cisco Live!
Amsterdam | February 5-9, 2024

LABDCN-2016

Kubernetes Networking with ACI and Calico

Filip Wardzichowski – Software Engineering Technical Leader

Miłosz Sabadach – Software Engineer

Learning Objectives or Table of Contents

Upon completion of this lab, you will be able to:

- Install Kubernetes with Calico CNI (Container Network Interface)
- Establish BGP Peering between each Kubernetes node running Calico router and ACI L3out using Floating SVI (Switch Virtual Interface)
- Understand routing and traffic flow between Kubernetes Pods, Services and how to expose applications externally

Disclaimer

This training document is to familiarize with Cisco ACI best practices how to connect Kubernetes Platform with Calico CNI. Although the lab design and configuration examples could be used as a reference, it's not a real design, thus not all recommended features are used, or enabled optimally. For the design related questions please contact your representative at Cisco, or a Cisco partner.

Scenario

With the increasing adoption of container technologies and Cisco® Application Centric Infrastructure (Cisco ACI®) as a pervasive data-center fabric technology, it is only natural to see customers trying to integrate these two solutions.

A key design choice every Kubernetes administrator must make when deploying a new cluster is selecting a network plugin. The network plugin is responsible for providing network connectivity, IP address management, and security policies to containerized workloads.

While Cisco offers a CNI (container network interface) plugin directly compatible and integrated with Cisco ACI, in this lab we will integrate with Cisco ACI with Project Calico https://projectcalico.docs.tigera.io/about/about-calico

Calico

Calico supports two main network modes: direct container routing (no overlay transport protocol) or network overlay using VXLAN or IPinIP (default) encapsulations to exchange traffic between workloads.

The direct routing approach means the underlying network is aware of the IP addresses used by workloads.

Conversely, the overlay network approach means the underlying physical network is not aware of the workloads' IP addresses. In that mode, the physical network only needs to provide IP connectivity between K8s nodes while container to container communications is handled by the Calico network plugin directly. This, however, comes at the cost of additional performance overhead as well as complexity in interconnecting your container-based workloads with external non-containerized workloads.

When the underlying networking fabric is aware of the workloads' IP addresses, an overlay is not necessary. The fabric can directly route traffic between workloads inside and outside of the cluster as well as allowing direct access to the services running on the cluster. This is the preferred Calico mode of deployment when running on premises. This guide details the recommended ACI configuration when deploying Calico in direct routing mode.

K8s routing architecture

In our lab each Kubernetes node acts as a router for all the endpoints (containers) that are hosted on it. We call that function a vRouter. The data path is provided by the Linux kernel, the control plane by a BGP protocol server, and management plane by the specific CNI plugin implementation.

Each endpoint can only communicate through its local vRouter. Each node has assigned chunk of of the pod subnet (for example, pod network is 10.2.0.0/16, each pod will get assigned subset from this pool, like 10.2.1.64/26). Each vRouter announces pod chunk subnet it is responsible for, to all the other vRouters and other routers on the infrastructure fabric using BGP, usually with BGP route reflectors to increase scale.

After taking into consideration the characteristics and capabilities of ACI and Calico, Cisco's current recommendation is to implement an alternative design where a single AS is

allocated through the whole Kubernetes cluster, and iBGP Full Node-To-Node Mesh is disabled for the cluster.

AS Per Cluster design – overview

In this design, a dedicated Cisco ACI L3Out is created for the entire Kubernetes cluster. This removes control-plane and data-plane overhead on the K8s cluster, thus providing improved performance and enhanced visibility to the workloads.

Each Kubernetes node has the same AS number and peers via eBGP with a pair of ACI Top-of-Rack (ToR) switches configured in a vPC pair. Having a vPC pair of leaf switches provides redundancy within the rack.

This eBGP design does not require running any route reflector nor full mesh peering (iBGP) in the K8s infrastructure; this results in a more scalable, simpler, and easier to maintain architecture.

In order to remove the need of running iBGP in the cluster the ACI BGP configuration requires the following features:

- AS override – The AS override function replaces the AS number from the originating router with the AS number of the sending BGP router in the AS Path of outbound routes.
- Disable Peer AS Check – Disables the peer autonomous number check.

An added benefit of configuring all the Kubernetes nodes with the same AS is that it allows us to use BGP Dynamic Neighbors by using the Cluster Subnet as BGP neighbors, greatly reducing the ACI config complexity.

Note: You need to ensure Kubernetes nodes are configured to peer only with the Border Leaves Switches in the same RACK.

Once this design is implemented, the following connectivity is expected:

- Pods running on the Kubernetes cluster can be directly accessed from inside ACI or outside through transit routing.
- Pod-to-pod and node-to-node connectivity happens over the same L3Out and external end point group (EPG).
- Exposed Services can be directly accessed from inside or outside ACI. Those services are load- balanced by ACI through ECMP 64-way provided by BGP.

Figure 1 shows an example of such a design with two racks and a six-node BareMetal K8s cluster:

- Each rack has a pair of ToR leaf switches and three Kubernetes nodes.
- ACI uses AS 65001 for its leaf switches.
- The six Kubernetes nodes are allocated AS 64512:

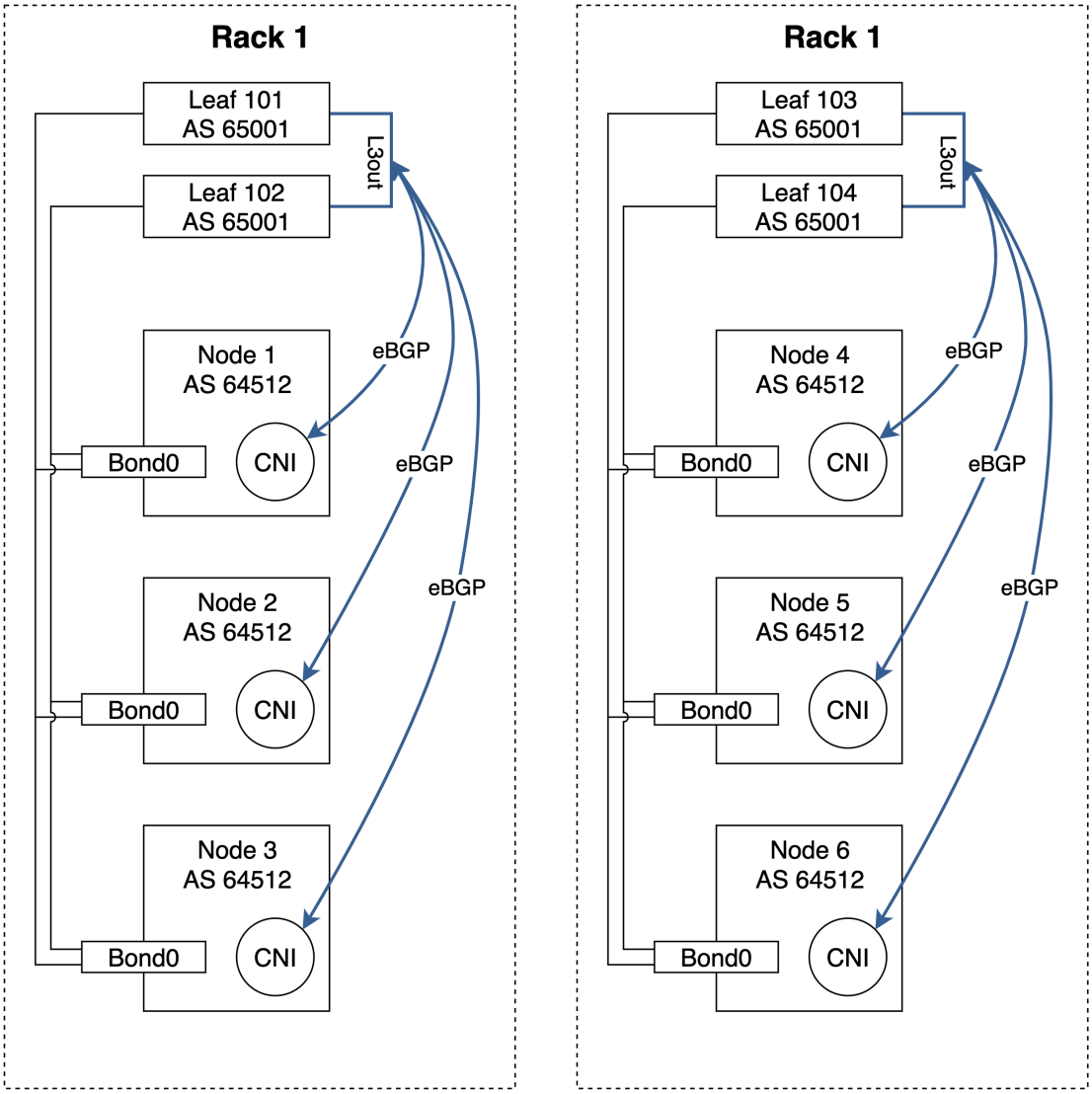


Figure 1 AS Per Cluster design

Physical connectivity

The physical connectivity is provided by a virtual Port-Channel (vPC) configured on the ACI leaf switches toward the Kubernetes nodes. The vPC is pre-configured and shared between all the pods in this lab.

Each student will use acc-provision tool to provision configuration of a dedicated L3Out to run eBGP with the vRouters in each Kubernetes node through the vPC port-channel.

Floating SVI design

The floating SVI feature enables you to configure an L3Out without specifying logical interfaces. The feature saves you from having to configure multiple L3Out logical interfaces to maintain routing when Virtual Machines (VMs) move from one host to another. Floating SVI is supported for VMware vSphere Distributed Switch (VDS) as of ACI Release 4.2(1) and on physical domains as of ACI Release 5.0(1). It is recommended to use the physical domains approach for the following reasons:

- Can support any hypervisor
- Can support mixed mode clusters (VMs and bare metal)

Using floating SVI also allows to have Layer-2 communications between the routing nodes across different leaf switches (located in different racks); this allows the design to use:

- A single node subnet for the whole Kubernetes cluster
- A single encapsulation (VLAN) for the whole cluster

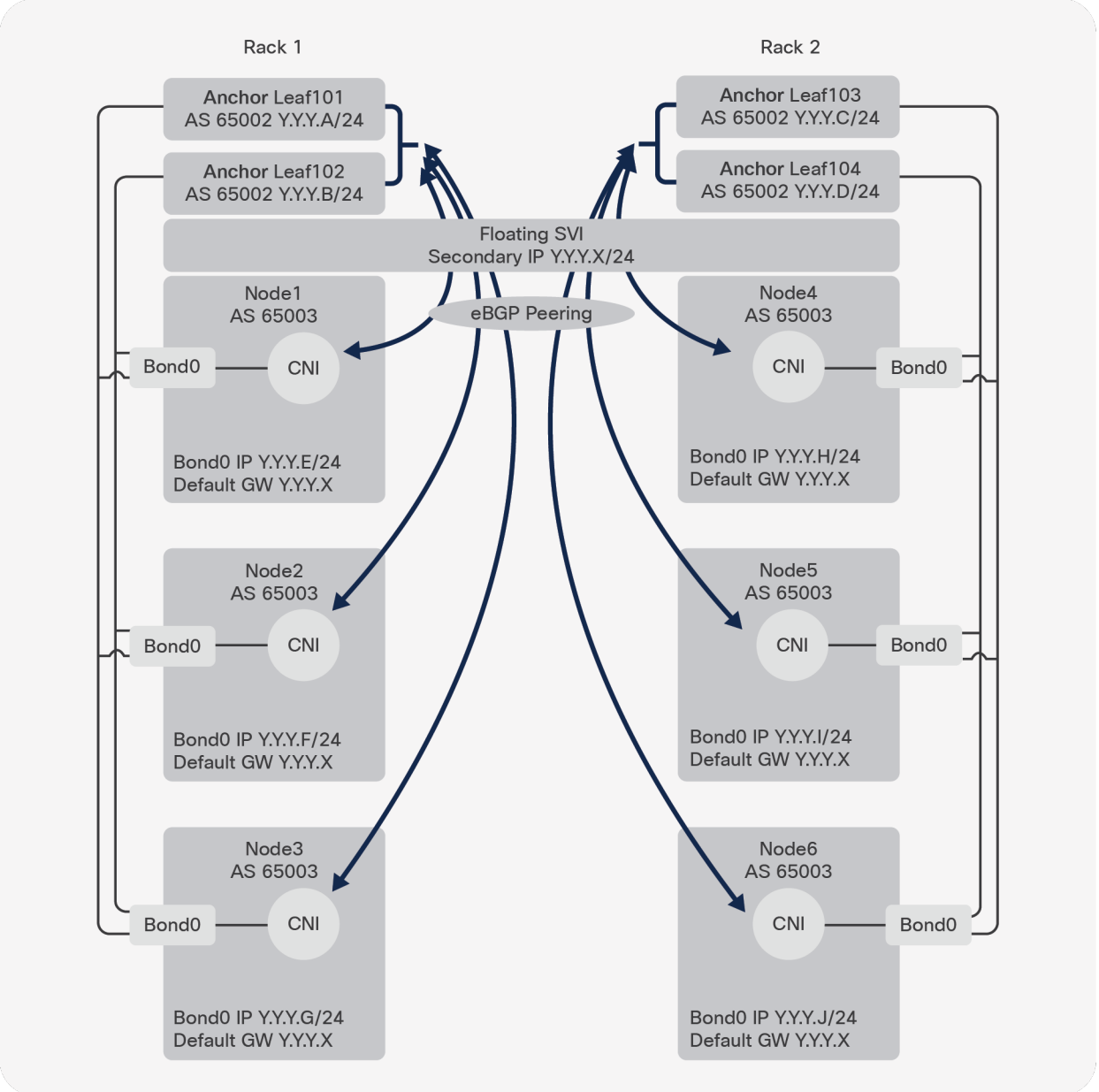


Figure 2 Floating SVI design

A strong recommendation for this design is to peer the Kubernetes node with the local (same rack) anchor nodes. This is needed to limit traffic tromboning because, currently, a compute leaf (in the example below, Leaf105) will install, as next hop for the routes received from the Kubernetes nodes, the TEP IP address of the anchor nodes where the eBGP peering is established. This leads to suboptimal traffic flow, as shown in Figure 3.

VM mobility is, however, still supported and can be used to perform, for example, maintenance on the hypervisor without the need to power off nodes in the K8s cluster.

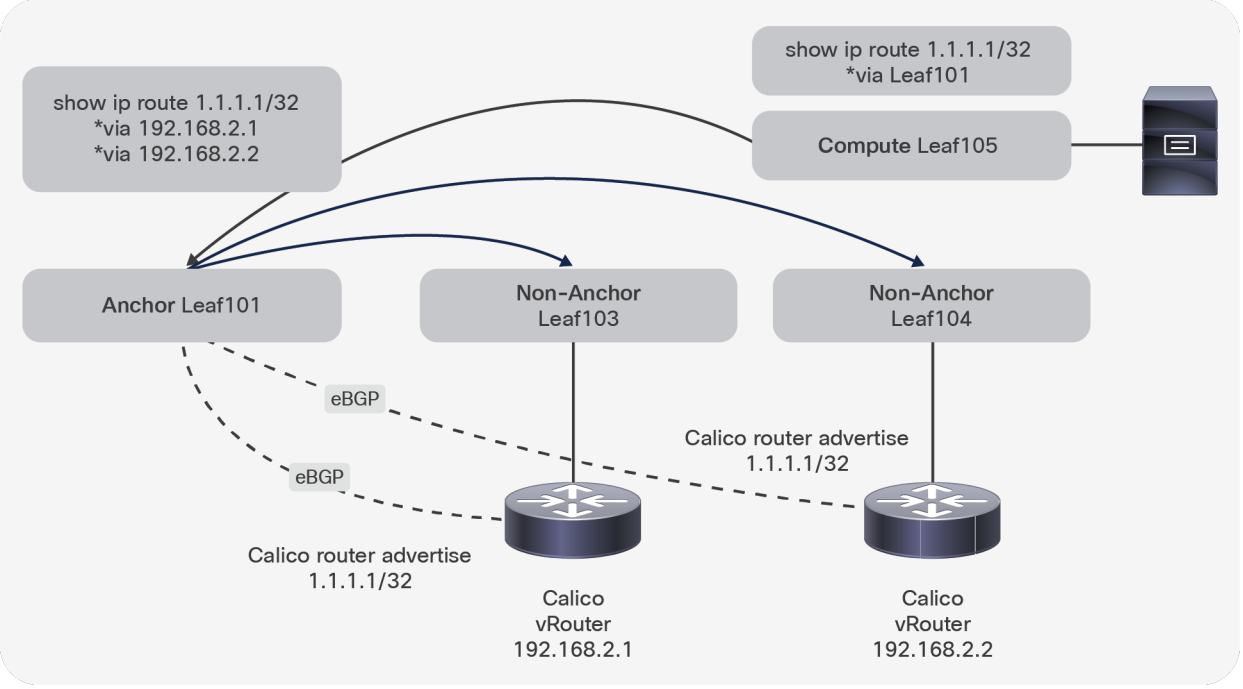


Figure 3 Tromboning with floating SVI

ACI BGP configuration

AS OVERRIDE

The AS override function replaces the AS number from the originating router with the AS number of the sending BGP router in the AS Path of outbound routes.

DISABLE PEER AS CHECK

Disables the peer autonomous number check.

BGP GRACEFUL RESTART

Both ACI and Calico are configured by default to use BGP Graceful Restart. When a BGP speaker restarts its BGP process or when the BGP process crashes, neighbors will not discard the received paths from the speaker, ensuring that connectivity is not impacted as long as the data plane is still correctly programmed.

This feature ensures that, if the BGP process on the Kubernetes node restarts (CNI Calico BGP process upgrade/crash), no traffic is impacted. In the event of an ACI switch reload, the feature is not going to provide any benefit, because Kubernetes nodes are not receiving any routes from ACI.

BGP TIMERS

The ACI BGP timers should be set to 1s/3s to match the Calico configuration.

MAX BGP ECMP PATH

By default, ACI only installs 16 eBGP/iBGP ECMP paths. This would limit spreading the load to up to 16 Kubernetes nodes. The recommendation is to increase both values to 64. Increasing the iBGP max path value to 64 also ensures that the internal MP-BGP process is installing additional paths.

BGP HARDENING

To protect the ACI against potential Kubernetes BGP misconfigurations, the following settings are recommended:

- Enabled BGP password authentication
- Set the maximum AS limit to one:
    - Per the eBGP architecture, the AS path should always be one.
- Configure BGP import route control to accept only the expected subnets from the Kubernetes cluster:
    - Pod subnet
    - Node subnet
    - Service subnet

Set a limit on the number of received prefixes from the nodes.

Lab topology:

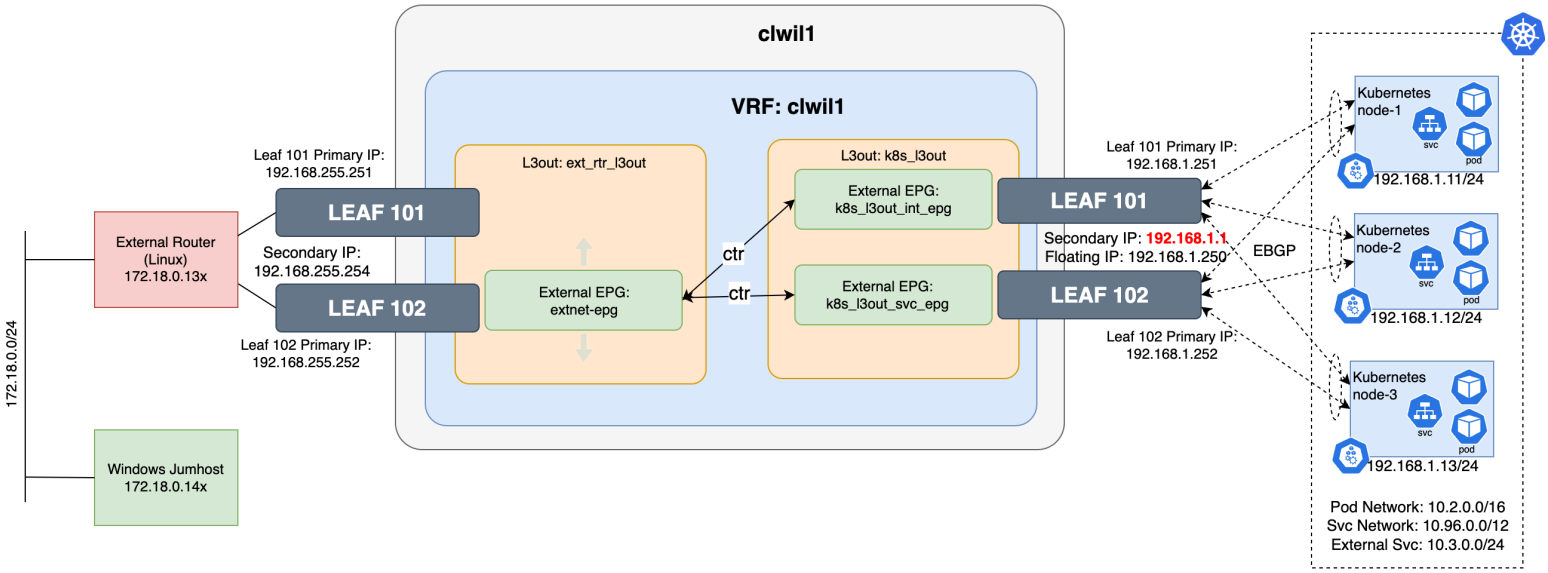


Figure 4 Lab topology

Task 1: Configure ACI L3out and generate Calico manifests

The VMs that we will be using for this lab are already deployed but are currently isolated from the network and our first step will be to configure the ACI L3OUT in our allocated tenants.

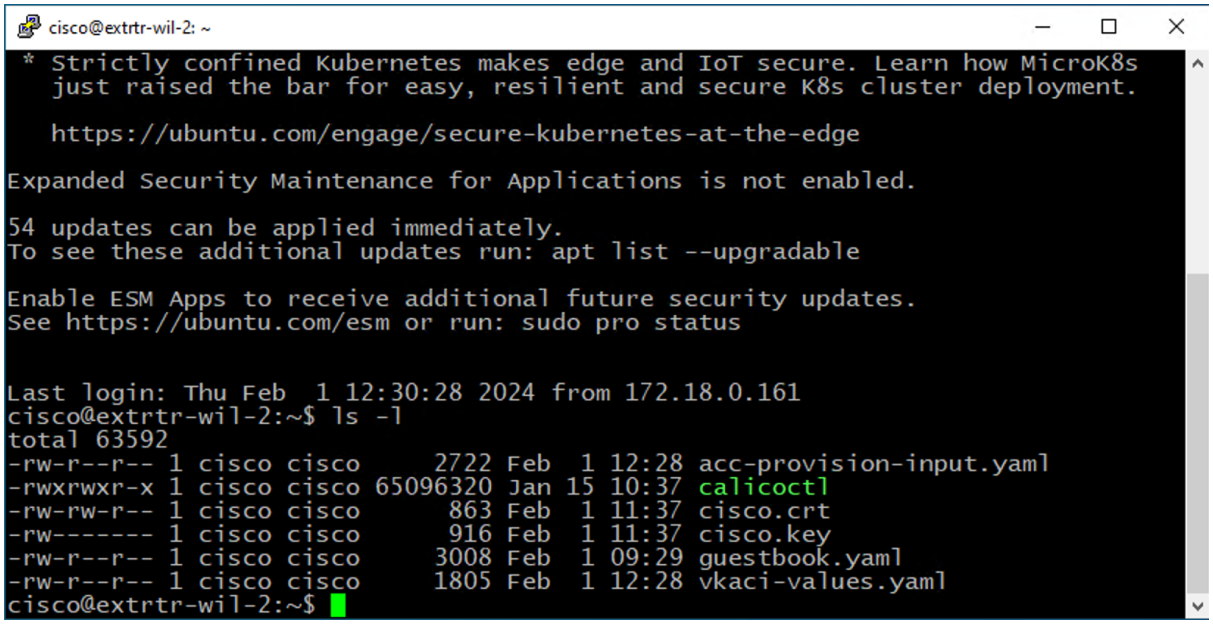
Step 1: Verify you are connected to the VPN and Windows Jump host.

In Windows Jump host open PuTTY:

Connect to **"clwilX-rtr"** machine using password:

Step 2: List all files in the folder:

```
ls -l
```



Step 3: Review prepared "*acc-provision-input.yaml*" file on the External Router node:

```
cat acc-provision-input.yaml
```

```
cisco@pod1-cl-rtr:~$ cat acc-provision-input.yaml
#
# Configuration for ACI Fabric
#
aci_config:
  apic_hosts:
    - 172.18.0.120                  # List of APIC hosts to connect for APIC API
  vrf:                              # VRF used to create all kubernetes EPs
    name: clwil1
    tenant: clwil1                     # This can be system-id or common

  cluster_l3out:
    name: k8s-l3out                # This is the l3out created by acc-provision
    aep: infra_aaep                 # Required field. The AEP for ports/VPCs used by this cluster
    svi:
      floating_ip: 192.168.1.250/24         # Required field. Should be an IP in the node subnet
      secondary_ip: 192.168.1.1/24          # Required field
      vlan_id: 250                          # Required field. Valid vlan-id for the whole cluster
      mtu: 9000                             # 9000 is the recommended value.
    bgp:
      secret: test                  # BGP password for the peerings generated by BGPPeer resource
      peering:
        prefixes: 500               # Maximum number of prefixes to limit how many prefixes we can accept
                                    # from a single K8s node. Default is 500
        remote_as_number: 64512     # By default, all Calico nodes use 64512 AS number
                aci_as_number: 65001        # Required field. This is derived frm APIC.
  l3out:
    name: extnet-l3out              # External l3Out
    external_networks:
```

```
    - extnet-epg
#
# Networks used by ACI containers
#
net_config:
  node_subnet: 192.168.1.0/24       # Subnet to use for nodes
  pod_subnet: 10.2.0.0/16           # Subnet to use for Kubernetes Pods
  extern_dynamic: 10.3.0.0/16       # Subnet to use for Service External IPs
                                    # and Service LoadBalancer IPs to be advertised over BGP.
  cluster_svc_subnet: 10.96.0.0/16  # Subnet to use for Kubernetes Service Cluster IPs
                                    # to be advertised over BGP

#
# Configuration for calico networking
#
calico_config:
  net_config: {}
topology:
  rack:                         # Each rack has a pair of ToR leaf switches and Kubernetes nodes
  - id: 1                       # "id" of a ToR, and is user assigned.
    aci_pod_id: 1
    leaf:                       # User should only provide anchor nodes in the input file
    - id: 101                   # Node id of anchor nodes
      local_ip: 192.168.1.251   # The IP address of the peer
    - id: 102
      local_ip: 192.168.1.252
    node:
    - name: node-1              # Kubernetes nodes names
    - name: node-2
    - name: node-3
```

Step 4: Run acc-provision tool that will configure L3out towards Kubernetes, and will generate manifest files that will be applied on the Kubernetes cluster to install Calico CNI.

```
acc-provision -a -u cisco -p clams24 -f calico-3.26.3 -c acc-provision-
input.yaml -z calico_manifests.tar.gz
```

```
cisco@cl-pod1-rtr:~$ acc-provision -a -u cisco -p clams24 -f calico-3.26.3 -c
acc-provision-input.yaml -z calico_manifests.tar.gz
INFO: Loading configuration from "acc-provision-input.yaml"
INFO: Using configuration flavor calico-3.26.3
INFO: Using infra_vlan from ACI: 3967
WARN: Unable to validate resources on APIC: 'aep'
INFO: Generating certs for calico based kubernetes controller
INFO:   Private key file: "user-calico-k8s-l3out.key"
INFO:   Certificate file: "user-calico-k8s-l3out.crt"
Using flavor:  calico-3.26.3
Generated the deployment tar file
INFO: cluster_l3out is under tenant: clwil2, vrf: clwil2
INFO: Provisioning configuration in APIC
WARN: User already exists (calico-k8s-l3out), recreating user
```

Step 5: Check connectivity to Kubernetes nodes:

```
cisco@cl-pod1-rtr:~$ ping -c 3 192.168.1.11
PING 192.168.1.2 (192.168.1.11) 56(84) bytes of data.
64 bytes from 192.168.1.11: icmp_seq=1 ttl=63 time=0.224 ms
64 bytes from 192.168.1.11: icmp_seq=2 ttl=63 time=0.148 ms
64 bytes from 192.168.1.11: icmp_seq=3 ttl=63 time=0.260 ms
```

Step 6: Login to first Kubernetes node:

```
ssh root@192.168.1.11
```

Step 7: Run Kubernetes installation:

```
kubeadm init --pod-network-cidr=10.2.0.0/16

I0201 14:48:10.551136    3890 version.go:256] remote version
is much newer: v1.29.1; falling back to: stable-1.28
```

```
[init] Using Kubernetes version: v1.28.6
[preflight] Running pre-flight checks
…
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each
as root:

kubeadm join 192.168.1.11:6443 --token n3ndd8.vewc3pz5053nmio3 \
      --discovery-token-ca-cert-hash
sha256:ebbe914fb98991dc9a2378a0c283711f5eedb959a801fe7e2fb28a314b968e25
```

Step 6: From above output copy and paste commands that will install kubeconfig on the ext-rtr.

Exit from the node-1 and login back to the ext-rtr

```
exit
mkdir -p $HOME/.kube
scp root@192.168.1.11:/etc/kubernetes/admin.conf ~/.kube/config
```

Step 7: Copy the kubeadm join command from the previous output

Exit from node-1 and login to node-2 and node-3 and paste it there:

```
ssh root@192.168.1.12
kubeadm join 192.168.1.11:6443 --token <token_from_kubeadm_init_output> \
      --discovery-token-ca-cert-hash sha256:
<cert_hash_from_kubeadm_init_output>
exit
ssh root@192.168.1.13
kubeadm join 192.168.1.11:6443 --token <token_from_kubeadm_init_output> \
      --discovery-token-ca-cert-hash sha256:
<cert_hash_from_kubeadm_init_output>
exit
```

Exit from the node-3 and login back to the ext-rtr

Step 8: on the Ext-rtr node, check Kubernetes status of the nodes:

```
kubectl get nodes -o wide
```

```
cisco@extrtr-wil-1:~$ kubectl get nodes -o wide
NAME     STATUS   ROLES           AGE   VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
node-1   Ready    control-plane   11h   v1.28.5   192.168.1.11   <none>        Ubuntu 22.04.3 LTS   5.15.0-91-generic   containerd://1.7.12
node-2   Ready    <none>          11h   v1.28.5   192.168.1.12   <none>        Ubuntu 22.04.3 LTS   5.15.0-91-generic   containerd://1.7.12
node-3   Ready    <none>          11h   v1.28.5   192.168.1.13   <none>        Ubuntu 22.04.3 LTS   5.15.0-91-generic   containerd://1.7.12
```

Check status of the Pods running in Kubernetes

```
kubectl get pods -A -o wide
```

```
cisco@cl-pod3-rtr:~$ kubectl get pods -A -o wide
NAMESPACE     NAME                                READY   STATUS              RESTARTS   AGE     IP              NODE     NOMINATED NODE   READINESS GATES
kube-system   coredns-6d4b75cb6d-hmqd7            0/1     ContainerCreating   0          4m49s   <none>          node-1   <none>           <none>
kube-system   coredns-6d4b75cb6d-hzfms            0/1     ContainerCreating   0          4m49s   <none>          node-1   <none>           <none>
kube-system   etcd-node-1                         1/1     Running             0          5m4s    192.168.1.2     node-1   <none>           <none>
kube-system   kube-apiserver-node-1               1/1     Running             0          5m4s    192.168.1.2     node-1   <none>           <none>
kube-system   kube-controller-manager-node-1      1/1     Running             0          5m4s    192.168.1.2     node-1   <none>           <none>
kube-system   kube-proxy-9tjdc                    1/1     Running             0          4m31s   192.168.1.4     node-3   <none>           <none>
kube-system   kube-proxy-bmn8t                    1/1     Running             0          4m31s   192.168.1.3     node-2   <none>           <none>
kube-system   kube-proxy-jq7k6                    1/1     Running             0          4m49s   192.168.1.2     node-1   <none>           <none>
kube-system   kube-scheduler-node-1               1/1     Running             0          5m6s    192.168.1.2     node-1   <none>           <none>
```

Note: Kubernetes cluster is running without Container Network Interface (CNI), hence only system related Pods that are bounded to node network (192.168.1.0/24) are in the running state. Core DNS pods are waiting for CNI to be up, to get the IP address assigned from the Pod network (10.2.0.0/16).

Step 9: Unarchive calico_manifests.tar.gz. It consists of 3 files.

```
tar -zxvf calico_manifests.tar.gz
```

Step 11: Check Calico manifests

```
cisco@extrtr-wil-1:~$ ll
total 65180
-rw-r--r-- 1 cisco cisco     2722 Feb  1 17:36 acc-provision-input.yaml
-rwxrwxr-x 1 cisco cisco 65096320 Jan 15 10:37 calicoctl*
-rw-rw-r-- 1 cisco cisco   124330 Feb  1 17:38 calico_manifests.tar.gz
-rw-rw-r-- 1 cisco cisco      863 Feb  1 11:37 cisco.crt
-rw------- 1 cisco cisco      916 Feb  1 11:37 cisco.key
-rw-rw-r-- 1 cisco cisco     6874 Feb  1 17:38 custom_resources_aci_calico.yaml
-rw-rw-r-- 1 cisco cisco      971 Feb  1 17:38 custom_resources_calicoctl.yaml
-rw-r--r-- 1 cisco cisco     3035 Feb  1 17:02 guestbook.yaml
-rw-rw-r-- 1 cisco cisco  1475575 Feb  1 17:38 tigera_operator.yaml
-rw-rw-r-- 1 cisco cisco      745 Feb  1 17:38 user-calico-k8s-l3out.crt
-rw-rw-r-- 1 cisco cisco      916 Feb  1 17:38 user-calico-k8s-l3out.key
-rw-r--r-- 1 cisco cisco     1805 Feb  1 17:36 vkaci-values.yaml
```

You should see 3 new files:

Tigera_operator.yaml consist of necessary Service Accounts, Roles and Deployment of the Tigera Operator. This operator exposes Customer Resource Definitions, to manage Calico CNI lifecycle and it's configuration in a declarative way. Tigera Operator has a builtin controller, that constantly compares actual state with the desired state of various resources.

Custom_resources_aci_calico.yaml consists of the "installation" manifests that describes cluster Pod network, and disables overlay.

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  # Configures Calico networking.
  calicoNetwork:
    # Note: The ipPools section cannot be modified post-install.
    ipPools:
    - blockSize: 26
      cidr: 10.2.0.0/16
      encapsulation: None
      natOutgoing: Disabled
      nodeSelector: all()
```

Other manifests are related with installation of "calicoctl" pod commandline tool, secrets, necessary roles and service accounts.

Custom_resources_calicoctl.yaml – consists of the BGPConfiguration and BGPPeer definitions to establish peering with Cisco ACI.

Step 10: Apply Tigera Operator manifest and validate status:

```
kubectl create -f tigera_operator.yaml
```

```
cisco@extrtr-wil-1:~$ kubectl create -f tigera_operator.yaml
namespace/tigera-operator created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgpfilters.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/apiservers.operator.tigera.io created
customresourcedefinition.apiextensions.k8s.io/imagesets.operator.tigera.io created
customresourcedefinition.apiextensions.k8s.io/installations.operator.tigera.io created
customresourcedefinition.apiextensions.k8s.io/tigerastatuses.operator.tigera.io created
serviceaccount/tigera-operator created
clusterrole.rbac.authorization.k8s.io/tigera-operator created
clusterrolebinding.rbac.authorization.k8s.io/tigera-operator created
deployment.apps/tigera-operator created
```

Verify tigera-operator Pod status in the namespace “tigera-operator”. It can take few minutes for new pods to enter in to Running state.

```
kubectl get pods -n tigera-operator
```

```
cisco@cl-pod3-rtr:~$ kubectl get pods -n tigera-operator
NAME                               READY   STATUS    RESTARTS   AGE
tigera-operator-6c87cfc869-qmb5g   1/1     Running   0          40s
```

To check Custom Resource Definitions provided by Tigera Operator, issue command:

```
kubectl get crds
```

```
cisco@extrtr-wil-1:~$ kubectl get crds
NAME                                                  CREATED AT
apiservers.operator.tigera.io                         2024-02-01T17:50:23Z
bgpconfigurations.crd.projectcalico.org               2024-02-01T17:50:23Z
bgpfilters.crd.projectcalico.org                      2024-02-01T17:50:23Z
bgppeers.crd.projectcalico.org                        2024-02-01T17:50:23Z
blockaffinities.crd.projectcalico.org                 2024-02-01T17:50:23Z
caliconodestatuses.crd.projectcalico.org              2024-02-01T17:50:23Z
clusterinformations.crd.projectcalico.org             2024-02-01T17:50:23Z
felixconfigurations.crd.projectcalico.org             2024-02-01T17:50:23Z
globalnetworkpolicies.crd.projectcalico.org           2024-02-01T17:50:23Z
globalnetworksets.crd.projectcalico.org               2024-02-01T17:50:23Z
hostendpoints.crd.projectcalico.org                   2024-02-01T17:50:23Z
imagesets.operator.tigera.io                          2024-02-01T17:50:23Z
installations.operator.tigera.io                      2024-02-01T17:50:23Z
ipamblocks.crd.projectcalico.org                      2024-02-01T17:50:23Z
ipamconfigs.crd.projectcalico.org                     2024-02-01T17:50:23Z
ipamhandles.crd.projectcalico.org                     2024-02-01T17:50:23Z
ippools.crd.projectcalico.org                         2024-02-01T17:50:23Z
ipreservations.crd.projectcalico.org                  2024-02-01T17:50:23Z
kubecontrollersconfigurations.crd.projectcalico.org   2024-02-01T17:50:23Z
networkpolicies.crd.projectcalico.org                 2024-02-01T17:50:23Z
networksets.crd.projectcalico.org                     2024-02-01T17:50:23Z
tigerastatuses.operator.tigera.io                     2024-02-01T17:50:23Z
```

Step 11: Apply custom resources to run installation of Calico CNI.

```
kubectl apply -f custom_resources_aci_calico.yaml
```

```
cisco@cl-pod3-rtr:~$ kubectl apply -f custom_resources_aci_calico.yaml
namespace/calico-system created
installation.operator.tigera.io/default created
apiserver.operator.tigera.io/default created
secret/bgp-secrets created
role.rbac.authorization.k8s.io/secret-access created
rolebinding.rbac.authorization.k8s.io/secret-access created
serviceaccount/calicoctl created
pod/calicoctl created
clusterrole.rbac.authorization.k8s.io/calicoctl created
clusterrolebinding.rbac.authorization.k8s.io/calicoctl created
namespace/aci-containers-system created
configmap/acc-provision-config created
```

Once applied, check running Pods and check if CoreDNS got assigned IP address from the Pod network.

```
kubectl get pods -A -o wide
```

```
cisco@extrtr-wil-1:~$ kubectl get pods -A -o wide
NAMESPACE        NAME                                       READY   STATUS    RESTARTS   AGE     IP              NODE     NOMINATED NODE   READINESS GATES
calico-system    calico-kube-controllers-7788dcf8db-trdpf   1/1     Running   0          19s     10.2.84.130     node-1   <none>           <none>
calico-system    calico-node-68c7z                          0/1     Running   0          19s     192.168.1.13    node-3   <none>           <none>
calico-system    calico-node-lhnkg                          0/1     Running   0          19s     192.168.1.11    node-1   <none>           <none>
calico-system    calico-node-srzp7                          0/1     Running   0          19s     192.168.1.12    node-2   <none>           <none>
calico-system    calico-typha-564f95945-l7728               1/1     Running   0          12s     192.168.1.13    node-3   <none>           <none>
calico-system    calico-typha-564f95945-xdlv4               1/1     Running   0          19s     192.168.1.12    node-2   <none>           <none>
calico-system    csi-node-driver-526fh                      2/2     Running   0          19s     10.2.139.64     node-3   <none>           <none>
calico-system    csi-node-driver-lzfk8                      2/2     Running   0          19s     10.2.84.131     node-1   <none>           <none>
calico-system    csi-node-driver-xmf6r                      2/2     Running   0          19s     10.2.247.0      node-2   <none>           <none>
kube-system      calicoctl                                  1/1     Running   0          20s     192.168.1.13    node-3   <none>           <none>
kube-system      coredns-5dd5756b68-fcbjx                   1/1     Running   0          5m56s   10.2.84.128     node-1   <none>           <none>
kube-system      coredns-5dd5756b68-rstwr                   1/1     Running   0          5m56s   10.2.84.129     node-1   <none>           <none>
kube-system      etcd-node-1                                1/1     Running   0          6m12s   192.168.1.11    node-1   <none>           <none>
kube-system      kube-apiserver-node-1                      1/1     Running   0          6m12s   192.168.1.11    node-1   <none>           <none>
kube-system      kube-controller-manager-node-1             1/1     Running   0          6m12s   192.168.1.11    node-1   <none>           <none>
kube-system      kube-proxy-ncs42                           1/1     Running   0          5m56s   192.168.1.11    node-1   <none>           <none>
kube-system      kube-proxy-pt8r2                           1/1     Running   0          5m52s   192.168.1.12    node-2   <none>           <none>
kube-system      kube-proxy-zvx29                           1/1     Running   0          5m39s   192.168.1.13    node-3   <none>           <none>
kube-system      kube-scheduler-node-1                      1/1     Running   0          6m12s   192.168.1.11    node-1   <none>           <none>
tigera-operator  tigera-operator-597bf4ddf6-zjwqw           1/1     Running   0          93s     192.168.1.13    node-3   <none>           <none>
```

At this stage Calico CNI works in the BGP Full Mesh mode. BGP has been established between nodes only, but peering with ACI L3out is not yet done. Check current operation mode using calicoctl. This command requires logging to the Kubernetes node and run it locally:

```
ssh 192.168.1.11
sudo ./calicoctl node status
# <sudo password: clams24 >
```

```
cisco@node-1:~$ sudo ./calicoctl node status
[sudo] password for cisco:
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-------+----------+-------------+
| PEER ADDRESS |     PEER TYPE     | STATE |  SINCE   |    INFO     |
+--------------+-------------------+-------+----------+-------------+
| 192.168.1.4  | node-to-node mesh | up    | 14:21:55 | Established |
| 192.168.1.3  | node-to-node mesh | up    | 14:22:03 | Established |
+--------------+-------------------+-------+----------+-------------+
```

Return back to the external router

```
exit
```

Step 12: Apply Cisco recommended eBGP configuration to Calico to change default BGP Mesh to peer with ACI L3out.

Apply last manifest using calicoctl, since it is using CRDs exposed by Tigera API. This command does not need priviledges elevation.

```
./calicoctl apply -f custom_resources_calicoctl.yaml
```

After this command has been applied, following resources have been configured:

- IPPool
- BGPConfiguration
- BGPPeer

You can check details using following commands:

```
kubectl get IPPool -o yaml
kubectl get BGPPeer -o yaml
kubectl get BGPConfiguration -o yaml
```

Step 13: Verify BGP peering with Cisco ACI L3out, it could take few seconds for Calico to restart all vRouters and establish BGP session with ACI.

Check current operation mode using calicoctl. This command requires logging to the Kubernetes node and run it locally:

```
ssh 192.168.1.11
cisco@node-1:~$ sudo ./calicoctl node status
[sudo] password for cisco:
Calico process is running.

IPv4 BGP status
+---------------+---------------+-------+----------+-------------+
| PEER ADDRESS  |   PEER TYPE   | STATE |  SINCE   |    INFO     |
+---------------+---------------+-------+----------+-------------+
| 192.168.1.251 | node specific | up    | 14:45:18 | Established |
| 192.168.1.252 | node specific | up    | 14:45:18 | Established |
+---------------+---------------+-------+----------+-------------+
```

Return back to the external router

```
exit
```

Calico automatically push configuration to the Kubernetes nodes that have been annotated with exact string: projectcalico.org/labels: {"rack_id":"1"}

```
cisco@node-1:~$ kubectl describe node node-1
Name:               node-1
Roles:              control-plane
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=node-1
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/control-plane=
                    node.kubernetes.io/exclude-from-external-load-balancers=
                    rack_id=1
Annotations:        csi.volume.kubernetes.io/nodeid: {"csi.tigera.io":"node-1"}
                    kubeadm.alpha.kubernetes.io/cri-socket: unix:///var/run/containerd/containerd.sock
                    node.alpha.kubernetes.io/ttl: 0
                    projectcalico.org/IPv4Address: 192.168.1.11/24
                    volumes.kubernetes.io/controller-managed-attach-detach: true
```

This label makes the Kubernetes cluster aware of the network topology. Each Kubernetes node peer with the L3out Anchor nodes within it's own Rack.

Step 14: Verify BGP prefixes received by ACI Leaf switches.

Open PuTTY terminal and login to APIC IP: 172.18.0.120

Username: cisco

Password: clams24

```
fabric 101 show ip bgp summary vrf clwilX:clwilX    # X = use your setup ID
```

```
krk-dmz-apic1# fabric 101 show ip bgp summary vrf clwil1:clwil1
------------------------------------------------------------------
 Node 101 (Leaf-101)
------------------------------------------------------------------
BGP summary information for VRF clwil1:clwil1, address family IPv4 Unicast
BGP router identifier 1.1.1.1, local AS number 65001
BGP table version is 35, IPv4 Unicast config peers 4, capable peers 3
7 network entries and 16 paths using 1680 bytes of memory
BGP attribute entries [6/1056], BGP AS path entries [0/0]
BGP community entries [0/0], BGP clusterlist entries [1/4]

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
192.168.1.11    4 64512     115     117       35    0    0 00:01:50 2
192.168.1.12    4 64512     115     118       35    0    0 00:01:50 2
192.168.1.13    4 64512     115     118       35    0    0 00:01:50 2
```

```
fabric 101 show ip route vrf clwilX:clwilX    # X = use your setup ID
```

```
krk-dmz-apic1# fabric 101 show ip route vrf clwil1:clwil1
------------------------------------------------------------------
 Node 101 (Leaf-101)
------------------------------------------------------------------
IP Route Table for VRF "clwil1:clwil1"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

0.0.0.0/0, ubest/mbest: 1/0
    *via 192.168.255.1, vlan147, [1/0], 00:19:06, static
10.2.84.128/26, ubest/mbest: 1/0
    *via 192.168.1.11%clwil1:clwil1, [20/0], 00:02:39, bgp-65001, external, tag 64512
10.2.139.64/26, ubest/mbest: 1/0
    *via 192.168.1.13%clwil1:clwil1, [20/0], 00:02:39, bgp-65001, external, tag 64512
10.2.247.0/26, ubest/mbest: 1/0
    *via 192.168.1.12%clwil1:clwil1, [20/0], 00:02:39, bgp-65001, external, tag 64512
10.3.0.0/16, ubest/mbest: 3/0
    *via 192.168.1.11%clwil1:clwil1, [20/0], 00:02:39, bgp-65001, external, tag 64512
    *via 192.168.1.13%clwil1:clwil1, [20/0], 00:02:39, bgp-65001, external, tag 64512
    *via 192.168.1.12%clwil1:clwil1, [20/0], 00:02:39, bgp-65001, external, tag 64512
192.168.1.0/24, ubest/mbest: 2/0, attached, direct
    *via 192.168.1.251, vlan148, [0/0], 00:12:10, direct
    *via 192.168.1.1, vlan148, [0/0], 00:12:10, direct
192.168.1.1/32, ubest/mbest: 1/0, attached
    *via 192.168.1.1, vlan148, [0/0], 00:12:10, local, local
192.168.1.251/32, ubest/mbest: 1/0, attached
    *via 192.168.1.251, vlan148, [0/0], 00:12:10, local, local
192.168.255.0/24, ubest/mbest: 2/0, attached, direct
    *via 192.168.255.251, vlan147, [0/0], 00:19:06, direct
    *via 192.168.255.254, vlan147, [0/0], 00:19:06, direct
192.168.255.251/32, ubest/mbest: 1/0, attached
    *via 192.168.255.251, vlan147, [0/0], 00:19:06, local, local
192.168.255.254/32, ubest/mbest: 1/0, attached
    *via 192.168.255.254, vlan147, [0/0], 00:19:06, local, local
```

Note: Cisco ACI Leaf switch receives following prefixes:

- External Service subnet (10.3.0.0/16) used to expose applications outside, this is usually routed subnet within the organization.
- Pod subnet chunk assigned per node – each node got assigned /26 subnet to address Pods that it runs. Only those specific prefixes are advertised by specific node,

because Calico wants to steer incoming traffic directly to the Pod that is running on particular node. Therefore Calico does not advertise whole prefix 10.2.0.0/16.

Congratulations! Go to the next page for Task 2.

TASK 2: Deploying an application

Deploy Guest Book

As a small Demo we can now deploy the Guestbook application. The application is already pre-configured to be deployed

Step 1: Login to clwilX-rtr from Windows Jumphost (password:)

Step 2: Deploy the Guestbook application

```
kubectl apply -f guestbook.yaml
```

```
cisco@cl-pod1-rtr:~$ kubectl apply -f guestbook.yaml
deployment.apps/redis-leader created
service/redis-leader created
deployment.apps/redis-follower created
service/redis-follower created
deployment.apps/frontend created
service/frontend created
```

Now we need to edit the frontend service so that we can expose it outside of the fabric. Use "kubectl edit" command that will open frontend service manifest in vi editor.

```
kubectl edit svc frontend
```

Change the type to NodePort, set the external IP to 10.3.0.1 and the externalTrafficPolicy to Local.

The config should be like the below one:

```
spec:
  clusterIP: 10.96.198.247
  clusterIPs:
  - 10.96.198.247
  externalIPs:
  - 10.3.0.1
  externalTrafficPolicy: Local
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - nodePort: 32241
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: guestbook
    tier: frontend
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

Verify the service is exposed with the following command:

```
cisco@node-1:~$ kubectl get svc frontend
NAME       TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
frontend   NodePort   10.96.198.247   10.3.0.1      80:32241/TCP   25m
```

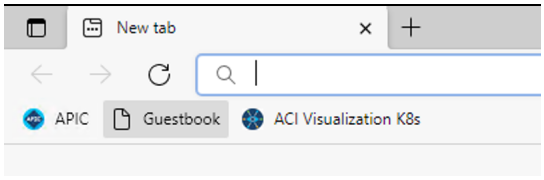
Step 3: Check what additional routes have been received comparing to previous task.

```
krk-dmz-apic1# fabric 101 show ip route vrf clwilX:clwilX    # X = use your setup
ID
------------------------------------------------------------------
```

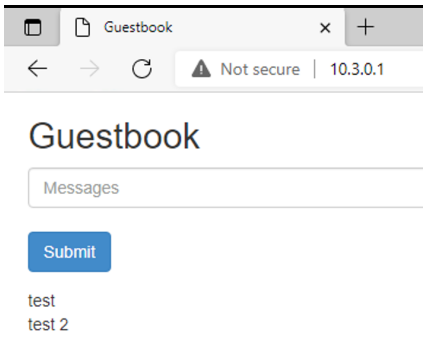
```
 Node 101 (Leaf-101)
--------------------------------------------------------------------
IP Route Table for VRF "clwil1:clwil1"
'*' denotes best ucast next-hop
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

0.0.0.0/0, ubest/mbest: 1/0
    *via 192.168.255.1, vlan46, [1/0], 06:53:05, static
10.2.84.128/26, ubest/mbest: 1/0
    *via 192.168.1.2%clpod1:clpod1_vrf, [20/0], 01:08:11, bgp-65001, external, tag 64512
10.2.139.64/26, ubest/mbest: 1/0
    *via 192.168.1.4%clpod1:clpod1_vrf, [20/0], 01:08:11, bgp-65001, external, tag 64512
10.2.247.0/26, ubest/mbest: 1/0
    *via 192.168.1.3%clpod1:clpod1_vrf, [20/0], 01:08:00, bgp-65001, external, tag 64512
10.3.0.0/16, ubest/mbest: 3/0
    *via 192.168.1.2%clpod1:clpod1_vrf, [20/0], 01:08:00, bgp-65001, external, tag 64512
    *via 192.168.1.4%clpod1:clpod1_vrf, [20/0], 01:08:00, bgp-65001, external, tag 64512
    *via 192.168.1.3%clpod1:clpod1_vrf, [20/0], 01:08:00, bgp-65001, external, tag 64512
10.3.0.1/32, ubest/mbest: 2/0
    *via 192.168.1.4%clpod1:clpod1_vrf, [20/0], 00:08:58, bgp-65001, external, tag 64512
    *via 192.168.1.3%clpod1:clpod1_vrf, [20/0], 00:08:58, bgp-65001, external, tag 64512
```

Note that you received host prefix for external and internal service.

Step 4: In windows jumphost, Open Edge browser and click to the “Guestbook” bookmark:



It will open page using IP address you exposed your Guestbook application outside.



You can type any text and see if it is displayed stored in the page. If yes, this means that frontend has communication with Redis backend.

Congratulations! Process to task 3 in the next page

TASK 3: Deploy visibility dashboard

Visualisation of Kubernetes using ACI (vkaci) is an open-source tool that generates a cluster topology and provides a visual representation, using neo4j graph database by accessing ACI and K8s APIs. This tool lets you quickly build visualisations of K8s and ACI end to end topologies.

The visualisation of the cluster network is based on two main views. One is an interactive graph network view, and the other is a detailed tree table view.

Step 1: Login to ext-rtr from Windows Jumphost (password:)

Step 2: Create vkaci namespace:

```
kubectl create ns vkaci-ns
```

Step 3: apply helm chart (Helm is an "package manager" for Kubernetes, it applies multiple manifests at once and has rich templating engine)

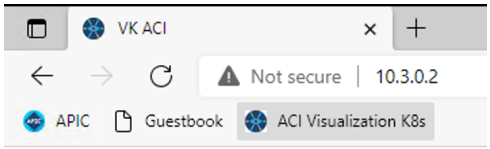
```
helm repo add vkaci https://datacenter.github.io/ACI-Kubernetes-Visualiser
helm install -n vkaci-ns vkaci vkaci/vkaci -f vkaci-values.yaml
```

Step 4: Check vkaci app pods status and wait until they will be in the Running state.

Note: It may happen that neo4j will not start before vkaci-app, which will cause vkaci-app to crash, but it will eventually restart.

```
kubectl get pods -n vkaci-ns
NAME                                READY   STATUS    RESTARTS        AGE
vkaci-app-64f6c97d5-fxzrf           1/1     Running   2 (3m46s ago)   4m5s
vkaci-app-neo4j-58876c45f7-xm6nc    1/1     Running   0               4m5s
```

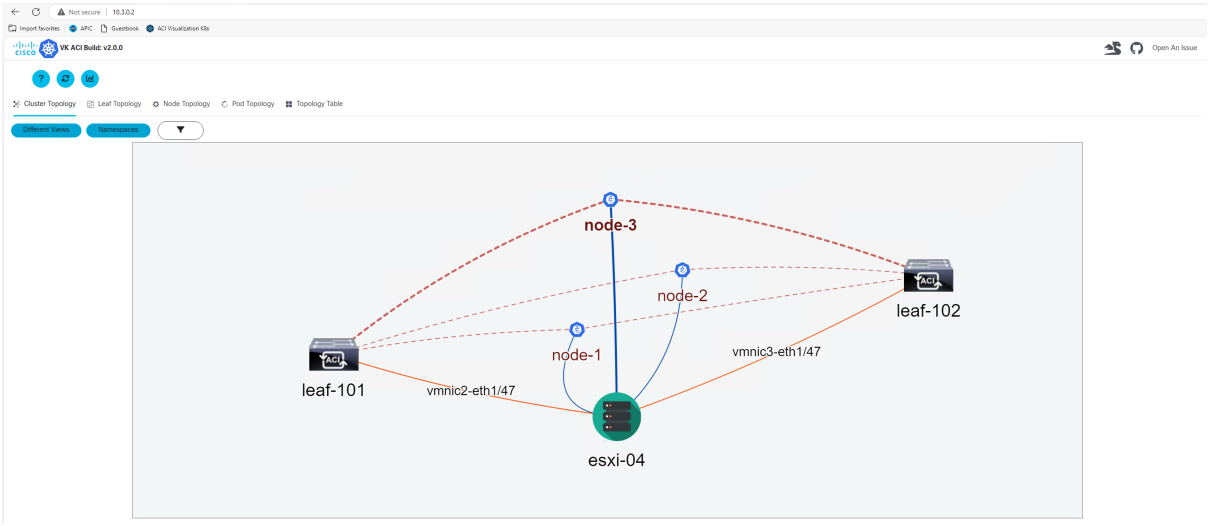
Step 5: In windows jumphost, Open Edge browser and click to the "ACI Visualization K8s" bookmark



Step 6: Browse through the application – see tabs Cluster Topology, Leaf Topology, Node Topology, Pod Topology and Topology Table.

Each view has its own filters.

Cluster topology – Click on the "Different Views" and look what is displayed. In the "View all" you can see all Pods, but this view could be busy – use filters and select Namespace: Default, then click on the Filter icon and specify labels app: guestbook



You will see exactly on which **nodes** application Pods are running, hypervisor name where the nodes are hosted (esxi-04) and how it is connected to ACI – All in one view! 😊