# Deploy Kubernetes the boring way

## on bare-metal, proxmox and openstack

Yunqi Shao

2026-04-15

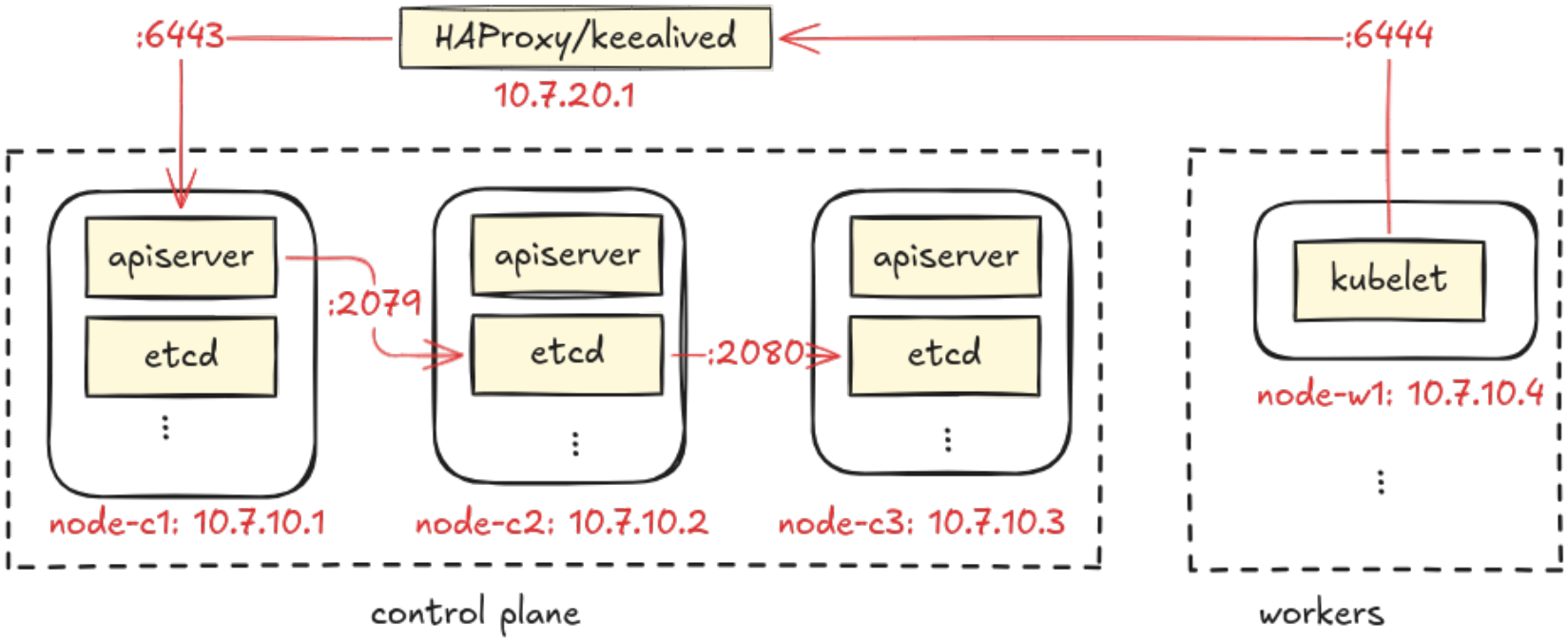
# Outline

# Intro and motivation

Kubernetes is:

- a scheduler for pod/container workloads
- a few specifications for declaring services

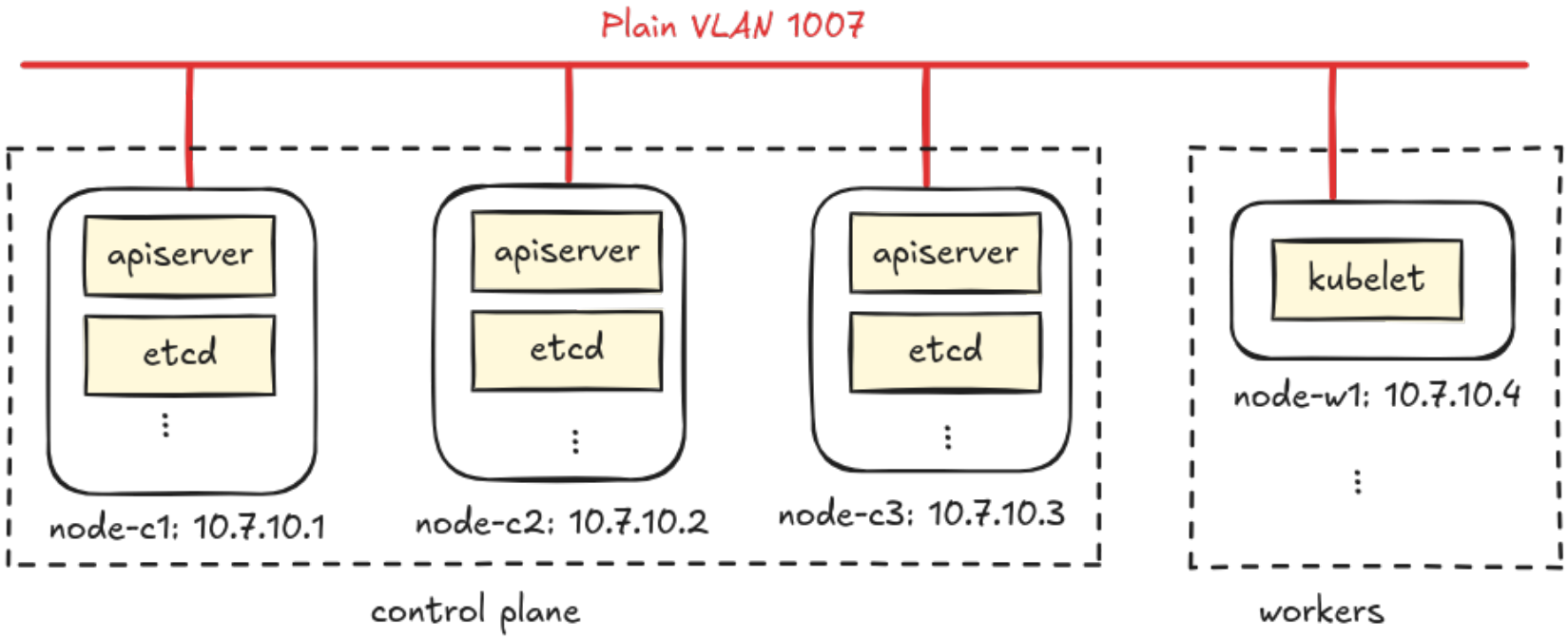
# Basic components of Kubernetes




The following services needs to be running "before Kubernetes":

- etcd: distributed KV database
- kube-apiserver: frontend
- kubelet: receives pod specs (from apiserver or files)
- kube-scheduler: assigns pods to nodes

# Network requirements




- Nodes needs to access each other's IP
- Important ports needs to be open

# kubeadm

```
# on first node
kubeadm init \
  --control-plane-endpoint=10.7.20.1:6444 \
  --skip-phases=addon/kube-proxy \
  --upload-certs

# on other nodes (kubeadm init provides join command)
kubeadm join 10.7.20.1:6444 \
  --token pl6ebz.mln3xh8gglwd75dt \
  --discovery-token-ca-cert-hash sha256:xxxxxxx
```

- bootstrap tool to "a minimum viable cluster up and running"
- initializes the cluster / joins new nodes
- generates certificates for internal service communication

# Running a cluster

1. Prepare OS and networking
2. Install loadbalancer for apiserver
3. Install kubernetes
4. `kubeadm join --apiserver ...`
5. `kubectl get nodes`

# Tools (after a year)

- dnsmasq: DHCP server (the same thing)
- Cobbler ⇒ matchbox+Terraform: PXE service
- Ansible: everything after node installation
  - Setup networking
  - Basic monitoring
  - Kubernetes components

# Terraform vs. Ansible

```
resource "openstack_compute_instance_v2" "worker" {
  for_each    = local.worker_nodes
  name        = each.key
  flavor_name = "k8s-compute"
  key_pair    = openstack_compute_keypair_v2.janne.id
  image_id    = openstack_images_image_v2.rocky10.id
}

resource "openstack_images_image_v2" "rocky10" {
  ...
}
```

- Terraform defines resources, Ansible defines steps
- Terraform plans and runs, Ansible has `--check`

# Terraform vs. Ansible

```
- name: Create a new instance
  openstack.cloud.server:
    state: present
    name: stratus-w3
    image: 4f905f38-e52a-43d2-b6ec-754a13ffb529

- name: Creats something else
  ...
```

- Terraform defines resources, Ansible defines steps
- Terraform plans and runs, Ansible has `--check`

# Terraform vs. Ansible

```
  # openstack_compute_instance_v2.bastion must be replaced
-/+ resource "openstack_compute_instance_v2" "bastion" {
      - config_drive        = true -> null # forces replacement
        name                = "ddls-bastion"
      + region              = (known after apply)
    }

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
```

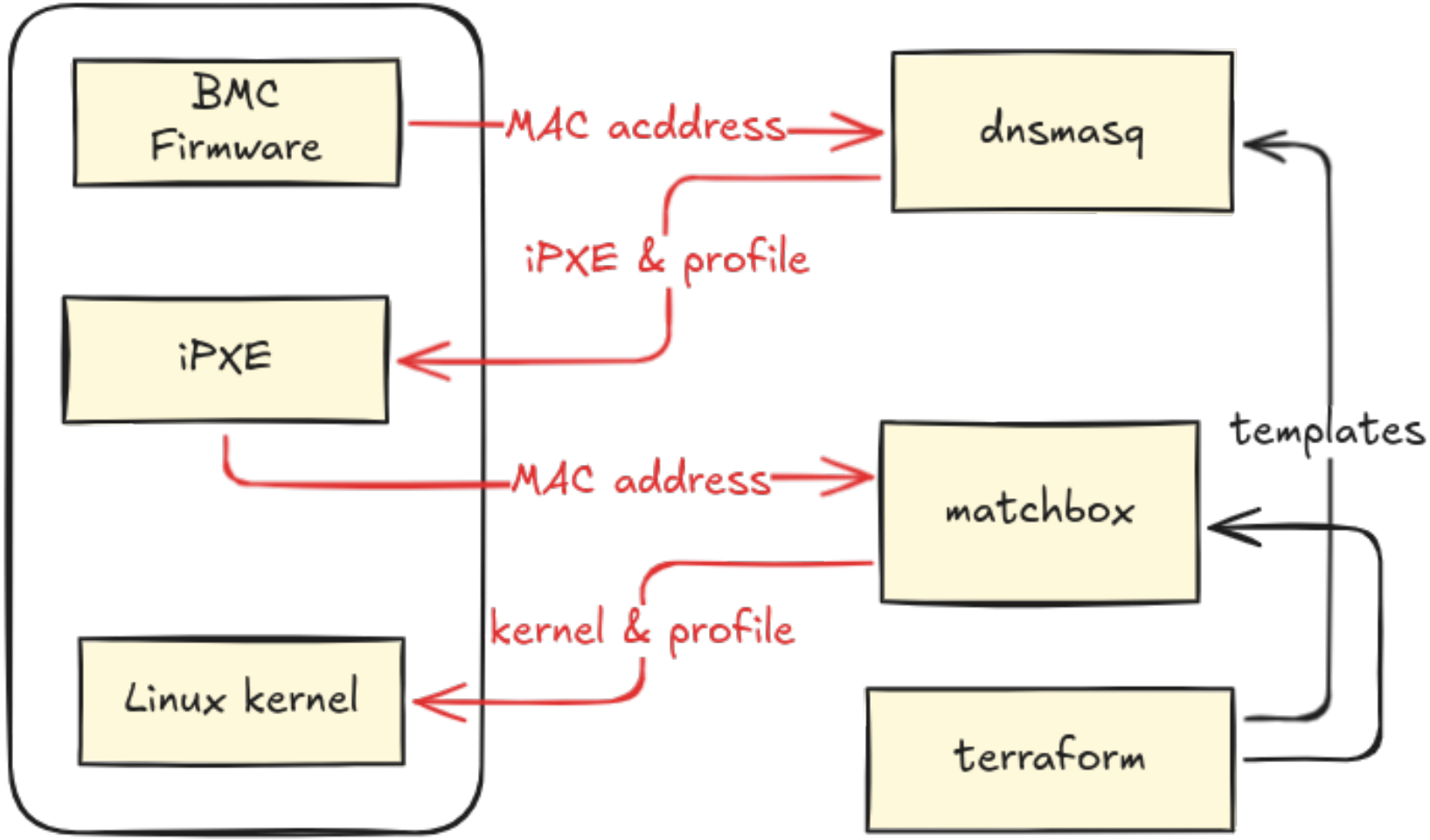
- Terraform defines resources, Ansible defines steps
- Terraform plans and runs, Ansible has `--check`

# Terraform vs. Ansible

```
TASK [k8s_base_network : Configure public interface] *********
--- before
+++ after
@@ -6,7 +6,7 @@
     "ipv4.ignore-auto-dns": "no",
     "ipv4.ignore-auto-routes": "no",
     "ipv4.may-fail": "yes",
-    "ipv4.method": "manual",
+    "ipv4.method": "disabled",
     "ipv4.never-default": "no",
  ...
```

- Terraform defines resources, Ansible defines steps
- Terraform plans and runs, Ansible has `--check`

# Deployment procedure



# New ansible roles

```
- hosts: stratus
    - old-stuff
    - { role: k8s_install, tags: 'k8s,base' }

- hosts: stratus_control
  gather_facts: no
  roles:
    - { role: k8s_controlplane, tags: 'k8s,base,control' }
    - { role: k8s_haproxy, tags: 'k8s,base,control' }
    - { role: k8s_manif_oidc, tags: 'k8s,manif,control' }
```

- Install kubernetes component
- Template HAproxy/keepalived
- Configure OIDC login

# Conclusion

Running Kubernetes is very easy, you just need:

- Some sort of connection on L3 (IP layer);
- Some sort of base OS;
- Container runtime;
- Run a few commands;

# Reconciliation

*Controllers are the core of Kubernetes, and of any operator.*

*It's a controller's job to ensure that, for any given object, the actual state of the world (both the cluster state, and potentially external state like running containers for Kubelet or loadbalancers for a cloud provider) matches the desired state in the object. Each controller focuses on one root Kind, but may interact with other Kinds.*

*We call this process reconciling.*

— *The Cluster API Book*

# Concept: CRDs and controllers

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
```

- each object in the database is a Resource and can be "reconciled"
- CRD: CustomResourceDefinitions

# Concept: services

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

- `ClusterIP`: IP for in-cluster access
- `NodePort`: expose on node port directly
- `LoadBalancer`: up to the controller to decide

# Concept: namespace

- Many resources are namespaced (pod, deploy, secret)
- Some are not (persistent volumes, nodes)
- CoreDNS can address service in a different namespace `nginx.(namespace.svc.cluster.local)`
- Example: looking at networking in a pod

# Summary

- one use Kubernetes by defining "resources"
- Kubernetes "reconciles" the "resources"
- services are connected together with ClusterIP and CoreDNS
- how does cluster network work?
- how do we expose the services?

# More jargon

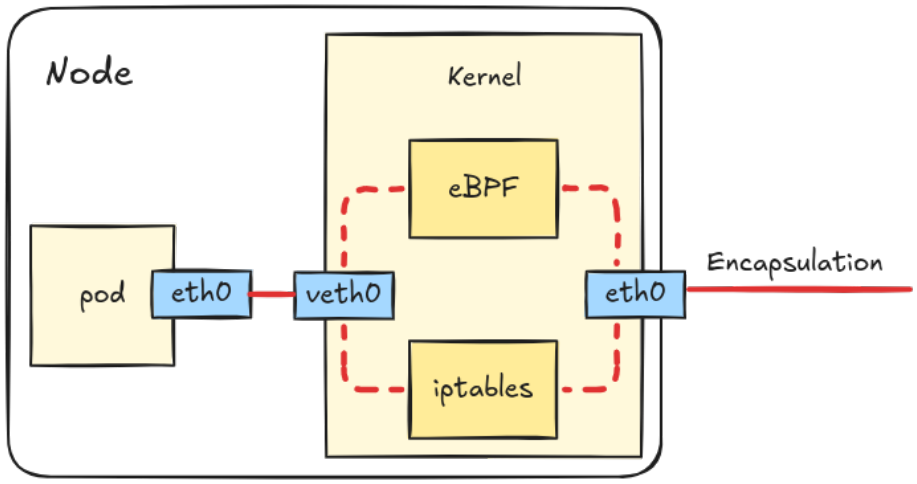
Kubernetes defines interfaces and leave a lot to the implementation. For a working cluster we must also have the networking part.

- CRI: container runtime interface (containerd, crio, docker)
- CNI: container network interface (cilium, calico, ...)
- CSI: container storage interface (ceph, cinder, proxmox, ...)

# Recap - networking at layer2-3

- layer2: address resolution mac address → LAN `10.0.0.0/16`,
- layer3: routing `10.0.0.0/8 via 10.0.0.1`
- Kubernetes assumes nodes are reachable at layer3
- we also want layer3 access across pods
- we need some software defined network

# CNI and kubeproxy




- Virtual ethernet pairs devices: veth / netkit
- Encapsulation tunneling: VXLAN / Geneve
- Kubeproxy: iptables / nftables / IPVS
- Cilium eBPF (can bypass kubeproxy if desired)

# Loadbalancer with Openstack

```
resource "openstack_lb_member_v2" "all" {
  for_each = local.lb-members
  name          = each.key
  pool_id       = each.value.pool.id
  address       = each.value.address
  protocol_port = each.value.port
}
```

OpenStack provides a LoadBalancer as a service, we can:

- Manually:
  - K8s: declare a HostPort service
  - Terraform: declare a Loadbalancer@129.16.x.y → HostPort
- With openstack loadbalancer controller
  - Declaring service as LoadBalancer@129.16.x.y

# Loadbalancer with bare-metal

```
{% for service in k8s_haproxy_nodeports %}
frontend {{ service.name }}
    bind *:{{ service.port }}
    default_backend {{ service.name }}-backend

backend {{ service.name }}-backend
    mode tcp
{% for server in k8s_haproxy_servers %}
    server {{ server.name }} {{ server.ip }}:{{ service.nodeport }}
{% endfor %}
{% endfor %}
```

With bare metal we do not have an API to announce an IP, usually do one of the three things:

- HAProxy as LB (setup service as host ports)
- Announce a L2 address in subnet (needs NAT to go public)
- Announce a BGP route (need ISP support)

# Loadbalancer with bare-metal

```
# to map 10.37.4.100 to public ip 129.16.125.174
# add 129.16.125.174/16 to gateway's public interface
firewall-cmd --permanent --direct --add-rule ipv4 nat POSTROUTING 0\
  -s 10.37.4.100 -o eth0 -j SNAT --to-source 129.16.125.174
firewall-cmd --permanent --direct --add-rule ipv4 nat PREROUTING 0\
  -d 129.16.125.174 -i eth0 -j DNAT --to-dest 10.37.4.100
```

With bare metal we do not have an API to announce an IP, usually do one of the three things:

- HAProxy as LB (setup service as host ports)
- Announce a L2 address in subnet (needs NAT to go public)
- Announce a BGP route (need ISP support)

# Summary

- Kubernetes expects functionally from provider APIs
- For us, NATs and loadbalancers are done statically

# Steps review

1. Prepare infrastructure (NAT, LB)
2. Prepare OS with access to internal VLAN
3. Install loadbalancer for apiserver
4. Install kubernetes
5. ‘kubeadm join –apiserver . . . ‘
6. ‘kubectl get nodes‘
7. install CNI (cilium)

# In concret steps

```
$ tofu apply
$ ipmi stratus-w3 chassis bootdev pxe
$ ipmi stratus-w3 chassis power cycle
$ ansible-playbook setup_stratus --limit stratus-w3
$ ssh stratus-w3 $(ssh stratus-c1 kubeadm token create\
                        --print-join-command)
```

# Alternative 1: immutable OS

```
variant: flatcar
version: 1.1.0
systemd:
  units:
    - name: kubeadm-join.service
      enabled: true
      contents: |
        [Unit]
        Description=Join kubernetes cluster as worker
        ConditionPathExists=!/etc/kubernetes/kubelet.conf
        [Service]
        ExecStart=...
filesystem:
  ...
```

- Challenge 1: distributing system updates
- Challenge 2: config change require restarting the node

# Alternative 2: ClusterAPI

```yaml
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: OpenStackCluster
metadata:
  name: xxxx
  namespace: xxx
spec:
  apiServerLoadBalancer:
    enabled: true
    provider: ovn
  externalNetwork:
    id: 96069ce5-c8e3-4dbf-a90c-xxxxxxx
  apiServerFloatingIP: 129.16.152.yy
```

- Cluster itself is defined as a Resource
- Challenge: need an API to deploy nodes (MAAS, Metal$^3$)
- Only teaser, come next session

# Design choices

- vanilla kubernetes experience
- minimal requirements

Pros.

- minimal requirements system across different providers
- familiar kernel and base distribution
- minimal API calls

Cons.

- not so agile
- not so automated