



A.D. 1308
unipg
UNIVERSITÀ DEGLI STUDI
DI PERUGIA

UNIVERSITÀ DEGLI STUDI DI PERUGIA

Dipartimento di Ingegneria

CORSO DI LAUREA MAGISTRALE
INGEGNERIA INFORMATICA E ROBOTICA
DATA SCIENCE AND DATA ENGINEERING

PROGETTO

Virtual Networks and Cloud Computing

-

Kubernetes 1.34
Features della nuova release sperimentate
in cluster locale

Dispensa di:
Daniele Angeloni

Professore:
Prof. Reali

Anno Accademico 2024/2025

Indice

1	Introduzione	1
1.1	Obiettivi	1
1.2	Prerequisiti	2
2	Creazione e configurazione del cluster	4
2.1	Architettura delle macchine virtuali	4
2.2	Installazione dei pacchetti Kubernetes	5
2.3	Configurazioni di sistema	6
2.4	Inizializzazione del cluster	7
2.5	Installazione del plugin di rete: Calico	7
2.6	Aggiunta del nodo worker	7
2.7	Verifica del cluster	8
3	Dimostrazione delle nuove funzionalità v1.34	9
3.1	Verifica cluster con Nginx e BusyBox	9
3.2	KYAML Output	11
3.3	Job con PodReplacementPolicy	13
3.4	ContainerRestartRules	16
3.5	Service Topology (PreferSameNode / PreferSameZone)	17
3.6	FQDN personalizzati per Pod	18
3.7	Horizontal Pod Autoscaler con nuove tolleranze	20
3.8	OpenTelemetry + Jaeger (Tracing App)	24
3.9	Mutating Admission Policy (CEL)	28
	Policy e binding	28
3.10	Image Volume (OCI artifact come volume)	30
	Pod con Image Volume (nuova funzionalità)	31
	Variante fallback con initContainer	31
3.11	Dynamic Resource Allocation (DRA)	33
4	Conclusioni e sviluppi futuri	35
	Riferimenti	37

Capitolo 1

Introduzione

Questo progetto documenta la creazione e gestione di un cluster **Kubernetes v1.34** e la dimostrazione pratica delle principali novità introdotte nell'ultima release del 27 agosto 2025.

Il cluster è stato realizzato su due macchine virtuali **Lubuntu 25.04**:

- `node1` – Master
- `node2` – Worker

Per l'installazione del cluster Kubernetes v1.34 è stato scelto di utilizzare `kubeadm`. Il tool `kubespray`, già visto a lezione, non supportava ancora ufficialmente la nuova release, causando errori nella fase di download dei pacchetti di sistema. `kubeadm`, in quanto distribuzione ufficiale di Kubernetes, ha invece garantito la compatibilità immediata con la versione 1.34. .

1.1 Obiettivi

Gli obiettivi principali del progetto sono stati:

- Installare e configurare un cluster Kubernetes v1.34 in ambiente locale.
- Mostrare con esempi pratici le nuove funzionalità introdotte nella release, tra cui:
 - KYAML in `kubect1`: nuovo formato di output alternativo a YAML, più compatto e leggibile.
 - Job con `PodReplacementPolicy`: consente di decidere se rimpiazzare solo i pod falliti o anche quelli terminati manualmente.
 - `ContainerRestartRules`: permette di definire regole di riavvio specifiche per singolo container, basate sugli exit code.
 - Service Topology: introduce politiche di routing del traffico con preferenza per pod sullo stesso nodo o nella stessa zona.

- FQDN personalizzati per Pod: possibilità di assegnare hostname e subdomain ai pod, ottenendo nomi DNS più leggibili.
- Horizontal Pod Autoscaler (HPA) con nuove tolleranze: scale-up più rapido e scale-down più prudente, riducendo oscillazioni.
- OpenTelemetry + Jaeger: supporto al tracing distribuito, con possibilità di raccogliere e visualizzare tracce di esecuzione.
- Mutating Admission Policy (CEL): consente di applicare regole di mutazione sugli oggetti Kubernetes senza webhook esterni.
- Image Volume: introduce i volumi basati su immagini OCI, montabili direttamente nei pod.
- Dynamic Resource Allocation (DRA): framework per la gestione dinamica di risorse speciali (GPU, FPGA, dispositivi custom).

1.2 Prerequisiti

Il progetto è stato eseguito sul seguente ambiente:

Macchine virtuali

- 2 VM con **Lubuntu 25.04**
- node1 (master) con IP 192.168.43.10
- node2 (worker) con IP 192.168.43.11
- Rete VirtualBox impostata come **NAT interna**

Pacchetti di sistema

- openssh-server (per accesso remoto e chiavi SSH)
- python3-pip

Kubernetes 1.34

Aggiunta del repository ufficiale e installazione dei pacchetti:

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/Release.key \
| sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-apt-keyring.gpg

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /" \
| sudo tee /etc/apt/sources.list.d/kubernetes.list
```

```
sudo apt install -y kubelet kubeadm kubectl  
sudo apt-mark hold kubelet kubeadm kubectl
```

Container runtime

- containerd come runtime predefinito.
- Configurato con `containerd config default` in `/etc/containerd/config.toml`.

Impostazioni di sistema

- Disabilitazione dello **swap**:

```
sudo swapoff -a  
sudo sed -i '/ swap / s/^/#/' /etc/fstab
```

- Abilitazione **IP forwarding**:

```
echo "net.ipv4.ip_forward=1" | sudo tee -a /etc/sysctl.conf  
sudo sysctl -p
```

Versioni utilizzate

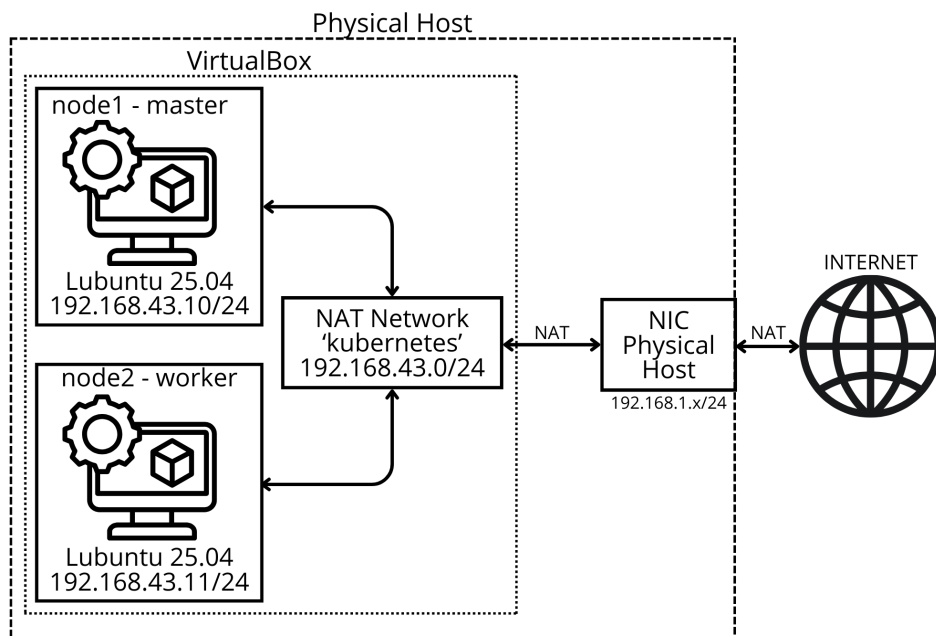
- Kubernetes (API Server): v1.34.0
- Kubeadm: v1.34.0
- Kubelet: v1.34.0
- Kubectl: v1.34.0
- Container runtime: containerd 2.0.5
- Etcd: v3.6.4

Capitolo 2

Creazione e configurazione del cluster

Come già introdotto, poiché Kubespray non supporta completamente ancora la versione 1.34, è stato utilizzato `kubeadm` per l'installazione e la configurazione del cluster

2.1 Architettura delle macchine virtuali



Sono state predisposte due VM con **Lubuntu 24.04**, configurate su una **NAT Network** per consentire la connessione tra di loro e verso internet, con indirizzi statici:

- node1 (master): 192.168.43.10
- node2 (worker): 192.168.43.11

Configurazione della rete

In ciascun nodo è stato rimosso il file '50-cloud-init.yaml' e modificato il file '01-network-manager-all.yaml':

```
sudo rm /etc/netplan/50-cloud-init.yaml
sudo nano /etc/netplan/01-network-manager-all.yaml
```

Esempio di configurazione su node1:

```
network:
  version: 2
  renderer: NetworkManager
  ethernets:
    enp0s3:
      dhcp4: no
      addresses: [192.168.43.10/24]
      routes:
        - to: default
          via: 192.168.43.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
```

Applicazione:

```
sudo netplan apply
```

Configurazione hostnames

Nei file /etc/hosts di entrambi i nodi:

```
127.0.0.1 localhost
192.168.43.10 node1 node1.example.com
192.168.43.11 node2 node2.example.com
```

Nei file /etc/hostname:

- node1 → node1
- node2 → node2

2.2 Installazione dei pacchetti Kubernetes

Su entrambi i nodi:

```
sudo apt update
sudo apt install -y apt-transport-https ca-certificates curl gpg
```

Aggiunta del repository ufficiale:

```
sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/Release.key \
| sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-apt-keyring.gpg

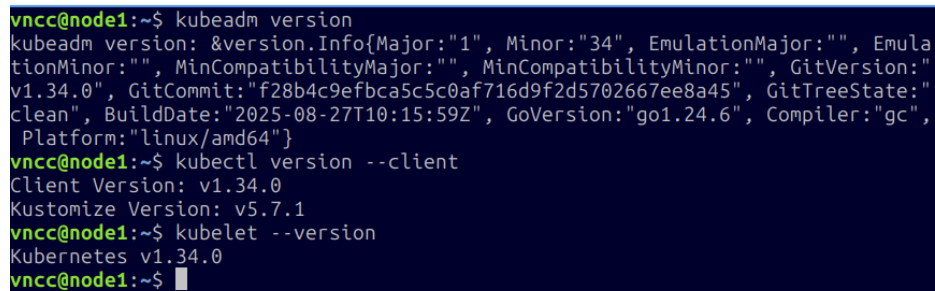
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /" \
| sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Installazione dei componenti:

```
sudo apt update
sudo apt install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

Verifica versione:

```
kubeadm version
kubectl version --client
kubelet --version
```



```
vncc@node1:~$ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"34", EmulationMajor:"", EmulationMinor:"", MinCompatibilityMajor:"", MinCompatibilityMinor:"", GitVersion:"v1.34.0", GitCommit:"f28b4c9efbca5c5c0af716d9f2d5702667ee8a45", GitTreeState:"clean", BuildDate:"2025-08-27T10:15:59Z", GoVersion:"go1.24.6", Compiler:"gc", Platform:"linux/amd64"}
vncc@node1:~$ kubectl version --client
Client Version: v1.34.0
Kustomize Version: v5.7.1
vncc@node1:~$ kubelet --version
Kubernetes v1.34.0
vncc@node1:~$
```

2.3 Configurazioni di sistema

Container Runtime: Containerd

Configurazione di Containerd come Container Runtime:

```
sudo apt install -y containerd
sudo mkdir -p /etc/containerd
containerd config default | sudo tee /etc/containerd/config.toml
sudo systemctl restart containerd
sudo systemctl enable containerd
```

Disabilitazione dello swap


```
sudo swapoff -a
sudo sed -i 's/^/#/' /etc/fstab
free -h    # verifica che la riga "Swap" sia a 0
```

Abilitazione IP forwarding

```
echo "net.ipv4.ip_forward=1" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
cat /proc/sys/net/ipv4/ip_forward    # deve stampare 1
```

2.4 Inizializzazione del cluster

Sul master node1:

```
sudo kubeadm init \
  --kubernetes-version=v1.34.0 \
  --pod-network-cidr=10.233.64.0/18 \
  --service-cidr=10.233.0.0/18
```

Configurazione di kubectl per l'utente:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

```
kubectl get nodes
```

Al termine, node1 appare in stato NotReady poiché manca ancora il plugin di rete.

2.5 Installazione del plugin di rete: Calico

Configurazione di Calico come plugin:

```
kubectl apply -f \
https://raw.githubusercontent.com/projectcalico/calico/v3.27.3/manifests/calico.yaml
```

```
kubectl get pods -n kube-system
```

Si attende lo stato Running per i pod calico-node e calico-kube-controllers.

2.6 Aggiunta del nodo worker

Sul node2 (worker) eseguire il comando di join con il token fornito da kubeadm init nel master, tramite:

```
sudo kubeadm join 192.168.43.10:6443 --token <token> \
  --discovery-token-ca-cert-hash sha256:<hash>
```

2.7 Verifica del cluster

Dal master:

```
kubectl get nodes  
kubectl get pods -A
```

```
vncc@node1:~$ kubectl get nodes -o wide  
NAME          STATUS    ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION   CONTAINER-RUNTIME  
node1         Ready     control-plane   15h   v1.34.0   192.168.43.10   <none>        Ubuntu Plucky Puffin (development branch) 6.12.0-15-generic containerd://2.0.5  
node2         Ready     <none>        15h   v1.34.0   192.168.43.11   <none>        Ubuntu Plucky Puffin (development branch) 6.12.0-15-generic containerd://2.0.5  
vncc@node1:~$
```

- Entrambi i nodi (node1, node2) in stato Ready e tutti i pod di sistema (coredns, calico-*, kube-proxy, ecc.) in esecuzione.

Capitolo 3

Dimostrazione delle nuove funzionalità v1.34

In questo capitolo vengono presentate le principali funzionalità introdotte con Kubernetes v1.34. Per ciascuna feature è stato predisposto un esempio pratico tramite manifest YAML e comandi per la dimostrazione. Le funzionalità già stabili o comunque che non portano ad instabilità sono state testate direttamente nel cluster locale, mentre per alcune ancora in fase *alpha* è stato realizzato un approccio dimostrativo, riportando comunque manifest ed esempi teorici dato che la configurazione dei relativi feature gate, portava appunto ad instabilità del cluster.

3.1 Verifica cluster con Nginx e BusyBox

Come primo test di validazione del cluster è stato realizzato un semplice deployment di Nginx, utilizzato come “hello world” per verificare che il sistema sia in grado di gestire correttamente i workload. L’applicazione è stata resa accessibile tramite un servizio di tipo ClusterIP, e la connettività è stata verificata da un pod BusyBox avviato all’interno del cluster.

Manifest YAML

Il file `manifests/test-cluster-nginx.yaml` contiene la definizione del deployment e del service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-cluster-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
    app: test-cluster-nginx
  template:
    metadata:
      labels:
        app: test-cluster-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: test-cluster-nginx
spec:
  selector:
    app: test-cluster-nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

Applicazione del manifest

```
kubectl apply -f manifests/test-cluster-nginx.yaml
```

Verifica del deployment e del service

```
kubectl get deploy test-cluster-nginx
kubectl get pods -l app=test-cluster-nginx -o wide
kubectl get svc test-cluster-nginx
```

Test interno con BusyBox

Per validare la raggiungibilità del servizio dall'interno del cluster:

```
kubectl run -it busybox --image=busybox --restart=Never -- sh
wget -q0- http://test-cluster-nginx
```

L'output atteso è la pagina HTML di default servita da Nginx.

```
vnc@node1:~$ kubectl run -it busybox --image=busybox --restart=Never -- sh
All commands and output from this session will be recorded in container logs,
including credentials and sensitive information passed through the command pro
mpt.
If you don't see a command prompt, try pressing enter.
/ # wget -qO- http://test-cluster-nginx
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ #
```

3.2 KYAML Output

Una delle novità introdotte in Kubernetes v1.34 è il nuovo formato di output KYAML, attualmente in fase *alpha*. Si tratta di un'alternativa all'output tradizionale YAML, pensata per fornire una rappresentazione più compatta e leggibile delle risorse, priva di campi ridondanti o difficili da interpretare.

Creazione del Pod di esempio

Il file `manifests/kyaml-example.yaml` definisce un semplice Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: kyaml-example
  labels:
    app: kyaml-example
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["sh", "-c", "echo KYAML Example && sleep 3600"]
```

Applicazione del manifest:

```
kubectl apply -f manifests/kyaml-example.yaml
```

Abilitazione KYAML nel client

Poiché la funzionalità è in *alpha*, va abilitata manualmente:

```
export KUBECTL_KYAML=true
```

Questa variabile deve essere impostata a ogni sessione, oppure inserita nel file `~/.bashrc`.

Generazione degli output

Per ottenere i due formati di output:

```
kubectl get pod kyaml-example -o yaml  
kubectl get pod kyaml-example -o kyaml
```

Confronto YAML vs KYAML

Estratto in formato YAML tradizionale:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: kyaml-example  
  namespace: default  
  labels:  
    app: kyaml-example  
  annotations:  
    kubectl.kubernetes.io/last-applied-configuration: |  
      {"apiVersion":"v1","kind":"Pod", ... }  
spec:  
  containers:  
  - name: busybox  
    image: busybox  
    command: ["sh","-c","echo KYAML Example && sleep 3600"]  
status:  
  phase: Running  
  podIP: 10.233.75.12  
  ...
```

Estratto in formato KYAML:

```
---  
{  
  apiVersion: "v1",  
  kind: "Pod",  
  metadata: {  
    name: "kyaml-example",  
    namespace: "default",  
    labels: { app: "kyaml-example" }  
  }  
}
```

```
  },
  spec: {
    containers: [{
      name: "busybox",
      image: "busybox",
      command: ["sh", "-c", "echo KYAML Example && sleep 3600"]
    }]
  },
  status: { phase: "Running", podIP: "10.233.75.12" }
}
```

Osservazioni

- Con `-o yaml` l'output include numerosi campi aggiuntivi (`managedFields`, condizioni dettagliate, eventi), risultando piuttosto verboso.
- Con `-o kyaml` l'output è molto più compatto e leggibile: mantiene le informazioni essenziali e utilizza una sintassi coerente in stile oggetto.

Questa differenza migliora la leggibilità e rende più semplice l'analisi delle risorse direttamente da riga di comando.

3.3 Job con PodReplacementPolicy

Con Kubernetes v1.34 è stato introdotto il campo `podReplacementPolicy` nei Job. Questa novità permette di controllare con maggiore precisione quando un Pod deve essere rimpiazzato:

- Nelle versioni precedenti i Pod venivano sostituiti solo in caso di errore, senza possibilità di distinguere altri scenari.
- Da v1.34 sono disponibili due modalità:
 - **Failed** → rimpiazza solo i Pod che terminano con errore.
 - **TerminatingOrFailed** → rimpiazza sia i Pod falliti sia quelli terminati manualmente o da eventi esterni.

Caso 1 – Job con policy Failed

Definizione del file `manifests/job-failed.yaml`:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-failed
spec:
```

```

completions: 1
podReplacementPolicy: Failed
template:
  spec:
    restartPolicy: Never
    containers:
    - name: demo
      image: busybox
      command: ["sh", "-c", "echo Pod running && sleep 30 && exit
1"]

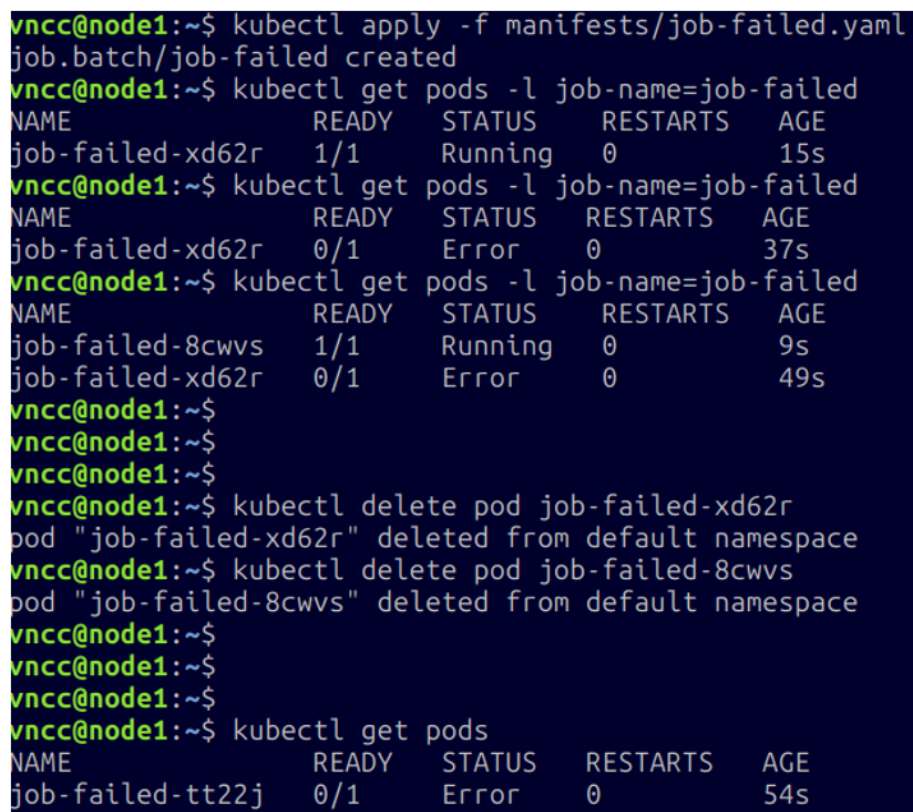
```

Applicazione ed esecuzione:

```

kubectl apply -f manifests/job-failed.yaml
kubectl get pods -l job-name=job-failed
# dopo l'uscita con codice 1 il Job ricrea un nuovo Pod
kubectl delete pod <nome-pod>

```



```

vncc@node1:~$ kubectl apply -f manifests/job-failed.yaml
job.batch/job-failed created
vncc@node1:~$ kubectl get pods -l job-name=job-failed
NAME                READY   STATUS    RESTARTS   AGE
job-failed-xd62r    1/1     Running   0           15s
vncc@node1:~$ kubectl get pods -l job-name=job-failed
NAME                READY   STATUS    RESTARTS   AGE
job-failed-xd62r    0/1     Error     0           37s
vncc@node1:~$ kubectl get pods -l job-name=job-failed
NAME                READY   STATUS    RESTARTS   AGE
job-failed-8cwvs    1/1     Running   0           9s
job-failed-xd62r    0/1     Error     0           49s
vncc@node1:~$
vncc@node1:~$
vncc@node1:~$
vncc@node1:~$ kubectl delete pod job-failed-xd62r
pod "job-failed-xd62r" deleted from default namespace
vncc@node1:~$ kubectl delete pod job-failed-8cwvs
pod "job-failed-8cwvs" deleted from default namespace
vncc@node1:~$
vncc@node1:~$
vncc@node1:~$
vncc@node1:~$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
job-failed-tt22j    0/1     Error     0           54s

```

Con la policy Failed, un Pod cancellato manualmente non viene rimpiazzato.

Caso 2 – Job con policy TerminatingOrFailed

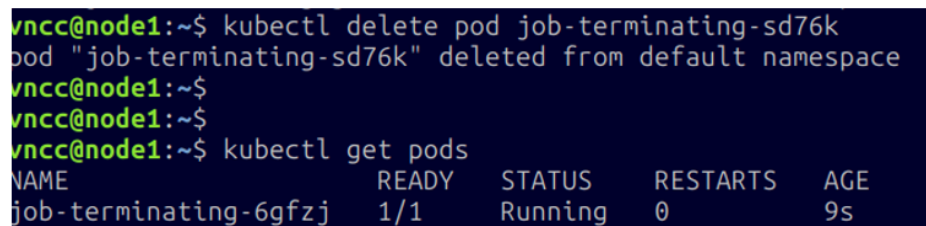
Definizione del file manifests/job-terminating.yaml:


```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-terminating
spec:
  completions: 1
  podReplacementPolicy: TerminatingOrFailed
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: demo
        image: busybox
        command: ["sh", "-c", "echo Pod running && sleep 30 && exit
1"]
```

Applicazione ed esecuzione:

```
kubectl apply -f manifests/job-terminating.yaml
kubectl get pods -l job-name=job-terminating
kubectl delete pod <nome-pod>    # eliminazione manuale di un Pod in
    esecuzione
```

Con la policy `TerminatingOrFailed`, il Job ricrea immediatamente un nuovo Pod al posto di quello terminato manualmente.



```
vncc@node1:~$ kubectl delete pod job-terminating-sd76k
pod "job-terminating-sd76k" deleted from default namespace
vncc@node1:~$
vncc@node1:~$
vncc@node1:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
job-terminating-6gfzj               1/1     Running   0           9s
```

Osservazioni

- Con **Failed** vengono rimpiazzati solo i Pod che terminano con errore (exit code diverso da 0).
- Con **TerminatingOrFailed** vengono rimpiazzati sia i Pod falliti che quelli terminati manualmente.

Questa distinzione non era disponibile nelle versioni precedenti e consente ora un controllo più fine sul comportamento dei Job. I Pod falliti o terminati manualmente restano visibili nello stato `Error` o `Completed`, ma grazie alla nuova opzione è il Job a decidere se generarne di nuovi o meno.

3.4 ContainerRestartRules

Fino a Kubernetes v1.33, i riavvii dei container erano controllati esclusivamente dal campo `restartPolicy` a livello di Pod (`Always`, `OnFailure`, `Never`). Tutti i container nello stesso Pod erano obbligati a seguire la stessa regola di riavvio.

Con Kubernetes v1.34 (ancora in fase *alpha*) è stata introdotta la funzionalità **ContainerRestartRules**, che permette di definire regole di riavvio specifiche per ciascun container, basate sul codice di uscita.

Questo rende possibile scenari più flessibili, ad esempio:

- un container che si riavvia anche se termina con successo (`exit 0`);
- un altro container che si riavvia solo se fallisce (`exit 1`).

Abilitazione

La funzionalità è sperimentale e richiede l'abilitazione del feature gate `ContainerRestartPolicy` su:

- `kube-apiserver` (control plane)
- `kubelet` (su tutti i nodi)

Durante la configurazione è stato tentato di abilitare il flag, ma l'avvio del control plane non è risultato stabile. Per questo motivo la funzionalità viene presentata qui solo come **dimostrazione teorica**, tramite un manifest di esempio.

Manifest di esempio

Definizione del file `manifests/container-restart-rules.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: restart-rules-example
spec:
  restartPolicy: Never
  containers:
  - name: container-ok
    image: alpine
    command: ["sh", "-c", "echo Uscita con 0 && exit 0"]
    restartPolicy:
      rules:
      - onExitCodes:
          containerName: container-ok
          operator: In
          values: [0]
```

```
        action: Restart
- name: container-ko
  image: alpine
  command: ["sh", "-c", "echo Uscita con 1 && exit 1"]
  restartPolicy:
    rules:
      - onExitCodes:
          containerName: container-ko
          operator: In
          values: [1]
          action: Restart
```

Conclusione

- Nelle versioni precedenti, con `restartPolicy: Never`, nessun container sarebbe stato riavviato.
- Con Kubernetes v1.34, grazie a `ContainerRestartRules`, è possibile definire regole di riavvio personalizzate per ogni container in base agli exit code.
- Il manifest mostra come cambia la gestione dei riavvii e rende evidente la maggiore flessibilità offerta da questa funzionalità.

3.5 Service Topology (PreferSameNode / PreferSameZone)

Nelle versioni precedenti, i Service distribuivano il traffico in modo uniforme verso tutti i Pod, senza alcuna preferenza topologica. Con Kubernetes v1.34 (ancora in fase *alpha*) sono state introdotte due nuove opzioni per il campo `internalTrafficPolicy`:

- **PreferSameNode** → preferisce i Pod sullo stesso nodo.
- **PreferSameZone** → preferisce i Pod nella stessa zona.

Questa funzionalità mira a migliorare la latenza e a ridurre il traffico di rete cross-node e cross-zone.

Tentativo pratico

È stato provato a creare un Service con la direttiva:

```
internalTrafficPolicy: PreferSameNode
```

Tuttavia l'API server ha restituito l'errore:

The Service "nginx-topology-svc" is invalid:
spec.internalTrafficPolicy: Unsupported value: "PreferSameNode":
supported values: "Cluster", "Local"

Nel cluster erano quindi accettati solo i valori già stabili (`Cluster`, `Local`). Per utilizzare le nuove opzioni introdotte in v1.34 è necessario abilitare i feature gate relativi, ma nonostante i tentativi la scelta non è stata adottata per mantenere la stabilità dell'ambiente.

Manifest di esempio (teorico)

Definizione del file manifests/service-topology.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-topology-svc
  namespace: default
spec:
  selector:
    app: nginx-topology
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  internalTrafficPolicy: PreferSameNode
```

Conclusione

- Prima di v1.34 erano supportati solo i valori `Cluster` e `Local`.
- Con Kubernetes v1.34 (alpha) sono disponibili anche `PreferSameNode` e `PreferSameZone`, per un instradamento più intelligente del traffico.

3.6 FQDN personalizzati per Pod

Un **FQDN** (Fully Qualified Domain Name) è un nome di dominio completo, ovvero l'indirizzo DNS che identifica in maniera univoca una risorsa all'interno della gerarchia del DNS.

- Prima di Kubernetes v1.34 i Pod ricevevano nomi DNS generati automaticamente, ad esempio:
 - `<pod-name>.<namespace>.pod.cluster.local`

L'hostname del Pod non era quindi personalizzabile e i nomi risultavano poco leggibili.

- Da Kubernetes v1.34 è possibile specificare direttamente i campi `hostname` e `subdomain` nel Pod, ottenendo un FQDN personalizzato e prevedibile. Questo rende più semplice integrare applicazioni legacy o sistemi che richiedono host fissi.

Manifest di esempio

Definizione del file `manifests/fqdn-example.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-vncc
  namespace: default
  labels:
    app: pod-vncc
spec:
  hostname: test-vncc
  subdomain: custom-vncc
  containers:
  - name: nginx
    image: nginx:latest
---
apiVersion: v1
kind: Service
metadata:
  name: custom-vncc
  namespace: default
spec:
  selector:
    app: pod-vncc
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  clusterIP: None    # headless service
```

Spiegazione

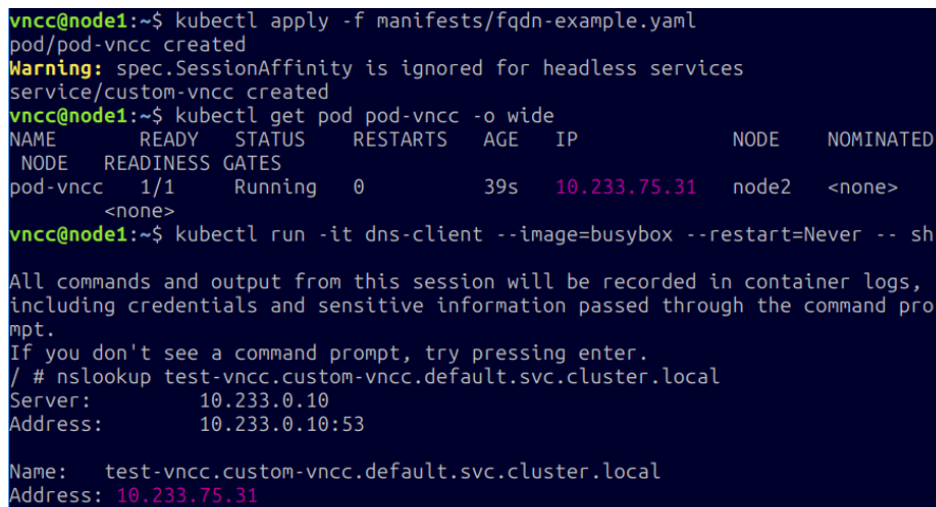
- `hostname: test-vncc` → nome host del Pod, scelto dall'utente.
- `subdomain: custom-vncc` → deve coincidere con il nome del Service headless.
- L'FQDN finale del Pod diventa:
`test-vncc.custom-vncc.default.svc.cluster.local`

Esecuzione

Applicazione e test delle risorse:

```
kubectl apply -f manifests/fqdn-example.yaml
kubectl get pod pod-vncc -o wide
kubectl run -it dns-client --image=busybox --restart=Never -- sh
#- nslookup test-vncc.custom-vncc.default.svc.cluster.local
```

Il comando `nslookup` conferma che il nome DNS personalizzato risolve all'IP del Pod `pod-vncc`, evidenziato in viola nell'immagine sotto riportata.



```
vncc@node1:~$ kubectl apply -f manifests/fqdn-example.yaml
pod/pod-vncc created
Warning: spec.SessionAffinity is ignored for headless services
service/custom-vncc created
vncc@node1:~$ kubectl get pod pod-vncc -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP            NODE     NOMINATED
NODE     READINESS GATES
pod-vncc  1/1     Running   0           39s   10.233.75.31  node2    <none>
vncc@node1:~$ kubectl run -it dns-client --image=busybox --restart=Never -- sh

All commands and output from this session will be recorded in container logs,
including credentials and sensitive information passed through the command pro
mpt.
If you don't see a command prompt, try pressing enter.
/ # nslookup test-vncc.custom-vncc.default.svc.cluster.local
Server:      10.233.0.10
Address:     10.233.0.10:53

Name:   test-vncc.custom-vncc.default.svc.cluster.local
Address: 10.233.75.31
```

Conclusione

- Prima di v1.34 i Pod avevano soltanto nomi DNS generati automaticamente, non personalizzabili e spesso poco leggibili.
- Da v1.34 è possibile definire `hostname` e `subdomain`, ottenendo FQDN personalizzati e più chiari.
- Questa funzionalità semplifica l'integrazione con applicazioni legacy e in scenari che richiedono hostname fissi.

3.7 Horizontal Pod Autoscaler con nuove tolleranze

Obiettivo

Dimostrare il funzionamento dell'**Horizontal Pod Autoscaler (HPA)** con le nuove tolleranze separate di `scaleUp` e `scaleDown`, introdotte in Kubernetes v1.34. L'applicazione di 'stress test' genera un carico CPU per 60 secondi,

permettendo di osservare sia la fase di aumento delle repliche sia quella di riduzione.

Differenze rispetto alle versioni precedenti

- Fino a Kubernetes v1.33 l'HPA utilizzava la stessa tolleranza per l'aumento e la diminuzione delle repliche, con conseguente rischio di *flapping* (repliche che salivano e scendevano rapidamente).
- Da Kubernetes v1.34 le tolleranze sono separate:
 - `scaleUp.tolerance` → più aggressivo, reagisce subito al carico.
 - `scaleDown.tolerance` → più prudente, riduce gradualmente le repliche.
- Risultato: scaling più stabile e prevedibile.

Prerequisito: metrics-server

Installazione del componente per raccogliere metriche di CPU e memoria e patch per cluster kubeadm locali:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

kubectl -n kube-system patch deployment metrics-server --type='json' \
  -p='[{"op":"add","path":"/spec/template/spec/containers/0/args/-","value":"--kubelet-insecure-tls"}]'
```

```
kubectl -n kube-system patch deployment metrics-server --type='json' \
  -p='[{"op":"add","path":"/spec/template/spec/containers/0/args/-","value":"--kubelet-preferred-address-types=InternalIP"}]'
```

```
kubectl -n kube-system rollout restart deployment metrics-server
```

Verifica del funzionamento:

```
kubectl top nodes
kubectl top pods
```

Se le metriche CPU/Memory sono visibili, il `metrics-server` è operativo.

Deployment di test

File manifests/hpa-example-deploy.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hpa-example
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hpa-example
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
      - name: stress
        image: polinux/stress
        command: ["stress"]
        args: ["--cpu", "2", "--timeout", "60s"]
        resources:
          requests:
            cpu: "100m"
          limits:
            cpu: "500m"
```

Questo Pod consuma CPU per 60 secondi, quindi il carico cala e l'HPA può ridurre le repliche.

Definizione dell'HPA

File manifests/hpa-example-autoscaler.yaml:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-example
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hpa-example
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  behavior:
    scaleUp:
```



```
    tolerance: 0.1
  scaleDown:
    tolerance: 0.3
```

Configurazione:

- parte da 1 replica
- scala fino a massimo 5 repliche
- obiettivo: 50% CPU usage
- scaleUp rapido, scaleDown prudente

Esecuzione

```
kubectl apply -f manifests/hpa-example-deploy.yaml
kubectl apply -f manifests/hpa-example-autoscaler.yaml
```

```
kubectl get hpa hpa-example -w
kubectl get pods -l app=hpa-example -w
```

Osservazioni

1. **Fase iniziale:** 1 Pod in esecuzione, CPU cresce rapidamente.
2. **Scale-up:** HPA rileva CPU > 50% e scala fino a 5 repliche.
3. **Fine carico (dopo 60s):** i processi terminano, i Pod non consumano più CPU.
4. **Scale-down:** HPA rileva CPU bassa, ma riduce gradualmente le repliche fino a 1, grazie alla tolleranza 0.3.

Debug utile

```
kubectl describe hpa hpa-example -w
kubectl get pods
```

```

vncc@node1:~$ kubectl get pods -l app=hpa-example -w
NAME                                READY    STATUS    RESTARTS   AGE
hpa-example-5b77c7856-9m8lr         1/1      Running   0           50s
hpa-example-5b77c7856-9pns7         1/1      Running   0           29s
hpa-example-5b77c7856-ks2pw         1/1      Running   0           29s
hpa-example-5b77c7856-z5wr6         1/1      Running   0           29s
^[[A^[[A^[[A^[[B^[[B^[[B^Cvncc@node1:~$
vncc@node1:~$
vncc@node1:~$
vncc@node1:~$ kubectl get hpa hpa-example -w
NAME                                REFERENCE          TARGETS          MINPODS   MAXPODS   REPLI
CAS    AGE
hpa-example-8m6s    Deployment/hpa-example    cpu: 0%/50%      1          5          4
hpa-example-8m31s   Deployment/hpa-example    cpu: <unknown>/50%  1          5          4
hpa-example-9m1s    Deployment/hpa-example    cpu: 610%/50%     1          5          4
hpa-example-9m16s   Deployment/hpa-example    cpu: 108%/50%     1          5          5

```

Conclusione

- Prima di v1.34: una sola tolleranza per scale-up e scale-down, con rischio di oscillazioni continue.
- Con v1.34: scale-up rapido e scale-down cauto → scaling più stabile e prevedibile.
- La demo con `polinux/stress` mostra chiaramente la fase di crescita fino a 5 repliche e la discesa graduale fino a 1.

3.8 OpenTelemetry + Jaeger (Tracing App)

Obiettivo

Mostrare l'integrazione di **OpenTelemetry Collector** con **Jaeger** per la raccolta e la visualizzazione di tracce distribuite. Una delle novità di Kubernetes v1.34 è la possibilità per il `kubelet` di inviare tracce OTLP direttamente al Collector, senza più bisogno di feature gate sperimentali (in questa versione la funzionalità è promossa a GA).

Manifest dello stack

Definizione del file `manifests/tracing-stack.yaml`, che comprende Jaeger, OpenTelemetry Collector e i relativi Service:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: jaeger
  namespace: monitoring
spec:
  replicas: 1

```

```

    selector:
      matchLabels:
        app: jaeger
  template:
    metadata:
      labels:
        app: jaeger
    spec:
      containers:
        - name: jaeger
          image: jaegertracing/all-in-one:1.53
          ports:
            - containerPort: 16686 # UI Web
            - containerPort: 14268 # Collector HTTP API
            - containerPort: 4317 # OTLP gRPC
            - containerPort: 4318 # OTLP HTTP
---
apiVersion: v1
kind: Service
metadata:
  name: jaeger-query
  namespace: monitoring
spec:
  selector:
    app: jaeger
  ports:
    - name: query
      port: 16686
      targetPort: 16686
    type: ClusterIP
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: otel-collector-config
  namespace: monitoring
data:
  config.yaml: |
    receivers:
      otlp:
        protocols:
          grpc:
            endpoint: 0.0.0.0:4317
          http:
            endpoint: 0.0.0.0:4318
    processors:
      batch:

```

```

    exporters:
      otlphttp:
        endpoint:
http://jaeger.monitoring.svc.cluster.local:14268/api/traces
        logging:           # aggiunto per debug
        loglevel: debug
    service:
      pipelines:
        traces:
          receivers: [otlp]
          processors: [batch]
          exporters: [otlphttp, logging]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: otel-collector
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: otel-collector
  template:
    metadata:
      labels:
        app: otel-collector
    spec:
      hostNetwork: true
      dnsPolicy: ClusterFirstWithHostNet
      containers:
        - name: otel-collector
          image: otel/opentelemetry-collector-contrib:0.104.0
          args: ["--config=/conf/config.yaml"]
          ports:
            - containerPort: 4317
            - containerPort: 4318
          volumeMounts:
            - name: config
              mountPath: /conf
      volumes:
        - name: config
          configMap:
            name: otel-collector-config
---
apiVersion: v1
kind: Service

```

```
metadata:
  name: otel-collector
  namespace: monitoring
spec:
  selector:
    app: otel-collector
  ports:
  - name: otlp-grpc
    port: 4317
    targetPort: 4317
  - name: otlp-http
    port: 4318
    targetPort: 4318
```

Passaggi eseguiti e test

1. Creazione del namespace `monitoring` e applicazione dello stack.
2. Configurazione del `kubelet` su entrambi i nodi, aggiungendo nel file `/var/lib/kubelet/config.yaml` la sezione (dopo aver controllato che girasse definitivamente nel worker e per evitare problemi di indirizzamento che si sono presentati):

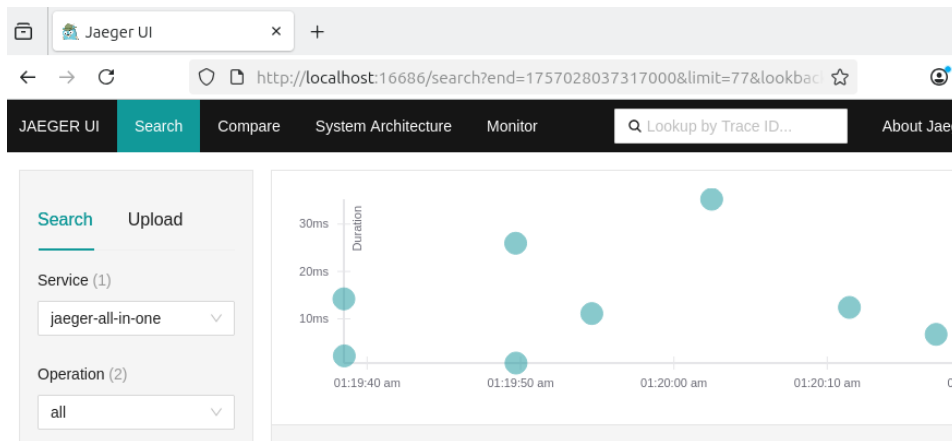
```
tracing:
  samplingRatePerMillion: 1000000
  endpoint: "192.168.43.11:4317"
  samplingStrategy: Always
```

3. Abilitazione del Collector con `hostNetwork` e modifica della ConfigMap per esporre i receiver OTLP su `0.0.0.0:4317` e `0.0.0.0:4318`.
4. Test di risoluzione DNS e connettività (`nslookup`, `telnet`, `netstat`) per verificare la raggiungibilità del Collector da entrambi i nodi.
5. Abilitazione dell'exporter `logging` nel Collector per stampare nei log eventuali spans ricevuti.
6. Generazione manuale di attività (creazione e cancellazione pod `busybox`) per stimolare il kubelet a produrre eventi.

Risultato osservato

- La UI di Jaeger è stata correttamente avviata e risponde in `http://localhost:16686`.
- L'OpenTelemetry Collector è in esecuzione e riceve connessioni OTLP sulle porte esposte.

- Il `kubelet` non ha più mostrato errori di DNS o di connessione rifiutata verso il Collector, segno che la pipeline è configurata correttamente.
- Tuttavia, non sono stati osservati spans effettivi né nei log del Collector né nella UI di Jaeger, nonostante la configurazione di sampling al 100% e la generazione di traffico artificiale.



Conclusione

La parte infrastrutturale (Collector, Jaeger, configurazione kubelet) è stata portata a termine con successo e testata. Il problema rimasto è che il `kubelet`, pur essendo configurato per esportare le tracce, non sembra emettere spans osservabili: si tratta probabilmente di una limitazione o bug nella release attuale, anche se la funzionalità `KubeletTracing` è stata appena promossa a GA. Per la demo è quindi possibile mostrare la configurazione e l'architettura completa, evidenziando come in un contesto reale ci si aspetterebbe di vedere anche le tracce del kubelet insieme a quelle delle applicazioni strumentate.

3.9 Mutating Admission Policy (CEL)

Obiettivo

Mostrare il funzionamento di una **Mutating Admission Policy** scritta in CEL (Common Expression Language). La novità introdotta in Kubernetes v1.34 (ancora in fase *alpha*) permette di modificare le richieste API direttamente tramite regole dichiarative, senza dover ricorrere a webhook esterni.

File di esempio

Policy e binding Definizione del file `manifests/mutate-add-env.yaml`, che applica la variabile `LOG_LEVEL=ERROR` ai Pod etichettati con `env=production`:

```

apiVersion: admissionregistration.k8s.io/v1alpha1
kind: MutatingAdmissionPolicy
metadata:
  name: add-env-production
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: ["v1"]
        operations: ["CREATE"]
        resources: ["pods"]
  mutations:
    - op: add
      path: "/spec/containers/0/env/-"
      value: {"name": "LOG_LEVEL", "value": "ERROR"}
      matchCondition: "object.metadata.labels.env == 'production'"
---

```

```

apiVersion: admissionregistration.k8s.io/v1alpha1
kind: MutatingAdmissionPolicyBinding
metadata:
  name: add-env-binding
spec:
  policyName: add-env-production

```

Pod con label env=production

```

apiVersion: v1
kind: Pod
metadata:
  name: test-prod
  labels:
    env: production
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "env; sleep 3600"]

```

Pod senza label

```

apiVersion: v1
kind: Pod
metadata:
  name: test-nolabel
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "env; sleep 3600"]

```

Passaggi e risultati attesi

1. Applicare la policy:

```
kubectl apply -f manifests/mutate-add-env.yaml
```

Nel cluster kubeadm locale appare l'errore:

```
no matches for kind "MutatingAdmissionPolicy"
```

Questo è normale: la feature è *alpha* e non è abilitata di default.

2. Applicare i Pod di test:

```
kubectl apply -f manifests/pod-test.yaml  
kubectl apply -f manifests/pod-test-nolabel.yaml
```

3. Con la policy attiva:

- `test-prod` riceve automaticamente la variabile d'ambiente `LOG_LEVEL=ERROR`.
- `test-nolabel` resta invariato.

4. Comando di verifica:

```
kubectl exec -it test-prod -- env | grep LOG_LEVEL
```

Output atteso:

```
LOG_LEVEL=ERROR
```

Conclusione

- Le **Mutating Admission Policy** basate su CEL permettono di applicare regole di mutazione in modo semplice e dichiarativo.
- Nelle versioni precedenti questo era possibile solo tramite webhook esterni, più complessi da gestire.

3.10 Image Volume (OCI artifact come volume)

Obiettivo

Mostrare la nuova tipologia di volume **Image Volume**, introdotta in Kubernetes v1.34 (ancora in fase *alpha*), che permette di montare direttamente il

contenuto di un'immagine OCI come volume, senza dover utilizzare `initContainer`.

File di esempio

Pod con Image Volume (nuova funzionalità) `manifests/pod-image-volume.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: image-volume-test
spec:
  containers:
    - name: app
      image: busybox
      command: ["cat", "/data/hello.txt"]
      volumeMounts:
        - name: imgvol
          mountPath: /data
  volumes:
    - name: imgvol
      image:
        reference: docker.io/library/testdata:latest
```

Variante fallback con `initContainer` `manifests/pod-image-volume-fallback.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: image-volume-fallback
spec:
  volumes:
    - name: data
      emptyDir: {}
  initContainers:
    - name: init
      image: busybox
      command: ["sh", "-c", "echo 'Hello from image!' > /data/hello.txt"]
      volumeMounts:
        - name: data
          mountPath: /data
  containers:
    - name: app
      image: busybox
      command: ["cat", "/data/hello.txt"]
      volumeMounts:
        - name: data
```

```
mountPath: /data
```

Passaggi

1. Provare la nuova feature:

```
kubectl apply -f manifests/pod-image-volume.yaml
kubectl logs image-volume-test
```

Risultato atteso: il Pod dovrebbe mostrare il contenuto del file `hello.txt` montato direttamente dall'immagine. **Risultato reale con containerd:**

```
cat: can't open '/data/hello.txt': No such file or directory
```

La funzionalità non è ancora supportata dal runtime `containerd`.

2. Visualizzazione della variante fallback per :

```
kubectl apply -f manifests/pod-image-volume-fallback.yaml
kubectl logs image-volume-fallback
```

Output atteso:

```
Hello from image!
```

In questo caso un `initContainer` copia il file in un volume `emptyDir`, metodo tradizionale comunque funzionante.

Conclusione

- Prima di Kubernetes v1.34 era necessario ricorrere a un `initContainer` per inizializzare il contenuto di un volume.
- Con v1.34 è stata introdotta la possibilità di definire direttamente un `image: volume`, che monta il contenuto di un'immagine OCI.
- Nel cluster `kubeadm` locale, basato su `containerd`, la feature non è ancora supportata dal runtime: per questo è stata mostrata anche la variante fallback, comunque per evidenziare chiaramente il miglioramento introdotto dalla nuova release in termini di praticità e immediatezza.

3.11 Dynamic Resource Allocation (DRA)

Obiettivo

Mostrare la novità introdotta in Kubernetes v1.34: il framework di **Dynamic Resource Allocation (DRA)**. Questa funzionalità consente di gestire risorse speciali (GPU, FPGA, SmartNIC, dispositivi custom) in modo dichiarativo e dinamico, senza legare i Pod a configurazioni rigide dei device plugin.

Come funziona

Con DRA sono stati introdotti nuovi oggetti API (ancora in *alpha*):

- **DeviceClass** → definisce un tipo di device (es. `nvidia-gpu`).
- **ResourceClaim** → rappresenta una richiesta di device dinamica da parte di un utente o workload.
- I Pod possono dichiarare una dipendenza da uno o più **ResourceClaim**, e il sistema assegna automaticamente le risorse disponibili.

Manifest di esempio

```
apiVersion: resource.k8s.io/v1alpha2
kind: DeviceClass
metadata:
  name: nvidia-gpu
---
apiVersion: resource.k8s.io/v1alpha2
kind: ResourceClaim
metadata:
  name: gpu-claim
spec:
  devices:
    requests:
      - name: gpu
        deviceClassName: nvidia-gpu
---
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  resourceClaims:
    - name: gpu-claim
  containers:
    - name: app
```

```
image: busybox
command: ["sh", "-c", "echo Using GPU resource; sleep 3600"]
resources:
  claims:
    - name: gpu-claim
```

Osservazioni

- **Prima di v1.34:** ogni device plugin definiva un modello di allocazione personalizzato, rendendo i Pod rigidi e difficili da migrare tra cluster.
- **Con DRA:** il modello diventa uniforme e Kubernetes gestisce direttamente le richieste dinamiche di risorse.
- Questa funzionalità rappresenta la base per un futuro in cui GPU e altre risorse speciali potranno essere allocate e condivise con la stessa semplicità di CPU e memoria.

Limitazioni nel cluster locale

Nel cluster kubeadm su VM Lubuntu non è stato possibile testare concretamente la funzionalità, perché il mio sistema non presenta le componenti hardware adeguate:

- È necessario disporre di device plugin compatibili (ad esempio per GPU NVIDIA).
- In assenza di hardware e plugin adeguati, i Pod rimangono infatti nello stato `Pending`.

Capitolo 4

Conclusioni e sviluppi futuri

La versione **Kubernetes v1.34** ha introdotto un numero significativo di nuove funzionalità, che apportano miglioramenti su diversi aspetti della piattaforma. Le novità spaziano da strumenti a supporto degli sviluppatori, a nuove API per la gestione dei carichi di lavoro, fino a funzionalità avanzate per il tracciamento e l'allocazione delle risorse.

Migliorie introdotte

- **Maggiore leggibilità e controllo** con il nuovo formato **KYAML**, che rende più semplice interpretare le risorse direttamente da riga di comando.
- **Gestione più fine dei Job**, grazie a **PodReplacementPolicy**, che consente di differenziare i casi di Pod falliti da quelli terminati manualmente.
- **Flessibilità nel riavvio dei container** con **ContainerRestartRules**, che permette regole specifiche basate sugli exit code.
- **Instradamento del traffico più efficiente** con le nuove modalità di **ServiceTopology**, che introducono la preferenza per Pod locali al nodo o alla zona.
- **DNS personalizzati** per i Pod, tramite hostname e subdomain definiti dall'utente, semplificando l'integrazione con sistemi legacy.
- **Autoscaling più stabile** con il nuovo comportamento dell'HPA, che distingue tra tolleranze di scale-up e scale-down.
- **Osservabilità avanzata** con l'integrazione di OpenTelemetry e Jaeger, rafforzata dal supporto GA al kubelet tracing.
- **Mutazioni delle richieste API** tramite le Mutating Admission Policy in CEL, che aprono la strada a regole di validazione e modifica senza necessità di webhook esterni.
- **Gestione dei volumi semplificata** con gli **ImageVolume**, che permettono di montare direttamente il contenuto di immagini OCI.

- **Allocazione dinamica delle risorse speciali** con il framework DRA, che fornisce un modello uniforme per GPU e altri device.

Sviluppi futuri

Il lavoro svolto ha mostrato concretamente le feature stabili già disponibili in Kubernetes v1.34, mentre per alcune ancora *'alpha'* è stato possibile illustrare la configurazione e il comportamento atteso. Alcuni sviluppi futuri prevedono:

- **Verifica pratica delle feature alpha** non appena queste raggiungeranno lo stato *beta* o *stable*, in modo da non compromettere la stabilità del cluster realizzato con `kubeadm`.
- **Test in ambienti con risorse hardware dedicate** (GPU, FPGA, SmartNIC) per validare concretamente il framework DRA.
- **Integrazione con strumenti di osservabilità esterni**, sfruttando a pieno il kubelet tracing ora in GA.

Considerazioni finali sui feature gate e funzionalità alpha

Durante le prove è stato tentato di abilitare le funzionalità *alpha* tramite **feature gate** nei file di configurazione del sistema. Tuttavia l'esperimento ha mostrato come questi flag vadano a toccare componenti delicati del cluster (`kube-apiserver`, `kubelet`, `scheduler`), causando ad oggi instabilità e difficoltà nell'avvio del control plane e degli altri componenti. Per questo motivo, nel cluster locale configurato con `kubeadm`, l'attivazione non è stata proseguita e le aggiunte nei file di configurazioni interni sono state commentate per rendere immediata una verifica futura.

Riferimenti

Durante la progettazione e la realizzazione del progetto sono state consultate le seguenti fonti:

- Hassanzadeh, S.D. – *Everything New in Kubernetes 1.34 with Examples*: <https://medium.com/@hassanzadeh.sd/everything-new-in-kubernetes-1-34-with-examples-coming-august-27-2025-d4a96963c597>
- Kubernetes Blog – *Kubernetes v1.34 Release Announcement* (27 agosto 2025): <https://kubernetes.io/blog/2025/08/27/kubernetes-v1-34-release/>
- GitHub – *Kubespary issue #12510: Support for Kubernetes v1.34*: <https://github.com/kubernetes-sigs/kubespary/issues/12510>
- Cloudsmith – *Kubernetes 1.34: What You Need to Know*: <https://cloudsmith.com/blog/kubernetes-1-34-what-you-need-to-know>
- GitHub – *Repository ufficiale del progetto Kubernetes*: <https://github.com/kubernetes/kubernetes>