EKS Workshop

17/04/2024

Introduzione

- **Obiettivi:**
  - Panoramica Docker e Kubernetes
  - Competenze base di interazione con clusters esistenti
  - Disamina dei tools principali ed esempi del loro utilizzo
  - Esempio di pipeline per il rilascio di software

- **Non obiettivi:**
  - Conoscenza approfondita di containers, Docker, Kubernetes/EKS
  - “Punto di arrivo” di competenze necessarie all’operatività completa sull’ambiente

- **Compiti a casa:**
  - Studio individuale delle tecnologie di cui sopra in base allo scope desiderato/richiesto
  - Tecnologia in rapidissima evoluzione → trial and error “decentralizzato”

Containers

- “Containerizzare” un’applicazione significa impacchettarla insieme a tutto ciò che le serve per funzionare: codice, librerie, configurazioni, dependency, runtime, ecc.

- Il risultato è un *container*, ovvero un’unità eseguibile, leggera e isolata, che può essere lanciata ovunque, su un laptop, un server bare-metal, una VM o nel cloud, con lo stesso comportamento.

- Vantaggi:
  - Rendere l'app **portabile** e **replicabile** in ogni ambiente
  - Evitare il classico “funziona sul mio pc”
  - Avere un sistema più **scalabile** e **manutenibile**
  - Facilitare il deployment e l'integrazione con strumenti di CI/CD

Containers: non una novità

1970
Unix V7
chroot

2000
FreeBSD
jails

2001-2006
Linux VServer,
OpenVZ, Solaris
snapshots

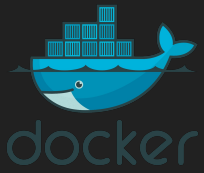
Google

2006
Process Containers
a.k.a.
cgroups

2008
RedHat
namespaces

Containers: non una novità

2008
LXC
tools per cgroups e namespaces



2013
Docker

2015
Kubernetes

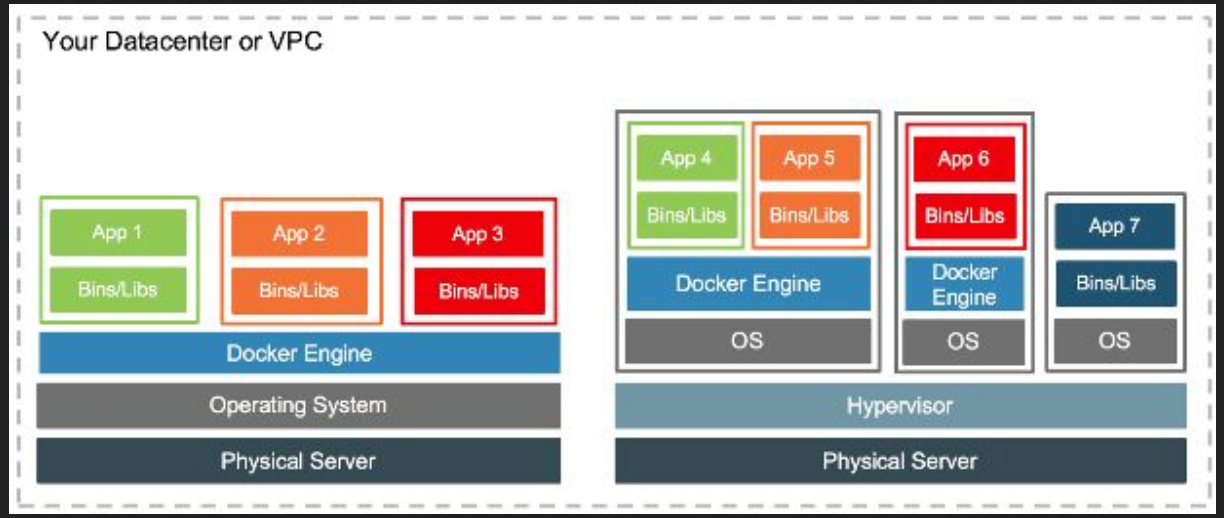
2017
Specifiche OCI

> 2020
mass adoption, CI/CD, rootless, AI tools, Mars exploration, ecc.

Containers e VM

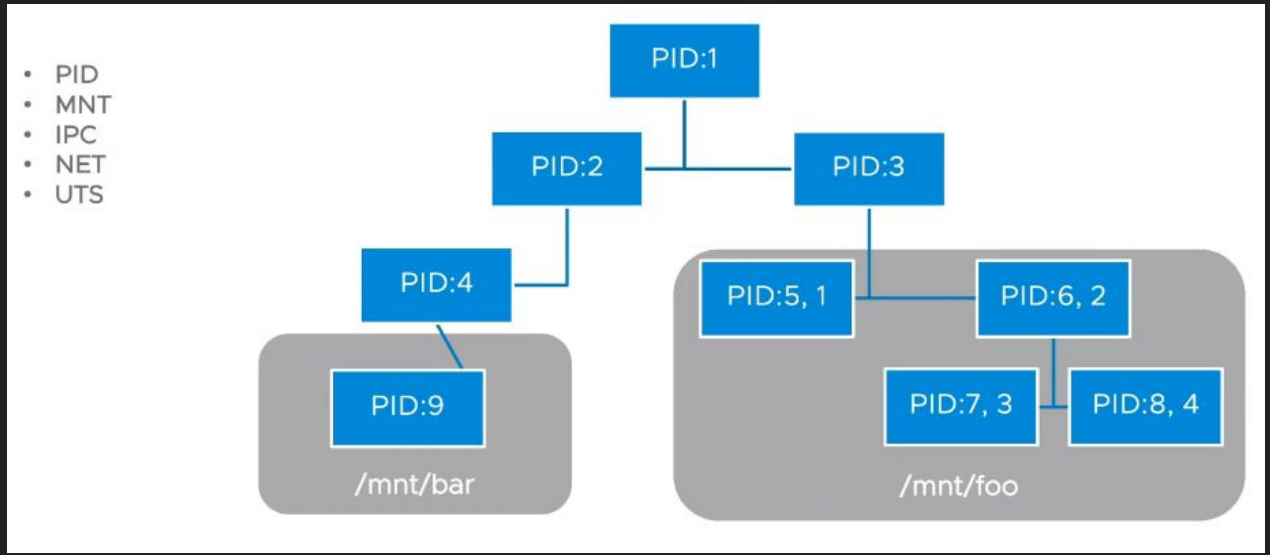
- **Docker** è una piattaforma che semplifica l'interazione con le features del kernel e automatizza il deployment di applicazioni dentro **container**.

- I container **condividono il kernel dell'host**, a differenza delle VM, e sono quindi più leggeri e veloci.



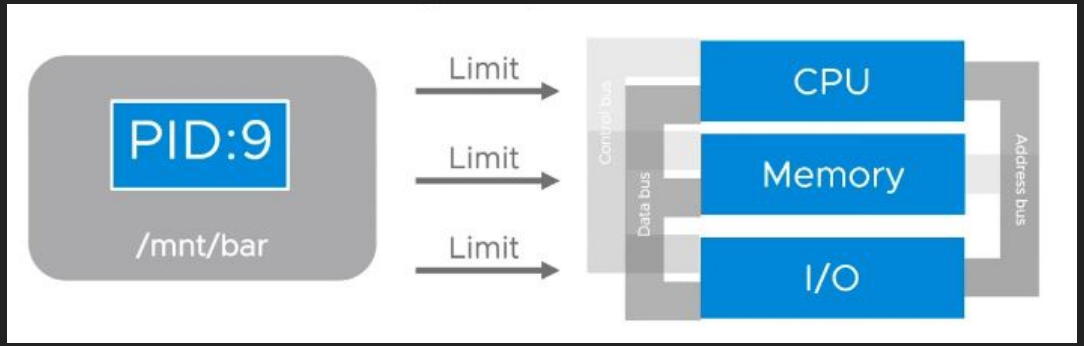
Docker e containers

- Si basa su **namespace** (isolamento) e **cgroups** (limitazione risorse) del kernel Linux.



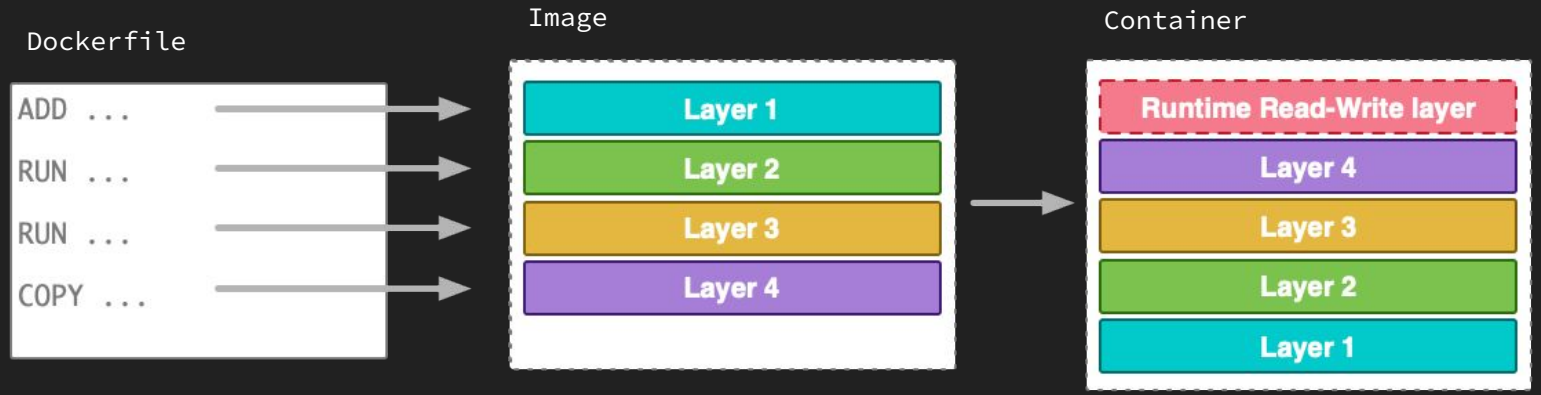
Docker e containers

- Si basa su **namespace** (isolamento) e **cgroups** (limitazione risorse) del kernel Linux.



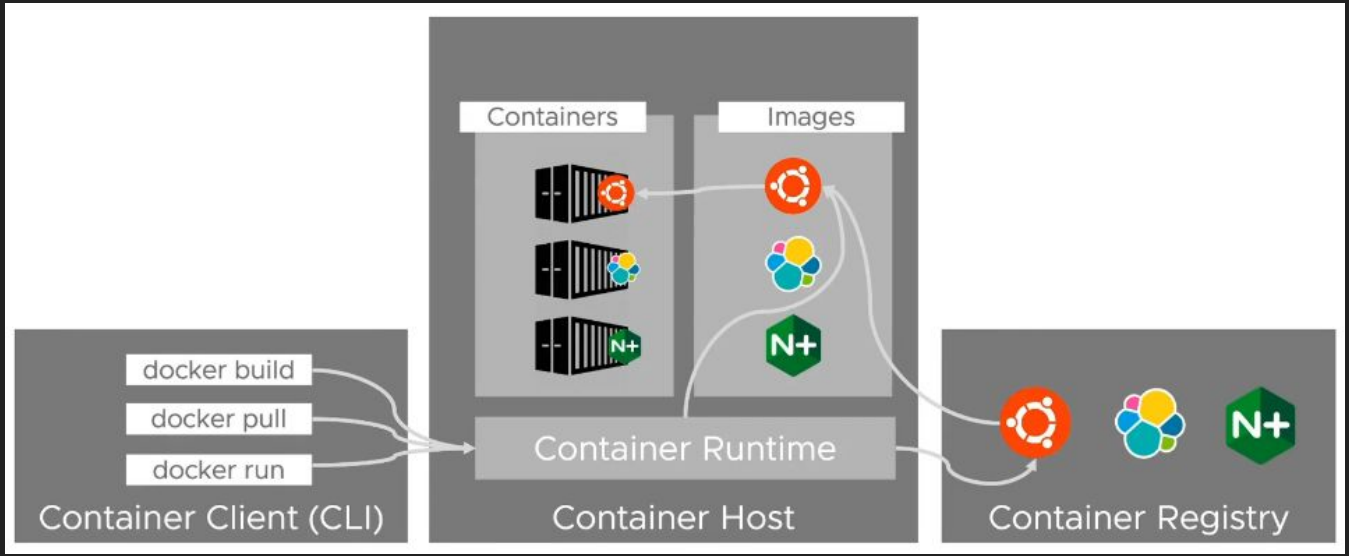
Docker e containers

- Un’immagine Docker è uno snapshot del filesystem del container (read-only), costruito con un **Dockerfile**.



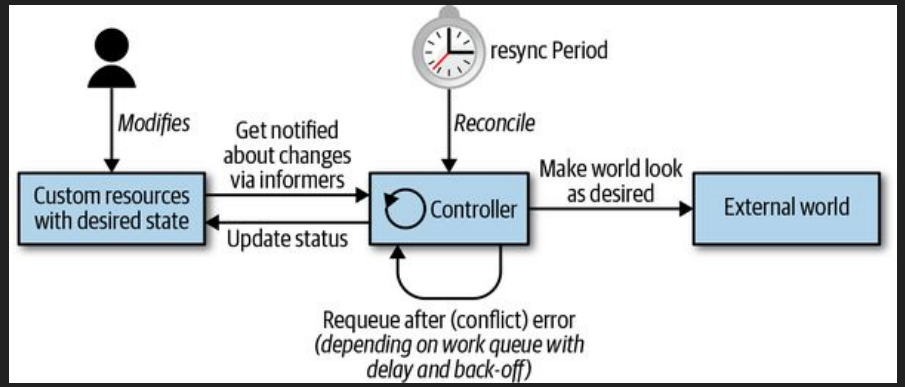
Docker e containers

- Per rendere fruibile una image, la si carica (**push**) su un **registry** (pubblico o privato).
- Per utilizzarla (**run**), la si preleva prima dal registry (**pull**).



Kubernetes: da singoli a cluster

- Quando i container crescono in numero, diventa difficile gestirli: serve **orchestrazione**.
- Kubernetes è una piattaforma che: gestisce scalabilità automatica (scale up/down), garantisce high availability, si occupa di networking e service discovery, automatizza il deployment e aggiornamenti.
- È dichiarativo: descrivi lo stato desiderato (via YAML) e Kubernetes si occupa di mantenerlo (**desired state**).



EKS

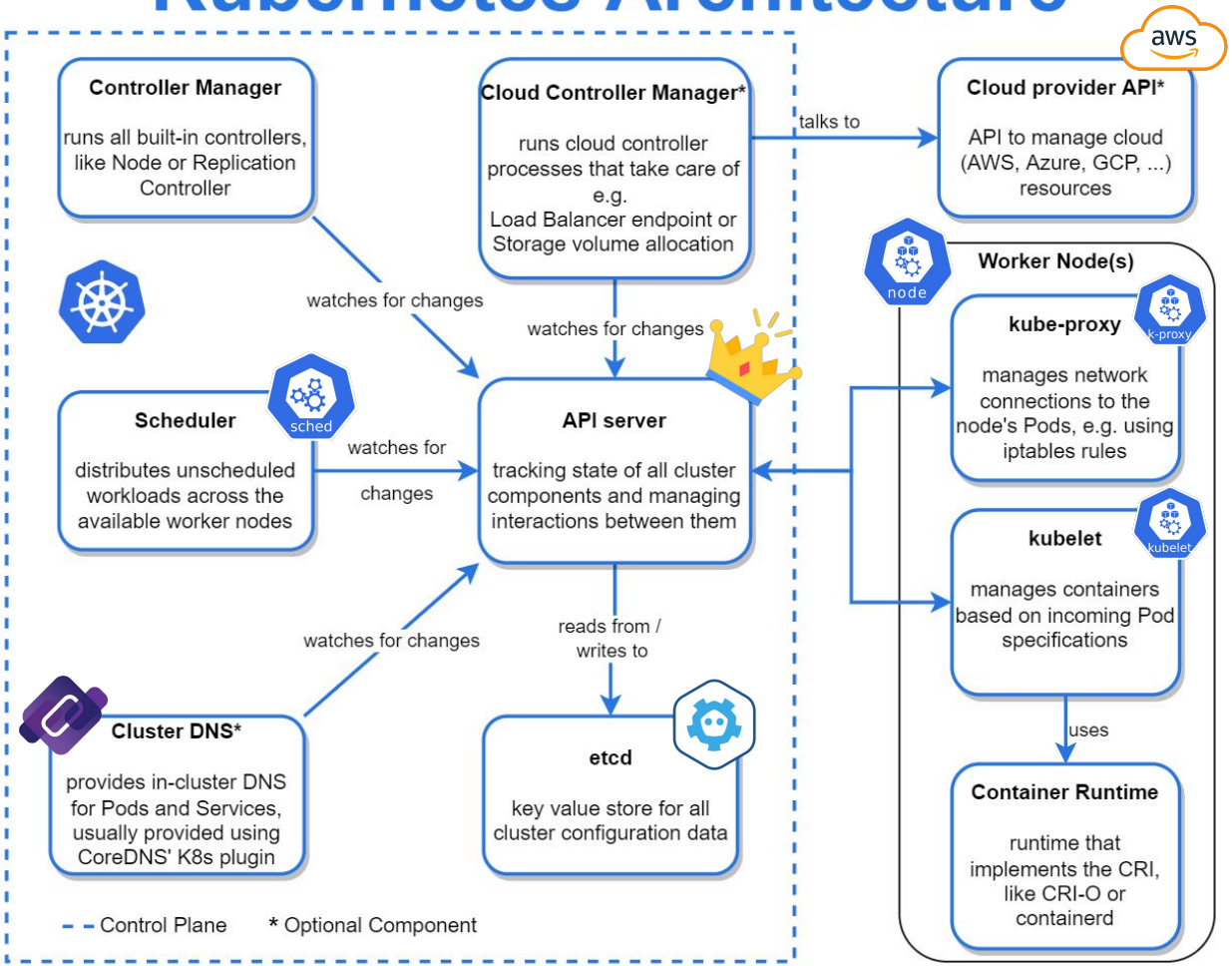
- **Amazon EKS** è il servizio gestito Kubernetes di AWS.

- **Gestione automatica del Control Plane**:
  - API server, scheduler, controller manager e etcd sono **hostati e mantenuti da AWS**.

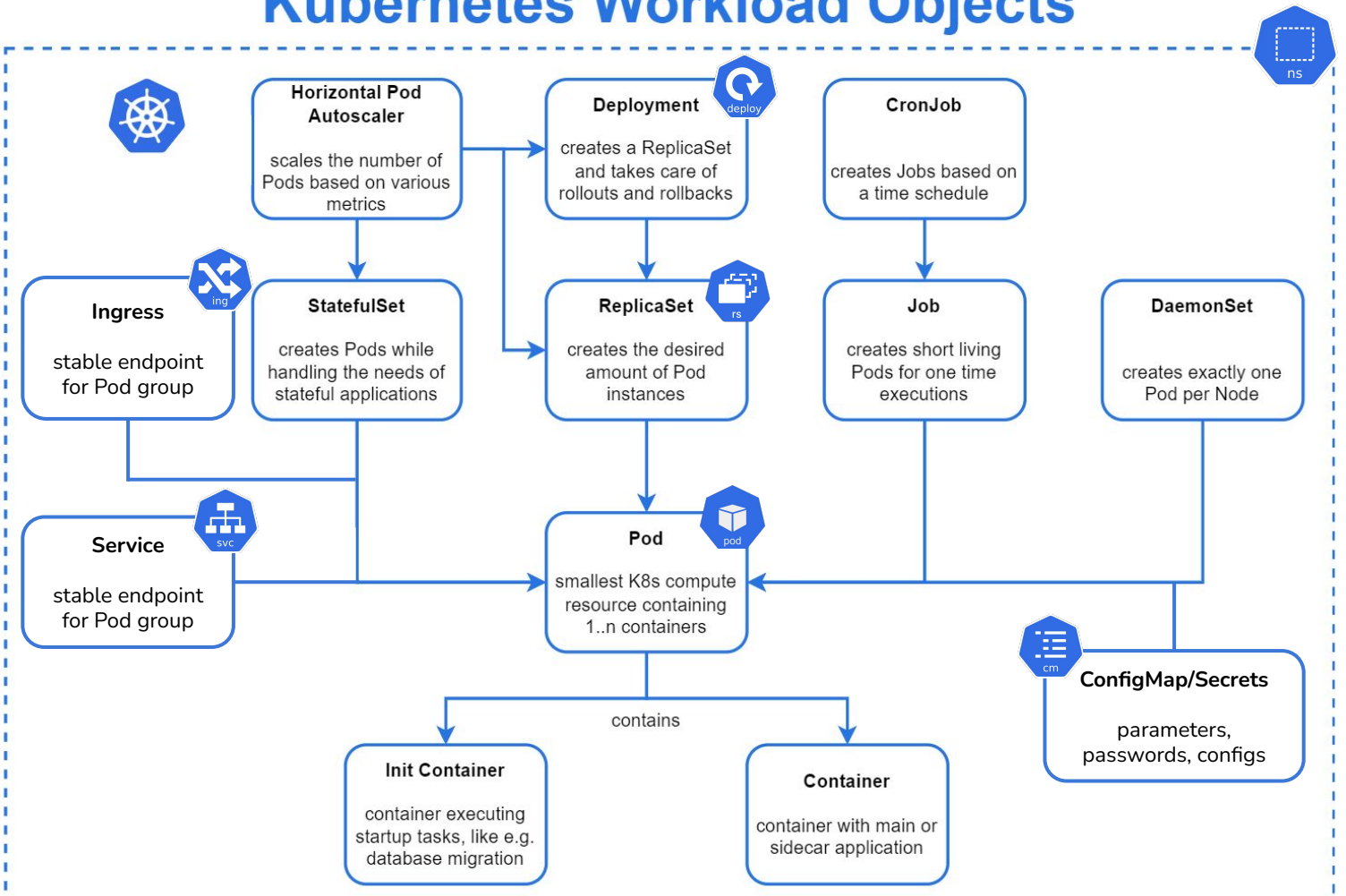
- I worker nodes possono essere:
  - **istanze EC2** gestite parzialmente o completamente da noi,
  - **fargate** (serverless, provisioning automatico),
  - o un mix dei due.

- EKS è **Kubernetes vanilla** → 100% compatibile con strumenti e manifest standard.

Kubernetes Architecture



Kubernetes Workload Objects



Kubernetes: componenti

- **Namespace:** componente di astrazione nel cluster per un raggruppamento logico di risorse

- **Pod**: unità minima di deploy in Kubernetes. Contiene uno o più container che:
  - condividono **IP**, **filesystem temporaneo** e **namespace**.
  - sono schedulati **insieme** sullo stesso nodo.

- **Deployment**
  - definisce **quanti Pod** devono esistere (repliche),
  - gestisce **rolling updates**, rollback,
  - monitora lo stato dei Pod e li **ricrea se falliscono**.

- Esempio: un web server con 3 repliche → Deployment crea e mantiene 3 Pod identici.

Kubernetes: componenti

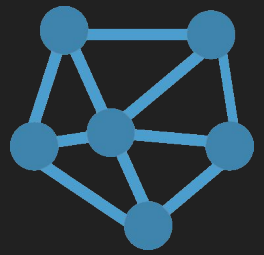
- **Service**
  - fornisce un **endpoint stabile** per un gruppo di Pod (che altrimenti avrebbero IP volatili)
  - agisce come un **load balancer** interno

- **Ingress**
  - gestisce il **traffico HTTP/HTTPS esterno**
  - si configura con regole tipo "host/path → service".
  - usa un Ingress Controller e crea i relativi load balancers nel clouder (es. AWS ALB/NLB).

- **ConfigMap**
  - contiene **configurazioni esterne** all'immagine (es. file .env, config YAML)
  - può essere montata come file o variabile d'ambiente nel container

- **Secret**
  - simili a ConfigMap, ma adatti a contenere dati confidenziali (ad es. token, password, keys, ecc.)

Kubernetes: networking



- Kubernetes adotta un **modello di rete "flat”:**
  - **ogni Pod ha un suo IP unico** (di solito IP privato)
  - tutti i **Pod possono comunicare tra loro**, **senza NAT**
  - **non serve port forwarding tra Pod** anche se sono su nodi diversi

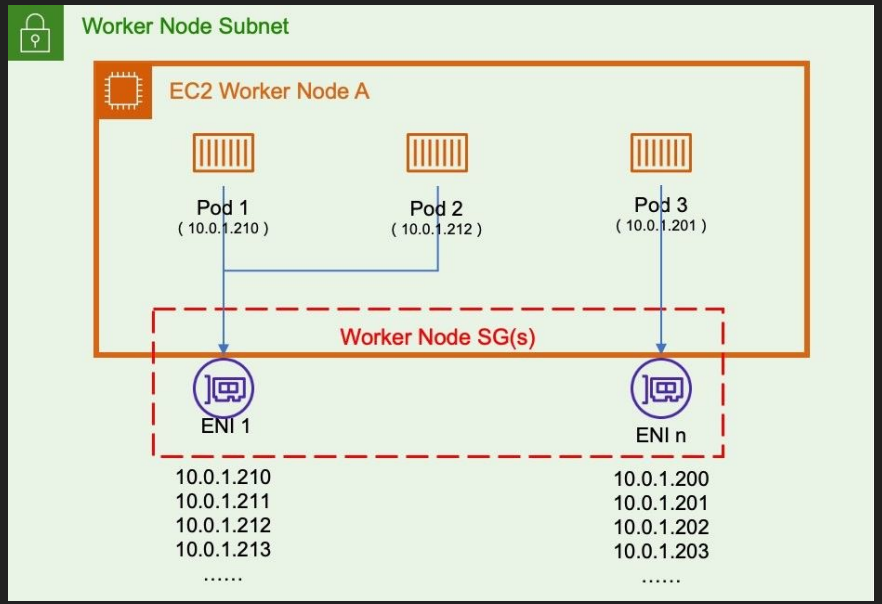
- **“Flat”** ha le seguenti implicazioni:
  - **unico spazio IP condiviso** tra tutti i Pod del cluster
  - se Pod A ha IP 10.244.1.5 e Pod B ha IP 10.244.2.8, possono parlarsi direttamente su quegli IP e porte
  - Kubernetes **non isola il traffico di default**, chiunque può parlare con chiunque

Kubernetes: networking

- Un **CNI (Container Network Interface)** è uno **plugin** usato da Kubernetes per collegare i Pod alla rete:
  - assegna un **IP** ad ogni Pod
  - configura le interfacce e il routing
  - gestisce il traffico di rete tra i Pod e verso l'esterno

- Kubernetes **non gestisce direttamente il networking dei Pod**: delega tutto al **plugin CNI**

- Le **NetworkPolicy** sono regole di sicurezza a livello di rete che controllano chi può comunicare con chi
  - enforcement fatto dai plugin CNI
  - diversi plugin per diversi tipi di filtraggio ingress/egress L4 e/o L7 (ad es. VPC CNI, Calico, Cilium, ecc.)



Kubernetes: approccio cloud native

Aspetto	Infrastruttura classica	Kubernetes/EKS
Provisioning	Manuale (SSH o scripts)	Dichiarativo (YAML)
Deployment	Manuale o scripts	Automatizzato (kubectl, Helm, ecc.)
Scaling	Limitato e non-standard	Auto-scaling configurabile
Fault tolerance	Failure = downtime*	Replica + self-healing
Aggiornamenti	Manuali e proni ad errori	Rolling updates
Risorse	Spesso over-provisioning	Ottimizzazione dinamica
Configurazione	Hardcoded nei servers	ConfigMap/Secrets dinamici

Esempio

/configmap.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: web-config
  namespace: default
data:
  WELCOME_MESSAGE: |
       "Hello from ConfigMap
       in Kubernetes!"
```

/deployment.yml (1)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25      # immagine del container
          ports:
            - containerPort: 80
          env:
            - name: WELCOME_MESSAGE
              valueFrom:
                configMapKeyRef:
                  name: web-config
                  key: WELCOME_MESSAGE
          resources:
          [...]
```

Definizione del Pod implicita

Esempio

/configmap.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: web-config
  namespace: default
data:
  WELCOME_MESSAGE: |
       "Hello from ConfigMap
       in Kubernetes!"
```

/deployment.yml (2)

```yaml
[...]
          resources:
            requests:
              cpu: "250m"          # garantiti 0.25 core
              memory: "128Mi"      # garantiti 128 MB
            limits:
              cpu: "500m"          # massimo 0.5 core
              memory: "256Mi"      # massimo 256 MB

          readinessProbe:          # per servire o no il traffico
            httpGet:
              path: /health
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 10
            failureThreshold: 3
          livenessProbe:
            httpGet:               # per decidere se killarlo
              path: /health
              port: 80
            initialDelaySeconds: 15
            periodSeconds: 20
            failureThreshold: 3
```

Definizione del Pod implicita

Esempio

/service.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: default
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

/ingress.yml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  namespace: default
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/load-balancer-name: ”custom-alb”
    alb.ingress.kubernetes.io/inbound-cidrs: "89.186.39.0/24"
    alb.ingress.kubernetes.io/healthcheck-path: ”/health”
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}]'
    alb.ingress.kubernetes.io/certificate-arn: "arn:aws:acm:xxxxxxxxxxx"
spec:
  rules:
    - host: sito.passstage.cloud
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80
```

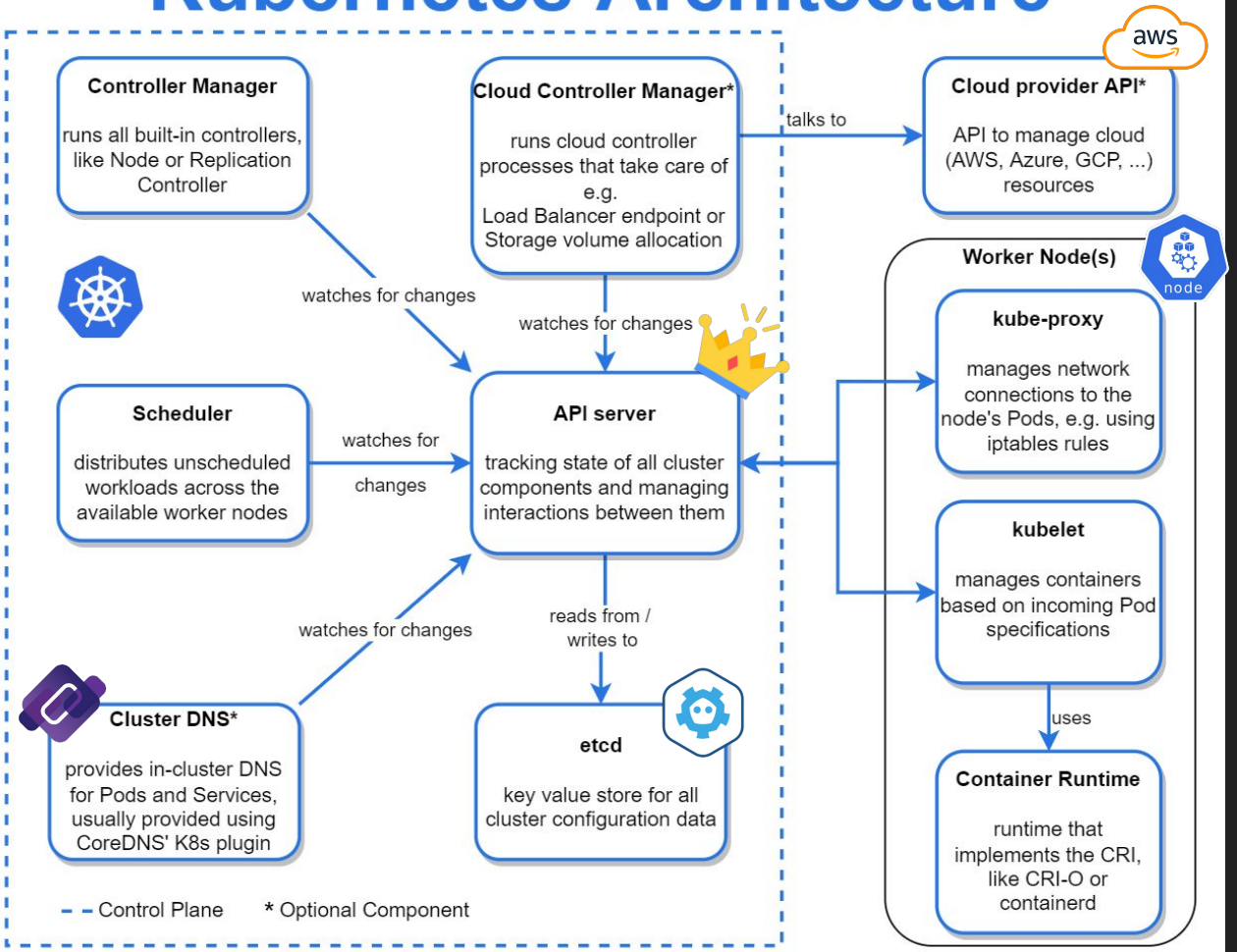
Interazione

- **Come facciamo a creare effettivamente gli oggetti dentro Kubernetes?**

- **Architettura API-based di Kubernetes**
  - **sistema "API-first"**: tutte le interazioni nel cluster passano attraverso l'API server (kube-apiserver)
  - il **kube-apiserver** è il cuore del controllo in Kubernetes: gestisce autenticazione, autorizzazione, validazione delle richieste e persistenza dello stato desiderato in **etcd**.
  - ogni operazione, come la creazione di un Pod, un Deployment o un Service, inizia con una **chiamata API REST**.

- Facciamo un recap...

Kubernetes Architecture



Kubernetes Architecture

Controller Manager

runs all built-in controllers, like Node or Replication Controller

Cloud Controller Manager*

runs cloud controller processes that take care of e.g. Load Balancer endpoint or Storage volume allocation

talks to

Cloud provider API*

API to manage cloud (AWS, Azure, GCP, ...) resources

watches for changes

watches for changes

Scheduler

distributes unscheduled workloads across the available worker nodes

watches for changes

API server

tracking state of all cluster components and managing interactions between them

send commands

watches for changes

reads from / writes to

Cluster DNS*

provides in-cluster DNS for Pods and Services, usually provided using CoreDNS' K8s plugin

etcd

key value store for all cluster configuration data

Control Plane

* Optional Component

Worker Node(s)

kube-proxy

manages network connections to the node's Pods, e.g. using iptables rules

kubelet

manages containers based on incoming Pod specifications

uses

Container Runtime

runtime that implements the CRI, like CRI-O or containerd

Interazione: tools

- **kubectl**: strumento da linea di comando che gli sviluppatori e gli amministratori usano per interagire con il cluster.
    - richiesta HTTP all'API server

- **Controller Manager e Scheduler**: Componenti di Kubernetes che agiscono autonomamente e periodicamente tramite l'API. Ad esempio, lo **Scheduler** assegna Pod ai nodi disponibili, mentre i **Controllers** monitorano lo stato del cluster e risolvono eventuali differenze.

- **Strumenti CI/CD (Gitlab CI, ArgoCD, etc.)**: Si connettono all'API di Kubernetes per fare deploy automatici di applicazioni, aggiornamenti di configurazioni, o scaling dinamico.

Interazione: esempio

Immaginiamo di voler creare un nuovo **Deployment** (ad esempio, l'app web con NGINX):

1. Comando **“kubectl apply -f deployment.yaml”**: applica le risorse contenute nel file specificato
2. **Richiesta HTTP al kube-apiserver**: kubectl invia una richiesta POST al kube-apiserver con il file YAML.
3. **Validazione e salvataggio**: Il kube-apiserver valida la richiesta e la salva in **etcd** (il database di stato di Kubernetes).
4. **Scheduler**: lo **Scheduler** decide su quale nodo eseguire i Pod e li distribuisce.
5. **Kubelet**: Ogni nodo ha un agente chiamato **kubelet** che riceve le istruzioni dal kube-apiserver per creare e gestire i Pod sui nodi.
6. **Controller**: Monitora costantemente lo stato del cluster. Se qualcosa non è come dovrebbe essere (ad esempio, un Pod non è in esecuzione), interviene e corregge lo stato.

Interazione: esempio

- **kubectl** è l'interfaccia di gestione più comune per Kubernetes.

- **Comandi comuni**:

```
$> kubectl get pods              # visualizza lo stato dei Pod nel cluster

$> kubectl logs <pod-name>       # visualizza i log di un Pod

$> kubectl describe <resource>   # fornisce informazioni dettagliate su risorse come Pod, Service, etc.

$> kubectl apply -f <file.yaml>  # crea o aggiorna una risorsa nel cluster
```

- wrappers: **k9s**, console EKS (read-only), ecc.

Autenticazione e autorizzazione

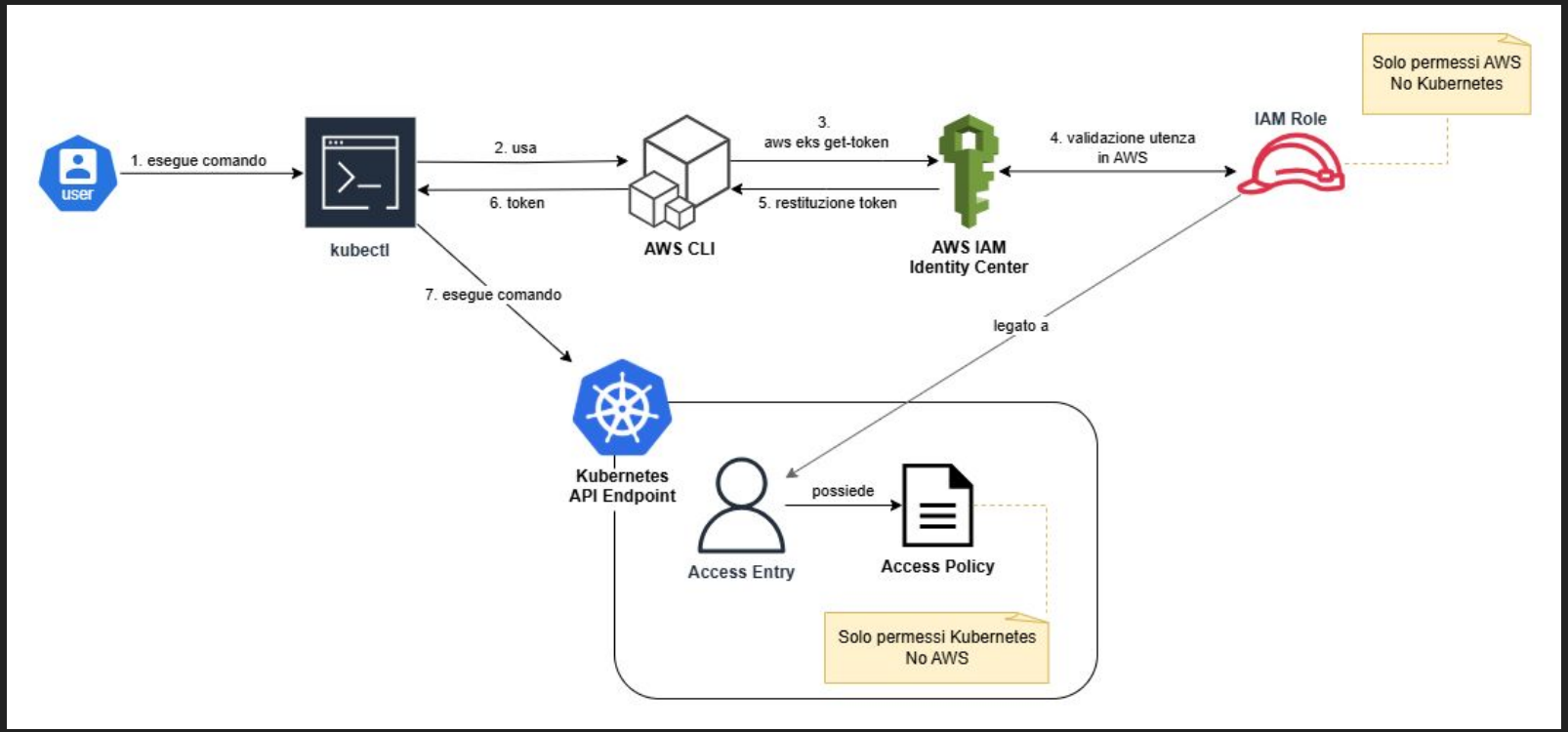
- Autenticazione e autorizzazione nei clusters EKS possono avvenire in due direzioni in base alla destinazione:
  - **inbound** (EKS API e IAM Access Entries): per tutti i comandi rivolti (tipicamente con kubectl) alle risorse **nel cluster**;
  - **outbound** (Pod Identities): per tutte le attività sulle API di AWS da workloads **interni al cluster**.

- Kubectl interagisce direttamente con l'API server del cluster e l'autenticazione avviene tramite un token generato da **AWS IAM**, legato all'utenza SSO per la quale si è autorizzati.

- Questo "passaggio" tra kubectl e IAM avviene mediante l'utilizzo di AWS CLI, il tool a riga comando di AWS.

- **IAM Roles**: Ogni utente o processo che interagisce con la EKS API, deve essere associato ad uno IAM Role con permessi specifici (es. lettura, scrittura, amministrazione del cluster). Quando un'entità esegue un comando con kubectl su una specifica risorsa, AWS IAM controlla se è autorizzato ad eseguirla.

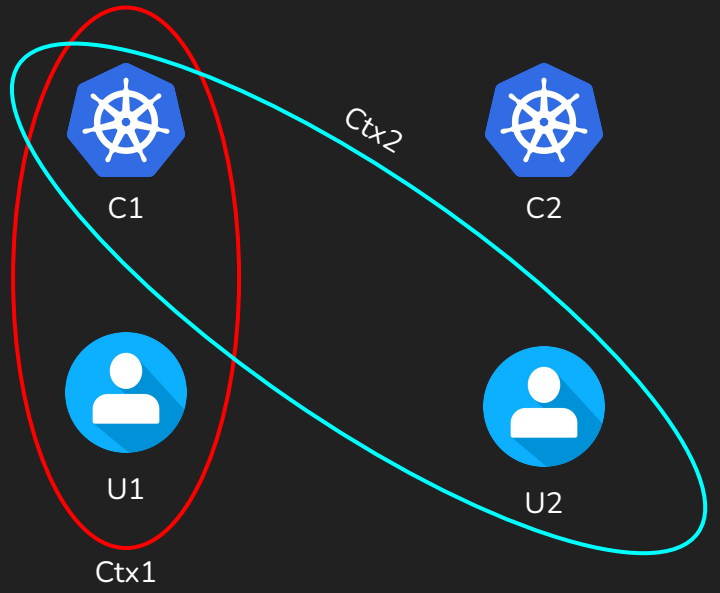
Autenticazione e autorizzazione kubectl



Configurazione kubectl

- Il tool kubectl utilizza un file di configurazione situato di default in **~/.kube/config**
- Suddiviso in sezioni specifiche per decidere quale utenza si collega a quale cluster

```yaml
apiVersion: v1
kind: Config

clusters:
  - cluster: C1
  - cluster: C2

users:
  - user: U1
  - user: U2

contexts:
  - context: Ctx1
  - context: Ctx2

current-context: Ctx1
```

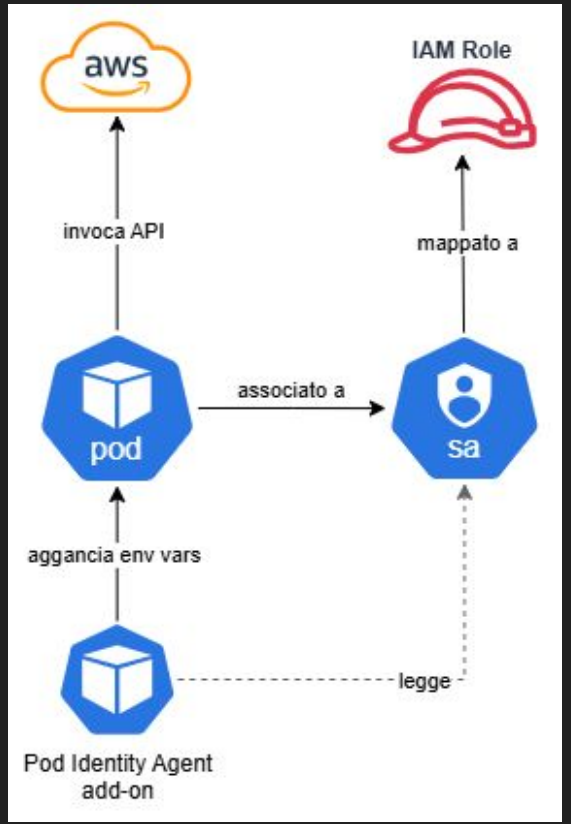


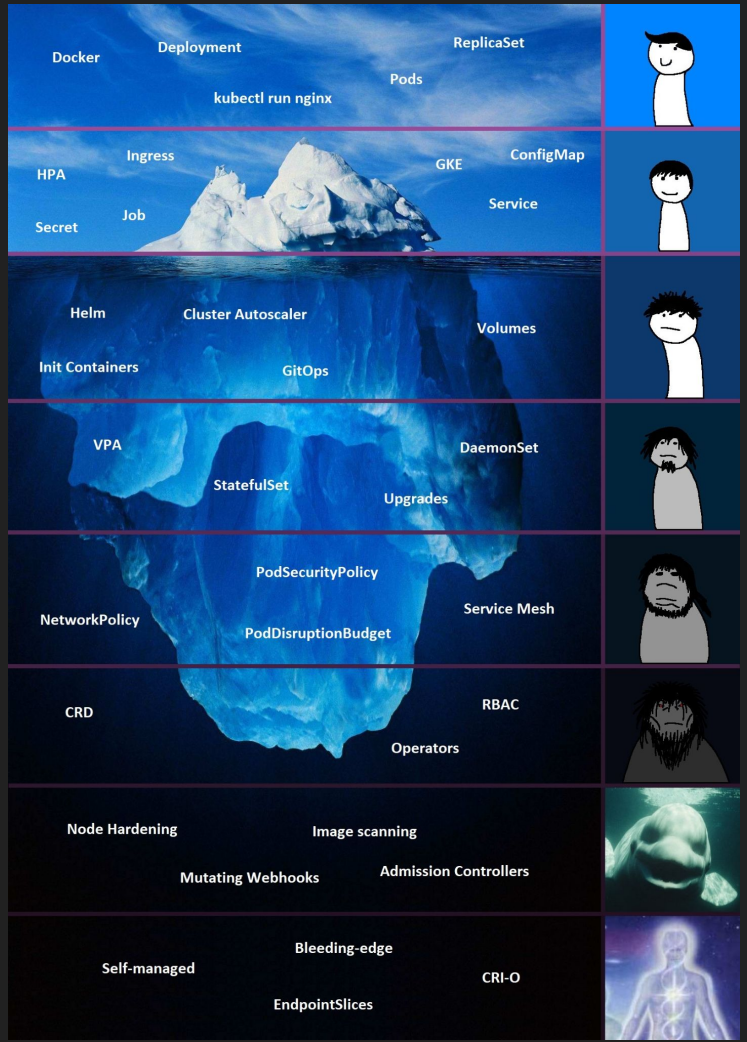
```yaml
apiVersion: v1
kind: Config

clusters:
  - cluster:
    certificate-authority-data: LS0tLS1...
    server: https://xxxxxxxx.yyy.eu-west-1.eks.amazonaws.com
  name: myAmazingCluster # C1

users:
  - name: myAmazingCluster/admin # U1
    user:
      exec:
        apiVersion: client.authentication.k8s.io/v1beta1
        args:
        - --region
        - eu-west-1
        - eks
        - --profile
        - stage
        - get-token
        - --cluster-name
        - myAmazingCluster
        command: aws

[...]
```

```yaml
[...]

  - name: myAmazingCluster/normal-user # U2
    user:
      exec:
        apiVersion: client.authentication.k8s.io/v1beta1
        args:
        - --region
        - eu-west-1
        - eks
        - --profile
        - dev
        - get-token
        - --cluster-name
        - myAmazingCluster
        command: aws

contexts:
  - context:
    name: myAmazingCluster/admin
      cluster: myAmazingCluster # C1
      namespace: kube-system
      user: myAmazingCluster/admin # U1

  - context:
    name: myAmazingClustere/normal-user
      cluster: myAmazingCluster # C1
      namespace: app-namespace
      user: myAmazingCluster/normal-user # U2

current-context: myAmazingCluster/admin
```

Pod identity

- Metodologia per consentire alle applicazioni di effettuare azioni sulle API di AWS **senza dover inserire credenziali hardcoded nel software**
- Sfrutta un addon di EKS chiamato **eks-pod-identity-agent** che espone una semplice proxy API sulla porta 80 del worker node su cui i Pods applicativi sono eseguiti (mediante CAP_NET_BIND_SERVICE), che accetta un “service account token” nel header Authorization e a sua volta invoca le API “AssumeRoleForPodIdentity”
- Il token e l’URL di questa API, vengono agganciati (inject) al Pod dell’applicativo mediante un “mutating webhook”
- Per capire a quali Pods effettuare la inject, viene usato come riferimento il **Service Account**, che deve essere quindi parte della definizione del Pod
- Ogni volta che dagli applicativi si effettuano chiamate alle API di AWS, i vari SDK scorrono la loro “credentials chain” e selezionano le prime credenziali disponibili, in questo caso, quelle fornite tramite injection




Docker
Deployment
ReplicaSet
Pods
kubectl run nginx
HPA
Ingress
GKE
ConfigMap
Secret
Job
Service
Helm
Cluster Autoscaler
Volumes
Init Containers
GitOps
VPA
DaemonSet
StatefulSet
Upgrades
PodSecurityPolicy
NetworkPolicy
Service Mesh
PodDisruptionBudget
CRD
RBAC
Operators
Node Hardening
Image scanning
Mutating Webhooks
Admission Controllers
Bleeding-edge
Self-managed
CRI-O
EndpointSlices

Fine parte 1

Risorse utili

- https://kubernetes.io/docs/home/
- https://www.eksworkshop.com/
- https://www.youtube.com/watch?v=2T86xAtR6Fo
- https://www.aquasec.com/blog/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016/#section-18