Università degli Studi di Perugia

Corso di Laurea Magistrale in Data Science

Progetto Virtual Networks and Colud Computing: Analisi delle novità in Kubernetes 1.33 e confronto con 1.32

Studente: *Alessandra Ruggeri*

Professore: *Prof. Gianluca reali*

Anno Accademico 2024/2025

Indice

Introduzione

L'obiettivo di questo lavoro è analizzare e confrontare alcune funzionalità di Kubernetes tra la versione **1.32.2** e la più recente **1.33.4**, all'interno di un ambiente di laboratorio realizzato con VirtualBox e Kubespray. Il progetto nasce dall'esigenza di comprendere in maniera pratica i cambiamenti introdotti nelle versioni più recenti del sistema di orchestrazione, con particolare attenzione alle funzionalità che hanno un impatto diretto sulla gestione dei Pod, dei servizi e delle risorse specializzate.

Il laboratorio è stato realizzato predisponendo due cluster distinti, ciascuno composto da due nodi, su cui sono stati condotti esperimenti mirati. In particolare, ci si è concentrati sulle differenze e sulle novità più significative tra le due versioni:

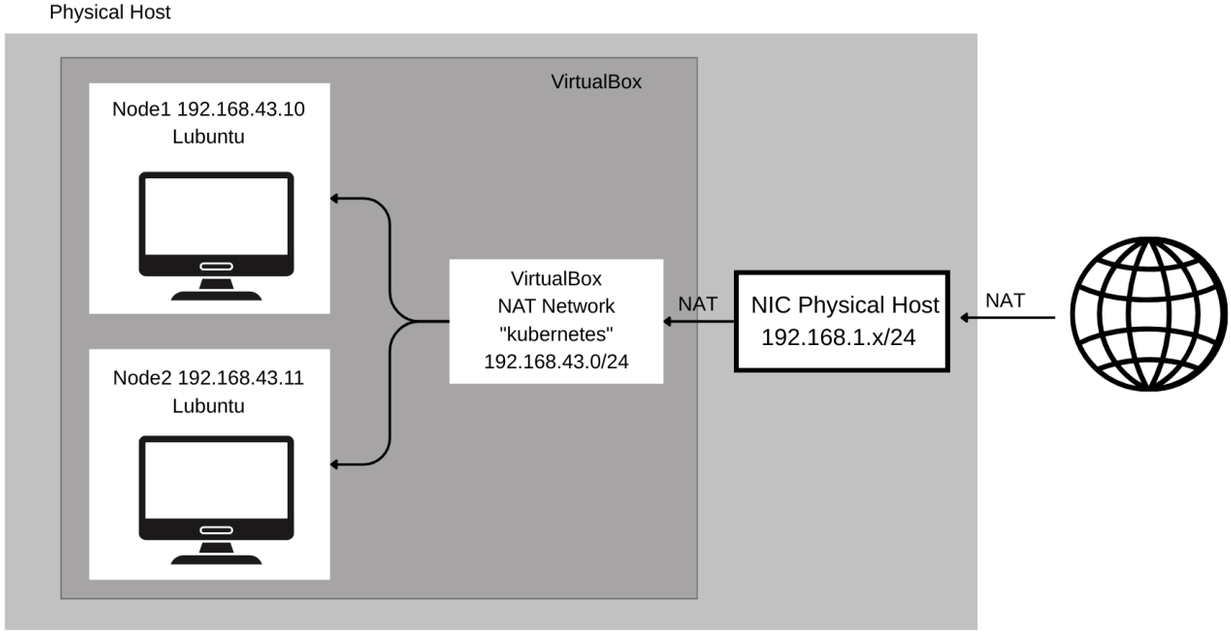
- **Endpoint vs EndpointSlice**: a partire dalla versione 1.33, la risorsa `Endpoints` viene ufficialmente deprecata a favore delle `EndpointSlice`, più scalabili ed efficienti nella gestione di un alto numero di Pod.
- **Sidecar nativo**: mentre in Kubernetes 1.32 era necessario ricorrere a workaround (definendo due container "pari" nello stesso Pod), in 1.33 è disponibile il supporto nativo ai sidecar tramite `initContainers` con `restartPolicy: Always`.
- **In-place Pod Resize**: funzionalità introdotta in 1.33 che permette di modificare le risorse (CPU e memoria) associate a un Pod senza doverlo ricreare, eliminando quindi il downtime associato a questa operazione.
- **Dynamic Resource Allocation (DRA)**: nuove API per l'allocazione dinamica di risorse specializzate (GPU, FPGA, SmartNIC), ancora in stato `beta` ma già utilizzabili in contesti sperimentali.

- **Image Volumes**: possibilità di montare direttamente un'immagine OCI come volume di sola lettura; in 1.33 la funzionalità è ancora limitata su `containerd`, ma è stato possibile dimostrarne l'uso tramite fallback con `initContainer` ed `emptyDir`.
- **Altre modifiche minori**: introduzione del campo `metadata.generation` per i Pod, rimozione del campo `kubeProxyVersion` dai Node e deprecazione definitiva del volume `gitRepo`.

Capitolo 1

Setup dell'ambiente

In questo capitolo vengono descritti i passaggi per la creazione dell'ambiente di laboratorio basato su VirtualBox e due macchine virtuali `Xubuntu 24.04`, denominate `node1` e `node2`, che fungeranno rispettivamente da *master* e *worker*.



1.1 Creazione rete VirtualBox

1. Aprire VirtualBox → *File* → *Strumenti* → *Rete* → tab *Reti con NAT*.
2. Cliccare su *Aggiungi (+)*.
3. Configurare come segue:
   - Nome rete: `kubernetes`
   - CIDR: `192.168.43.0/24`
   - Gateway (NAT): `192.168.43.1`
   - DHCP: disattivato (si useranno IP statici)

1.2 Creazione delle VM

1. Importare due volte il file `xubuntu24-o4.ova` (oppure clonare una VM esistente con *copia completa*).
2. Assegnare risorse consigliate:
   - CPU: 2 vCPU
   - RAM: 4 GB
   - Disco: $\geq$ 25 GB
3. Rinominare le VM in `node1` (master) e `node2` (worker).
4. Per entrambe: Impostazioni → Rete → Scheda 1 → Rete con NAT → selezionare `kubernetes`.

1.3 Configurazione IP statici

Verificare il nome dell'interfaccia di rete:

```
ip -br a
```

(solitamente `enp0s3` in VirtualBox).

Modificare/creare il file `/etc/netplan/01-network-manager-all.yaml`.

node1 (master)

```
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]
```

node2 (worker)

```
network:
  version: 2
  renderer: NetworkManager
  ethernets:
    enp0s3:
      dhcp4: no
      addresses: [192.168.43.11/24]
      routes:
        - to: default
          via: 192.168.43.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
```

Disabilitare eventuali file di configurazione creati da cloud-init:

```
sudo mv /etc/netplan/50-cloud-init.yaml /etc/netplan/50-cloud-init.yaml.bak 2>/dev/null || true
```

Applicare la configurazione:

```
sudo netplan apply
```

1.4 Hostname e file `/etc/hosts`

Impostare l'hostname di ciascuna VM:

```
# su node1
echo node1 | sudo tee /etc/hostname
sudo hostnamectl set-hostname node1


# su node2
echo node2 | sudo tee /etc/hostname
sudo hostnamectl set-hostname node2
```

Configurare il file `/etc/hosts` (uguale su entrambe le VM):

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

Dopo il riavvio, verificare la connettività con:

```
ping -c2 node2   # da node1
ping -c2 node1   # da node2
```

1.5 Pacchetti base e configurazione SSH

Installazione pacchetti base

Su entrambi i nodi:

```
sudo apt update
sudo apt install -y openssh-server python3-pip
```

Generazione chiave SSH

Generare una chiave SSH su ciascun nodo. Quando viene chiesta la passphrase, premere invio (vuota).

```
ssh-keygen -t rsa
```

Copia della chiave pubblica

Dal nodo `node1` copiare la chiave pubblica su entrambi i nodi:

```
ssh-copy-id 192.168.43.10   # verso se stesso (node1)
ssh-copy-id 192.168.43.11   # verso node2
```

Verificare l'accesso senza password:

```
ssh node2 hostname   # deve rispondere "node2"
ssh node1 hostname   # deve rispondere "node1"
```

Configurazione sudo senza password

Modificare il file `sudoers` con:

```
sudo visudo
```

Sotto la riga:

```
%sudo   ALL=(ALL:ALL) ALL
```

aggiungere:

```
root ALL=(ALL) NOPASSWD: ALL
vncc  ALL=(ALL) NOPASSWD: ALL
```

Verificare:

```
sudo whoami   # deve rispondere "root"
```

1.6 Installazione di Kubespray (solo su node1)

Pacchetti di base

```
sudo apt update
sudo apt install -y git python3-venv python3-pip
python3 --version
```

Clonare Kubespray e preparare l'ambiente

```
git clone https://github.com/kubernetes-sigs/kubespray.git
cd kubespray

python3 -m venv kubespray-venv
source kubespray-venv/bin/activate

pip install -U pip setuptools wheel
pip install -U -r requirements.txt
```

Creazione inventario

```
cp -rfp inventory/sample inventory/mycluster
```

1.7 Configurazione dell'inventario Kubespray

File `inventory.ini`

Aprire il file `inventory/mycluster/inventory.ini` e sostituirne il contenuto con:

```
[kube_control_plane]
node1 ansible_host=192.168.43.10

[etcd:children]
kube_control_plane

[kube_node]
node2 ansible_host=192.168.43.11
```

File `all.yml`

Aprire il file `inventory`/$mycluster/group_vars/all/all.yml$, cercare la sezione `upstream_dns_servers:` e impostarla come segue:

```
upstream_dns_servers:
```

```
  - 8.8.8.8
  - 8.8.4.4
```

File `k8s-cluster.yml`

Aprire il file $inventory/mycluster/group_vars/k8s_cluster/k8s-cluster.yml$ e configurare le variabili principali:

```
kube_version: 1.32.2   # <-- IMPORTANTE: versione cluster
kube_network_plugin: calico

kube_service_addresses: 10.233.0.0/18
kube_pods_subnet: 10.233.64.0/18
kube_network_node_prefix: 24

container_manager: containerd
resolvconf_mode: host_resolvconf
```

1.8 Reset e installazione del cluster (Kubernetes 1.32.2)

Prima di procedere con l'installazione definitiva, è consigliato clonare le macchine virtuali in modo da avere una base pronta anche per la successiva installazione del cluster in versione 1.33.

Preparazione finale (fix sudo e utente)

Eseguire su entrambi i nodi:

```
# 1. Abilita sudo senza password per l'utente vncc
echo 'vncc ALL=(ALL) NOPASSWD: ALL' | sudo tee /etc/sudoers.d/99-vncc-nopasswd >/dev/null
sudo chmod 440 /etc/sudoers.d/99-vncc-nopasswd

# 2. Rinomina il file che sovrascriveva le regole
sudo mv /etc/sudoers.d/10-installer /etc/sudoers.d/10-installer.bak

# 3. Verifica sintassi sudoers
```

```
sudo visudo -cf /etc/sudoers && echo "sudoers OK" || echo "ERRORE sudoers"

# 4. Verifica che sudo non chieda password
sudo -n true && echo "sudo -n OK" || echo "sudo -n FALLITO"
```

Dal nodo `node1`, testare l'accesso via SSH:

```
ssh node1 'sudo -n id'
ssh node2 'sudo -n id'
```

Entrambi i comandi devono rispondere `uid=0(root)` senza password.

Verifica inventory

Controllare che il file `inventory/mycluster/inventory.ini` abbia il seguente contenuto:

```
[kube_control_plane]
node1 ansible_host=192.168.43.10 ansible_user=vncc

[etcd:children]
kube_control_plane

[kube_node]
node2 ansible_host=192.168.43.11 ansible_user=vncc
```

Attivazione ambiente Python (solo su node1)

Dopo ogni riavvio del nodo master, riattivare l'ambiente virtuale:

```
cd ~/kubespray
source kubespray-venv/bin/activate
```

Reset del cluster (pulizia)

Per sicurezza, è consigliato ripulire eventuali installazioni precedenti:

```
ansible-playbook -i inventory/mycluster/inventory.ini --become --become-user=root reset.yml
```

Installazione del cluster

Infine, avviare l'installazione di Kubernetes 1.32.2:

```
ansible-playbook -i inventory/mycluster/inventory.ini \
  --become --become-user=root \
  --private-key=~/.ssh/id_rsa \
  cluster.yml
```

L'operazione richiede circa 20–30 minuti. L'installazione è corretta se il report finale mostra `failed=0`.

Dopo aver installato il cluster con Kubespray, è necessario predisporre lo strumento `kubectl` per l'utente non root, in modo da poter amministrare il cluster senza utilizzare `sudo`. Inoltre, vengono eseguiti alcuni test di base per verificare che il cluster funzioni correttamente.

Installazione di kubectl

Sul nodo master, installare `kubectl` tramite snap:

```
sudo snap install kubectl --classic
```

Configurazione per l'utente vncc

Creare la directory di configurazione e copiare il file `admin.conf` generato da Kubernetes:

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

Questa operazione consente all'utente `vncc` di usare `kubectl` senza privilegi amministrativi.

Test di base del cluster

Verificare che i nodi siano registrati e in stato `Ready`:

```
kubectl get nodes
```

Controllare che i Pod di sistema siano in esecuzione:

```
kubectl get pods -A
```

Se il setup è corretto, l'output deve includere entrambi i nodi (`node1` e `node2`) in stato `Ready`.

1.9 Setup del cluster 1.33

Per installare un secondo cluster aggiornato alla versione **1.33.4** si è partiti dai cloni delle macchine virtuali già preparate per il cluster 1.32. Questo approccio consente di mantenere la stessa configurazione di rete e hostname, garantendo coerenza tra i due ambienti di test.

Step 1 — Avvio dei cloni

Avviare le VM clonate e, se si desidera, rinominarle ad esempio in `node1-133` (master) e `node2-133` (worker). È fondamentale mantenere gli stessi indirizzi IP (`192.168.43.10` e `192.168.43.11`) e gli stessi hostname (`node1`, `node2`). Per evitare conflitti di rete, assicurarsi che il cluster 1.32 sia spento durante il lavoro con il cluster 1.33.

Step 2 — Attivazione ambiente Kubespray

Sul nodo master (`node1-133`) attivare l'ambiente Python già predisposto:

```
cd ~/kubespray
source kubespray-venv/bin/activate
```

Step 3 — Modifica della versione di Kubernetes

Aprire il file `k8s-cluster.yml` e modificare la variabile `kube_version`:

```
nano inventory/mycluster/group_vars/k8s_cluster/k8s-cluster.yml
```

Sostituire la riga:

```
kube_version: 1.32.2
```

con:

```
kube_version: 1.33.4
```

Salvare e chiudere.

Step 4 — Reset e reinstallazione del cluster

A) Abilitare sudo senza password (su entrambe le VM). Eseguire i seguenti comandi su `node1-133` e `node2-133`:

```
# 1) NOPASSWD per l'utente vncc
echo 'vncc ALL=(ALL) NOPASSWD: ALL' | sudo tee /etc/sudoers.d/99-vncc-nopasswd >/dev/null
sudo chmod 440 /etc/sudoers.d/99-vncc-nopasswd

# 2) Disattiva eventuale override dell'installer (se presente)
if [ -f /etc/sudoers.d/10-installer ]; then
  sudo mv /etc/sudoers.d/10-installer /etc/sudoers.d/10-installer.bak
fi

# 3) Verifica sintassi sudoers
sudo visudo -cf /etc/sudoers && echo "sudoers OK" || echo "ERRORE sudoers"

# 4) Test locale: non deve chiedere password
sudo -n true && echo "sudo -n OK" || echo "sudo -n FALLITO"
```

B) Test via SSH dal master. Dal nodo master (`node1-133`) verificare:

```
ssh node1 'sudo -n id'
ssh node2 'sudo -n id'
```

Entrambi i comandi devono restituire `uid=0(root)` senza prompt di password.

C) Verifica inventory e riattivazione venv. Controllare che il file `inventory.ini` indichi l'utente corretto:

```
nano ~/kubespray/inventory/mycluster/inventory.ini
```

Il contenuto deve essere:

```
[kube_control_plane]
node1 ansible_host=192.168.43.10 ansible_user=vncc
```

```
[etcd:children]
kube_control_plane

[kube_node]
node2 ansible_host=192.168.43.11 ansible_user=vncc
```

Riattivare quindi l'ambiente virtuale:

```
cd ~/kubespray
source kubespray-venv/bin/activate
```

Verificare che nel file `k8s-cluster.yml` sia stata impostata la versione corretta:

```
kube_version: 1.33.4
```

D) Reset e rilancio dei playbook. Infine, eseguire il reset e la reinstallazione del cluster con Ansible:

```
ansible-playbook -i inventory/mycluster/inventory.ini \
  --become --become-user=root reset.yml

ansible-playbook -i inventory/mycluster/inventory.ini \
  --become --become-user=root \
  --private-key=~/.ssh/id_rsa cluster.yml
```

Capitolo 2

Endpoints vs EndpointSlice

In Kubernetes un `Service` fornisce un punto di accesso stabile ai Pod che compongono un'applicazione. Fino alla versione 1.32, ogni Service generava un oggetto `Endpoints`, che conteneva in un'unica lista tutti gli indirizzi IP dei Pod collegati. Questo approccio funziona, ma diventa poco scalabile con numeri elevati di Pod, poiché l'oggetto cresce a dismisura ed è costoso da gestire.

Per affrontare questo problema è stato introdotto `EndpointSlice`, una nuova risorsa che distribuisce gli IP dei Pod su più oggetti più piccoli, semplificando la gestione e rendendo l'architettura più scalabile. A partire da Kubernetes 1.33, gli `Endpoints` sono ufficialmente deprecati, e la rappresentazione standard è costituita esclusivamente dagli `EndpointSlice`.

2.1 Cluster 1.32

Deployment e Service

Per simulare un carico elevato sono stati creati numerosi Pod `busybox` e un Service che li espone. I file usati sono i seguenti:

- `demo-busybox.yaml`: definisce un Deployment di 60 Pod basati su `busybox:stable`, con risorse minime (5m CPU, 8Mi memoria richiesti) e comando `sleep infinity`.
- `demo-service.yaml`: definisce un Service di tipo `ClusterIP` che seleziona i Pod con etichetta `app=demo`.

I manifest sono stati applicati con:

```
kubectl apply -f demo-busybox.yaml
kubectl apply -f demo-service.yaml
```

Dopo la creazione, il numero di Pod è stato scalato a 150 con:

```
kubectl scale deploy demo-busybox --replicas=150
```

Verifica tramite Endpoints

Con il comando:

```
kubectl get endpoints demo-service
```

il sistema ha mostrato che il Service `demo-service` aveva collegato oltre 100 Pod in un unico oggetto `Endpoints`:

```
NAME           ENDPOINTS                                          AGE
demo-service   10.233.75.1:80,10.233.75.10:80,10.233.75.104:80
               + 101 more...                                      37m
```

Il conteggio degli IP registrati ha confermato la numerosità:

```
kubectl get endpoints demo-service \
  -o jsonpath='{range .subsets[*].addresses[*]}{.ip}{"\n"}{end}' | wc -l
104
```

Per avere un mapping diretto IP → Pod sono stati estratti i primi 20 elementi:

```
10.233.75.1   demo-busybox-5fdb67bd84-qcwww
10.233.75.10  demo-busybox-5fdb67bd84-bnlmw
10.233.75.104 demo-busybox-5fdb67bd84-7dqc4
10.233.75.105 demo-busybox-5fdb67bd84-7k5n9
...
```

Infine, l'oggetto `Endpoints` è stato salvato per analisi successive:

```
kubectl get endpoints demo-service -o yaml > endpoints-132.yaml
\head -n 40 endpoints-132.yaml
```

L'header mostra chiaramente che tutti gli IP si trovano sotto un'unica chiave `subsets`, rendendo l'oggetto molto pesante in cluster di grandi dimensioni.

Verifica tramite EndpointSlice

Sullo stesso cluster sono stati controllati anche gli `EndpointSlice`, che già in 1.32 sono disponibili come risorsa parallela:

```
kubectl get endpointslice -l kubernetes.io/service-name=demo-service
```

Il risultato è stato:

```
NAME                 ADDRTYPE   PORT
demo-service-2mtf6   IPv4       80
demo-service-8fc9s   IPv4       80
```

Gli slice erano 2, ciascuno con un sottoinsieme degli IP dei Pod. Dal file `endpointslices-132.yaml` si può vedere la struttura moderna, in cui ogni endpoint ha campi aggiuntivi come `conditions` (ready, serving, terminating) e `nodeName`.

```
- addressType: IPv4
  apiVersion: discovery.k8s.io/v1
  endpoints:
  - addresses:
    - 10.233.75.43
    conditions:
      ready: true
      serving: true
      terminating: false
    nodeName: node2
    targetRef:
      kind: Pod
```

```
      name: demo-busybox-5fdb67bd84-ctcgc
...
```

Questo dimostra che, anche se l'oggetto `Endpoints` era ancora presente e usato ufficialmente, Kubernetes aveva già introdotto il nuovo modello in parallelo.

2.2 Cluster 1.33

Gli stessi file `demo-busybox.yaml` e `demo-service.yaml` sono stati utilizzati sul cluster 1.33.

La differenza emersa è immediata: il comando

```
kubectl get endpoints demo-service -v=8
```

ha prodotto il warning ufficiale di deprecazione:

```
Warning: v1 Endpoints is deprecated in v1.33+; use discovery.k8s.io/v1 EndpointSlice
```

Sebbene l'oggetto `Endpoints` venga ancora restituito (per compatibilità), il sistema invita chiaramente all'uso degli `EndpointSlice`. Questi ultimi, come già visto in 1.32, riportano gli IP suddivisi in più oggetti più leggeri, con metadati arricchiti e una semantica più robusta.

Conclusione

Il confronto mostra due scenari distinti:

- **Kubernetes 1.32**: ogni Service genera un grande oggetto `Endpoints`, affiancato da alcuni `EndpointSlice`. L'oggetto principale rimane comunque `Endpoints`, con tutti gli IP in un'unica lista.
- **Kubernetes 1.33**: l'oggetto `Endpoints` è ufficialmente deprecato e non rappresenta più la fonte primaria. L'unico meccanismo standard è `EndpointSlice`, che garantisce scalabilità e gestibilità.

Gli output raccolti (file `endpoints-132.yaml` ed `endpointslices-132.yaml`) testimoniano il cambiamento architetturale, mentre il warning lato server su 1.33 sancisce la transizione definitiva.

Capitolo 3

Sidecar su Kubernetes 1.32

3.1 Cluster 1.32

In questa sezione viene mostrato il pattern sidecar in un cluster Kubernetes **1.32**, dove non esiste ancora un supporto nativo. Un container Nginx serve una semplice pagina HTML, mentre un container `BusyBox` funge da sidecar leggendo continuamente gli access log di Nginx da un volume condiviso e stampandoli su `stdout`. Questo dimostra come, in assenza di un campo dedicato, il sidecar sia implementato come *workaround*: due container nello stesso Pod che condividono uno o più volumi.

Step 1 — Namespace dedicato

È stato creato un namespace `sidecar-132` per isolare questa demo dal namespace `default` usato in precedenza:

```
kubectl create namespace sidecar-132
kubectl get ns sidecar-132
```

Step 2 — ConfigMap (HTML e configurazione Nginx)

Sono state create due ConfigMap:

- `cm-index-html.yaml`, contenente un file `index.html` molto semplice da servire tramite Nginx.

- `cm-nginx-conf.yaml`, con la configurazione di Nginx per scrivere gli access log su `/var/log/nginx/access.log`, percorso montato in un volume condiviso (`emptyDir`).

Le ConfigMap sono state applicate con:

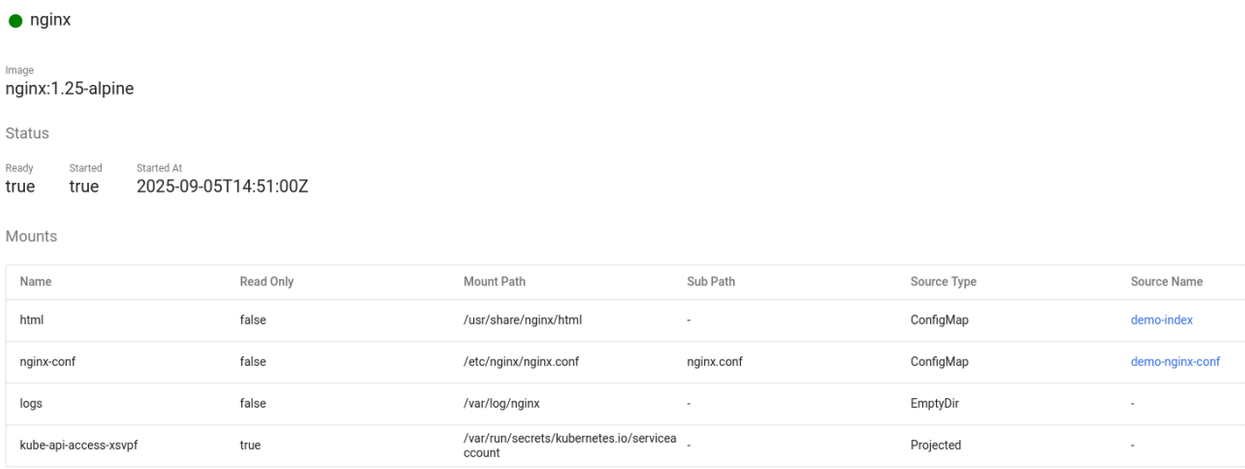
```
kubectl apply -f cm-index-html.yaml
kubectl apply -f cm-nginx-conf.yaml
```

Step 3 — Deployment e Service

È stato creato il manifest `web-with-sidecar-132.yaml`, che definisce:

- **Un Deployment** con un Pod contenente due container:
  - `nginx` → serve la pagina HTML e scrive gli access log.
  - `log-reader` (BusyBox) → sidecar che esegue `tail -F /var/log/nginx/access.log`.

  Entrambi i container montano tre volumi:
  - ConfigMap con la pagina HTML
  - ConfigMap con il file `nginx.conf`
  - `emptyDir` per i log, condiviso tra i due container
- **Un Service NodePort** `web-sidecar-svc`, esposto sulla porta 30080.



● nginx

Image
nginx:1.25-alpine

Status

Ready	Started	Started At
true	true	2025-09-05T14:51:00Z

Mounts

Name	Read Only	Mount Path	Sub Path	Source Type	Source Name
html	false	/usr/share/nginx/html	-	ConfigMap	demo-index
nginx-conf	false	/etc/nginx/nginx.conf	nginx.conf	ConfigMap	demo-nginx-conf
logs	false	/var/log/nginx	-	EmptyDir	-
kube-api-access-xsvpf	true	/var/run/secrets/kubernetes.io/serviceaccount	-	Projected	-

● log-reader

Image
busybox:stable

Status

Ready	Started	Started At
true	true	2025-09-05T14:51:20Z

Commands

```
sh
-c
tail -F /var/log/nginx/access.log
```

Mounts

Name	Read Only	Mount Path	Sub Path	Source Type	Source Name
logs	false	/var/log/nginx	-	EmptyDir	-
kube-api-access-xsvpf	true	/var/run/secrets/kubernetes.io/serviceaccount	-	Projected	-

Verifica

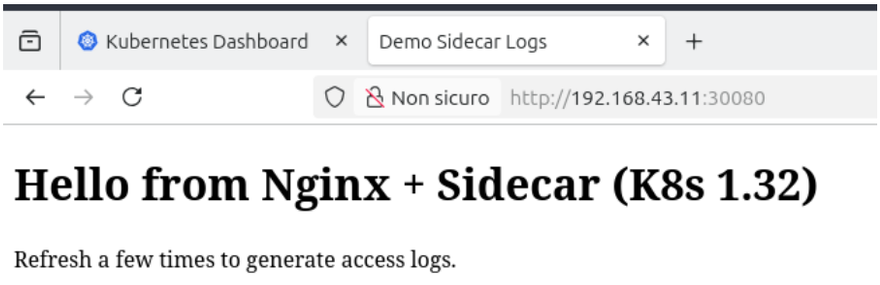
Con i comandi:

```
kubectl get pods -n sidecar-132 -l app=websidecar
kubectl logs -n sidecar-132 -l app=websidecar -c log-reader --tail=20
```

il sidecar (`log-reader`) stampa righe di log ogni volta che si accede a Nginx. Generando traffico di test (verificabile con il seguente comando)

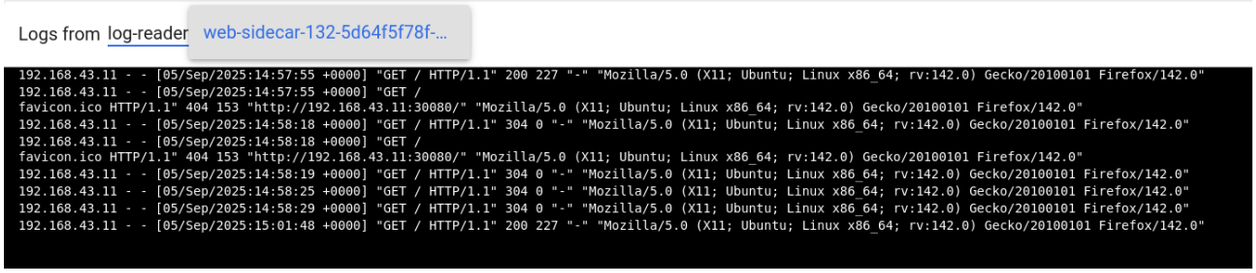
```
curl http://192.168.43.11:30080/
```

Alternativamente si può accedere alla pagina web e fare refresh.



Risultato

Ogni accesso genera una riga in `/var/log/nginx/access.log`, subito mostrata dal sidecar nei log del container



3.2 Cluster 1.33

In questa sezione viene riproposta la stessa demo del sidecar, ma sul cluster **Kubernetes 1.33**. La logica rimane identica: Nginx serve una semplice pagina HTML e un container BusyBox segue in tempo reale gli access log scritti in un volume condiviso. La differenza sostanziale rispetto alla versione 1.32 riguarda la dichiarazione del sidecar: ora è possibile definirlo nativamente tramite un `initContainer` con `restartPolicy: Always`.

Step 1 — Namespace dedicato

```
kubectl create namespace sidecar-133
kubectl get ns sidecar-133
```

Step 2 — ConfigMap (HTML e configurazione Nginx)

Sono state definite due ConfigMap, analoghe a quelle del cluster 1.32:

- `cm-index-html-133.yaml`, con la pagina HTML aggiornata.
- `cm-nginx-conf-133.yaml`, con la configurazione di Nginx per scrivere i log su `/var/log/nginx/access.log`.

```
kubectl apply -f cm-index-html-133.yaml
kubectl apply -f cm-nginx-conf-133.yaml
```

Step 3 — Deployment e Service

Il manifest `web-with-sidecar-133.yaml` definisce:

- Un Deployment con:
  - `nginx` come container principale.
  - `log-reader` come sidecar *nativo*, dichiarato tra gli `initContainers` con `restartPolicy: Always`.
- Un Service NodePort `web-sidecar-133-svc`, esposto sulla porta `30081`.

```
kubectl apply -f web-with-sidecar-133.yaml
kubectl get pods -n sidecar-133 -l app=websidecar133
```

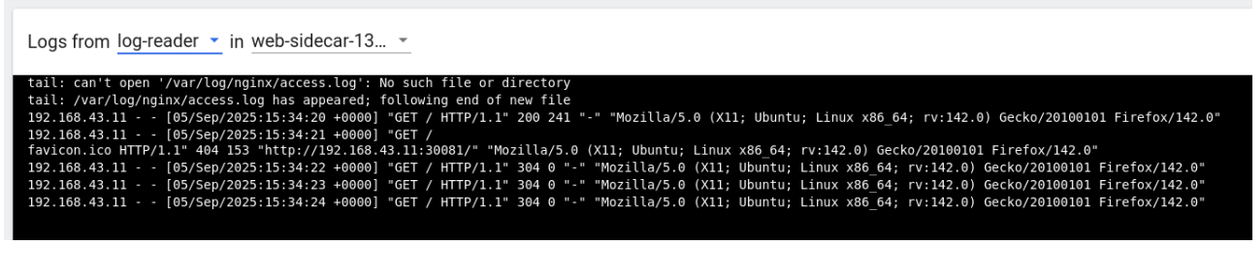
Step 4 — Test e log

Generando traffico verso il servizio:

```
curl http://192.168.43.11:30081/
```

ogni accesso viene registrato e mostrato dal sidecar:

```
kubectl logs -n sidecar-133 -l app=websidecar133 -c log-reader --tail=20 -f
```



3.3 Differenze nello spegnimento del Pod

Sul cluster 1.32, in cui il sidecar è semplicemente un container "pari" affiancato al main, terminando il processo principale di `nginx` con un comando di `kill`, il Pod non viene interrotto nel suo complesso. Lo stato passa momentaneamente a `1/2 NotReady`, perché il sidecar continua a vivere mentre

il container principale è morto. Successivamente, il kubelet interviene riavviando esclusivamente il container `nginx`, senza toccare il sidecar. Il Pod torna quindi in `2/2 Running`, con il sidecar che non ha mai subito interruzioni.

```
vncc@node1:~$ kubectl get pods -n sidecar-133 -l app=websidecar133
NAME                              READY   STATUS    RESTARTS       AGE
web-sidecar-133-9d856f7f5-s95n7   2/2     Running   3 (115s ago)   6m31s
vncc@node1:~$ kubectl exec -n sidecar-133 -c nginx web-sidecar-133-9d856f7f5-s95n7 -- kill 1
vncc@node1:~$ kubectl get pods -n sidecar-133 -l app=websidecar133
NAME                              READY   STATUS      RESTARTS        AGE
web-sidecar-133-9d856f7f5-s95n7   1/2     Completed   3 (2m12s ago)   6m48s
vncc@node1:~$ kubectl get pods -n sidecar-133 -l app=websidecar133
NAME                              READY   STATUS    RESTARTS      AGE
web-sidecar-133-9d856f7f5-s95n7   2/2     Running   4 (45s ago)   7m28s
```

La situazione è diversa sul cluster 1.33. Qui il sidecar è stato dichiarato come `initContainer` con `restartPolicy: Always`, sfruttando quindi il supporto nativo introdotto da questa versione. Quando si forza la chiusura del processo principale di `nginx`, anche il sidecar viene considerato terminato. Il Pod passa in stato `1/2 Completed` e non resta attivo con un sidecar "orfano". Interviene il controller del Deployment, che provvede a ricreare il Pod per intero. Dopo poco tempo entrambi i container tornano in esecuzione.

```
vncc@node1:~$ kubectl get pods -n sidecar-132
NAME                              READY   STATUS    RESTARTS      AGE
web-sidecar-132-5d64f5f78f-6cbl7   2/2     Running   2 (80s ago)   80m
vncc@node1:~$ ^C
vncc@node1:~$ kubectl exec -n sidecar-132 -c nginx web-sidecar-132-5d64f5f78f-6cbl7 -- k
ill 1
vncc@node1:~$ ^C
vncc@node1:~$ kubectl get pods -n sidecar-132
NAME                              READY   STATUS     RESTARTS        AGE
web-sidecar-132-5d64f5f78f-6cbl7   1/2     NotReady   2 (2m29s ago)   81m
vncc@node1:~$ kubectl get pods -n sidecar-132
NAME                              READY   STATUS    RESTARTS      AGE
web-sidecar-132-5d64f5f78f-6cbl7   2/2     Running   3 (18s ago)   81m
```

3.4 Differenze chiave tra 1.32 e 1.33

Il confronto tra le due versioni di Kubernetes evidenzia alcune differenze sostanziali legate al modo in cui viene gestito il sidecar. Nel cluster **1.32** non esiste ancora un concetto nativo di sidecar: per ottenere questo comportamento è stato necessario definire due container "pari" nello stesso Pod, uno principale (Nginx) e uno ausiliario (BusyBox), condivisione di un volume per i log e una serie di accor-

gimenti operativi. Nel cluster **1.33**, invece, il sidecar diventa una funzionalità supportata direttamente dall'API, dichiarato come `initContainer` con `restartPolicy: Always`. Questo elimina la necessità di workaround e rende il ruolo del sidecar esplicito nel manifest, leggibile e standard.

Differenze nei manifest

```yaml
# Kubernetes 1.32 (workaround)
spec:
  containers:
    - name: nginx
      ...
    - name: log-reader
      image: busybox:stable
      command: ["sh","-c","tail -F /var/log/nginx/access.log"]

# Kubernetes 1.33 (sidecar nativo)
spec:
  initContainers:
    - name: log-reader
      image: busybox:stable
      restartPolicy: Always
      command: ["sh","-c","tail -F /var/log/nginx/access.log"]
  containers:
    - name: nginx
      ...
```

Capitolo 4

In-place Pod Resize

4.1 Cluster 1.33

In questa fase è stata verificata la funzionalità di *in-place Pod resize*, introdotta in Kubernetes 1.33. L'obiettivo è quello di dimostrare come sia possibile aggiornare le risorse CPU e memoria associate a un Pod senza doverlo ricreare, a differenza delle versioni precedenti.

Creazione del namespace

Per isolare la demo è stato creato un namespace dedicato chiamato `resize-demo`. Questo consente di raccogliere gli output e successivamente eliminare facilmente tutte le risorse collegate.

Pod di base con risorse iniziali

All'interno del namespace è stato creato un Pod semplice, descritto nel file `resize-pod.yaml`. Il container utilizza l'immagine `busybox:stable` ed è configurato con richieste e limiti molto bassi (50m CPU e 32MiB di memoria come richieste; 100m CPU e 64MiB come limiti). Il Pod è stato avviato e, dopo pochi secondi, è risultato in stato `Running/Ready`.

Per avere una fotografia dello stato iniziale sono stati acquisiti due elementi chiave: le risorse richieste e i limiti del container, e il conteggio dei riavvii. Entrambi hanno confermato che il Pod stava girando con i valori di default e che il numero di `RESTARTS` era pari a zero.

```
vncc@node1:~$ kubectl -n resize-demo get pod resize-pod \
  -o jsonpath='{.spec.containers[0].resources}'; echo
{"limits":{"cpu":"100m","memory":"64Mi"},"requests":{"cpu":"50m","memory
":"32Mi"}}
```

```
vncc@node1:~$ kubectl -n resize-demo get pod resize-pod \
  -o jsonpath='{.status.containerStatuses[0].restartCount}'; echo
0
```

Successivamente è stata preparata una patch (`resize-patch.yaml`) che incrementa significativamente CPU e memoria richieste e disponibili. La patch è stata applicata utilizzando la nuova subresource `resize`, con il comando:

```
kubectl -n resize-demo patch pod resize-pod \
  --subresource=resize --type=merge \
  --patch-file=resize-patch.yaml
```

Una volta applicata la patch, la verifica ha mostrato che le richieste sono state aggiornate a `200m` CPU e `128Mi` di memoria, mentre i limiti sono passati a `300m` CPU e `192Mi` di memoria. Il conteggio dei riavvii è rimasto invariato (zero), a conferma che il Pod non è stato terminato né ricreato. Inoltre, nel dettaglio del Pod (`kubectl describe`) era presente un evento che riportava l'avvenuto resize.

```
vncc@node1:~$ kubectl -n resize-demo get pod resize-pod \
  -o jsonpath='{.spec.containers[0].resources}'; echo
{"limits":{"cpu":"300m","memory":"192Mi"},"requests":{"cpu":"200m","memory":"128Mi"}}
```

```
vncc@node1:~$ kubectl -n resize-demo get pod resize-pod \
  -o jsonpath='{.status.containerStatuses[0].restartCount}'; echo
0
```

4.2 Cluster 1.32

Per verificare il comportamento del resize in-place in Kubernetes 1.32 è stato utilizzato lo stesso approccio visto per il cluster 1.33: creazione di un namespace dedicato, deploy di un Pod con risorse iniziali molto basse, tentativo di modificarne i valori e confronto dei risultati.

È stato creato un namespace chiamato `resize-demo` e al suo interno un Pod definito nel file `resize-pod-132.yaml`. Il container, basato su `busybox:stable`, è stato avviato con richieste minime (50m CPU e 32MiB di memoria) e limiti bassi (100m CPU e 64MiB di memoria). Dopo pochi secondi, il Pod è entrato in stato `Running/Ready`, fornendo così la baseline per l'esperimento.

Lo stato iniziale è stato verificato tramite i comandi `kubectl`, che hanno mostrato correttamente i valori di CPU e memoria impostati e un conteggio dei riavvii pari a zero. A questo punto il Pod risultava stabile e pronto per il tentativo di patch.

```
vncc@node1:~$ kubectl -n resize-demo get pod resize-pod \
  -o jsonpath='{.spec.containers[0].resources}'; echo
{"limits":{"cpu":"100m","memory":"64Mi"},"requests":{"cpu":"50m","memory":"32Mi"}}
vncc@node1:~$ kubectl -n resize-demo get pod resize-pod \
  -o jsonpath='{.status.containerStatuses[0].restartCount}'; echo
0
```

È stato preparato un file di patch (`resize-patch.yaml`) che incrementava in maniera significativa le risorse richieste e i limiti del container. La patch è stata applicata con il comando:

```
kubectl -n resize-demo patch pod resize-pod \
  --type=merge \
  --patch-file=resize-patch.yaml
```

```
The Pod "resize-pod" is invalid:
* spec.containers[0].image: Required value
* spec: Forbidden: pod updates may not change fields other than `spec.containers[*].imag
e`,`spec.initContainers[*].image`,`spec.activeDeadlineSeconds`,`spec.tolerations` (only
additions to existing tolerations),`spec.terminationGracePeriodSeconds` (allow it to be
set to 1 if it was previously negative)
```

Come previsto, il server ha rifiutato l'operazione restituendo un errore di validazione: *"spec: Forbidden: pod updates may not change fields other than ..."*. Questo dettaglio, conferma come la modifica dei campi `resources` non sia consentita nei Pod già creati.

Capitolo 5

Dynamic Resource Allocation (DRA) in Kubernetes 1.33

5.1 Introduzione a Dynamic Resource Allocation

Dynamic Resource Allocation (DRA) permette a Kubernetes di gestire risorse speciali come GPU o FPGA. I driver dichiarano le risorse disponibili e gli utenti le richiedono in modo dichiarativo, tramite le nuove API `resource.k8s.io`.

Il framework introduce alcuni oggetti fondamentali. La `DeviceClass` rappresenta la categoria di un tipo di dispositivo, definita a livello di cluster. La `ResourceSlice` è un oggetto creato dal driver su ciascun nodo worker e descrive la disponibilità effettiva di dispositivi di una determinata classe. La `ResourceClaim` è la richiesta formulata da un utente o da un workload per ottenere una o più risorse appartenenti a una certa classe. Infine, il `Pod` può dichiarare uno o più resource claim nella propria specifica; lo scheduler provvede ad assegnarlo a un nodo con slice compatibili.

Il flusso di DRA prevede che l'amministratore definisca una `DeviceClass`, il driver pubblichi i `ResourceSlice` disponibili, l'utente crei un `ResourceClaim` e infine un Pod lo utilizzi. Lo scheduler assegna il Pod a un nodo compatibile e il driver prepara la risorsa richiesta.

5.2 Verifica delle API DRA

È stato verificato che il gruppo `resource.k8s.io` fosse esposto dal server Kubernetes in versione `v1beta2`, caratteristica di Kubernetes 1.33.

```
kubectl api-versions | grep resource.k8s.io
# Output atteso:
# resource.k8s.io/v1beta2
```

5.3 Creazione del namespace e della DeviceClass

È stato creato il namespace `dra-example-driver` e successivamente è stata applicata la `DeviceClass` definita nel manifest `deviceclass-133.yaml`, compatibile con `v1beta2`.

```
kubectl create ns dra-example-driver
kubectl apply -f deviceclass-133.yaml
kubectl get deviceclasses
```

5.4 Installazione del driver di esempio

Il driver di esempio è stato installato tramite Helm, correggendo i manifest per sostituire `v1beta1` con `v1beta2`. In questo modo è stato distribuito un DaemonSet sul nodo worker.

```
cd ~/dra-example-driver
helm template dra-example-driver deployments/helm/dra-example-driver \
  -n dra-example-driver \
| sed 's#resource.k8s.io/v1beta1#resource.k8s.io/v1beta2#g' \
| kubectl apply -n dra-example-driver -f -
```

Il Pod del driver è risultato in stato `Running` sul nodo `node2` e il cluster ha pubblicato una `ResourceSlice`.

```
kubectl -n dra-example-driver get pods -o wide
kubectl get resourceslices -o wide
```

5.5 Creazione di un ResourceClaim

È stato creato un `ResourceClaim` a partire dal manifest `rc-gpu.yaml`, con la richiesta di un dispositivo appartenente alla DeviceClass `gpu.example.com`.

```
kubectl apply -f rc-gpu.yaml
kubectl -n default get resourceclaim some-gpu -o yaml
```

5.6 Creazione di un Pod che usa il ResourceClaim

È stato definito un Pod, specificato nel manifest `pod-uses-gpu.yaml`, che referenzia il `ResourceClaim` tramite il campo `resourceClaimName`. Tale Pod è stato schedulato correttamente sul nodo `node2` ed è entrato in stato `Running`.

```
kubectl apply -f pod-uses-gpu.yaml
kubectl -n default get pod pod0 -o wide
kubectl -n default describe pod pod0
```

5.7 Risultati finali

Al termine della configurazione, il cluster Kubernetes 1.33 ha mostrato di supportare le API di *Dynamic Resource Allocation* (`v1beta2`). È stata creata una `DeviceClass` chiamata `gpu.example.com`,

resa disponibile tramite un driver di esempio che simulava la presenza di una GPU sul nodo worker. Su questa base è stato definito un `ResourceClaim` (`some-gpu`) e un Pod, configurato per usarlo, è stato correttamente schedulato su `node2` ed è entrato in stato `Running`.

Durante l'esperimento, tuttavia, i log del driver hanno riportato in maniera ricorrente il seguente messaggio:

```
failed registration process: RegisterPlugin error -- no handler registered for plugin
type: DRAPlugin
```

```
vncc@node1:~/dra-example-driver$ kubectl -n dra-example-driver logs -l app.kubernetes.io/name=dra-example-driver --tail=200
I0906 08:41:43.509003       1 health.go:70] "connecting to registration socket" path="unix:///var/lib/kubelet/plugins_regis
try/gpu.example.com-reg.sock"
I0906 08:41:43.510321       1 health.go:83] "connecting to DRA socket" path="unix:///var/lib/kubelet/plugins/gpu.example.co
m/dra.sock"
I0906 08:41:43.512206       1 health.go:103] "starting healthcheck service" addr="[::]:51515"
E0906 08:41:43.653602       1 nonblockinggrpcserver.go:159] "handling request failed" err="failed registration process: Reg
isterPlugin error -- no handler registered for plugin type: DRAPlugin at socket /var/lib/kubelet/plugins_registry/gpu.examp
le.com-reg.sock" logger="registrar" requestID=2 method="/pluginregistration.Registration/NotifyRegistrationStatus"
I0906 08:41:51.525014       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:42:01.515199       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:42:11.519468       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:42:21.544498       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:42:31.507372       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:42:41.523784       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:42:51.522747       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:43:01.518406       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:43:11.536363       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:43:21.521198       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:43:31.517483       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:43:41.543074       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
E0906 08:43:45.911600       1 nonblockinggrpcserver.go:159] "handling request failed" err="failed registration process: Reg
isterPlugin error -- no handler registered for plugin type: DRAPlugin at socket /var/lib/kubelet/plugins_registry/gpu.examp
le.com-reg.sock" logger="registrar" requestID=28 method="/pluginregistration.Registration/NotifyRegistrationStatus"
I0906 08:43:51.509501       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:44:01.525253       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:44:11.511312       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
I0906 08:44:21.513542       1 driver.go:107] PrepareResourceClaims is called: number of claims: 0
```

Il flusso `DeviceClass` → `ResourceSlice` → `ResourceClaim` → `Pod` si è interrotto prima dell'ultimo passo.

Questo risultato ha mostrato che, anche se gli oggetti `DeviceClass`, `ResourceSlice`, `ResourceClaim` e `Pod` sono stati creati correttamente, in Kubernetes 1.33 il plugin DRA non funziona ancora del tutto perché il kubelet non riesce a registrarlo. Le nuove API in versione *beta* esistono e possono essere usate, ma hanno ancora limiti e instabilità.

Capitolo 6

Image Volumes

La funzionalità *Image Volume*, introdotta in Kubernetes v1.33 (beta), permette di montare direttamente un'immagine OCI come volume di sola lettura. In questo modo i contenuti statici diventano subito disponibili nei Pod, senza dover usare ConfigMap di grandi dimensioni o initContainer che copiano file in volumi temporanei. Questo approccio semplifica la gestione di modelli, binari o pacchetti di configurazione da fornire ai container.

6.1 Image Volumes: primo tentativo e limiti riscontrati

Dopo aver verificato che il cluster fosse operativo, il passo successivo è stato abilitare il feature gate necessario agli *Image Volumes*. Sul componente *apiserver* è bastato aggiungere il parametro `-feature-gates=ImageVolume=true` e riavviare il servizio: l'opzione è stata recepita correttamente al primo tentativo.

La situazione è risultata più complessa per il *kubelet*. Inizialmente le modifiche sono state applicate al file `/var/lib/kubelet/config.yaml`, ma non hanno avuto effetto. Analizzando meglio il sistema si è scoperto che il kubelet non leggeva quel file, bensì era gestito da systemd tramite due percorsi differenti: `/etc/kubernetes/kubelet.env` e `/etc/kubernetes/kubelet-config.yaml`. Solo dopo aver individuato questi file è stato possibile intervenire nei punti corretti e riavviare il servizio in modo che il feature gate risultasse attivo.

Una volta abilitata la funzionalità lato configurazione, è stato creato un Pod di test con il manifest `pod-image-volume.yaml`. In questo Pod era dichiarato un volume di tipo immagine basato su `quay.io/crio/artifact:v2`, montato sul percorso `/volume`. Il Pod è partito senza errori ed è stato mostrato come *Running*, ma entrando al suo interno i comandi di ispezione (`ls`, `cat`) non mostravano né la directory `/volume` né i file previsti. Anche il comando `kubectl describe pod` non riportava anomalie, segno che dal punto di vista dell'API server la richiesta era stata accettata e il Pod era stato schedulato correttamente.

Il problema risiedeva quindi non nell'API, ma nel livello di runtime. In Kubernetes, l'API server accetta la dichiarazione di un volume di tipo immagine se il feature gate è attivo, ma il montaggio effettivo dipende dal supporto implementato dal runtime conforme alla Container Runtime Interface (CRI). Nel nostro caso il cluster usava `containerd 2.1.4`, che nelle build standard non supporta ancora gli *Image Volumes*. Di conseguenza, il kubelet non è stato in grado di istruire containerd a esporre un filesystem derivato dall'immagine, e il punto di mount è rimasto inesistente. Il risultato pratico è che il Pod appare in esecuzione senza errori, ma i file attesi non compaiono e i comandi restituiscono `No such file or directory`.

6.2 Soluzione con initContainer ed emptyDir

Una possibile soluzione, già ampiamente diffusa nelle versioni precedenti di Kubernetes, si fonda sull'uso congiunto di un initContainer e di un volume di tipo `emptyDir`. In questo modo si aggira la dipendenza dal supporto nativo del runtime, affidandosi ad una modalità pienamente supportata.

Prima è stato definito il manifest `pod-image-volume-fallback.yaml`, che dichiara un volume `emptyDir` condiviso da due container: l'initContainer e il container applicativo. L'initContainer viene eseguito all'avvio del Pod e ha il compito di creare la directory `/data` e di scrivere al suo interno un file denominato `hello.txt`. Questo avviene utilizzando il filesystem nativo del container di inizializzazione, che è sempre disponibile e non richiede alcun supporto particolare da parte del runtime. Il container applicativo monta lo stesso volume e accede al file prodotto, riproducendo così lo scenario che l'Image Volume avrebbe dovuto gestire automaticamente.

Sono stati applicati i comandi:

```
kubectl apply -f pod-image-volume-fallback.yaml
kubectl wait --for=condition=ready pod/image-volume-fallback --timeout=120s
kubectl logs image-volume-fallback
```

L'output ha mostrato la stringa `Hello from image!`, confermando che il file era stato correttamente generato e letto dal container applicativo. In maniera facoltativa, è stato possibile verificare la presenza fisica del file all'interno del container con:

```
kubectl exec -it image-volume-fallback -- ls -l /data
kubectl exec -it image-volume-fallback -- cat /data/hello.txt
```

La differenza tra i due approcci è la seguente:

Con l'*Image Volume* nativo il Pod dichiara un volume di tipo `image:` e il runtime dovrebbe montare i layer come filesystem in sola lettura, ma con `containerd 2.1.4` il mount non avviene e il percorso `/volume` resta vuoto. Con il fallback invece si usa un volume `emptyDir`, gestito direttamente dal kubelet e sempre disponibile: l'`initContainer` popola il volume copiando i file nel percorso `/data`, che il container applicativo può poi leggere.

In questo modo non servono feature sperimentali del runtime, i file vengono condivisi correttamente tra i container del Pod e l'esperimento si conclude con successo (`/data/hello.txt` presente e leggibile).

6.3 Endpoints vs EndpointSlice

File manifest e di output

demo-busybox.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-busybox
spec:
  replicas: 60
```

```
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
      - name: bb
        image: busybox:stable
        imagePullPolicy: IfNotPresent
        command: ["sh","-c","sleep infinity"]
        resources:
          requests:
            cpu: "5m"
            memory: "8Mi"
          limits:
            cpu: "20m"
            memory: "32Mi"
```

6.3.1 demo-service.yaml

demo-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
spec:
  type: ClusterIP
  selector:
    app: demo
  ports:
```

```
  - port: 80
    targetPort: 80
```

6.4 Sidecar

cm-index-html.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-index
  namespace: sidecar-132
data:
  index.html: |
    <!doctype html>
    <html>
      <head><meta charset="utf-8"><title>Demo Sidecar Logs</title></head>
      <body>
        <h1>Hello from Nginx + Sidecar (K8s 1.32)</h1>
        <p>Refresh a few times to generate access logs.</p>
      </body>
    </html>
```

cm-nginx-conf.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-nginx-conf
  namespace: sidecar-132
data:
  nginx.conf: |
    events {}
    http {
```

```
    include       mime.types;
    default_type  application/octet-stream;

    # Log su volume condiviso
    access_log /var/log/nginx/access.log;
    error_log  /var/log/nginx/error.log warn;

    sendfile        on;
    keepalive_timeout  65;

    server {
      listen 80;
      server_name _;

      root /usr/share/nginx/html;
      index index.html;

      location / {
        try_files $uri $uri/ =404;
      }
    }
  }
```

web-with-sidecar-132.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-sidecar-132
  namespace: sidecar-132
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: websidecar
  template:
    metadata:
      labels:
        app: websidecar
    spec:
      volumes:
        - name: html
          configMap:
            name: demo-index
        - name: nginx-conf
          configMap:
            name: demo-nginx-conf
            items:
              - key: nginx.conf
                path: nginx.conf
        - name: logs
          emptyDir: {}
      containers:
        - name: nginx
          image: nginx:1.25-alpine
          ports:
            - containerPort: 80
          volumeMounts:
            - name: html
              mountPath: /usr/share/nginx/html
            - name: nginx-conf
              mountPath: /etc/nginx/nginx.conf
              subPath: nginx.conf
            - name: logs
              mountPath: /var/log/nginx
        - name: log-reader
          image: busybox:stable
```

```
          command: ["sh","-c","tail -F /var/log/nginx/access.log"]
          volumeMounts:
            - name: logs
              mountPath: /var/log/nginx
---
apiVersion: v1
kind: Service
metadata:
  name: web-sidecar-svc
  namespace: sidecar-132
spec:
  type: NodePort
  selector:
    app: websidecar
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
```

cm-index-html-133.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-index
  namespace: sidecar-133
data:
  index.html: |
    <!doctype html>
    <html>
      <head><meta charset="utf-8"><title>Demo Sidecar Logs (1.33)</title></head>
      <body>
        <h1>Hello from Nginx + Native Sidecar (K8s 1.33)</h1>
        <p>Refresh a few times to generate access logs.</p>
```

```
      </body>
    </html>
```

cm-nginx-conf-133.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-nginx-conf
  namespace: sidecar-133
data:
  nginx.conf: |
    events {}
    http {
      include       mime.types;
      default_type  application/octet-stream;

      # Log su volume condiviso
      access_log /var/log/nginx/access.log;
      error_log  /var/log/nginx/error.log warn;

      sendfile        on;
      keepalive_timeout  65;

      server {
        listen 80;
        server_name _;

        root /usr/share/nginx/html;
        index index.html;

        location / {
          try_files $uri $uri/ =404;
        }
```

```
      }
    }
```

web-with-sidecar-133.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-sidecar-133
  namespace: sidecar-133
spec:
  replicas: 1
  selector:
    matchLabels:
      app: websidecar133
  template:
    metadata:
      labels:
        app: websidecar133
    spec:
      volumes:
        - name: html
          configMap:
            name: demo-index
        - name: nginx-conf
          configMap:
            name: demo-nginx-conf
            items:
              - key: nginx.conf
                path: nginx.conf
        - name: logs
          emptyDir: {}
      initContainers:
        - name: log-reader
```

```
          image: busybox:stable
          restartPolicy: Always
          command: ["sh","-c","tail -F /var/log/nginx/access.log"]
          volumeMounts:
            - name: logs
              mountPath: /var/log/nginx
      containers:
        - name: nginx
          image: nginx:1.25-alpine
          ports:
            - containerPort: 80
          volumeMounts:
            - name: html
              mountPath: /usr/share/nginx/html
            - name: nginx-conf
              mountPath: /etc/nginx/nginx.conf
              subPath: nginx.conf
            - name: logs
              mountPath: /var/log/nginx
---
apiVersion: v1
kind: Service
metadata:
  name: web-sidecar-133-svc
  namespace: sidecar-133
spec:
  type: NodePort
  selector:
    app: websidecar133
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30081
```

6.5 In-place Pod Resize

resize-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resize-pod
  namespace: resize-demo
  labels:
    app: resize
spec:
  containers:
  - name: app
    image: busybox:stable
    command: ["sh","-c","sleep infinity"]
    resources:
      requests:
        cpu: "50m"
        memory: "32Mi"
      limits:
        cpu: "100m"
        memory: "64Mi"
```

resize-patch.yaml

```yaml
spec:
  containers:
  - name: app
    resources:
      requests:
        cpu: "200m"
        memory: "128Mi"
      limits:
        cpu: "300m"
```

```
        memory: "192Mi"
```

resize-pod-132.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: resize-pod
  namespace: resize-demo
  labels:
    app: resize
spec:
  containers:
  - name: app
    image: busybox:stable
    command: ["sh","-c","sleep infinity"]
    resources:
      requests:
        cpu: "50m"
        memory: "32Mi"
      limits:
        cpu: "100m"
        memory: "64Mi"
```

6.6 OCI

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","set -e; mkdir -p /data; echo 'Hello from image!' >
    ↪  /data/hello.txt"]
    volumeMounts:
    - name: data
      mountPath: /data
  containers:
  - name: app
    image: busybox
    command: ["sh","-c","cat /data/hello.txt; sleep 3600"]
    volumeMounts:
    - name: data
      mountPath: /data
```

6.7 Dynamic Resource Allocation

deviceclass-133.yaml

```
apiVersion: resource.k8s.io/v1beta2
kind: DeviceClass
metadata:
  name: gpu.example.com
spec:
  selectors:
  - cel:
      expression: 'true'
```

rc-gpu.yaml

```
apiVersion: resource.k8s.io/v1beta2
```

```yaml
kind: ResourceClaim
metadata:
  name: some-gpu
  namespace: default
spec:
  devices:
    requests:
    - name: one-gpu
      exactly:
        deviceClassName: gpu.example.com
        count: 1
```

pod-uses-gpu.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod0
  namespace: default
spec:
  resourceClaims:
  - name: gpu
    resourceClaimName: some-gpu
  containers:
  - name: app
    image: busybox:stable
    command: ["sh","-c","echo hello DRA && sleep 3600"]
```