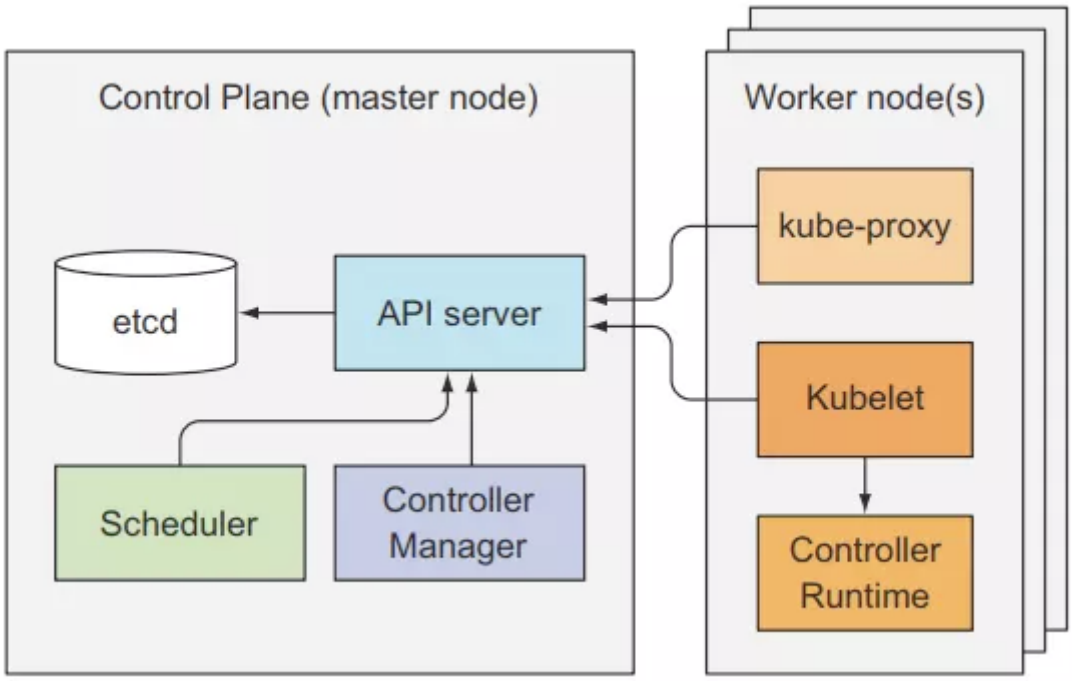
Kubernetes

Table of contents

Architecture



Một kubernetes sẽ có 2 thành phần chính đó là: master node (còn gọi là control plane) và worker node.

Trong master node bao gồm 4 component

- etcd
- API server
- Controller Manager
- Scheduler

Trong worker node sẽ gồm có 3 component:

- kubelet
- kube-proxy
- container runtime

Ngoài ra, k8s cũng có những add-on khác như:

- K8S Dashboard
- Ingress controller
- ...

Etcd

- Là một database(dạng key-value) lưu trữ các thông tin về các resource trong k8s.

API server

- Là thành phần trung tâm, được sử dụng bởi các component khác hoặc client.
- API server cung cấp một REST API để có thể thực hiện các hành động CRUD(create,read,update,delete) lên cluster state và lưu state vào etcd.
- API server cũng chịu trách nhiệm authentication, authorization, validation config của các resource.

Controller Manager

- Có nhiệm vụ create, update hoặc delete các resource thông qua API Server.
- Mỗi controller sẽ thực hiện các công việc khác nhau. Một số controller như: Replicaset, DaemonSet, Deployment

Scheduler

- Phụ trách việc lựa chọn worker để deploy pod.

Kubelet

- Đây là thành phần cài đặt trên pc cá nhân, có trách nhiệm tương tác với master node và quản lý các container.

Kube proxy

- Là thành phần quản lý traffic và network của worker node mà liên quan đến pod.

Container runtime

- Là engine để run các container. nổi bật nhất là docker. comment: <> (End Architecture page)

Pod

Giới thiệu Pod

- Pod là thành phần cơ bản để deploy và chạy ứng dụng trong k8s.
- Pod được dùng để nhóm và chạy một hoặc nhiều container với nhau trên cùng một worker node.
- Những containers trong cùng một pod, cùng chia sẻ network namespace → có thể truy cập với nhau qua `localhost`
- Mỗi pod được tạo sẽ được gán 1 IP, và các pod trong cluster có thể truy cập với nhau qua IP này.
- Để dễ dàng cho việc scaling, chỉ nên thiết kế 1 container trong 1 pod. Tuy nhiên, nếu app có những process phụ thuộc vào nhau → thiết kế nhiều container trong 1 pod.
- Ví dụ triển khai 1 pod trên k8s cluster.(yêu cầu tối thiểu phải có k8s cluster on cloud hoặc on-premise và đã cài đặt kubectl)
    - Chuẩn bị file config `kubia.yaml` như sau để tạo pod:

```
# kubia.yaml

apiVersion: v1
kind: Pod
metadata:
    name: kubia
spec:
    containers:
        - image: luksa/kubia
          name: kubia
          ports:
              - containerPorts: 8080
```

Đây là 1 structure cơ bản không chỉ để tạo pod mà còn để tạo các resource khác trong k8s. Một số thông số cần lưu ý

- kind: loại resource cần tạo trong k8s, có thể là Pod, Replicaset, Service
- metadata: chứa các thông tin như name, labels, annotations
- spec: tuỳ vào loại resource chúng ta cần tạo, ở đây là pod sẽ chứa các thông tin liên quan đến các cấu hình trong pod như image, containers, volume, ...

- Để tạo pod, chúng ta dùng câu lệnh:

```
$ kubectl apply -f kubia.yaml
```

- Bây giờ kiểm tra xem pod đó đã được tạo hay chưa, có bị lỗi không

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-26b5b   1/1     Running   0          29h
```

- Để kiểm tra log của log bằng câu lệnh

```
$ kubectl logs kubia
```

`kubia` là tên của pod, đã khai bảo trong file config

- Sử dụng câu lệnh dưới để forward port từ pod

```
$ kubectl port-forward kubia 8888:8080

Forwarding from 127.0.0.1:8888 -> 8080
Forwarding from [::1]:8888 -> 8080
```

- Kiểm tra bằng cách send request đến địa chỉ.

```
$ curl localhost:8888

You've hit kubia-26b5b
```

Pod labels

- Labels là cặp key-value được gán vào không chỉ pod và còn cho các resource khác trong k8s.
- Pod labels có nhiệm vụ định danh và tổ chức các pods.
- Một resource có thể có nhiều hơn 1 labels.
- Ví dụ config một pod với labels

```
apiVersion: v1
kind: Pod
metadata:
    name: kubia-label
    labels:
        env: prod
spec:
```

```
    containers:
        ...
```

Ở đây, chúng ta sử dụng label `env=prod`

- Bây giờ tạo label cho pod `kubia` (đã khởi tạo ở ví dụ trước)

```
$ kubectl label pod kubia env=debug
```

- Overwrite label pod ở ví dụ trên

```
$ kubectl label pod kubia-label env=debug --overwrite
```

- Show pod với label selector

```
$ kubectl get pod -l env=prod # all pod have lable env=prod

$ kubectl get pod -l env # all pod have env lable

$ kubectl get pod -l '!env' # all pod doesn't have env lable
```

Pod Annotation

- Giống như labels, annotation được gán cho mọi resource trong k8s
- Chỉ có tác dụng lưu trữ các thông tin metadata(author, created date, ...) phục vụ cho các thirdparty tools hoặc k8s administrator.
- Để annotate cho một pod, dùng command sau:

```
$ kubectl annotate pod kubia cmctelecom.vn/k8s="annotate k8s"
```

Namespace

- Namespace trong k8s được dùng để phân chia tài nguyên giữa các môi trường hoặc các tenant. Ví dụ: chia tài nguyên giữa các môi trường dev, production, QA.
- Các resource trong các namespace là hoàn toàn độc lập với nhau.
- Mặc định các resource được tạo trong namespace `default`
- Show toàn bộ namespace trong cluster

```
$ kubectl get namespace

# hoặc

$ kubectl get ns
```

- Show pod trong một namespace cụ thể

```
$ kubectl get pod --namespace kube-system
```

- Để tạo namespace:

```
$ kubectl create namespace custom-namespace
```

- Tạo pod trong custom namespace

```
$ kubectl apply -f kubia.yaml -n custome-namespace
```

Pod Deleting

- Xoá pod bằng pod name

```
$ kubectl delete pod kubia
```

- Xoá pod bằng label

```
$ kubectl delete pod -l env=debug
```

- Xoá toàn bộ pod

```
$ kubectl delete pod --all
```

- Xoá toàn bộ resource trong 1 namespace

```
$ kubectl delete all --all -n custom-namespace
```

Controller

ReplicationController

- Đảm bảo đủ số lượng pod chạy trong cluster không thừa không thiếu
- Khi 1 pod bị crash → replication controller sẽ tạo pod mới thay thế.
- Có 3 thành phần chính trong 1 replication controller
    - label selector: để xác định pod.
    - replica count: số lượng pod tối thiểu phải có
    - pod template: template dùng để tạo pod mới.
- Ví dụ file config `kubia-rc.yaml` cho replication controller

```
# kubia-rc.yaml

apiVersion: v1
kind: ReplicationController
metadata:
    name: kubia-rc
spec:
    replicas: 3
    selector:
        app: kubia
    template:
        metadata:
            labels:
                app: kubia
            spec:
                containers:
                    - name: kubia
                        image: luksa/kubia
                        ports:
                            - containerPort: 8080
```

Khi submit `kubia-rc.yaml` lên cluster, K8S tạo 1 replication controller có tên là `kubia-rc`. Controller này có nhiệm vụ đảm bảo trong cluster phải có đủ 3 pod với label `app=kubia` . Do mới đầu chưa đủ số lượng pod, nên nó sẽ tạo 3 pod mới.

- Submit file config lên cluster

```
$ kubectl create -f kubia-rc.yaml # tạo replication controller
```

```
$ kubectl get rc # check info của replication controller
$ kubectl describe rc kubia-rc
```

- Để scale số lượng pod, có thể dùng command:

```
$ kubectl scale rc kubia --replicas=10 # scale up

$ kubectl scale rc kubia --replicas=2 # scale down
```

- Để xoá replication controller

```
$ kubectl delete rc kubia-rc # xoá controller xoá luôn pod

$ kubectl delete rc kubia-rc --cascade=false # xoá controller nhưng
không xoá pod
```

Replicaset

- Replicaset có chức năng tương tự ReplicationController, nhưng được tối ưu hơn ở phần label selector.
- Replicaset có thể dùng nhiều label selector cùng lúc ngoài ra còn hỗ trợ dùng expression nâng cao hơn
- Ví dụ file config `kubia-replicaset.yaml` để tạo replicaset

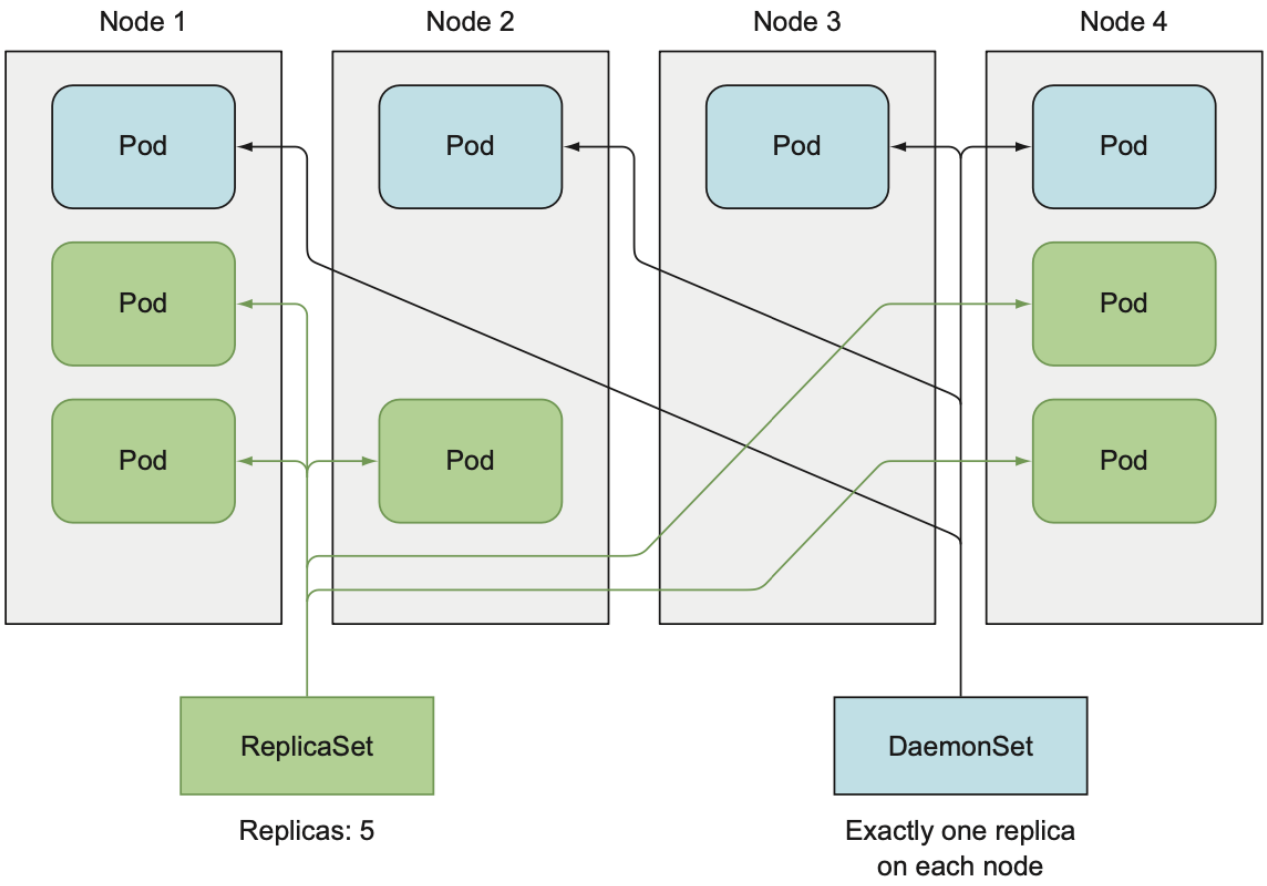
```yaml
# kubia-replicaset.yaml

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: kubia-replicaset
spec:
  replicas: 3
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
          - kubia-rs
  template:
    ...
```

Submit tạo replicaset trên cluster

```
$ kubectl create -f replicaset.yaml # tạo replicaset
$ kubectl get rs # list all replicaset
```

DaemonSet



- Đảm bảo chỉ có duy nhất 1 pod được deploy trên mỗi worker node.
- Phù hợp với các ứng dụng log collector hoặc monitor worker node.
- Khi có 1 worker node được thêm mới, daemonset sẽ tự động deploy pod lên worker node này.
- Ví dụ file config để tạo daemonset:

```
# daemonset.yaml

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
```

```
    metadata:
      labels:
        app: ssd-monitor
    spec:
      containers:
        - image: luksa/ssd-monitor
          name: main
```

Submit file config lên cluster

```
$ kubectl create -f daemonset.yaml # tạo daemonset
$ kubectl get ds # list all daemonset
```

Jobs

- K8S hỗ trợ để chạy những completable tasks qua Jobs Controller.
- Phù hợp với những ad-hoc task.
- Ví dụ file config tạo Job controller

```
# jobs.yaml

apiVersion: batch/v1
kind: Job
metadata:
  name: batch-job
spec:
  template:
    metadata:
      labels:
        app: batch-job
    spec:
      restartPolicy: OnFailure
      containers:
        - name: main
          image: luksa/batch-job
```

```
$ kubectl create -f jobs.yaml
$ kubectl get jobs
```

CronJob

- K8S cũng hỗ trợ chạy các cronjob qua CronJobs Controller

- Ví dụ file config tạo CronJobs Controller

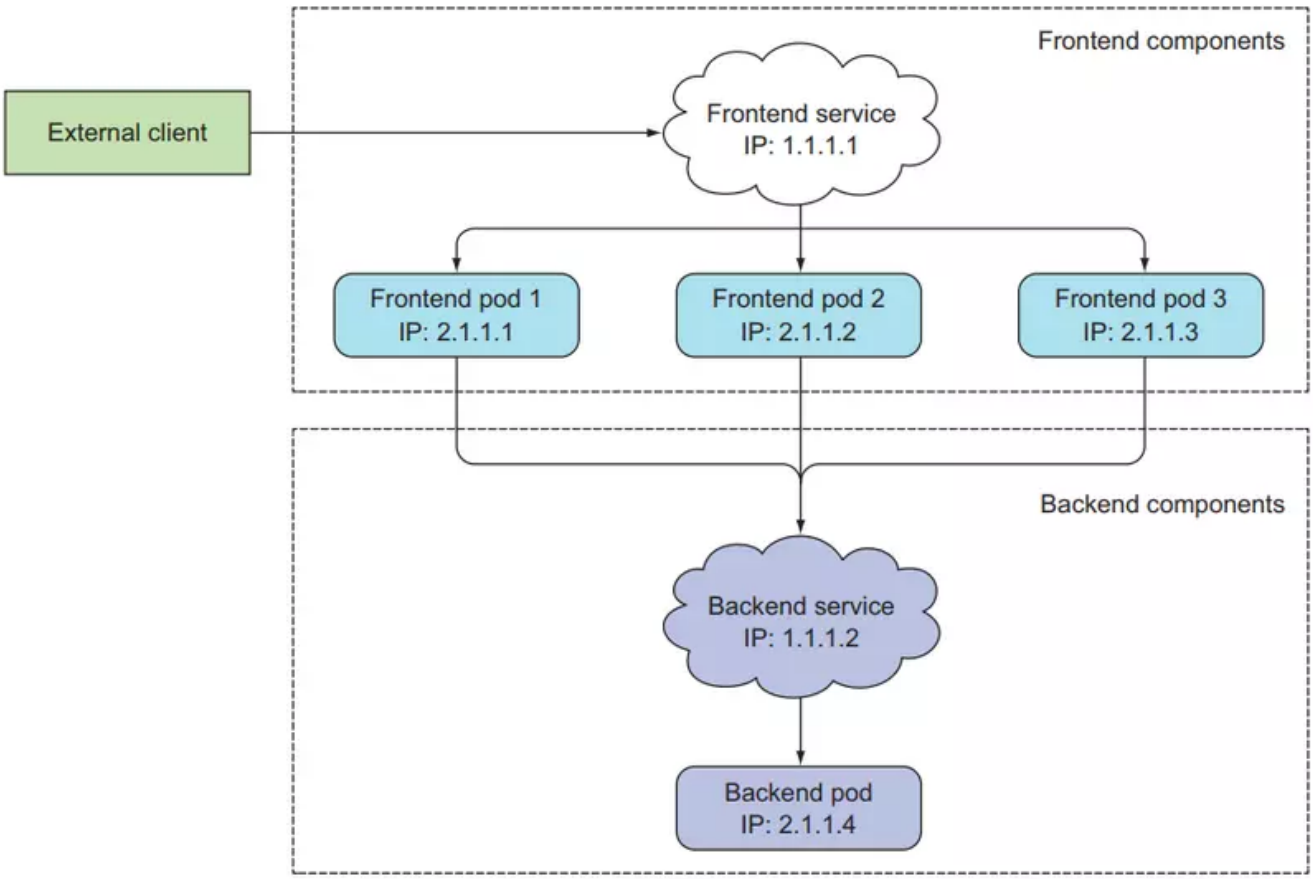
```
# cronjobs.yaml

apiVersion: batch/v1
kind: CronJob
metadata:
  name: schedule-job
spec:
  schedule: "0,7,10 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: schedule-job
        spec:
          restartPolicy: OnFailure
          containers:
            - name: main
              image: luksa/batch-job
```

```
$ kubectl create -f cronjobs.yaml
$ kubectl get cronjobs
```

Services

- Service là một resource trong k8s, có nhiệm vụ tạo ra 1 entry point cho group các pod
- Mỗi service sẽ có một IP address và port không đổi, chỉ trừ trường hợp xoá đi tạo lại.
- Client sẽ mở connect tới service và request sẽ được dẫn đến các pod phía sau nó.



- Ngoài ra, service còn có nhiệm vụ để các pod trong cùng 1 cluster có thể giao tiếp được với nhau.
- Vậy 1 câu hỏi được đặt ra là: các pod khi tạo đều có IP, tại sao chúng ta không sử dụng các IP này để connect mà lại cần dùng service? Có 2 lý do chính sau:
  - Thứ nhất, các pod chạy bình thường thì sẽ không có vấn đề gì, tuy nhiên trong trường hợp pod bị crash cần tạo lại → sinh ra IP mới → phải update lại cấu hình trong code.
  - Thứ hai, giả sử chúng ta có 3 pod cùng serving. Vậy phía client sẽ biết gửi request đến pod nào → service sẽ giải quyết cho chúng ta bằng cách tạo 1 entry point không đổi cho group các pod, client chỉ cần tương tác với entry point này là được.
- Có 4 loại service cơ bản:
  - ClusterIP
  - NodePort
  - LoadBalancer
  - Ingress

ClusterIP

- Là loại service sẽ tạo một IP address và local DNS để các pod giao tiếp bên trong với nhau, không thể access từ bên ngoài.
- Ví dụ tạo service cho group pod có label là `kubia`

```
# service.yaml

apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
    - port: 80 # service port
      targetPort: 8080 # serving port
  selector:
    app: kubia
```

Submit lên cluster để tạo service

```
$ kubectl create -f service.yaml
$ kubectl get svc
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)
kubia           ClusterIP      10.100.0.1       <none>
80/TCP
```

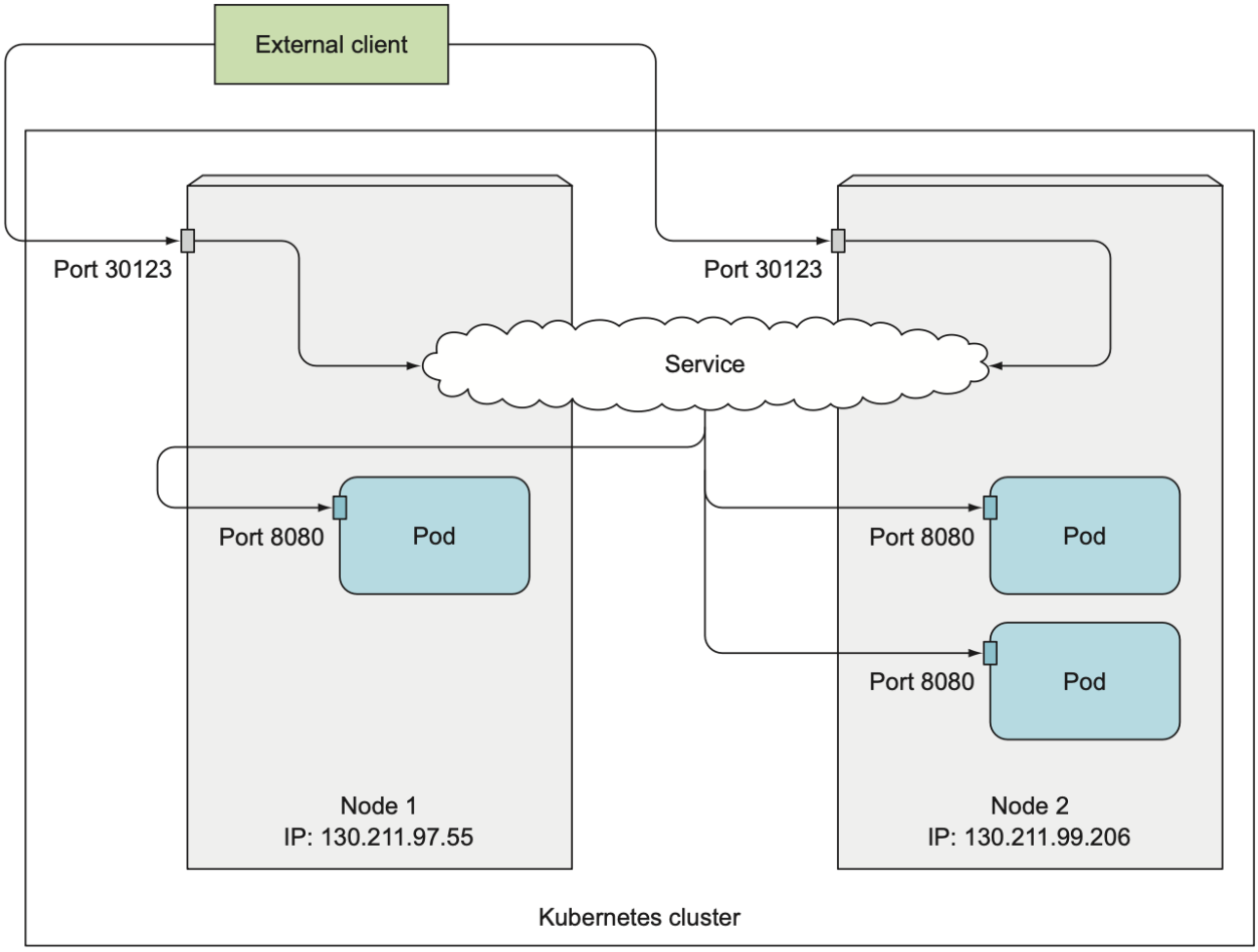
Test service:

```
$ kubectl exec kubia-7nog7 -- curl -s http://10.0.240.111 # kubia-
7nog7 là pod name

$ kubectl exec kubia-7nog7 -- curl -s http://kubia.default # kubia là
service name, default là namespace
```

NodePort

- Là 1 cách để expose pod để client connect từ bên ngoài vào.
- Giống ClusterIP, nodeport cũng tạo endpoint để truy cập bên trong, đồng thời nó sẽ mở một port trên toàn bộ worker node để client bên ngoài giao tiếp với pod qua port đó.
- NodePort range từ 30000-32767



- Ví dụ file config để tạo Nodeport

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-nodeport
spec:
  type: NodePort
  ports:
    - port: 80 # cluster/service port
      nodePort: 30123 # node port
      targetPort: 8080 # serving port
  selector:
    app: kubia
```

Test service:

```
# giả sử worker node có public IP là 139.21.41.43
$ curl http://139.21.41.43:30123
```

LoadBalancer

- Khi sử dụng k8s trên cloud, sẽ có hỗ trợ loadbalancer service(sẽ không tạo được service này nếu môi trường không hỗ trợ loadbalancer)
- Là dạng mở rộng của nodeport, nó sẽ tạo ra một public IP, client có thể truy cập pod qua public ip này.


LoadBalancer

- Ví dụ file config loadbalancer

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-nodeport
spec:
  type: LoadBalancer
  ports:
    - port: 80 # loadbalancer port
      targetPort: 8080 # serving port
  selector:
    app: kubia
```

Ingress

- Do mỗi service mà sử dụng Loadbalancer service sẽ cần 1 public ip → tốn chi phí triển khai → Do vậy Ingress được sinh ra giải quyết vấn đề này.
- Ingress hỗ trợ chỉ cần một public IP access cho rất nhiều services bên trong.
- Ngoài ra còn hỗ trợ gán domain cho service bên trong cluster.


Ingress

- Ví dụ tạo ingress config:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kubia-ingress
spec:
  rules:
    - host: kubia.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: kubia
                port:
                  number: 80
```

Volume

- Volume hiểu đơn giản là một mount point từ một hệ thống file server vào trong container.
- Volume không phải 1 resource của K8S nên không thể tạo và xoá như resource độc lập khác mà được define khi khởi tạo pod.
- Có nhiều loại volume khác nhau, mỗi loại có 1 công dung riêng không giống nhau.

emptyDir

- là loại volume đơn giản nhất dùng để share data giữa các container trong cùng một pod.
- volume này chỉ tồn tại trong lifecycle của pod, khi pod bị xoá → volume cũng bị mất.
- ví dụ file config tạo pod với volume

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  containers:
    - image: luksa/fortune
      name: html-generator
      volumeMounts:
        - name: html
          mountPath: /var/htdocs
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
      ports:
        - containerPort: 80
  volumes:
    - name: html
      emptyDir: {}
```

Ở đây, chúng ta có 2 container và 1 emptyDir volume là html, nó được mount vào container html-generator ở folder /var/htdocs, đồng thời cũng được mount vào container web-server ở folder /usr/share/nginx/html. Do đó, khi html-generator tạo file html thì bên web-server sẽ nhìn thấy và đọc được file này.

hostPath

- Là loại volume để mount từ trong pod ra ngoài worker node.

- Data trong volume này chỉ tồn tại trên worker node và không bị xoá khi xoá pod.
- Tuy nhiên, do các vấn đề về security, nên loại volume này không được khuyến khích sử dụng.

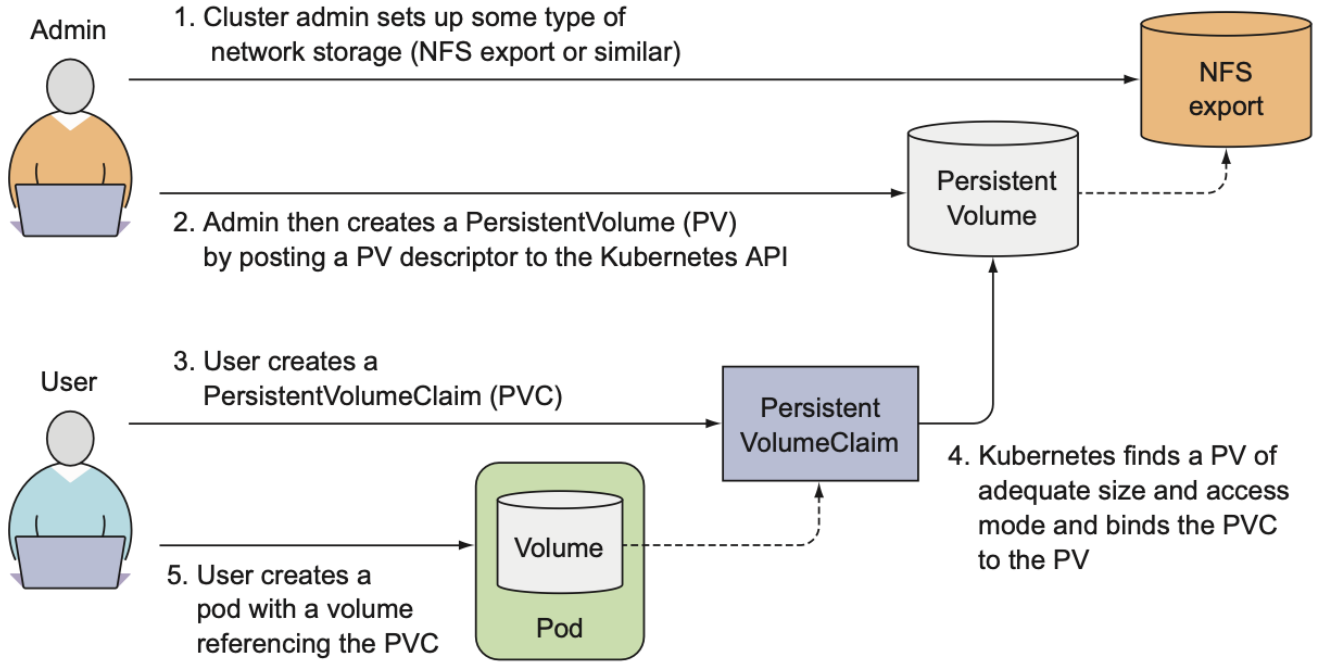
Cloud Storage

- Loại volume này chỉ hỗ trợ trên các nền tảng cloud (aws,azure,gcp,...)
- Dùng để lưu trữ persistent data, dữ liệu của chúng ta vẫn tồn tại cho các container.
- Tương ứng mỗi nền tảng cloud sẽ có những cấu hình khác nhau, ví dụ aws có awsElasticBlockStore, azure có azureDisk, ...
- Ví dụ config tạo mongodb pod với aws volume

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
    - image: mongo
      name: mongodb
      volumeMounts:
        - name: mongodb-data
          mountPath: /data/db
      ports:
        - containerPort: 27017
  volumes:
    - name: mongodb-data
      awsElasticBlockStore:
        volumeID: vol-xxx
        fsType: ext4
```

PersistentVolumes & PersistentVolumeClaims

- Là loại volume có khả năng tách pod với công nghệ lưu trữ bên dưới. Tức là chúng ta tạo pod mà không cần quan tâm công nghệ storage bên dưới là gì.
- Trong đó:
  - PersistentVolumes(PV) là resource tương tác với kiến trúc lưu trữ.
  - PersistentVolumeClaims(PVC) sẽ request storage từ PersistentVolumes.
- Thông thường trên một hệ thống k8s cluster sẽ có 2 role là:
  - Administrator: là người triển khai và quản lý k8s, cài những add-on cần thiết, sẽ là người quyết định công nghệ lưu trữ.
  - Developer: là những người viết file config yaml để deploy application lên cluster, tức là những người sử dụng không cần quan tâm đến công nghệ lưu trữ phía sau.
- Một kubernetes administrator sẽ là người setup kiến trúc storage bên dưới và tạo các PersistentVolumes để cho kuberetes developer request và sử dụng.



- Các bước khởi tạo PV và PVC
  - Bước 1: Administrator khởi tạo PersistentVolume trên cluster (lưu ý PV không phụ thuộc vào bất kỳ namespace nào):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  awsElasticBlockStore:
    volumeID: vol-0d4d4ef91a4fd03ae
    fsType: ext4
```

    Ở đây Administrator cần phải chỉ định size cho PV và các access mode cho nó. Ngoài ra còn sử dụng công nghệ lưu trữ là gì trên cloud hay on-premise.

  - Bước 2: Developer khởi tạo PersistentVolumeClaims, tiêu thụ PersistentVolume.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
```

```
  resources:
    requests:
      storage: 10Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: ""
```

Ở đây, developer tạo 1 PVC có request là 10GB storage, nếu có PV nào đáp ứng được yêu cầu này thì PVC được bind vào PV đó.

- Bước 3: Kiểm tra pv và pvc

```
$ kubectl get pv

$ kubectl get pvc
```

- Bước 4: Sử dụng PVC trong pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
    - image: mongo
      name: mongodb
      volumeMounts:
        - name: mongodb-data
          mountPath: /data/db
      ports:
        - containerPort: 27017
  volumes:
    - name: mongodb-data
        persistentVolumeClaim:
                claimName: mongodb-pvc
```

Recycling PersistentVolume

- Khi tạo PV chúng ta sử dụng thuộc tính persistentVolumeReclaimPolicy, có 3 giá trị là retain, recycle và delete
  - retain mode: khi xoá PVC, thì PV vẫn còn nhưng ở trạng thái Release(tức là đã có dữ liệu) → nếu có PVC bind vào sẽ gây ra lỗi.
  - recycle: khi xoá PVC, PV vẫn tồn tại, nhưng lúc này data trong PV cũng bị xoá đi luôn → PVC mới có thể bind vào
  - delete: khi xoá PVC thì xoá luôn PV

Dynamic Provisioning PersistentVolume

- Khi dùng PV, Administrator vẫn cần phải tạo thủ công → K8S có hỗ trợ tạo PV tự động
- K8S tạo PV qua storageclass với provisioner (chỉ support trên các cloud platform, dưới môi trường không phải cloud thì phải các cài provisioner này)
- Tạo storage class với aws provisioner

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ext4
```

- Tạo PVC sử dụng storage class

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  resources:
    requests:
      storage: 10Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
```

- Lúc này sau khi submit lên cluster để tạo PVC, trên giao diện aws ec2 → volume sẽ có một volume có storage là 10GB như vừa tạo comment: <> (End Volume page)

ConfigMap & Secret

- Hầu hết các ứng dụng của chúng ta cần config để chạy (ví dụ như config cho database, config port, ...)
- Để truyền config vào container trong pod, chúng ta dùng
  - Biến môi trường(environment)
  - Hoặc k8s configmap

env

- K8S cung cấp cho cách để truyền các config vào trong container bằng cách thêm qua biến môi trường `env` khi khởi tạo pod.
- Ví dụ file config tạo pod truyền config qua `env`:

```
apiVersion: v1
kind: Pod
spec:
    containers:
        - image: luksa/fortune:env
          name: html-generator
          env:
              - name: INTERVAL
                value: "30"
```

- Nhược điểm của phương pháp này:
  - env không thể update trong container khi container đã chạy, muốn update phải xoá pod và chạy lại.
  - khi số lượng pod lớn → config sẽ bị lặp lại hoặc file config tạo resource k8s sẽ trở nên dài hơn nếu pod sử dụng nhiều config.

ConfigMap

- Là loại resource giúp tách config của container khỏi file cấu hình tạo pod. Được đinh nghĩa theo kiểu key-value
- Cơ chế: Giá trị trong configmap sẽ được truyền vào trong container như một env.
- Ưu điểm: Bởi vì được tách riêng nên các config được tập trung ở một chỗ và có thể sử dụng lại nhiều lần ở những container khác nhau.
- Cách để tạo ConfigMap
  - dùng qua cli

```
$ kubectl create configmap <config_name> --from-literal=<key1>=
<value1>

--from-literal=<key2>=<value2>
```

- dùng file config hoặc directory

```
$ kubectl create configmap <config_name> --from-file=
<path_to_file>

# hoặc

$ kubectl create configmap <config_name> --from-file=
<path_to_dir>
```

- Ví dụ tạo configmap qua cli

```
$ kubectl create configmap fortune-config --from-literal=sleep-
interval=25
```

- Truyền configmap vào pod
  - truyền 1 entry trong configmap

```
apiVersion: v1
kind: Pod
metadata:
    name: fortune-env-from-config-map
spec:
    containers:
        - image: luksa/fortune
          name: html-generator
          env:
              - name: INTERVAL
                valueFrom:
                    configMapKeyRef:
                        name: fortune-config
                        key: sleep-interval
```

  - truyền nhiều hơn 1 entry

```
apiVersion: v1
kind: Pod
metadata:
```

```
    name: fortune-env-from-config-map
spec:
    containers:
        - image: luksa/fortune
          name: html-generator
          envFrom:
              - prefix: CONFIG_
                valueFrom:
                    configMapKeyRef:
                        name: fortune-config
```

Configmap with file

- configmap, ngoài việc sử dụng value ở dạng plaintext, thì value còn có thể ở dạng file config
- Ví dụ
    - dưới đây là 1 file config nginx

```
# nginx.conf

server {
    listen  80;
    server_name kubia-example.com;
    gzip on;
    gzip_types text/plain application/xml;

    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
    }
}
```

    - Ta tạo configmap với key là tên của file `nginx.conf`, value là nội dung của file.

```
$ kubectl create configmap nginx-configmap --from-file nginx.conf
```

    - Tiếp theo tạo 1 pod sử dụng configmap chúng ta vừa tạo:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-cm
spec:
  containers:
    - image: nginx:alpine
```

```
        name: web-server
        ports:
          - containerPort: 80
        volumeMounts:
          - name: config
            mountPath: /etc/nginx/conf.d
                       subPath: nginx.conf
            readOnly: true
    volumes:
      - name: config
        configMap:
          name: nginx-configmap
                  defaultMode: "6600"
```

Ở đây, configmap được sử dụng như một volume, với content là content trong Configmap. Lúc này volume sẽ chứa 1 file với tên là `nginx.conf` , sau đó nó được mount vào folder trong container.

- Khi sử dụng Configmap ở dạng volume, nếu thay đổi giá trị của configmap, thì nó sẽ tự động được update bên trong volume mà không cần xoá pod đi tạo lại.
    - Ví dụ update configmap

```
$ kubectl edit nginx-configmap
```

    - Và sau đó reload lại container

```
$ kubectl exec nginx-cm -- nginx -s reload
```

Secret

- Cơ chế giống như ConfigMap, Secret cũng dùng để lưu trữ config nhưng ở dạng sensitive data.
- Không phải ai cũng quyền read Secret.
- Ví dụ ta tạo secret cho file https nginx

```
$ kubectl create secret generic nginx-https --from-file=https.key
```

- Và sử dụng trong pod tương tự như cách dùng configmap volume

```
```

```
apiVersion: v1
kind: Pod
metadata:
    name: pod-secret
spec:
    containers:
        - image: nginx:alpine
          volumeMounts:
              - name: certs
                mountPath: /etc/nginx/certs
    volumes:
        - name: certs
          secret:
              secretName: nginx-https
```

Automatic scaling

Khi nói về scaling, chúng ta có 2 kiểu scaling đó là horizontal scaling và vertical scaling

- horizontal scaling: tức là tăng số lượng worker đang xử lý công việc hiện tại lên nhiều hơn.
- vertical scaling: tức là thay vì thêm số lượng thì ta tăng thêm resource của worker node.

Trong K8S, chúng ta có thể horizontal scale bằng cách thay đổi số `replicas` trong Replicaset hoặc Deployment và vertical scale bằng cách tăng resource request và limit khi khởi tạo Pod. Tuy nhiên, các cách này chúng ta phải xử lý một cách thủ công. Rõ ràng đây không phải là một cách hay bởi vì chúng ta không thể ngồi theo dõi hệ thống cả ngày xong rồi gõ câu lệnh scale khi cần.

→ K8S sẽ hỗ trợ chúng ta monitor và scale up/down hệ thống tự động khi nhận thấy có sự thay đổi về các metrics(CPU, RAM, ...) trong k8s cluster. Ngoài ra, khi sử dụng k8s trên nền tảng cloud, k8s còn hỗ trợ tự động thêm worker node khi cần thiết.

Horizontal Pod Scaling

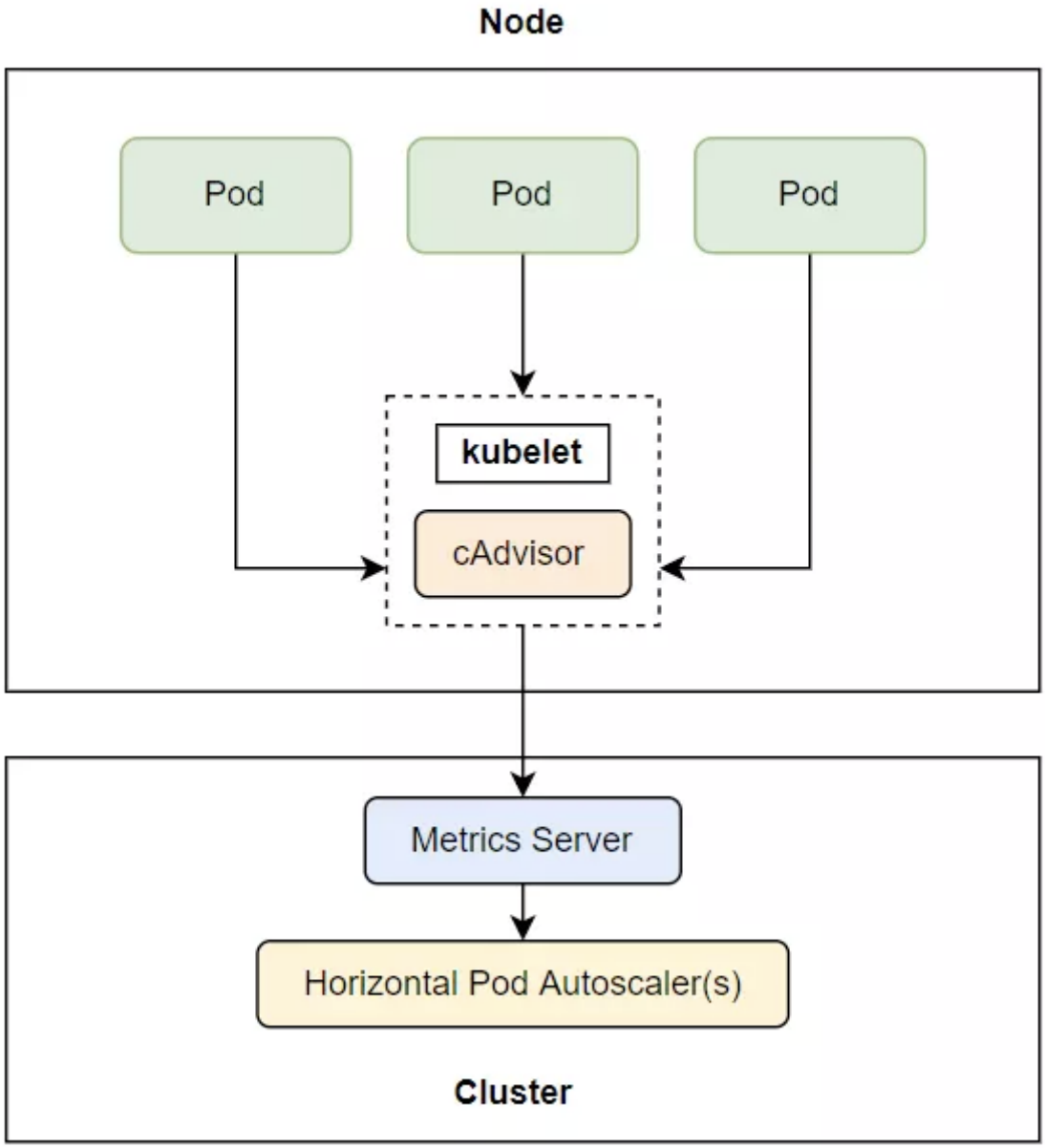
- Horizontal pod scaling là cách tự động tăng/giảm số lượng `replicas` trong các scalable controller(Replicaset, Deployment, ...)
- Công việc này được thực hiện bởi Horizontal Controller khi chúng ta tạo ra một HorizontalPodAutoscaler(HPA) resource.
- Controller này sẽ định kỳ kiểm tra các pod metrics và tính toán số lượng `replicas` phù hợp dựa trên giá trị của pod metrics hiện tại và giá trị metrics chúng ta chỉ định trong HPA, sau đó nó thay đổi trường `replicas` trong các resource (Replicaset, Deployment, StatefulSet)

1. Quá trình auto scaling

- Quá trình auto scaling được chia làm 3 giai đoạn:
  - Thu thập metrics của các pod được quản lý bởi resource mà chúng ta chỉ định trong HPA.
  - Tính toán số lượng pod cần thiết dựa vào metrics đã thu thập được
  - Update trường `replicas`

1. Thu thập metrics

  - Horizontal controller sẽ không trực tiếp thu thập các metrics mà nó sẽ lấy qua 1 thằng khác, gọi là metrics server.
  - Ở trên mỗi worker node, sẽ có một thằng agent gọi là cAdvisor, có nhiệm vụ thu thập metrics của pod và worker node, sau đó những metrics này được tổng hợp ở metrics server và Horizontal controller sẽ lấy metrics qua nó.



2. Tính toán số pod

- Sau khi có được metrics, Horizontal Controller sẽ tiến hành bước tính toán số pod cần thiết dựa vào các metrics ta define trong HPA.
- Nó sẽ cần tính toán dựa vào 2 thằng metrics phía trên, với input là một nhóm pod metrics và output là số lượng pod cần để scale.
    - Nếu chỉ có 1 single metric, việc tính toán dựa vào công thức

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Ví dụ nếu currentMetric là 200m, giá trị desiredMetric là 100m, currentReplicas là 2 thì

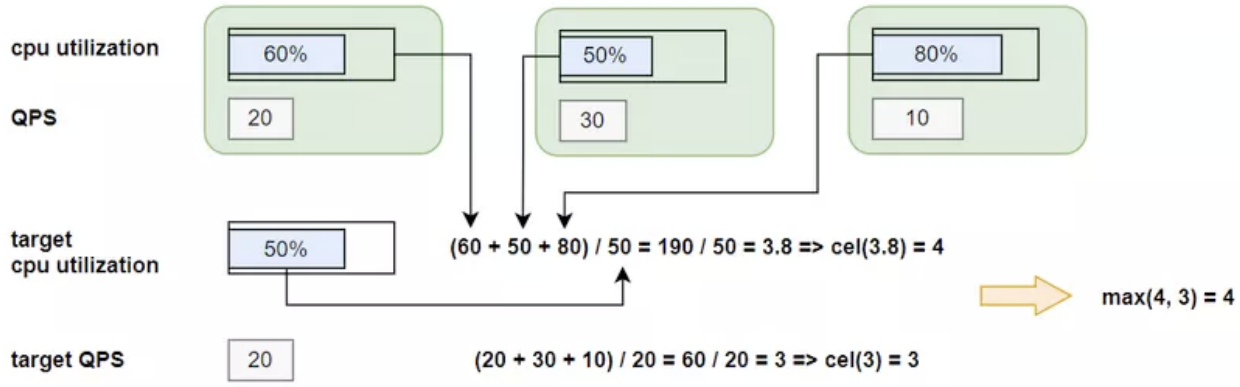
```
disiredReplicas = ceil(2 * (200m/100m)) = 4
```

Số `replicas` được scale từ 2 → 4.

- Nếu sử dụng nhiều hơn 1 metrics, việc tính toán cũng khá đơn giản

```
disiredReplicas = max(ceil(sum(metric_one,metric_two,...) /
disiredMetric), currentReplicas)
```

Ví dụ chúng ta sử dụng CPU và QPS(Queue-per-second)



c. Cập nhật `replicas`

- Bước cuối trong quá trình auto scaling đó là cập nhật lại số replicas theo như tính toán.
- Horizontal Controller sẽ
- Hiện tại, auto scaling mới hỗ trợ các resource sau:
  - Replicaset(ReplicaController)
  - Deployment
  - StatefulSet

1. Scaling theo CPU

- Giờ chúng ta sẽ tạo Deployment và một HPA với cấu hình sử dụng CPU metrics.
  - Tạo deployment với replicas là 3

```
# kubia-deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubia-deployment
spec:
  replicas: 3
  selector:
    app: kubia
  template:
```

```
    metadata:
      labels:
        app: kubia
    spec:
      containers:
        - image: luksa/kubia:v1
          name: nodejs
          resources:
            requests:
              cpu: 100m # 100 milicore
```

- Tạo HPA, với Utilization

```
# hpa.yaml

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: kubia-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: kubia-deployment
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 30
```

- Kiểm tra thử

```
$ kubectl create -f kubia-deployment.yaml

$ kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
kubia-deployment   3/3     3            3           69s

$ kubectl create -f hpa.yaml

$ kubectl get hpa
NAME        REFERENCE                     TARGETS         MINPODS
MAXPODS   REPLICAS   AGE
kubia-hpa   Deployment/kubia-deployment   <unknown>/30%   1
3         0          4s
```

Nếu thấy Target là `unknow/30%` thì phải chờ để cAdvisor có thể thu thập metrics.

- check lại

```
NAME        REFERENCE                      TARGETS   MINPODS
MAXPODS   REPLICAS   AGE
kubia-hpa   Deployment/kubia-deployment   0%/30%    1         3
3          12m
```

Lúc này cAdvisor đã report dữ liệu lên metric-server, và Horizontal Controller đã lấy được data.

- kiểm tra deployment

```
$ kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
kubia-deployment   1/1     1            1           18m
```

Như chúng ta có thể thấy trước khi tạo HPA, replicas là 3, bây giờ đã scale down xuống 1, do HPA đã kiểm tra thấy không có CPU nào được sử dụng → tự động scale down về minimum để giảm chi phí.

- trigger scale up

Trước tiên, chúng ta phải expose service để client có thể gọi vào

```
$ kubectl expose deployment kubia --port=80 --target-port=8080
```

Lúc này ta sẽ dùng lệnh `watch` để xem quá trình scale up

```
$ watch -n 1 kubectl get hpa, deployment
```

Mở 1 terminal khác rồi chạy command, nó có tác dụng là liên tục send request vào pod.

```
$ kubectl run -it --rm --restart=Never loadgenerator --
image=busybox -- sh -c "while true; do wget -O - -q http://kubia;
done"
```

Lúc này quay lại terminal có câu lệnh watch, chúng ta có thể thấy TARGET đã vượt quá 30% và pod được tự động scale up theo số number tính toán được ở đây là 3.

```
NAME                                                  REFERENCE
TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/kubia-hpa   Deployment/kubia-
deployment   79%/30%   1         3         3          38m

NAME                                READY   UP-TO-DATE   AVAILABLE
AGE
deployment.apps/kubia-deployment   3/3     3            3
42m
```

Deploy EKS cluster with eksctl

Dưới dây là cách triển khai k8s cluster (eks cluster) trên aws với eksctl

1. Cài đặt kubectl: https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html
2. Cài đặt eksctl: https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html
3. Tạo EKS cluster

Để tạo eks cluster với tên `my-cluster` ở region `ap-southeast-1` , chúng ta có thể chạy command sau: (—without-nodegroup: không khởi tạo worker node)

```
$ eksctl create cluster --name my-cluster --region ap-southeast-1 --
without-nodegroup
```

Quá trình khởi tạo mất 1 vài phút, khi xuất hiện output log như bên dưới → tạo eks cluster thành công.

```
[✓]  EKS cluster "my-cluster" in "ap-southeast-1" region is ready
```

1. Tạo worker node(nodegroup) Để tạo worker node, chúng ta có thể dùng 2 cách là : dùng qua cli command hoặc tạo file config
    1. Dùng command

        ```
        $ eksctl create nodegroup \
                        --cluster my-cluster \
                        --region ap-southeast-1 \
                        --name my-workers \
                        --node-type t3.medium \
                --nodes 2 \
                        --ssh-access \
                        --ssh-public-key my-key
        ```

        Sau khi chạy command phía trên sẽ tạo 2 worker node(ec2 - t3.medium) cho k8s cluster `my-cluster`

    2. Dùng template

        Tạo template file `eks-nodegroup.yaml`

        ```
        apiVersion: eksctl.io/v1alpha5
        kind: ClusterConfig
        ```

```
metadata:
    name: my-cluster
    region: ap-southeast-1
managedNodeGroups:
- name: my-workers
    amiFamily: Ubuntu2004
    instanceType: t3.medium
    desiredCapacity: 2
    ssh:
        publicKeyName: my-key
```

Chạy command để apply

```
$ eksctl create nodegroup --config-file eks-nodegroup.yaml
```

2. Tạo kubeconfig

Để remote access vào eks cluster → tạo kubeconfig trên PC. Chúng ta chạy lệnh sau:

```
$ aws eks update-kubeconfig --region ap-southeast-1 --name my-cluster
```

1. Kiểm tra k8s cluster-info

```
# check cluster
$ kubectl cluster-info

# check nodes
$ kubectl get nodes
```