Introducing Ansible, Terraform, and Kubernetes

Originating from Google, Kubernetes is an open re-write of Google’s internal application management system Borg. You should definitely take some time and read the paper on Borg. It contains lots of knowledge about managing systems at scale and explains some of the thinking behind Kubernetes.

Kubernetes is written in Golang and the source is available on GitHub. It typically needs a head node and a set of worker nodes. The head node runs an API server, a scheduler and a controller. The workers run an agent called *kubelet* and a proxy that enables service discovery across the cluster. The state of the cluster is stored in Etcd. We will leave a discussion on Kubernetes primitives for another time.

etcd is a distributed key value store that provides a reliable way to store data across a cluster of machines. It’s open-source and available on GitHub. etcd gracefully handles leader elections during network partitions and will tolerate machine failure, including the leader.

Your applications can read and write data into etcd. A simple use-case is to store database connection details or feature flags in etcd as key value pairs. These values can be watched, allowing your app to reconfigure itself when they change.

Advanced uses take advantage of the consistency guarantees to implement database leader elections or do distributed locking across a cluster of workers.

In this lab, we will use Ansible and Terraform to create a Kubernetes cluster on Amazon AWS. The head node will also run etcd and we will start as many workers as we want.

Terraform declarative approach works very well in describing and provisioning infrastructure resources, while it is very limited when you have to install and configure;

Terraform and Ansible overlap. You will use Terraform to provision infrastructure resources, then pass the baton to Ansible, to install and configure software components.

Terraform allow us to describe the target infrastructure; then it takes care to create, modify or destroy any required resource to match our blueprint. Regardless of its declarative nature, Terraform allows some programming patterns. In this project, resources are grouped in files; constants are externalised as variables and we will use of templating. We are not going to use Terraform Modules.

Before we get started, we need to install a few prerequisites just in case you do not have them on your machine already.

We will be using Ansible, which has a Python package that can be installed via the Python Package installer (i.e Pip).

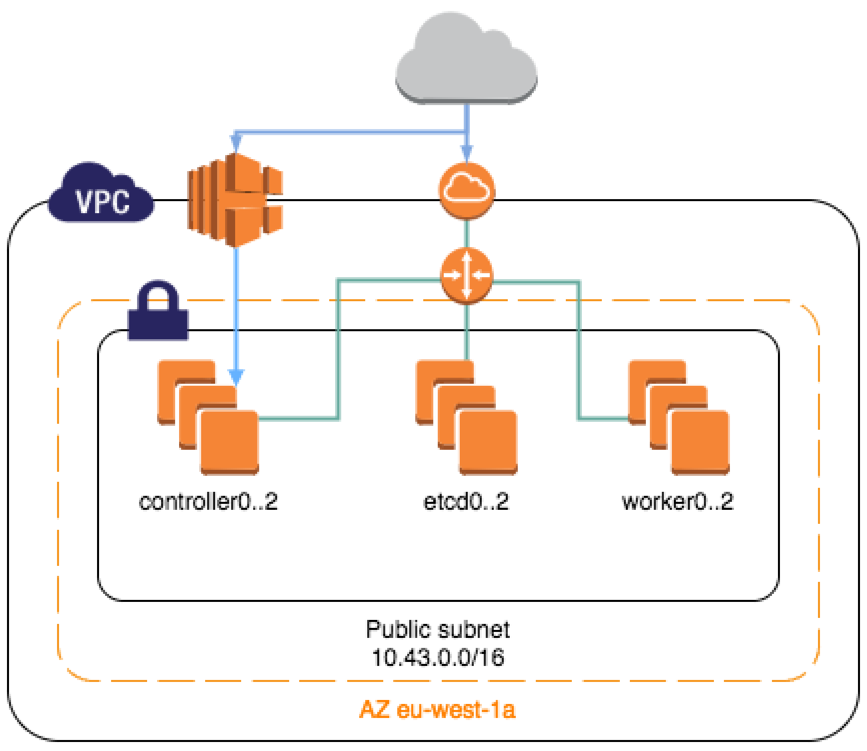
we'll all use the Ansible core module for AWS cloudformation. This can also be installed using Pip.

To use your Kubernetes cluster you will need a client that talks to the k8s API. This client is called *kubectl*. It is extremely powerfull and enjoyable to use.

Our target platform will be the following:

3 EC2 instances for Kubernetes Control Plane: *Controller Manager*, *Scheduler*, *API Server*.

- 3 EC2 instances for the HA *etcd* cluster.
- 3 EC2 instances as Kubernetes workers (aka *Minions* or *Nodes*, in Kubernetes jargon)
- Container networking using Kubenet plugin (relying on CNI)
- HTTPS communication between all components
- All Kubernetes and *etcd* components run as services directly in the VM (not in Containers).



Before proceeding, we have to understand how Ansible identifies and find hosts.

Dynamic Inventory

Ansible works on groups of hosts. Each host must have a unique handle and address to SSH into the box.

The most basic approach is using a static inventory, a hardwired file associating groups to hosts and specifying the IP address (or DNS name) of each host.

A more realistic approach uses a Dynamic Inventory, and a static file to define groups, based on instance tags. For AWS, Ansible provides an EC2 AWS Dynamic Inventory script out-of-the-box.

Step 1. Provision your AWS instances with Terraform.

Using the .tf files given in the lab, provision an AWS cluster as described above.

Step 2. Install and configure Kubernetes using Ansible playbooks.

Using the ansible playbooks given, install and configure the Kubernetes software system on your AWS cluster.

Step 3. Configure Kubernetes

There is still one important step: setting up the routing between Workers (aka Nodes or Minions) to allow Pods living on different machines to talk each other. As a final smoke test, we'll deploy a *nginx* service.

Before starting, we have to configure Kubernetes CLI on our machine to remotely interact with the cluster.

For running the following steps, we need to know Kubernetes API ELB public DNS name and Workers public IP addresses. Terraform outputs them at the end of provisioning. In this simplified project, we have to note them down, manually.

For running the following steps, we need to know Kubernetes API ELB public DNS name and Workers public IP addresses. Terraform outputs them at the end of provisioning. In this simplified project, we have to note them down, manually.

Setup Kubernetes CLI

This step is not part of the platform set up. We configure Kubernetes CLI locally to interact with the remote cluster.

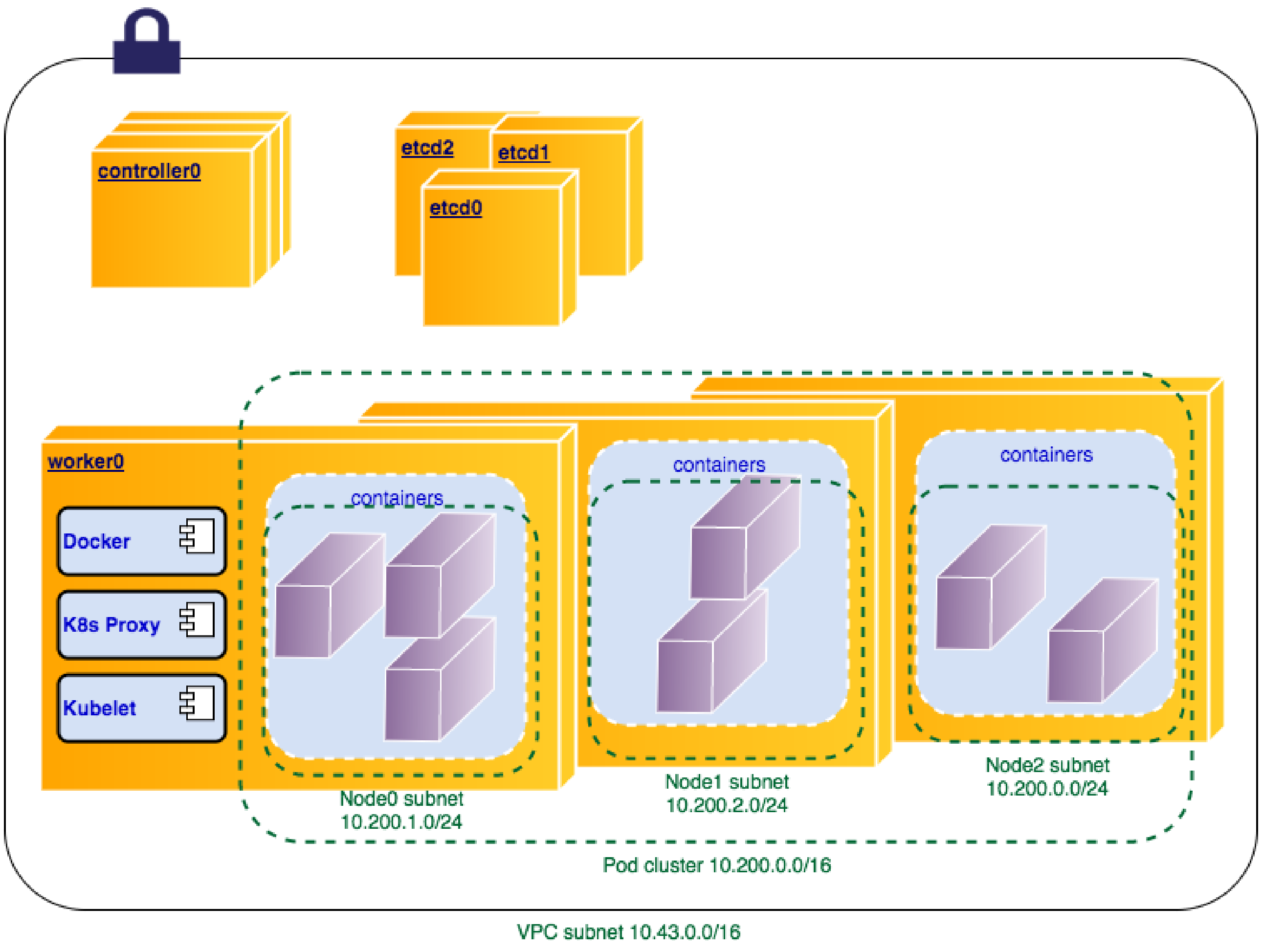
Setting up the client requires running few shell commands. The save the API endpoint URL and authentication details in the local kubeconfig file. They are all local shell commands, but we will use a playbook (`kubectl.yaml`) to run them.

The client uses the CA certificate, generated by Terraform in the first part. User and token must match those in the token file (`token.csv`), also used for Kubernetes API Server setup. The API load balancer DNS name must be passed to the *playbook* as a parameter.

Kubernetes CLI is now configured, and we may use `kubectl` to control the cluster.

Kubernetes uses subnets for networking between Pods. These subnets have nothing to do with the subnet we defined in AWS.

Our VPC subnet is 10.43.0.0/16, while the Pod subnets are part of 10.200.0.0/16 (10.200.1.0/24, 10.200.2.0/24 etc.). We have to setup routes between workers instances for these subnets.



As we are using the *Kubenet* network plugin, Pod subnets are dynamically assigned. Kube Controller decides Pod subnets within a Pod Cluster CIDR (defined by `--cluster-cidr` parameter on `kube-controller-manager`startup). Subnets are dynamically assigned and we cannot configure these routes at provisioning time, using Terraform. We have to wait until all Kubernetes components are up and running, discover Pod subnets querying Kubernetes API and then add the routes.

In Ansible, we might use the `ec2_vpc_route_table` module to modify AWS Route Tables, but this would interfere with route tables managed by Terraform. Due to its stateful nature, tampering with Terraform managed resources is not a good idea.

The solution (hack?) adopted here is adding new routes directly to the machines, after discovering Pod subnets, using `kubectl`. It is the job of `kubernetes-routing.yaml` *playbook*, the Ansible translation of the following steps:

Query Kubernetes API for Workers Pod subnets. Actual Pod subnets (the second column) may be different, but they are not necessarily assigned following Workers numbering.

```
$ kubectl get nodes --output=jsonpath='{range .items[*]}
{.status.addresses[?(@.type=="InternalIP")].address}
{.spec.podCIDR}{"\n"}{end}'
10.43.0.30 10.200.2.0/24

10.43.0.31 10.200.0.0/24

10.43.0.32 10.200.1.0/24
```

Then, on each Worker, add routes for Pod subnets to the owning Node

```
$ sudo route add -net 10.200.2.0 netmask 255.255.255.0 gw
10.43.0.30 metric 1

$ sudo route add -net 10.200.0.0 netmask 255.255.255.0 gw
10.43.0.31 metric 1

$ sudo route add -net 10.200.1.0 netmask 255.255.255.0 gw
10.43.0.32 metric 1
```

… and add an IP Tables rule to avoid internal traffic being routed through the Internet Gateway:

```
$ sudo iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j
MASQUERADE
```

The last step is a smoke test. We launch multiple *nginx* containers in the cluster, then create a Service exposed as NodePort (a random port, the same on every Worker node). The are three local shell commands. The `kubernetes-nginx.yaml`) is the Ansible version of them.

```
$ kubectl run nginx --image=nginx --port=80 --replicas=3

$ kubectl expose deployment nginx --type NodePort

$ kubectl get svc nginx --output=jsonpath='{range .spec.ports[0]}
{.nodePort}'

32700
```

The final step is manual (no playbook!). To test the service we fetch the default page from *nginx*.

All Workers nodes directly expose the Service. Get the exposed port from the last command you run, get Workers public IP addresses from Terraform output.

This should work for all the Workers:

```
$ curl http://<worker0-ip-address>:<port>

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>
```