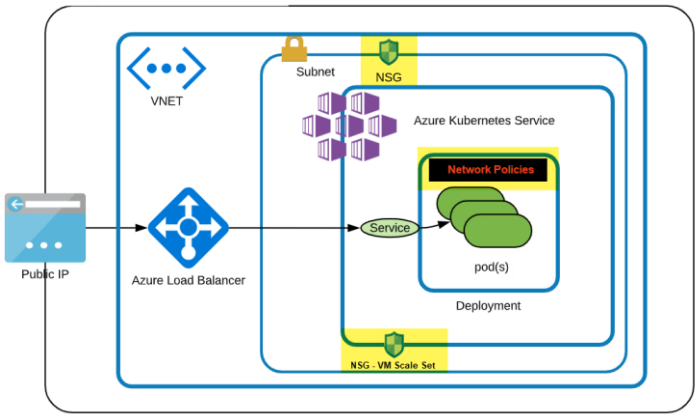
or

```
just delete namespace >  k delete ns student
```

Lab 2: DNS and External access failure resolution

Objective: The goal of this exercise is to troubleshoot and resolve Pod DNS lookups and DNS resolution failures.

Layout: Cluster layout as shown below has NSG applied to AKS subnet, with Network Policies in effect.



Step 1: Set up the environment

1. Setup up AKS as outlined in this section.
2. Run the below

```
az aks update -g $rg -n $aks --network-policy azure
```

3. Confirm update complete

```
az aks show -g aks02day2-rg -n aks02day2 `
  --query "networkProfile.{plugin:networkPlugin,policy:networkPolicy,dataplane:networkDataplane}" -o table
```

4. Create and switch to the newly created namespace

```
kubectl create ns student
kubectl config set-context --current --namespace=student
# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
```

5. Change directory to Lab2 > cd Lab2

Step 2: Verify DNS Resolution works within cluster

1. Create pod for DNS validation within Pod

```
kubectl run dns-pod --image=nginx --port=80 --restart=Never
kubectl exec -it dns-pod -- bash

# Run these commands at the bash prompt
apt-get update -y
apt-get install dnsutils -y
exit
```

2. Test and confirm DNS resolution resolves to the correct IP address.

```
kubectl exec -it dns-pod -- nslookup kubernetes.default.svc.cluster.local
# should say "recursion not available"

kubectl exec -it dns-pod -- sh
```

```
# nslookup
> set norecurse
> kubernetes.default.svc.cluster.local
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   kubernetes.default.svc.cluster.local
Address: 10.0.0.1
```

Step 3: Break DNS resolution

1. From Lab2 apply broken1.yaml

```
kubectl apply -f https://raw.githubusercontent.com/jvargh/aks/refs/heads/main/day-3/broken1.yaml
```

2. Confirm below results in 'connection timed out; no servers could be reached'

```
kubectl exec -it dns-pod -- nslookup kubernetes.default.svc.cluster.local
```

Step 4: Troubleshoot DNS Resolution Failures

1. Verify DNS resolution works within the AKS cluster

```
kubectl exec -it dns-pod -- nslookup kubernetes.default.svc.cluster.local

# If response 'connection timed out; no servers could be reached' then proceed below with troubleshooting
```

2. Validate the DNS service which should show port 53 in use

```
kubectl get svc kube-dns -n kube-system
```

3. Check logs for pods associated with kube-dns

```
$coredns_pod=$(kubectl get pods -n kube-system -l k8s-app=kube-dns -o=jsonpath='{.items[0].metadata.name}')
kubectl logs -n kube-system $coredns_pod
```

4. If a custom ConfigMap is present, verify that the configuration is correct.

```
kubectl describe cm coredns-custom -n kube-system
```

5. Check for networkpolicies currently in effect. If DNS related then describe and confirm no blockers. If network policy is a blocker then have that removed.

```
kubectl get networkpolicy -A
NAMESPACE     NAME                POD-SELECTOR
kube-system   block-dns-ingress   k8s-app=kube-dns


kubectl describe networkpolicy block-dns-ingress -n kube-system
# should show on Ingress path not allowing DNS traffic to UDP 53
```

6. Remove the offending policy

```
kubectl delete networkpolicy block-dns-ingress -n kube-system
```

7. Verify DNS resolution works within the AKS cluster. Below is another way to create a Pod to execute task as nslookup and delete on completion

```
kubectl run -it --rm --restart=Never test-dns --image=busybox --command -- nslookup kubernetes.default.svc.cluster.local
# If the DNS resolution is working correctly, you should see the correct IP address associated with the domain name
```

8. Check NSG has any DENY rules that might block port 80. If exists, then have that removed

```
# Below CLI steps can also be performed as a lookup on Azure portal under NSG
```

Step 5: Create external access via Loadbalancer

1. Expose dns-pod with service type Load Balancer.

```
kubectl expose pod dns-pod --name=dns-svc --port=80 --target-port=80 --type LoadBalancer
```

2. Confirm allocation of External-IP.

```
kubectl get svc
```

3. Confirm External-IP access works within cluster.

```
kubectl exec -it dns-pod -- curl <EXTERNAL-IP>
```

4. Confirm from browser that External-IP access fails from internet to cluster.

```
curl <EXTERNAL-IP>
```

Step 6: Troubleshoot broken external access via Loadbalancer

1. Check if AKS NSG applied on the VM Scale Set has an Inbound HTTP Allow rule.

2. Check if AKS Custom NSG applied on the Subnet has an ALLOW rule and if none then apply as below.

```
$custom_aks_nsg = " aks-agentpool-96834568-nsg" # <- verify from AKS VNet > AKS Subnet
$nsg_list=az network nsg list --query "[?contains(name,'$custom_aks_nsg')].{Name:name,
ResourceGroup:resourceGroup}" --output json

# Extract Custom AKS Subnet NSG name, NSG Resource Group
$nsg_name=$(echo $nsg_list | jq -r '.[].Name')
$resource_group=$(echo $nsg_list | jq -r '.[].ResourceGroup')
echo $nsg_list, $nsg_name, $resource_group

$EXTERNAL_IP="<insert>"
az network nsg rule create --name AllowHTTPInbound `
       --resource-group $resource_group --nsg-name $nsg_name `
       --destination-port-range 80 --destination-address-prefix $EXTERNAL_IP `
       --source-address-prefixes Internet --protocol tcp `
       --priority 100 --access allow
```

3. After ~60s, confirm from browser that External-IP access succeeds from internet to cluster.

```
curl <EXTERNAL-IP>
```

Step 6: What was in the broken files

1. Broken1.yaml is a NP that blocks UDP ingress requests on port 53 to all Pods

```yaml
# Run below network policy in Powershell. This blocks incoming DNS requests.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: block-dns-ingress
  namespace: kube-system
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector: {}
    ports:
    - protocol: UDP
      port: 53
```

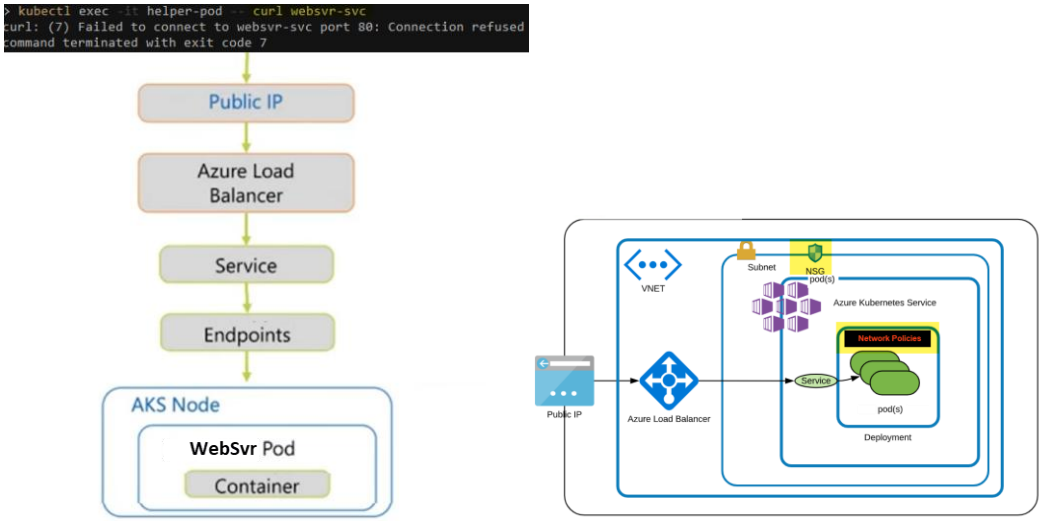
Step 7: Cleanup

```
k delete pod/dns-pod
or
k delete ns student
az network nsg rule delete --name AllowHTTPInbound `
       --resource-group $resource_group --nsg-name $nsg_name
```

Lab 3: Web Server fails to respond

Objective: The aim of this lab is to identify and resolve an issue where traffic directed through a Load Balancer fails to reach the intended pod. The focus will be on troubleshooting the problem until it is resolved.

Layout: Web Server Pod with Service of type LoadBalancer allowing External IP access. Cluster layout as shown below has NSG applied to AKS subnet, with Network Policies in effect.



Step 1: Set up the environment.

1. Setup up AKS as outlined in this section.
2. Create and switch to the newly created namespace

```
kubectl create ns student
kubectl config set-context --current --namespace=student
# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
```

3. Change directory to Lab4 > cd Lab4
4. Run the following in Powershell >`.\working.ps1`

Spec in working.ps1 is seen below. This sets up the Web Server **Pod** and **Service** of type Load Balancer.
There's an External-IP that points to the web server making it accessible from outside cluster.

Commented [JV1]:

```
$kubectl_apply = @"
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-html
data:
  index.html: |
    ...
    Hello from Websvr
    ...
---
apiVersion: v1
kind: Pod
metadata:
  name: websvr
  labels:
    app: websvr
spec:
  containers:
  - name: websvr
    image: nginx:latest
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: webcontent
      mountPath: /usr/share/nginx/html/index.html
      subPath: index.html
  volumes:
    - name: webcontent
      configMap:
        name: nginx-html
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: websvr
  name: websvr-svc
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: websvr
  type: LoadBalancer
"@
$kubectl_apply | kubectl apply -f -

# svc similar to: k expose pod nginx --name= websvr-svc --port=80 --target-port=8080 --type LoadBalancer
```

5. Ensure Websvr is Running. Ensure LB service has an External IP assigned.

```
kubectl get po -l app=websvr -o wide
kubectl get svc -l app=websvr -o wide
kubectl get node -o wide
```

```
$ k get po -o wide
NAME         READY   STATUS    RESTARTS      AGE   IP             NODE
helper-pod   1/1     Running   1 (49m ago)   50m   10.244.0.43    aks-systempool-24510098-vmss000003
websvr       1/1     Running   0             70m   10.244.0.42    aks-systempool-24510098-vmss000003
$ k get svc -l app=websvr -o wide
NAME         TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE   SELECTOR
websvr-svc   LoadBalancer   10.0.200.58   20.237.125.27   80:30703/TCP   66m   app=websvr
$ k get node -o wide
NAME                                STATUS   ROLES   AGE   VERSION   INTERNAL-IP   EXTERNAL-IP
aks-systempool-24510098-vmss000003   Ready    agent   25h   v1.24.9   10.224.0.4    <none>
```

```
$ kubectl get pod -o wide
NAME         READY   STATUS    RESTARTS       AGE     IP             NODE
helper-pod   1/1     Running   1 (8m3s ago)   8m22s   10.244.0.25    aks-nodepool1-96643998-vmss000000
websvr       1/1     Running   0              10m     10.244.0.24    aks-nodepool1-96643998-vmss000000
$ k get svc -l app=websvr -o wide
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE   SELECTOR
websvr-svc   LoadBalancer   10.0.43.186    52.154.210.193   80:31419/TCP   10m   app=websvr
$ k get node -o wide
NAME                                STATUS   ROLES   AGE    VERSION   INTERNAL-IP    EXTERNAL-IP
aks-nodepool1-96643998-vmss000000   Ready    agent   5h7m   v1.24.9   10.224.252.0   <none>
```

Step 2: Create and use a helper pod to troubleshoot.

1. Create a helper pod and install the required utilities.

```
kubectl run helper-pod  --image=nginx
k exec -it helper-pod -- bash
```

```
# Install below in bash shell
apt-get update -y
apt-get install -y nmap
exit
```

2. Make a note of EXTERNAL_IP, CLUSTER_IP, WebsvrPod_IP, Helper_Pod_IP, and Node name.

```
$EXTERNAL_IP=$(kubectl get svc websvr-svc -o jsonpath="{.status.loadBalancer.ingress[*].ip}")
$CLUSTER_IP=$(kubectl get svc websvr-svc -o jsonpath="{.spec.clusterIP}")
$WebsvrPod_IP=$(kubectl get pod websvr -o jsonpath="{.status.podIP}")
$Helper_Pod_IP=$(kubectl get pod helper-pod -o jsonpath="{.status.podIP}")
$Node_Name=$(kubectl get node -o jsonpath="{.items[*].status.addresses[?(@.type=='Hostname')].address}")

echo $EXTERNAL_IP   # 52.154.210.193 only in this instance
echo $CLUSTER_IP    # 10.0.43.186. only in this instance
echo $WebsvrPod_IP  # 10.244.0.24. only in this instance
echo $Helper_Pod_IP # 10.244.0.25. only in this instance
echo $Node_Name     # aks-nodepool1-96643998-vmss000000. only in this instance
```

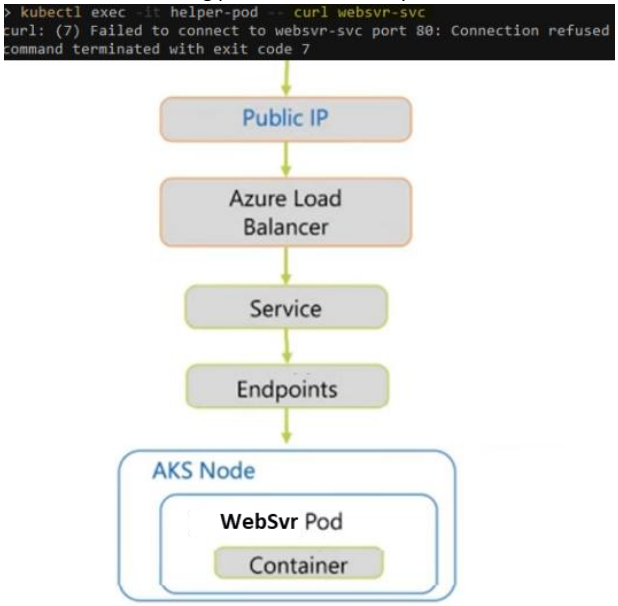
Step 3: Verify connectivity to Web Server.

1. Check connectivity by attempting to reach the web server using Public IP. This check should fail.

```
kubectl exec -it helper-pod -- curl -m 7 ${EXTERNAL_IP}:80
curl: (7) Failed to connect to 52.154.210.193 port 80: Connection refused
command terminated with exit code 7
```

Step 4: Troubleshoot networking

Below is the networking path we need to analyze.



1. Do a port scan using 'nmap' on Websvr-Service, EXTERNAL_IP, and WebsvrPod_IP.

```
kubectl exec -it helper-pod -- nmap -F websvr-svc
kubectl exec -it helper-pod -- nmap -F $EXTERNAL_IP
kubectl exec -it helper-pod -- nmap -F $WebsvrPod_IP
kubectl exec -it helper-pod -- nmap -F $CLUSTER_IP
kubectl exec -it helper-pod -- curl $WebsvrPod_IP
```

```
                    > kubectl exec -it helper-pod -- nmap -F websvr-svc
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 02:52 UTC
Stats: 0:00:00 elapsed; 0 hosts completed (0 up), 1 undergoing Ping Scan
Ping Scan Timing: About 100.00% done; ETC: 02:52 (0:00:00 remaining)
Nmap scan report for websvr-svc (10.0.43.186)
Host is up (0.000037s latency).
rDNS record for 10.0.43.186: websvr-svc.student.svc.cluster.local
Not shown: 99 filtered ports
PORT   STATE  SERVICE
80/tcp closed http

                    > kubectl exec -it helper-pod -- nmap -F $EXTERNAL_IP
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 02:53 UTC
Nmap scan report for websvr-svc.student.svc.cluster.local (52.154.210.193)
Host is up (0.000032s latency).
Not shown: 99 filtered ports
PORT   STATE  SERVICE
80/tcp closed http

                    > kubectl exec -it helper-pod -- nmap -F $WebsvrPod_IP
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 02:53 UTC
Nmap scan report for 10-244-0-24.websvr-svc.student.svc.cluster.local (10.244.0.24)
Host is up (0.000014s latency).
Not shown: 99 closed ports
PORT   STATE SERVICE
80/tcp open  http

                    > kubectl exec -it helper-pod -- nmap -F $CLUSTER_IP
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 02:53 UTC
Nmap scan report for websvr-svc.student.svc.cluster.local (10.0.43.186)
Host is up (0.000033s latency).
Not shown: 99 filtered ports
PORT   STATE  SERVICE
80/tcp closed http

                    > kubectl exec -it helper-pod -- curl $WebsvrPod_IP
...
Hello from Websvr
...
```

In cases of External and Service (Cluster IP), we see **port 80 is in closed state** and refuses to accept incoming connections. However, websvr-pod has port 80 open.

Since curl on the Pod IP port 80 works, this confirms Web Server is running and Pod configuration is valid.

Step 5: Run network capture from Node running Web Server Pod and Helper Pod.

Get below which will be needed for tcpdump

```
echo $Helper_Pod_IP
10.244.0.25 # this is only an example.
echo $WebsvrPod_IP
10.244.0.24 # this is only an example.
echo $Node_Name
aks-nodepool1-96643998-vmss000000 # this is only an example.
```

1. Open a new terminal. Running below creates a debug Pod on Node running Web Server Pod. Install tcpdump on the debug Pod as shown below. Aside from Node-Name above, you can also use 'kubectl get nodes'

```
kubectl debug node/<Node-Name> -it --image=mcr.microsoft.com/dotnet/runtime-deps:6.0

# Install below
apt-get update -y; apt-get install tcpdump -y
```

2. From Step 1, run tcpdump using SRC IP (Helper pod in Step 3) and DST IP (Web server pod in Step 1)

```
Order is important. Src=Helper-Pod and Dst=Websvr-Pod
tcpdump -en -i any src <HELPER_POD_IP> and dst <WEBSVR_POD_IP>
```

```
root@aks-nodepool1-96643998-vmss000000:/# tcpdump -en -i any src 10.244.0.25 and dst 10.244.0.24
tcpdump: data link type LINUX_SLL2
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
```

3. On the original terminal. Generate traffic to capture output by running below from Helper pod terminal.

```
kubectl exec -it helper-pod -- curl websvr-svc
```

```
> kubectl exec -it helper-pod -- curl websvr-svc
curl: (7) Failed to connect to websvr-svc port 80: Connection refused
command terminated with exit code 7
```

Based on the trace provided, the **helper-pod's** IP address establishes a connection with **websvr-svc**, utilizing the *Cluster IP* address. This subsequently redirects the connection to the IP address of Websvr. However, the connection is made using port number 8080.

```
03:03:53.501596 caliea9cf7acc82 Out ifindex 28 ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 80: 10.244.0.25.32952 > 10.244.0.24.8080:
tions [mss 1460,sackOK,TS val 344069866 ecr 0,nop,wscale 7], length 0
```

Looking at the Web Server Pod YAML, it has **containerPort** set to 8080.

```
spec:
  containers:
  - image: nginx:latest
    imagePullPolicy: Always
    name: websvr
    ports:
    - containerPort: 8080
      protocol: TCP
```

Service definition for **service/websvr-svc** has **targetPort** also set to 8080.

```
  ports:
  - nodePort: 30703
    port: 80
    protocol: TCP
    targetPort: 8080
```

From port scan earlier, we found that the Websvr application was listening on port 80. This means the Websvr-svc target port configuration is incorrect and the configuration of the Websvr container Port does not make any difference.

Step 6: Fixing the issue.

1. To fix this issue, we must update websvr-svc targetPort and set it to 80.

Use '**`kubectl edit svc websvr-svc`**' to update targetPort to 80.

Before

```
ports:
- nodePort: 32003
  port: 80
  protocol: TCP
  targetPort: 8080
```

After

```
> kubectl describe svc websvr-svc
Name:                     websvr-svc
Namespace:                student
Labels:                   app=websvr
Annotations:              <none>
Selector:                 app=websvr
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.0.43.186
IPs:                      10.0.43.186
LoadBalancer Ingress:     52.154.210.193
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
```

Use 'kubectl get ep' to confirm its pointing to port 80 on web server Pod.

2. Once done, the web server should be reachable over the External IP as well as the internal service Cluster IP.

```
kubectl exec -it helper-pod -- nmap -F websvr-svc
kubectl exec -it helper-pod -- nmap -F $EXTERNAL_IP
kubectl exec -it helper-pod -- nmap -F $CLUSTER_IP
```

```
> kubectl exec -it helper-pod -- nmap -F websvr-svc
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 03:10 UTC
Nmap scan report for websvr-svc (10.0.43.186)
Host is up (0.000036s latency).
rDNS record for 10.0.43.186: websvr-svc.student.svc.cluster.local
Not shown: 99 filtered ports
PORT   STATE SERVICE
80/tcp open  http

> kubectl exec -it helper-pod -- nmap -F $EXTERNAL_IP
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 03:10 UTC
Nmap scan report for websvr-svc.student.svc.cluster.local (52.154.210.193)
Host is up (0.000085s latency).
Not shown: 99 filtered ports
PORT   STATE SERVICE
80/tcp open  http

> kubectl exec -it helper-pod -- nmap -F $CLUSTER_IP
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-30 03:11 UTC
Nmap scan report for websvr-svc.student.svc.cluster.local (10.0.43.186)
Host is up (0.000036s latency).
Not shown: 99 filtered ports
PORT   STATE SERVICE
80/tcp open  http
```

```
kubectl exec -it helper-pod -- curl websvr-svc
kubectl exec -it helper-pod -- curl $EXTERNAL_IP
```

```
> kubectl exec -it helper-pod -- curl websvr-svc
...
Hello from Websvr
...
> kubectl exec -it helper-pod -- curl $EXTERNAL_IP
...
Hello from Websvr
```

The traffic flow between the external client and the application pods is managed by the targetPort and containerPort fields of the service definition.

The **targetPort** is the port used by the Service to route traffic to the Application pods. The **containerPort** is the port on which the application listens in the pods.

When a client sends a request to the external IP address of the LoadBalancer service, the request is directed to the targetPort on the pods.
The service routes the traffic to the correct pod based on the rules defined in the service's selector. The traffic then reaches the container listening on the containerPort and is processed by the application.
In this way, the targetPort and containerPort fields ensure that external traffic is correctly mapped to the correct pods and processed by the correct container within the pod.

Step 7: Challenge

In the above example, image nginx only listens on port 80. Test this web server image tadeugr/aks-mgmt, which listens on 80 and 8080 using Pod sample as shown. Hence if you edit Service, targetPort to 80 and 8080 both ports should work.

```
Run
kubectl delete pod/websvr service/websvr-svc
cd Lab4; .\working2.ps1

# below should work now, even though container and targetPort=8080, since app listens on 8080 and 80
kubectl exec -it helper-pod -- curl websvr-svc

# confirmed that app listens on both 80/8080 by below.
kubectl get pod -o wide # get pod-ip
kubectl exec -it helper-pod -- curl <pod-ip>:8080
kubectl exec -it helper-pod -- curl <pod-ip>:80

# what this new image looks like in working2.ps1
---
apiVersion: v1
kind: Pod
metadata:
  name: websvr
  labels:
    app: websvr
spec:
  containers:
  - name: websvr
    image: tadeugr/aks-mgmt
      command: ["/bin/bash", "-c"]
      args: ["/start.sh; tail -f /dev/null"]
    ports:
    - containerPort: 8080
---
```

Step 8: Cleanup

```
kubectl delete pod/helper-pod pod/websvr service/websvr-svc pod/node-debugger-aks-systempool-24510098-vmss000003-lqrvr
```

Lab 4: Enable AKS Monitoring and Logging

Objective: Enable Container Insights to view container performance and enable Container Diagnostics to view logging.

Step 1: Set up the environment.

1. Setup up AKS as outlined in this section.

2. Create and switch to the newly created namespace.

```
kubectl create ns student
kubectl config set-context --current --namespace=student
# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
```

3. Confirm Container Insights has been set up. This was setup during AKS cluster creation in Lab setup section. From AKS blade in portal > Monitor > Insights, confirm metrics collection.

Step 2: Deploy and Monitor apps that spike CPU/Memory utilization

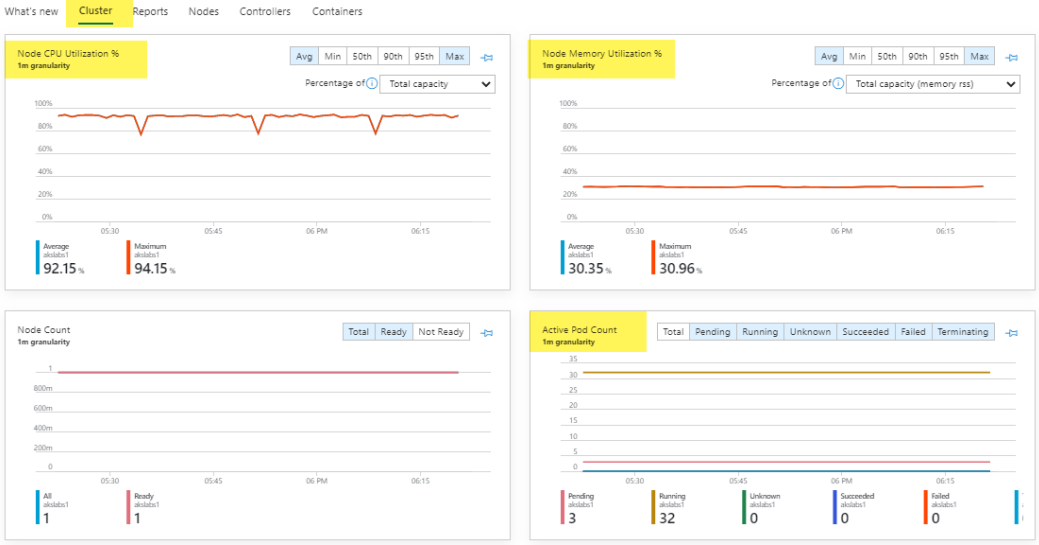
1. Assuming namespace 'student' still exists, deploy below to turn on the CPU and Memory load

```
$kubectl_apply = @"
---
# deployment to generate high cpu
apiVersion: apps/v1
kind: Deployment
metadata:
    name: openssl-loop
    namespace: student
spec:
    replicas: 3
    selector:
      matchLabels:
        app: openssl-loop
    template:
      metadata:
        labels:
          app: openssl-loop
      spec:
        containers:
        - args:
          - |
            while true; do
              openssl speed >/dev/null;
            done
          command:
          - /bin/bash
          - -c
          image: polinux/stress
          name: openssl-loop
---
# deployment to generate high memory
apiVersion: apps/v1
kind: Deployment
metadata:
    name: stress-memory
    namespace: student
spec:
    replicas: 3
    selector:
      matchLabels:
        app: stress-memory
```

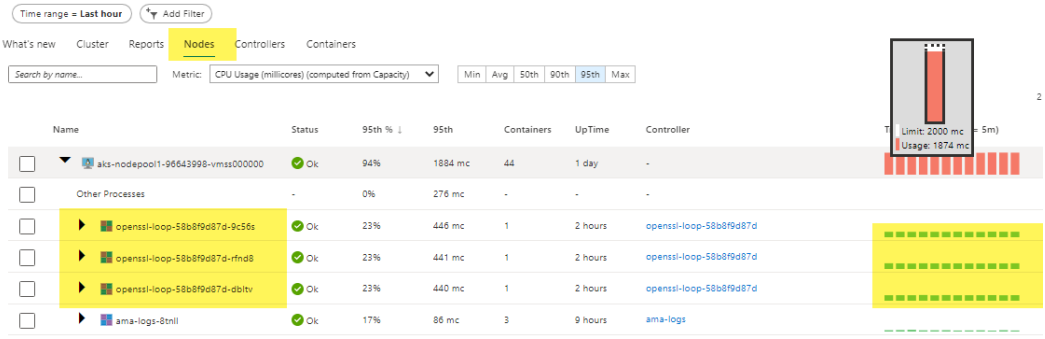
```
    template:
      metadata:
        labels:
          app: stress-memory
      spec:
        containers:
        - image: polinux/stress
          name: stress-memory-container
          resources:
            requests:
              memory: 50Mi
            limits:
              memory: 50Mi
          command: ["stress"]
          args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
---
"@
$kubectl_apply | kubectl apply -f -
```

'kubectl get pods' should have stress-memory pods in 'CrashLoopBackOff' and empty-loop pods in 'Pending'.

2. From Insights tab, validate the CPU/Memory consumption

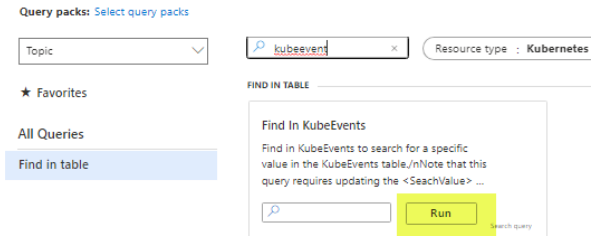


From Nodes tab, see if the top consuming Pods match those deployed.

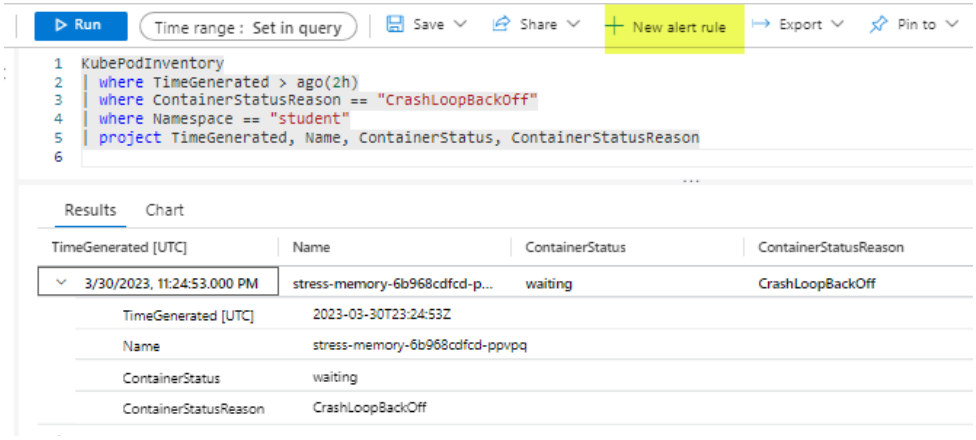

Time range = Last hour
Add Filter
What's new
Cluster
Reports
Nodes
Controllers
Containers
Search by name...
Metric:
CPU Usage (millicores) (computed from Capacity)
Min
Avg
50th
90th
95th
Max
Limit: 2000 mc
Usage: 1874 mc
Name
Status
95th %
95th
Containers
UpTime
Controller
aks-nodepool1-96643998-vmss000000
Ok
94%
1884 mc
44
1 day
-
Other Processes
0%
276 mc
openssl-loop-58b8f9d87d-9c56s
23%
446 mc
1
2 hours
openssl-loop-58b8f9d87d
openssl-loop-58b8f9d87d-rfnd8
441 mc
openssl-loop-58b8f9d87d-dbltv
440 mc
ama-logs-8tnll
17%
86 mc
3
9 hours
ama-logs

Step 3: View container logs and generate an alert resulting in email

1. From Logs, search and select KubeEvents and run the below query to get the Pod results.




```
KubePodInventory
| where TimeGenerated > ago(2h)
| where ContainerStatusReason == "CrashLoopBackOff"
| where Namespace == "student"
| project TimeGenerated, Name, ContainerStatus, ContainerStatusReason
```

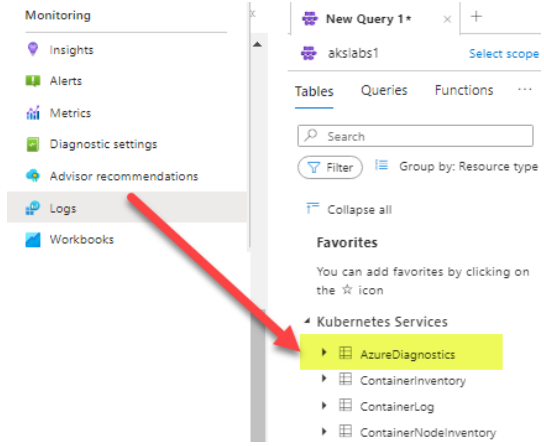


2. Create an alert as highlighted above. Confirm Email has been received on next alert.
   - Set threshold to 0.
   - In 'Actions' create an Action group with Email ID if it doesn't exist.
   - Set Alert rule name and create Alert.

3. Confirm email receipt on next occurrence of the threshold.

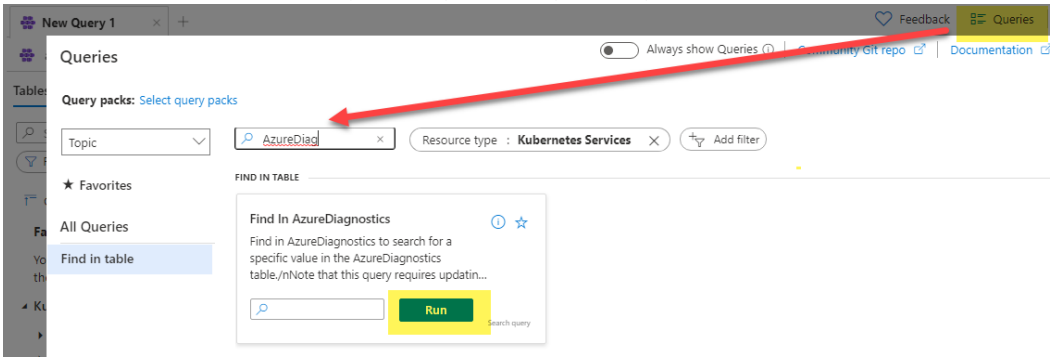
Step 4: Search Diagnostics logs

1. Ensure AzureDiagnostics section is seen in Logs. If available, run below commands to Create and Delete objects. This should generate additional log data.




```
k create ns test-diag
k create deploy deploy-diag-alert --image busybox -n test-diag
k delete deploy deploy-diag-alert -n test-diag
```

Queries section should lead to the Query finder to get AzureDiagnostics logs if it exists.



3. Run the below queries to view log data. Log details are found in log_s. Using parse_json() you can drill down to display content of embedded fields, objects, or arrays.

```
AzureDiagnostics
| where Category contains "kube-audit"
| extend log=parse_json(log_s)
| extend verb=log.verb
| extend resource=log.objectRef.resource
| extend ns=log.objectRef.namespace
| extend name=log.objectRef.name
| where resource == "pods"
| where ns=="test-diag"
| project TimeGenerated, verb, resource, name, log_s
```

```
AzureDiagnostics
| where Category contains "kube-audit"
| extend log=parse_json(log_s)
| extend verb=log.verb
| extend resource=log.objectRef.resource
| extend ns=log.objectRef.namespace
| extend name=log.objectRef.name
| where resource == "pods"
| where ns=="test-alert"
| project TimeGenerated, verb, resource, name, log_s
```

Results | Chart

TimeGenerated [UTC]	verb	resource	name	log_s
3/30/2023, 6:47:19.166 PM	patch	pods	deploy-alert-7ff8f5657b-dksrx	{"kind":"Event","apiVersion":"audit.k8s.io/v1","level".

TimeGenerated [UTC]	2023-03-30T18:47:19.1668024Z
verb	patch
resource	pods
name	deploy-alert-7ff8f5657b-dksrx
log_s	{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Request","auditID":"3787f01a-ce22-4c6c-875b-70bcf56ab403","stage":"ResponseComplete","requestURI":"/a
annotations	{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":""}
apiVersion	audit.k8s.io/v1
auditID	3787f01a-ce22-4c6c-875b-70bcf56ab403
kind	Event
level	Request
objectRef	{"resource":"pods","namespace":"test-alert","name":"deploy-alert-7ff8f5657b-dksrx","apiVersion":"v1","subresource":"status"}
requestObject	{"metadata":{"uid":"003af057-b07b-4625-b26f-7fc9ba5411cd"},"status":{"containerStatuses":[{"containerID":"containerd://8c07c0bc7d75c5bde6221e736
requestReceivedTimestamp	2023-03-30T18:47:19.153501Z
requestURI	/api/v1/namespaces/test-alert/pods/deploy-alert-7ff8f5657b-dksrx/status
responseStatus	{"metadata":{},"code":200}

To get graphical view, run below. This gets line chart of all the created pods in ns 'test-diag'

```
AzureDiagnostics
| where Category contains "kube-audit"
| extend log=parse_json(log_s)
| extend verb=log.verb
| extend resource=log.objectRef.resource
| extend name=log.objectRef.name
| extend ns=log.objectRef.namespace
| where resource == "pods"
| where verb=="create"
| where ns=="test-diag"
| summarize count() by bin(TimeGenerated, 1m), tostring(name), tostring(verb)
| render timechart
```

Step 5: Challenge

Repeat labs 1 to 5 and use the Logs section above to query and analyze the logs.

Step 6: Final cleanup

```
az group delete -n <aksrg> -y
```

Lab 5: Using Linux toolset to analyze failed application

Objective: There's a working application and a non-working application running on the cluster. For the faulty app curl fails with time out. This lab will use tools on the Linux node hosting the application to diagnose the issue.

Prerequisites:

1. An available AKS cluster.

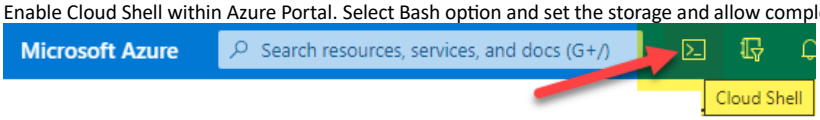
Step 1: Set up the environment.

1. Setup up AKS as outlined in this section.

2. Create and switch to the newly created namespace.

```
kubectl create ns student
kubectl config set-context --current --namespace=student

# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
```

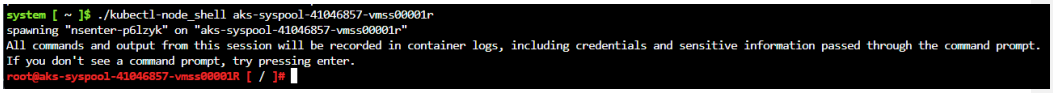
3. Enable Cloud Shell within Azure Portal. Select Bash option and set the storage and allow completion.



From AKS blade > Overview > Connect, run the '`az account..`' and '`az aks get-credentials..`' commands in the Cloud Shell. Use `kubectl get nodes` to verify it works.

4. Download **kubectl-node-shell** using below steps. When executed it creates an **Nsenter** pod, which will have the advanced privileges to run iptables. This level of access is not available with debug pods to connect to Nodes.

```
curl -LO https://github.com/kvaps/kubectl-node-shell/raw/master/kubectl-node_shell
chmod +x ./kubectl-node_shell
sudo mv ./kubectl-node_shell /usr/local/bin/kubectl-node_shell

$ ./kubectl-node_shell <node-name from above>
```



5. Scripts setup the deployment with 3 Pod replicas, and service of type Loadbalancer running on port 4000. They are both identical applications, except for the image.

```
# Working
# ---
apiVersion: v1
kind: Service
metadata:
  name: working-app-clusterip
spec:
  type: LoadBalancer
  ports:
  - port: 4000
    protocol: TCP
    targetPort: 4000
```

```
  selector:
    app: working-app
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: working-app-deployment
  labels:
    app: working-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: working-app
  template:
    metadata:
      labels:
        app: working-app
    spec:
      containers:
      - name: working-app
        image: jvargh/nodejs-app:working
        ports:
        - containerPort: 4000
---
```

 # Faulty

```
# ---
apiVersion: v1
kind: Service
metadata:
  name: faulty-app-clusterip
spec:
  type: LoadBalancer
  ports:
  - port: 4000
    protocol: TCP
    targetPort: 4000
  selector:
    app: faulty-app
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: faulty-app-deployment
  labels:
    app: faulty-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: faulty-app
  template:
    metadata:
      labels:
        app: faulty-app
    spec:
      containers:
      - name: faulty-app
        image: jvargh/nodejs-app:faulty
        ports:
        - containerPort: 4000
---
```

3. Make a note of the Cluster and External IPs associated with Faulty and Working apps.

```
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)
faulty-app-clusterip    LoadBalancer   10.0.189.236   20.121.181.241   4000:31738/TCP
kubernetes              ClusterIP      10.0.0.1       <none>           443/TCP
working-app-clusterip   LoadBalancer   10.0.81.248    20.81.101.33     4000:31369/TCP
```

```
$ curl -m 5 20.81.101.33:4000
Hello World - AKS Triage and Troubleshooting Labs
$ curl -m 5 20.121.181.241:4000
curl: (28) Connection timed out after 5000 milliseconds
```

4. Create pod for validation within Pod

```
kubectl run test-pod --image=nginx --port=80 --restart=Never
```

5. Allow Inbound access through Custom NSG

```
$custom_aks_nsg="custom_aks_nsg" # <- verify
$nsg_list=az network nsg list --query "[?contains(name,'$custom_aks_nsg')].{
ResourceGroup:resourceGroup}" --output json

# Extract NSG Resource Group
$resource_group=$(echo $nsg_list | jq -r '.[].ResourceGroup')
echo $nsg_list, $nsg_name, $resource_group

az network nsg rule create --name AllowHTTPInbound `
       --resource-group $resource_group --nsg-name $custom_aks_nsg `
       --destination-port-range * --destination-address-prefix * `
       --source-address-prefixes Internet --protocol tcp `
       --priority 100 --access allow
```

6. Validation Test

1. Test internal access within cluster

```
kubectl exec -it test-pod -- curl working-app-clusterip:4000  # works
kubectl exec -it test-pod -- curl faulty-app-clusterip:4000   # fails with Connection refused
```

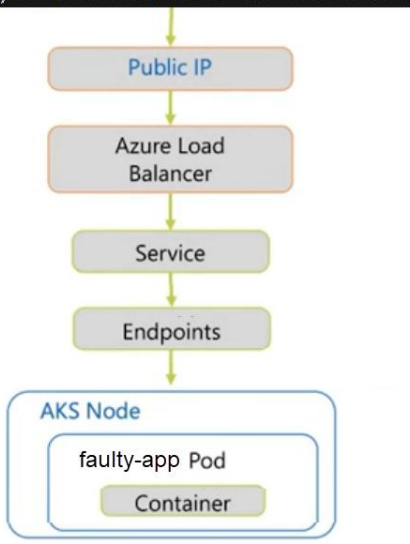
7. Test external access to cluster

```
curl <Working-App-External-IP>:4000  # works
curl <Faulty-App-External-IP>:4000  # fails with `Unable to connect to the remote server`
```

Step 2: Walk through the Kubernetes view

Use this step to confirm that from Internet through to the Pod, Kubernetes setup is configured correctly.

```
$ curl -m 5 20.121.181.241:4000
curl: (28) Connection timed out after 5000 milliseconds
```



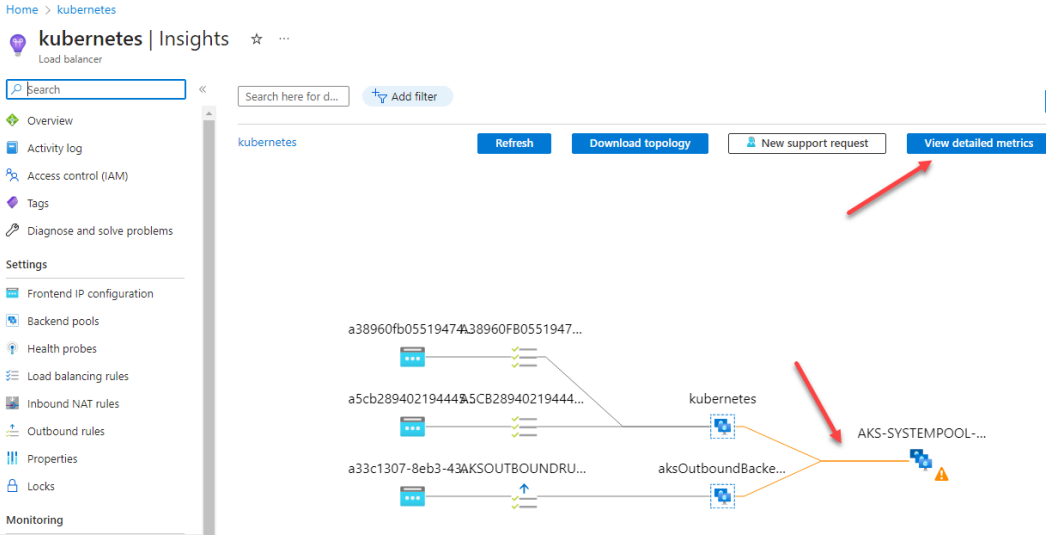
1. Curl request hits the Load balancer Public IP assigned to the service. Service IP get added as a Front End IP rule to the existing AKS Loadbalancer.

2. Service ties into the endpoints by forwarding requests to the pods, and in turn to the Application container.

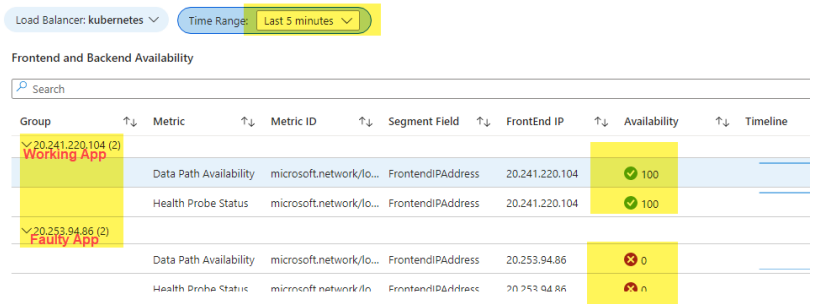
```
$ k get svc,ep
NAME                              TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)
service/faulty-app-clusterip      LoadBalancer   10.0.189.236   20.121.181.241   4000:31738/TCP
service/kubernetes                ClusterIP      10.0.0.1       <none>           443/TCP
service/working-app-clusterip     LoadBalancer   10.0.81.248    20.81.101.33     4000:31369/TCP

NAME                                ENDPOINTS                                                  AGE
endpoints/faulty-app-clusterip      10.244.0.57:4000,10.244.0.58:4000,10.244.0.59:4000          4h31m
endpoints/kubernetes                52.226.53.16:443                                           2d
endpoints/working-app-clusterip     10.244.0.54:4000,10.244.0.55:4000,10.244.0.56:4000          4h32m
```

Step 3: Verify Loadbalancer Insights and Metrics

From LoadBalancer blade, go to AKS LB > Insights. Ensure the Loadbalancer is functional and capturing metrics. Can see from below there's an issue with the backend pool.



From **Detailed metrics** > '**Frontend and Backend Availability**' section, you should see the **Failing app FE IP** is Red for Availability but Working app FE IP is Green for Availability. Change 'Time Range' to 5m.



Step 4: Perform Network trace to the Faulty app

Use this step to confirm if the Faulty app is even listening. We should see Working app responding but Faulty app does not.

IP addresses listed below applies to this example, for reference only. Replace with your own

```
test-pod-ip =        #10.244.0.61
working-app-svc =    #10.0.81.248
faulty-app-svc =     #10.0.189.236
working-pod-IPs =    #10.244.0.54, 55, 56
faulty-pod-IPs =     #10.244.0.57, 58, 59
```

1. Get the test-pod IP and destination service IP and run tcpdump on the associated Node of the Pod

2. Working app provides trace

`kubectl exec -it test-pod – curl <working-app-svc>:4000`

3. From Cloud Shell in Azure Portal, run below command . Get node from '`kubectl get pods -o wide`'.

`kubectl-node_shell <Node associated with pods>`

4. Setup trace from test-pod and the Pod network.

`tcpdump -en -i any src <test-pod IP> and dst net 10.244.0.0/16`

5. From another terminal, execute curl to Faulty and Working app's service.

`kubectl exec -it test-pod -- curl <faulty-app-svc>:4000`

`kubectl exec -it test-pod -- curl <working-app-svc>:4000`

Working App

```
> k exec -it test-pod -- curl 10.0.81.248:4000
Hello World - AKS Triage and Troubleshooting Labs
```

```
root@aks-systempool-24510098-vmss000002:/# tcpdump -en -i any src 10.244.0.61 and dst net 10.244.0.0/16
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
^C00:02:07.865742 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 76: 10.244.0.61.53498 > 10.244.0.56.4000: 0,nop,wscale 7], l
ength 0
00:02:07.865786 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 68: 10.244.0.61.53498 > 10.244.0.56.4000: F, length 0
00:02:07.865848 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 148: 10.244.0.61.53498 > 10.244.0.56.4000: 52], length 80
00:02:07.867464 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 68: 10.244.0.61.53498 > 10.244.0.56.4000: Fh 0
00:02:07.867711 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 68: 10.244.0.61.53498 > 10.244.0.56.4000: F3], length 0
00:02:07.867893 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 68: 10.244.0.61.53498 > 10.244.0.56.4000: Fh 0
```

Faulty App

```
> k exec -it test-pod -- curl 10.0.189.236:4000
curl: (7) Failed to connect to 10.0.189.236 port 4000: Connection refused
command terminated with exit code 7
```

```
root@aks-systempool-24510098-vmss000002:/# tcpdump -en -i any src 10.244.0.61 and dst net 10.244.0.0/16
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
^C00:02:58.594377 Out ee:ee:ee:ee:ee:ee ethertype IPv4 (0x0800), length 76: 10.244.0.61.52818 > 10.244.0.58.4000: le
ngth 0
```

From trace above, there's only response from Working App pod. No response from Faulty App pod.

Step 5: Advanced tcpdump

This section captures to file, copy from nsenter pod to local desktop where Wireshark will visualize the trace. Need two consoles.

1. From Cloud Shell run **`kubectl-node_shell <Node associated with pods>`**. Run below commands.

```
cd /tmp
tcpdump -nn -s0 -vvv -i any -w capture.cap
```

where,

```
-nn: display IP addresses and port numbers in numeric
-s0: set snapshot=0 i.e., capture entire packet
-l: output asap without buffering
-vvv: max verbosity
```

2. From 2nd console run below to view the HTML output

```
kubectl exec -it <test-pod> -- curl <working-app-pod>
```

3. On Cloud Shell, break the tcpdump (CTRL+c) and capture.cap should be written to /tmp

4. From 2nd console use below command to download capture.cap. Use '`k get pod`' to get nsenter pod name.

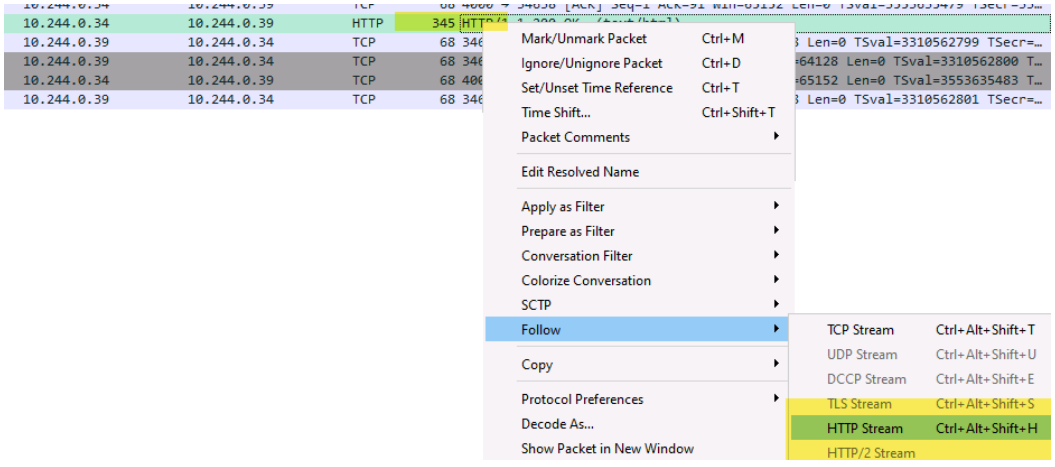
```
kubectl cp <nsenter-pod>:/tmp/capture.cap capture.cap
```

Wireshark will need to be installed for next step. See link
https://2.na.dl.wireshark.org/win64/Wireshark-win64-4.0.4.exe

5. Open capture.cap in Wireshark. Use below filter to refine view.

```
ip.addr == <test-pod> # might not need this "and ip.addr == <working-app-pod>"
```

6. Use Analyze > Follow > HTTP Stream to view the HTTP flow as seen below



Wireshark · Follow HTTP Stream (tcp.stream eq 105) · capture.cap

```
GET / HTTP/1.1
Host: working-app-clusterip:4000
User-Agent: curl/7.74.0
Accept: */*

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 49
ETag: W/"31-Hfanw0m0tg112gOoNhc2XwSwRz4"
Date: Mon, 10 Apr 2023 18:49:49 GMT
Connection: keep-alive
Keep-Alive: timeout=5

Hello World - AKS Triage and Troubleshooting Labs
```

7. For long running traces that need to be saved to storage account, use utility below. Helm install creates storage account and daemon set creates tcpdump Pods on all nodes, that continuously writes capture to storage account.

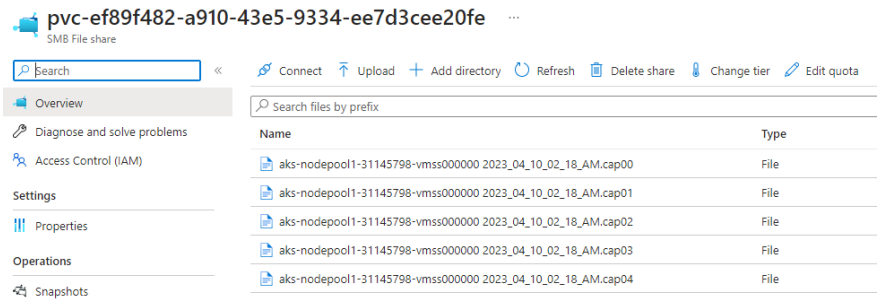
https://github.com/amjadaljunaidi/tcpdump/blob/main/README.md

Uninstall Helm chart to stop tracing and capture will be left intact in storage account.

8. To just focus on one node than all nodes, as above, use Lab5 > tcpdump-pod.yaml. Change node name and use below command. Storage account > file share should have tcpdump contents.

```
kubectl apply -f tcpdump-pod.yaml
```

View from Storage



View from Pod

```
$ k get pod
NAME          READY   STATUS    RESTARTS   AGE
tcpdump-pod   1/1     Running   0          4m29s
$ k exec -it tcpdump-pod -- bash
root@aks-nodepool1-31145798-vmss000000:/#
root@aks-nodepool1-31145798-vmss000000:/# cd /root/tcpdump/
root@aks-nodepool1-31145798-vmss000000:~/tcpdump# ls -lh
total 2.8G
-rwxrwxrwx 1 root root 477M Apr 10 02:19 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap00'
-rwxrwxrwx 1 root root 477M Apr 10 02:19 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap01'
-rwxrwxrwx 1 root root 477M Apr 10 02:20 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap02'
-rwxrwxrwx 1 root root 477M Apr 10 02:21 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap03'
-rwxrwxrwx 1 root root 477M Apr 10 02:22 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap04'
-rwxrwxrwx 1 root root 477M Apr 10 02:22 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap05'
-rwxrwxrwx 1 root root 9.1M Apr 10 02:22 'aks-nodepool1-31145798-vmss000000 2023_04_10_02_18_AM.cap06'
```

On completion delete using “kubectl delete -f tcpdump-pod.yaml”. Delete storage account to delete file share.

Step 6: Walk through the Linux Kernel view

Use this step to confirm that from Linux Kernel level everything is configured correctly, allowing packets to flow. Also, it is not a Firewall issue since we have Working-App Pods able to be called from the Internet.

1. Run below command. This provides higher level privileges on the Node.

```
kubectl get pods -o wide # gives node name to use below
kubectl-node_shell <Node associated with pods>
```

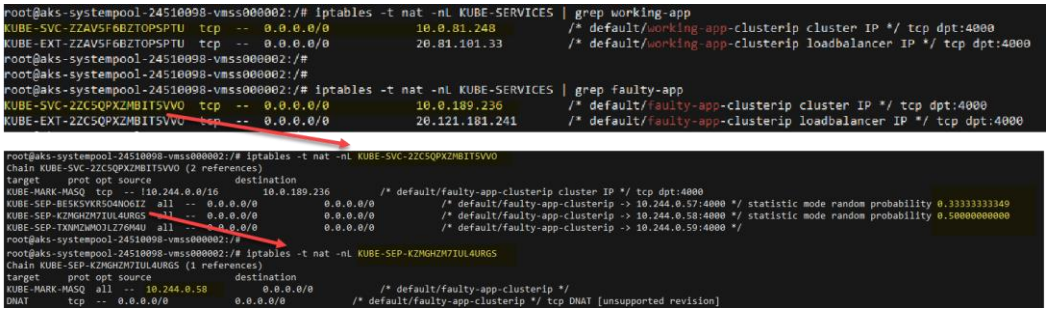
IPTables has a chain structure. Managed by kube-proxy pods in cluster.

2. View faulty app's iptable NAT table and show the KUBE-SERVICES chain, using below command to show the Services Internal and External IPs.

```
iptables -t nat -nL KUBE-SERVICES | grep faulty-app
```

3. Walk down the chain by using below command below, which gives the Endpoints for the Service. Also gives selection probability of the Endpoint. Running this again gives on an Endpoint, gives the Pod IP associated with the Endpoint.

```
iptables -t nat -nL <kube-service id>
```

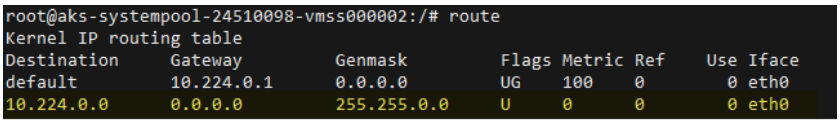


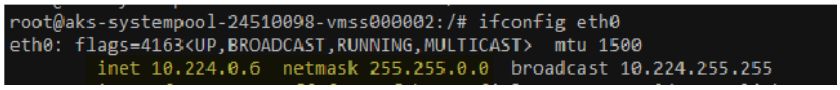
In **KUBE-SERVICES chain**, if SRC=0.0.0.0/0 or ANY and Protocol=TCP then forward from KUBE-SERVICES to KUBE-SVC chain, as the next hop for incoming packet. KUBE-SVC represents the Service's Cluster IP.

In **KUBE-SVC chain**, if SRC=0.0.0.0/0 or ANY and Protocol=TCP then forward from KUBE-SVC to KUBE-SEP chain, as the next hop for incoming packet. KUBE-SEP represents the ENDPOINT. Notice there are 3 rules for 3 Pod's. Pod1 gets 1/3 traffic, Pod2 gets 1/2 traffic and rest goes to Pod3. This could affect latency due to statistical load balancing based on probability, especially around multi-zone balancing where Pod's are distributed across zones and latency caused by hops.

In **KUBE-SEP chain**, if IP=<Pod-IP> then direct incoming packet to the designated Pod.

4. Validate route associated with Pod network and its eth0 interface. This route should map to AKS route table for Kubenet.





This should validate the route in the route table for the Kubenet networking associated with the AKS.

Routes

Name	Address prefix	Next hop type	Next hop IP address
aks-systempool-24510098...	10.244.0.0/24	Virtual appliance	10.224.0.6

3. Walk through the running containers using below command

```
crictl ps | grep faulty
```

```
root@aks-systempool-24510098-vmss000002:/# crictl ps
CONTAINER           IMAGE               CREATED             STATE    NAME        POD ID          POD
c0440b868631f       3544b275fea0f       5 hours ago         Running  faulty-app  49f8265a50e4c   faulty-app-deployment-7678db5f44-9bcz2
602dc9199a9bb       3544b275fea0f       5 hours ago         Running  faulty-app  159422144e810   faulty-app-deployment-7678db5f44-qktnv
2958d8e828c79       3544b275fea0f       5 hours ago         Running  faulty-app  0bd5650d2fb26   faulty-app-deployment-7678db5f44-4dmlz
```

Map this to '`kubectl get pods -o wide | grep faulty`' to match Pod names.

4. From 'kubectl get pod' to get one of the faulty pod names to use here to grep. Use the obtained **Container ID** of one of the faulty app containers to return the **Process ID**.

```
crictl inspect --output go-template --template '{{.info.pid}}' <container_id>
```

```
root@aks-systempool-24510098-vmss000002:/# crictl ps | grep faulty-app-deployment-7678db5f44-4dmlz
2958d8e828c79       3544b275fea0f       5 hours ago         Running             faulty-app
root@aks-systempool-24510098-vmss000002:/# crictl inspect --output go-template --template '{{.info.pid}}' 2958d8e828c79
151227
```

5. Use the **Process ID** to enter the Pods' Network namespace using command **nsenter**. This allows us to execute commands into the Pod namespace. In this case, command 'ip address show' displays Pod IP. Running 'k get pods' confirms from the IP that we're on the right pod.

```
root@aks-systempool-24510098-vmss000002:/# nsenter -t 151227 -n ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if61: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 0a:c5:5f:0f:27:a9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.0.57/32 scope global eth0
       valid_lft forever preferred_lft forever
```

```
$ k get pods -A -o wide | grep faulty-app
default             faulty-app-deployment-7678db5f44-4dmlz    1/1     Running   10.244.0.57   a
default             faulty-app-deployment-7678db5f44-9bcz2    1/1     Running   10.244.0.59   a
default             faulty-app-deployment-7678db5f44-qktnv    1/1     Running   10.244.0.58   a
```

Step 7: Confirm if App is listening.

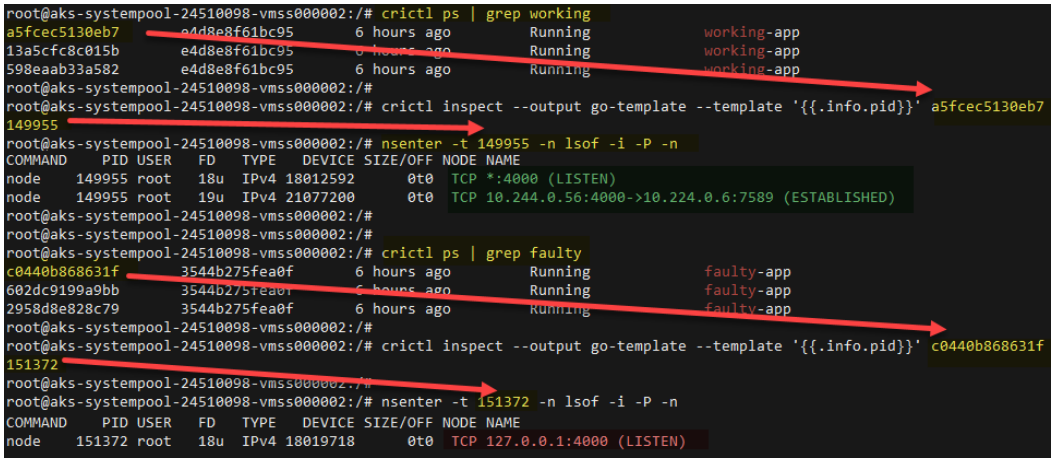
This step uses lsof (List Open Files) utility using following parameters:

- The *-i parameter* is used to display information about network connections.
- The *-P parameter* is used to prevent the conversion of port numbers to port names. When used with the -i parameter, it will display the port number instead of the name.
- The *-n parameter* is used to prevent the conversion of network addresses to hostnames. When used with the -i parameter, it will display the IP address instead of the hostname.

Command to use: `nsenter -t <Process ID_Working or Faulty container> -n lsof -i -P -n`

From below, working container is listening on ANY IPs i.e., *:4000.

Faulty container is tied to local loopback or 127.0.0.1 instead of ANY as above.


root@aks-systempool-24510098-vmss000002:/# crictl ps | grep working
a5fcec5130eb7 e4d8e8f61bc95 6 hours ago Running working-app
13a5cfc8c015b e4d8e8f61bc95 6 hours ago Running working-app
598eaab33a582 e4d8e8f61bc95 6 hours ago Running working-app
root@aks-systempool-24510098-vmss000002:/#
root@aks-systempool-24510098-vmss000002:/# crictl inspect --output go-template --template '{{.info.pid}}' a5fcec5130eb7
149955
root@aks-systempool-24510098-vmss000002:/# nsenter -t 149955 -n lsof -i -P -n
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
node 149955 root 18u IPv4 18012592 0t0 TCP *:4000 (LISTEN)
node 149955 root 19u IPv4 21077200 0t0 TCP 10.244.0.56:4000->10.224.0.6:7589 (ESTABLISHED)
root@aks-systempool-24510098-vmss000002:/#
root@aks-systempool-24510098-vmss000002:/#
root@aks-systempool-24510098-vmss000002:/# crictl ps | grep faulty
c0440b868631f 3544b275fea0f 6 hours ago Running faulty-app
602dc9199a9bb 3544b275fea0f 6 hours ago Running faulty-app
2958d8e828c79 3544b275fea0f 6 hours ago Running faulty-app
root@aks-systempool-24510098-vmss000002:/#
root@aks-systempool-24510098-vmss000002:/# crictl inspect --output go-template --template '{{.info.pid}}' c0440b868631f
151372
root@aks-systempool-24510098-vmss000002:/#
root@aks-systempool-24510098-vmss000002:/# nsenter -t 151372 -n lsof -i -P -n
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
node 151372 root 18u IPv4 18019718 0t0 TCP 127.0.0.1:4000 (LISTEN)

Step 8: Fixing the issue

Issue was in the Docker file where working app was set to bind to 0.0.0.0 or default/Any address, but faulty app was set to bind to a fixed loopback 127.0.0.1 address, as seen below.

Working

docker-app > ! Dockerfile-working

```
FROM node:16

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json A
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 4000
CMD ["node", "server.js", "4000", "0.0.0.0"]
```

docker-app > JS server.js > ...

```
'use strict';

const express = require('express');

// Constants
const PORT = process.argv[2];
const HOST = process.argv[3];
// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World - AKS Triage and Troubleshooting Labs');
});

app.listen(PORT, HOST, () => {
  console.log(`Running on http://${HOST}:${PORT}`);
});
```

Faulty

docker-app > ! Dockerfile-faulty

```
FROM node:16

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AN
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 4000
CMD ["node", "server.js", "4000", "127.0.0.1"]
```

Step 9: Challenge

From docker-app folder fix the Dockerfile for Faulty app, create a new image, and create new Pod using this image to check if it resolves issue.

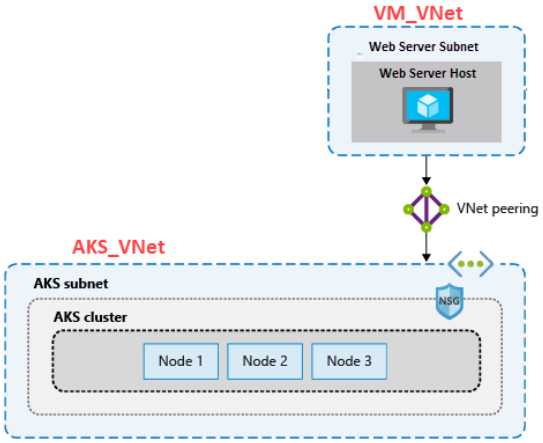
Step 10: Cleanup

```
k delete ns student
az network nsg rule delete --name AllowHTTPInbound `
       --resource-group $resource_group --nsg-name $nsg_name
```

Lab 6: Endpoint Connectivity issues across Virtual Networks

Objective: The goal of this exercise is to troubleshoot a scenario where pods fail to connect to endpoints in other virtual networks and to resolve connectivity issues.

Layout: 2 VNets exist. One for AKS and the other for VM Linux host. Private Endpoint joins the 2 VNets.



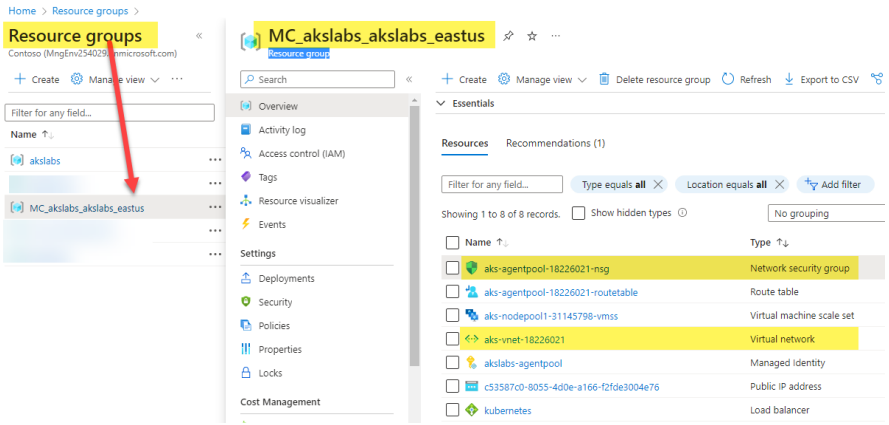
Step 1: Set up the environment

1. Setup up AKS as outlined in this section.
2. Create and switch to the newly created namespace

```
kubectl create ns student
kubectl config set-context --current --namespace=student
# Verify current namespace
kubectl config view --minify --output 'jsonpath={..namespace}'
```

3. Change directory to Lab3 > cd Lab3
4. Install VM, VM_VNet and Nginx. Create NSG and attach to VM Subnet

Values required for the below variables can be found in AKS MC Resource Group as seen in below figure.




```
# Don't replace. Use as-is
$resource_group="MC_akslabs_akslabs_eastus" # <- replace only if different from your environment
$vm_vnet="vmlabs-vnet"
$vm_name="vmlabs"
$vm_nsg="vmlabs_nsg"


# Create VNet for WebServer VM
az network vnet create --name $vm_vnet --resource-group $resource_group --address-prefixes 10.0.0.0/16 --subnet-name default --subnet-prefix 10.0.0.0/24

# Create VM in the VNet
az vm create --resource-group $resource_group --name $vm_name --image UbuntuLTS --vnet-name $vm_vnet --subnet default --public-ip-address `""` --size Standard_DS1_v2 --admin-username azureuser --generate-ssh-keys --os-disk-delete-option "Delete" --nic-delete-option "Delete" --data-disk-delete-option "Delete"

# Install nginx on the VM and open port 80 for incoming traffic using the below cmd
az vm extension set --publisher Microsoft.Azure.Extensions --version 2.1 --name CustomScript --vm-name $vm_name --resource-group $resource_group --settings "{'commandToExecute':'sudo apt-get update -y && sudo apt-get install -y nginx'}"

# Install the CustomScript extension on the VM and runs below cmd to install nginx.
az vm open-port --port 80 --resource-group $resource_group --name $vm_name

# Create NSG
az network nsg create -g $resource_group -n $vm_nsg

# Get VM subnet from VNet
$subnet_name=$(az network vnet subnet list -g $resource_group --vnet-name $vm_vnet --query "[0].name")

# Attach NSG to VM subnet
az network vnet subnet update -g $resource_group --vnet-name $vm_vnet -n $subnet_name --network-security-group $vm_nsg
```

3. Create VNet Peering between AKS_VNet and VM_VNet

```
# Insert here the AKS and VM VNet names, VM name and Resource Group name, AKS NSG name
$aks_vnet="aks-vnet-18226021"  # <- replace with one from your environment
$aks_nsg="aks-agentpool-18226021-nsg" # <- replace with one from your environment
$vm_ip=$(az vm show -g $resource_group -n $vm_name --show-details --query 'privateIps' -o tsv)
```

```
echo $vm_ip
```

 # Get the IDs for AKS_VNet

```
$aks_vnet_id=$(az network vnet show -g $resource_group -n $aks_vnet --query id --out tsv)
echo $aks_vnet_id
```

 # Get the IDs for VM VNet

```
$vm_vnet_id=$(az network vnet show -g $resource_group -n $vm_vnet --query id --out tsv)
echo $vm_vnet_id
```

 # Create both halves of the VNet Peer

```
az network vnet peering create -n peerAKStoVM -g $resource_group --vnet-name $aks_vnet --remote-vnet
$vm_vnet_id --allow-forwarded-traffic --allow-vnet-access

az network vnet peering create -n peerVMtoAKS -g $resource_group --vnet-name $vm_vnet --remote-vnet
$aks_vnet_id --allow-forwarded-traffic --allow-vnet-access
```

Check status of the 2 halves of VNet Peer

```
az network vnet peering show --name peerAKStoVM -g $resource_group --vnet-name $aks_vnet
az network vnet peering show --name peerVMtoAKS -g $resource_group --vnet-name $vm_vnet
```

 # Confirm "allowForwardedTraffic": true, "allowVirtualNetworkAccess": true from below

```
az network vnet peering show --name peerAKStoVM -g $resource_group --vnet-name $aks_vnet --query
'{AllowForwardedTraffic:allowForwardedTraffic, AllowVirtualNetworkAccess:allowVirtualNetworkAccess}'
```

Step 2: Setup Test pod and verify that the VM web server is accessible

```
# Create test-pod
kubectl run test-pod --image=nginx --port=80 --restart=Never
kubectl exec -it test-pod -- bash
# Run below on test-pod bash shell
apt-get update -y
apt-get install ping -y
exit

# End-to-End test: Curl should return HTML page
kubectl exec -it test-pod -- curl -m 5 $vm_ip
```

Step 3: Break Networking

From Lab3, run broken1.ps1

```
cd Lab3; .\broken1.ps1
```

Step 4: Troubleshoot connectivity issue

1. Assume the VM Web Server is functional and web application is running, there's no need to SSH and validate.
- If testing is needed options are, create public VM in same VNet and curl test.
- Bastion VM is another option or Associate a newly created Public IP with VM instance

2. Validate peering is setup right: Steps to setting up VNet peering

3. Check curl connectivity which should result in connection timeout after 5s

```
kubectl exec -it test-pod -- curl -m 5 $vm_ip
```

4. Check if NSG on the AKS or VM subnets have any DENY rules that might block incoming/outgoing traffic. Check link on custom network security group blocking traffic.

```
az network nsg rule list -g $resource_group --nsg-name $aks_nsg
az network nsg rule list -g $resource_group --nsg-name $vm_nsg

# From VM NSG, below should return "access": "Deny"
az network nsg rule list -g $resource_group --nsg-name $vm_nsg --query "[?destinationPortRange=='80']" --query "[0].{access:access}"
```

5. Check if Peering setup is Connected and up from both ends. Check link on peering in same subscription.

```
# Specifically check the "allowForwardedTraffic" and "allowGatewayTransit" values are enabled.
az network vnet peering show --name peerAKStoVM -g $resource_group --vnet-name $aks_vnet
az network vnet peering show --name peerVMtoAKS -g $resource_group --vnet-name $vm_vnet

# Below should return false
az network vnet peering show --name peerVMtoAKS -g $resource_group --vnet-name $vm_vnet --query "{allowForwardedTraffic:allowForwardedTraffic,allowGatewayTransit:allowGatewayTransit}"
```

Step 5: Restore service

1. Enable peering on VM VNet

```
az network vnet peering update -n peerVMtoAKS -g $resource_group --vnet-name $vm_vnet --set allowForwardedTraffic=true allowVirtualNetworkAccess=true
```

2. Check curl connectivity which should result in connection timeout after 5s. Issue persists.

```
kubectl exec -it test-pod -- curl -m 5 $vm_ip
```

3. Remove NSG rule to Allow web traffic from VM Web Server on port 80.

```
az network nsg rule delete -n DenyPort80Inbound -g $resource_group --nsg-name $vm_nsg
```

Step 6: Validate connectivity

After 60s, check curl connectivity which should return HTML page from Web Server on hosted VM.

```
kubectl exec -it test-pod -- curl -m 5 $vm_ip
kubectl exec -it test-pod -- ping $vm_ip
```

Step 7: What was in the broken files

Broken1.ps1 was used to

1. Create a rule on VM NSG to deny access to HTTP traffic
2. Disable critical peering parameters i.e., Block Traffic Forwarding and Block remote VNet access

```
$resource_group="MC_akslabs_akslabs_eastus" # <- replace with AKS Resource Group from your environment
$vm_vnet="vmlabs-vnet"
$vm_nsg="vmlabs_nsg"

# Use VM NSG to block Inbound access on port 80
az network nsg rule create -n DenyPort80Inbound `
    -g $resource_group --nsg-name $vm_nsg --destination-port-range 80 `
    --destination-address-prefix * --priority 100 --access deny

# Block peering access on VM end
# Disable peering from VM Vnet
az network vnet peering update -n peerVMtoAKS `
    -g $resource_group --vnet-name $vm_vnet `
    --set allowForwardedTraffic=false allowVirtualNetworkAccess=false
```

Step 8: Cleanup

```
kubectl delete pod/test-pod
```

```
az network vnet peering delete -n peerAKStoVM -g $resource_group --vnet-name $aks_vnet
az network vnet peering delete -n peerVMtoAKS -g $resource_group --vnet-name $vm_vnet

# DETACH VM NSG from VM subnet and DELETE NSG
az network vnet subnet update -n $subnet_name -g $resource_group --vnet-name $vm_vnet -n $subnet_name
--network-security-group '""'
az network nsg delete -g $resource_group -n $vm_nsg

# Delete VM, NSG and it VNet
az vm delete --resource-group $resource_group --name $vm_name -y
az network vnet delete --name $vm_vnet --resource-group $resource_group
az network nsg delete -g $resource_group -n $vm_name”NSG”

k delete ns student
```