Tesina

Virtual Networks and Cloud Computing

Corso di Laurea in Ingegneria Informatica e Robotica – A.A. 2024-2025

Dipartimento di Ingegneria

docente

Prof. Gianluca Reali

Deployment di un servizio di classificazione in Kubernetes con l'utilizzo della libreria Python client

studenti

Giuseppe Gallo giuseppe.gallo2@studenti.unipg.it

Data ultimo aggiornamento: 4 febbraio 2025

Indice

Capitolo 1

Installazione di Kubernetes e configurazione del cluster

1.1 Configurazione della rete

Come primo passo, bisogna fare in modo che ogni macchina virtuale abbia un indirizzo IP statico per far si che siano sempre raggiungibili l'una con l'altra. Si cambia dunque il file di configurazione di rete `/etc/netplan/01-network-manager-all.yaml` su entrambi i nodi:

```
network:
version: 2
renderer: NetworkManager
ethernets:
enp0s3:
dhcp4: no
addresses: [<ip-nodo>]
gateway4: 192.168.43.1
nameservers:
addresses: [8.8.8.8, 8.8.4.4]
```

Poi si attiva la configurazione:

```
sudo netplan apply
```

Si modificano poi i nomi degli host modificando il file `/etc/hostname`.
Per il nodo master:

```
node1
```

Per il nodo worker:

```
node2
```

Per ultimo si modifica il file `/etc/hosts` su entrambe le macchine per consentire la risoluzione degli indirizzi host tramite alias. Si aggiungono le seguenti righe:

```
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
```

1.2 Configurazione dell'ambiente

Si configurano:

- il file /etc/host delle due macchine al fine di associare un nome di dominio rispetto al loro indirizzo ip
- il file /etc/netplan/01-network-manager-all.yaml

Occorre poi installare SSH su entrambi i nodi al fine di installare Kubernetes tramite Ansible. Per installarlo, eseguire il seguente comando su entrambi i nodi:

```
sudo apt install openssh-server
```

Successivamente si creano le chiavi su entrambi gli host in modo tale da permettere la connessione senza password ma attraverso una coppia di chiavi pubblica/privata. Si esegue il seguente comando su entrambi gli host.

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

Dopo la generazione bisogna fare in modo che gli host si scambino la chiave. Su entrambi gli host eseguire i seguenti comandi:

```
ssh-copy-id 192.168.43.10
ssh-copy-id 192.168.43.11
```

Durante l'esecuzione del playbook, Ansible richiede privilegi root. Oltre a questo è necessario che non venga richiesta la password per le operazioni root. Bisogna quindi modificare il file `/etc/sudoers` nel seguente modo:

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

E' necessario in ultimo installare Python, poichè Ansible lo richiede:

```
sudo apt install python3-pip
sudo pip3 install --upgrade pip
```

1.3 Installazione di Kubernetes tramite Kubespray

Kubespray è uno strumento che automatizza l'installazione di Kubernetes e del cluster utilizzando un playbook Ansible. Come primo step, scaricare la repository da github:

```
sudo apt install git
git clone https://github.com/kubernetes-sigs/
   ↪ kubespray.git
cd kubespray
```

Al momento della scrittura della tesina l'ultima versione disponibile è la versione 2.27.

E' consigliabile installare Ansible e le sue dipendenze all'interno di un ambiente virtuale Python, in modo da non entrare in contrasto con le eventuali versioni già presenti sulla macchina host. Sul nodo master eseguire:

```
VENVDIR=kubespray-venv
KUBESPRAYDIR=kubespray
python3 -m venv $VENVDIR
source $VENVDIR/bin/activate
cd $KUBESPRAYDIR
pip install -U -r requirements.txt
```

Nella versione attuale non è più presente l'eseguibile python con cui creare automaticamente il file di configurazione iniziale del cluster che Ansible usa per istanziare i nodi sulle macchine corrette. E' necessario prima creare la cartella `mycluster` copiando la cartella guida `sample`:

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

All'interno di `mycluster` modificare il file `inventory.ini` nel seguente modo:

```
[kube_control_plane]
 node1 ansible_host=192.168.43.10

[etcd:children]
```

```
kube_control_plane

[kube_node]
 node2 ansible_host=192.168.43.11
```

Si modifica il file `inventory/mycluster/group_vars/all/all.yaml`:

```
upstream_dns_servers:
- 8.8.8.8
- 8.8.4.4
```

Prima di eseguire il playbook è bene eseguire un reset per verificare che tutto sia installato e pronto all'uso, dato che l'esecuzione del playbook di configurazione richiede abbastanza tempo.

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

E eseguire successivamente il playbook:

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

Finita l'installazione occorre installare kubectl sul nodo master per la gestione del cluster:

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

ed eseguire:

```
sudo kubectl get nodes
```

e verificare che i due nodi siano stati creati e siano pronti all'uso:

```
NAME STATUS ROLES AGE VERSION
node1 Ready control-plane 2d2h v1.32.0
node2 Ready <none> 2d2h v1.32.0
```

Infine, per evitare di utilizzare sempre `sudo` per interagire con kubectl, eseguire i seguenti comandi:

```
mkdir .kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.
   ↪ kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Capitolo 2

Deployment

Si vuole ora creare un'applicazione che addestri un semplice modello di ML attraverso un Job, salvi questi dati in un Volume condiviso in modo tale da salvare i pesi del modello e creare un altro pod che usi questi pesi per fare predizioni. Il pod verrà dotato di un servizio capace di gestire richieste HTTP e esposto all'esterno del cluster tramite un Service, in modo da poter effettuare richieste dal terminale Windows dell'host.

Come primo approccio si mostra come istanziare il job per l'addestramento in maniera dichiarativa utilizzando kubectl e il file yaml.

2.1 Creazione del Job tramite yaml

Come prima cosa si scrive l'eseguibile Python per l'addestramento del modello. In fase di build di utilizzerà un immagine Jupyter che ha all'interno tutte le librerie necessarie per l'addestramento. Al suo interno verrà copiato lo script in modo da poterlo eseguire quando il job verrà creato.

Si crea quindi il Dockerfile:

Dockerfile

```
FROM jupyter/scipy-notebook

COPY train.py ./train.py
```

E lo script per l'addestramento:

train.py

```
from sklearn import svm
from sklearn import datasets
from joblib import dump, load

clf = svm.SVC()
X, y = datasets.load_iris(return_X_y=True)
clf.fit(X,y)

print("Model finished training")

dump(clf, 'svc_model.model')
```

Occorre creare un registro locale in modo tale che Kubernetes abbia accesso all'immagine.

Al momento della creazione del job, essa verrà scaricata dal registro locale del nodo worker. Si crea quindi tale registro sulla porta 5000 del nodo worker con il comando:

```
docker run -d -p 5000:5000 --name beppe-registry
    ↪ registry:2
```

Dalla cartella dove si trovano lo script Python e il dockerfile si esegue poi il build dell'immagine. All'interno del nome, occorre inserire il prefisso `localhost:5000` per indicare a Docker che l'immagine dovrà essere caricata sul registro appena creato.

```
docker build -t localhost:5000/ml-training .
```

Si esegue quindi il push dell'immagine sul registro locale:

```
docker push localhost:5000/ml-training
```

Per verificare che l'immagine sia stata caricata correttamente:

```
curl http://localhost:5000/v2/ml-training/tags/
    ↪ list
```

che restituisce:

```
{"name":"ml-training","tags":["latest"]}
```

Si crea infine lo yaml in modo tale da istanziare il job in maniera dichiarativa:

job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: ml-training-job
spec:
  template:
    containers:
    - name: ml-training
      image: localhost:5000/ml-training:latest
      command: ["python3","/home/jovyan/train.py"]
      resources:
        requests:
          memory: "4Gi"
          cpu: "1"
        limits:
          memory: "6Gi"
          cpu:"2"
    restartPolicy: Never
backoffLimit: 4
```

Dalla cartella dove è presente lo yaml si esegue:

```
kubectl -f apply job.yaml
```

E' possibile osservare la corretta riuscita del comando con `kubectl get pods`:

```
studente@node1:~/Progetto/ml_training_job$ kubectl get pods
NAME                    READY   STATUS      RESTARTS   AGE
ml-training-job-v4n6b   0/1     Completed   0          30s
```

E' possibile anche verificare che il modello ha completato il training dai log del pod:

```
studente@node1:~/Progetto/ml_training_job$ kubectl logs ml-training-job-v4n6b
Model finished training...
```

2.2 Libreria Python client

La libreria Python client di Kubernetes permette di configurare e interagire con il cluster attraverso script Python. Ciò può essere utile per automatizzare operazioni all'interno del cluster, come la creazione di pod, job, service ecc. oppure gestire

gli accessi a determinate risorse, creare oggetti custom o gestire lo scaling delle applicazioni.

Di seguito una spiegazione delle varie classi presenti all'interno della libreria:

- **AppsV1Api**: utilizzata per gestire risorse relative a controller e workload
- **ApiextensionsV1Api**: permette di creare e gestire risorse personalizzate tramite Custom Resource Definitions (CRD) oltre ai classici pod, job, deployment ecc. A differenza di CustomObjectsApi gestisce la **definizione** di nuovi tipi di risorse.
- **AuthenticationApi**: gestisce i meccanismi di autenticazione per le richieste API. Permette di verificare e gestire l'accesso agli endpoint Kubernetes, supportando vari metodi di autenticazione
- **AuthorizationApi**: gestisce la parte relativa all'autorizzazione delle richieste API. Mentre l'AuthenticationApi verifica l'identità di chi effettua la richiesta (ne effettua quindi l'autenticazione), AuthorizationApi verifica che l'entità autenticata abbia i permessi per eseguire le operazioni su una determinata risorsa del cluster
- **AutoscalingApi**: gestisce la scalabilità automatica dei pod in base alle risorse disponibili
- **batchV1api**: per lavorare con le risorse relative a job e cronjob nel cluster
- **CertificatesApi**: utilizzata per gestire e interagire con i certificati SSL/-TLS all'interno del cluster. Gestisce la creazione e il rinnovo dei certificati necessari per garantire la comunicazione sicura tra i componenti di Kubernetes.
- **CoordinationApi**: utilizzata per gestire meccanismi di coordinamento e sincronizzazione tra i vari componenti del cluster
- **CoreApi**: una delle API più importanti poichè permette di interagire con tutte le risorse fondamentali del cluster (Pod, Service, Namespace, ConfigMap, Secret, ReplicationController ecc.)
- **CustomObjectsApi**: gestisce l'uso effettivo delle risorse personalizzate. E' possibile quindi creare, eliminare e modificare risorse definite tramite CRD con le ApiExtensionsV1Api.
- **DiscoveryApi**: consente di scoprire e ottenere informazioni sui servizi e sulle risorse disponibili nel cluster. Interagisce con l'API server per ottenere la lista delle risorse supportate e informazioni relative ad esse.

- **EventsApi**: consente di accedere e gestire gli eventi generati all'interno del cluster, ovvero messaggi di log che descrivono azioni o cambiamenti nel cluster (creazione di pod, problemi con i container, risorse non sufficienti ecc.)
- **FlowcontrolApiserverApi**: il flow control garantisce che le richieste ai server API siano trattate in modo ordinato e rispettino i limiti di risorse. Questa API consente di gestire le politiche di controllo del flusso di dati, come la gestione delle code delle richieste in base alla loro priorità.
- **InternalApiserverApi**: serve per interagire con le API interne di Kubernetes, ovvero utilizzate dai componenti per comunicare fra di loro e per gestire risorse interne.
- **NetworkingApi**: gestisce le risorse di rete come i Services, gli Ingress, i LoadBalancer e le reti virtuali.
- **NodeApi**: interagisce con i nodi per ottenerne una lista, informazioni specifiche, gestirne la disponibilità e/o lo scaling ecc.
- **PolicyApi**: fornisce metodi per interagire con le policy di sicurezza, gestione del flusso di rete e altre configurazioni che influenzano il comportamento e la sicurezza del cluster
- **RbacAuthorizationApi**: utilizzato per gestire le risorse relative al controllo dell'accesso basato su ruoli (RBAC, Role-Based Access Control)
- **ResourceApi**: gestisce l'utilizzo delle risorse fisiche da parte del cluster
- **SchedulingApi**: si occupa della gestione e pianificazione dei pod, ovvero il processo che decide su quale nodo un pod debba essere eseguito tenendo conto delle risorse disponibili e delle restrizioni di configurazione.
- **StorageApi**: permette di interagire con il sistema di archiviazione di Kubernetes, come la gestione di volumi di storage persistenti (PV e PVC)

2.3 Utilizzo della libreria

Nel contesto del progetto, la libreria verrà utilizzata per istanziare il servizio di classificazione (e il relativo service) e il volume condiviso con il job precedentemente creato. Per completezza si effettuerà l'intero deployment tramite un unico script includendo anche il job.

Come primo passo, occorre installare la libreria tramite il comando:

```
pip install kubernetes
```

Dopo aver effettuato l'accesso come root, i file di configurazione di Kubernetes saranno situati in `/etc/kubernetes/admin.conf`. Essi saranno necessari per poter comunicare con il cluster attraverso il client Python di Kubernetes. E' possibile utilizzare la libreria per verificare la configurazione e che i nodi siano accessibili:

```python
from kubernetes import client, config

# Carica la configurazione dal file kubeconfig
config.load_kube_config()

# Crea un oggetto API per interagire con il server Kubernetes
v1 = client.CoreV1Api()

# Ottieni la lista dei nodi
nodes = v1.list_node()

# Stampa informazioni sui nodi
for node in nodes.items:
    print(f"Nome nodo: {node.metadata.name}")
    print(f"Status: {node.status.conditions[-1].type}")
    print(f"Versione Kubernetes: {node.status.node_info.
        ↪ kubelet_version}")
    print("------")
```

che darà come output:

```
Nome nodo: node1
Status: Ready
Versione Kubernetes: v1.32.0
------
Nome nodo: node2
Status: Ready
Versione Kubernetes: v1.32.0
------
```

2.3.1 Creazione del deployment

Come già detto, tramite le API si vuole creare un deployment costituito da:

- un **Job** che addestri un algoritmo di ML e salvi i risultati all'interno di un volume condiviso

- un **Pod** che prenda dal volume i pesi generati dall'addestramento e li utilizzi per fare predizioni. All'interno del Pod verrà implementato un servizio Flask per gestire richieste HTTP di classificazione provenienti dall'esterno del cluster (in questo caso dell'OS host)
- un **Service** di tipo NodePort in modo da esporre il Pod all'esterno del cluster. In questo caso siamo a conoscenza dell'IP del nodo e siamo sicuri che sia un nodo healty. Non è quindi necessario utilizzare altri tipi di Service.
- un **Volume** necessario per mantenere i dati dell'addestramento al terminamento del Job e passarli correttamente al Pod di classificazione
- una **ConfigMap** per montare il file python contenente il servizio di classificazione all'interno del Pod durante la sua istanziazione. In questo modo sarà possibile apportare eventuali modifiche al file senza dover rifare il build dell'immagine docker.

2.3.2 Deployment tramite Python client

Assicurandosi che l'immagine per l'istanziazione del job sia disponibile sul registro locale, si può scrivere lo script per eseguire il deployment. Il file verrà diviso per una migliore analisi.

- **Inizializzazione delle librerie e setup iniziale** Vengono prima caricate le librerie necessarie e si carica la configurazione del cluster. Successivamente si crea un istanza delle CoreApi per la gestione di Volumi, Pod e ConfigMap e un'istanza del BatchApi per la gestione del Job.

```
from kubernetes import client, config
from kubernetes.client.rest import ApiException
import time

#Caricamento della configurazione del cluster
config.load_kube_config()

# Inizializzazione dei client API
v1 = client.CoreV1Api()
batch_v1 = client.BatchV1Api()
```

- **Creazione del Volume e del VolumeClaim** Viene creato successivamente un Volume e un Volume Claim (ovvero una richiesta di spazio all'interno di un volume) al fine di salvare i dati del Job.

```
# Creazione del Persistent Volume (PV)
```

```
try:
    v1.read_persistent_volume(name="beppe-pv")
    print(f"Il PersistentVolume 'beppe-pv' esiste gi")
except ApiException as e:
    if e.status == 404: # Non trovato, possiamo crearlo
        print(f"Il PersistentVolume 'beppe-pv' non esiste, lo
            ↪ creeremo.")

pv = client.V1PersistentVolume(
    api_version="v1",
    kind="PersistentVolume",
    metadata=client.V1ObjectMeta(name="beppe-pv"),
    spec=client.V1PersistentVolumeSpec(
        capacity={"storage": "1Gi"},
        access_modes=["ReadWriteOnce"],
        persistent_volume_reclaim_policy="Retain",
        host_path=client.V1HostPathVolumeSource(path="/
            ↪ home/studente/Scrivania")
    )
)

v1.create_persistent_volume(body=pv)
print("Persistent Volume creato.")




# Creazione del Persistent Volume Claim (PVC)
try:
    v1.read_namespaced_persistent_volume_claim(name="ml-
        ↪ data-pvc",namespace="default")
    print(f"Il PersistentVolumeClaim 'ml-data-pvc' esiste gi.")
except ApiException as e:
    if e.status == 404: # Non trovato, possiamo crearlo
        print(f"Il PersistentVolumeClaim 'ml-data-pvc' non
            ↪ esiste, verr creato.")
pvc = client.V1PersistentVolumeClaim(
    api_version="v1",
    kind="PersistentVolumeClaim",
    metadata=client.V1ObjectMeta(name="ml-data-pvc"),
    spec=client.V1PersistentVolumeClaimSpec(
        access_modes=["ReadWriteOnce"],
```

```
            resources=client.V1ResourceRequirements(
                requests={"storage":"300Mi"}
            )
        )
)

v1.create_namespaced_persistent_volume_claim(namespace="
    ↪ default",body=pvc)
print("Persistent Volume Claim creato.")
```

- **Creazione del Job** Viene poi creato lo script inserendo all'interno il Volume e il Volume Claim. Quest'ultimo viene montato in */home/volume*. Alla creazione del container si esegue il train con il comando **"command=["python3", "/home/jovyan/train.py"]"**. Inoltre, si imposta la restart policy su "OnFailure" in modo che riparta in caso di fallimento e il numero massimo di tentativi su "backoff_limit=4".

```
# Verifica se il job esiste gi
try:
    # Prova a ottenere il job
    batch_v1.read_namespaced_job(name="ml-training-job",
        ↪ namespace="default")
    print(f"Il job 'ml-training-job' esiste gi. Verr eliminato e
        ↪ rilanciato.")

    # Elimina il job se esiste
    batch_v1.delete_namespaced_job(name="ml-training-job
        ↪ ", namespace="default")
    print(f"Il job 'ml-training-job'  stato eliminato.")
    # Aspetta che il job sia effettivamente eliminato
    while True:
        try:
            batch_v1.read_namespaced_job(name="ml-
                ↪ training-job", namespace="default")
            time.sleep(1) # Aspetta 1 secondo e riprova
        except ApiException as e:
            if e.status == 404:
                print(f"Il job  stato eliminato con successo.")
                break
except ApiException as e:
    if e.status == 404:
        print(f"Il job 'ml-training-job' non esiste. Verr creato
            ↪ un nuovo job.")
```

```
    else:
        print(f"Errore durante il controllo del job: {e}")


# Funzione per la creazione di un Job
job = client.V1Job(
    metadata=client.V1ObjectMeta(name="ml-training-job"),
    spec=client.V1JobSpec(
        template=client.V1PodTemplateSpec(
            metadata=client.V1ObjectMeta(labels={"app":"ml-
                ↪ training"}),
            spec=client.V1PodSpec(
                containers=[
                    client.V1Container(
                        name="trainer",
                        image="localhost:5000/ml-training:
                            ↪ latest",
                        command=["python3", "/home/jovyan/
                            ↪ train.py"],
                        volume_mounts=[
                            client.V1VolumeMount(
                                mount_path="/home/volume",
                                name="beppe-pv",
                            )
                        ]
                    )
                ],
                restart_policy="OnFailure",
                volumes=[
                    client.V1Volume(
                        name="beppe-pv",
                        persistent_volume_claim=client.
                            ↪ V1PersistentVolumeClaimVolumeSource
                            ↪ (
                            claim_name="ml-data-pvc"
                        ),
                    )
                ]
            )
        ),
        backoff_limit=4, # Tentativi massimi di fallimento
    )
)
```

```
batch_v1.create_namespaced_job(namespace="default",body=
    ↪ job)
```

- **Creazione della ConfigMap** E' necessario creare una ConfigMap in modo da iniettare lo script contenente l'applicazione che si occupa di fare predizione e rispondere alle richieste HTTP all'interno del Pod al momento della creazione dello stesso.

```
# Creazione della ConfigMap
CONFIGMAP_NAME = "classifier-py-configmap"
APP_PATH = "/home/studente/Progetto/prediction_app/
    ↪ prediction-app.py"

try:
    v1.read_namespaced_config_map(name=
        ↪ CONFIGMAP_NAME, namespace="default")
    print(f"La ConfigMap 'classifier-py' esiste gia'.")
except ApiException as e:
    if e.status == 404:
        print(f"La ConfigMap 'classifier-py' non esiste, verra'
            ↪ creata.")
        config_map = client.V1ConfigMap(
            metadata=client.V1ObjectMeta(name=
                ↪ CONFIGMAP_NAME),
            data={"prediction-app.py": open(APP_PATH ).
                ↪ read()} # Carica il contenuto del file
                ↪ prediction-app.py
        )
        v1.create_namespaced_config_map(namespace="default
            ↪ ", body=config_map)
        print("ConfigMap creata.")
```

- **Creazione del Pod** All'istanziazione del pod si installa flask e si esegue lo script "prediction-app.py". In questo caso si utilizza sempre l'immagine "scipy-notebook" come per il job. E' possibile notare come venga montato sia il Volume contenente i dati del train (in */home/volume* sia la ConfigMap contente l'eseguibile per la classificazione (in */app*). La ConfigMap viene montata come un Volume.

```
try:
    # Prova a leggere il Pod
    v1.read_namespaced_pod(name="classifier-pod",
        ↪ namespace="default")
```

```
    print(f"Il Pod 'classifier-pod' esiste gia'. Verra' eliminato e
        ↪ ricreato.")

    # Elimina il Pod se esiste
    v1.delete_namespaced_pod(name="classifier-pod",
        ↪ namespace="default")
    print(f"Il Pod 'classifier-pod' e' stato eliminato.")
    # Aspetta che il pod sia effettivamente eliminato
    while True:
        try:
            v1.read_namespaced_pod(name="classifier-pod",
                ↪ namespace="default")
            time.sleep(1) # Aspetta 1 secondo e riprova
        except ApiException as e:
            if e.status == 404:
                print(f"Il pod e' stato eliminato con successo.")
                break
except ApiException as e:
    if e.status == 404: # Il Pod non esiste, quindi possiamo
        ↪ crearne uno nuovo
        print(f"Il Pod 'classifier-pod' non esiste. Verra' creato
            ↪ un nuovo Pod.")
    else:
        print(f"Errore durante il controllo del Pod: {e}")

# Creazione del Pod che monta ConfigMap e PVC
pod = client.V1Pod(
    metadata=client.V1ObjectMeta(
        name="classifier-pod",
        labels={"app": "classifier-pod"}
        ),
    spec=client.V1PodSpec(
        containers=[
            client.V1Container(
                name="classifier",
                image="jupyter/scipy-notebook:latest",
                command=["sh", "-c", "pip install flask &&
                    ↪ python3 /app/prediction-app.py"],
                volume_mounts=[
                    client.V1VolumeMount(
                        mount_path="/app",
                        name="config-volume"
                    ),
```

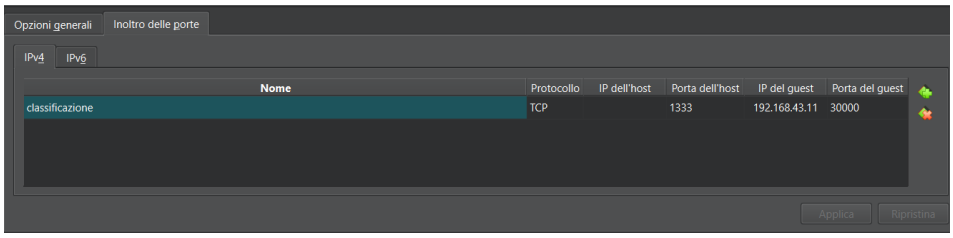
```
                    client.V1VolumeMount(
                        mount_path="/home/volume",
                        name="beppe-pv"
                    )
                ]
            )
        ],
        volumes=[
            client.V1Volume(
                name="config-volume",
                config_map=client.V1ConfigMapVolumeSource(
                    ↪ name="classifier-py-configmap")
            ),
            client.V1Volume(
                name="beppe-pv",
                persistent_volume_claim=client.
                    ↪ V1PersistentVolumeClaimVolumeSource(
                    ↪ claim_name="ml-data-pvc")
            )
        ],
        restart_policy="Never"
    )
)

v1.create_namespaced_pod(namespace="default", body=pod)
print("Pod creato con successo.")
```

- Infine, si istanzia un **Service** di tipo NodePort che permette di esporre l'applicazione all'esterno del cluster. In questo caso si mappa sulla porta 30000 dell'host.

```
# Definizione del Service
service = client.V1Service(
    api_version="v1",
    kind="Service",
    metadata=client.V1ObjectMeta(name="classifier-service"),
    spec=client.V1ServiceSpec(
        selector={"app": "classifier-pod"},
        ports=[
            client.V1ServicePort(
                protocol="TCP",
                port=5000, # Porta del servizio
                target_port=5000, # Porta target del pod
```

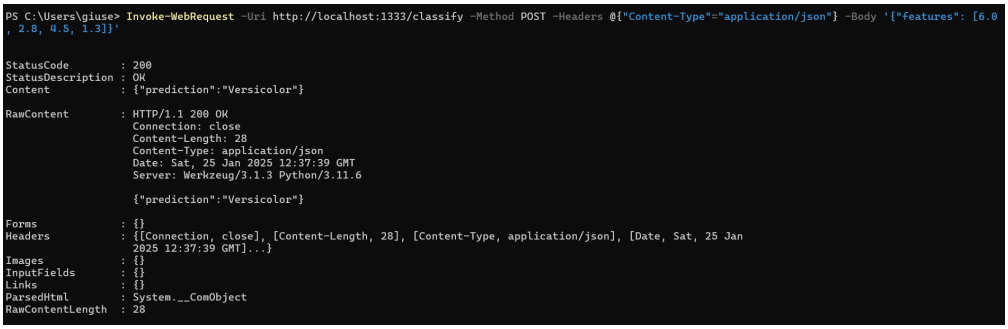
```
                    node_port=30000 # Porta esposta sul nodo
                )
            ],
            type="NodePort"
        )
)

# Creazione del Service nel namespace "default"
v1.create_namespaced_service(namespace="default", body=
    ↪ service)
print("Service creato con successo")
```

Per fare in modo che la porta 30000 sia correttamente raggiungibile dall'esterno bisogna fare in modo che VirtualBox la esponga. Essa è stata mappata con la porta 1333 dell'host, in modo da poter fare richieste all'indirizzo **http://localhost:1333/classify**



2.3.3 Test del funzionamento

Si effettua una richiesta POST da terminale Windows verso il cluster per verificare che la risposta sia corretta.



```
PS C:\Users\giuse> Invoke-WebRequest -Uri http://localhost:1333/classify -Method POST -Headers @{"Content-Type"="application/json"} -Body '{"features": [20.
1, 10.5, 10.4, 6.2]}'

                                                                                                                    StatusCode        : 200
                                                                                                 StatusDescription : OK
                                                Content           : {"prediction":"Virginica"}

RawContent        : HTTP/1.1 200 OK
                    Connection: close
                    Content-Length: 27
                    Content-Type: application/json
                    Date: Sat, 25 Jan 2025 12:36:29 GMT
                    Server: Werkzeug/3.1.3 Python/3.11.6

                    {"prediction":"Virginica"}

Forms             : {}
Headers           : {[Connection, close], [Content-Length, 27], [Content-Type, application/json], [Date, Sat, 25 Jan
                    2025 12:36:29 GMT]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 27
```

```
PS C:\Users\giuse> Invoke-WebRequest -Uri http://localhost:1333/classify -Method POST -Headers @{"Content-Type"="application/json"} -Body '{"features": [20.
1, 6.5, 10.4, 6.2]}'

                                                                                                                    StatusCode        : 200
                                                                                                 StatusDescription : OK
                                                Content           : {"prediction":"Virginica"}

RawContent        : HTTP/1.1 200 OK
                    Connection: close
                    Content-Length: 27
                    Content-Type: application/json
                    Date: Sat, 25 Jan 2025 12:36:22 GMT
                    Server: Werkzeug/3.1.3 Python/3.11.6

                    {"prediction":"Virginica"}

Forms             : {}
Headers           : {[Connection, close], [Content-Length, 27], [Content-Type, application/json], [Date, Sat, 25 Jan
                    2025 12:36:22 GMT]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 27
```