Community Experience Distilled

DevOps for Networking

Boost your organization's growth by incorporating networking in the DevOps culture

Steven Armstrong

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

1

Impact of Cloud On Networking

This chapter will look at ways that networking has changed in the private data centre and evolved in the last few years. It will focus emergence of AWS for public cloud and OpenStack for private cloud have changed the way developers want to consume networking. It will look at some of the networking services AWS and OpenStack provide out the box and look at some of the features they provide. It will show examples of how these cloud platforms have made networking a commodity much like infrastructure.

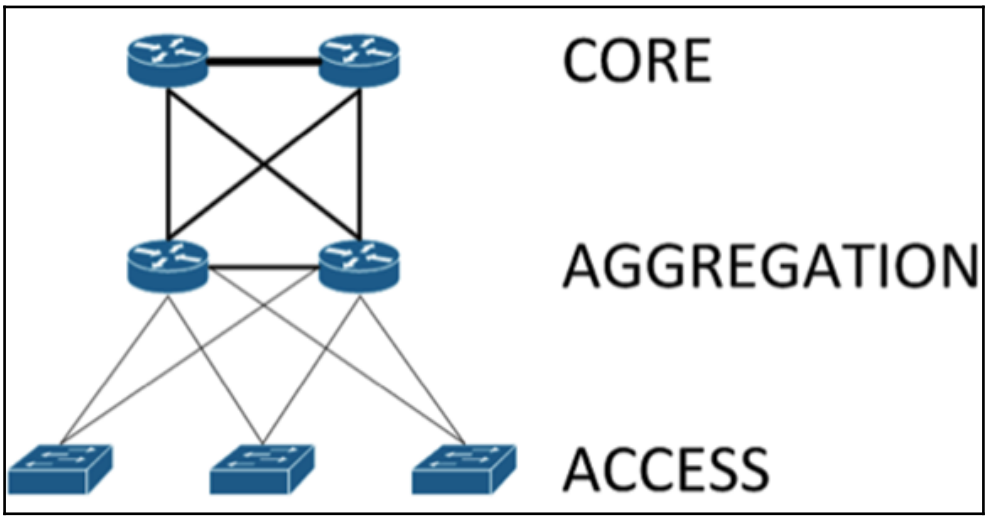
In this chapter the following topics will be covered:

- The difference between spanning tree networks and leaf spine networking
- Changes that have occurred in networking with the introduction of public cloud
- The Amazon Web Services Approach to Networking
- The OpenStack Approach to Networking

The difference between spanning tree networks and leaf spine networking

Traditionally companies private data centres have implemented 3 tier Layer 2 networks based on the **spanning tree protocol** (**SPT**). The implementation of SPT provides a number of options for network architects in terms of implementation, but it also adds a layer of complexity to the network. Implementation of the SPT gives network architects the certainty that it will prevent layer 2 loops from occurring in the network.

A typical representation of a 3 tier Layer 2 SPT based network can be shown below:



- The core layer provides routing services to other parts of the data centre and contains the core switches
- The aggregation layer provides connectivity to adjacent access layer switches and the top of the spanning tree *core.*
- The bottom of the tree is the *access*layer, this is where bare metal or virtual machines connect to the network and is segmented using grouping different vlans.

The use of Layer 2 networking and SPT mean that at the access layer of the network will use vlans spread throughout the network. The vlans sit at the access layer, which is where virtual machines or bare metal servers are connected. Typically these vlans are grouped by type of application, and firewalls are used to further isolate and secure them.

Traditional networks are normally segregated into some combination of the following:

- **Front End:** typically has web servers that require external access will sit
- **Business Logic:** often contains stateful services
- **Back End:** typically contains database servers

Applications talk to each other by tunnelling between these firewalls, with specific ACL rules that are serviced by network teams and governed by security teams.

When using spanning tree protocol in a layer 2 network, all switches go through an election process to determine the root switch, which is granted to the switch with the lowest bridge

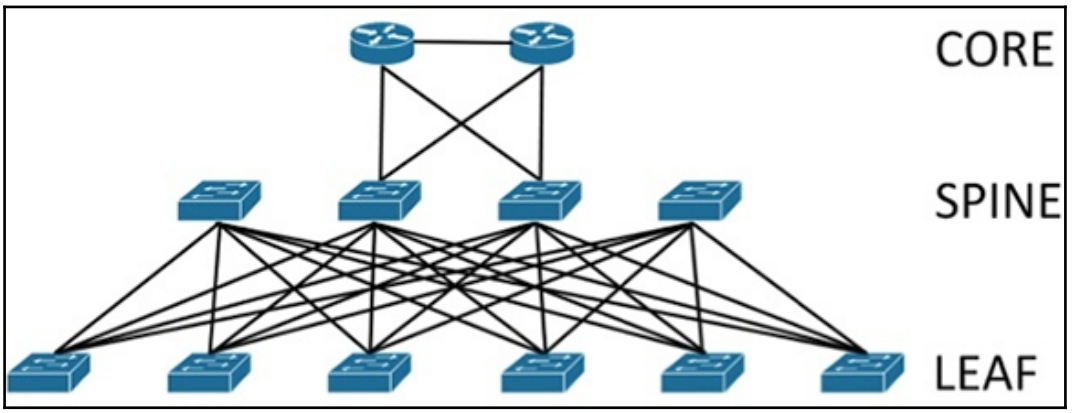
id, with a bridge id encompassing the bridge priority and MAC address of the switch. Once elected the root switch becomes the base of the spanning tree, all other switches in the spanning tree are deemed non-root will calculate their shortest path to the root and then block any redundant links so there is one clear path. The calculation process to work out the shortest path is referred to as network convergence.

Network architects designing the layer 2 network need to be careful about the placement of the root switch, as all network traffic will need to flow through it, so it should be selected with care and given an appropriate bridge priority as part of the network reference architecture design. If at any point, switches have been given the same bridge priority then the bridge with the lowest MAC address wins.

Network architects should also design the network for redundancy so that if a root switch fails, there is a nominated backup root switch typically with a priority of one value less that the nominated root switch, which will take over as when a root switch fails. In the scenario the root switch fails the election process will begin again and the network will converge which can take some time.

The use of Spanning Tree protocol is not without its risks, if it does fail due to user configuration error, data centre equipment failure, software failure on a switch or bad design then the consequences to a network can be huge. The result can be that loops might form within the bridged network which can result in a flood of broadcast, multicast, unknown-unicast storms that can potentially take down the entire network leading to long network outages. The complexity associated with network architects or engineers troubleshooting spanning tree protocol issues is non-trivial so it is paramount that the network design is sound.

In recent years with the emergence of cloud computing we have seen data centres move away from a spanning tree protocol in favor of a leaf-spine networking architecture.



In a leaf-spine architecture:

- Spine switches are connected into a set of core switches
- Spine switches are then connected with leaf switches with each leaf switch deployed top of rack
- This means that any leaf switch can connect to a spine switch in one hop

A leaf spine architecture is built on layer 3 routing principles to optimise throughput and reduce latency, with both leaf and spine switches communicating with each other via **external border gate protocol** (**eBGP**) as the routing protocol for the IP fabric. eBGP establishes a TCP connection to each of its BGP peers before BGP updates can be exchanged between the switches. Leaf switches in the implementation will sit at top of rack and can be configured in **multi-chassis link aggregation** (**MLAG**) mode using NIC bonding. MLAG was originally used with Spanning Tree Protocol (STP) so that two or more switches are bonded to act like a single switch and could use for redundancy and appear as one switch to the STP, this provided multiple uplinks for redundancy in the event of a failure as the switches are peered and it worked around the need to disable redundant paths. Leaf switches can often have internal border gate protocol (iBGP) configured between the pairs of switches for resiliency.

In a leaf spine architecture spine switches do not connect to other spine switches and leaf switches do not connect directly to other leaf switches unless bonded top of rack using MLAG NIC bonding. All links in a leaf spine architecture are setup to forward with no looping. Leaf spine architectures are typically configured to implement **equal cost multi-pathing** (**ECMP**) which allows all routes to be configured on the switches so they can access any spine switch in the layer 3 routing fabric. ECMP means that leaf switches routing table has the next-hop configured to forward to each spine switch. In an ECMP setup each leaf

node has multiple paths of equal distance to each spine switch, so if a spine or leaf switch fails, there is no impact as long as there are other active paths to another adjacent spine switches. ECMP is used to load balance flows and supports the routing of traffic across multiple paths. This is in contrast to the STP which switches off all but one path to the root when the network coverages.

Typically leaf spine architectures designed for high performance use 10G access ports at leaf switches mapping to 40G spine ports. When device port capacity becomes an issue, new leaf switches can be added by connecting it to every spine on the network while pushing the new configuration to every switch. This means that network teams can easily scale out the network horizontally without managing or disrupting the switching protocols or impacting the network performance.

The benefits of a leaf spine architecture are:

- Consistent latency and throughput in the network
- Consistent performance for all racks
- Network once configured becomes less complex
- Simple scaling of new racks by adding new leaf switches at top of rack
- Consistent performance, subscription and latency between all racks
- East to west traffic performance is optimised (virtual machine to virtual machine communication) to support micro-service applications
- Removes VLAN scaling issues, controls broadcast and fault domains

The one drawback of a leaf spine topology is the amount of cables it consumes in the data centre.

Modern switches have now moved towards open source standards so they are pluggable, the open standard for virtual switches is Open vSwitch, which was born out of the necessity to come up with an open standard that allowed a virtual switch to forward traffic to different virtual machines on the same physical host and physical network. Open vSwitch uses **Open vSwitch database** (**OVSDB**) that has a standard extensible schema.

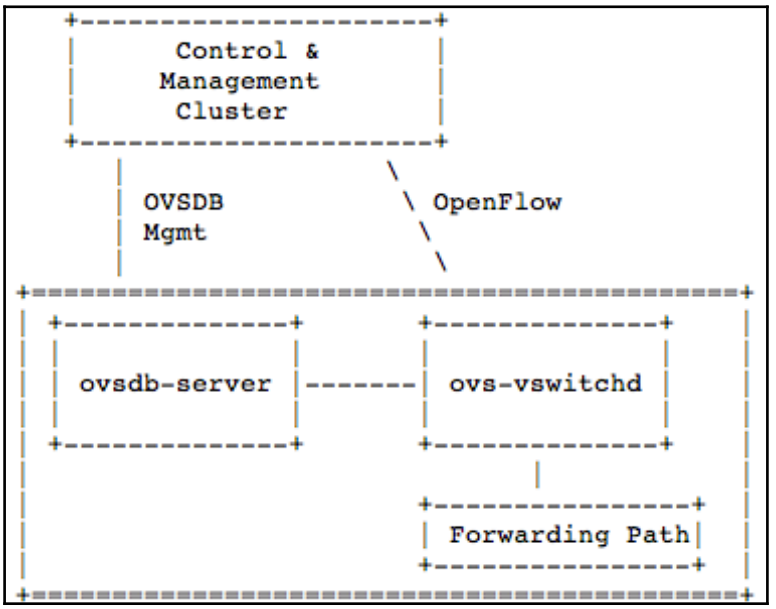
Open vSwitch is typically deployed at the hypervisor level with the following hypervisors now implementing Open vSwitch as the virtual switching technology:

- KVM
- Xen
- Hyper-V

Hyper-V has recently moved to support Open vSwitch using the implementation created by Cloudbase. Open vSwitch talks OpenFlow from virtual switch to physical switch to

communicate and can be programmatically extended to fit the needs of vendors.

In the diagram below you can see that the Open vSwitch architecture. Open vSwitch is typically installed on compute that is running KVM, Xen or Hyper-V virtualisation layer:



The ovsdb-server contains the OVSDB schema that holds all switching information for the virtual switch. The ovs-vswitchd daemon talks OpenFlow to any Control and Management Cluster which could be any SDN controller that can communicate using the OpenFlow protocol.

Controllers use OpenFlow to install flow state on the virtual switch and OpenFlow dictates what action to when packets are received by the virtual switch. When Open vSwitch receives a packet it has never seen before and has no matching flow entries, it sends this packet to the controller. The controller then makes a decision on how to handle this packet based on the flow rules to either block or forward. The ability for Configuration of quality of service (QoS) and other statistics is possible on Open vSwitch. Open vSwitch is used to configure security rules and provision ACL rules at the switch level on a hypervisor.

A leaf spine architecture allows overlay networks to be easily built, meaning that cloud and tenant environments are easily connected to the layer 3 routing fabric. Hardware Vxlan Tunnel Endpoints (VTEPs) ips are associated with each leaf switch or a pair of leaf switches in MLAG mode and are connected to each physical compute host via VXLAN to each Open

vSwitch that is installed on a hypervisor.

This allows an SDN controller to build an overlay network that create VXLAN tunnels to the physical hypervisors utilising Open vSwitch, new VXLAN tunnels are created automatically if new compute is scaled out, then SDN controllers can create new VXLAN tunnels on the leaf switch as they are peered with the leaf switches hardware VTEP.

Modern switch vendors such as Arista, Cumulus and many others use OVSDB and this allows SDN controllers to integrate at the control and management cluster level. As long as an SDN controller uses OVSDB and OpenFlow protocol, they can seamlessly integrate with the switches and are not tied into specific vendors. This gives end users a greater depth of choice when choosing switch vendors and SDN controllers which can be matched up as they communicate using the same open standard protocol.

Changes that have occurred in networking with the introduction of public cloud

It is unquestionable that the emergence of the **Amazon Web Services** (**AWS**) which was launched in 2006 changed and shaped the networking landscape forever. Today in 2016 the AWS VPC secures a set of Amazon EC2 instances (VMs) that can be connected to any existing network using aVPNconnection. This simple construct has changed the way that developers want and expect to consume networking.

In 2016 we live in a consumer based society with mobile phones allowing us instant access to the internet, films, games or an array of different applications to meet our every need, instant gratification if you will, so it is easy to see the appeal of AWS has to end users.

AWS allows developers to provision instances (VMs) in their own personal network, to their desired specification by selecting different flavours (CPU, RAM and disk) using a few button clicks or alternately a simple call to an API.

So now a valid question, why should developers be expected to wait long periods of time for either infrastructure or networking tickets to be serviced in on premise data centres when AWS is available? It really isn't a hard question answer when first reflecting, the solution has to either be move to AWS or create a private cloud solution that enables the same agility. However, with everything the answer isn't always that straight forward, arguments against using AWS and public cloud such as:

- Not knowing where the data is actually stored and in which data centre
- Not being able to hold sensitive data offsite

- Not being able to assure the performance
- High running costs

All are genuine blockers for some businesses that are highly regulated and need to meet regularity standards that may be enforced on businesses so as with most solutions it isn't the case of one size fits all.

In private data centres there is a cultural issue that teams have been set-up to work in silos and are not setup to succeed in an agile world, so a lot of the time utilising AWS is a quick fix for broken operational models.

Ticketing systems, a staple of broken internal operational models, are not a concept that aligns itself to speed, tickets typically take days or weeks to complete so requests are queued before virtual or physical servers can be provided to developers or access changes such as a simple modification ACL rules implemented. These options are required for the developers to do their job be it scaling up servers or prototyping new features so all of these hinder delivery of products to market.

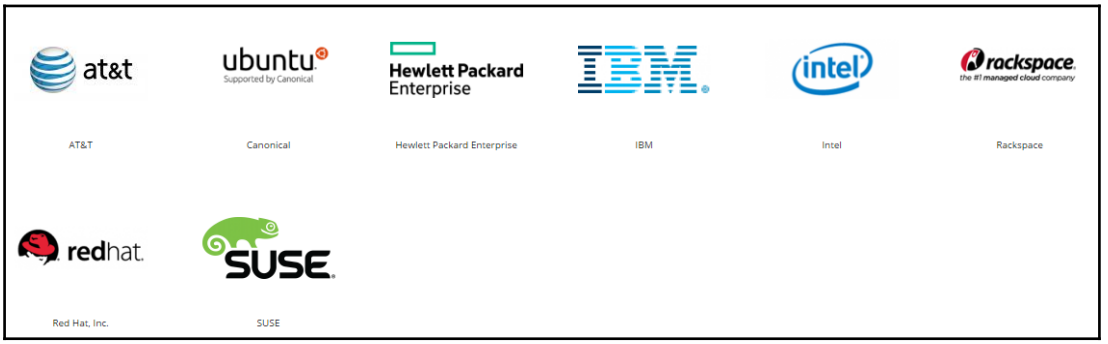
Put simply AWS has changed the expectations of developers, where they service their needs as quickly as making an alteration to an application on their mobile phone, free from slow internal IT operational models associated with companies.

But for start-ups and businesses that can use AWS that aren't deterred by regulatory requirements, it skips the need to hire teams to rack servers, configure network devices and pay for the running costs datacentres. It means they can start viable businesses and run them on AWS by putting in a credit card details the same way as you would purchase a new book on Amazon or EBay.

The reaction to AWS was met with trepidation from competitors, as it disrupted the cloud computing industry, from it spawned the idea for a new private cloud In 2010, which was a joint venture by Rackspace andNASA which launched an open-source cloud-software initiative known as OpenStack, which came about as Nasa couldn't put their data in a public cloud.

The OpenStack project intended to help organisations offer cloud-computing services running on standard hardware and directly set-out to mimic the model provided by AWS but do it as an open sourced project, that could be used to give AWS like ability for private cloud.

Since its inception in 2010 OpenStack has grown to have over 500 member companies as part of the OpenStack Foundation with platinum members and gold members that are comprised of the biggest IT vendors in the world that are driving the community.



OpenStack as it is open source project, which means its source code is publically available and its underlying architecture is available for analysis unlike AWS which acts like a magic box of tricks but it is not really known how it works underneath its shiny exterior.

OpenStack is typically used today to provides an Infrastructure as a Service (IaaS) function within the private cloud, where it makes commodity x86 compute, centralised storage and networking features available to end users to self-service their needs be it via the horizon dashboard or through a set of common API's.

Vendors such as Redhat, HP, Suse, Canonical, Mirantis and many more provide different distributions of OpenStack to customers, complete with different methods of installing the platform. Although the source code and features are the same, the business model for these vendors is that they harden OpenStack for enterprise use and their differentiator to customers is their professional services.

Typically OpenStack vendors will support build out, on-going maintenance and upgrades or any customisations a client needs that are fed back to the community. The beauty of OpenStack is that if vendors customise OpenStack for clients and create a real differentiator or competitive advantage, they cannot fork OpenStack or uniquely sell this feature and they have to contribute the source code for feature back to the upstream OpenStack project. This means that competing vendors all contribute to its success of OpenStack and benefit from each other's innovative work. The OpenStack project is not just for vendors though and everyone can contribute code and features to push the project forward.

OpenStack maintains a release cycle where an upstream release is created every six months and is governed by the OpenStack Foundation. It is important to note that many public clouds too such as RackSpace and GoDaddy are based on OpenStack too, so it is not exclusive to private clouds, but it has undeniably become increasingly popular as a private cloud alternative to AWS.

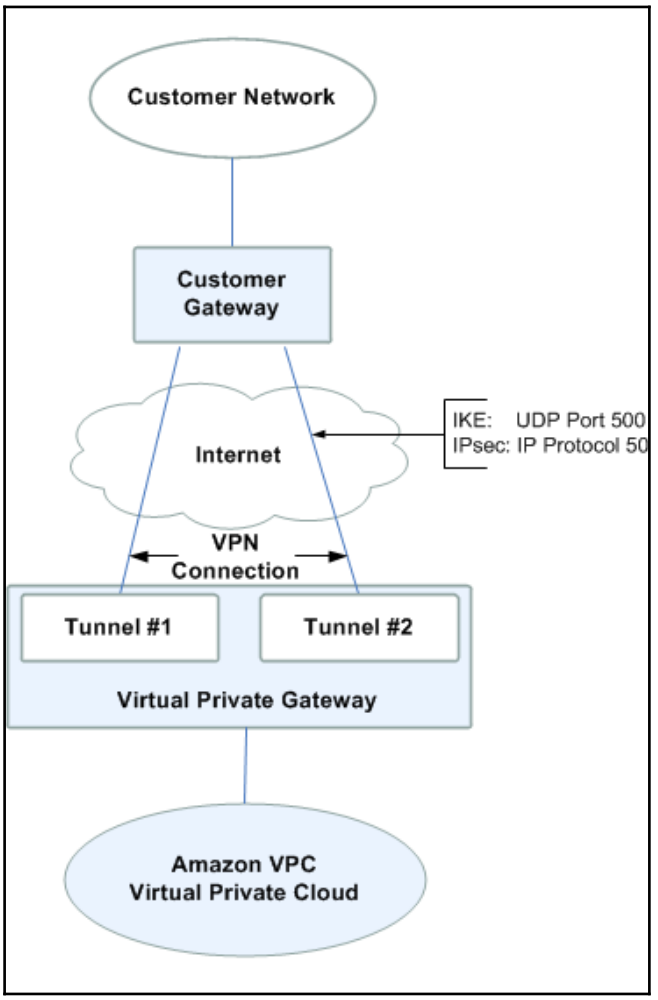
So how does AWS and OpenStack work in terms of networking? Both AWS and OpenStack

are made up of some mandatory and modular projects that are all integrated to make up its reference architecture. Some mandatory projects must be used such as compute and networking, which are the staple of any cloud solution, while others are modular bolt-ons to enhance or extend capability. This means that end users can cherry pick the projects they are interested in to make up their own personal portfolio.

The Amazon Web Services Approach to Networking

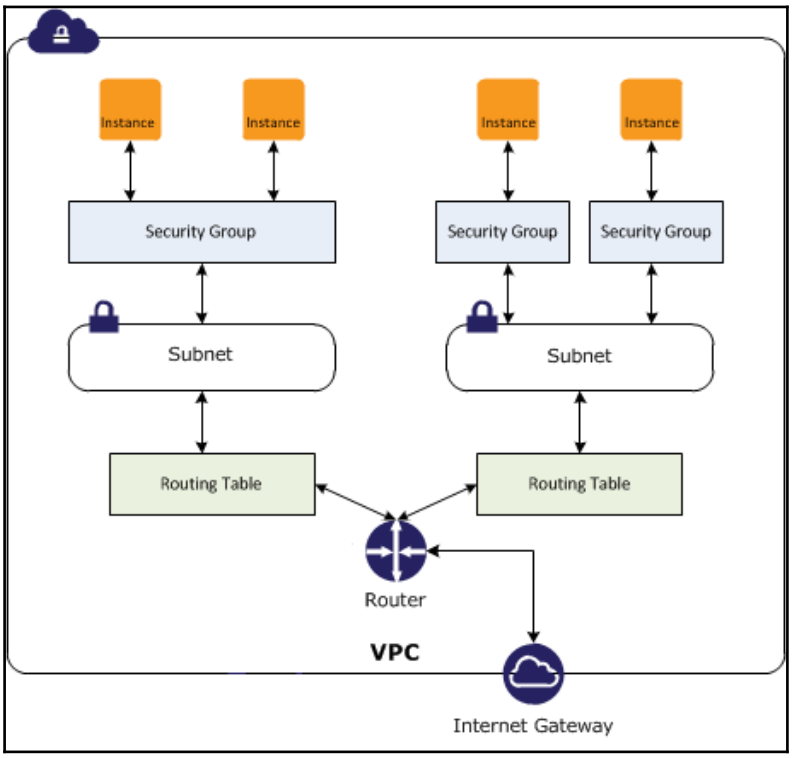
First we will look at AWS, a tenant network in AWS is instantiated using a virtual private cloud (VPC) which post 2013 deprecated AWS classic mode. A VPC is the new default setting for new customers wishing to access AWS. VPCs can also be connected to customer networks (private data centres), allowing AWS cloud to extend a private data centre for agility. The concept of connecting a private data centre to an AWS VPC is using something AWS refers to as a customer gateway and virtual private gateway. A virtual private gateway in simple terms is just two redundant VPN tunnels which are instantiated from the customer's private network.

Customer gateways expose a set of external static addresses from a customer site, which are typically nat-t to the hide the source address. UDP port 4500 is required to be accessible in the external firewall in the private datacentre. Multiple VPC's can be supported from one customer gateway device.



A VPC gives an isolated view of everything an AWS customer has provisioned in AWS public cloud. Different user accounts can then be set-up against a VPC using the AWS **Access Management** (IAM) service, which has customisable permissions.

An example of a VPC is shown below complete with instances (virtual machines) mapped one or more security groups and connected to different subnets connected to the VPC router:



A VPC closely simplifies networking greatly by putting the constructs into software and allows users the freedom to perform the following network functions:

- Create instances (VM's) mapped to subnets
- DNS entries are applied to instances
- Assignment of Public and Private IP addresses
- Create or associate subnets
- Custom routing
- Apply security groups with ACL rules

By default when an instance is instantiated in a VPC it will either be placed on a default subnet or custom subnet if specified. All VPC's come with a default router when the VPC is created, the router can have additional custom routes added and routing priority can also be set to forward traffic to particular subnets.

When the instance is spun up in AWS it will automatically be assigned a mandatory private ip address by DHCP as well as a public ip and DNS entry too unless dictated otherwise.

Private ips are used in AWS to route east to west traffic between instances when virtual machine need to talk with adjacent virtual machines on the same subnet, while public ips are available from the internet.

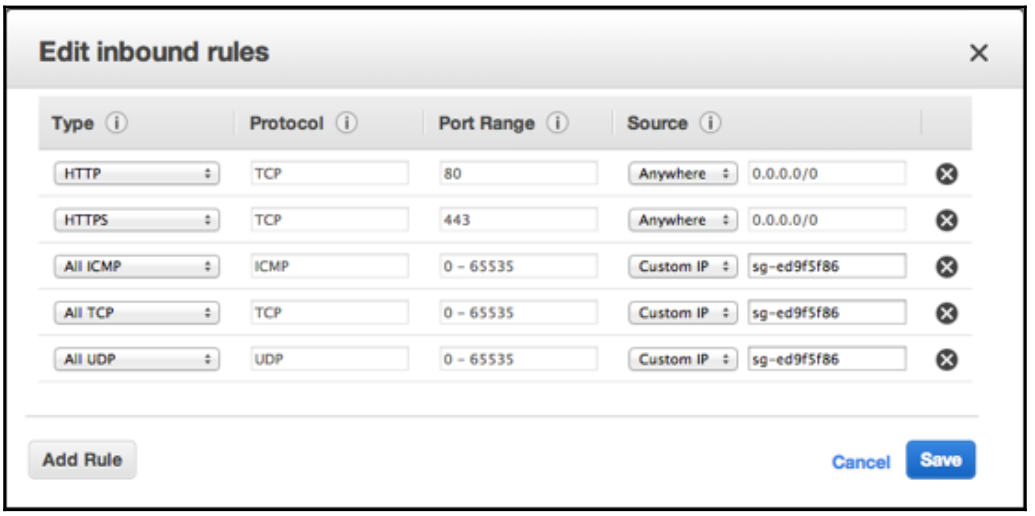
If a persistent public IP address is required for an instance, AWS offers the feature of elastic ip addresses, which is limited to five per VPC account, which means a failed instances ip can be quickly mapped to another instance. It is important to note it can take up to 24 hours for a public ip addresses DNS TTL to propagate when using AWS.

In terms of throughput, AWS instances can support a maximum transmission unit (MTU) of 1500 that can be passed to an instance in AWS so this needs to be considered when considering application performance.

Security groups in AWS are a way of grouping permissive ACL rules so don't allow explicit denies. AWS *security groups*act as a virtual firewalls for instances and can be associated with one or more instances network interfaces In a VPC, you can associate a network interface with up to five security groups, adding up to 50 rules to a security group, with a maximum of 500 security groups per VPC. A VPC in AWS account automatically has a default security group which will be automatically applied if no other security groups are specified.

Default security groups allows all outbound traffic and allow all inbound traffic only from other instances in a VPC that also have the default security group. The default security group cannot be deleted. Custom security groups when first created allow no inbound traffic but all outbound traffic is allowed.

Permissive ACL rules associated with security groups govern inbound traffic are added using the AWS console (GUI) as shown below or can be programmatically added using APIs. Inbound ACL rules associated with security groups can be added by specifying type, protocol, port range and the source address.



A VPC has access to different regions and availability zones of shared compute, which dictate the data centre that the AWS instances (virtual machines) will be deployed in. Regions in AWS are geographic areas that are completely isolated by design, where availability zones are isolated locations in that specific region so an availability zone is a subset of a region.

AWS gives users the ability to place their resources in different locations for redundancy as sometimes the health of a specific region or availability zone can suffer issues. Therefore, AWS users are encouraged to use more than one availability zones when deploying production workloads on AWS. Users can choose to replicate their instances and data across regions if they choose to.

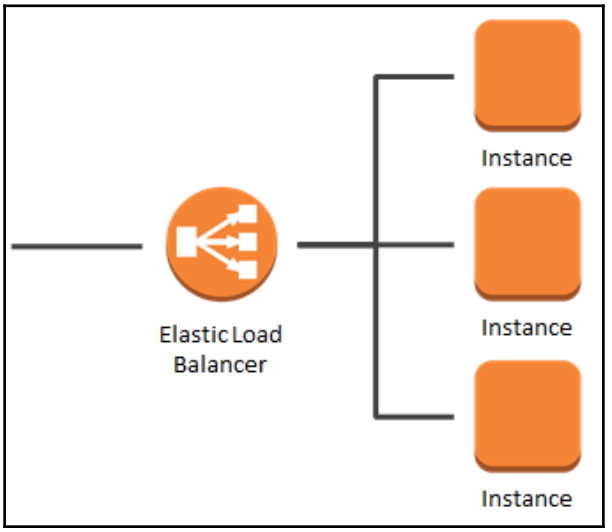
Within each isolated AWS region there are child availability zones, each availability zone is connected to sibling availability zones using low latency links. All communication from one region to another is across the public internet so utilising different regions that are geographically far away from each other will acquire latency and delay. Encryption of data should also be considered when hosting applications that send data across Regions.

An example of available of available AWS regions can be shown below:

Code	**Name**
us-east-1	**US East (N. Virginia)**
us-west-2	US West (Oregon)

Code	Name
us-west-1	US West (N. California)
eu-west-1	EU (Ireland)
eu-central-1	EU (Frankfurt)
ap-southeast-1	Asia Pacific (Singapore)
ap-northeast-1	Asia Pacific (Tokyo)
ap-southeast-2	Asia Pacific (Sydney)
ap-northeast-2	Asia Pacific (Seoul)
sa-east-1	South America (São Paulo)

AWS also allows **Elastic Load Balancing** (**ELB**) to be configured with a VPC as a bolt-on service. ELB which can either be internal or external, when an ELB is external it allows the creation of an internet-facing entry point into your VPC using an associated DNS entry and balances load between different instances. Security groups are assigned to ELBs to control the access ports that need to be used.



The OpenStack Approach to Networking

OpenStack is deployed in a data centre on multiple controllers, these controllers are home to all the OpenStack services, and can be installed on either virtual machines, bare metal servers or containers. The OpenStack controllers host all the OpenStack services in a highly available and redundant fashion.

Different OpenStack vendors provide different installers to install OpenStack, some examples of installers are RedHat Director (based on OpenStack triple O), Mirantis Fuel, HPs HPE installer (Based On Ansible) which all install OpenStack controllers and be used to scale out compute nodes on the OpenStack cloud acting as an OpenStack workflow management tool.

A breakdown of the core OpenStack services that are installed on a controller are as follows:

- **Keystone** is the identity service for OpenStack allowing user access which issues tokens and can be integrated with LDAP or Active directory
- **Glance** is the image service for Openstack storing all image templates for virtual machines or bare metal servers
- **Cinder** is the block storage service for Openstack which allows centralised storage volumes to be provisioned and attached to vms or bare metal servers which can then be mounted
- **Nova** is the compute service for Openstack used for provisioning vms and utilises different scheduling algorithms to work out where to place virtual machines on available compute
- **Horizon** is the Openstack dashboard that users connect to view the status of vms or bare metal servers that are running in a tenant network
- **Rabbitmq** is the message queue system for Openstack
- **Galera** is the database used to store all Openstack data in the nova (compute) and neutron (networking) databases holding vm, port and subnet information
- **Swift** is the object storage service for Openstack and can be used as a redundant storage backend that stores replicated copies of objects on multiple servers. Swift is not like traditional block or file based storage – objects can be any unstructured data.
- **Ironic** is the baremetal provisioning service for Openstack. Originally a fork of part of the Nova codebase, it allows provisioning of images on to bare metal servers and uses IPMI and ILO or DRAC interfaces to manage physical hardware
- **Neutron** is the networking service for OpenStack and contains ML2 and L3 agents and allows configuration of network subnets and routers

In terms of neutron networking services, neutrons architecture is very similar in constructs to AWS. Before provisioning any instances, first the OpenStack networking needs to be configured.

A Project, often referred to in OpenStack as a tenant, gives an isolated view of everything that a team has provisioned in an OpenStack cloud. Different user accounts can then be set-up against a Project (tenant) in keystone and integrated with LDAP or Active Directory and supports customisable permissions.

OpenStack neutron performs the following network functions:

- Create instances (VM's) mapped to networks
- Assigns ip addresses using it's in built DHCP service
- DNS entries are applied to instances from named servers
- Assignment of private and Floating IP addresses
- Create or associate network subnets
- Create routers
- Apply security groups

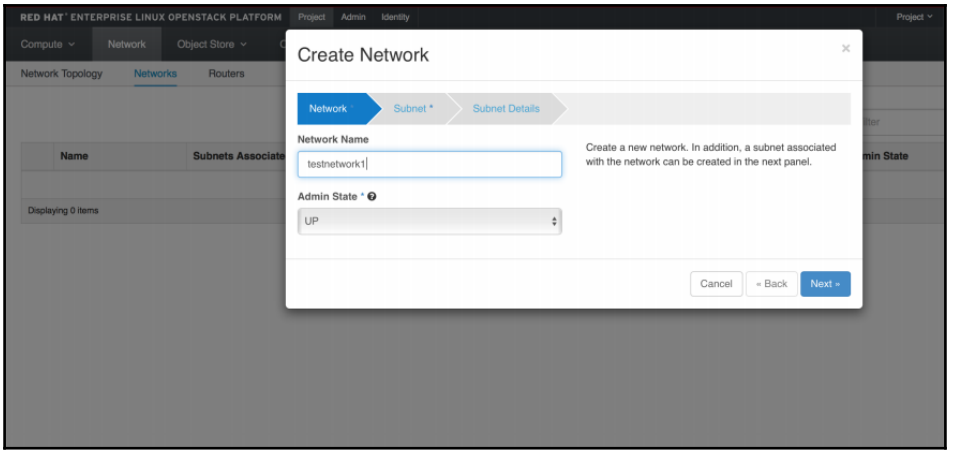
OpenStack is setup into its modular layer 2 (ML2) and layer 3 (L3) agents that are configured on the OpenStack controllers. OpenStacks ML2 plug-in allows OpenStack to integrate with switch vendors that utilise either Open vSwitch or Linux Bridge and acts as an agnostic plug-in to switch vendors that they create plug-ins against to make their switches OpenStack compatible. The ML2 agent runs on the hypervisor communicating over RPC to the compute host server.

OpenStack compute hosts are typically deployed using a hypervisor that utilises Open vSwitch, most OpenStack vendor distributions use the KVM hypervisor by default in their reference architectures, so this is deployed and configured on each compute host by the chosen OpenStack installer.

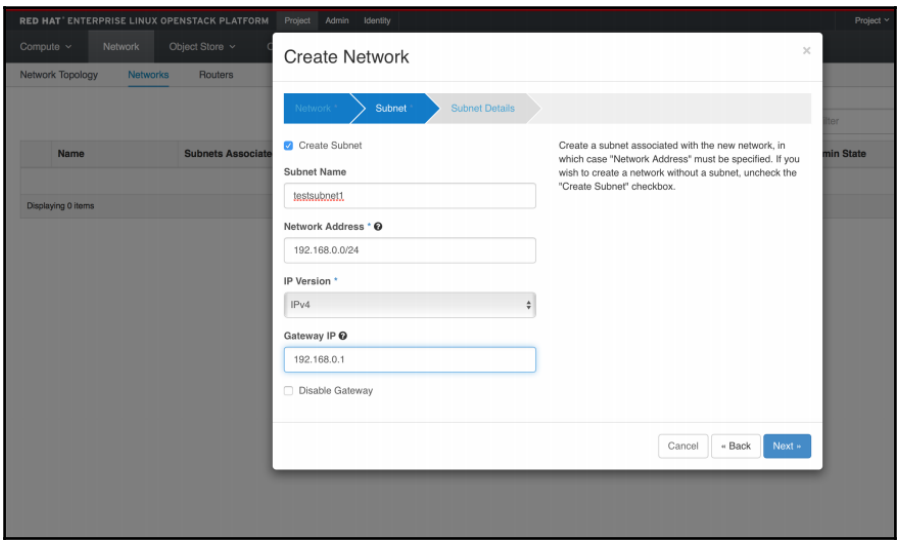
Compute hosts in OpenStack are connected to the access layer of the STP 3 tier model, or in modern networks connected to the leaf switches, with vlans connected to each individual OpenStack compute host. Tenant networks then are used to provide isolation between tenants and utilise vxlan and GRE tunnelling to connect the layer 2 network.

Open vSwitch runs in kernel space on the KVM hypervisor and looks after firewall rules by using OpenStack security groups that pushes down flow data via OVSDB from the switches. *The* neutron L3 agent allows OpenStack to route between tenant networks and uses neutron routers which are deployed within the tenant network to accomplish this, without a neutron router networks are isolated from each other and everything else.

Typically when setting up simple networking using neutron in a Project (tenant) network, two different networks, an internal network and an external network will be configured. The internal network will be used for east west traffic between instances. This is created as shown below in the horizon dashboard with an appropriate network name:

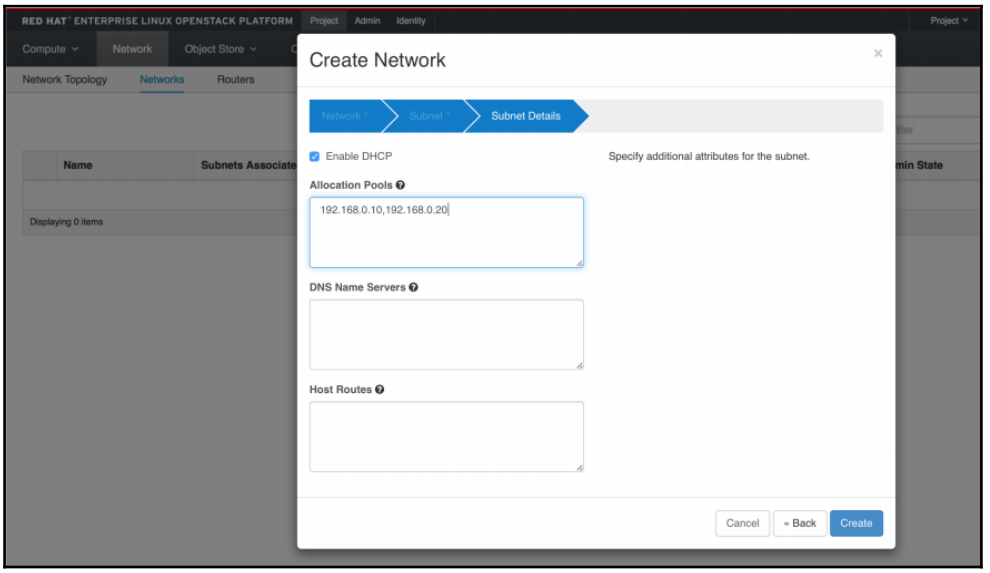


The subnet name and subnet range are then specified in the subnet section as shown below:

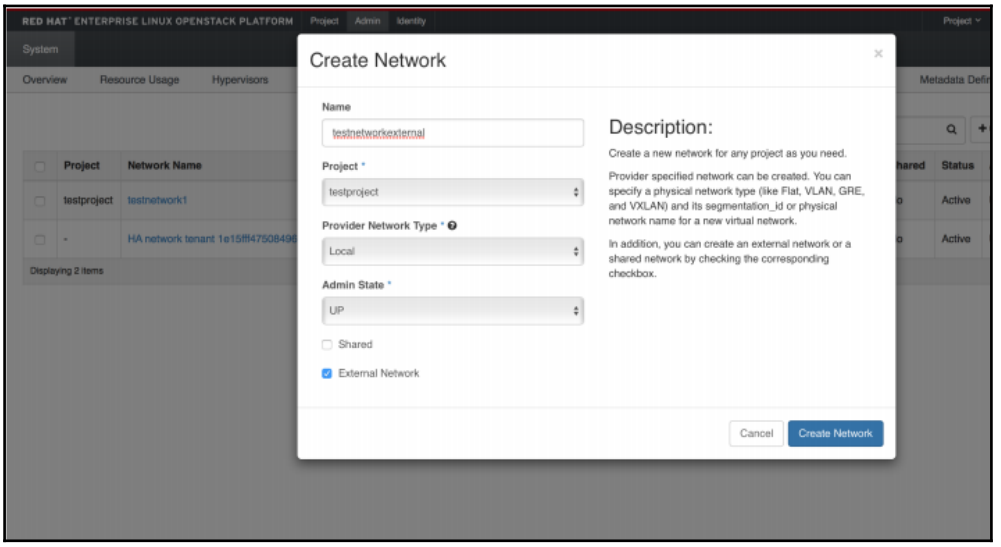


Finally DHCP is enabled on the network and any named allocation pools (specifies only a

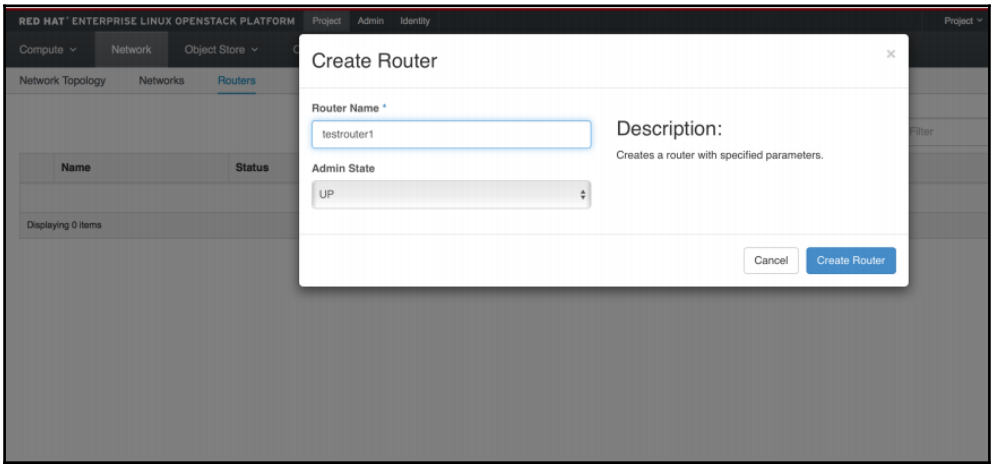
range of addresses that can be used in a subnet) are optionally configured alongside any named DNS servers as seen in the following screenshot.



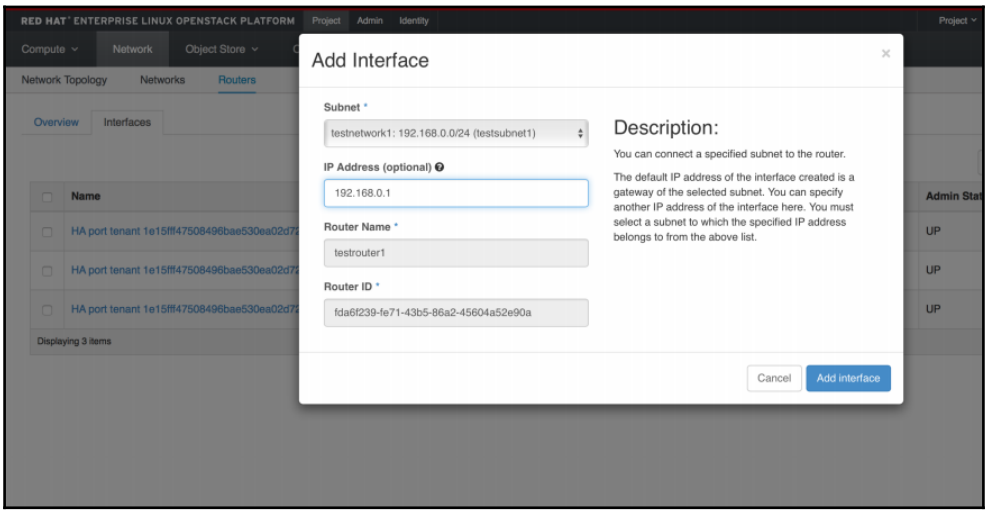
An external network will also need to be created to make the internal network accessible from outside of OpenStack, when external networks are created by an administrative user the set **External Network** check box needs to be selected as shown in the next screenshot:



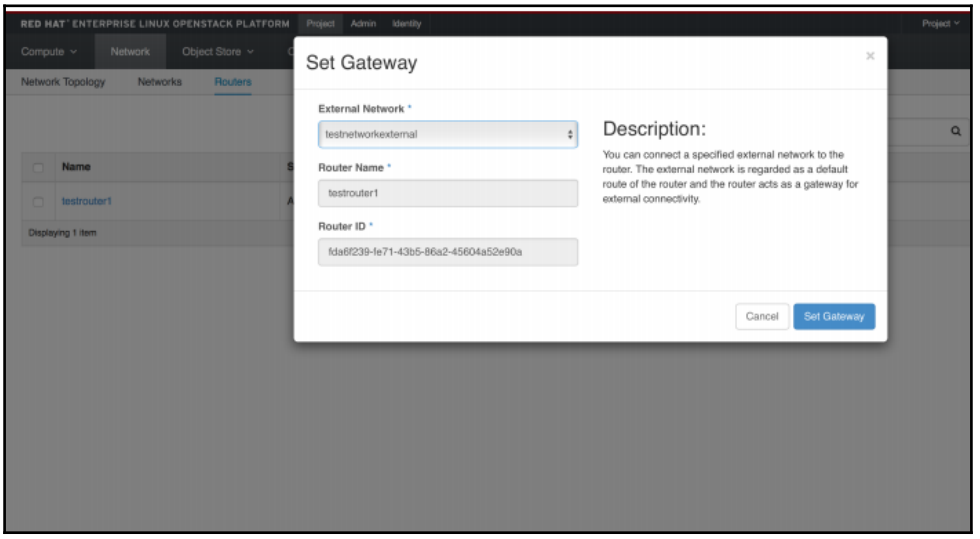
A router is then created in OpenStack to route packets to the network as shown in the following figure:



The created router will then need to be associated to the networks; this is achieved by adding an interface on the router for the private network as shown below:

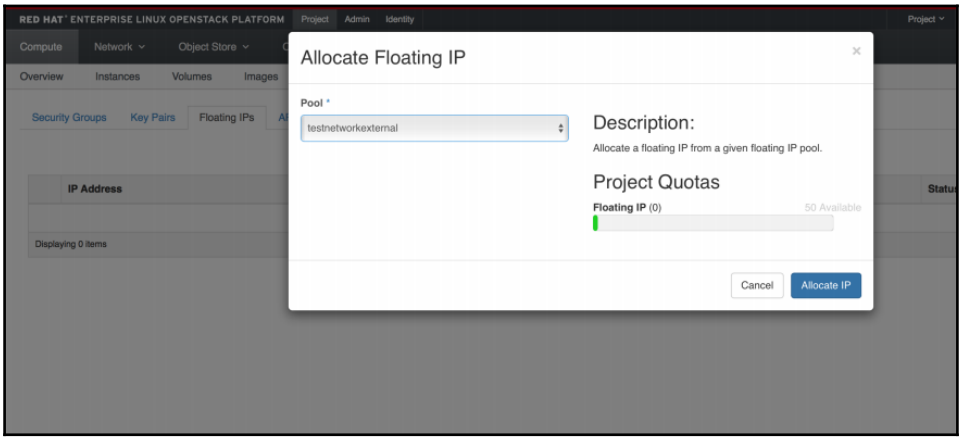


The external network that was created then needs to be set as the routers gateway as seen in the following figure:

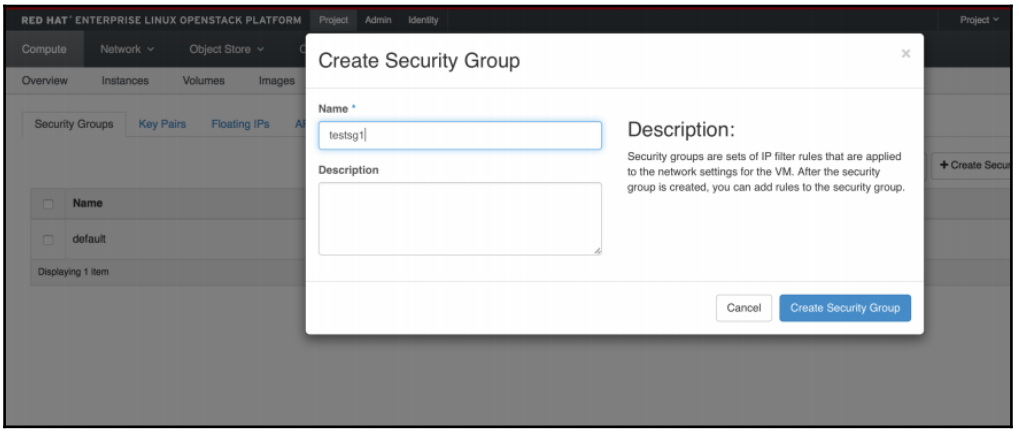


This then completes the network set-up; the final configuration for the internal and external network can be found below, which shows one router connected to an internal and external network seen in the following figure:

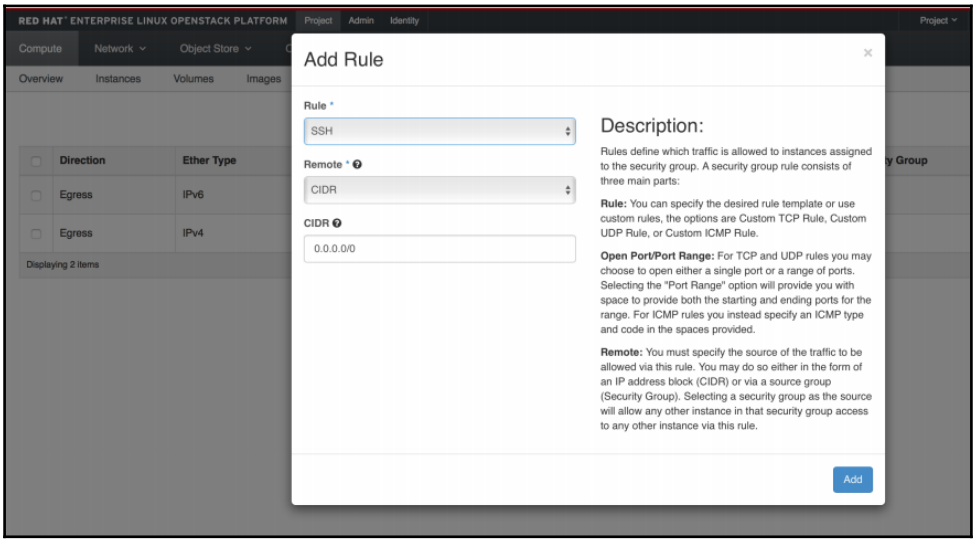
In OpenStack instances are provisioned onto the internal private network by selecting the private network as the NIC when deploying instances. OpenStack has the convention of assigning pools of public ips (floating ip) addresses from an external network for instances that need to externally routable outside of OpenStack. To set up this set of floating ips an OpenStack administrator will set up an allocation pool using the external network from an external network as shown in the following figure:



OpenStack like AWS utilises security groups to setup ACL, firewall rules between instances. Unlike AWS OpenStack supports both ingress and egress ACL rules, whereas AWS allows all outbound communication, OpenStack can deal with both ingress and egress rules. Bespoke security groups are created to group ACL rules as shown below:

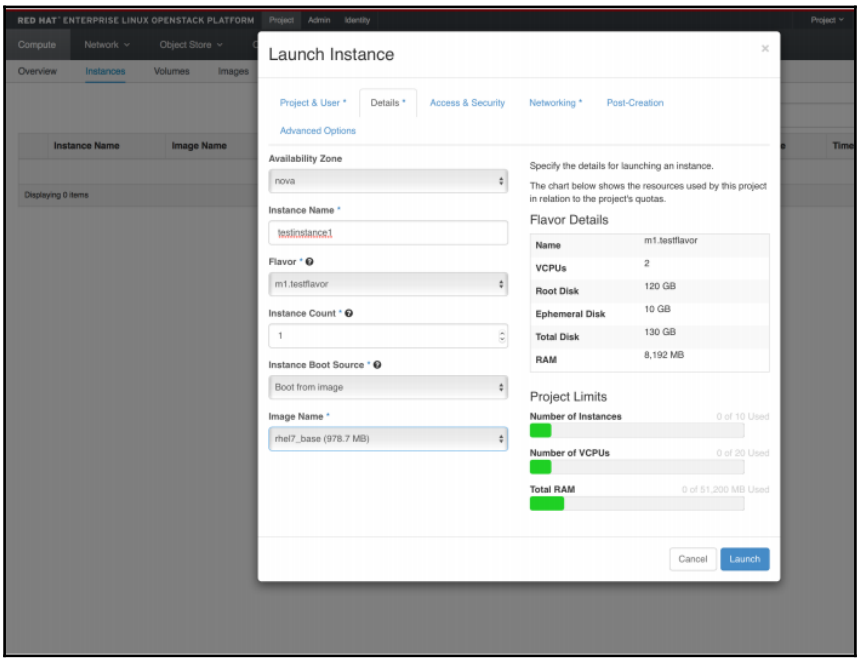


Ingress and Rules can then be created against that security group, below an ssh access is configured as an ACL rule against the parent security group which are pushed down to Open VSwitch into kernel space on each hypervisor as seen in the next screenshot:

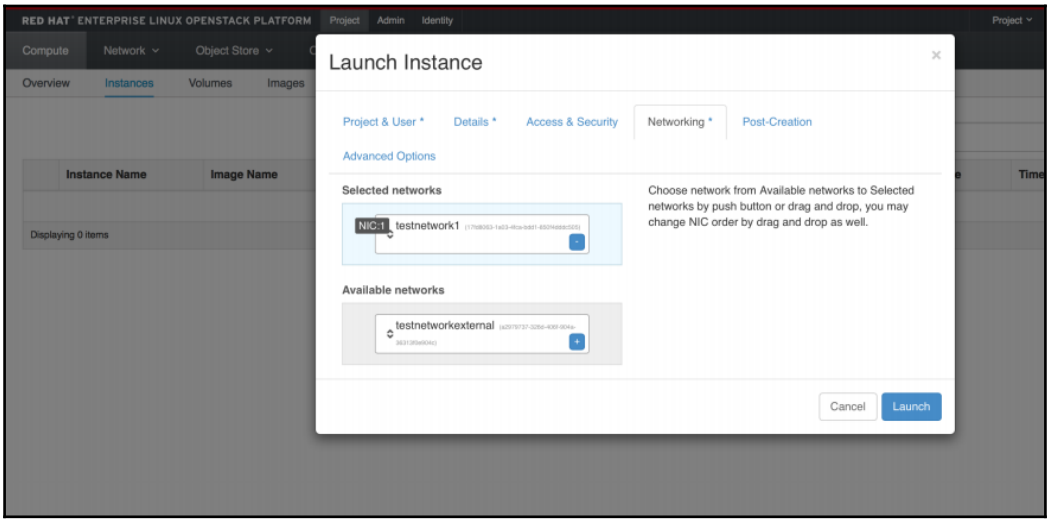


Once the Project (tenant) has two networks, one internal and one external and an appropriate security group has been configured, instances are ready to be launched on the private network.

This is done by selecting **launch instance** in horizon, an **Availability Zone**, **instance name**, **flavour** (CPU, RAM and disk space) are selected alongside an image (base operating system) seen in the following figure:

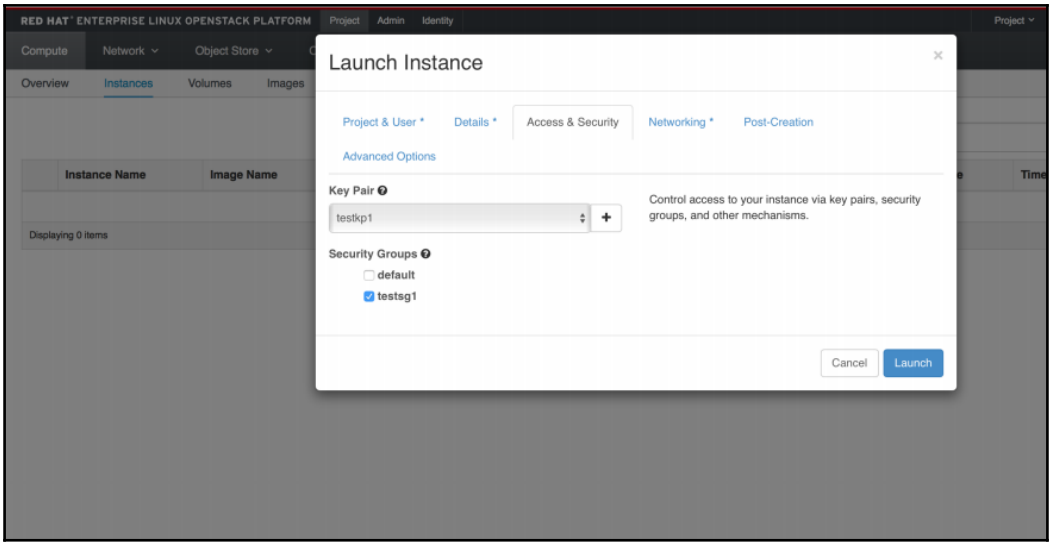


The private network is then selected as the NIC for the instance under the **Networking** tab as shown below:

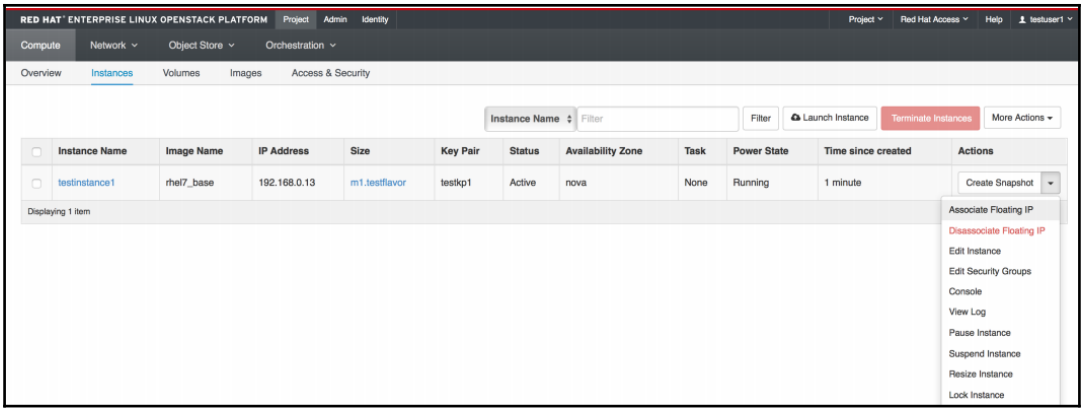


This will mean when the instance is launched it will use OpenStacks internal DHCP service to pick an available ip address off of the allocated subnet range.

A security group should also be selected to govern the ACL rules of the instance, in this instance the testsg1 security group is selected shown in the following figure:

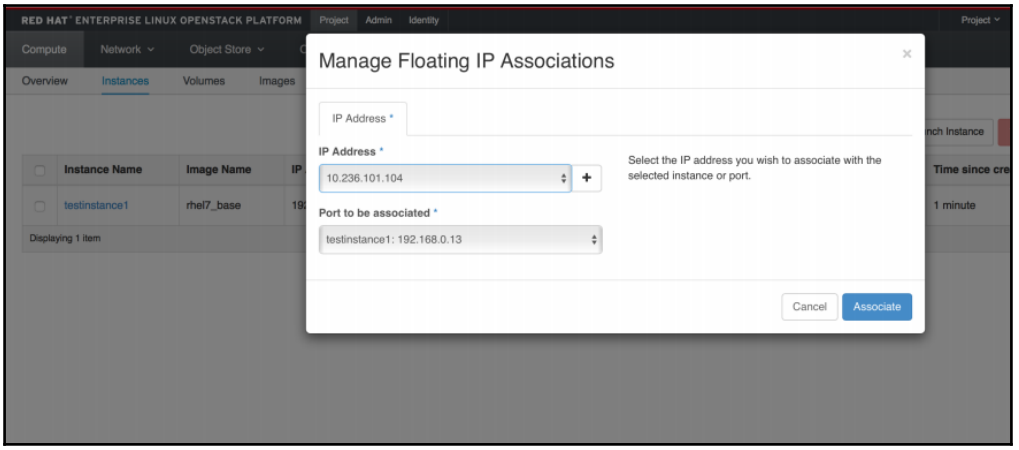


Once the instance has been provisioned then a floating ip address can be associated from the external network:



A floating ip address from the external network floating ip address pool is then selected

and associated with the instance:



The floating ip addresses https://en.wikipedia.org/wiki/Network_address_translation OpenStack instances that are deployed on the internal public ip address to the external networks floating ip address, which will allow the instance to be accessible from outside of OpenStack.

OpenStack like AWS, as seen on instance creation, also utilises regions and availability zones and compute hosts in Openstack (Hypervisors) can be assigned to different availability zones.

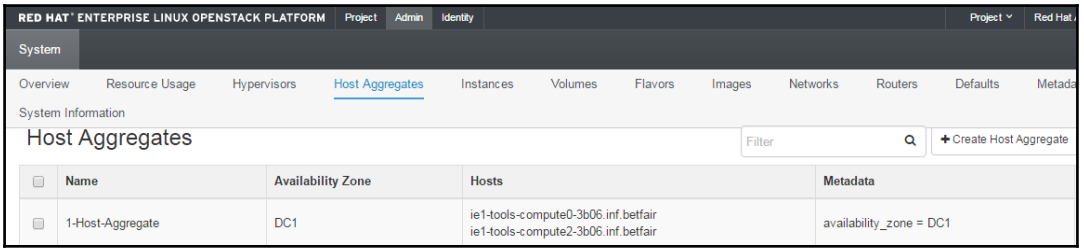
An availability zone in OpenStack is just a virtual separation of compute resources. In OpenStack, availability zone can be further segmented into host aggregates, it is important to note that compute host can be assigned only to one availability zone, but can be a part of multiple host aggregates in that same availability zone.

Nova uses a concept called **nova scheduler rules** which dictate the placement of instances on compute hosts at provisioning time. A simple example of a nova scheduler rule is the AvailabiltyZoneFilter filter which means that if a user selects an availability zone at provisioning time, then the instance will land only on any of the compute instances grouped under that availability zone.

Another example of AggregateInstanceExtraSpecsFilter filter that means that if a custom flavour (CPU, RAM and Disk) is tagged with a key value pair and a host aggregate is tagged with the same key value pair then if a user deploys with that flavour the AggregateInstanceExtraSpecsFilter will place all instances on compute hosts under that host aggregates.

These host aggregates can be assigned to specific teams, which mean that teams can be selective about what applications they share their compute with and can prevent noisy neighbour syndrome. There is a wide array of filters that can be applied in OpenStack in all sorts of orders to dictate instance scheduling utilising traditional cloud with large groups of contended compute to more bespoke models where ring fencing and isolation of compute resources is required.

An example of a host aggregates are shown below which groups. The example shows a host aggregate called **1-Host-Aggregate**, grouped under an **Availability Zone** called **DC1** containing two compute hosts (hypervisors) which could be allocated to a particular team:



When an instance is provisioned in OpenStack the following high level steps are carried out:

- Nova compute service will issue a request for a new instance (virtual machine) using the image selected from the glance images service
- The nova request may then be queued by rabbitmq before being processed (rabbitmq allows OpenStack to deal with multiple simultaneous provisioning requests)
- Once the request for a new instance is processed the request will write a new row into the nova Galera database in the nova database
- Nova will look at the nova scheduler rules defined on the OpenStack controllers use those rules to place the instance on an available compute node (KVM hypervisor)
- If an available hypervisor is found that meets the nova scheduler rules then the provisioning process will begin
- Nova will check if image already exists on the matched hypervisor, if it doesn't the image will be transferred from the hypervisor and booted from local disk
- Once nova will issue a neutron request which will create a new vport in OpenStack and map it to the neutron network
- The shared vport will then be written to both the nova and neutron databases in

Galera to correlate the instance with the network

- Neutron will issue a DHCP request to assign the instance a private ip address from an unallocated ip address from the subnet associated the selected network.
- A private ip address will then be assigned and the image will start to start up on the private network
- The neutron metadata service will then be contacted to retrieve cloud init information on boot which will assign a DNS entry to the instance from the named server, if specified
- Once cloud-init has run the instance will be ready to use
- Floating ips can then be assigned to the instance to nat to external networks to make the instances publically accessible

Like AWS OpenStack also offers a **Load-Balancer-as-a-Service** (**LBaaS**) option that allows incoming requests to be evenly among designated instances using a virtual ip (VIP). The features and functionality supported by LBaaS are dependent on the vendor plug-in that is utilised.

Popular LBaaS plugins in OpenStack are:

- Citrix Netscaler
- F5
- HaProxy
- AVI networks

These load balancers all expose varying degrees of features to the Openstack LBaaS agent. The main driver between using LBaaS on OpenStack is that it allows users to utilise LBaaS as a broker to the load balancing solution, allowing users to use the OpenStack API or configure the load balancer via the horizon GUI.

LBaaS allows load balancing to be set-up within a tenant network in OpenStack. Utilising LBaaS means that if for any reason a user wishes to utilise a new load balancer vendor as opposed to their incumbent one, as long as they are using OpenStack LBaaS this is made much easier. As all calls or administration are being done via the LBaaS APIs or Horizon, no changes would be required to the orchestration scripting required to provision and administrate the load balancer and they wouldn't be tied into each vendors custom APIs and the load balancing solution becomes a commodity.

Summary

In this chapter we have covered some of the basic networking principles that are used in today's modern data centres, with special focus on the AWS and OpenStack cloud technologies which are two of the most popular solutions.

Having read this chapter you should now be familiar with the difference between spanning tree and leaf spine network architectures, it should have demystified AWS networking and you should now have a basic understanding of how private and public networks could be configured in OpenStack.

In the forthcoming chapters we will build on these basic networking constructs and look at how they can be programmatically controlled using configuration management tools and used to automate network functions. But first we will focus on some of the software defined networking controllers that can be used to extend the capability of OpenStack even further than neutron in the private clouds and some of the feature set and benefits they bring to ease the pain of managing network operations.

2

The Emergence of Software Defined Networking

This chapter we will detail the emergence of open protocols that have helped aid software defined networking (SDN) solutions. It will focus specifically on the Nuage VSP SDN solution and look at some of the scaling benefits and features this provides over and above the out the box experience from AWS and OpenStack. It will articulate why these networking solutions have become a necessity for notoriously complex private cloud networks, by simplifying networking using software constructs while aiding automation of the network by providing a set of programmable API's and SDK's.

This chapter will focus on the following topics in detail:

- Current SDN solutions on the market
- How the Nuage SDN solution works
- Integrating OpenStack with the Nuage VSP Platform
- The Nuage VSP Software Defined Object Model
- How the Nuage VSP can support Greenfield and Brownfield Projects
- The Nuage VSP Multicast Support

Current SDN solutions on the market

Software Defined Networking (**SDN**) solutions from vendors are made up of a centralised controller that is implemented to become the nerve centre of the network. SDN controllers rely heavily on OVSDB and OpenFlow open protocols, integrating directly with switches to route packets in the network as well as applying policy. As long as a switch can talk OVSDB and OpenFlow then it can integrate with common SDN Controllers.

There are now a wide variety of software defined networking controllers currently on the market:

- CISCO ACI
- Nokia Nuage VSP
- Juniper Contrail
- VMware NSX
- Open Daylight
- MidoNet Midokura
- Brocade

The aim of SDN controllers is to provide an easy to use solution for network functions, with the SDN controllers abstracting the network functions from hardware devices and instead exposing a set of APIs and graphical user interfaces (GUIs) that can be programmatically altered to control standard network operations.

It is a little doubtful that the emergence of AWS and its easy to use and simplified network functionalities have influenced network vendors to adapt their solutions. Vendors have now adopted and implemented open protocols to allow centralised management of network functions and the ability for end users to be able manage the whole network using the SDN controller.

One of the main use cases for software defined networking is the private data centre space and the emergence of OpenStack has meant that SDN solutions are now becoming more common.

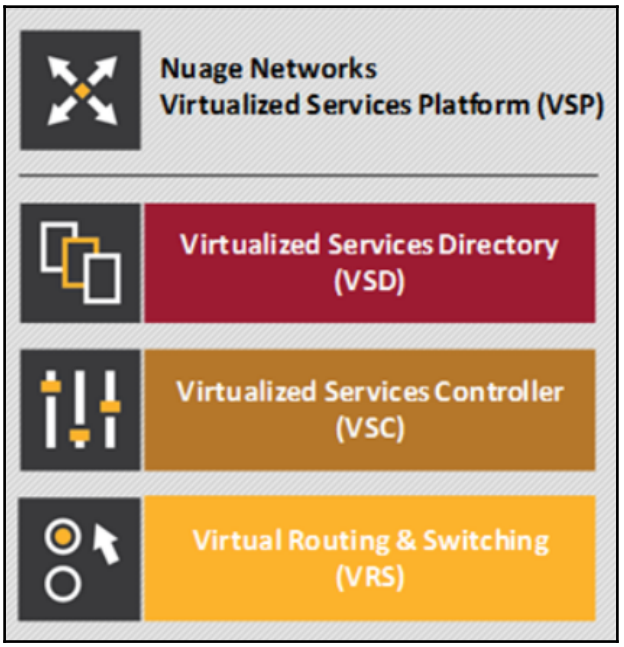
How The Nuage SDN solution works

One of the market leading SDN solutions is the Nuage SDN (VSP) platform which is Nokia's SDN solution (formerly Alcatel Lucent).

The Nuage VSP platform is made up of three main components the VSD, VSC and VRS

- **Virtualized Service Directory (VSD):** Is the policy engine for the overall platform and provides a graphical user interface and exposes a restful API for network engineers to use and interact with network functions.
- **Virtualized Service Controller (VSC):** Is the SDN controller for Nuage and uses OpenFlow and OVSDB management protocol to distribute switching and routing information to hypervisors, bare metal servers or containers.
- **Virtual Routing and Switching (VRS):** This is Nuage's customized version of

Open vSwitch which is installed on compute nodes (hypervisors)



Nuage VSP can integrate with OpenStack, CloudStack and VMWare cloud platforms. The Nuage VSP Platform creates an overlay network that has the ability to secure virtual machines, bare metal servers and containers (Docker and Kubernetes) in an isolated tenant network, so it is highly flexible. Nuage also supports multicast between tenant networks by routing multicast traffic via hypervisors on the underlay and flooding it to specific instances within a tenant network.

The Nuage VSPs SDN Controller (VSC) integrates with switches using OVSDB via hardware VTEPs exposed by switches at the access layer of the network. VSCs are deployed redundantly and communicate with each other with Multipath Border Gate Protocol (MP-BGP) and program VXLAN encapsulation to the switches as they are hardware VTEP aware.

The VSD component is set-up in an active cluster containing three VSD servers, which are load balanced using HAProxy or any other viable load balancer solution. The HAProxy load balancer provides a virtual ip (VIP) which load balances three VSDs servers in round-robin mode.

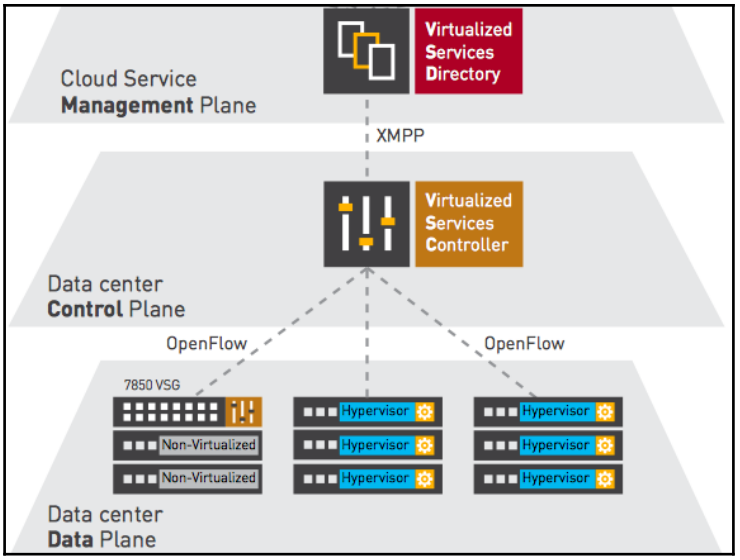
The VSDs VIP exposes the graphical user interface (GUI) for the Nuage VSP platform and

API entry point to programmatically control the overlay network using REST calls. Any operation carried out on the Nuage VSD GUI initiates a REST API call to the VSD, so both the GUI and the REST API are carrying out identical programmatic calls and all operations are exposed via the REST API.

The Nuage VSD governs layer 3 domains, zones, subnets and ACL policies. The VSD communicates policy information to the VSC using XMPP and the VSC uses OpenFlow to push down flow information to a customised version of Open vSwitch (VRS) on the compute hosts (hypervisor).

Nuage VSP allows bare metal servers to be connected to overlay networks too by pushing down Open Flow Data to the Virtualised Services Gateway (VSG) and leaking routing information into the overlay network.

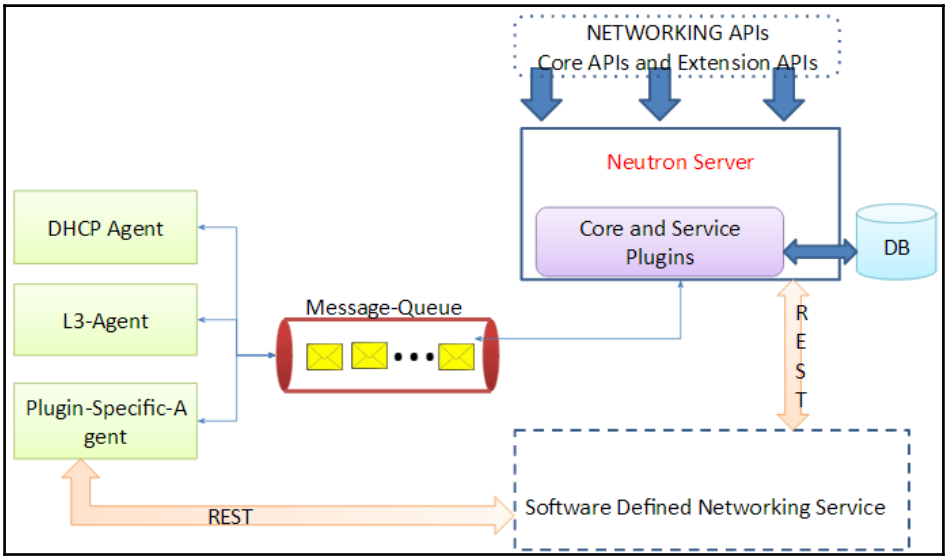
An overview of the VSP platform protocol integration can be found below:



Integrating OpenStack with the Nuage VSP Platform

Traditionally private data centre networks can be very complex, so utilising vanilla OpenStack neutron to meet all use cases may not provide all the features that are required yet, although the features in neutron is maturing very quickly with every new OpenStack release so they are as feature rich as dedicated SDN controllers.

Neutron lends itself to integration with SDN controllers by providing a REST API extension, so SDN controllers can easily be used to extend the base networking functions provided by neutron if required to provide a very rich set of features. The use of SDN solutions have helped OpenStack to scale massively, as it moves the networking aspect of OpenStack away from the centralised rabbitmq message queuing and instead requests are moved to the dedicated SDN controllers. This means that one OpenStack cloud can potentially scale the amount of compute instances that are supported horizontally, without having to worry about bottlenecks or scaling issues associated with the current neutron network architecture.

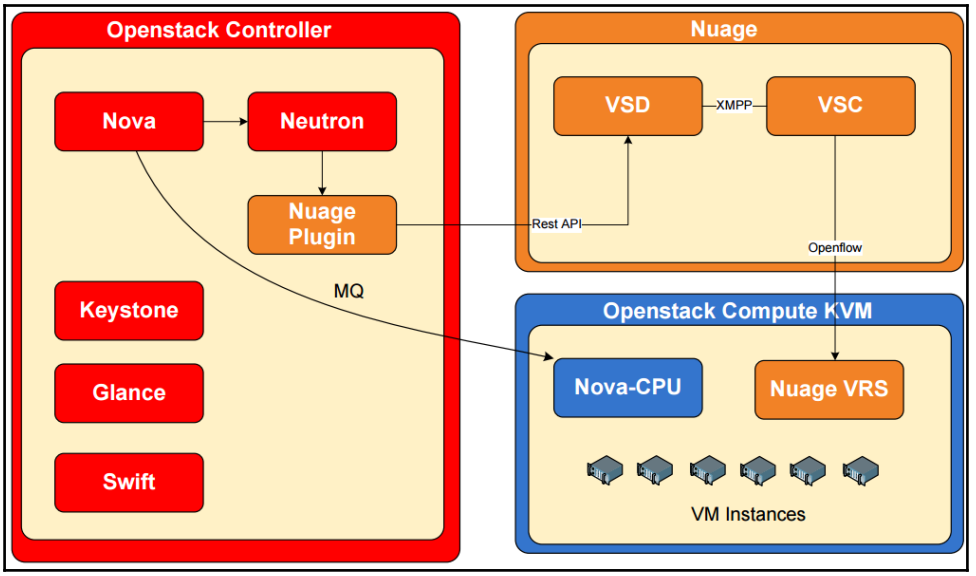


OpenStack is one of the most popular private cloud solutions and the Nuage VSP platform integrates with OpenStack using the Nuage plug-in. The Nuage plug-in is installed on each of the Highly Available (HA) OpenStack controllers. The Neutron ML2 and L3 agents are both switched off on the Controllers in favor of a Nuage plug-in.

Nuage integrates with OpenStack by setting up a net partition; one Nuage VSP can be mapped to multiple instances of OpenStack via the use of net-partitions. Net partitions are a way of telling the Nuage VSP platform which OpenStack instance to map it's subnets to and wait for vport commands, which signify that OpenStack nova has provisioned a new instance which needs to be governed by policy.

When Nuage VSP is integrated with OpenStack, OpenStack vendor installers need to either support Nuage natively or the installer will need to be customized slightly to install the Nuage plug-in on OpenStack Controllers. The Nuage OpenvSwitch (VRS) also needs to be installed on each compute node (hypervisor) that is deployed.

The Nuage Plug-in integrates with the OpenStack Controllers and KVM Compute using the following workflow:



When a Neutron command is issued to OpenStack, the Nuage plug-in uses REST API calls to communicate with the Nuage VSD to say that a new network has been created or a new vport on that network has been created, this is possible due to Neutrons SDN controller pluggable REST API architecture.

The VSD then communicates with the VSC to push flow data using Extensible Messaging and Presence Protocol (XMPP). The VSC (SDN Controller) then administers flow data (OpenFlow) to the Nuage VRS (Open vSwitch) and the Nuage VRS is used to secure OpenStack instances with firewall policies. Firewall policies can either be Security Groups

or Nuage ACL rules depending if OpenStack managed mode or Nuage VSD managed mode are selected.

Nuage or OpenStack Managed Networks

The Nuage OpenStack plugin can be used in two modes of operation to manage networks that are provisioned in OpenStack.

- VSD managed mode.
- OpenStack managed mode

VSD-managed mode, allows Nuage to take control of OpenStack networks, the rich set of features provided within the Nuage VSP platform are used as the default for all networking functions. Network functions are provisioned directly via the VSD using the Nuage REST API by the GUI or direct API calls.

OpenStack-managed mode requires no direct provisioning on the VSD, all commands are issued via neutron, however, functionality is limited to the commands OpenStack Neutron supports.

All networks that are created in Nuage are replicated in OpenStack in a one to one mapping with the Nuage VSD being the master in VSD managed mode, while OpenStack Neutron is the master of configuration in OpenStack-managed mode.

In OpenStack managed mode all ACL rules are governed by OpenStack Security Groups while in VSD managed mode ACL rules are held instead in the Nuage VSD with security groups disabled.

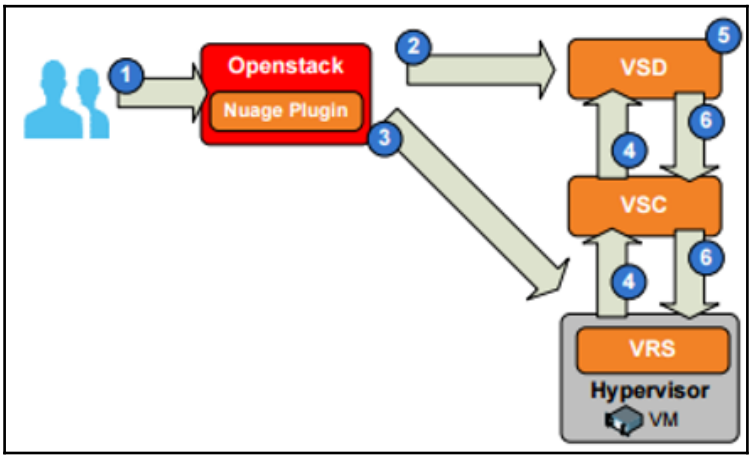
Nuage integrates with OpenStack by setting up a net partition; one Nuage VSP Platform can be mapped to multiple instances of OpenStack via the use of net-partitions. Net partitions are a way of mapping a Nuage layer 3 subnet and organization to a neutron subnet.

A Nuage VSP platform with an organization called *Company*, whenever a subnet is created under the organization, it is subsequently assigned a unique nuage_subnet_uuid on creation. In order to map the organization and Nuage subnet to OpenStack Neutron the following command is issued:

```
neutron subnet-create "Subnet Application1" 10.102.144.0/24 --nuagenet
nuage_subnet_uuid --net-partition "Company" --name "Subnet Application1"
```

Once a net partition has been established by the Nuage VSP Platform and OpenStack, the firewall policies are secured at the compute host (hypervisor) using the Nuage VRS. The

following workflow is triggered when a new instance is created on a Nuage managed subnet:



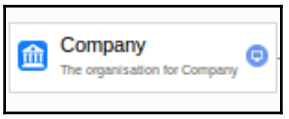
1. An instance is added to an OpenStack network and subnet owned by the Nuage VSP.
2. A placeholder Vport is created (VM id, MAC) on the VSD by the Nuage Plugin, within the requested layer 3 domain.
3. Nova service creates the VM on the Hypervisor. This is detected by the VRS (VM id, MAC).
4. The VRS queries the VSC, the VSC then queries the VSD in order to retrieve the associated networking information from the placeholder vport.
5. The VSD matches the VM id, MAC against the vport it created and associates the VM to the correct network services.
6. The policy is downloaded from the VSD through the VSC to the VRS using Open Flow and the required flows are dynamically created.

The Nuage VSP Software Defined Object Model

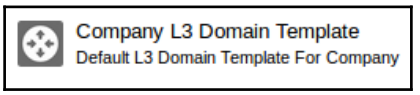
As Nuage creates the overlay network in software, it uses a simple object model to do this, the Nuage VSP software defined object model provides a graphical hierarchy of the network meaning that the structure of the overlay can be easily viewed and audited.

Object Model Overview:

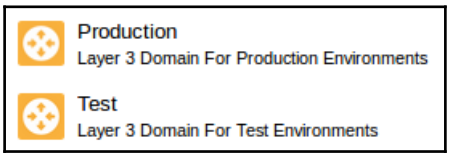
Organisation: Governs all Layer 3 domains



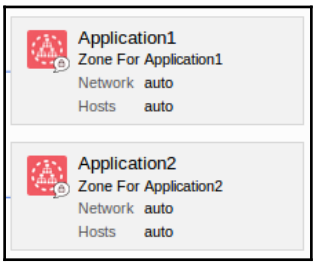
Layer 3 domain Template: A layer 3 domain template is required before child layer 3 domains are created. The Layer 3 domain template is used to govern overarching default policies that will be propagated to all child layer 3 domains. If a layer 3 domain template is updated at template level then the update will be implemented on all Layer 3 domains that have been created underneath it immediately.



Layer 3 domain: Can be used segment different environments so users cannot hop from subnets deployed in a under a layer 3 **Test** domain to a layer 3 **Production** domain.

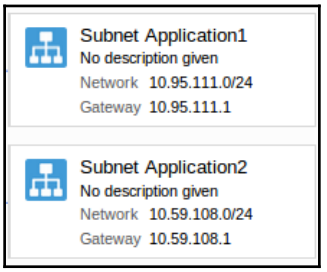


Zones: A zone segments firewall policies at application level, so each micros-service application can have its own zone and associated ingress and egress policy per Layer 3 Domain.



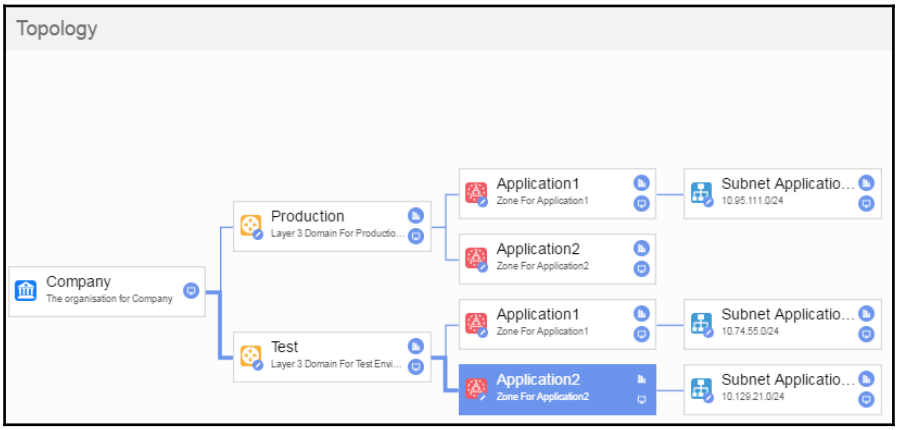
Layer 3 Subnet: This is where VMs or bare metal servers that are deployed through

In this example we see **Subnet Application1** and **Subnet Application2**



The hierarchy in Nuage VSD is shown below is as follows:

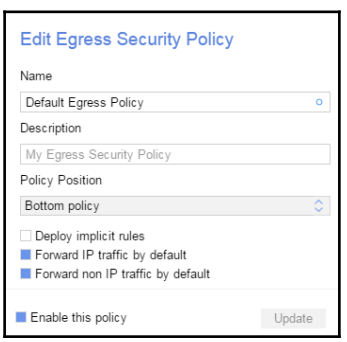
- One organisation has been created called **Company**.
- Two layer 3 domains called **Test** and **Production** have been created underneath the **Company**.
- The **Test** layer 3 domain has a zone for **Application1** and **Application2** with 1 child subnet underneath the Application1 and Application2 zones.
- The **Production** layer 3 domain has a zone for **Application1** and **Application2** with 1 child subnet underneath the **Application1** only. **Application2** zone does not have a child subnet yet.



For security and compliance purposes demonstrating to auditor's segmentation between **Development** and **Production** environments is very important, traditionally **Development** environments do not having the same stringent production controls applied. Isolation of production applications is done using the convention of least privilege possible is normally a security requirement to minimise access and reduce the probability of a security breach.

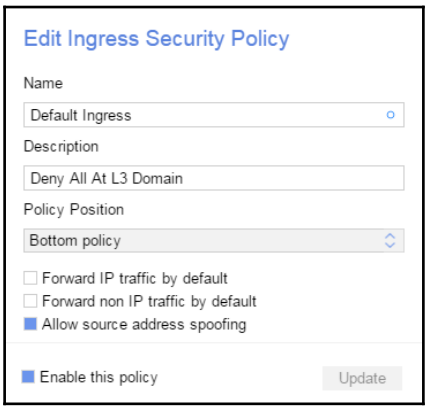
Nuage VSP Platform can set-up segregation between environments using its layer 3 domain template construct, a domain template can be set-up with a default **Deny All** policy at ingress and egress level. This is given the highest priority of all the policies and will explicitly drop all packets no matter the protocol for inbound and outbound connections, unless explicitly allowed by the policy for that specific application. The default **Deny All** is the bottom policy on the list of ACL rules applied to an application.

The explicit drop on egress security policy domain template is shown below as the bottom policy:



While the contents of the egress security policy are shown below with the highest possible priority:

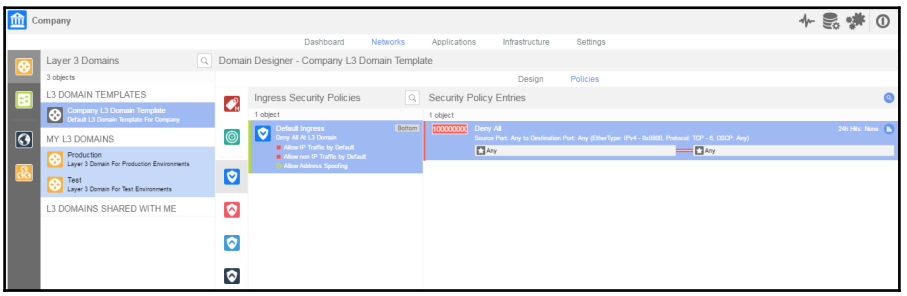
Likewise the explicit drop on ingress is applied to the domain template as the bottom policy:



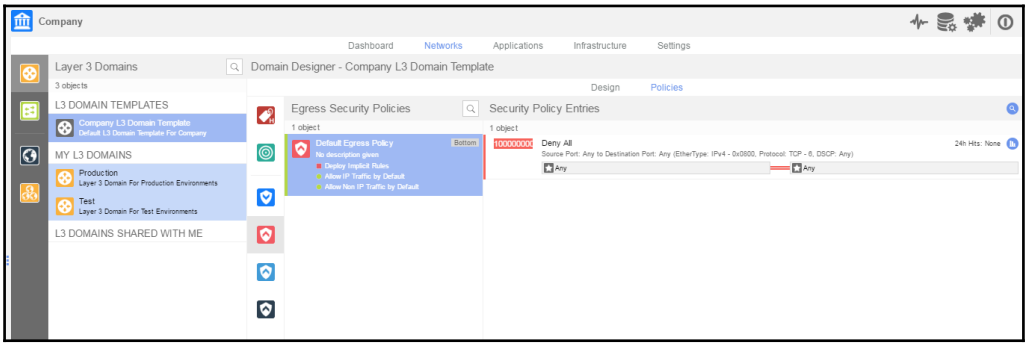
While the explicit drop on the ingress security policy on the domain template is shown below:

The default ingress and egress polcies aplied to the **Company L3 Domain Template** are shown below, which shows the policy applied to all the child layer 3 domains, in this instance **Production** and **Test**.

The domain template **Company L3 Domain Template** is shown to be linked to the child layer 3 domains **Production** and **Test showing the inherited egress policy from the domain template**:



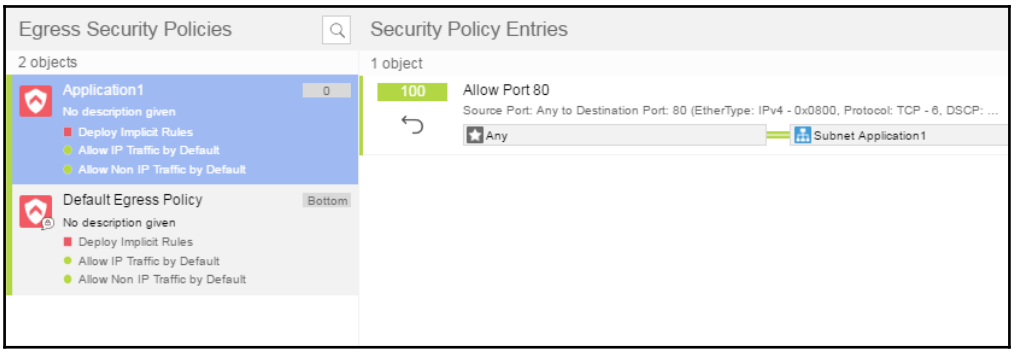
Likewise the domain template **Company L3 Domain Template** is shown to be linked to the child layer 3 domains **Production** and **Test showing the inherited ingress policy from the domain template**:



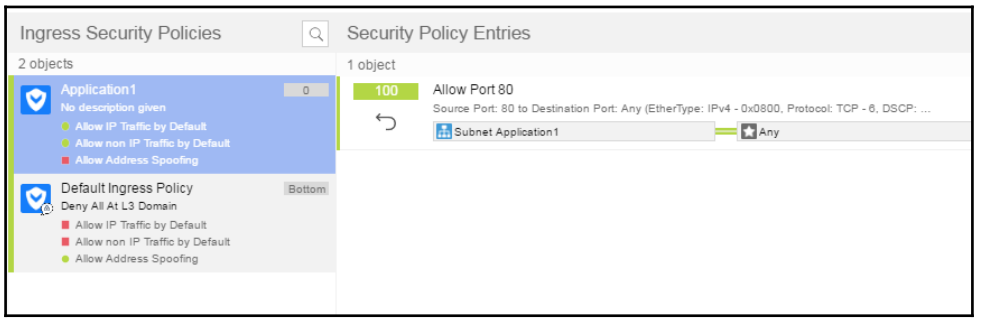
It is important to note that as policies are pushed down to the VRS using OpenFlow, ACL rules for ingress and egress in Nuage work on the principle that:

- **Egress:** is a packet flowing from the VRS to the subnet or zone.
- **Ingress:** is a packet flowing from the subnet or zone to the VRS.

As an example an Egress ACL rule will specify that any egress traffic coming from the VRS from port 80 will be forwarded to the **Subnet Application1**:



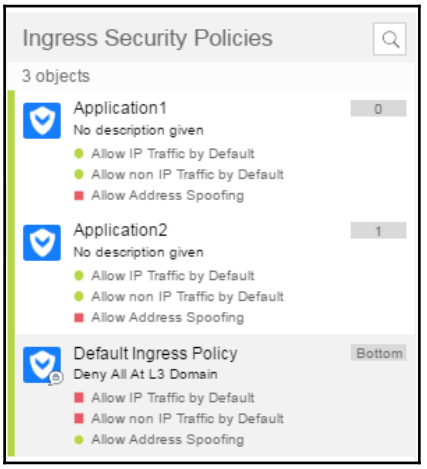
In this example an Ingress ACL rule will specify that any ingress traffic can leave the **Subnet Application1** on port 80 will be forwarded to the VRS:



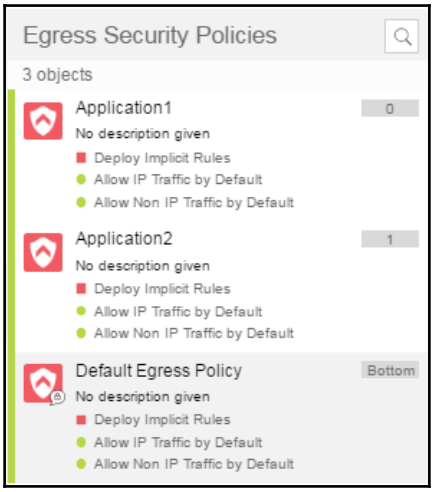
If application owners work on the principle that their layer 3 subnet, where their application is deployed on, is always specified in an ACL rule as either the source or destination in their individual application policy then ACL rules for an application will only exist in that self-contained policy. If this concept is adhered to it allows ACL rules for each application to be encapsulated in separate policies, within a layer 3 domain, which in turn means auditing them is much simpler for security teams. It also means that applications support least privilege, meaning only necessary ports are opened so applications can communicate with an explicit drop applied to anything outside those rules.

Two policies are shown for two applications **Application1** and **Application2** are shown which have separate policies for Ingress and Egress, with the **Default Ingress Policy** specifying the explicit drop all for any flows not explicitly allowed.

The ingress security policies are shown here:



While the egress security policies are shown here:



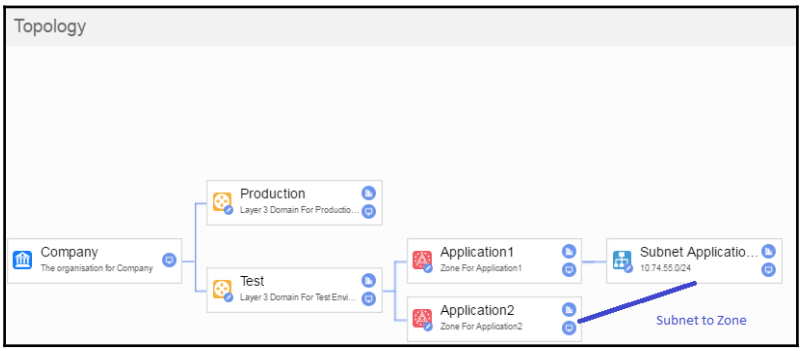
Nuage VSP Platform Layer 3 Domain templates allow a second level of segmentation using zones, so traditionally networks were split into three zones, where applications would be deployed:

- Front End
- Business Logic
- Backend

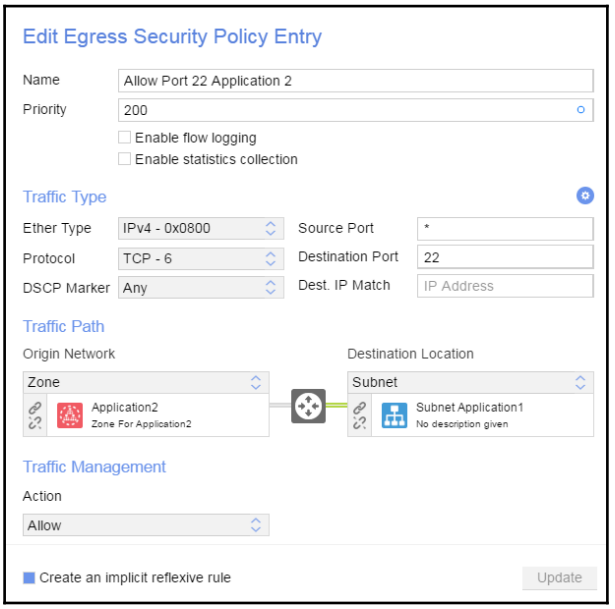
As micro-service architectures have grown to prominence, each applications profile doesn't always fit into these three broad profiles. Sometimes applications can be both a Front End application and contain Business Logic too, so where the micro-service application would be placed in the traditional three tiered structure?

Instead of the Front End, Business Logic and Backend segregation policies can be applied at zone level, meaning micro-segmentation of applications is possible between each subnet in Nuage. So how does this translate to Nuage?

If an application wishes to talk to another application it will have an ACL rule that specifies Subnet to Zone communication for east to west communication between applications sitting on adjacent subnets in a layer 3 domain. Nuage allows this by allowing applications to talk Subnet to Zone



To allow this communication **Application1** could have an ACL policy to allow **Application2** zone to allow traffic to flow into the subnet on port 22 allowing east to west communication, so no matter how many different subnets are used then **Application1** will always be allowed to talk to any applications sitting under the **Application2** zone:



In terms of security policies this allows development and security teams to understand which applications are talking to each other and the ports they are using by reviewing the application policy:

How The Nuage VSP Platform can support Greenfield and Brownfield Projects

Overlay networks are typically set-up as new network (greenfield) sites but a completely new network in isolation is not useful, unless there is a planned big bang migration of all applications from the legacy network to the new network in a single migration.

If instead a staged application migration is chosen, where only a percentage of applications are migrated to the overlay, then the new overlay network will need to communicate with the legacy network and require to operate in a brownfield set-up.

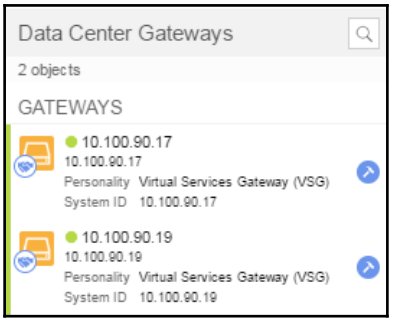
A brownfield set-up normally means applications are migrated in stages to the new platform, as opposed to all in one go, which builds confidence in the new network and new technology associated with that network. A staged deployment will typically involve performance testing the migrated applications in the new network, prior to throttling live traffic away from the incumbent legacy network to the migrated application in the new overlay network.

A major requirement for a staged migration is connectivity back to the legacy network for application dependencies that are still deployed on the legacy network, so migrated applications can operate effectively.

The Nuage VSP Platform uses its Virtualised Service Gateway (VSG) to provide the connectivity between the new overlay and legacy network. A pair of VSGs is connected redundantly in virtual chassis mode and connects to interfaces on routers sitting in the legacy network. The VSG performs a route table lookup based on the destination IP of a packet coming in on its VLAN from the attached router interface, it then updates the destination MAC with the next hop address and forward the packet on the corresponding

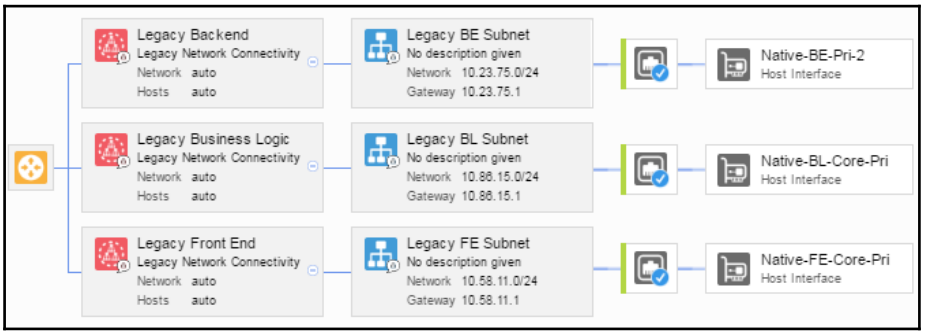
VXLAN segment. All packets are routed from the legacy network to the VSG via an underlay network. This bridges the new overlay network and the legacy network with VXLAN is being terminated on the VSG.

The pair of active VSGs is shown below in the Nuage VSD:



The VSG allows communication with the legacy network by leaking routes to the overlay network. Each VSG will receive and advertise IPv4 routes using a BGP session, this BGP session will be established between the VSG, the VSC and leaf switch when utilising a leaf spine topology using iBGP. The VSG must advertise its local system IP to legacy routers in the legacy network and all routes received from the native network will then be subsequently leaked from the native network via the underlay network into selected layer 3 domains in the overlay.

The set-up required to leak routes in the Nuage VSP Platform is the creation of a GRThubDomain layer 3 domain. In this example host interfaces are connected into the Front End, Business Logic and Back End routers in the legacy network:

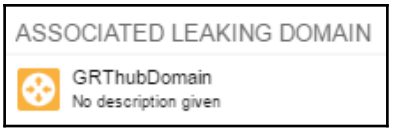


The Nuage VSP platform then allows the newly created **GRThubDomain** to be associated

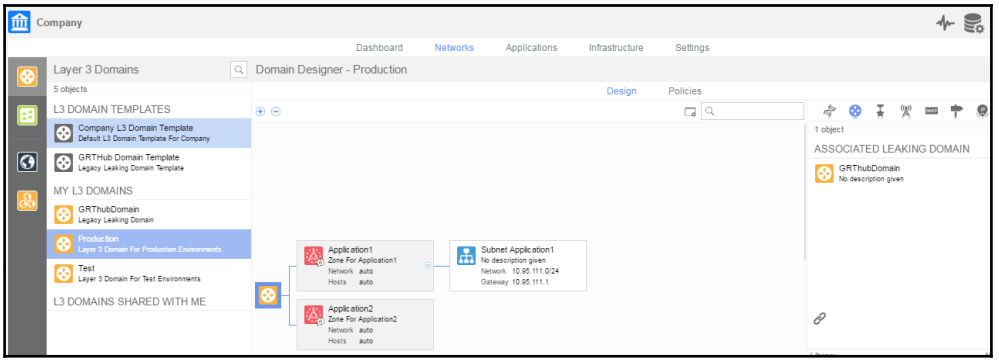
with the **Production** or **Test** layer 3 domains by associating a leaking domain against them.

In the below example the **GRThubDomain** is associated with the **Production** layer 3 domain.

The leaking domain in the Nuage GUI is displayed using the following icon showing a leaking Domain called **GRThubDomain**:



The Production domain with associated leaking domain is shown below in the Nuage GUI:



The association of a leaking domain allows the Nuage VSP Platform to leak routes into from the legacy network through to the new overlay network, meaning applications in the overlay network can communicate with applications in the legacy network, so long as they have appropriate ingress and egress ACL policies specified.

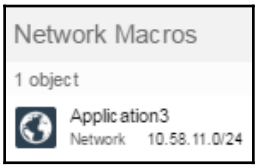
The **Test** and **Production** layer 3 domains as explained before, have a **Deny All** for ingress and egress as part of the **Company L3 Domain Template**. So although all routes are leaked into the overlay they are dropped by the VRS unless explicitly stated otherwise.

The Nuage VSP platform has the ability to apply ACL rules to the routes leaked from the external legacy network by utilising a concept called **network macros**. In the Nuage VSP Platform a network macro is simply a fancy name for an external network range.

If an application, **Application3** in this instance, resides in the legacy network and its routing has already been exposed by the **GRThubDomain** leaking domain and leaked into the **Test**

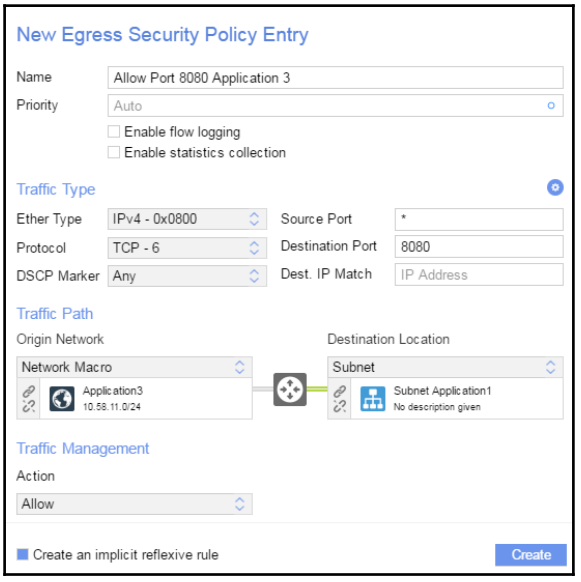
layer 3 domain, then a network macro can be set-up to describe the range required and isolate connectivity to it using a Nuage ACL rule.

In this instance the network range 10.58.11.0/24 is where **Application3** resides is part of the Front End range on the **GRThubdomain** that is leaked into the overlay network. The network macro for **Application3** created and shown below:



An egress ACL policy can then be configured to allow **Application1** to communicate with **Application3** by creating a network macro to subnet ACL rule, which allows the **Application3** network macro to connect to **Subnet Application1** on port 8080.

The egress security policy to allow communication between the Application3 network macro and Subnet Application 1 on port 8080 is shown below:

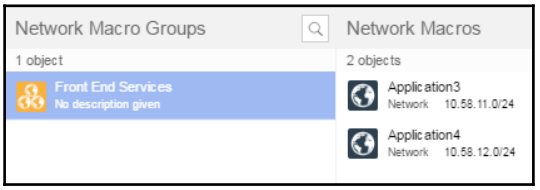


After creation the ACL list is updated to show the new network macro ACL:

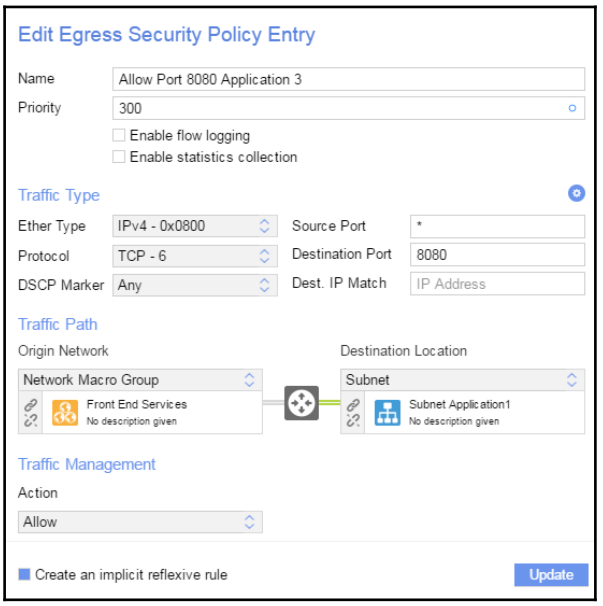
This allows the Nuage VSP to lock down policy and ACL rules on the legacy network with the same granularity that is enforced if an application resided within the same layer 3 domain. Network macros can also be used to route between multiple cloud technologies as well as different data centres so are a very powerful way of connecting networks and controlling policy between them.

Multiple network macros can be grouped together into a network macro group, which allows multiple ranges to be controlled by one ACL rule. These are then exploded out at OpenFlow level on the VRS at the hypervisor. Nuage currently has a limit of 100 ACL rules per vport, so only 100 ACL rules can currently be applied to a single instance (virtual machine) so it is important to be careful when grouping network macros.

An example of a network macro group can be shown below and then the **Front End Services** network macro group can be used in the egress ACL rule as opposed to specifying individual policies for **Application3** and **Application4**:



The egress security policy to allow port 8080 connection between the Front End Services network macro and Subnet Application1:



The applied ACL implementing the Front End Services network macro can be found below:

The Nuage VSP Multicast Support

The Nuage VSP Platform also has the ability to route multicast between layer 3 domains, zones, subnets or vports in the overlay network by configuring dedicated vlans on compute hosts (hypervisors) on the underlay with dedicated VLANs for multicast.

Multicast is the distribution of packets from one or more sources to a receiver destination so Nuage allows transmission and reception of broadcast and multicast traffic within a subnet with either no restriction or in a controlled fashion.

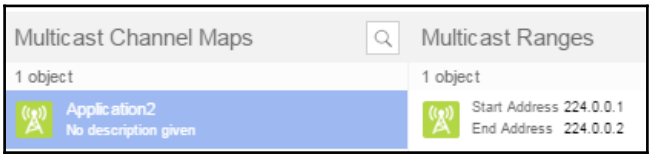
In a controlled scenario multicast port channels are used to group multicast ranges, which are then mapped to specific vports of an instance, this to access managed multicast streams that are routed from the data center's underlay infrastructure.

The Nuage VSD is used to restrict which multicast groups vports, subnets or zones multicast traffic can be flooded to. The multicast forwarding from the underlay switch uses a specific VLAN or group of VLANs to forward on multicast to the Nuage VRS on the hypervisor. The multicast streams are then replicated to the vports that are attached to specific virtual machines, to transmit multicast into the overlay in a controlled fashion.

The Nuage VSP Platform uses multicast channel map objects as shown below, which are associated with an organization, to dictate if multicast is flooded to the domain, zone, subnets or vports of instances.

In the example below a Multicast Channel map is used to create multicast ranges for **Application2** which broadcasts multicast, this will route multicast from **SubnetApplication2**, via the underlay VLAN on the hypervisor, to the Nuage VRS and then flood it into the **Subnet Application1**.

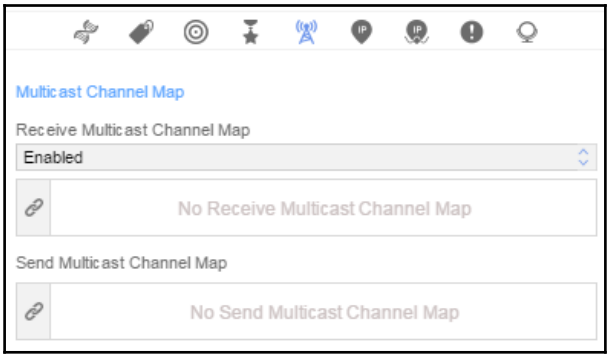
The multicast channel map icon is shown below:



While the multicast channel map is applied to the Subnet Application1:

Vports attached to instances can be used as either sender or receivers of multicast, if the application attached to the vport requires multicast traffic to operate.

The Receive and Send Multicast Channel Map option on the vport in the Nuage VSP is shown below:



Summary

In this chapter we have covered some of the advanced networking features provided by the Nuage VSP SDN solution and also touched upon some of the other SDN solutions that are available on the market. Having read this chapter you should now be familiar with the Nuage SDN controller and understand the rich set of features an SDN controller can bring to OpenStack and the private cloud.

Given the programmability SDN controllers, AWS and OpenStack solutions bring, we will now shift focus and look at the cultural changes that are necessary in organisations to make the most of these fantastic technologies. Implementing new technologies without changing operational models is not enough, people and process are key to a successful DevOps model.

The role of the network engineer is undergoing its biggest evolution in years so businesses cannot simply implement new technology and expect faster delivery without dealing with people and cultural issues. CTO's have a responsibility to set up their networking teams up for success by implementing DevOps transformations that include network functions and network teams also need to learn new skills such as coding to push forward automation using grass root initiatives.

3
Bringing DevOps to Network Operations

This chapter will switch the focus from technology to people and process. The DevOps initiative is about breaking down silos between teams and changing company's operational models. It will highlight methods to unblock IT staff and allow them to work in a more productive fashion. It will primarily focus on the evolving role of the network engineer, which is changing like the operations engineer before them, and the need for network engineers to learn new skills to survive as the industry moves towards a completely programmatically controlled data centre.

This chapter will look at two differing roles, that of the CTO / senior manager and engineer, discussing at length some of the initiatives that can be utilised to facilitate the desired cultural changes that are required to create a successful DevOps transformation for a whole organisation or even just allow a single department improve their internal processes by automating everything they do.

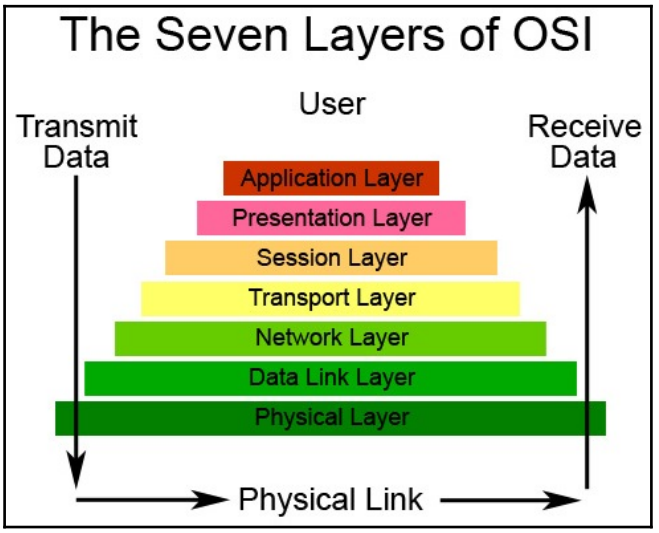
In this chapter the following topics will be covered:

- Initiating a Change in Behaviour
- Top Down DevOps initiatives for Networking Teams
- Bottom Up DevOps Initiatives for Networking Teams

Initiating a Change in Behaviour

The networking OSI model contains 7 layers, but it is widely suggested that the OSI model has an additional 8th layer called the **user** layer, which governs how end users integrate and interact with the network. People are undoubtedly a harder beast to master and manage

than technology so there is no *one size fits all* solution to the vast amount of people issues that exist.



Initiating cultural change and changes in behaviour is the most difficult task an organisation will face and that doesn't occur overnight. To change behavior there must first be reasons and drivers to do so. It is important to first outline the benefits that changes will bring to an organisation so managers can make business justifications for implementing any change.

Reasons for Implementing DevOps

When implementing DevOps some myths are often perpetuated, such as DevOps only works for start-ups, it won't bring any value to a particular team, or that it is simply a buzz word and a fad.

The quantifiable benefits of DevOps initiatives are undeniable when done correctly. Some of these benefits include improvements to the following:

- Velocity of change
- Mean time to resolve
- Improved uptime
- Increased number of deployments
- Cross skilling between teams

- Removal of the bus factor of one

Any team in the IT industry would benefit from these improvements, so really teams can't afford to not adopt DevOps, as it will undoubtedly improve their business functions.

By implementing a DevOps initiative it promotes repeatability, measurement and automation. Implementing automation naturally improves the *velocity of change* and *increased number of deployments* a team can do in any given day. Automation of the deployment process allows teams to push fixes through to production quickly as well as allowing an organisation to push new products and features to market.

A bi-product of automation is that the *mean time to resolve* will also become quicker, if changes are automated they can be applied much more efficiently than if they were carried out manually. Manual changes depend on the velocity of the engineer implementing the change rather than an automated script that can be measured more accurately.

Implementing DevOps also means measuring and monitoring efficiently too, so having effective monitoring is crucial, as it means the pace in which root-cause analysis can carried out improves. Having effective monitoring helps to facilitate the process of mean time to resolve, so when a production issue occurs, the source of the issue can be found quicker than numerous engineers logging onto consoles and servers trying to debug issues.

Instead a well implemented monitoring system can provide a quick notification to localise the source of the issue, silencing any resultant alarms that result from the initial root-cause, allowing the issue to be highlighted and fixed efficiently.

The monitoring then hands over to the repeatable automation which can then push out the localised fix to production. This process provides a highly accurate feedback loop, where processes will improve daily. If alerts are missed they will ideally be built into the monitoring system over time as part of the incident post mortem.

Effective monitoring and automation results in quicker *mean time to resolve* which makes for happier customers and results in *improved uptime* of products. Utilising automation and effective monitoring also means that all members of a team have access to see how processes work and how fixes and new features are pushed out.

This will mean less of a reliance on key individuals removing the *bus factor of one* where a key engineer needs to do the majority of tasks in the team as he is the most highly skilled individual and has all of the system knowledge stored in his head.

Using a DevOps model means that the very highly skilled engineer can instead use their talents to help *cross skill* other team members and create effective monitoring that can help any team member carry out the root cause analysis they normally do manually. This builds the talented engineers deep knowledge into the monitoring system, so the monitoring

system as opposed to the talented engineer becomes the go to point of reference when an issue first occurs, or ideally the monitoring system becomes the source of truth that alerts on events to prevent customer facing issues. To improve *cross skilling* the talented engineer should ideally help write automation too, so they are not the only member of the team that can carry out specific tasks.

Reasons for Implementing DevOps for Networking

So how do some of those DevOps benefits apply to traditional networking teams? Some of the common complaints with silo'd networking teams today are the following:

- Reactive
- Slow often using ticketing systems to collaborate
- Manual processes carried out using admin terminals
- Lack of pre-production testing
- Manual mistakes leading to network outages
- Constantly in firefighting mode
- Lack of automation in daily processes

Network teams like infrastructure teams before them are essentially used to working in silo'd teams, interacting with other teams in large organisations via ticketing systems or using sub-optimal processes. This is not a streamlined or optimised way of working, which led to the DevOps initiative that sought to break down barriers between *Development* and *Operations* staff, but its remit has since widened.

Networking does not seem to have been initially included in this DevOps movement yet, but software delivery can only operate as fast as the slowest component. The slowest component will eventually become the bottleneck or blocker of the entire delivery process. That slowest component often becomes the star engineer in a silo'd team that can't process enough tickets in a day manually to keep up with demand, thus becoming the *bus factor of one*. If that engineer goes off sick then work is blocked, the company becomes too reliant and cannot function efficiently without them.

If a team is not operating in the same way as the rest of the business then all other departments will be slowed down as the silo'd department is not agile enough. Put simply, the reason networking teams exist in most companies is to provide a service to development teams. Development teams require networking to be deployed so they deliver applications to production so the business can make money from those products.

So networking changes to ACL policies, load balancing rules and provisioning of new subnets for new applications can no longer be deemed acceptable if they take days, months or even weeks. Networking has a direct impact on the *velocity of change*, *mean time to resolve*, *uptime* as well as the *number of deployments* which are four of the key performance indicators of a successful DevOps initiative. So networking needs to be included in a DevOps model by companies otherwise all of these quantifiable benefits will become constrained.

Given the rapid way AWS, OpenStack and Software Defined Networking (SDN) can be used to provision network functions in the private and public cloud, it is no longer acceptable for network teams to not adapt their operational processes and learn new skills. But the caveat is that the evolution of networking has been quick and they need the support and time to do this.

If a cloud solution is implemented and the operational model does not change then no real quantifiable benefits will be felt by the organisation. Cloud projects traditionally do not fail because of technology, cloud projects fail because of the incumbent operational models that hinder them from being a success. There is zero value to be had from building a brand new OpenStack private cloud, with its open set of extensible APIs to manage compute, networking and storage if a company doesn't change its operational model and allow end users to use those APIs to self-service their requests.

If network engineers are still using the GUI to point and click and cut and paste then this doesn't bring any real business value as the network engineer that cuts and pastes the slowest is the bottleneck. The company may as well stick with their current processes as implementing a private cloud solution with manual processes will not result in a speeding up time to market or mean time to recover from failure.

However, cloud should not be used as an excuse to deride your internal network staff with, as incumbent operational models in companies are typically not designed or set-up by current staff, they are normally inherited. Moving to public cloud doesn't solve the problem of the operational agility of a company's network team, it is a quick fix and bandage that disguises the deeper rooted cultural challenges that exist.

However, smarter ways of working allied with use of automation, measurement and monitoring can help network teams refine their internal processes and facilitate the developers and operations staff that they work with daily. Cultural change can be initiated in two different ways, grass roots bottom up initiatives coming from engineers or top down management initiatives.

Top Down DevOps Initiatives for Networking Teams

Top down DevOps initiatives are when a CTO, Directors or Senior Manager have buy in from the company to make changes to the operational model. These changes are required as the incumbent operational model is deemed sub-optimal and not setup to deliver software at the speed of competitors, which inherently delays new products or crucial fixes from being delivered to market.

When doing DevOps transformations from a top down management level, it is imperative that some ground work is done with the teams involved, if large changes are going to be made to the operational model it can often cause unrest or stress to staff on the ground.

When implementing operational changes upper management need have to have the buy in of the people on the ground as they will operate within that model daily. Having teams buy in is a very important aspect otherwise the company will end up with an unhappy workforce, which will mean the best staff will ultimately leave.

It is very important that upper management engage staff when implementing new operational processes and deal with any concerns transparently from the outset, as opposed to going for an offsite management meeting and coming back with an enforced plan which is all too common a theme.

Management should survey the teams to understand how they operate on a daily basis, what they like about the current processes and where their frustrations lie. The biggest impediment to changing an operational model is misunderstanding the current operational model. All initiatives should ideally be led and not enforced. So let's focus on some specific top down initiatives that could be used to help.

Analyse Successful Teams

One approach would be for the management is to look at other teams within the organisation that's processes are working well and are delivering in an incremental agile fashion, if no other team in the organisation is working in this fashion then reach out to other companies.

Ask if it would be possible to go and look at the way another company operate for a day. Most companies will be happy enough to agree, as they enjoy showing their achievements, so this shouldn't be difficult to set-up as long as they aren't a direct competitor. It is good to attend some DevOps conferences and look at who is speaking, so approach the speakers and they will undoubtedly be happy to help.

Management teams should initially book a meeting with the high performing team and do a question and answer session focusing on the following points, if it is an external vendor then an introduction phone call can suffice.

Some important questions to ask in the initial meeting are the following:

- Which processes normally work well?
- What tools they actually use on a daily basis?
- How is work assigned?
- How do they track work?
- What is the team structure?
- How do other teams make requests to the team?
- How is work prioritised?
- How do they deal with interruptions?
- How are meetings structured?

It is important not to re-invent the wheel, if a team in the organisation already has a proven template that works well, then that team could also be invaluable in helping facilitate cultural change within the networks team. It will be slightly more challenging if focus is put on an external team as the evangelist as it opens up excuses such as it being easier for them because of x, y and z in their company.

A good strategy, when utilising a local team in the organisation as the evangelist, is to embed a network engineer in that team for a few weeks and task them with watching how the other teams operate and reporting back their findings. This is imperative so the network engineers on the ground understand the processes.

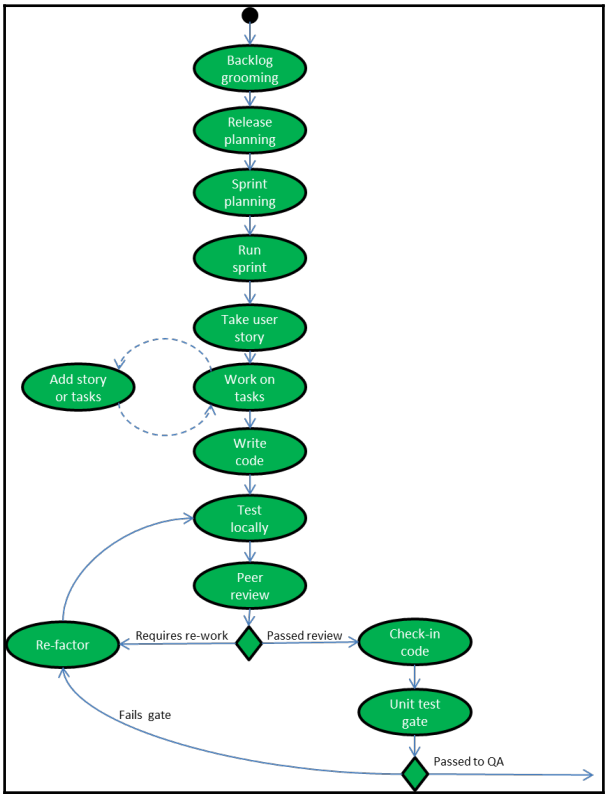
Flexibility is also important, as only some of the successful team's processes may be applicable to a network team so don't expect two teams to work identically. The sum of parts and personal individuals in the team really do mean that every team is different, so focus on goals rather than the implementation of strict process. If teams achieve the same outcomes in slightly different ways, then as long as work can be tracked and is visible to management it shouldn't be an issue as long as it can be easily reported on.

Make sure pace is prioritised, but make sure teams are comfortable with new processes, so empower the network team to choose how they want to work by engaging with the team. However, before selecting any tooling it is important to start with process and agree on the new operational model to prevent tooling driving processes, this is a common mistake in IT.

Map Out Activity Diagrams

A good piece of advice is to use an activity diagram as a visual aid to understand how a team's interactions work and where they can be improved.

A typical development activity diagram, with manual hand-off to a quality assurance team is shown below:



Utilizing activity diagrams as a visual aid is important as it highlights sub-optimal business process flows, in the example we see a development teams activity diagram. This process is sub-optimal as it doesn't include the quality assurance team in the **Test Locally** and **Peer Review** phases. Instead it has a formalised QA hand off phase, which is very late in the development cycle, and a sub-optimal way of working as it promotes a development and QA silo which is a DevOps anti-pattern.

A better approach would be to have QA engineers work on creating test tasks and creating

automated tests while the development team works on coding tasks. This would allow the development **Peer Review** process to have QA engineers review and test developer code earlier in the development lifecycle and make sure every piece of code written has appropriate test coverage before the code is checked-in.

Another short-coming in the process is that it does not cater for software bugs found by the quality assurance team or in production by customers, so mapping these streams of work into the activity diagram would also be useful to show all potential feedback loops.

If a feedback loop is missed in the overall activity diagram then it can cause a breakdown in the process flow, so it is important to capture all permutations in the overarching flow that could occur before mapping tooling to facilitate the process.

Each team should look at ways of shortening interactions to aid *mean time to resolve* and improve the *velocity of change* at which work can flow through the overall process.

Management should book aside an afternoon with the development, infrastructure, networking and test teams and map out what they believe the team processes to be in their individual teams. Keep it high level, this should represent a simple activity swim-lane utilising the start point where they accept work and the process the team goes through to deliver that work.

Once each team has mapped out the initial approach, they should focus on optimising it and removing the parts of the process they dislike and discuss ways the process could be improved as a team. It may take many iterations before this is mapped out effectively, so don't rush this process, it should be used as a learning experience for each team. The finalised activity diagram will normally include management and technical functions combined in an optimised way to show the overall process flow. Try not to bother using Business Process Management (BPM) software at this stage a simple white board will suffice to keep it simple and informal.

It is good practice to utilise two layers of an activity diagram, so the first layer can be a box that simply says **Peer Review** which then references a nested activity diagrams outlining what the teams peer review process is. Both need refined but the nested tier of business processes should be dictated by the individual teams as these are specific to their needs so it's important to leave teams the flexibility they need at this level.

It is important to split the two out tiers otherwise the overall top layer of activity diagram will be too complex to extract any real value from, so try and minimise the complexity at the top layer, as this will need to be integrated with other teams processes. The activity doesn't need to contain team specific details such as how an internal team's peer review process operates as this will always be subjective to that team; this should be included but will be a nested layer activity that won't be shared.

Another team should be able to look at a team's top layer activity diagram and understand the process without explanation. It can sometimes be useful to first map out a high performing teams top layer activity diagram to show how an integrated joined up business process should look.

This will help teams that struggle a bit more with these concepts and allow them to use that team's activity diagram as a guide. This can be used as a point of reference and show how these teams have solved their cross team interaction issues and facilitated one or more teams interacting without friction. The main aim of this exercise is to join up business processes so they are not silo'd between teams so the planning and execution of work is as integrated as possible for joined up initiatives.

Once each team has completed their individual activity diagram and optimised it to the way the team wants, the second phase of the process can begin. This involves layering each team's top layer of their activity diagrams together to create a joined up process.

Teams should use this layering exercise as an excuse to talk about sub-optimal processes and how the overall business process should look end to end. Utilise this session to remove perceived bottlenecks between teams, completely ignoring existing tools and the constraints of current tools, this whole exercise should be focussing on process not tooling.

A good example of a sub-optimum process flow that is constrained by tooling would be a stage on a top layer activity diagram that says *raise ticket with ticketing system*. This should be broken down so work is people focused, what does the person requesting the change actually require?

Developers are building features for products, so if that new feature needs a network change, then networking should be treated as part of that feature change. So the time taken for the network changes needs to be catered for as part of the planning and estimation for that feature rather than a ticketed request that will hinder the *velocity of change* when it is done reactively as an afterthought.

This is normally a very successful exercise when engagement is good, it is good to utilise a senior engineer and manager from each team in the combined activity diagram layering exercise with more junior engineers involved in each team included in the team specific activity diagram exercise.

Change the Network Teams Operational Model

The network team's operational model at the end of the activity diagram exercise should ideally be fully integrated with the rest of the business. Once the new operational model has been agreed with all teams, it is time to implement it.

It is important to note that because the teams on the ground created the operational model and joined up activity diagram it should be signed off by all parties as the new business process. So this removes the issue of an enforced model from management as those using it have been involved in creating it. The operational model can be iterated and improved over time but interactions shouldn't change greatly although new interaction points may be added that have been initially missed. A master copy of the business process can then be stored and updated so anyone new joining the company knows exactly how to interact with other teams.

Short term it may seem the new approach is slowing down development estimates as automation is not in place for network functions, so estimation for developer features become higher when they require network changes.

This is often just a truer reflection of reality, as estimations didn't take into account network changes and then they became blockers as they were tickets, but once reported it can be optimised and improved over time.

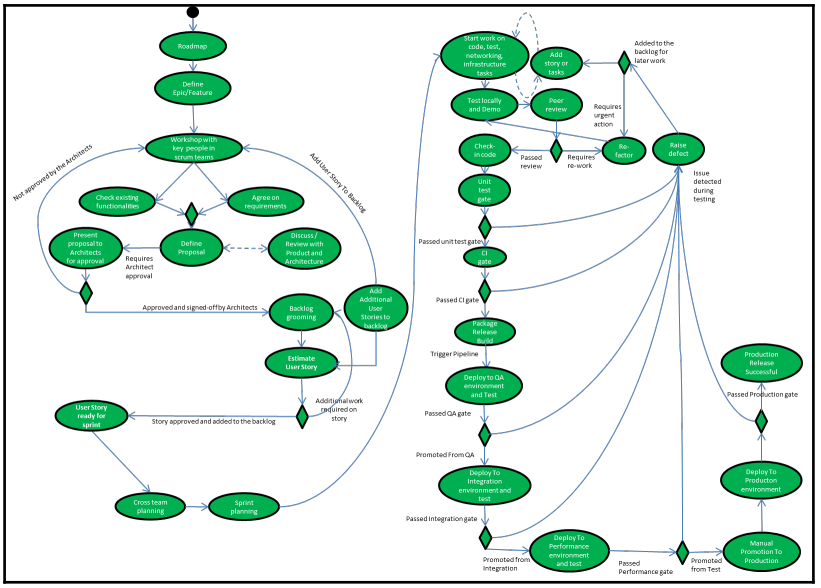
Once the overall activity diagram has been merged together and agreed with all the teams, it is important to remember if the processes are properly optimised, there should not be pages and pages of high level operations on the diagram. If the interactions are too verbose it will take any change hours and hours to traverse each of the steps on the activity diagram.

The activity diagram below shows a joined up business process, where work is either defined from a single roadmap producing user stories for all teams. New user stories, which are units of work, are then estimated out by cross functional teams including developers, infrastructure, quality assurance and network engineers. Each team will review the user story and working out which cross functional tasks are involved to deliver the feature.

The user story then becomes part of the sprint with the cross functional teams working on the user story together making sure it has everything it needs to work prior to check-in. After peer review the feature or change is then handed off to the automated processes to deliver the code, infrastructure, network changes to production.

The checked-in feature then flows through unit testing, quality assurance, integration, performance testing quality gate**s** which will include any new tests that were written by the quality assurance team prior to check-in. Once every stage is passed the automation is

invoked by a button press to push the changes to production. Each environment has the same network changes applied so network changes are made first on test environments prior to production. This relies on treating networking as code, meaning automated network processes need to be created so the network team can be as agile as the developers.



Once the agreed operational model is mapped out only then should the DevOps transformation begin. This will involve selecting the best of breed tools at every stage to deliver the desired outcome with the focus on the following benefits:

- Velocity of change
- Mean time to resolve
- Improved uptime
- Increased number of deployments
- Cross skilling between teams
- Removal of the bus factor of one

All business processes will be different for each company so it is important to engage each department and have the buy in from all managers to make this activity a success.

Changing the Network Teams Behaviour

Once a new operational model has been established in the business, it is important to help prevent the network team from becoming the bottleneck in a DevOps focused continuous delivery model.

Traditionally network engineers will be used to carrying out command line operations and logging into admin consoles on network devices to make changes. Infrastructure engineers adjusted to automation as they already had scripting experience in bash and PowerShell coupled with a firm grounding in Linux or Windows operating systems, so transitioning to configuration management tooling was not a huge step.

However, it may be more difficult to persuade network engineers from making that same transition initially. Moving network engineers towards coding against API's and adopting configuration management tools may initially appear daunting, as it is a higher barrier to entry, but having an experienced automation engineer on hand, can help network engineers make this transition.

It is important to be patient, so try to change this behaviour gradually by setting some automation initiatives for the network team in their objectives. This will encourage the correct behaviour and try and incentivise it too. It may be useful to start off automation initiatives by offering training or purchasing particular coding books for teams.

It may also be useful to hold an initial automation hack day, this will give network engineers a day away from their day jobs and time to attempt to automate a small process, that is repeated everyday by network engineers. If possible make this a mandatory exercise, so that it is adopted and make other teams available to cover for the network team so they aren't distracted. This is a good way of seeing which members of the network team may be open to evangelising DevOps and automation. If any particular individual stands out, then work with them to help push automation initiatives forward to the rest of the team by making them the champion for automation.

Establishing an internal DevOps meet-up where teams present back their automation achievements is also a good way of promoting automation in network teams and this keeping the momentum going. Encourage each team across the business to present back interesting things they have achieved each quarter and incentivise this too by allowing each team time off from their day job to attend if they participate. This leads to a sense of community and illustrates to teams they are part of bigger movement that is bringing real cost benefits to the business. This also helps to focus teams on the common goal of making the company better and breaks down barriers between teams in the process.

One approach that should be avoided at all costs is having other teams write all the network automation for networking teams, ideally it should be the networking team that evolve and

adopt automation, so giving the network team a sense of ownership over the network automation is very important. This though requires full buy in from networking teams and discipline not to revert back to manual tasks at any point even if issues occur.

To ease the transition offer to put an automation engineer into the network team from infrastructure or development but this should only be a temporary measure. It is important to select an automation engineer that is respected by the network team and knowledgeable in networking, as no one should ever attempt to automate something that they cannot operate by hand, so having someone well versed in networking to help with network automation is crucial, as they will be training the network team so have to be respected. If an automation engineer is assigned to the network team and isn't knowledgeable or respected, then the initiative will likely fail, so choose wisely.

It is important to accept at an early stage, that this transition towards DevOps and automation may not be for everyone, so every network engineer will be able to make the journey. It is all about the network team seizing the opportunity and showing initiative and willingness to pick up and learn new skills. It is important to stamp out disruptive behaviour early on which may be a bad influence on the team. It is fine to have for people to have a cynical skepticism at first, but not attempting to change or build new skills shouldn't be tolerated, as it will disrupt the team dynamic and this should be monitored so it doesn't cause automation initiatives to fail or stall, just because individuals are proving to be blockers or being disruptive.

Bottom-Up DevOps Initiatives for Networking Teams

Bottom up DevOps initiatives are when an engineer, team leads or lower management don't necessarily have buy in from the company to make changes to the operational model. However, they realise that although changes can't be made to the overall incumbent operational model, they can try and facilitate positive changes using DevOps philosophies within their team that can help the team perform better and make their productivity more efficient.

When implementing DevOps initiatives from a bottom up initiative, it is much more difficult and frustrating at times as some individuals or teams may not be willing to change the way they work and operate as they don't have to. But it is important not to become disheartened and do the best possible job for the business.

It is still possible to eventually convince upper management to implement a DevOps initiative using grass roots initiatives to prove the process brings real business benefits.

Evangelise DevOps in the Networking Team

It is important to try and stay positive at all times, working on a bottom up initiative can be tiring but it is important to roll with the punches and not take things too personally. Always remain positive and try to focus on evangelizing the benefits associated with DevOps processes and positive behavior first within your own team. The first challenge is to convince your own team of the merits of adopting a DevOps approach before prior to even attempting to convince other teams in the business.

A good way of doing this is by showing the benefits that DevOps approach has made to other companies such as Google, Facebook and Etsy focusing on what they have done in the networking space. A pushback from individuals may be the fact that these companies are **unicorns** and DevOps has only worked for companies for this reason, so be prepared to be challenged. Seek out initiatives that have been implemented by these companies that the networking team could adopt and are actually applicable to your company.

In order to facilitate an environment of change, work out your colleagues drivers are, what motivates them? Try tailor the sell to individuals motivations, the sell to an engineer or manager may be completely different, an engineer on the ground may be motivated by the following:

- Doing more interesting work
- Help automate menial daily tasks
- Build new skills and a better CV
- Learn sought after configuration management skills
- Learn how to code

A manager on the other hand will probably be more motivated by offering to measure KPI's that make his team look better such as:

- Time taken to implement changes
- Mean time to resolve failures
- Improved uptime of the network

Another way to promote engagement is to invite your networking team to DevOps meet-ups arranged by forward thinking networking vendors. They may be amazed that most networking and load balancing vendors are now actively promoting automation and DevOps and not yet be aware of this. Some of the new innovations in this space may be enough to change their opinions and make them interested in picking up some of the new approaches so they can keep pace with the industry.

Seek Sponsorship from a respected Manager or Engineer

After making the network team aware of the DevOps initiatives it is important to take this to the next stage. Seek out a respected manager or senior engineer in the networking team that may be open to trying out DevOps and automation. It is important to sell this person the dream, state how you are passionate about implementing some changes to help the team, and that you are keen to utilize some proven best practices that have worked well for other successful companies.

It is important to be humble, try not preach from the alter of DevOps which can be very off putting, make reasonable arguments and justify them avoiding sweeping statements. Try not to appear to be trying to undermine the manager or senior engineer, instead ask for their help to achieve the goal by seeking their approval to back the initiative or idea. A charm offensive may be necessary at this stage to convince the manager or engineer that it's a good idea but gradually building up to the request can help otherwise it may appear insincere if it comes out the blue. Potentially test the water over lunch or drinks and gauge if it is something they would be interested in, there is little point trying to convince people that are stubborn as they probably will not budge unless the initiative comes from above.

Once you have found the courage to broach the subject it is now time to put forward numerous suggestions on how the team could work differently. Ask for the opportunity to try this out on a small scale and offer to lead the initiative and ask for their support and backing. It is likely that the manager or senior engineer will be impressed at your initiative and allow you to run with the idea but they may choose the initiative you implement so never suggest anything you can't achieve you may only get one opportunity at this so it is important to make a good impression.

Try and focus on a small task to start with that's typically a pain point and attempt to automate it. Anyone can write an automation script but try and make the automation process easy to use, find what the team likes in the current process and try and incorporate aspects of it. For example if they often see the output from a command line displayed in a particular way, write the automation script so that it still displays the same output, so the process is not completely alien to them.

Try not to hardcode values into scripts and extract them into a configuration files to make the automation more flexible so it could potentially be used again in different ways. By showing engineers the flexibility of automation it will encourage them to use it more, show other in the teams how you wrote the automation and ways they could adapt it to apply it to other activities. If this is done wisely then automation will be adopted by enthusiastic members of the team and you will gain enough momentum to impress the sponsor enough to take it forward onto more complex tasks.

Automate a Complex Problem with the Networking team

The next stage of the process after building confidence by automating small repeatable tasks, is to take on a more complex problem, this can be used to cement the use of automation within the networking team going forward.

This part of the process is about empowering others to take charge, and lead automation initiatives themselves in the future, so will be more time consuming. It is imperative that the more difficult to work with engineers that may have been deliberately avoided while building out the initial automation is involved this time.

These engineers more than likely have not been involved in automation at all at this stage. This probably means the most certified person in the team and alpha of the team, nobody said it was going to be easy, but it will be worth it in the long run convincing the biggest skeptics of the merits of DevOps and automation. At this stage automation within the network team should have enough credibility and momentum to broach the subject citing successful use cases.

It's easier to involve all difficult individuals in the process rather than presenting ideas back to them at the end of the process. Difficult senior engineers or managers are less likely to shoot down your ideas in front of your peers if they are involved in the creation of the process and have contributed in some way.

Try and be respectful, even if you do not agree with their viewpoints, but don't back down if you believe you are correct or give up. Make arguments fact based and non-emotive, write down pros and cons and document any concerns without ignoring them, you have to be willing to compromise but not to the point of devaluing the solution.

There may actually be genuine risks involved that need addressed so valid points should not be glossed over or ignored. Where possible seek backup from your sponsor if you are not sure on some of the points or feel individuals are being unreasonable.

When implementing the complex automation task work as a team, not as an individual, this is a learning experience for others as well as yourself. Try and teach the network team a configuration management tool, they may just be scared try out new things, so go with a gentle approach. Potentially stopping at times to try out some online tutorials to familiarize everyone with the tool and try out various approaches to solve problems in the easiest way possible.

Try and show the network engineers how easy it is to use configuration management tools and the benefits. Don't use complicated configuration management tools as it may put them

off. The majority of network engineers can't currently code, something that will potentially change in the coming years, as stated before infrastructure engineers at least had a grounding in bash or PowerShell to help get started, so pick tooling that they like and give them options, try not to enforce tools they are not comfortable with on them. When utilizing automation one of the key concerns for network engineers, is peer review as they have a natural distrust that the automation has worked. Try and build in gated processes to address these concerns, automation doesn't mean any peer review so create a light-weight process to assist. Make the automation easy to review by utilizing source control to show diffs and educate the network engineers on how to do this.

Coding can be a scary prospect initially, so propose to do some team exercises each week on a coding or configuration management task. Work on it as a team, this makes it less threatening and it is important to listen to feedback. If the consensus is that something isn't working well or isn't of benefit then look at alternate ways to achieve the same goal that works for the whole team. Before releasing any new automated process test it in pre-production environment, alongside an experienced engineer and have them peer review it and try to make it fail against numerous test cases. There is only one opportunity to make a first impression, with a new process, so make sure it is a successful one.

Try and setup knowledge sharing session between the team to discuss the automation and make sure everyone knows how to do operations manually too, so they can easily debug any future issues or extend or amend the automation. Make sure output and logging is clear to all users as they will all need to support the automation when it is used in production.

Summary

In this chapter we have covered practical initiatives, that when combined, will allow IT staff to implement successful DevOps models in their organisation. Rather than just focussing on departmental issues, it has promoted the using a set of practical strategies to change the day to day operational models that constrain teams. It also focuses on the need for network engineers to learn new skills and techniques in order to make the most of a new operational model and not become the bottleneck for delivery.

This chapter has provided practical real world examples that could help senior managers and engineers to improve their own companies, emphasising collaboration between teams and showing that networking departments now required to automate all network operations to deliver at the pace expected by businesses.

In the coming chapters we will look at ways of applying automation to networking and concentrate on configuration management tools such as Ansible. These configuration management tools can be used to increase the pace that network engineers can implement

changes as well as making sure all network changes that are made are done in the same way and are less error prone.

4

Configuring Network Devices Using Ansible

This chapter will focus on some of the most popular networking vendors in the market today namely Cisco, Juniper and Arista and look at how each of these market leading vendors have developed their own proprietary operating system to control network operations. The aim of this book is not to discuss which network vendors solution is better, but instead look at ways network operators can utilise configuration management tooling today to manage network devices, now that network vendors have created API's and SDKs to programmatically control the network.

Once the basics of each operating system have been established we will then shift focus to the hugely popular open source configuration management tool from RedHat called Ansible. It will look at ways it can be used to configure network devices programmatically and assist with network operations. This chapter will show practical configuration management processes that can be used to manage network devices.

In this chapter the following topics will be covered:

- Network Vendors Operating Systems
- Introduction to Ansible
- Ansible Modules Currently Available For Network Automation
- Configuration Management Processes To Manage Network Devices

Network Vendors Operating Systems

Market leading networking vendors such as Cisco, Juniper and Arista have all developed their own operating systems that allow network operators to issue a series of commands to network devices via a **command line interface CLI**.

Each vendor's CLI is run from their bespoke operating systems:

- Cisco IOS and NXOS
- Juniper Junos
- Arista EOS

All of these operating systems have meant that it has become easier to programmatically control switches, routers and security devices provided by these vendors, as they seek to simplify operating network devices.

The rise of DevOps in industry has also meant that it is no longer acceptable to not provide programmatic APIs or an SDK to aid automation, with networking vendors now integrating with configuration management tooling such as Puppet, Chef and Ansible to plug into DevOps tool chains.

Cisco IOS and NXOS operating system

The Cisco IOS operating system when released, was the first of its kind, providing a set of command lines that network operators could use to mutate the state of the network. However it still had its challenges, it had a monotholic architecture which meant that all processes shared the same memory space, with no protection between parallel processes, so it didn't align itself well to parallel updates but at the time it was the clear market leader. This changed network operations and meant that network engineers would each individually log onto network switches and routers to make updates using its fully featured **command line interface CLI**.

At the time this greatly reduced the complexity of network operations and Cisco standardised the way the networking industry carried out network operations in a data center. Network operators would log onto appliances and run an industry standard series of command lines to make changes to routers or switches and Cisco ran certification programmes to teach administrators how to operate the equipment and learn all the commands.

Today with efficiency and cost reductions key to company's businesses surviving, and a shift towards more agile processes, this model in the modern data centres has an obvious

scaling issue with *x* amount of network engineers required per network device. The emergence of private clouds has meant that the number of network devices each network engineer needs to manage has grown dramatically so automation has become a key to managing the growing amount of devices in a consistent way.

Cisco as the networking market has evolved in recent years have since developed a new operating system called **NXOS** which has allowed itself to integrate with open source technologies lend itself to automation. The NXOS operating system is deployed with all new Nexus switches and routers and this operating system has shifted Cisco towards open and modular standards by integrating with open protocols such as BGP, EVNP and VXLAN and the appliances can even run LXC containers.

Cisco have also provided a set of REST API's that allows network operators to run native Linux and bash shells to carry out regular administration commands server side. In a world where AWS and OpenStack programmatic APIs are available to mutate network infrastructure, networking vendors needed to adapt to survive or they risked being left behind so Cisco have made their own switches and routers as easy to configure and operate as the virtual appliances.

The **NXOS** operating system allows the use the RedHat enterprise Linux rpm package manager to control software updates. This means that software updates can be done on the NXOS in an industry standard way, the same as patching a Linux guest operating system would be carried out by an infrastructure systems administrator. Consequently, Cisco network devices no longer feel like alien appliances to Linux systems administrator and more like native Linux to end users which has undoubtedly made them simpler to administrate.

The Cisco NXOS operating system means that the speed that network changes can be pushed increases, as operations staff can use their own tool chains and configuration management tools to automate updates. The NXOS operating system has become less vendor specific therefore lowering the barrier to entry to use networking products and automation of its product suits have become easier.

Juniper Junos operating system

The Juniper Junos operating systems driver is programmatically controlled network operations, Junipers Junos operating system was created to provide a **command line interface CLI** that users can execute to retrieve facts about the running system. The Junos operating system is based on a clearly defined hierarchical model as opposed to using a series of unrelated configuration files. The hierarchical model also comes complete with operational and configuration modes of operation.

Intuitively operational mode is used to upgrade the operating system, monitor the system and also check the status of juniper devices. Configuration mode on the other hands allows network operators to configure user access and security, interfaces, hardware and set the protocols should be used on the device which gives a clear separation of roles between those installing the system and those operating it. The Junos operating system supports all open protocols such as BGP, VXLAN and EVPN as well as in built roll forward and roll back capability.

Juniper provide a Python library called PyEZ for the Junos operating system as well as a PowerShell option for Windows administrators that utilises wrapped PowerShell wrapped in Python. The Python library PyEZ can retrieve any configuration information using tables and views that allow network operators to script against run time information provided by the Junos operating system. Once a table items have been extracted by utilising a python script using a get() method, tables can subsequently be treated as a Python dictionary and iterated, which allows users to carry out complex scripting if required to automate all network operations. The Junos PYEZ library is also fully extensible and network operators can add functionality they deem appropriate using its widget system.

Arista EOS operating system

The Arista EOS operating system is based on open standards to promote automation of network functions. It relies upon a centralised **CloudVision eXchange (CVX)** and the CVX servers hold the centralised state of the network. The EOS operating system separates the functional control on every switch using Sysdb, which is the Arista EOS operating systems database. The Arista Sysdb is an in-memory database running in user space and contains the complete state of the Arista switch. Sysdb is maintained in memory on the device so if an Arista switch is either restarted or powered down all information for that switch is lost.

The CVX server acts as an aggregator all states from every switch's Sysdb into a network-wide database depending on what services are enabled on the cluster of CVX servers. When state changes in Sysdb on a switch then the change is pushed to the CVX centralised database, which then updates its configuration and notifies agents running on CVX of the change.

The Arista EOS operating system supports modern open protocols such as MLAG, ECMP, BGP and VXLAN. It utilises overlay technologies such as VXLAN allowing applications to be deployed and remain portable in the modern data center. Arista heavily promotes the use of the leaf spine architecture with ECMP, which allows a scale out model to be implemented; this aligns itself to modern cloud solutions such as OpenStack and makes it agnostic to SDN controller solutions.

The Arista EOS operating system is a Linux based operating system designed to be programmatically controlled. The main driver for the EOS operating system is to allow network operators to carry out network operations well-structured set of APIs including the eAPI, CLI command as well as Python, Ruby and GO libraries available as part of its SDK portfolio.

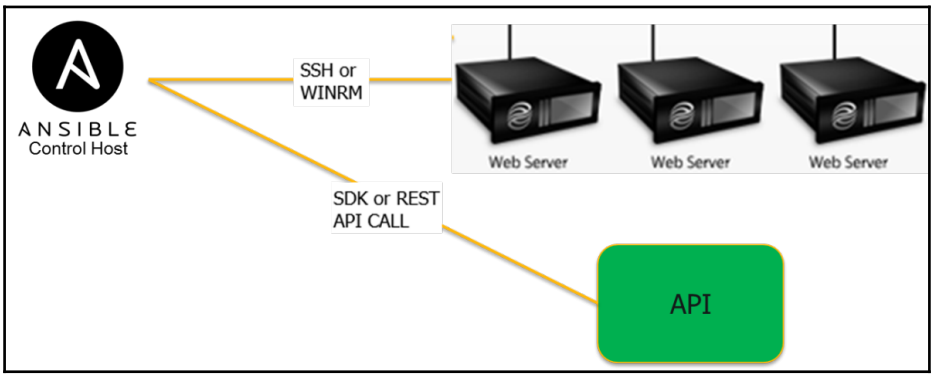
The EOS operating system also allows **smart system upgrade (SSU)** to allow scale out of Arista appliances with live patching and upgrades also made less difficult, this helps to support businesses 99.99% uptime targets. Switches can now be racked and cabled in the data center by data center operations teams then handed over to Arista's **zero touch provisioning (ZTP)** process that automates the initialisation of switches and **zero touch replacement (ZTR)** allows switches to be replaced in the data center.

The Arista EOS solution is also packaged with its CloudVision product which can be used to automate networking workflow tasks through the portal if users require a visual view of switches and routers and CloudVision allows integration with SDN controllers using OVSDB, eAPI or OpenFlow. Like Cisco and Juniper the EOS API lends due to it having multiple SDK options this Arista products can be easily managed by configuration management tools such as Puppet, Chef and Ansible so that no network operation is done manually.

Introduction to Ansible

Ansible is primarily a push based configuration management tool that uses a single **Ansible Control Host** and can connect to multiple Linux guest operating systems via SSH to configure them and recently added WINRM support so it can now also configure Windows guests in the same way as Linux based operating systems. As Ansible can connect to multiple servers simultaneously it aids operators by allowing them to carry out uniform operations across multiple Linux or Windows servers at the same time. This allows Ansible to help simplify the automation of repeatable tasks by defining them in Ansible so they can be consistently executed against target servers. Ansible can also be used as a centralised orchestration tool that can connect to API endpoints and sequence API operations.

Below we can see an example of the way an Ansible Control Host connects to servers or acts as a centralised orchestration tool:



Every operation Ansible carries out should be idempotent as a standard, meaning that if the desired state is already configured on a server, then Ansible will check the intended state from a playbook or role and not take any action if a server is already in the correct state, only if the state is different from what is specified in a playbook or role will the operation be executed to mutate the state of the server.

Ansible is a Python based configuration management tool that controls servers from a Linux based Control Host, using YAML files to define and describe desired state. Ansible is packaged with a rich set of extensible modules which are primarily written in Python, but can also be written in any language that a user wishes. Ansible modules allow Python SDK's or REST API's to be wrapped in Ansible's plug-in boilerplate and then utilised from Ansible roles or playbooks in an easy to use architecture multiple times. Before going into more detailed examples it is important to understand some of the Ansible terminology.

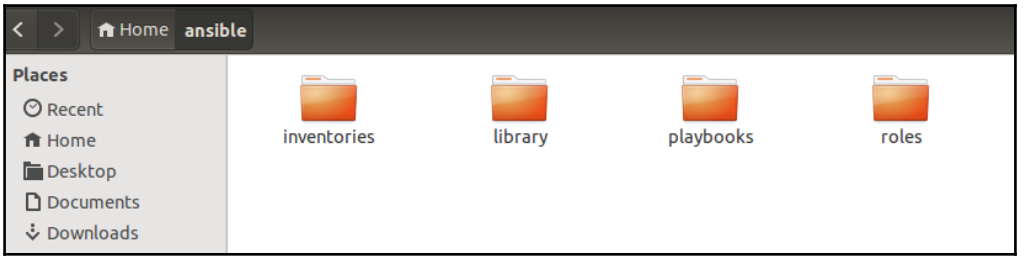
Ansible Directory Structure

Ansible is made up of a series of YAML files that are laid out in a customisable directory structure.

In this customised structure the **Ansible Controller Node** has the following directory structure:

- Inventories folder to hold the Ansible inventory
- Library folder to hold any custom python plug-ins
- Playbooks folder to hold all playbooks
- Roles folder to hold all the Ansible roles

This directory structure is shown below:



This provides logical groupings of all Ansible components, which will be useful as the amount of playbooks or roles grow in size. It is best practice to version the Ansible folder structure in a source control management system such as GIT.

Ansible Inventory

An Ansible inventory file is simply a set of DNS hostnames or IP addresses defined in a YAML file. This allows Ansible to connect to those target hosts and execute specific commands on servers.

Ansible allows users to use inventory files to group servers into particular types or use cases. For example in networking terms, when utilising Ansible to set-up a leaf spine architecture, a network operator could have a group for leaf switches and another for the spine switches. This is because a different set of run-book commands would be required to configure each, so limits can be applied upon execution to only execute a command against a small subset of servers limited to one particular group.

An example of an inventory file defining leaf and spine switches can be found below showing the definition of two groups in the inventory file, one for leaf switches called **leaf** and one for spine switches called **spine** containing all the DNS entries for the switches:

```
[spine]
spineswitch01
spineswitch02

[leaf]
leafswitch01
leafswitch02
leafswitch03
leafswitch04
```

The same inventory can be described in an abbreviated format:

```
[spine]
spineswitch[01-02]

[leaf]
leafswitch[01-04]
```

Ansible modules

An Ansible module is typically written in Python or can be written in any other programming language. An Ansible modules code defines a set of operations to add or remove functionality from a guest operating system or alternately execute a command against an API if it is being used for orchestration. Ansible modules can be used to wrap either a simple command line, API call or any other operation a user desires that can be coded programmatically. Modules are set-up so they can be re-used in multiple playbooks or roles so they promote code re-use and standardization of operations.

Code specified in an Ansible module is wrapped in Ansibles module boilerplate, which structures the layout of the module. The boilerplate promotes a set of standards, so each module is idempotent by design, meaning that the code will first detect the state of the system and then determine if a change in state is required or not before executing the operation.

When a state change is executed in Ansible it is donated by a yellow output on the console. If no action has been taken it will display a green colour to state that the operation ran successfully but no state change was made, while red console output indicates a failure on the module.

Ansible modules expose a set of command line arguments for the module that can either be mandatory or optional and can have default values. Modules that adhere to the Ansible standard are created with a state variable that contains *present* or *absent*, as one of the command line variables. A module when set to *present* will add the feature that has been specified by the playbook and when set to *absent* will remove the specified feature. All modules will typically have code to deal with both of these use cases.

Once an Ansible module has been written it is placed in the library folder, which means it is available as a library to the Python interpreter and the code can then be utilised by defining it in an Ansible playbook or role. Ansible comes with a set of pre-packaged core and extras modules that can all be accessed by writing some YAML to describe the operation that is

required, all modules are packaged with documentation that are part of the boilerplate and available on the Ansible website.

Core modules are maintained by the Ansible core team in joint initiatives with software vendors and are generally of the high quality. Extras modules can also be of a good quality but are not maintained by vendors and sometimes maintained by users that have committed back the modules to Ansible to help out the open source community.

A simple core yum module donated by yum: can be seen below, that takes two command line variables name of the rpm to install and the state which determines whether to install or remove it from the target server:

```
- name: install the latest version of Apache
  yum: name=httpd state=present
```

Ansible roles

Roles are a further level of abstraction in Ansible and also defined using YAML files. Roles can be called from playbooks, this aims to simplify playbooks as much as possible. As increased sets of functionality are added to playbooks, they can become cluttered and difficult to maintain from a single file. So roles allow operators to create minimal playbooks that then pull all the information from the Ansible directory structure which then determines the configuration steps that need to execute on servers or be run locally.

Ansible roles attempt to strip out repeatable parts of playbooks and group them into roles so they can be used by multiple playbooks if required. Roles are groupings to determine what the server profile should actually be, rather than just focusing on multiple ad-hoc instructions, so a playbook could be called spine.yml and the playbook could contain a set of modular roles used to define the particular spine switches run-list to build the spine switch on each target server specified in the Ansible inventory. Some of these roles should be modular enough that they can be reused when creating leaf switches.

Ansible Playbooks

An Ansible playbook is a YAML file that dictates the run-list to carry out on a particular set of hosts that are defined in an inventory file. A playbook specifies an ordered set of instructions to execute commands locally from **Ansible Controller Node** or on a target set of hosts specified in the Ansible inventory file.

An Ansible playbook can be used to create a run-list that call out to modules or specific roles which dictate the operations that should be executed against a server.

In this example we see a playbook targeting the spine hosts in the inventory file and executing multiple roles to set-up the spine servers:

```
---
- hosts: spine
  gather_facts: no
  connection: local

  roles:
    - common
    - interfaces
    - bridging
    - ipv4
    - bgp
```

An alternate playbook could not use roles at all, and call Ansible yum core module directly to install the apache **httpd-2.2.29** yum package on the inventory group called server:

```
---
- hosts: server
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd-2.2.29 state=present
```

Playbooks can also specify while conditions to dictate if an action in the playbook should be executed or not based on the output of a proceeding operation and the register command is used to store JSON output from a command that can then be utilised in playbooks or roles by subsequent tasks.

Ansible playbooks from version 2.x onwards can now utilize block recue functionality too. So if an operation is nested in a block command fails then the rescue section of the playbook is invoked. This can be useful for doing cleanup of failed actions to make playbooks more robust.

The usefulness of a block rescue operation shouldn't be underestimated, when requiring to

copy a large database dmp file to a back-up location this operation could sometimes be error prone due to the volume of data being copied. So if the disk space is too low on the target directory, then that operation could fail half way through leaving only part of the file copied and the server in an unusable state and the server could run out of disk space. So a recue command could be used to clean-up the copied file immediately so the server isn't left in a bad state if the copy operation fails. After the rescue command has completed the playbook will exit with an error but remain in its original state.

In the below example we can see a playbook using the copy: module to copy the source file /var/files/db.dmp to /backups/db.dmp and the file:module being used to delete the file if the original command fails:

```
---
- hosts: servers
  remote_user: root
  tasks:
  - block:
      - copy: src=/var/files/db.dmp dest=/backups/db.dmp owner=armstrongs group=admin mode=0644
  rescue:
      - file: path=/backups/db.dmp owner=armstrongs state=absent group=admin mode=0644
```

Executing an Ansible Playbook

After a playbook and inventory has been created utilising the specified folder structure it can now be executed by specifying the ansible-playbook command

In the below example the

- ansible-playbook tells ansible that a yaml playbook file should be specified.
- -i flag is used to specify the inventory file
- -l limits the execution only to the servers under the inventory group (servers)
- -e passes additional variables to the playbook in this example production
- -v sets the verbosity of the output

```
ansible-playbook -i inventories/inventory -l servers -e environment=production playbooks/devops-for-networking.yml -v
```

Ansible vars and jinja2 templates

Ansible var files are just another YAML file that specify a set of variables that will be substituted into a playbook at run-time using the Ansible include_vars statement.

Var files are just a way of breaking out variables that are required by playbooks or roles at runtime. This means that different var files can be passed at runtime without having to hardcode variables into playbooks or roles.

An example of a var file syntax is shown below, this shows the contents of a common.yml var file containing one defined variable called cert_name:

```
# sslcert vars
cert_name: cert1
```

The example below shows this example the common.yml above and other environment.yml variables both being loaded into the playbook. The {{ environment }} is useful as it means that different values could be passed from the ansible-playbook command line to control the variables that are imported into the playbook using the -e "environment=production" option at runtime:

```
- name: Include vars
  include_vars: "../roles/networking/vars/{{ item }}.yml"
  with_items:
   - "common"
   - "{{ environment }}"
```

The common.yml var files variables value cert1 can then be used by specifying {{ cert_name }} variable in the playbook:

```
"{{ cert_name }}"
```

Ansible also has the ability to utilise Python jinja2 templates that can be transformed at runtime, to populate the configuration files information utilising a set of var files, for example the {{ environment }} variable in the example above can be specified at run-time to

load variables that populate unique environment information. The jinja2 template then once transformed using the template module will be parameterised to use the variables specified in the environment.yml file.

In the below example we can see the Ansible template: module being executed as part of a role copying a jinja2 template network_template.j2 copied transformed to /etc/network.conf:

```
- template: src=/networking/network_template.j2 dest=/etc/network.conf owner=bin group=admin mode=0644
```

Pre-Requisites Using Ansible to Configure Network Devices

The base constructs covered in the *Introduction to Ansible* section in this chapter are all relevant to the Ansible networking modules, a networking team wishing to utilise Ansible for configuration management. Before starting it is important to check with the networking vendors that the version of the networking operating system can be used with Ansible. The next step is to configure a small provisioning server to utilise as the **Ansible Control Host,** this is typically created on the Management Network so it has access appropriate to all switches.

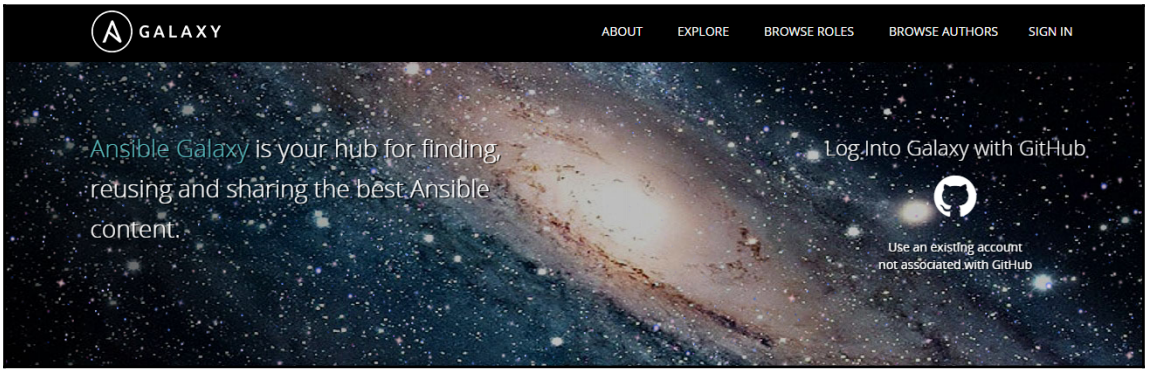
The server can be relatively small in size as it will just be required to connect over SSH to the Linux based networking operating systems. Ensure that the API command line is enabled on the network device, it is also a good idea to creating a temporary user account on each of the networking devices, which will allow you to setup a public key on the **Ansible Control Host** and scp the created id_rsa.pub to the authorized_keys folder on the network devices using the temporary account. This will allow Ansible to use that private key to connect to all of the hosts without the need for dealing with passwords. The temporary password can then be deleted from each of the network devices once this setup activity has been completed, you could even use Ansible to do this as a first activity.

All being well the next step would be to create the Ansible folder structure on the provisioning server and fill out the Ansible inventory file with all the DNS names of all the network devices and installing Ansible on a Linux provisioning server. Ansible is now packaged by RedHat in rpm format so this should just be a simple yum install as long as the **Ansible Control Host** has outbound internet access to the RedHat repositories when using a centos image or RedHat Enterprise Linux, it will of course work on any Linux based operating system as is still available as a PyPi package that can be installed on Ubuntu.

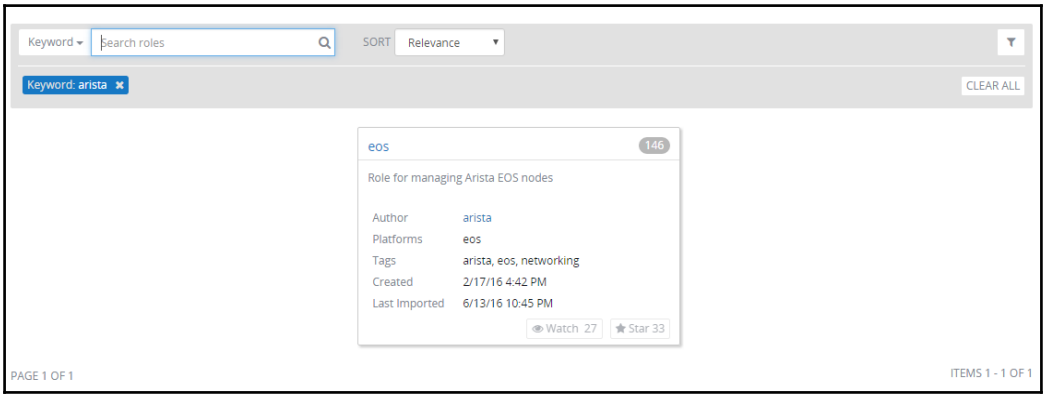
Ansible Galaxy

If the network engineer was looking for a start point and not versed in coding they could look for examples on Ansible Galaxy which hosts open source community roles that carry out many complex commands utilising pre-existing roles.

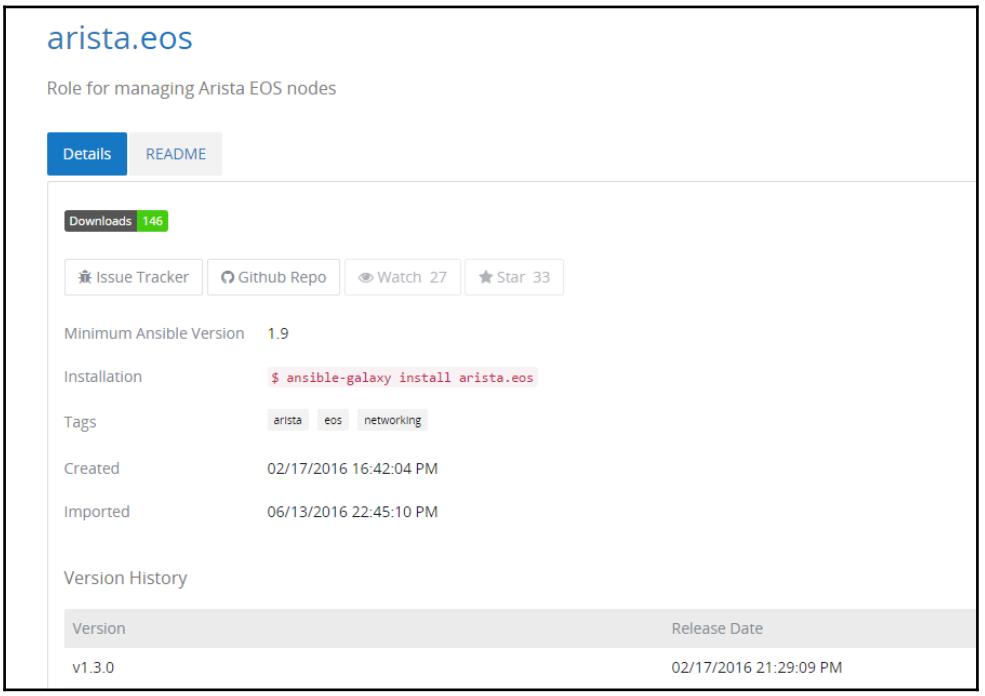
The network engineer can navigate to the Ansible Galaxy repository at https://galaxy.ansible.com/



Ansible Galaxy houses thousands of Ansible roles that have been developed by the Open Source community. Users can browse roles and search for a particular networking vendor. In this example a search for Arista has returned the **eos** role as shown below:



Each role returned has a link to their corresponding GitHub repository:

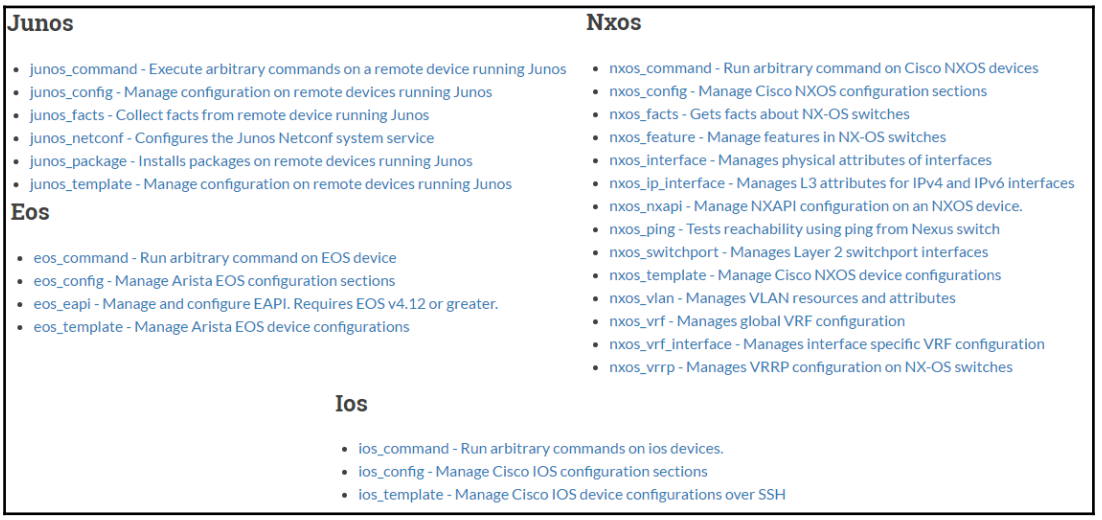


Ansible Galaxy is a very useful tool, where users can take roles as a start point and customize them to meet their needs. Rather than just taking from the community any new roles that may be of use to others should be contributed back to Ansible community.

Ansible Core Modules Available For Network Operations

Since the release of Ansible 2.0, the Ansible configuration management tool been packaged with some of the core networking modules from Arista, Citrix, Cumulus and Juniper. Ansible can be used to edit configuration for any network device, it isn't restricted to just these modules. Ansible Galaxy has a wide range of roles that have been developed by the open source community.

A subnet of the Ansible 2.x networking modules can be shown below focusing upon the Juniper Junos, Arista EOS, Cisco NXOS and IOS:



Junos

- junos_command - Execute arbitrary commands on a remote device running Junos
- junos_config - Manage configuration on remote devices running Junos
- junos_facts - Collect facts from remote device running Junos
- junos_netconf - Configures the Junos Netconf system service
- junos_package - Installs packages on remote devices running Junos
- junos_template - Manage configuration on remote devices running Junos

Eos

- eos_command - Run arbitrary command on EOS device
- eos_config - Manage Arista EOS configuration sections
- eos_eapi - Manage and configure EAPI. Requires EOS v4.12 or greater.
- eos_template - Manage Arista EOS device configurations

Nxos

- nxos_command - Run arbitrary command on Cisco NXOS devices
- nxos_config - Manage Cisco NXOS configuration sections
- nxos_facts - Gets facts about NX-OS switches
- nxos_feature - Manage features in NX-OS switches
- nxos_interface - Manages physical attributes of interfaces
- nxos_ip_interface - Manages L3 attributes for IPv4 and IPv6 interfaces
- nxos_nxapi - Manage NXAPI configuration on an NXOS device.
- nxos_ping - Tests reachability using ping from Nexus switch
- nxos_switchport - Manages Layer 2 switchport interfaces
- nxos_template - Manage Cisco NXOS device configurations
- nxos_vlan - Manages VLAN resources and attributes
- nxos_vrf - Manages global VRF configuration
- nxos_vrf_interface - Manages interface specific VRF configuration
- nxos_vrrp - Manages VRRP configuration on NX-OS switches

Ios

- ios_command - Run arbitrary commands on ios devices.
- ios_config - Manage Cisco IOS configuration sections
- ios_template - Manage Cisco IOS device configurations over SSH

Ansible 2.x has sought to simplify networking modules by giving them a standard set of operations across all modules to make it feel more intuitive to network engineers. As many network engineers are not familiar with configuration management tooling, having a set of standards across modules simplifies the initial barrier to entry, as network engineers are able to see commands that they would utilise everyday being used as part of a playbook or a role, so Ansible can initially be utilised as a scheduling tool, before network operators delve into more complex modules.

One of the main fears network engineers have when first using configuration management tooling is not trusting the system or understanding what is going on under the covers. So being able to easily read playbooks or roles and see the operations that are being executed builds confidence in the tooling and makes adoption easier.

It is fully expected more complex networking modules will be built out over time by the open source community some of which are already available with roles from Arista, Juniper and Cisco available in Ansible Galaxy. However, the following Ansible core modules have been standardised to allow configuration of Arista, Cisco and Juniper network devices in the same way. These modules can be used in any playbook or role.

_command module

The main module packaged with a vendors networking modules in Ansible 2.x is the _command module. This is a conscious choice by Ansible as it is more intuitive to network engineers initially to use native network commands initially when switching to

configuration management tooling.

This module allows Ansible to connect to hosts using SSH as network devices operating system are primarily Linux based operating systems.

The _command module allows network operators to and applies configuration changes to switches by connecting from the **Ansible Control Host**. The syntax used by Ansible on this command is identical to what network operators would execute on network devices using the CLI.

In the below example the EOS command **show ip bgp summary** command is executed by the eos_command and it connects to every specified {{ inventory_hostname }} which is a special Ansible variable that substitutes the DNS name of every node listed in the host group specified in inventory file. It then registers the output of the command in the eos_command_output variable.

```
tasks:
    - name: execute show ip bgp
      eos_command:
        commands:
          - show ip bgp summary
        host={{ inventory_hostname }}
      register:
        eos_command_output
```

Junos syntax is identical. In the below example a similar network command executed on Junos to show interfaces with the json output captures in the junos_command_output variable.

```
tasks:
    - name: show interfaces and capture in variable
      junos_command:
        commands:
          - show interfaces
      register:
        junos_command_output
```

The Cisco example shows the NXOS but the configuration is also the same in IOS, the nxos_command issues a show version command and places the result in the

nxos_command_output variable:

```
tasks:
    - name: show version and capture in variable
      nxos_command:
        commands:
         - show version
      register:
        nxos_command_output
```

_config module

The _config module is used to configure updates in a deterministic way that could be used for implementing change requests, by batching up a number of commands.

This module allows operators to updating of selected lines or blocks of running configuration programmatically on the network device. The module will connect to the device, extracting the running configuration before pushing batch updates in a completely deterministic way.

In the example below the Arista switches configuration will loaded by the module, the no spanning-tree vlan 4094 command will be executed on the EOS operating system if the running configuration doesn't match the existing state so the desired end state will be implemented on the switch.

```
tasks:
  - name: set no spanning tree on vlan
    eos_config:
     lines:
       - no spanning-tree vlan 4094
     host={{ inventory_hostname }}
    register:
      eos_command_output
```

_template module

The _template module is used to update configuration utilising a jinja2 template file. This can be extracted from the running configuration of a network device, updated and then pushed back to the device.

Another use case for the _template module would be allowing network administrators to extract the running config into a jinja2 template from one network device and apply it to others switches to propagate the same changes.

The _template module will only push incremental changes unless the force command is specified as a command line variable, which will carry out overwrite.

In the example below the eos_config jinja2 template is pushed to the Arista device and will do an incremental change to the configuration if the jinja2 template has configuration changes.

```
tasks:
    - name: push eos_config.j2 template to EOS
      eos_template:
        src: eos_config.j2
      register:
        eos_command_output
```

Configuration Management Processes To Manage Network Devices

DevOps is all about people and process not tooling, so just focussing on some examples of playbooks or roles in isolation against a switch or firewall wouldn't help network engineer's deal with the real world networking challenges that they encounter every day.

A network engineer could easily type in those commands into a network operating system as they could type commands into an Ansible playbook, so it is important to look at where the use of a configuration management tool such as Ansible adds real business value.

Implementing a new tool in isolation doesn't really help the network teams improve or improve efficiency as a standalone activity, but the modules that have been created in Ansible to manage Arista, Juniper and Cisco are facilitators of process that help simplify

and standardise processes and approaches, but it really is the process that wraps them that utilises these modules that is the key differentiator.

Ansible can be used to help with network operations in many ways but it is good to try and categorise tasks into the following categories:

- Desired State
- Change Requests
- Self-Service Operations

Desired State

A day one set of playbooks are initially used to set the desired state of the network, utilising a set of roles and modules to build out brand new network devices and are used to set up the networks intended state. An example of a day one playbook could be the first time a network engineer needs to configure a leaf spine architecture utilising Arista leaf and spine switches, which can seem a pretty daunting activity at first. But the beauty is that the state of the whole underlay network could be described in Ansible, but the same can be said for a firewall or any other device.

In the case of the leaf spine network activities will include configuring multiple leaf and spine switches, so creating a set of roles to abstract the common operations and calling them from a playbook is desirable, as the same configuration will need to be carried out on multiple servers.

A network engineer will begin by setting up the **Ansible Control Host** as covered in the Ansible Pre-requisites section. They will then create their inventory file for the leaf spine architecture to configure the network devices.

The network engineer should define the inventory for all the network devices they plan to configure, in the below example we see two host groups containing two spine switches and four leaf switches:

```
[spine]
spineswitch[01-02]

[leaf]
leafswitch[01-04]
```

The network operator will also need to specify the playbook containing the roles that they wish to execute in the spine.yml playbook as shown below to first build out the spine switches with the desired configuration.

In the example playbook below we see the playbook targets the spine host group and executes common, interfaces, bridging, ipv4 and bgp roles against the servers:

```
---
- hosts: spine
  gather_facts: no
  connection: local

  roles:
    - common
    - interfaces
    - bridging
    - ipv4
    - bgp
```

The executed roles carry out the following configuration:

- **Common role**: is used to configure the ip routing table on the spine
- **Interfaces role**: is used to configure interfaces on the spine
- **Bridging role**: is used to configure all necessary vlans and switch ports on the spine
- **Ipv4 role**: is used to configure the spines ip interfaces
- **Bgp**: is used to configure BGP protocol to allow the switches to be meshed together

All these re-usable roles combined will be used to configure the Arista spine switches and utilise the eos_command module heavily.

Similarly a lot of the same modules can be utilised to configure the leaf switches in the leaf.yml playbook which targets the leaf host group in the inventory and executes common, interfaces, bridging, ipv4, bgp, ecmp and mlag roles as shown below:

```
---
- hosts: leaf
  gather_facts: no
  connection: local

  roles:
    - common
    - interfaces
    - bridging
    - ipv4
    - bgp
    - ecmp
    - mlag
```

The executed roles are used to carry out the following configuration:

- **Common role**: used to configure the ip routing table on the spine
- **Interfaces role**: used to configure interfaces on the spine
- **Bridging role**: used to configure all necessary vlans and switch ports on the spine
- **Ipv4 role**: used to configure the spines ip interfaces
- **Bgp**: used to configure BGP protocol to allow the switches to be meshed together
- **Ecmp**: used to ensure equal cost multi-pathing is configured in the leaf spine topology
- **Mlag**: used to configure the switches redundantly at top of rack using mlag

This shows that roles can be reused if they are kept granular enough, with var files providing the necessary configuration changes to the roles so it is important to avoid any hardcoded values.

So the leaf spine build out is a day one playbook but why should a network engineer be interesting in taking all this time to set this up? This of course is a common misconception these playbooks and roles have described the whole desired state of the network and once the initial roles are written it can be used to mutate the desired state of the network at any point in the future.

The Ansible playbooks and roles could also be used to build second data center in the same way, used as a disaster recovery solution, help to mutate the state if a data center re-ip is required or even scale out more spine and leaf switches in the data center.

Taking the last example, in terms of scaling out a data center, this would be as simple as adding more spine or leaf switches to the Ansible inventory, once the additional Arista switches have been zero touch provisioned after being racked and cabled by a data center operations team.

The network operator would then only need to make a small update to the var files to

specify the vlans that need to be used and update the inventory.

In the example shown below the infrastructure is scaled to fifteen spine switches and fourty four leaf switches by modifying the inventory file:

```
[spine]
spineswitch[1-15]

[leaf]
leafswitch[1-44]
```

A pretty extreme scale out example, but this should highlight the point and benefits of investing in automation, as such a scale out would take a network engineer weeks, while Ansible can carry out the same operations in minutes once the initial roles have been built out.

So it really is worth the investment, this also means that the switches are built out consistently the same way as all the other switches, which alleviates manual error and makes the delivery of network changes more precise. Some people believe that automation is all about pace but in networking it should really be about consistency.

The same spine.yml and leaf.yml playbooks could also be executed against existing switches during the scale out, as Ansible is idempotent by nature, meaning only state changes will be pushed to the switches if the configuration has changed. If roles are not idempotent then the modules being called are at fault.

This idempotency means the same day one playbook forming a site.yml that calls both spine.yml and leaf.yml could be run over existing switches and not change any configuration and be re-used without having to target just the changed switches. It is important to note that all Ansible changes should be tested against a test environment before being run in production.

Change Requests

So network engineers now need a separate process for change requests right? They could do if they wish to break the desired state that has been described in the day one playbooks. All network changes going forward should be pushed through the same configuration mechanism; there should be no such thing as a separate stream of work or an ad-hoc command.

Making changes outside the process will only serve to break the Ansible playbooks and

roles that were used to maintain the desired state outdated and break the automation. It is important to note that utilising network automation is an all or nothing approach that needs to be adopted by all team members and no changes should be done outside of the process or it breaks the model of repeatability and reliable changes. If features are lacking the day one playbooks should be extended to incorporate the changes.

Self-Service Operations

With the use of Ansible for network operations, one of the typical bottle necks is the reluctance for network engineers to give development teams access to carry out network changes themselves, so this places a bottle neck on networking teams as typically a company will have more developers than network engineers.

This reluctance is because network changes are traditionally complex and a developer's forte is to develop code and create applications, not log onto networking devices to make firewall changes for their application.

However, if network engineers created a self-service playbook that defined a safe set of workflow actions, then developers could use to interface with network devices in a safe way, this opens up a whole world of opportunity to remove that bottleneck.

This puts network engineers in the position in a subject matter expert (SME) role to help architect and use their network experience to create network automation that embodies networking best practices, to serve the needs of development teams.

This is instead of network engineers carrying out manual actions such as opening firewall ports manually when a developer raises a ticket, it is of course a change in role, but an automated approach is the way the industry is evolving.

Take the example of a firewall request, a development team have created a new application and need a test environment to deploy it in. When configuring the test environment it needs networking and a network engineer will ask the developer the ports they need opened in the firewall.

The developer doesn't know how to answer this question yet as they haven't finalised the application and want to start incrementally developing it in the test environment. Therefore, each time a new port needs to be opened; it means a new network ticket is required to open the incremental port the development team discovers. This is not the optimum use of the network engineer or the developer's time as it causes frustration on both sides. A network engineer's time is better spent optimising the network or adding improved alerting, not processing tickets to open firewall ports.

Instead Ansible could be used to create a self-service file. A developer could create a jinja2 template that could be checked into source control that lists the configuration file used to make firewall changes using the `template:` module. This shows the existing firewall line items and is available to developers to add new line items and submit a pull request to open a port on the firewall.

The network engineer then reviews the change and approves or rejects it. Ansible upon approval can be automatically triggered to push the change to a test environment; this makes sure the config is valid.

In the example below we see the playbook which replaces the `firewall.config` file with the updated jinja2 `firewall.j2` template and then reloads the firewall configuration from the new template:

```
tasks:
  - name: Replace firewall module
    template: src=/firewall_template/firewall.j2 dest=/etc/firewall.conf owner=bin group=admin mode=0644
  - name: Reload config
    fw_config: state=reload
```

This allows network teams to enable self-service model. This speeds up the pace of network changes. It also removes the networking team as the bottleneck and pushes them to create appropriate tests and controls for network changes.

Self-service doesn't mean network engineers are no longer required, far from it, it means that they become the gate keepers of the process instead of constantly rushing to keep up with the never ending chain of ad-hoc requests they receive on a daily basis.

Summary

In this chapter we have looked at how Ansible can be used for server side configuration management of network devices and looked at some of the industry leaders network vendors such as Arista, Cisco and Juniper have all changed their operational models to use open standards and protocols that are well suited to automation.

However, one of Ansibles main strengths is its ability to orchestrate API's and help schedule software releases. Load balancing applications is a fundamental piece of the software development release process, so in the the following chapter we will look at configuration management principles that can help orchestrate load balancers and help networking teams easily maintain complex load balancing solutions.

5
Orchestrating Load Balancers Using Ansible

This chapter will focus on some of the popular load balancing solutions that are available today and the approaches that they take to load balancing applications.

With the emergence of cloud solutions such as AWS and OpenStack we will look at the impact this has had on load balancing at distributed load balancing approaches as well as the traditional centralised load balancing strategy. This chapter will show practical configuration management processes that can be used to orchestrate load balancers using Ansible to help automate the load balancing needs for applications.

In this chapter the following topics will be covered:

- Centralised and Distributed Load Balancers
- Popular Load Balancing Solutions
- Load Balancing Immutable and Static Servers
- Using Ansible to Orchestrate Load Balancers

Centralised and Distributed Load Balancers

With the introduction of micro-service architectures allowing development teams to make changes to production applications more frequently as they do not need to release the full application each time. Developers have moved away from building monolith applications, as implementing a micro-service architecture breaks an application into smaller manageable chunks, allowing application features to be released to customers on a more frequent basis, as the business does not have to re-deploy the whole product each time they release. This means only a small micro-service needs to be re-deployed when a new feature is released

and that different teams can own different micro-services if they work in different locations or time zones this can be useful.

This of course means development teams need a good way of testing dependencies and the onus is put on adequate testing and the development mock services so micro-service applications can be effectively tested against multiple software versions, so it is a huge shift in mind-set for a business but a necessary one to compete. This has meant that being able to utilise the same load balancing in test environments as production has become even more important, so the control of some of the load balancing should ideally sit with development teams as opposed to being a request to the network team due to how dynamic changes need to be.

As a result of the shift towards micro-services architectures, the networking and load balancing landscape has needed to evolve too to support those needs. Both micro-service and monolith applications still need to be supported in the data center today though, as a lot of legacy monolith applications still exist. So although the end goal for a business is a micro-service architecture, the reality for most companies is having to adopt a hybrid approach catering to centralised and distributed load balancing methods.

Centralised Load Balancing

Traditionally load balancers were installed as external physical appliances that had routing set-up to serve web-sites, terminate SSL requests on the physical appliances, then route requests by context switching or serving requests directly to the appropriate backend servers based on the packet headers before serving content or carrying out a transaction and displaying the result to the end user.

This was optimal for monolith configurations as applications typically were self-contained or followed a 3 tier model with a front end web-server, utilising stateful firewalls, with a business logic and database layer part of the network. This didn't require a lot of east to west traffic within the network as the traffic was north to south and networks were designed to minimise the amount of time taken to process the request and serve it back to the end user and it was always served by the core network each time.

Distributed Load Balancing

With the evolution towards micro-service applications, the way that applications operate has changed somewhat. Applications are less self-contained and need to talk to dependant micro-services applications that exist within the same tenant network, or even across multiple tenants.

This means that east-west traffic within the data center is much higher and that traffic in the data center doesn't always go through the core network like it once did. Clusters of micro-services applications are instead instantiated and then load balanced within the tenant network using x86 software load balancing solutions with the end-point of the micro-services clusters VIP exposed to adjacent micro-services that need to utilise it.

With the growing popularity of virtual machines, containers and overlay networks, this means that software load balancing solutions are now used to load balance applications within the tenant network, as opposed to having to pin back to a centralised load balancing solution.

The traditional load balancing vendors have had to adapt and produce virtualised or containerised versions of their physical appliances to stay competitive with open source software load balancing solutions, which are typically used with micro-services.

Popular Load Balancing Solutions

As applications have moved from monoliths to micro-services, load balancing requirements have undoubtedly changed. Today we have seen a move towards open source load balancing solutions, which are tightly integrated with virtual machines and containers to serve east to west traffic between a VPC in AWS or a tenant network in OpenStack as opposed to pinning out to centralised physical appliances.

Open source load balancing solutions are now available from Nginx and HAProxy to help developers load balance their applications or AWS elastic load balancing feature. Just a few years ago Citrix Netscalers and F5 Big IP solutions had the monopoly in the enterprise load balancing space but the load balancing landscape has changed significantly with a multitude of new solutions available.

New load balancing start-ups such as AVI networks focus on x86 compute and software solutions to deliver load balancing solutions, which have been created to assist with both modern micros-service applications and monolith applications to support both distributed and centralised load balancing strategies.

The aim of this book is not about which load balancing vendor solution is the best, there is no *one size fits all* solution and the load balancing solution chosen will depend on traffic patterns, performance and portability that is required by a business.

This book will not delve into performance metrics, its goal is to look at the different load balancing strategies that are available today from each vendor and the configuration management methods that could be utilised to fully automate and orchestrate load balancers and help network teams automate load balancing network operations.

Citrix Netscaler

The **Citrix Netscaler** provides a portfolio of products to deal with load balancing requirements. The Netscaler has various different products available to end users such as the MPX, SDX, VPX and more recently the CVX appliances, with flexible license costs available for each product based on the throughput they are requirements.

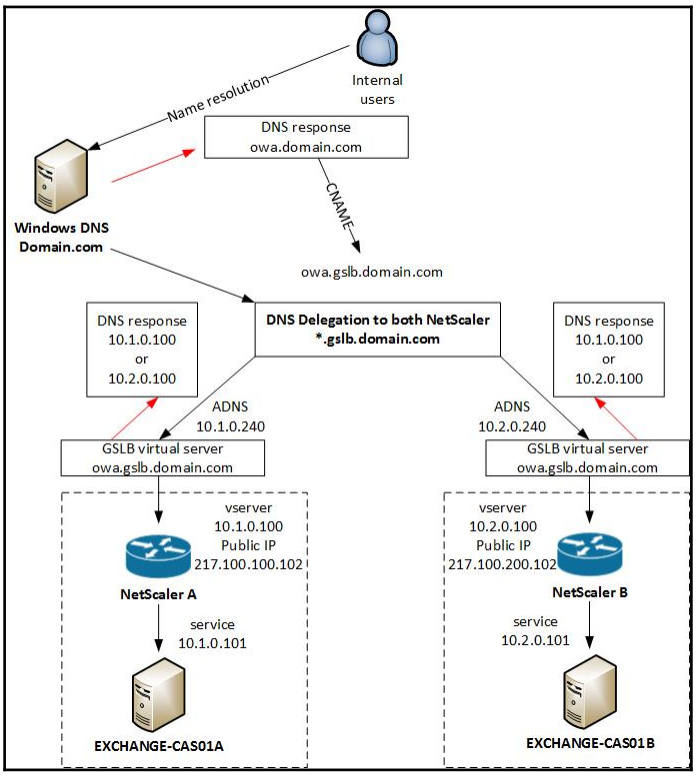
All of the Citrix Netscaler family of products share the same common set of API's. The Citrix Netscaler has a REST API and a Python, Java and C# Nitro SDK which exposes all the Netscaler operations that are available in the GUI to the end user. They can programmatically control the objects and entities that need to be set-up to programmatically control load balancing.

The Netscaler MPX appliance is a centralised physical load balancing appliance that is used to deal with a high number of **transactions per second (TPS)**, the MPX has numerous security features and complies with **Restriction of hazardous substances (RoHS)** and **Federal Information Processing Standard (FIPS)**so the solution can be used by heavily regulated industries that require businesses to comply with certain regulatory standards.

The MPX is typically used to do SSL offloading, as it allows a massive amount of SSL throughput which can be very useful for very highly performant applications so the offloading can be done on the hardware. The MPX can be used to direct traffic to different tenant networks using layer 4 load balancing and layer 7 context switching or alternately direct traffic to a second load balancing tier.

The Netscaler SDX appliance is also a centralised physical appliance that is used to deal with a high number of TPS,although it is not as powerful as the MPX, the SDX allows multiple VPX appliances to be set-up as HA pairs and deployed on the SDX to allow increased throughput and resiliency.

The Netscaler also supports **global server load balancing (GSLB)** which allows load to be distributed across multiple VPX HA pairs in a scale out model utilising a CNAME that directs traffic across multiple HA pairs.



The VPX can be installed on any x86 hypervisor and be utilised as a VM appliance and a new CVX is now available that puts the Netscaler inside a Docker container so they can be deployed within a tenant network as opposed to being set-up in a centralised model. All appliances allow SSL certificates to be assigned and used.

Every Netscaler appliances be it the MPX, SDX, VPX or CVX utilise the same object model and code which has the following prominent entities defined in software to carry out application load balancing:

- **server:** a server entity on a Netscaler binds a virtual machine or bare metal servers ip address to a the server entity, this means the ip address is a candidate for load balancing once it is bound to other Netscaler entities.
- **monitor:** The monitor entity on the Netscaler are attached to services or service groups and provide health checks that are used to monitor the health of attached server entities. If the health check, which could be as simple as a web-ping are not positive the service or service group will be marked as down and the Netscaler will not direct traffic to it.

- **service group:** A service group is a Netscaler entity used to bind a group one or more servers to a lbvserver entity, a service group can have one or more monitors associated with it to health check the associated servers.
- **service:** The service entity is used to bind one server entity and one or more monitor health checks to an lbvserver entity which specifies the protocol and port to check the server on.
- **lbvserver:** An lbvserver entity determines the load balancing policy such as round robin or least connection and is connected to a service group entity or multiple service entities and will expose a virtual ip address that can be served to end users to access web applications or a web service endpoints.
- **gslbvserver:** When DNS load balancing between Netscaler appliances is required a gslbvserver entity is used to specify the gslb domain name and TTL.
- **csvserver:** The csvserver is used to provide layer 7 context switching from a gslbvserver domain or lbvserver ip address to other lbvservers. This is used to route traffic using the Netscaler appliance.
- **gslbservice:** The gslbvservice binds the gslbvserver domain to one or more gslbservers to distribute traffic across Netscaler appliances
- **gslbserver:** The gslbserver entities are is the gslb enabled ip addresses of the Netscaler appliances.

Simple load balancing can be done utilising the server, monitor, service group/service and lbvserver combination. With the gslbvserver and csvserver context switching allowing more complex requirements for complex routing and resiliency.

F5 Big IP

The F5 Big IP suite is based upon F5s very own custom TMOS real time operating system that is self-contained and runs on Linux. TMOS is at the heart of every F5 appliance and allows inspection of traffic, it makes forwarding decisions based on the type of traffic acting much in the same way as a firewall would, only allowing pre-defined protocols to flow through the F5 system.

TMOS also features iRules which are programmatic scripts written using F5s very own **Tool Command Language (TCL)** that enables users to create unique functions triggered by specific events, this could be used to content switch traffic or red-order http cookies, the TCL is fully extensible and programmable and can carry out numerous operations.

The F5 Big IP solution is primarily a hardware load balancing solution, which provides multiple sets of physical hardware boxes that customers can purchase based on their throughput requirements and hardware boxes can be clustered together for redundancy. It

supports

The F5 Big IP suite provides a multitude of products that provide services catering for load balancing and traffic management and even firewalling.

The main load balancing services provided by the F5 Big IP Suite are as follows:

- **Big IP DNS:** F5s global load balancing solution
- **Local Traffic Manager:** the main load balancing product of the F5 Big IP suite

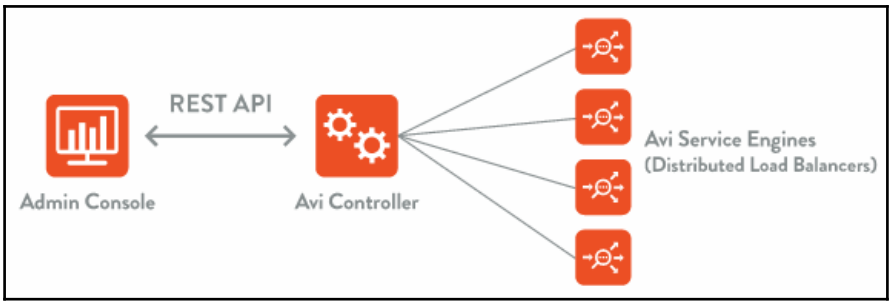
The F5 Big IP solution like Citrix Netscaler implements an object model to allow load balancing to be programmatically defined and virtualised and allows SSL certificates to be associated with entities.

The following local traffic manager object entities allow F5 Big IP to load balance applications:

- **pool members:** The pool member entity is mapped a virtual server or physical servers ip address and can be bound to one or more pools. A pool member can have health monitors associated
- **monitor:** The monitor entity returns the status on specific pool members and acts as a health check
- **pool**: The pool entity is a logical grouping of a cluster of pool members that are associated, a pool can have health monitors associated with it as well as **quality of service (QoS)**
- **virtual severs**: The virtual server entity is associated with a pool or multiple pools, the virtual server determines the load balancing policy such as round robin or least connections. The F5 solution also will offer load balancing solutions based on capacity or fastest connection. Layer 7 profiles utilising iRules can be configured against a virtual server and is used to expose an ip address to access pool members.
- **iRules:** iRules utilise the programmatic TCL so users can author particular load balancing rules based on events such as context switching to different pools.
- **rate classes:** rate classes implement rate shaping and are used to control bandwidth consumption on particular load balancing operations to cap throughput.
- **traffic classes:** traffic class entities are used to regulate traffic flow based on particular events

AVI Networks

AVI networks are a relatively new start-up but have a very interesting load balancing product which truly embraces the software defined mandate. It is an enterprise x86 software load balancing solution comprised of the AVI controller that can be deployed on x86 compute. Hence it is a pure software solution that deploys distributed AVI service engines into tenant networks it integrates with an AWS VPC and an OpenStack tenant.



The AVI Networks solution offers automated provisioning of load balancing services on x86 hypervisors and can automatically scale out to meet load balancing needs elastically based on utilisation rules that users can configure.

The AVI Networks solution supports multiple or isolated tenants and has a real-time application monitoring and analytics engine that can work out where latency is occurring on the network and the locations packets are being routed from. It also supports a display that shows load balancing entities so users have a visual view of load balancing and it additionally supports anti-DDoS support.

All commands that are issued via the GUI or API utilise the same REST API calls. The AVI Networks solution supports a Python and REST API. The AVI Networks object model has numerous entities that are used to define load balancing in much the same way as Netscalers and F5:

- **health monitor profile:** The health monitor pool profile entity specifies health checks for a pool of servers using health attributes.
- **pool:** The pool entityspecifies the ip addresses of virtual or physical servers in the form of a server list and has associated health monitor profiles, it also allows an event to be specified using a datascript if a pool goes down. One or more pools are bound to the virtual service entity.
- **custom policy:** The custom policy allows users to programmatically specify policies against a virtual service

- **app profile:** The app profile entity allows each application to be modeled with associated http attributes, security, DDoS, caching, compression and PKI attributes specified as part of the app profile associated with a virtual service.
- **analytics profile:** Theanalytics profile makes use of the AVI analytics engine and captures threat, metrics, health score as well as latency thresholds and failure codes that are mapped to the virtual service entity.
- **TCP/UDP profile:** The TCP/UDP profile governs if TCP or UDP is used and any DDoS L3/L4 profiles are set.
- **SSL profile:** The SSL entity governs SSL ciphers that will be used by a virtual service entity.
- **PKI profile:** The PKI profile entity is bound to the virtual service entity and specifies CA authority for the virtual service.
- **policy set:** The policy set entity allows users to set security teams to set policies against each virtual service governing request and response polices.
- **virtual service:** The virtual service entity is the entry point ip address to the load balanced pool of servers and is associated with all profiles to define the application pools load balancing and is bound to the TCP/UDP, app, SSL, SSL cert, policy and analytics profiles.

Nginx

Nginx supports both commercial and open source versions it is an x86 software load balancing solution. Nginx can be used as both an http and TCP load balancer supporting http, TCP and even UDP and can also support SSL/TLS termination.

Nginx can be set-up for redundancy in a highly available fashion using *keepalived* so if there is an outage on one Nginx load balancer it will seamlessly fail over to a backup with zero downtime. Nginx Plus the commercial offering, is more fully featured than the open source version, supporting features such as active health checks, session persistence and caching.

Load balancing on Nginx is set-up by declaring syntax in the nginx.conf file it works on the principle of wanting to simplify load balancing. Unlike Netscalers, F5s and AVI Networks it does not utilise an object model to define load balancing rules, instead Nginx describes load balanced virtual or physical machines as backend servers using declarative syntax.

In the following simple example we see three servers 10.20.1.2, 10.20.1.3 and 10.20.1.4 all load balanced on port 80 using Nginx declarative syntax and it is served on www.devopsfornetworking.com/devops_for_networking:

```
http {
  upstream backend {
        server 10.20.1.2;
        server 10.20.1.3;
        server 10.20.1.4;
  }

  server {
    listen 80;
    server_name www.devopsfornetworking.com;
    location / {
      proxy_pass http://devops_for_networking;
    }
```

By default Nginx will load balance servers using round robin load balancing method, but it also supports other load balancing methods.

The Nginx least_conn load balancing method forwards to backend servers with the least connections at any particular time, while the Nginx ip_hash method of load balancing means that users can tie the same source address to the same target backend server for the entirety of a request. This is useful as some applications require that all requests are tied to the same server using sticky sessions while transactions are processed.

While the proprietary Nginx Plus version supports an additional load balancing method called least_time, which calculates the lowest latency of backend servers based on number of active connections and subsequently forwards requests appropriately based on those calculations.

The Nginx load balancer uses a weighting system at all times when load balancing; all servers by default have a weight of 1. If a weight other than 1 is placed on a server it will not receive requests unless the other servers on a backend are not available to process requests. This can be useful when throttling specific amounts of traffic to backend servers.

In the example below we see that the backend servers have load balancing method least connection configured. Server 10.20.1.3 has a weight of 5 meaning only when 10.20.1.2 and 10.20.1.4 are maxed out will requests is sent to 10.20.1.3 backend server:

```
http {
  upstream backend {
        least_conn;
        server 10.20.1.2;
        server 10.20.1.3 weight=5;
        server 10.20.1.4;
  }

  server {
    listen 80;
    server_name www.devopsfornetworking.com;
    location / {
      proxy_pass http://devops_for_networking;
    }
```

By default using round-robin load balancing in Nginx won't stop forwarding requests to servers that are not responding so it utilises max_fails and fail_timeouts for this.

In the example below we can see server 10.20.1.2 and 10.20.1.4 have the max_fail count of 2 and a fail_timeout of 1 second, if this is exceeded then Nginx will stop directing traffic to these servers:

```
http {
  upstream backend {
        server 10.20.1.2 max_fails=2  fail_timeout=1s;
        server 10.20.1.3 weight=5;
        server 10.20.1.4 max_fails=2  fail_timeout=1s;
  }

  server {
    listen 80;
    server_name www.devopsfornetworking.com;
    location / {
      proxy_pass http://devops_for_networking;
    }
```

HAProxy

HAProxy is an open source x86 software load balancer that is session aware and can provide layer 4 load balancing and also carry out layer 7 context switching based on the content of the request as well as SSL/TLS termination. HaProxy is primarily used for http load balancing and can be set-up in a redundant fashion using keepalived configuration, using two apache configurations, so if the master fails then the slave will become the master to make sure there is no interruption in service or end users.

HAProxy uses declarative configuration files to support load balancing as opposed to an object model that proprietary load balancing solutions such as Netscaler, F5 and AVI Networks have adopted.

The HAProxy configuration file has the following declarative configuration sections to allow load balancing to be set-up:

- **backend:** A backend declaration can contain one or more servers in it, backend servers are added in the format of a DNS record or an ip address. Multiple backend declarations can be set-up on a HAProxy server. The load balancing algorithm can also be selected such as round robin or least connection.

  In the example below we see two backend servers 10.11.0.1 and 10.11.0.2 load balanced using the round-robin algorithm on port 80.

```
backend web-backend
   balance roundrobin
   server netserver1 10.11.0.1:80 check
   server netserver1 10.11.0.2:80 check
```

- **check:** Checks avoid users having to manually remove a server from the backend if for any reason it becomes unavailable and mitigates outages. HAProxys default health always attempts to establish a TCP connection to the server using the default port and ip. HAProxy will automatically disable servers that are unable to serve requests to avoid outages. Servers will only be re-enabled when it passes its check. HAProxy will report whole backends as unavailable if all servers on a backend have failed their health checks.

  A number of different health checks can be put against backend servers by utilising the option {health-check} line item, for example tcp-check in the example below is shown which can check on port 8080s health even though port 443 is being balanced

```
backend web-backend
    balance roundrobin
    option tcp-check
    server netserver1 10.10.0.1:443 check port 8080
    server netserver2 10.10.0.2:443 check port 8080
```

- **access control list (acl) :** access control list declarations are used to inspect headers and forward to specific backend servers based on the headers. An acl in HAProxy will try to find conditions and trigger actions based on this.
- **frontend:** The frontend declaration allows traffic different kinds of traffic to be supported by the HAProxy load balancer.

In the example below HAProxy will accept http traffic on port 80, with an acl matching requests only if the request starts with /network and forwarding it to the high-perf-backend if the acl /web-network is matched

```
frontend http
  bind *:80
  mode http
  default_backend web-backend
  acl www.devopsfornetworking.com /web-network
  use_backend high-perf-backend if web-network
```

Load Balancing Immutable and Static Infrastructure

With the introduction of public and private cloud solutions such as AWS and OpenStack there has been a shift towards utilising immutable infrastructure from the traditional static servers. This has raised a point of contention with the *pets versus cattle* or as Gartner defines it *bi-modal*.

Gartner has said that two different strategies need to be adopted one for new micro-services *cattle* and one for legacy infrastructure *pets*. *Cattle* are servers that are killed off once they have served their purpose or have an issue, typically lasting one release iteration, while *pets* are servers that will have months or years of uptime and will be patched and

cared for by operations staff.

It is said that a *cattle* approach favours the stateless micro-service, where a *pet* on the other hand is any application that is a monolith or potentially a single appliance or something that contains data such as a database.

Immutable infrastructure and solutions such as OpenStack and AWS are said by many to favour only the *cattle*, with monoliths and databases remaining pets needing a platform that caters for long lived servers.

Personally, I find the *pets* versus *cattle* debate to be a very lazy argument and somewhat tiresome. Instead it should be treated as a software delivery problem and a question of stateless read applications, stateful applications and data. Cloud ready applications still need data and state so I am puzzled by the distinction.

However, what is undisputed is that the load balancer is key to immutable infrastructure, as at least one version of the application always needs to be exposed to a customer or other micro-services to maintain that applications incur zero downtime and remain operational at all times.

Static and Immutable Servers

Traditionally an environment management team was used by companies to rack, cable, boot, install an operating system and patch servers, before making the physical server available to developers.

Static infrastructure can still exist within a cloud environment, for example databases are still typically deployed as static servers, given the volume of data that is held. Static servers therefore mean a set of long lived servers that typically will contain state.

Immutable servers on the other hand mean that virtual machines are always deployed fresh with a new operating system and software release on them. So this moves away from the pain of doing in place upgrades and makes sure that snowflake server configuration doesn't exist. How many times when doing release management has a release work on four out of five machines and hours or days were wasted debugging why a particular software upgrade wasn't working on a particular server. Immutable infrastructure builds servers from a known state promoting the same configuration from quality assurance, integration, performance testing and production.

Parts of cloud infrastructure that can be made completely immutable to reap these benefits. The operating system is one such candidate, rather than doing in place patching, a single golden image can be patched and updated using automation such as Packer or OpenStacks

Disk Image Builder. Then it will can be used to deploy virtual machines or containers using that base operating system.

Applications that are required to maintain a caching layer are more stateful by nature as that cache needs to be available at all times to serve other applications. These caching applications should be deployed as clusters, which are load balanced, and rolling updates will be done to make sure one version of the cache data is always available and a new software release of that caching layer synchronises to the new release before the pervious release is destroyed.

Data on the other hand is always persistent, so can be stored on persistent storage and then mounted by the operating system when doing an immutable rolling update, mounting either the data on persistent or shared storage in the process. It is possible to separate the operating system and the data to make all virtual machines stateless, for instance OpenStack Cinder can be utilised to store persistent data on volumes that can be attached to virtual machines.

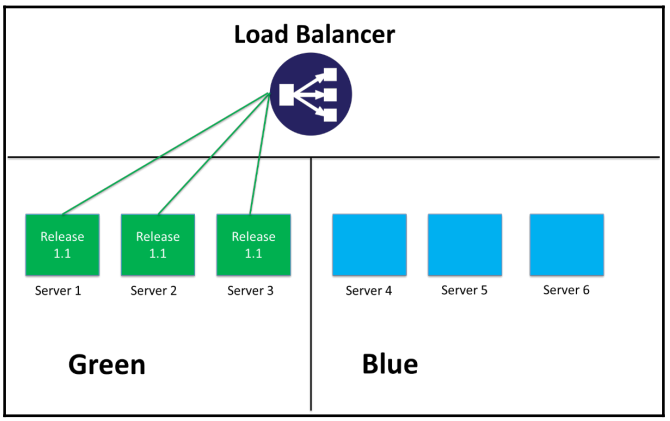
While with all that considered most applications can be designed to be deployed immutably through proper configuration management, even monoliths, as long as they are not a single point of failure. If any applications are they should be re-architected as releasing software should never result in downtime. Although applications are stateful each state can be updated in stages so that an overall immutable infrastructure can be maintained.

Blue/ Green Deployments

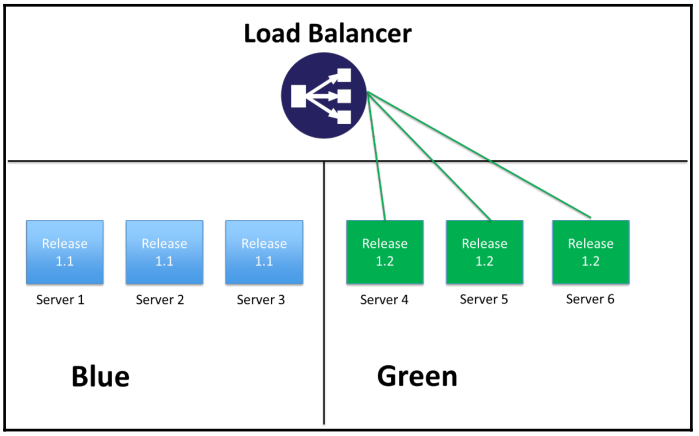
The **blue green deployment** process is not a new concept, before cloud solutions came to prominence, production servers would typically have a dark and love set of servers that would be utilised for carrying out deployments. These are typically known as blue and green servers, which alternated per release. The blue green model in simple terms, means requests that when an upgrade was done, a load balancer by moving over one box at a time or simply DNS load balancing would route traffic to the newly upgraded set of servers and once the TTL had propagated to end users requests would be served by the new boxes. This meant that if there was an issue with the release rollback was not an issue.

A typical blue green deployment process is described below:

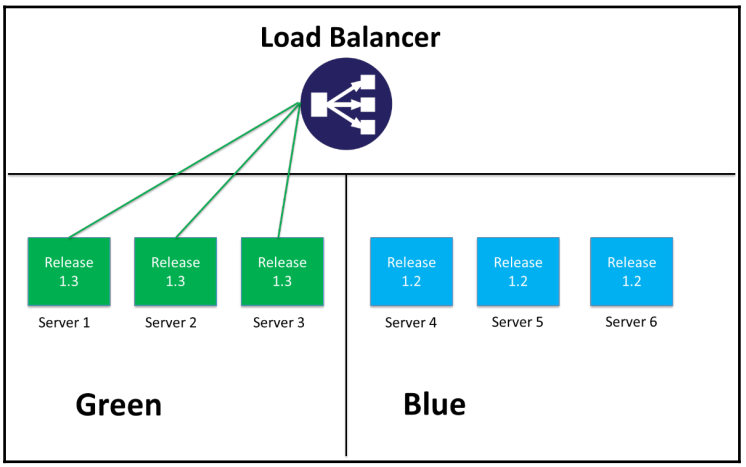
1. **Release 1.1** would be deployed on **servers 1**,**2** and **3** and served on a load balancer to customers and made Green (live).



2. **Release 1.2** would be deployed on **servers 4**, **5** and **6** and then be patched to the latest patch version, upgraded to the latest release and tested. When ready, the operations team would toggle the load balancer to serve boxes **4**, **5** and **6** as the new production release as shown below and the previously Green (live) deployment would become Blue and vice-versa:

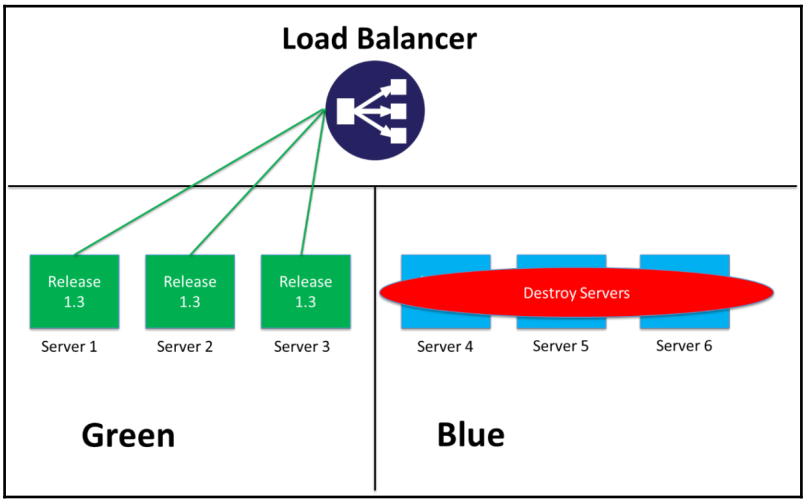


3. When the operations team came to do the next release **servers1**, **2** and **3** would be patched to the latest version, upgraded to **release 1.3** from **release 1.1**, tested and when ready the operations team would direct traffic to the new release utilising the load balancer making **release 1.2** Blue and **Release 1.3** Green as shown in the following figure.



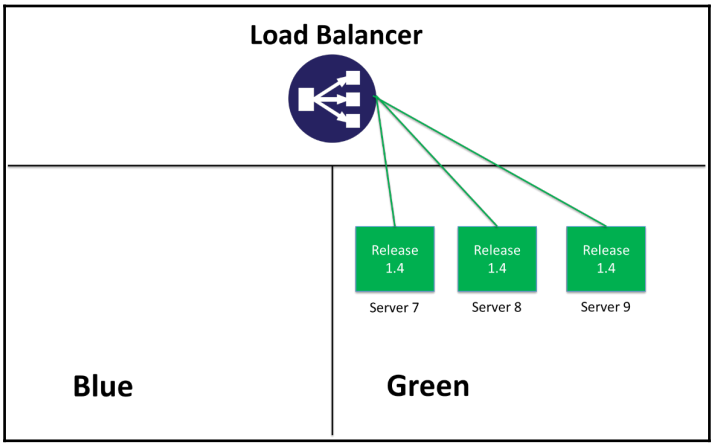
This was typically the procedure of running a blue green deployment using static servers.

However, when using an immutable model, instead of utilising long lived static servers such as **Servers 1**,**2**,**3**,**4**,**5** and **6**, after a release was successful the servers would be destroyed as shown below as they have served their purpose:



The next time **servers 4**, **5** and **6** were required, instead of doing an in place upgrade, three new virtual machines would be created from the golden base image in a cloud environment. These golden images would already be patched up to the latest version, so

brand new **servers 7**,**8** and **9** with the old servers destroyed and the new **release 1.4** would be deployed on them as shown below, once **server 7**,**8** and **9** were live **servers 1**,**2** and **3** would be destroyed as they have served their purpose:



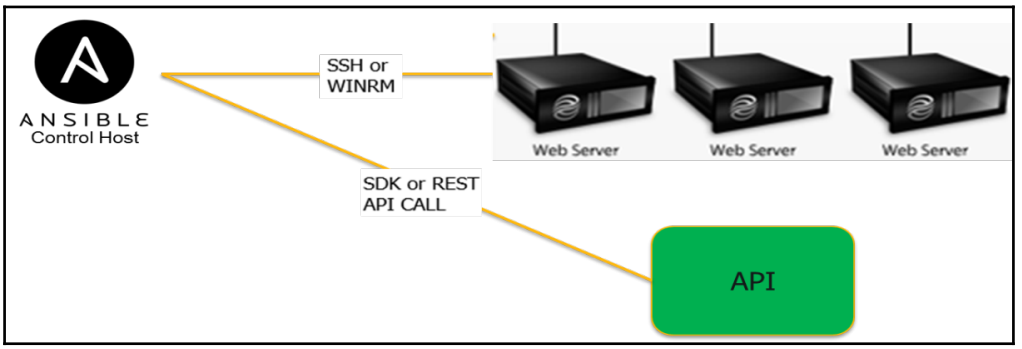
Using Ansible to Orchestrate Load Balancers

In Chapter 4 we covered the basics of Ansible and how to use an Ansible Control host, playbooks, and roles for configuration management of network devices. Ansible though has multiple different core operations that can help with orchestrating load balancers which we will look at in this chapter.

Delegation

Ansible delegation is a powerful mechanism that means from a playbook or role Ansible can carry out actions on the target servers specified in the inventory file by connecting to them using SSH or WINRM or alternately execute commands from the Ansible Control Host.

The diagram below shows these two alternative connection methods with the Ansible Control host either logging into boxes using SSH or WINRM to configure them or running an API call from the Ansible Control Host directly:



Both of these options can be carried out from the same role or playbook using delegate_to which makes playbooks and roles extremely flexible as they can combine API calls and server side configuration management tasks.

An example of delegation can be found below where the Ansible extras HAProxy modules are used, with the delegate_to used to trigger an orchestration action that disables all backend services in the inventory file:

```
tasks:
- name: disable server in networking backend pool
  haproxy: state=disabled host={{ inventory_hostname }} backend=networking
  delegate_to: 127.0.0.1
```

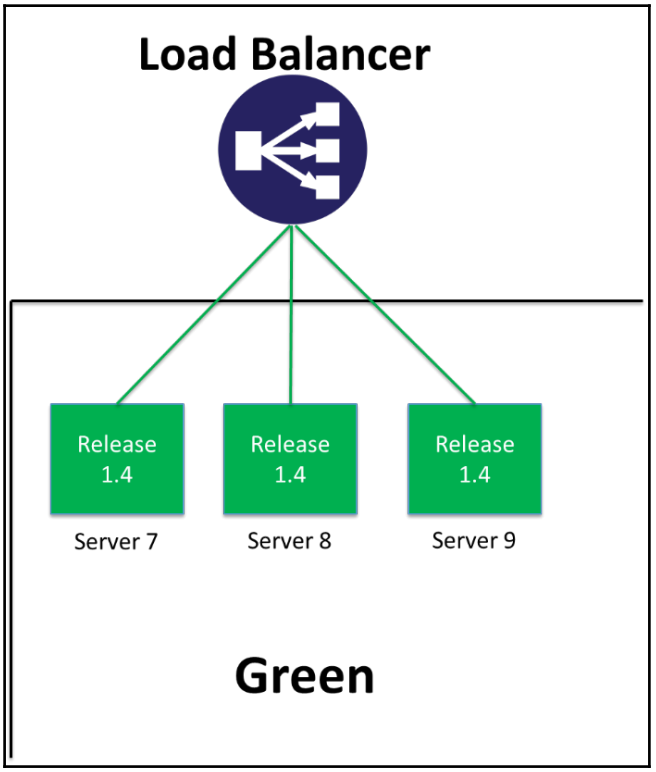
Rolling Updates

In order to release software without interruptions to service, a zero downtime approach is preferable, as it doesn't require a maintenance window to schedule a change or release. Ansible supports a *serial* option which passes a percentage value to a playbook.

The *serial*option allows Ansible to iterate over the inventory and only carry out the action against a percentage of the boxes, completing the necessary playbook, before moving onto the next portion of the inventory. It is important to note that Ansible passes inventory as an unordered dictionary so the percentage of the inventory that is processed will not be in a specific order.

Using the *serial* option that a blue/green strategy could be employed in Ansible where boxes will need to be taken out of the load balancer, upgraded, before being put back into service. Rather than doubling up on the number of boxes three boxes are required as shown below

which all serve **release 1.4**:

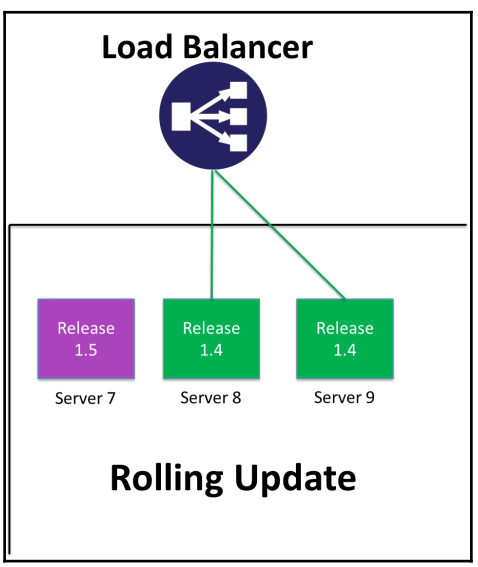


Utilising the Ansible playbook below, using a combination of the delegate_to and serial each of the servers can be upgraded using a rolling update:

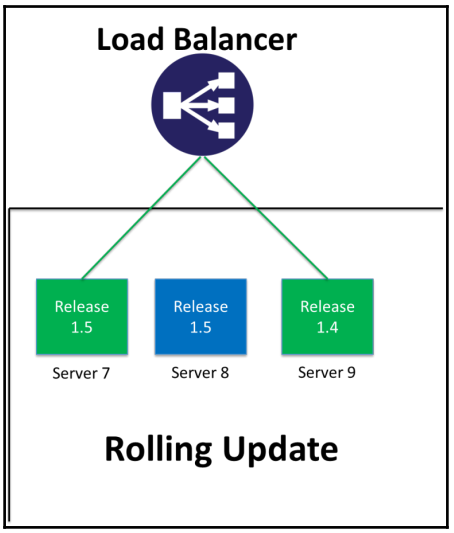
```
---

- hosts: application1
  serial: 30%

  tasks:

  - name: take out of load balancer pool
    haproxy: state=disabled host={{ inventory_hostname }} backend=backend_nodes
    delegate_to: 127.0.0.1

  - name: actual steps would go here
    yum: name=application1-1.5 state=present

  - name: add back to load balancer pool
    haproxy: state=enabled host={{ inventory_hostname }} backend=backend_nodes
    delegate_to: 127.0.0.1
```

The playbook will execute the following steps:

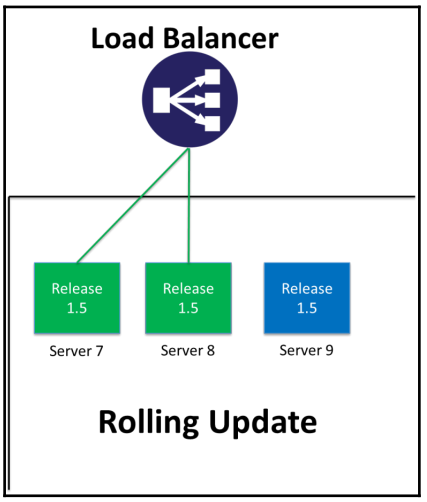
1. The serial 30% will mean that only one server at a time is upgraded. So **Server 7** will be taken out of the HAProxy backend_nodes pool by disabling the service calling the HAProxy using a local delegate_to action on the Ansible Control Host. A yum update will then be executed to upgrade server version new application1 release **version 1.5** on **server 7** as shown below:



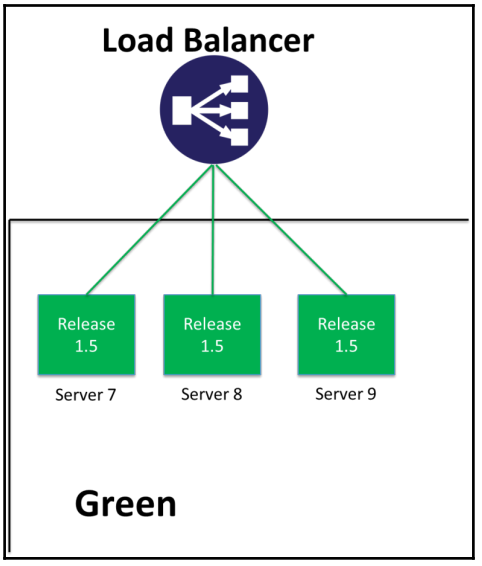
2. **Server 7** will then be enabled again and put into service on the load balancer using a local delegate_to action. The serial command will iterate onto server 8 and disable it on HAProxy, before doing a yum update to upgrade server version new **application1** release **version 1.5** as shown below:



3. The rolling update will then enable **server 8** on the load balancer and the serial command will iterate onto **server 9**, disabling it on HAProxy before doing a yum update which will upgrade the server with the new **application1** release **version 1.5** alternating when necessary between execution on the local server and the server as shown below:



4. Finally the playbook will finish by enabling **server 9** on the load balancer and all servers will be upgraded to **Release 1.5** using Ansible as shown below:



Dynamic Inventories

When dealing with cloud platforms using just static inventories is sometimes not enough. It is useful to understand the inventory of servers that are already deployed within the estate and target sub-sets of them based on characteristics or profiles.

Ansible has an enhanced feature called the dynamic inventory, allows users to query a cloud platform of their choosing with a python script, this will act as an auto-discovery tool that can be connected to AWS or OpenStack, returning the server inventory in JSON format.

This allows Ansible to load this JSON file into a playbook or role so can be iterated over, in the same way a static inventory file can be via variables.

The dynamic inventory fits into the same command line constructs, although instead of passing the following static inventory:

```
ansible-playbook -I inventories/inventory -l qa -e current_build=9 playbooks/add_hosts_to_netscaler.yml
```

Then a dynamic inventory script openstack.py for the OpenStack cloud provider could be passed instead:

```
ansible-playbook -I inventories/openstack.py -l qa -e environment=qa playbooks/devops-for-networking.yml
```

The dynamic inventory script can be set-up to allow specific limits, in the above case the only server inventory that has been returned is the quality assurance servers which is controlled using the -l qa limit.

Typically when using Ansible with immutable servers the static inventory file can be utilised to spin up new virtual machines, while the static inventory can be used to query the estate and do supplementary actions when they have already been created.

Tagging Meta-data

When using dynamic inventory in Ansible, meta-data becomes very important component, as servers deployed in a cloud environment can be sorted and filtered using meta-data that is tagged against virtual or physical machines.

When provisioning AWS or OpenStack instances in a public or private cloud, meta-data can be tagged against servers to group them.

In the example below we can see a playbook creating new OpenStack servers using the os_server OpenStack module, it will iterate over the static inventory, tagging each newly created group and release meta-data on the machine:

```
tasks:

- os_server:
     state: present
     name: "{{ inventory_hostname }}"
     image: centos6
     flavor: 4
     nics:
       - net-name: network1
     meta:
       group: qa
       release: 9
```

The dynamic inventory can then be filtered using the -l argument to just specify boxes with the group qa. This will return a consolidated list of servers.

Jinja2 Filters

Jinja2 filters allow Ansible are also very useful, as they control conditions in which to execute a particular command from a playbook or role. There are a wide variety of different jinja2 filters available out the box with Ansible or custom filters can be written.

An example of a playbook using a jinja2 filter would be only add the server to the Netscaler if its meta-data openstack.metadata.build value is equal to the current build version:

```
---

- hosts: application1
  serial: 30%

  tasks:

  - name: "add into load balancer pool"
          server_add_netscaler:
            state: present
            name: "{{ inventory_hostname }}"
            ns_proto: "http"
          delegate_to: 127.0.0.1
          when: openstack.metadata.build == {{ current_build }}
```

So executing the ansible-playbook add_hosts_to_netscaler.ymlwith a limit -l on qa would only return boxes in the qa metadata group as the inventory. Then the boxes can be further filtered at playbook or role using the when jinja2 filter to only execute the add into load balancer pool command if the openstack.metadata.build number of the box matches the

current_build variable of 9:

```
ansible-playbook -I inventories/openstack.py -l qa -e current_build=9 playbooks/add_hosts_to_netscaler.yml
```

The result of this would be that only the new boxes would be added to the Netscaler lbvserver VIP.

Removal of boxes could be done in a similar way in the same playbook with a *not equal to* condition:

```
- name: "remove from load balancer pool"
        server_add_netscaler:
          state: absent
          name: "{{ inventory_hostname }}"
          ns_proto: "http"
        delegate_to: 127.0.0.1
        when: openstack.metadata.build != {{ current_build }}
```

This could all be combined along with the serial percentage to roll percentages of the new release into service on the load balancer and decommission the old release utilising dynamic inventory, delegation, jinja2 filters and the serial rolling update features of Ansible together for simple orchestration of load balancers.

Creating Ansible Networking Modules

As Ansible can be used to schedule API commands against a load balancer, it can be easily utilised to build out a load balancer object model that popular networking solutions such Citrix Netscaler, F5 Big IP or AVI Networks utilise.

With the move to micro-service architectures load balancing configuration needs to be broken out to remain manageable, so it is application centric, as opposed to living in a centralised monolith configuration file.

This means that there are operational concerns when doing load balancing changes, so Ansible can be utilised by network operators to build out the complex load balancing rules, apply SSL certificates and set-up more complex layer 7 context switching or public ip addresses and provide this as a service to developers.

Utilising the Python API's provided by load balancing vendors, each operation could then be created as a module with a set of YAML var files describing the intended state of the load balancer.

In the below example we look at how Ansible var files could be utilised by developers to create a service and health monitor for every new virtual server. These services are then bound to the lbvserver which was created by the Network team, with a roll percentage of 10% which can be loaded into the playbooks serial command. The playbook or role is utilised to control out the bindings and actions, while the var file describes the desired state of the objects:

```
---
netscaler:
    lbvserver:
      name: "devops_for_networking"
      subnet: "10.20.124.0/23"
      servicetype: "HTTP"
      lbmethod: "TOKEN"
      rule: HTTP.REQ.HEADER("x-ip").VALUE(0)
      persistencetype: "NONE"
      port: 80

    lbmonitor:
      monitorname: "mon-devops_for_networking"
      type: "HTTP-ECV"
      send: "GET /www/networking/v1.0/health"
      recv: "OK"
      lrtm: "ENABLED"
      downtime: 5

    service:
      servicetype: "HTTP"
      maxclient: 0
      port: 80

    roll_percentage: 10%
```

Summary

In this chapter we have looked at the varied load balancing solutions are available from proprietary vendors to open source solutions and have discussed the impact that micro-services has had on load balancing, moving it from a centralised to distributed model to help serve east west traffic.

We then looked at blue/green deployment models and the merits of immutable and static servers and how software releases can be orchestrated using Ansible in either model. In the process illustrating how useful Ansible is at orchestrating load balancers by utilising dynamic inventory, rolling updates, delegation and jinja2 filters which can all be used to help fulfil load balancing requirements.

In the next chapter we will look at applying these same automation principles to SDN Controllers, primarily focusing on the Nuage solution. It will cover configuring firewall rules and other SDN commands, so the whole network can be programmatically controlled and automated.

6
Orchestrating SDN Controllers Using Ansible

This chapter will focus on SDN controllers and the ways they can enable network teams to simplify their daily tasks.

We will look at why SDN Controllers have been adopted and highlight some of the immediate business benefits they will bring if utilised correctly. It will focus on ways in which network operations need to be divided so network operations can scale, by utilizing automation.

This chapter will discuss the benefits of utilsing software defined networking and look at practical configuration management processes that can be used to orchestrate SDN Controllers API's and object models. Finally we will look at how Ansible can be used to execute and wrap some of these configuration management processes, using Nuage VSP as a practical example.

In this chapter the following topics will be covered:

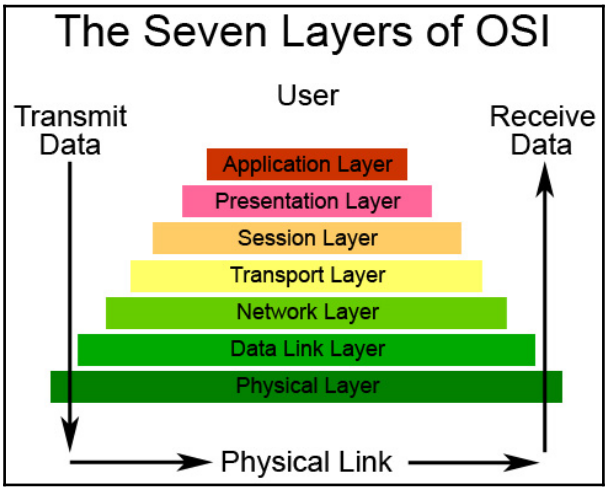
- Arguments against Software Defined Networking
- Why Would A Company Utilise Software Defined Networking
- Splitting Up Network Operations
- Immutable Networking
- Using Ansible to Orchestrate SDN Controllers

Arguments against Software Defined Networking

With the emergence of AWS public cloud, networking is now being treated more like a commodity and has moved from silicon to software. This has allowed developers the ability to mutate the network to best serve the applications, rather than retro fit applications into an aging network, which is likely not optimised for modern micro-service applications.

It would therefore seem nonsensical that any business would want to treat their internal data center networking any differently. However, like all new ideas, before acceptance and adoption comes fear and uncertainty, inherently co-related with the new or different ways of working.

Common arguments against using a clos leaf spine architecture and SDN controllers center around one common theme, that it requires change and change is hard. We then harp back to the mythical 8th layer of the OSI model and that is the *user* layer:



The network operators have to feel comfortable with any solution that is implemented. This is very important, but by the same token the *user* layer is equally the end-user of the networking service that is provided by the network team. So ease of use is important on two levels, both network operations and the self-service operations provided to developers.

Before a company considers putting in software defined networking, they need to be doing it for the correct reasons, make it requirements based, simply implementing a new tool, in this case an SDN Controller, will not solve operational issues alone.

Work out with the business what the new operational model should be and utilise software defined networking as a facilitator for that new business process, focusing on speed of operations and the aim to remove networking as the bottleneck for application delivery. In short network operations needs to be DevOps friendly or it will inhibit software delivery and slow down the software delivery.

Added Network Complexity

Typically some of the arguments used against using overlay networks are that they are more complex than traditional layer 2 networks, with much more moving parts and could cause a bigger variety of failures.

Although the constructs of an overlay and underlay network may be different, it is still relatively new concept and a lot of people fear change. As long as base requirements in terms of network availability, redundancy, performance and speed of change are met then any there should be no reason not to implement software defined networking.

The fear of overlay networks, can be likened to operations staff, that initially argued against the introduction of hypervisors and virtualisation, which was initially viewed as an added layer of complexity or magic box that couldn't be trusted.

However, the portability and opportunities introduced by running a hypervisor, greatly outweighed any performance implications for the vast majority of applications. The benefits included increased portability, flexibility and speed of operations. There are of course edge cases and some applications that don't fit into the virtualised model but the benefits that virtualisation bring for 99% of the data center, meaning, as a business solution it can't really be ignored.

Overlay networks give the same benefits to networking as hypervisors did to servers. Of course when implementing an overlay network, the underlay should be built for redundancy, so that if a failure occurs it occurs on the underlay and does not impact the overlay. A scale out model should be possible on the underlay, with a leaf spine architecture the introduction of more racks paired with leaf switches or even a new spine to prevent over-subscription of links should be possible.

On the theme of added complexity of an overlay network, tells a systems reliability engineer or network engineer that is debugging an ill performing link in a layer 2 spanning tree network that a spanning tree network isn't complex and they will show you the debugging diagram they mapped out to try and solve the issue as evidence of the complexity.

So networks are complex beasts at the best of times, however, by utilising underlay and

overlay networks the focus is that the underlay is scalable and can be easily scaled out by the network operators, while the overlay is what the end users integrate with as part of the self-service so it should have easy to understand software constructs.

If implemented correctly an overlay network ticks both boxes and focuses on componentising the network into two pieces. The overlay is the user friendly, described in software piece, much like AWS or OpenStack and the underlay is the gritty hard-core networking piece that still needs to be well designed by a network architect and built for scale.

Lack of Software Defined Networking Skills

Another argument against not implementing software defined network is lack of skills. There is a viewpoint companies will have to hire completely new staff to implement software defined networking.

However, this can be offset by partnering with an SDN vendor or utilising provided training programmes for staff. It is a business transformation putting in software defined networking, as such network staff will need to skill up over a period of time.

But networking staff like all IT staff will need to evolve and build new skills like other teams in IT. It is a big change at first but good networking staff will be excited and embrace change and benefits that come from implementing software defined networking.

However, change can be daunting and can seem like a monumental cultural shift or effort at times. To initiate successful change in large or even small companies then it usually has to come with top down sponsorship or backing.

Adopting software defined networking will mean changing the businesses operational model and automation will need to be embraced at every level, network tasks in the overlay simply can't be manual when using an SDN controller. The business also needs to look at ways of automating the underlay, in this book we have already looked at the API's can be utilised to configure network devices so really both the underlay and overlay need to be automated.

The term software defined data center is somewhat overused by vendors, but the principles behind it shouldn't really be ignored if a network team wants to provide a great user experience to all sister teams. If any company puts in a software defined networking solution then it will add no true value if automation isn't written to speed up network operations utilising the rich set of API's, if companies are going to put in a software defined network and have network engineers manually enter commands on network devices or use a GUI, the company may as well not bother as they can do that with any out the box switch

or router, they are wasting the opportunity a software defined overlay network offers.

Just putting in the software defined networking solution and still having developers raise network tickets will give a business zero business value, it will not increase efficiency, time to market or reliability of changes. To get the business benefits out of software defined networking it is all or nothing, it is either completely automated or it will over time become fragmented and broken over time. If any network engineer does manual updates then it has the opportunity to break the whole operational model, it changes the desired state of the network and it could break the automation completely.

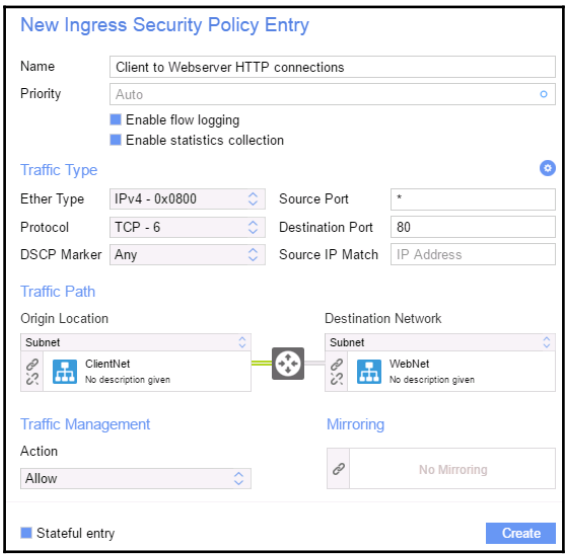
When putting in software defined networking, automate all the common operations first and allow developers to self-serve themselves and make it immutable. In Chapter 3 we already looked at ways of initiating cultural change, humans are creatures of habit, they tend to stick with what they know, network engineers have spent years gathering networking certifications on how to configure spanning tree algorithm and layer 2 networks so this is a huge cultural shift.

Companies Require Stateful Firewalling To Support Regularity Requirements

One of the main issues highlighted with software defined networking has been the lack of stateful firewalling, due to Open vSwitch being based on flow data and being traditionally stateless. Until recently reflexive rules were utilised to emulate stateful firewalling at the kernel user space level.

However, recent developments with Open vSwitch has allowed stateful firewalling to be implemented. So this is no longer an issue with Open vSwitch as **connection tracking(conntrack)** previously only available as part of iptables, has now been decoupled from iptables, meaning that, it is now possible to match on connections as well as flow data.

The Nuage VSP platform has also introduced stateful firewalling as part of its 4.x release, with reflexive rules being replaced for stateful rules for all ICMP and TCP ACL rules on the Nuage VRS, which is a customised version of Open vSwitch:



Why Would A Company Utilise A Software Defined Networking Solution?

Traditionally all good enterprise networks should be built with the following goals in mind:

- Performance
- Scalability
- Redundancy

The network's first and foremost needs to be *performant* to meet customer needs. Customers can be end users in the data center or end users of the application in the public domain. With continuous delivery and deployment, if networking blocks a developer in a test environment, it is hampering a potential feature or bug fix reaching production, so it is not acceptable to have sub-standard pre-production networks and they should be designed as scaled down functional replicas of production.

Scalability focuses on the ability to scale out the network to support company growth and demand, as more applications are added, how does the network horizontally scale? Is it cost effective? Can it easily be adapted to cater for new services such as 3rd party VPN access or point to point network integration? All these points need to be given proper consideration when creating a flexible and robust network design.

Redundancy, is having a network with no single points of failures. This is so that the network can recover from a switch failure or an issue with a core router and not cause outages to customers. Every part of the network should be set-up to maximise uptime.

These three points seem to have been the staple on which good networks were designed and built in the past. However, as applications have moved from monoliths to micro-services, monolith applications have tended to have one time set-up operations and then remained fairly static, while micro-service applications on the other hand have required more dynamic networks that are subject to greater variance of change.

The needs of the modern network have evolved and networks need to be updated rapidly to deal with the requirements of micro-service architectures, without having to wait on a network engineer to process a ticket. With continuous delivery forming feedback loops, it is imperative that the process is quick and lean and issues can be fixed quickly otherwise the whole process will break down and grind to a stand-still.

Software Defined Networking Adds Agility and Precision

Software defined networking or in particular overlay networking, still focuses on performance, scalability and redundancy; they should never be compromised, but also introduces the following benefits:

- Agility
- Mean time to recover
- Precision and Repeatability.

Software defined networking puts the network into a software overlay network with associated object model, which allows the network to be programmable by exposing a rich set of API's. This means that workflows can be used to set-up network functions, the same way infrastructure can be controlled in a cloud or virtualisation environment.

As the network is programmable, requesting a new subnet or making an ACL change can be done as quickly as spinning up a virtual machine on a hypervisor. It removes the traditional blockers or operational inhibitors such as requiring to raise a ticket to a networks operation change to mutate the network for a developer and allows the network to be controlled via the administration of an API call.

Mean time to recover is also improved because network changes are programmable, so network inventory can be source controlled. This versions the network so any change is delivered via source control management and allows network changes to be modular and

easy to track.

If a breaking change has occurred to the overlay network, a version tree in the source control management system can be used to see what has changed since the network was last working release. The same programmable script can then be used to quickly roll back the network change back to the previous version and remove the issue. This is of course the beauty of implementing an immutable network rather than static networks, where the state is always as clean as the day one network and can be rolled forward or back on demand.

Repeatability is catered for by software defined networking using programmatic operational workflows. So that all network changes that carry out the same operation are done in the same way by all users and this can be done using the API workflows approved by the network team.

This means that network changes in a continuous delivery model will be checked into source control, pushed to a test environment using automated workflow actions for the desired operation such as creating a new ACL rule, tested and verified then will be promoted onto the next test environment, applied identically using automation before being applied to production. This repeatability of using an overlay network means all the constructs of a quality assurance environment can be the same as a production environment.

A Good Understanding of Continuous Delivery Is Key

Companies looking to utilise software defined networking should already have a well-established continuous delivery model for code and infrastructure or be locked into building a continuous delivery model for their company before considering software defined networking.

If companies have a mandate of automating everything, which is inclusive of networking functions they would greatly benefit from using an SDN controller to facilitate network automation. This will mean that the company is already set-up to knowledge share and culturally be utilising a DevOps model.

To emphasise the point, if overlay networks are modified by network engineers by hand rather than programmatically, it will bring no business value and the company will have missed the point.

Operational models need to change when implementing software defined networking and if an issue occurs it needs to be built back into the automation to provide the fix.

Automation is built by humans so automated processes need to be continually iterated and improved to become more and more robust over time.

It is important to appreciate that edge cases will occur when using automation and sometimes cause issues. But my implementing continuous improvement processes on the automation, it means the reliance on single network engineers to make network changes is removed.

The highly skilled network engineer can instead build all their knowledge into the automation. This means that every automated network change is done with the same care and precision as the best network engineer in the company.

Those pre-approved and well defined changes can be made by anyone in the company if they are automated, not just the best engineer, so the bottleneck is removed from the network team freeing them up to work on more interesting tasks than the mundane repeatable tasks that are more accurately done using automation.

Software Defined Networking Helps Companies With Over Complex Networks

Companies that have very complex legacy networks would also be a prime candidate to benefit from software defined network, instead of fixing the existing network, which may not be possible due to having to adhere to 99% uptime targets. Instead a new green-field network could be created with an overlay network in parallel to the existing network.

This will teams will migrate their applications to the new network and simplify the complexity of the network in the process, while utilising all the benefits brought by SDN overlay network, routing back to the legacy network for application dependencies.

Likewise, if networks are running more than one hundred hypervisors, this is a scale at which an SDN solution would be of benefit, be it extending OpenStack Neutron capabilities to allow companies to run OpenStack at scale, as opposed to deploying multiple smaller OpenStack clouds to cope with bottlenecks.

Splitting Up Network Operations

With the introduction of software defined networking in a company or business has to be a shift in operational responsibilities. If a company runs multiple micro-service applications, a fairly typical situation is that a company has 100 developers that develop around 200 micro-services, which combined are utilised to deploy the company's customer facing web-site.

Each of the 100 developers are split into a set of delivery teams that contain 10 or so developers, each forming a scrum teams, and each team look after a set amount of micro-services relative to their complexity. The company has 10 network engineers that are required to serve the networking needs of the 100 developers as well as maintaining uptime of the network.

However, in this model if all network operations are done manually then the network engineers will not be able to keep up with the change requests, so they will either have to work late nights and subsequently become burned out or the productivity of those developers will be impacted as the network engineers will become the bottleneck for throughput.

This model will simply not scale, so operational change is required as one network engineer will be required for every 10 developers and in future as the company expands it will want to invest in development staff to create products, it will not expect to have to scale up network staff to support those operations, so automation becomes a must as developer to network engineer ratio may increase:

The business may then look at software defined networking as the solution to solve their scaling problems, with the mind-set of simplifying the network. This means that developers can carry out network changes more quickly to support developer demand.

But simply putting in a software defined networking solution such as CISCO ACI, Juniper Contrail, VMware NSX or Nuage Networks will not help the situation unless processes are automated and the inefficient business processes are addressed.

New Responsibilities in API Driven Networking

The role of a network engineer in a software defined network therefore has to evolve, they have to devolve some power to the developers like operations staff were required to for

creation of infrastructure. But software defined networking shouldn't mean giving complete open access of the API to developers, this is also a recipe for disaster, efficient controls need to be put in place that act as a quality gate not as an inhibitor for productivity.

Some operational workflows in an overlay network should still be controlled by a qualified network engineer and governed by security, but not to the detriment of developer's daily requirements. It wouldn't be fair to expect a developer, to be well versed enough in networking, to log onto a router and set up their routing requirements for their application unaided so there has to be some middle-ground.

Allowing a developer access to network devices in an uncontrolled manner poses the risk of a network outage which goes against one of the three main networking principles and compromises redundancy and network engineers have a responsibility for uptime of the system.

Overlay Architecture Set-up

When setting up an overlay network it will typically be built in a green-field environment as part of an application migration programme and target environment for a legacy network. The application migration could either be done in a piecemeal format or done in one step, where everything is migrated, then switched on as part of a migration big bang go live activity.

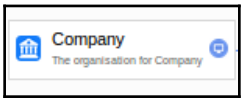
Regardless of the application migration approach, it is very important that the overlay network is set-up to achieve the following goals:

- Agility
- Minimise mean time to recover
- Repeatability
- Scalability

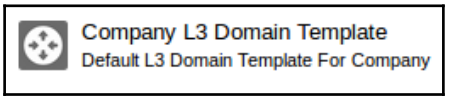
The performance of the network will be determined by the underlay components and silicon used, but the definition of the overlay network in terms of constructs and workflow of the SDN object model need to be correct to make sure that any operation can easily be carried out quickly, repeatability and that the design scales and can support roll-back. The SDN before implementation should be performance tested to make sure the virtualisation overhead does not impact performance.

So to quickly recap on the Nuage VSP object model that was covered in Chapter 2:

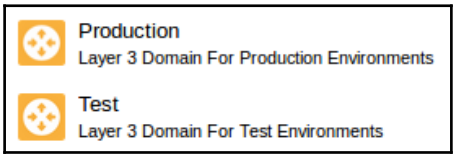
Organisation: Governs all Layer 3 domains



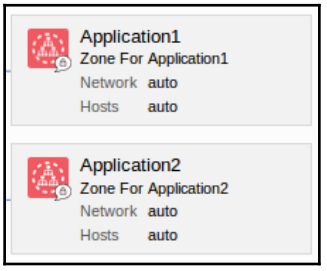
Layer 3 domain Template: A **layer 3 domain template** is required before child layer 3 domains are created. The **Layer 3 domain template** is used to govern overarching default policies that will be propagated to all child layer 3 domains. If a **layer 3 domain template** is updated at template level then the update will be implemented on all Layer 3 domains that have been created underneath it immediately.



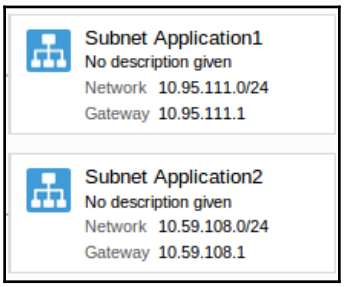
Layer 3 domain: Can be used segment different environments so users cannot hop from subnets deployed in a under a layer 3 **Test** domain to a layer 3 **Production** domain.



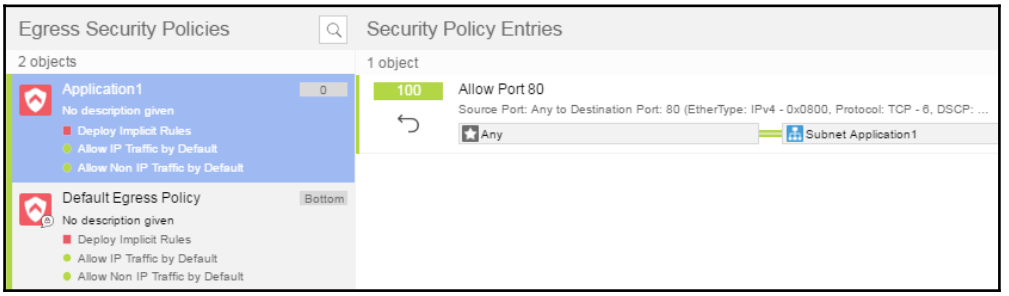
Zones: A zone segments firewall policies at application level, so each micros-service application can have its own zone and associated ingress and egress policy per Layer 3 Domain.



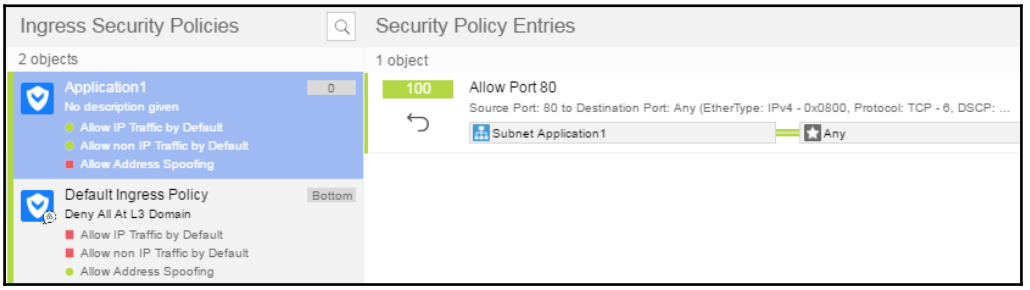
Layer 3 Subnet: This is where VMs or bare metal servers that are deployed. In this example we see **Subnet Application1** and **Subnet Application2**



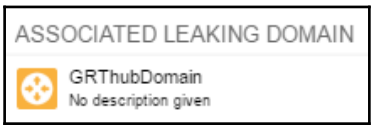
Application Specific Egress Policy: Unique application policies for egress rules that can be used to view each individual applications connectivity rules



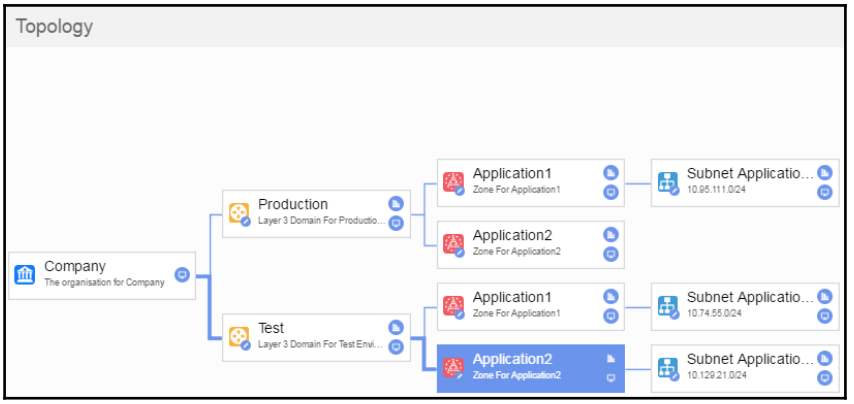
Application Specific Ingress Policy: Unique application policies for ingress rules that can be used to view each individual applications connectivity rules



Leaking Domain: This is used to leak routes into the overlay network via a layer 3 subnet, to bridge connectivity between the green-field network and a legacy network

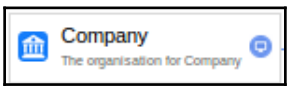


So utilising Nuage VSP as an example we had an organisation, two layer 3 domains dictating **Test** and **Production**, with a zone for each micro-service application encapsulating its unique micro-subnets and virtual machines:

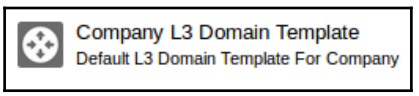


In terms of network set-up automation could be used by network team would be in control of the following constructs in the overlay network:

Organisation:

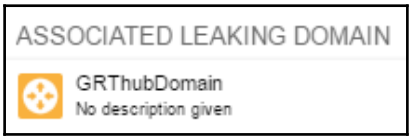


Layer 3 domain Template:



Layer 3 domain:

Leaking Domain:

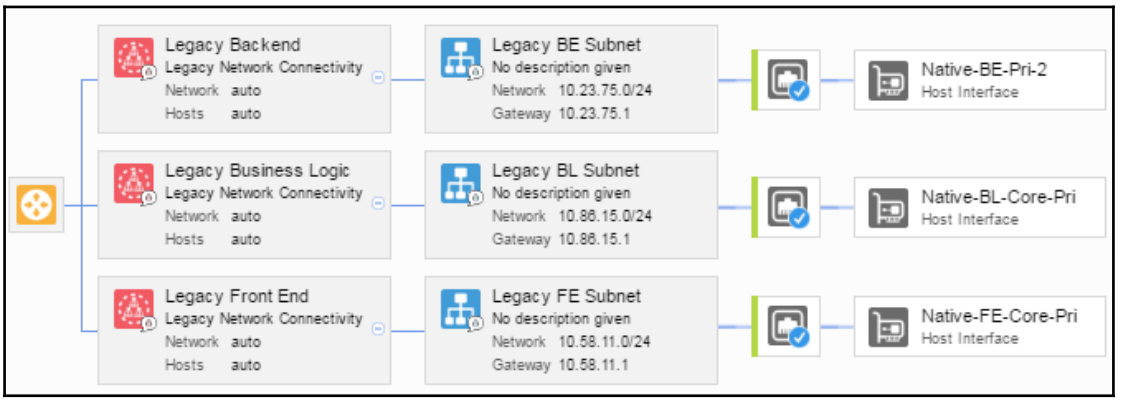


The organization is most likely a *day-one* set-up activity, while the domain template policies can be defined and dictated by the network and security team. Any security policies applied across all networks, regardless of the domain they are deployed are governed by the domain template. So test environments will have identical template policies to production and meet all security, governance and regularity requirements.

While development teams can create unique test environments under the **Test** layer 3 domain with the same subsequent policies, without the need to audit each and every one. The application security rules that developers use can then be agreed between security and development teams.

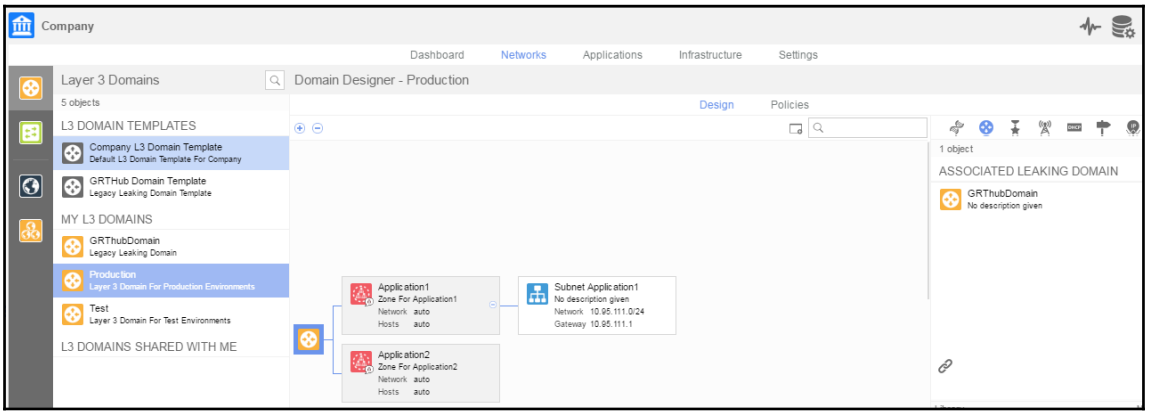
The other *day-one* set-up activity will likely be setting up access to legacy network, that teams will be migrating applications from as for a time, so they will still have dependent applications residing in that network.

The Nuage VSG and associated leaking domain can be used to do this, which leaks routes into the overlay network and into specific layer 3 domains. The Nuage VSP Platform allows network teams to define the **GRThubDomain leaking domain** in software. In this example a leaking domain is set-ip host interfaces are connected into the **Front End**, **Business Logic** and **Back End** routers in the legacy network:



The Nuage VSP platform then allows the newly created **GRThubDomain** to be associated with the **Production** or **Test** layer 3 domains by associating a leaking domain against them.

In the below example the **GRThubDomain leaking domain** is associated with the **Production** layer 3 domain to allow legacy network routes to be accessible from zones and subnets residing under the **Production** layer 3 domain:



The network team will also be responsible for monitoring the network and making sure that it is scaled out appropriately as more compute is introduced, so leaf switches will be introduced and ordered as and when new racks are scaled out. While new spine switches are introduced to avoid saturation of links.

Self-Service Networking

It is important to focus on the network operations that developers typically require network tickets for as a start point. These are the common pain points for developers that prove to be blockers for productivity. Network operations can be effectively separated by looking at the common themes on network ticketing systems that have been raised by development teams.

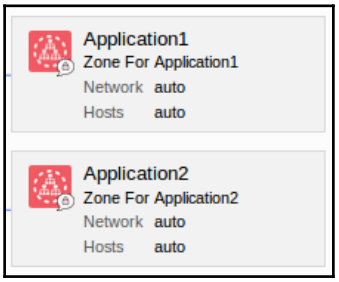
These are typically the mundane operations that network operators should make self-service:

- Opening firewall ports
- Creation of new development environments
- Connectivity to other applications

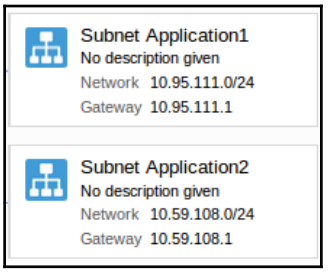
These operations should be set up as self-service operations in a software defined network.

In terms of the Nuage VSP object model, network operators should allow developers the ability to control the following object model entities:

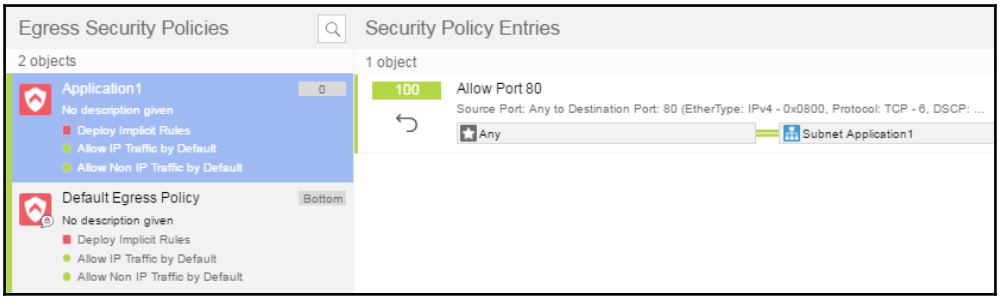
Zones:



Layer 3 Subnet:



Application Specific Egress Policy:



Application Specific Ingress Policy:

This will allow the network operations team to provide development teams with the organisation, layer 3 domains and the layer 3 domain template.

Underneath either the **Test** or **Production** layer 3 domains teams can have the flexibility to create new zones unique to each micro-service application, then any associated subnets and virtual machines that they need to provision.

The subnets will be micro subnets so something akin to a /26, /27 or /28 may be acceptable. The network team will provide the subnet schema and a booking system where teams if they are on-boarding an application or creating a new application can reserve the address space in an IPAM solution to prevent clashes with other teams.

As long as each delivery team follows those constructs the networking team does not need to be involved in the provisioning of new applications or onboarding, it will become self-service like AWS.

However, in order to properly facilitate development teams the network team should

ideally along with the operations team create the self-service automation that the development teams can use to carry out the following in Nuage VSP:

- Creation of Zones
- Deletion of Zones
- Creation of Subnets
- Deletion of Subnets
- Creation of Ingress Rules
- Deletion of Ingress Rules
- Creation of Egress Rules
- Deletion of Egress Rules
- Creation Of Network Macros (External Subnets)
- Deletion Of Network Macros (External Subnets)

No matter the SDN solution implemented, the self-service constructs required will be similar, in order to scale network operations a lot of the operations have to be automated and made self-service.

Ideally these self-service workflow applications could be added to Ansible playbooks or roles and included in the deployment pipelines to provision the networking along with the infrastructure.

Immutable Networking

To fully take advantage of the benefits of software defined networking, utilising immutable networking brings multiple benefits over static networking. Like infrastructure as code before it, networking as code and the utilisation of immutable networking means that every time an application is deployed its networking is freshly deployed from the source control management system.

If application connectivity is wrong in test environments then the application connectivity will be wrong in production environments, so wrong connectivity changes should never reach production.

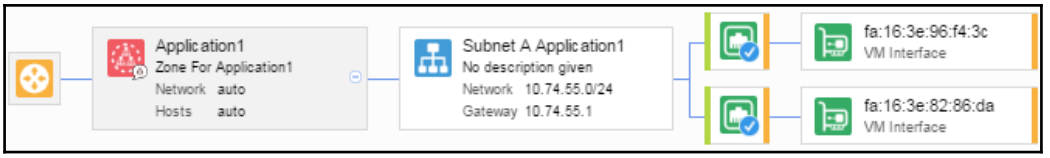
A/B Immutable Networking

Networking should be integrated and become part of the application release cycle, with networks being built from scratch every single release and loaded from the source control management system. Networks can be deployed using **immutable A/B Networking**.

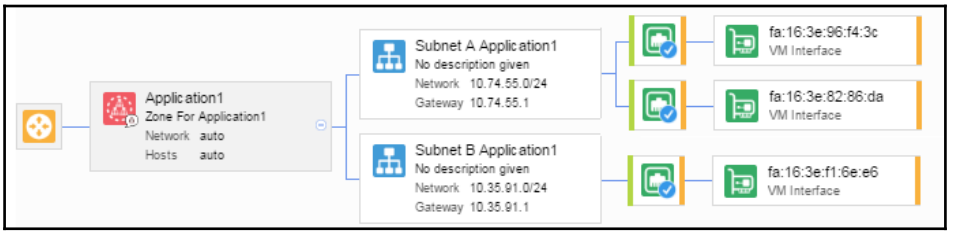
Utilising Nuage VSP integrated with OpenStack as an example:

- A network will reside under a layer 3 domain.
- Each zone will be unique to a particular application.
- Underneath the zone a subnet will be created in both Nuage and OpenStack.
- Virtual machines for each release will be created in OpenStack and associated with the Nuage Subnet

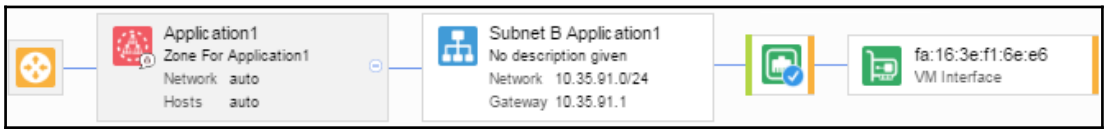
The first release of **Application1version 1.1** is deployed to the **Test** layer 3 domain, deploying two virtual machines on **Subnet A Application1,** sitting under the**Application1**zone:



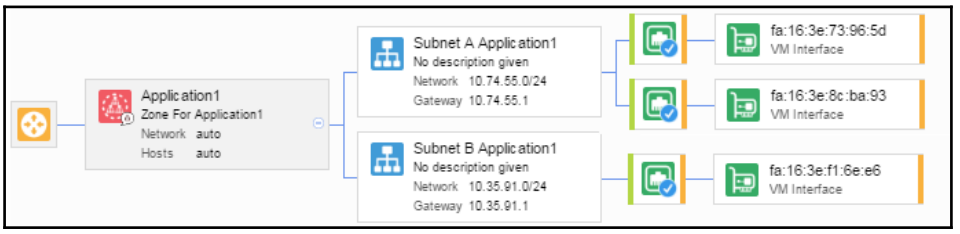
The second release of the application **version 1.2** is deployed to the **Test** layer 3 domain, scaling down the release and deploying one virtual machine on **Subnet B Application1**, sitting under the **Application1** zone:



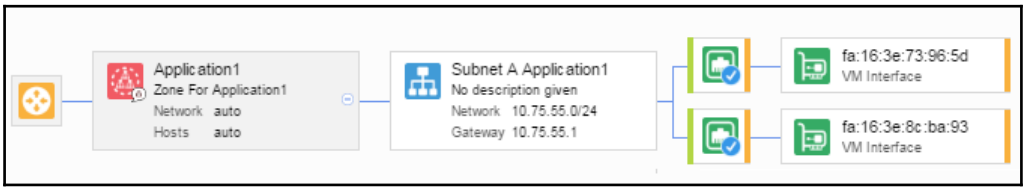
Once the **release 1.2** has been put into service on the load balancer, doing a rolling deployment, the new virtual machine on **Subnet B Application1** will be in service, **Subnet A Application1** can then be destroyed along with its virtual machines as part of the deployment clean-up phase:



The next release of Application 1, **release 1.3** will then be deployed into **Subnet A Application1**, and scaled up again to two virtual machines



Once the **release 1.2** has been put into service on the load balancer, doing a rolling deployment, the new virtual machines on **Subnet A Application1** will be in service, **Subnet B Application1** can then be destroyed along with its associated virtual machine as part of the deployment clean-up phase:



Releases will alternate between **Subnet A Application1** and **Subnet B Application1** for every release, building the network from source control each time and cleaning up the previous release each time.

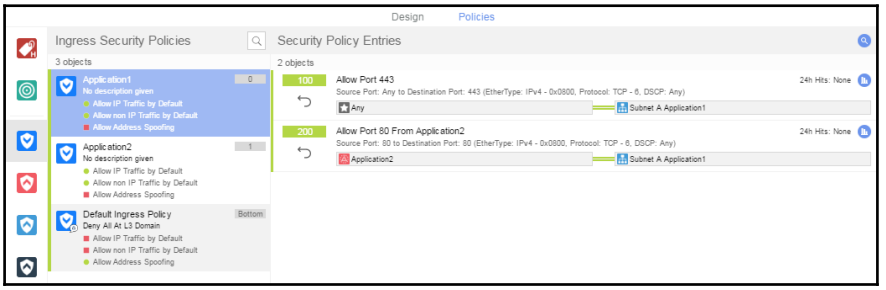
Clean-up of Redundant Firewall Rules

One of the major tech debt issues with firewalls is that over time they accumulate lots of out of date ACL rules as applications are retired or network connectivity changes. It is often a risk to do clean-up as network engineers are scared that they will potentially cause an outage. As a result manual clean-up of firewall rules is required by the network team.

When utilising A/B immutable network deployments, egress and ingress policies are

associated with subnets, meaning in Nuage VSP when a subnet is deleted all ACL policies associated with that subnet will be automatically cleaned up too as part of the release process.

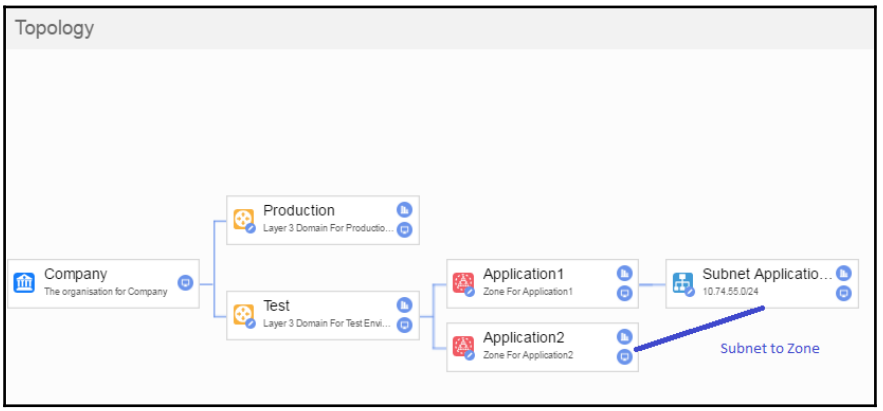
In the below example, **Subnet A Application1** has the following connectivity, so when the subnet is deleted as part of the release process all these subnet specific ACL rules will be cleaned up:



It is important to note that as ACL rules exist subnet to zone for application dependencies, if an A subnet deployment is in service, then the B deployment will be brought up in parallel with its associated ACL ingress and egress rules to replace the A deployment.

All applications dependant on that application will have an ACL rule pointing at the zone, this means they will not lose connectivity to the application as their rules will be current subnet either A or B to the dependant applications Zone.

To illustrate this in the below example currently deployed Subnet **Application1** has a subnet to zone ACL rule to connect to **Application2**. So despite **Application2** egress and ingress policies alternating between A and B deployments each time it is released:



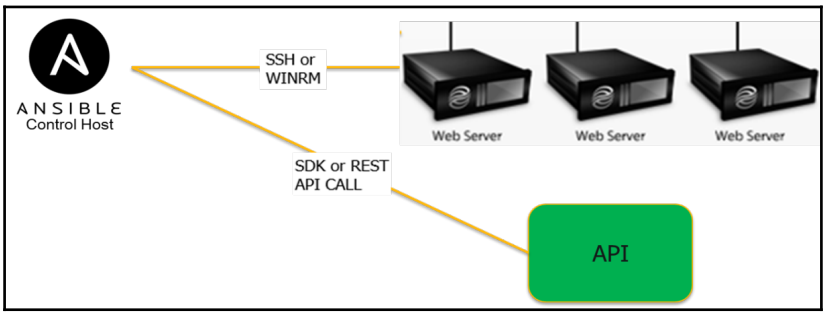
The required ACL rules are always available for **Application1** as a dependency as it subscribes to connectivity at the zone level as opposed to the subnet level:

Application Decommissioning

The use of immutable subnets makes the decommissioning of applications easy, when they are no longer required. As the clean-up logic already exists for subnets and associated ACL rules, that logic can be re-used to do a full clean-up of the application has to be retired. So a clean-up pipeline can easily be provided by the operations and networking team for development teams to clean-up applications that are no longer required. There allocated subnet ranges can then be released by the IPAM solution so they are available to new micro-service applications.

Using Ansible to Orchestrate SDN Controllers

Ansible as discussed in Chapter 5, can be used to issue configure servers as well as issuing commands directly to an SDK or REST API:



This is very useful when orchestrating SDN Controllers which provide Restful API end-points and an array of SDKs to control software defined object models that allow network operators to automate all network operations.

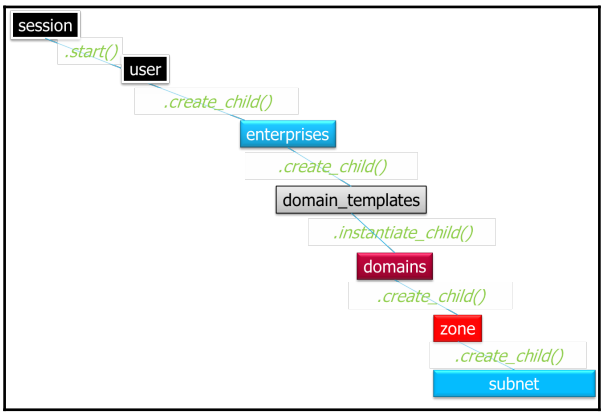
In terms of the Nuage VSP platform the VSD which builds the overlay network, is all REST API calls behind the scenes, so all operations can be orchestrated using the Nuage Java or Python SDK which wrap REST API calls. The Nuage VSPK would simply need to be installed on the Ansible Control Host and then it can be used to orchestrate Nuage.

As Ansible is written in Python, modules can be easily created to orchestrate each object model in Nuage entity tree. Using the Nuage VSPK modules could alternately be written in any programming language that is available such as Java, but Ansibles boilerplate for Python is probably the simplest way of creating modules.

The Nuage VSPK object model has parent and child relationships between entities, so lookups need to be done on parent objects to return the child entities using the unique identifier associated with the entity.

The example below highlights the list of operations required to build the Nuage VSPK object tree.

1. A new Nuage session is started
2. A user is used to create a child enterprise
3. A domain template is created as a child of the enterprise
4. A domain is an instantiated child of the domain template
5. A child zones is created against the domain
6. A child subnet is created against the zone



Using SDN for Disaster Recovery

One of the main benefits of using Ansible for orchestration is it can be used to create a set of day one playbooks to build out the initial network prior to it being used for self-service by developers. So the initial set-up of the Nuage organisation, **layer 3 domain template** and layer 3 domains can be created among any other necessary operations as a day one playbook or role.

The Nuage Python VSPK can be utilised to easily create the organisation called **Company**, layer 3 domain template called **L3 Domain Template** and two layer 3 domains called **Test** and **Prod** as per the Nuage VSPK object model as shown below:

```
#Open a session with VSD
session = vsdk.NUVSDSession(username=csproot,password=vsd_pass,enterprise=csp,api_url="https://nuage:8443",version="3.2")

#Start the session and get user credentials
session.start()
user=session.user

#Create an organisation
Organization = vsdk.NUEnterprise(name="Company",description="Company Description")
user.create_child(Organization)

#Create a Template
domain_template = vsdk.NUDomainTemplate(name="L3 Domain Template")

#Create Test domain
Organization.create_child(domain_template)
domain_test = vsdk.NUDomain(name="Test")
Organization.instantiate_child(domain_test,domain_template,commit=True)

#Create Production Domain
Organization.create_child(domain_template)
domain_prod = vsdk.NUDomain(name="Production")
Organization.instantiate_child(domain_prod,domain_template,commit=True)
```

Each of these Python commands can easily wrapped in Ansible to create a set of modules to

create a day one playbook utilising delegate_to localhost which will execute each module on the Ansible Control Host and then connect to the Nuage VSPK.

Each module by default should be written so it is idempotent and detects if the entity exists, before issuing a create command, if the entity already exists then it shouldn't issue a create command if the overlay network is already in the desired state.

The day one playbook can be used to build the whole network from scratch in the event of a disaster if the whole network needs to be restored. The day one playbook should be stored in source control. While each deployment pipeline will build the applications zones and subnets and virtual machines under the initially defined structure. A leaking domain governing legacy network connectivity and leaking domain association can also be added to the day one playbook if required.

Storing A/B Subnets and ACL Rules in YAML files

Ansible can also be utilised to store self-service subnet and ACL rule information in var files that will be called from a set of self-service playbooks as part of each development team's delivery pipelines. Each application environment can be stored in a set of var files defining each of the A/B subnets.

A playbook to create A or B subnets would be used to run delegate_to local host to carry out the creation actions against the Nuage VSD API.

The playbook would be set-up to:

1. Create the Zone if not already created
2. Create the Subnet in Nuage mapped to OpenStack using subnet YAML file
3. Apply ACL policies for ingress and egress rules to the policies applying them directly to the subnet

As with the day one playbook, unique modules can be written for each of the VSPK commands; in this example the Python VSPK creates a zone called **Application1** and a subnet called **Subnet A Application1**

```
#Create a Zone in the domain
zone = vsdk.NUZone(name="Application1")
domain.create_child(zone)

#Create a Subnet in the zone
subnetA = vsdk.NUSubnet(name="Subnet A Application1",address="10.74.55.0",netmask="255.255.255.0",gateway="10.74.55.1")
zone.create_child(subnetA)
```

So these commands can also be wrapped in Ansible modules should be completely

idempotent and the state is determined by the var files that are stored in source control.

The logic in the playbook would load the var files by pulling them from source control at deployment time. The playbook would then use the jinja2 filter conditions to detect if either the A or B subnet or neither was present using the when conditions.

If neither subnet was present subnet A would be created, or if subnet A was present then subnet B would be created.

The playbook could read this information from the environment specific var file that is specified below, as it is idempotent it will run over the zone, creating it if it doesn't already exist and using the playbook jinja2 when conditions to either create subnet A or B:

```
---
layer3_domain: Test
zone: Application1
subnets:

      - name: Subnet A Application1
        address: 10.74.55.0/24
        gateway: 10.74.55.1

      - name: Subnet B Application1
        address: 10.35.91.0/24
        gateway: 10.35.91.1
```

A unique set of A and B subnets would be checked into source control as a pre-requisite for every required environment, with one or more environments per layer 3 domain.

ACL rules should ideally be consistent across all environments encapsulated in a layer 3 domain, so an explicit set of ACL rules would be created and assigned to the applications unique policy for ingress and egress rules that would span all environments.

Each environment could have its own unique policy for egress and ingress per layer 3 subnet. The Ansible playbook could then append a unique identifier for environment to the policy name if multiple environments existed under the **Test** layer 3 domain to server integration, UAT or other test environments.

The unique ACL rules for an application can be filled in by development teams as part of the on-boarding to the new platform based on the minimum connectivity required to make the application function, with a deny all applied at layer 3 domain template.

The ACL rules should always be subnet to zone for inter-dependencies and each ACL rule will be created with the subnet as the source, so that when subnets are destroyed the ACL rules will automatically be cleaned up.

An example of how the self-service ACL rules file would look is displayed below which would create 2 ingress and 1 egress rule against the Application1 policy:

```
---
acl_rules:
  ingress:
    - name: ""
      protocol: "TCP"
      src_type: "ANY"
      src_port: "*"
      dst_port: 443
    - name: ""
      protocol: "TCP"
      src_type: "ANY"
      src_port: "*"
      dst_port: 80

  egress:
    - name: "native-dbs-1521"
      protocol: "TCP"
      dst_type: "Zone"
      dst: "Application2"
      dst_port: 80
```

The self-service playbook could be provided to development teams so that they always have a standard way to create zones and subnets. The YAML structure of the var files will also provide templates of what the desired state of the network should be. This means that pointing the automated pipelines at another Nuage endpoint would mean the whole network could be built out programmatically from source control.

Summary

In this chapter we have looked at different networking operations that SDN controllers can help automate and sought to debunk some of the common misconceptions associated with software defined networking.

We then looked at ways which companies can benefit from using software defined networking and looked at ways in which SDN solutions can help solve some of the challenges associated with network operations.

We then looked at how network operations needs to adapt and embrace automation so development teams can self-serve a subset of different networking tasks and ways in which networking can be divided and responsibilities shared. We then focused on the benefits of immutable A/B networking and how it can help simplify the network and build consistent programmatically controlled networks while keeping firewall rules clean.

In the next chapter we will look at continuous integration and how network operations can be take some of the best practices from development teams and apply them to networking operations, so that networking is versioned properly and can be used to roll forward and roll-back changes.

7

Using Continuous Integration Builds For Network Configuration

This chapter will focus on continuous integration, what the process entails, and why it is applicable to network operations. We will look at why continuous integration processes are vitally important when automating network operations.

This chapter will discuss the benefits of configuration management tooling and look at practical configuration management processes that can be used to set-up continuous integration processes and tooling that is available to support continuous integration processes.

In this chapter the following topics will be covered:

- Continuous Integration Overview
- Continuous Integration Tooling Available
- Network Continuous Integration

Continuous Integration Overview

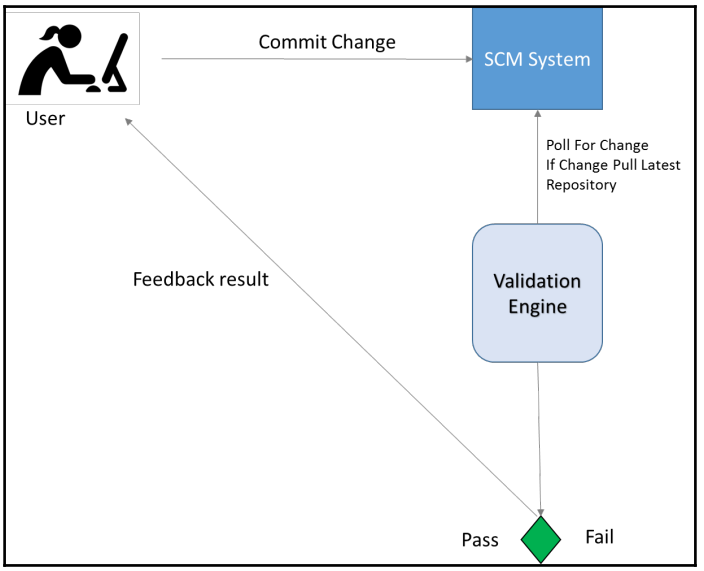
Continuous integration is a process used to improve the quality of development changes. A continuous integration process, when applied to developers, takes new code changes and integrates it with the rest of the code base. This is done early in the development lifecycle, creating an instant feedback loop and associated pass or failure against the change.

Prior to continuous integration developers would sometimes only find out that code changes did not work when a release needed to be packaged. At this point all developer changes were combined by a release management or operations team. By the time the release was ready to be packaged a developer would have moved on to new tasks and not have been currently working on that piece of work anymore meaning fixing the issue incurred more time delaying the release schedule.

A good continuous integration process should be triggered every time a developer commits a change, meaning that they have a prompt feedback cycle to tell them if their change is good, rather than finding out weeks or months later that their commit had an issue which will slow down the release process.

Continuous integration works on the premise of fixing as far left as possible meaning at development time, with the furthest right being production. What this phrase really means is that if an issue is found earlier in the development cycle then it will cost less to fix and have less of an impact to the business as it will ideally never reach production.

A continuous integration process follows the following steps, commit change to **source control management (SCM)**, the repository change is validated and a pass or failure is issued back to the user:



The output of the continuous integration process should be what is shipped to test environments and production servers. It is important to make sure it is the same binary

artifacts that have went through the CI and relative testing are the same ones that will eventually be deployed onto the production servers.

Processes such as continuous integration are used to create feedback loops that show issues as soon as they occur which saves cost. This means the change is fresh in the implementers mind and they will be able to fix it or revert the change quickly, with developers currently iterating the code collaboratively and fixing issues as soon as they occur.

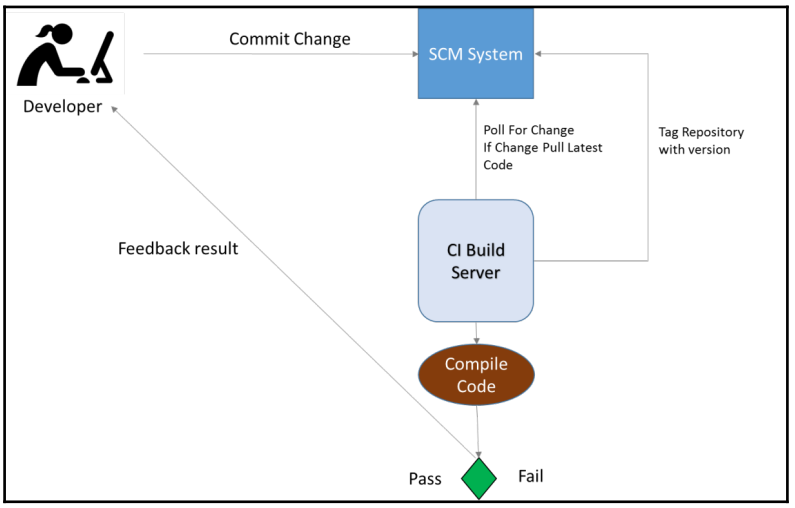
Although all IT staff may not follow identical deployment strategies, feedback loops and validation should not unique to just developers. Sure a compilation process may not be required when making network changes, but other validations can be done against a network device or a change on a SDN controller or load balancer to validate the changes are correct.

Developer Continuous Integration

A continuous integration process in its purest form takes a developer code change, integrates it with other developers latest changes and makes sure it compiles correctly. The continuous integration process can then optionally runs a set of unit or integration tests on the code base, packages the compiled binaries and then upload the build package to an artifact repository, tagging the code repository and package with a unique version number.

So a simple continuous integration process can be summarised as the following feedback loop:

1. Developer commits code change to SCM system and integrated with the code base
2. The code base is pulled down to a CI Build Server
3. The code is compiled checking the new commit is valid and non-breaking and repository is tagged with the version number
4. Return Pass or Fail exit condition and feedback to users
5. Repeat steps 1-5 for next code change



Steps 1 and 2 of the process are processes taken care of by source control management systems. Some of the popular source control management systems over the past ten years have been Subversion, ClearCase, Team Foundation Server, Perforce and CM Synergy. While distributed source control management systems have moved from centralised to distributed source control management systems such as GIT and Mercurial and in recent years.

Step 3, 4 and 5 are wrapped in a continuous integration build servers, which act as the scheduler for validation using tools such as Cruise Control, Hudson or more recently Jenkins, Travis and Thoughtworks Go.

Step 4 can is be carried out using compilation tools such as Maven, Ant, MsBuild, Rake or even a simple make file which is the main validation step in the process.

The process is carried out polling for every new commit and repeating the process providing a continuous feedback loop.

Additional steps such as unit or integration tests can be subsequently bolted on to the process after the compilation is successful for increased validation of the change. Just because code compiles doesn't mean it is always functional. When all compilation and tests are packaged a 6th step may be introduced to package the software and deploy it to an artifact repository.

All good continuous integration processes should work on the premise of compile, test and package. So a code release should be packaged once and the same package should be distributed to all servers at deployment time.

Database Continuous Integration

After continuous integration was set up to help improve the quality of code releases, developers that controlled database changes generally thought about doing similar processes for database changes. As database changes are always a big part of any enterprise release process having broken database releases can prevent software release being deployed and released to customers.

As a result, database schema changes or database programmatic stored procedures would equally benefit from being integrated earlier on in the continuous integration process and tested in a similar way using quick validation and feedback loops.

In a way developers have an easy ride when it comes to continuous integration as most code is compiled and either works or it doesn't returning a binary pass or fail in terms of compilation. Scripting languages are of course the exception to this rule, but these can be supplemented using unit tests to provide the code validation on various code operations and both code are improved by good test coverage.

When doing database schema changes a number of test criteria needs to be met prior to pushing the code to production. Good Database developers will provide roll forward and roll-back scripts when making SQL changes which will be applied to production databases and normally test these on their development machines prior to checking them into a source control management system.

Database developers typically do this using a roll-forward and roll-back release script and store them in source control management systems. The roll-back is only performed in the case of an emergency when it is being applied to production if the roll-forward for any reason fails.

So a typical database release process will have the two following steps:

1. Apply SQL Table or Column Creation, Update or Deletion or stored procedure using release script
2. If apply fails toll-back SQL Table or Column Creation, Update or Deletion or stored procedure using rollback release script

So prior to any production release, a database developers roll-forward and roll-back scripts should be tested. As multiple database developers are part of the same release these database release scripts should be applied in the same sequenced order as they would be applied to production as one developers change could break another developers changes.

Before setting up a database continuous integration a few pre-requisites are required:

- A database schema matching production with a relative data set and all the same characteristics such as indexing so we are testing against a like live version.
- The continuous integration process should also utilise the same deployment runner script that is used to sequence the database release scripts and provide rollback in case of failure.

Testing roll-back scripts is as integral to testing roll-forward scripts so the database **Continuous Integration (CI)** process will need valid tests to encompass rollback.

A common database deployment workflow applied by a database developer on their local workstation would look like this:

1. Apply roll-forward database script using deployment runner script to CI test database
2. Apply roll-back database script using deployment runner script to CI test database
3. Apply roll-forward database script using deployment runner script to CI test database
4. Apply roll-back database script using deployment runner script to CI test database

If the above set of steps are successful then the roll-forward and roll-back database scripts are sound in terms of syntax and won't fail when applied to the production database.

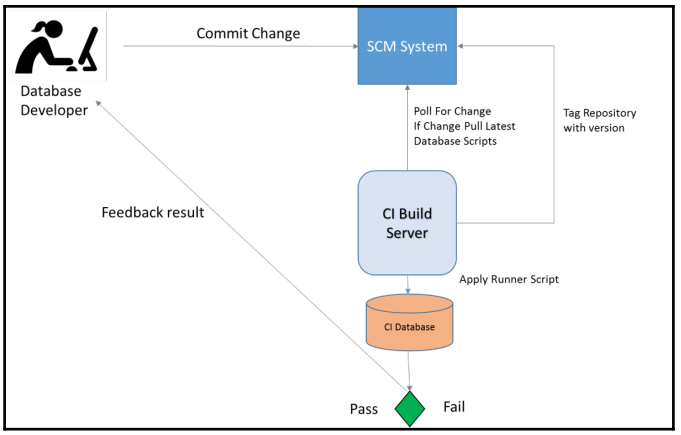
The above steps also check the validity of the sequencing using the deployment runner and check that the integrated database deployment scripts work together and do not conflict on roll-back either.

Already using continuous integration, we have ruled out multiple possible scenario that could cause a failure in production. However, the above continuous integration process alone is not enough, as with a code compilation, just because SQL not returning an error doesn't mean the database roll-forward and roll-back scripts are technically valid, so database changes still need to be supplemented with functional tests.

Continuous integration is about putting quality checks earlier in the delivery lifecycle and creating feedback loops. Continuous integration is not about proving that a release is 100% valid, it should instead be looked at as a way of proving that a checking process has been followed which prove that a release is not broken.

A simple continuous integration database process would provide the following feedback loop for database developers:

1. Developer commits roll-forward and roll-back change to SCM system and it is integrated with the code base
2. The code base is pulled down to a CI Build Server
3. Apply roll-forward database script using deployment runner script to CI test database
4. Apply roll-back database script using deployment runner script to CI test database
5. Apply roll-forward database script using deployment runner script to CI test database
6. Apply roll-back database script using deployment runner script to CI test database
7. Return Pass or Fail exit condition and feedback to users
8. Repeat steps 1-7 for next database change



Once the release is ready to go-live the database CI will have the final changes applied preparing it for the next release and iteration of database changes and next batch of database scripts that will be applied by the next release. Alternately the CI database schema can be refreshed from production.

A good concept is to always create a baseline of the database so that if a database developer unwittingly commits a bad roll-forward and roll-back database then the CI database can be easily restored to the desired state and not prove a bottle-neck for development.

Of course this is one way of dealing with validation of database changes and others are possible. Microsoft offers database projects for this very purpose but the validation engine is not important, having validation in the process early in the release life-cycle is the

important takeaway.

It is important to make sure that nothing goes to production unless it goes through the CI process, there is no point setting up a great process and then skipping it as it makes the CI database schema invalid and could have massive consequences.

Tooling Available For Continuous Integration

Many different flavours of configuration management tooling are available to help build continuous integration processes, so there is a rich variety of different options of options to choose from which can seem daunting at first.

Tools should be used picked to facilitate processes and will be selected by teams or users. As described in Chapter 3 it is important to first map out requirements that need to be solved and the desired process before selecting any tooling.

By the same token it is important to avoid tools sprawl, which is all too common in large companies and have only one best fit tool for every operation rather than multiple tools doing the same thing as there is an operational overhead for the business. So if configuration management tooling already exists in a company for continuous integration then it will more than likely it will be able to meet the needs.

When considering the tooling for carrying out continuous integration processes the following tools are required:

- Source Control Management (SCM) System
- Validation Engine

The source control management system is primarily used for storing code or configuration management configuration in a source control repository.

The validation engine is used to schedule the compilation of code or validate configuration. So continuous integration build servers are used for the scheduling and numerous compilation or test tools can be used to provide the validation.

Source Control Management Systems

Source control management systems provide the center of a continuous integration process, but no matter the **Source Control Management (SCM)** system that is chosen, at a base level

should have the followings essential features:

- Be accessible to all users that need to push changes
- Store the latest version of files
- Have a centralised URL that can be browsed by users to see available repositories
- Have a role based access permission model
- Support roll-back of versions and version trees on files
- Show which user committed a change along with the date and time of the change
- Support tagging of repositories, this can be used to check-out a tag to show all the files that contributed to a release
- Support multiple repository branches for parallel development
- Have the ability to merge files and deal with merge conflicts
- Have a command line
- Plugs into Continuous Integration Build servers

Most SCM systems will also support additional features such as:

- Have a programmable API or SDK
- Easily integrated with developer IDEs
- Integrate with Active directory or LDAP for role based access
- Support integration with change management tools, where a SCM commit can be associated with a change ticket
- Support integration with peer review tools

SCM systems can either be centralised or distributed, in recent years distributed source control management systems have increased in popularity.

Centralised SCM Systems

When source control management systems were originally created to facilitate development teams, a centralised architecture was used to build these systems. A centralised source control management system would be used to store code and developers would access the repository they were required to make code changes against and make edits against a live centarlised system.

For developers to remain productive the centralised SCM system would always need to be available and online.

- Developers would access the repository where they wish to make code changes

- Check-out the file they wished to edit
- Make changes
- Then check the file back into the code central branch.

The source control management system will have a locking mechanisms to avoid collisions where only one file can be edited by one user at a time. If two developers accessed the file at the same time the online source control management system would say it was locked by another developer and they would have to wait until the other developer made their change prior to being allowed to check out the code and make the subsequent change.

Developers when making changes would make a direct connection to repositories hosted in the centralised SCM system to make code updates. When a developer made a change, this in turn would write the changes in state to a centralised database updating the state of the overall repository.

The state change would then be synchronised to other developer's views automatically. One of the criticisms of centralised SCM systems was the fact that developers sometimes wanted to work offline, so some centralised source control management systems introduced the concept of snapshot views, which was an alternative to the permanently live and updated repository view and also introduced offline update features.

A snapshot view in a centralised SCM system was a snapped copy of the live repository at a given point in time. Best practice would dictate that before committing any development changes to the centralised server, the snapshot view should be updated, any merge conflicts dealt with locally before checking-in any changes that were made in the snapshot view.

Developers would integrate with the centralised SCM system using the command line interface or GUI that was integrated with a developers IDE for ease of use so they didn't need to jump between the command line and the IDE.

Examples of good centralised source control management systems are:

- ClearCase
- CM Synergy
- Team Concert
- Team Foundation Server
- Subversion
- Perforce

Distributed SCM Systems

Distributed SCM systems do not have a central master and instead replicate changes to multiple places. Users will create replicas of a repository and then can pull or push using their own local copy sitting on their local development machine. Each repository in a distributed system will have an owner or maintainer and users will submit changes in the form of pull requests. Developers will create a pull request which is like a merge request, but instead the repository maintainer can then approve if they accept the pull request or not. Once accepted the commit will be pulled into the branch.

One of the main benefits of a distributed SCM system is the ability to work on the repository offline. Changes can be committed to the local repository and then once back online pushed to the master branch when developers are ready.

Distributed SCM systems are more merge friendly and efficient, so work better with agile development which often means multiple small repositories for each micro-service rather than large centralised code basis for monolith applications.

Examples of distributed SCM systems are:

- GIT
- Mercurial
- Veracity

Branching Strategies

Branching strategies are used to meet the needs of modern software development, with multiple branches serving different use-cases and supporting multiple versions of the code.

SCM systems traditionally relied on a mainline branch, often referred to as the trunk or master branch. A mainline branching strategy meant that the mainline/trunk branch is always the clean and working version of the code and the files on this branch are representative of the code in production.

Development branches were then created for active development on the latest releases while release branches were used for maintenance releases if bugs were identified on the production system.

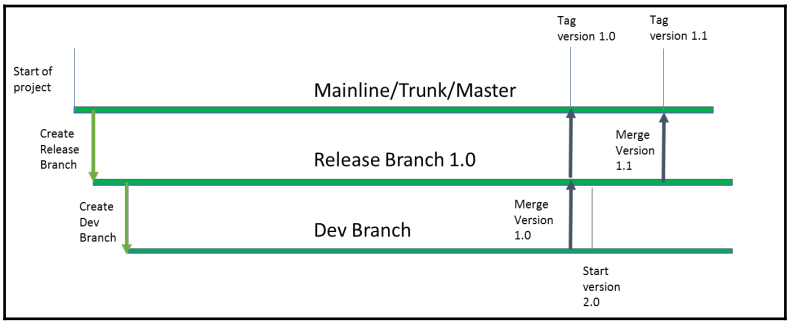
There are many different branching strategies that can be implemented, in the below example the mainline branching strategy is illustrated.

The **Mainline/Trunk/Master** branch is kept clean and all releases are done by merging changes to it and this branch is tagged every time a release is done. This allows a diff to be

done between tags to see what has changed.

The development branch is used for active development and creates version 1.0, then merges to the **Release Branch 1.0**, which in turn immediacy merges back to **Mainline/Trunk/Master**.

The development branch then starts active development on version 2.0, while the **Release Branch 1.0** is used for 1.x maintenance releases if a bug fix is required:



The mainline branching strategy typically meant a lot of merging and coordination and release managers were required to coordinate merges and releases of versions on release days.

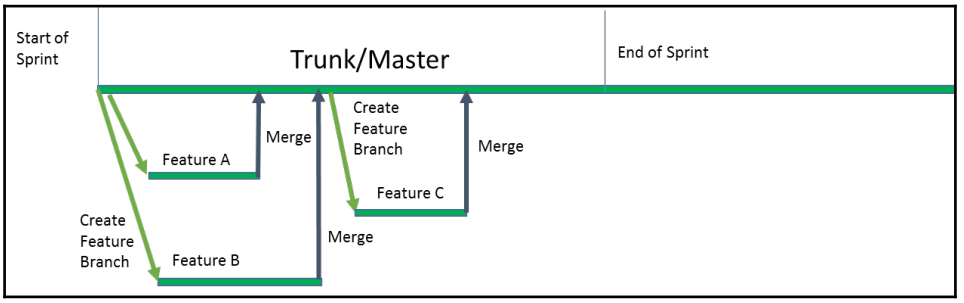
Centralised configuration management systems were set-up to favour a mainline approach to software development and this was good when supporting waterfall development, which was sufficient when teams were producing only one release per quarter so the laborious merge process was not happening daily.

However, with the transition to agile software development meant that implementing the mainline strategy became more difficult as teams release more frequently and move towards continuous deployment and delivery models.

An alternate branching strategy better suited to agile development is using feature branches, as development is done on a per sprint basis every two weeks. So the master or mainline branch is still used but very short-lived feature branches are created by developers during a sprint. Distributed SCM systems put the developer in charge of the merging as opposed to using a centralised release management team for these operations.

In the below example we can see an example of feature branching, where three different feature branches **Feature A**, **Feature B** and **Feature C** are created during a two week sprint. When developers have finished development their features merged back into the **Trunk/Master branch**.

Every time a commit is done from a merged feature branch or directly to **Trunk/Master** a continuous integration process will be started which will validate the changes, every successful check-in then becomes a potential release candidate. After it is packaged by the continuous integration process the release is ready for deployment as shown in the following figure:



Some purists will argue against the merits of feature branching at all but it is down to the individual teams to govern which approach works best for them and is subjective. Some will also argue that it adds an additional level of control until adequate testing is created on the Trunk/Master.

With all branches when a commit is done it should trigger a CI build and associated validation of whatever has been committed. This creates feedback loops at every stage. Anything that goes into any branch should be governed by a CI build to gate-keep good changes and highlight breaking changes as soon as they happen so they can be fixed.

Continuous Integration Build Servers

Various continuous integration build servers are available to help schedule validation steps or tests. One of the first continuous integration build servers was Cruise Control from Thoughtworks that has since evolved into Thoughtworks Go.

Cruise Control allowed users to configure an XML file, which set up different continuous integration build jobs. Each build job ran a set of command line options, normally a compilation process, against a code repository and returned a green build if it was successful and a red build if the build was broken. Cruise Control would highlight the errors in the form of build logs providing feedback to users via the Cruise Control dashboard or by email.

The market leading build server at the moment is Cloudbees Jenkins which is an open source project and a fork of the original Hudson project. Jenkins really took away the need

to configure XML files and moved all set-up operations into the graphical user interface or API. It comes with a plethora of plug-ins that can pretty much carry out any continuous integration operation possible. It also has recently delved into continuous delivery as of Jenkins 2.x.

The next evolution of CI systems has moved towards cloud based solutions with Travis being a popular choice. This allows users to check-in a Travis YAML file which creates the build configuration from source control and can be versioned along with the code. This is something Jenkins 2.0 is doing now using the jenkinsfile and that the Jenkins job builder project had been doing for the OpenStack project.

There are many different options when looking for continuous integration build servers consider the following, no matter the continuous integration build system that is chosen, at a base level it should have the followings essential features:

- Dashboard for feedback
- Notion of Green and Red builds
- Scheduling capability for generic command lines
- Pass or Fail builds based on exit conditions 0 being a pass
- Plug-in to well-known compilation tools
- Ability to poll source control management systems
- Ability to integrate with unit testing framework solutions such as Junit, Nunit and more
- Role based access control
- Ability to display change lists of the latest commits to a repository that have been built

Most SCM systems will also support additional features such as:

- Have a programmable API or SDK
- Provide email or messaging integration
- Integrate with Active directory or LDAP for role based access
- Support integration with change management tools, where a SCM commit can be associated with a change ticket
- Support integration with peer review tools

Network Continuous Integration

So why should network engineers be interested in continuous integration? A network team should be interested in continuous integration if they want to improve the following points which were focused on in Chapter 3:

- Velocity of change
- Mean time to resolve
- Improved uptime
- Increased number of deployments
- Cross skilling between teams
- Removal of the bus factor of one

The ability to easily trace changes that network changes and see which engineer made a change is something that continuous integration brings to the table. This information will be available by looking at the latest commit on a continuous integration build system. Roll-back will be as simple as deploying the last tagged release configuration as opposed to trawling through device logs to see what changes were applied to a network device if an error occurs.

Every network engineer can look at the job configuration on the continuous integration build system and see how it operates so every network engineer knows how the process works so it helps with cross skilling.

Having continual feedback loops will allows network teams to continuously improve processes, if a network process is sub-optimal then the network team can easily highlight the pain points in the process and fix them as the change process is evident to all engineers and done in a consistent manner.

When network teams use continuous integration processes it moves network teams out of fire-fighting mode and into tactical continuous improvement and optimisation mode. Continuous integration means that the quality of network changes will improve as every network change has associated validation steps that are no longer manual and error prone.

Instead these checks and validations are built in and carried out every time a network operator commits a network change to the source control management system. These changes can be built up over time to make network changes less error prone and give network engineers the same capabilities as developers and infrastructure teams.

Utilsing network continuous integration also takes the fear out of making production changes, as they are already validated and verified as part of the continuous integration process, so production changes can be viewed as just a business as usual activity. Not

something that needs to be planned weeks in advance or worried about. The view is if an activity is problematic then do it more often, continually iterate it, improve it and make people less afraid of doing it.

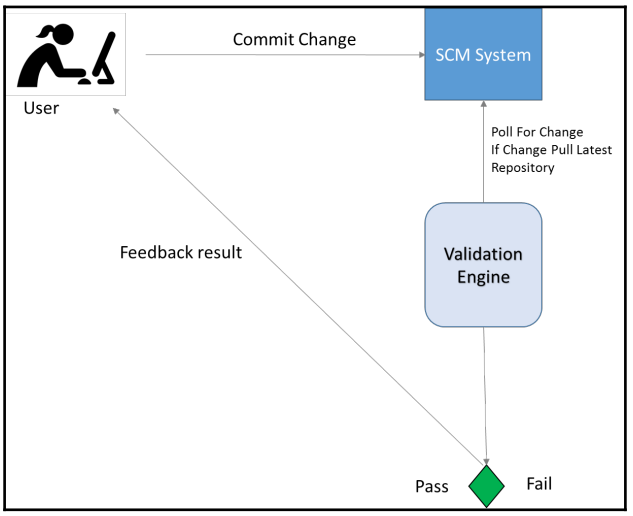
Having covered the topics such as different SCM branching strategies, continuous integration build servers and shown how continuous integration can be used for code and database changes. It should now be clear what continuous integration is and that it is not just about compilation of code. Instead continuous integration is instead about validating parallel changes, making sure they all work together and providing feedback loops to users.

The DevOps movement is about interacting with others and removing bottlenecks so continuous integration can be equally applied to networking. Automation of processes and collaboration between teams using similar concepts is very important so continuous integration really is the glue that holds infrastructure and networking as code together.

To a network engineer concepts such as continuous integration may seem alien at first, but instead of talking about deep dive compilation processes it should be focusing on process. If any network engineer was asked if they could have a quick and easy to use process that validated all their network changes before production, providing quick feedback loops, then the answer will be yes. Continuous integration can therefore be a useful tool which would mean less broken production changes.

In this book in Chapters 4, 5 and 6 we looked at way that network changes could be treated as code, using configuration management tooling such as Ansible to configure network devices, load balancers and SDN controllers.

So when considering the diagram below, the question regarding continuous integration of network changes is not asking if continuous integration possible for network changes. It should instead be questioning which validation engines can be used for network changes after a SCM commit has taken place to give a quick feedback loop of **Pass** or **Fail** to network operators:



Network Validation Engines

The challenge when creating continuous integration builds for network changes is what to use for the validation engine. Network changes when using Ansible rely heavily on YAML configuration files, so the first validation that can be done is checking if the YAML var files.

The var files are used to describe the desired state of the network so checking that these YAML files are valid in terms of syntax is one valid check. So to do this a tool such as *YAMLlint* can be used to check if the syntax of the files that are committed into source control management are valid.

Once the YAML var files are checked into source control the continuous integration build should create a tag to state a new release has happened, all source control management systems should have a tagging or base-lining feature. Tagging versions means that the current network release version can be diffed against the previous to see what file changes have occurred on the YAML var files if an issue is detected at any stage making all network changes transparent.

So what other validation is possible? When focusing on configuration of network devices, we are pushing configuration changes to a networking operating system such as Juniper Junos or Arista EOS. So being able to run the newly committed changes and make sure the syntax is programmatically correct against those operating systems as part of the continuous integration process is highly desirable. Most network device operating systems

as discussed in Chapter 4 are Linux based, so having a network operating system to issue commands to as part of the CI process doesn't seem too absurd.

The same can be said when checking the configuration used to orchestrate load balancers or SDN controllers, having a test environment attached to the CI process is also highly desirable in theory. By utilising a software version of the load balancer or emulated version of the SDN controller would be highly beneficial so network engineers can pre-flight their network changes to make sure the API calls and syntax is correct.

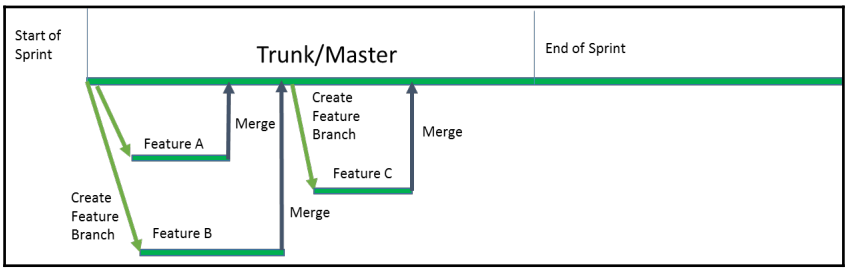
However, there are challenges simulating an SDN controller or creating or simulating a production environment depending on the vendor, there may have a huge overhead in terms of setting up a continuous integration environment due to cost. Network devices, load balancers and SDN vendors are evolving to support automation and *DevOps friendly* processes such as continuous integration. Therefore, networking vendors are starting to appreciate the validity of giving small test environments, this is where virtualised or containerised versions of load balancers or SDN controllers would be useful as an API endpoint to validate the desired state that has been set-up in YAML files.

Alternately the vendors could provide a vagrant box to test if the desired configuration specified in YAML var files that is checked into source control management systems is valid before it is propagated to the first test environment. Any enhancements that can be done to processes to make it fail as fast as possible and shift issues as far left as possible in the development lifecycle should be implemented where possible.

So with all of these validators, let's look at how these processes can be applied to network devices or alternately orchestration. The number of validators used may depend on the network vendors that are being used, so we will look at the start point for a continuous integration build for network devices regardless of vendor and then look at more advanced options that could be used if the vendor provides a software load balancer or SDN emulation.

Simple Continuous Integration Builds for Network Devices

As network changes are always required daily by large organisations that are implementing micro-service applications. To meet those demands then networking should be as self-service as possible. To keep up with demand, network teams will likely need to use a feature branch source control management strategy or allow self-service YAML files to be committed direct to master as shown below:



Each commit should be peer reviewed before it is merged. Ideally the self-service process should allow development teams to package network changes alongside their code changes and follow a self-service approach.

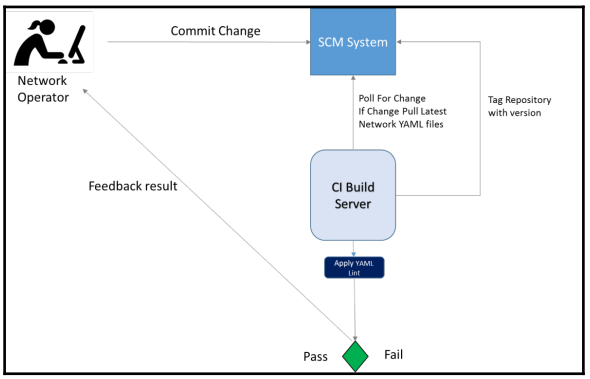
The first continuous integration build that should be set-up for network devices or orchestration should focus on version controlling the Ansible YAML files and running a simple YAML validation on the desired state.

Each continuous integration build that runs will also tag the repository. Tagging the source control management repository means that release versions can be compared or easily rolled back. It will also act as an audit log to show which user made changes and what exactly has changed in the environment. No changes should be made to a production system that have not went through the continuous integration process.

So a simple network continuous integration build will follow these simple validation steps:

1. YAML files are checked for syntax
2. Repository is tagged in source control management system if successful

Therefore, a simple network continuous integration build would follow these steps. The network operator would commit the YAML files to the SCM system to change the desired state of the network, the continuous integration build server would tag the build if the YAML Lint operation finds that all the YAML files in the repository have valid syntax and return a positive result:

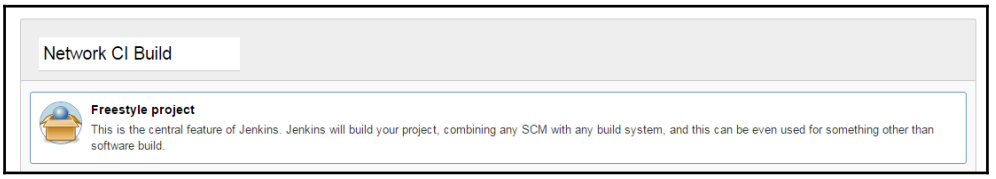


Configuring a Simple Jenkins Network CI Build

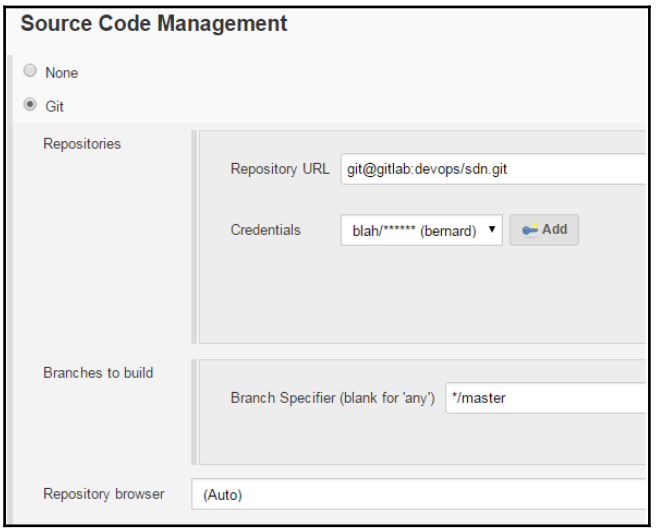
This simple continuous integration build for network devices can be set-up in the Jenkins CI build server. Rake and the yamllint gem should be configured on the Jenkins slave that the build will be executed.

Once this has been completed a new Jenkins CI build can be created in a matter of minutes.

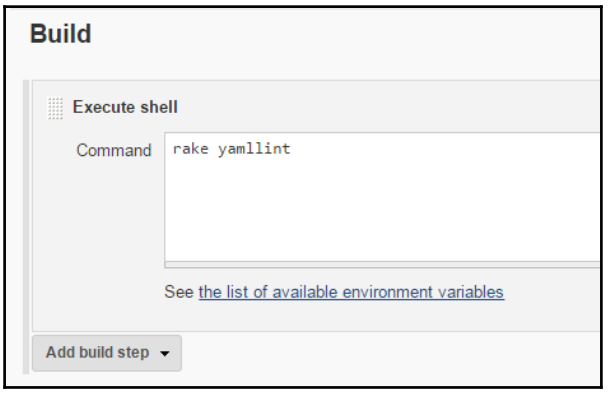
First select a new Jenkins freestyle job:



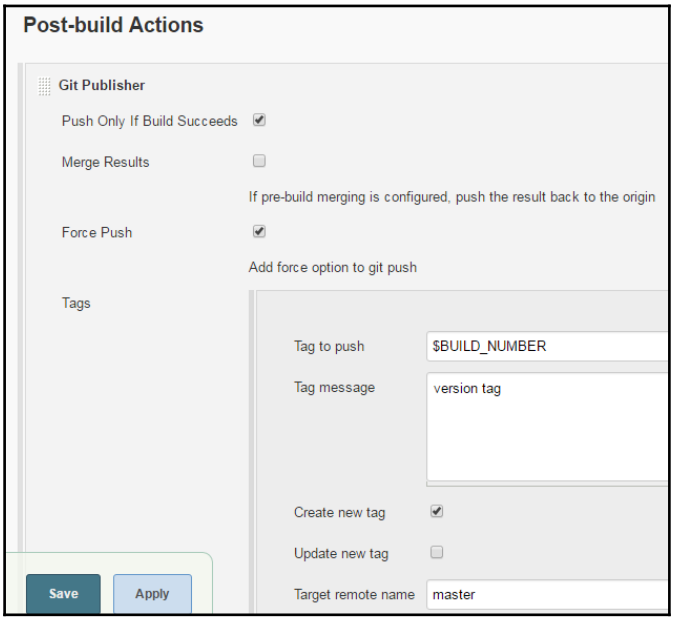
Then configure source control management system to use, in this instance GIT, specifying git@gitlab:devops/sdn.git as the repository and the ***/master** branch of the project along with the SSH key required to provide access to the repository:



Now for the validation step a shell command build step is selected, which will to run **rake yamllint** on the repository after configuring a rakefile in the git@gitlab:devops/sdn.git repository so the YAML files can be parsed:

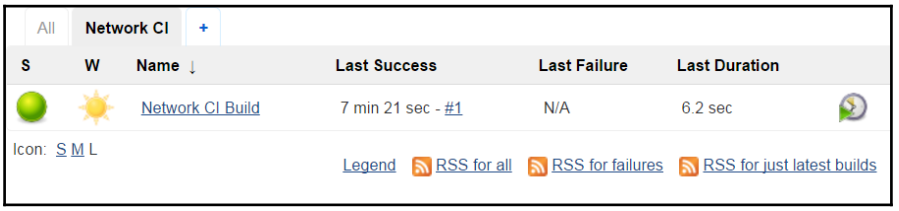


Finally configure the build job to tag the Jenkins build version against the **devops/sdn.git** gitlab repository and save the build:



This has configured a very simple Jenkins CI build process that will poll the GIT repository for new changes, run yamllint against the repository and then tag the GIT repository if the build is successful.

The build health will be shown in Jenkins, the green ball means the build is in a healthy state so the YAML files are currently in a good state, the duration of the check shows it took 6.2 seconds to execute the build as shown in the following screenshot:



Adding Validations to Network Continuous Integration Builds

After highlighting the need for more robust validation when pre-flight configuration of network devices as far left in the development lifecycle as possible to reduce the cost to fix. Having the ability to push mission critical configuration changes to a networking operating system such as Cisco NXOS, Juniper Junos or Arista EOS would be a good continuous integration validation.

So like databases verify SQL syntax is correct, being able to run the newly committed changes and make sure the networking commands or orchestration commands applied to network devices syntax is programmatically correct should be part of the continuous integration build.

Continuous integration can then help the quality of network changes as an incorrect change would never be pushed to a network device, load balancer or SDN controller. Of course the functionality of the configuration pushed may not be what is required but there should never be a situation where the configuration has a syntax error at deployment time.

As network devices, load balancer and SDN controller changes are mission critical this brings an added layer of validation checks to any network changes and checks in a quick and automated way providing quick feedback if a network change is not what is required.

Continuous Integration for Network Devices

Before setting up a network device continuous integration a few pre-requisites are required:

- A network operating system will be required with production configuration pushed to it and all live settings which can be hosted on a virtual appliance.
- The continuous integration build tool such as Jenkins will need to have an Ansible Control Host set-up on the agent so it can execute Ansible playbooks.
- All playbooks should be written with a block rescue so subsequent cleanup is built in if the execution of the playbook fails.

A typical network device release process will have the two following steps:

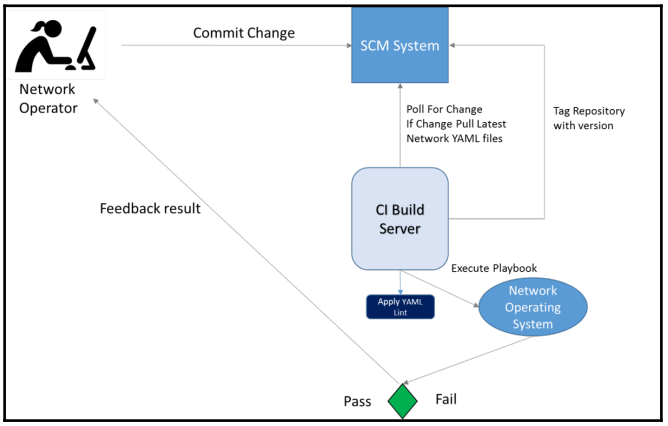
1. Apply network change self-service playbook
2. As the playbook is idempotent only changes will be shown if a change has occurred

The Ansible playbook should provide resilience for roll-forward and roll-back in terms of

state change. The above steps also check the validity of the sequencing using the Ansible playbook and check that the calls being made to the network device are valid.

A simple continuous integration network build process would provide the following feedback loop for network operators:

1. Network operator commits Ansible playbook or YAML var file change to SCM system and it is integrated with the code base
2. The code base is pulled down to a CI Build Server
3. YAML files are checked using yamllint
4. Ansible Playbook is applied to push network changes to the device
5. Return Pass or Fail exit condition and feedback to users
6. Repeat steps 1-5 for next network device change



Continuous Integration Builds for Network Orchestration

Before setting up a network orchestration for load balancers or SDN controllers a few pre-requisites are required:

- A software load balancer or an emulated SDN controller will be required with production configuration pushed to it and all live settings.
- The continuous integration build tool such as Jenkins will need to have an Ansible controller set-up on the agent so it can execute Ansible playbooks as well as the SDK that will allow the network orchestration modules to be executed.

- All playbooks should be written with a block rescue so subsequent cleanup is built in if the execution of the playbook fails.

A typical network device release process will have the two following steps:

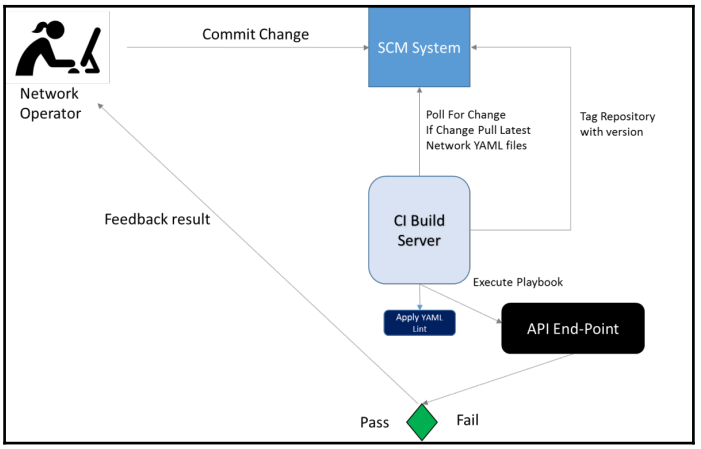
1. Apply network change self-service playbook
2. As the playbook is idempotent only changes will be shown if a change has occurred

The Ansible playbook, like with the network device changes should provide resilience for roll-forward and roll-back in terms of state change. Some test servers may be needed on a virtulisation platform to simulate the load balancing so health checks can be tested too.

A simple continuous integration network orchestration CI process would provide the following feedback loop for network operators:

1. Network operator commits Ansible playbook or YAML var file change to SCM system and it is integrated with the code base
2. The code base is pulled down to a CI Build Server
3. YAML files are checked using yamllint
4. Ansible Playbook is applied to orchestrate the API and create the necessary load balancer or SDN changes
5. Return Pass or Fail exit condition and feedback to users

Repeat steps 1-5 for next network orchestration change



Summary

In this chapter we have looked at what continuous integration is and how continuous integration processes can be applied to code and databases. The chapter then looked at ways that continuous integration can be applied to assist with network operations to provide feedback loops.

We also explored different source control management methodologies, the difference between centralised and distributed source control management systems and how branching strategies are used with waterfall and agile processes.

We then looked into the vast array of tools available for creating continuous integration processes focusing on some examples using Jenkins to set up a simple network continuous integration build.

In the next chapter we will look at various test tools and how they can be applied to continuous integration processes for added validation, this will allow unit tests to be created for network operations to make sure the desired state is actually implemented on devices.