UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base

Corso di **Cloud and Network Infrastructures**

Automatic Horizontal Scaling of Pods in Minikube Using Metrics-Server

Anno Accademico 2022/2023

Ivan De Martis

Matricola M63 001238

Prof. Roberto Canonico

Sommario

Soluzioni di orchestrazione dei container 3
Kubernetes 3
Architettura 3
Minikube 6
Setup and Configuration 7
Start cluster 7
Stato dei pods 8
Addon Metric-Server 8
Nodes metrics 8
Pod metrics 9
Node information 11
Istanziazione pod: Nginx 13
Istanze di Nginx 13
Nginx-service 14
Nginx-nodeport 15
Apache JMeter 15
Automatic Horizontal Scaling of Pods Using Metrics-Server 16
Componente di scaling 16
Modulo "*Main*" 17
Modulo "*Scalability Controller*" 18
Modulo "*Metric Module*" 19
Test plan 20
Thread Group 20
HTTP Request 20
Constant Throughput Timer 21
Summary Report 21
Testing 21
Port forwarding 21
Scaling 22

Soluzioni di orchestrazione dei container

L'aumento della popolarità dei container ha reso fondamentale l'introduzione di soluzioni per gestirli, coordinarli ed orchestrarli che semplifichino e automatizzino questi processi complessi. Alcune delle tecnologie principali per l'orchestrazione dei container sono:

- **Docker Compose**: permette di definire e gestire applicazioni multi-container basate su Docker.
- **Kubernetes**: strumento open-source. È progettato per gestire l'implementazione, la scalabilità e la gestione di applicazioni containerizzate su cluster di server.
- **Apache Mesos**: è un sistema di gestione spesso utilizzato in scenari in cui è necessario un utilizzo efficiente delle risorse per applicazioni sia di tipo container che non.
- **Amazon Elastic Container Service (ECS)**: è un servizio di orchestrazione di container completamente gestito offerto da Amazon Web Services (AWS).
- **OpenShift**: è una piattaforma di sviluppo e deployment containerizzata basata su Kubernetes. Fornisce funzionalità aggiuntive per la gestione delle applicazioni e l'automazione.

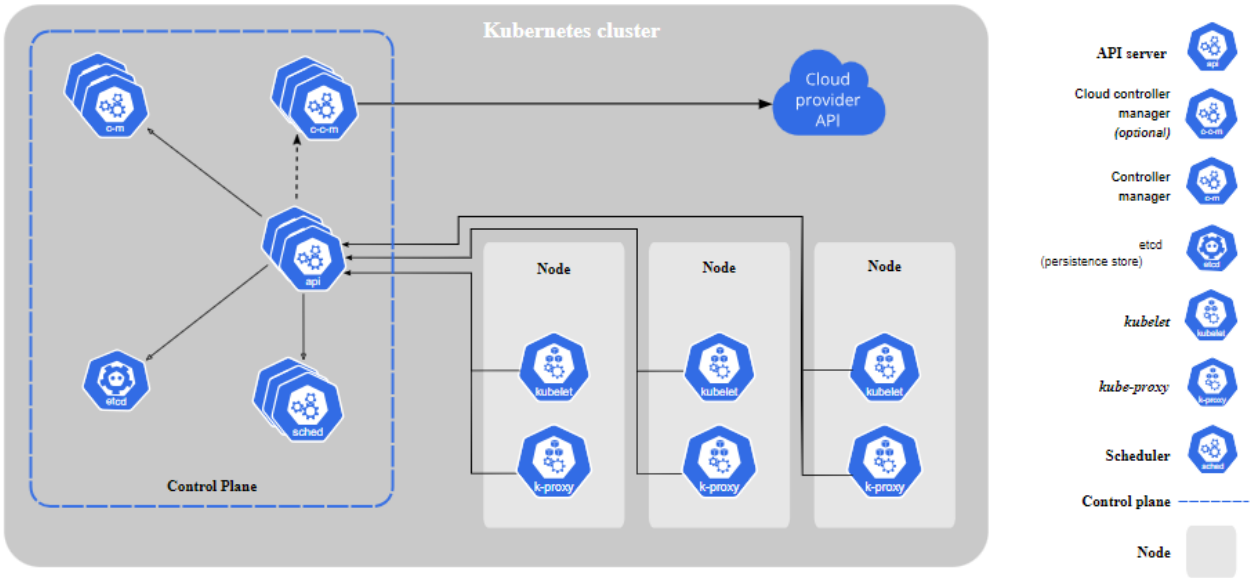
Queste sono solo alcune delle soluzioni di orchestrazione dei container disponibili. La scelta di una soluzione dipende dalle esigenze dell'organizzazione, dalla complessità dell'applicazione e dai requisiti di scalabilità.

Kubernetes

Kubernetes è una piattaforma open source per il rilascio, la scalabilità e il funzionamento di applicazioni basate su container. Basato sul sistema Borg di Google, Kubernetes è stato rilasciato come progetto open source nel 2015 ed è ora mantenuto dalla Cloud Native Computing Foundation (CNCF). È stato adottato da molti fornitori di servizi cloud, tra cui Google, Amazon e Microsoft. Docker è il runtime di container più comunemente utilizzato all'interno dei cluster Kubernetes, anche se questa piattaforma supporta anche altri runtime di container. Kubernetes è particolarmente utile per le aziende che eseguono molti microservizi o che hanno bisogno di **scalare rapidamente le loro applicazioni** (come ad esempio Spotify, Pinterest e Airbnb) in quanto offre diverse funzionalità quali alta disponibilità, scalabilità, disaster recovery.

Architettura

L'architettura di Kubernetes è composta da un cluster che include diversi elementi, analizzati nei seguenti paragrafi e mostrati dalla figura seguente.



Pod
Un pod è l'unità più piccola di calcolo che può essere creata e gestita in Kubernetes. È un concetto chiave per organizzare i containers e risorse all'interno del cluster.

- **Composizione.** Un pod è costituito da uno o più container che condividono risorse di archiviazione e di rete. Inoltre, il pod contiene una specifica su come eseguire questi container.
- **Scheduling:** I contenuti di un pod sono sempre collocati e schedulati insieme, il che significa che i container all'interno di un pod vengono eseguiti sulla stessa macchina virtuale o nodo fisico.
- **Indirizzo IP Assegnato:** Il *Control Plane* assegna a un pod un indirizzo IP che viene utilizzato per la comunicazione interna con altri pod all'interno del cluster.
- **Modello di "Host Logico" Specifico dell'Applicazione:** Un pod modella un "*host logico*" specifico dell'applicazione. Contiene uno o più container applicativi che sono strettamente accoppiati tra loro. Ad esempio, un'applicazione Web potrebbe avere un container per il server Web e un altro container per il server di database all'interno dello stesso pod.
- **Container Init:** Oltre ai container dell'applicazione, un pod può contenere anche "*init container*" che vengono eseguiti solo durante l'avvio del pod. Questi possono essere utilizzati per eseguire compiti di inizializzazione prima che i container principali dell'applicazione vengano avviati.

Nodi
I nodi possono essere divisi in due categorie principali:

- **Master Node:** responsabile della gestione dello stato dell'intero cluster. Include vari componenti chiave come *l'API Server*, il *Controller Manager*, l'*etcd* e il *Scheduler*.
- **Worker Nodes:** i nodi worker di Kubernetes sono responsabili dell'esecuzione dei container degli utenti all'interno di pod e del mantenimento delle risorse dei nodi. È composto da più componenti:
    - **Container Runtime:** è il motore dei containers, necessario per eseguire i container. Il Container Runtime Interface (**CRI**) definisce come il *kubelet* comunica con il runtime di container per avviare e gestire i container.
    - **Kubelet:** Il kubelet è un agente presente su ogni nodo worker. Esso gestisce la comunicazione con il *control plane* del cluster. Il kubelet monitora costantemente lo stato dei pod assegnati al nodo, assicurandosi che siano in esecuzione e sani. Interagisce con il *control plane* per sincronizzare lo stato desiderato del pod con lo stato attuale.
    - **Kube Proxy:** Kube Proxy è responsabile della gestione delle regole di rete all'interno del nodo worker. Si occupa di inoltrare le connessioni di rete ai servizi del cluster, garantendo la comunicazione tra i pod all'interno del cluster. Kube Proxy può eseguire funzioni come bilanciamento del carico, mascheramento IP e gestione delle regole di firewall.

Control Plane
Include tutti i componenti necessari per gestire il cluster Kubernetes. Questi componenti lavorano insieme per gestire il ciclo di vita delle applicazioni, garantendo che le applicazioni containerizzate vengano distribuite, scalate e gestite correttamente. Alcuni dei componenti chiave includono:

- **API Server:** componente cruciale del Control Plane che agisce come l'interfaccia primaria per tutte le interazioni con il cluster e fornisce varie funzionalità importanti:
    - **Interfaccia REST:** Il server API espone un'interfaccia REST che consente a client e applicazioni di interagire con il Control Plane di Kubernetes. Questa interfaccia viene utilizzata per gestire risorse come pod, servizi, distribuzioni e altro ancora.

- **Autenticazione e Autorizzazione:** Il server API gestisce l'autenticazione e l'autorizzazione degli utenti assicurandosi che solo gli utenti autorizzati possano accedere ed eseguire operazioni all'interno del cluster, in base ai loro ruoli e autorizzazioni.
- **Validazione delle Richieste:** Il server API convalida le richieste in arrivo per garantire che rispettino le specifiche delle risorse di Kubernetes e le politiche. Ciò aiuta a mantenere la coerenza e la correttezza della configurazione del cluster.
- **Mutation:** Il server API gestisce gli aggiornamenti, le modifiche e le creazioni di risorse. Quando si crea o modifica una risorsa, ad esempio distribuendo un nuovo pod, il server API gestisce le modifiche e si assicura che siano correttamente riflesse nello stato del cluster.
- **Controllo di Ammissione:** I meccanismi di controllo di ammissione nel server API applicano politiche e regole specifiche alle richieste in arrivo. Ciò aiuta a prevenire azioni non autorizzate o potenzialmente dannose e garantisce che le risorse vengano create secondo le politiche specificate.

- **Controller Manager.** Gestisce i controller che regolano lo stato desiderato del sistema, come:

  - **Kube Controller Manager:** ospita una serie di controller incorporati che svolgono ruoli essenziali nel garantire che lo stato del cluster sia allineato con lo stato desiderato.
  - **Cloud Controller Manager:** Il cloud controller manager è un insieme replicato di processi che opera all'interno del Control Plane. Questo manager è responsabile di interazioni specifiche con il *provider cloud* sottostante. Ad esempio, se Kubernetes è eseguito su un ambiente di cloud pubblico come AWS o GCP, il cloud controller manager gestirà le operazioni legate a tali piattaforme.

- **Scheduler.** Quando viene creato un nuovo pod, il processo di scheduling determina in quale nodo verrà effettivamente eseguito. Questa decisione è basata su diversi fattori:

  - **Requisiti delle Risorse:** Lo scheduler prende in considerazione i requisiti di risorse specificati per il pod, ad esempio CPU e memoria. Cerca di assegnare il pod a un nodo che possa soddisfare questi requisiti.
  - **Località dei Dati:** Se possibile, lo scheduler cerca di assegnare i pod in modo tale che abbiano accesso ai dati necessari. Questo può aiutare a ridurre il ritardo dovuto alla latenza di rete.
  - **Specifiche di Affinità e Anti-Affinità:** indicano le preferenze di dove un pod dovrebbe o non dovrebbe essere schedulato rispetto ad altri pod o risorse del cluster.
  - **Interferenza tra Carichi di Lavoro:** Lo scheduler cerca di evitare che i pod di carichi di lavoro diversi possano interferire tra loro sullo stesso nodo.
  - **Deadline:** Se il pod ha una scadenza o un limite di tempo entro cui deve essere eseguito, lo scheduler tiene conto di questo requisito per assicurarsi che venga pianificato in tempo.

- **etcd:** È un datastore utilizzato per memorizzare le informazioni sullo stato del cluster. Garantisce che tutte le modifiche vengano gestiti in modo coerente e che lo stato sia sempre accessibile e affidabile.

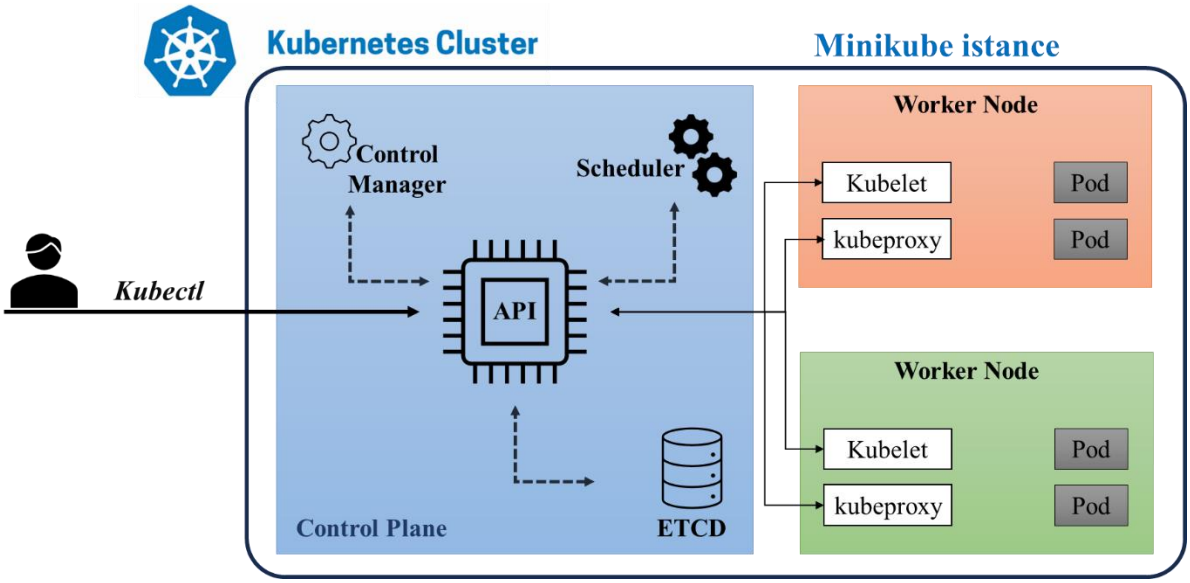
  - **Memorizza informazioni sugli oggetti e sulla configurazione:** può includere informazioni sui pod, i servizi, le configurazioni delle applicazioni e altro ancora.
  - **Endpoint locale su ciascun host:** Ogni nodo nel cluster Kubernetes offre un endpoint locale per etcd. Questo endpoint viene utilizzato per la scoperta dei servizi e per la lettura/scrittura di valori di configurazione.
  - **Cambiamenti riflessi in tutto il cluster:** Tutte le modifiche apportate allo stato del cluster tramite etcd vengono riflessi su tutti i nodi del cluster.

Minikube

Minikube è uno strumento versatile per la gestione di **cluster Kubernetes locali**, ma offre anche funzionalità avanzate che lo rendono una risorsa potente per sviluppatori e operatori di container. Di seguito, verranno analizzate alcune delle sue caratteristiche più avanzate:

1. **Driver Personalizzati**: Minikube è comunemente utilizzato con il driver Docker ma supporta anche una varietà di driver personalizzati. Questi driver includono VirtualBox, Hyper-V, KVM, Parallels e molti altri. La scelta del driver può influire sulla performance e sulla compatibilità con l'ambiente.
2. **Configurazione Avanzata**: Minikube consente una configurazione dettagliata dei cluster Kubernetes locali. È possibile specificare la versione di Kubernetes da utilizzare, impostare limiti di risorse personalizzati, attivare o disattivare addon specifici e configurare la rete in base alle esigenze.
3. **Gestione del Ciclo di Vita**: Minikube offre comandi per la gestione del ciclo di vita del cluster, tra cui l'avvio, l'arresto, l'aggiornamento e la rimozione.
4. **Integrazione con Strumenti di CI/CD**: Minikube può essere utilizzato come parte di una pipeline di integrazione continua (CI) o distribuzione continua (CD) per testare applicazioni containerizzate in un ambiente locale prima di distribuirle su un cluster di produzione.
5. **Risorse e Scalabilità**: Minikube consente di configurare le risorse allocate al cluster, consentendo di gestire la memoria, la CPU e altre risorse in base alle esigenze delle applicazioni.
6. **Persistenza dei Dati**: Per applicazioni che richiedono la persistenza dei dati, Minikube offre opzioni per la gestione di volumi e persistenza dei dati, consentendo di simulare scenari reali in cui i dati devono essere conservati tra riavvii del cluster.
7. **Ambiente di Sviluppo Isolato**: Grazie alla sua natura locale, Minikube offre un ambiente di sviluppo isolato, consentendo agli sviluppatori di testare applicazioni in un ambiente controllato prima di spostarle in ambienti di produzione condivisi.

In conclusione, Minikube è una risorsa versatile e potente per la gestione di cluster Kubernetes locali, fornendo una vasta gamma di funzionalità avanzate per lo sviluppo, il testing e la personalizzazione dell'ambiente Kubernetes locale. La sua flessibilità e la sua capacità di integrazione con altri strumenti lo rendono uno strumento prezioso per gli sviluppatori e gli operatori di container. Di seguito una figura che mostra una istanza di Minikube al livello architetturale.



Setup and Configuration

Di seguito vengono mostrate le varie operazioni eseguite per configurare l'ambiente e comandi di interesse per la gestione e il controllo.

Start cluster

"*minikube start*" è un comando che avvia un cluster di Kubernetes locale utilizzando Minikube.

```
PS C:\WINDOWS\system32> minikube start
* minikube v1.31.2 on Microsoft Windows 11 Home 10.0.22621.2215 Build 22621.2215
* Using the docker driver based on user configuration
* Using Docker Desktop driver with root privileges
* Starting control plane node minikube in cluster minikube
* Pulling base image ...
* Creating docker container (CPUs=2, Memory=4000MB) ...
* Preparing Kubernetes v1.27.4 on Docker 24.0.4 ...
  - Generating certificates and keys ...
  - Booting up control plane ...
  - Configuring RBAC rules ...
* Configuring bridge CNI (Container Networking Interface) ...
* Verifying Kubernetes components...
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
* Enabled addons: storage-provisioner, default-storageclass
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Di seguito una spiegazione passo per passo delle operazioni effettuate dal comando:

1. **Configurazione del Driver Docker.** Viene utilizzato il driver Docker per eseguire e gestire il cluster.
2. **Preparazione dell'Immagine.** Minikube scarica un'immagine base necessaria per creare il cluster.
3. **Creazione del Container Docker.** Minikube crea un container Docker con le specifiche di CPU e memoria specificate (2 CPU e 4000MB di memoria). Questo container sarà la macchina virtuale su cui verrà eseguito il cluster Kubernetes.
4. **Preparazione di Kubernetes.** Minikube prepara l'installazione di Kubernetes versione 1.27.4 su Docker versione 24.0.4. Questo processo include le seguenti fasi:
    - **Generazione di certificati e chiavi**: Minikube crea certificati e chiavi per garantire la sicurezza delle comunicazioni all'interno del cluster.
    - **Avvio del nodo di controllo**: Il nodo di controllo di Kubernetes viene avviato. Questo nodo gestisce la gestione e il coordinamento del cluster.
    - **Configurazione delle regole RBAC (Role-Based Access Control):** Si configurano le regole di controllo degli accessi basate su ruoli per gestire l'autenticazione e l'autorizzazione all'interno del cluster.
5. **Configurazione del Bridge CNI (Container Networking Interface).** Minikube configura l'interfaccia di rete per consentire la comunicazione tra i container all'interno del cluster.
6. **Addon Abilitati.** Minikube abilita due addon nel cluster: *storage-provisioner* e *default-storageclass*. Questi addon forniscono funzionalità aggiuntive per la gestione dello storage.
7. **Completamento.** Una volta che il processo di avvio è stato completato con successo, kubectl, lo strumento di linea di comando per interagire con il cluster Kubernetes, è configurato per utilizzare il cluster Minikube come cluster predefinito nel namespace "*default*".

In sintesi, il comando minikube start ha creato un cluster Kubernetes Minikube sul sistema utilizzando il driver Docker Desktop, configurando tutti i componenti necessari e abilitando alcuni addon. Ora è possibile iniziare a utilizzare il cluster Minikube per lo sviluppo e il test delle applicazioni Kubernetes.

Stato dei pods

"*kubectl get po -A*" permette di ottenere lo stato dei pod all'interno dei *namespace* nel cluster Minikube.

```
PS C:\WINDOWS\system32> kubectl get po -A
NAMESPACE     NAME                               READY   STATUS    RESTARTS   AGE
kube-system   coredns-5d78c9869d-bqxrh           1/1     Running   0          2m48s
kube-system   etcd-minikube                      1/1     Running   0          3m3s
kube-system   kube-apiserver-minikube            1/1     Running   0          3m
kube-system   kube-controller-manager-minikube   1/1     Running   0          3m
kube-system   kube-proxy-28kvj                   1/1     Running   0          2m48s
kube-system   kube-scheduler-minikube            1/1     Running   0          3m
kube-system   storage-provisioner                1/1     Running   0          2m59s
```

Di seguito una spiegazione di cosa rappresenta ciascuna colonna dell'output:

- **NAMESPACE**: mostra il namespace a cui appartengono i pod. "*kube-system*" è un namespace utilizzato per i componenti principali di Kubernetes.
- **NAME**: mostra il nome del pod.
- **READY**: mostra i container all'interno del pod pronti e in esecuzione, rispetto al numero totale di container nel pod. "1/1" significa che c'è un solo container nel pod ed è pronto e in esecuzione.
- **STATUS**: mostra lo stato attuale del pod. "*Running*" significa che il pod è in esecuzione correttamente, mentre "*Pending*" indicherebbe che il pod sta aspettando risorse per essere avviato.
- **RESTARTS**: mostra quante volte il pod è stato riavviato.
- **AGE**: mostra da quanto tempo il pod è stato avviato.

Addon Metric-Server

"*minikube addons enable metrics-server*" serve per abilitare l'addon "Metrics-Server" nel cluster Minikube.

```
PS C:\WINDOWS\system32> minikube addons enable metrics-server
* metrics-server is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
  - Using image registry.k8s.io/metrics-server/metrics-server:v0.6.4
* The 'metrics-server' addon is enabled
```

In particolare, il comando ha abilitato l'addon utilizzando un'immagine denominata *registry.k8s.io/metrics-server/metrics-server:v0.6.4* per installare il Metrics-Server nel cluster. Questa immagine contiene il software necessario per raccogliere e distribuire le metriche all'interno del cluster.

"Metrics-Server" è un componente Kubernetes che raccoglie metriche sulle risorse del cluster, come CPU e utilizzo della memoria, e le rende disponibili per il sistema Kubernetes. Questo addon è utile per **monitorare le risorse e le prestazioni** delle applicazioni in esecuzione all'interno del tuo cluster Kubernetes.

In sostanza, grazie a questo addon, il cluster Minikube è configurato per raccogliere e rendere disponibili metriche sulle risorse del cluster stesso.

Nodes metrics

Il comando mostrato di seguito è utilizzato per recuperare le metriche di utilizzo delle risorse **di un nodo specifico** in un cluster Kubernetes.

```
PS C:\WINDOWS\system32> kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/minikube
```

Si è scelto, per una maggiore chiarezza, di riportare il risultato del comando nella figura seguente e di analizzarne il contenuto.

```
{
    "kind":"NodeMetrics",
    "apiVersion":"metrics.k8s.io/v1beta1",
    "metadata":{
        "name":"minikube",
        "creationTimestamp":"2023-09-11T16:17:19Z",
        "labels":{
            "beta.kubernetes.io/arch":"amd64",
            "beta.kubernetes.io/os":"linux",
            "kubernetes.io/arch":"amd64",
            "kubernetes.io/hostname":"minikube",
            "kubernetes.io/os":"linux",
            "minikube.k8s.io/commit":"fd7ecd9c4599bef9f04c0986c4a0187f98a4396e",
            "minikube.k8s.io/name":"minikube",
            "minikube.k8s.io/primary":"true",
            "minikube.k8s.io/updated_at":"2023_09_09T11_53_05_0700",
            "minikube.k8s.io/version":"v1.31.2",
            "node-role.kubernetes.io/control-plane":"",
            "node.kubernetes.io/exclude-from-external-load-balancers":""
            }
        },
            "timestamp":"2023-09-11T16:16:58Z",
            "window":"55.506s",
            "usage":{
                "cpu":"170179881n",
                "memory":"1042420Ki"
                }
}
```


- `**kind**` e `**apiVersion**`: Questi campi indicano il tipo di risorsa restituita e la versione dell'API a cui appartiene. Nel caso dei risultati restituiti, stiamo ottenendo le metriche di un nodo, quindi il tipo di risorsa è "*NodeMetrics*" e la versione dell'API è "*metrics.k8s.io/v1beta1*".
- `**metadata**`: Questo campo contiene metadati relativi alla risorsa. Nel caso specifico, i metadati includono il nome del nodo ("minikube"), un timestamp di creazione e alcune etichette (labels) associate al nodo. Le etichette forniscono informazioni sul nodo, come l'architettura, il sistema operativo, il nome e la versione di Minikube.
- `**timestamp**`: Questo campo indica il momento in cui sono state registrate le metriche.
- `**window**`: Rappresenta la finestra temporale all'interno della quale sono state raccolte le metriche.
- `**usage**`: Questo campo contiene le metriche effettive di utilizzo delle risorse del nodo:
    - `**cpu**`: Indica l'utilizzo della CPU del nodo in nanocores (n).
    - `**memory**`: Indica l'utilizzo della memoria del nodo in kibibytes (Ki).

In alternativa, è possibile utilizzare il seguente comando che fornisce l'uso delle risorse e ne stima una percentuale rispetto al totale:

```
PS C:\WINDOWS\system32> kubectl top node minikube
NAME       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube   130m         1%     1216Mi          15%
```

Le metriche di utilizzo delle risorse sono importanti per monitorare e gestire le prestazioni dei nodi all'interno di un cluster Kubernetes. Consentono di identificare eventuali problemi di sovrautilizzo o sottoutilizzo delle risorse e di prendere decisioni informate sulla scalabilità delle applicazioni.

Pod metrics

Il comando mostrato di seguito viene utilizzato per recuperare le metriche di utilizzo delle risorse **di un pod specifico** nel namespace "*kube-system*" all'interno del cluster Kubernetes.

```
PS C:\WINDOWS\system32> kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/metrics-server-7746886d4f-9btkj
```

Si è scelto, per una maggiore chiarezza, di riportare il risultato del comando nella figura seguente e di analizzarne il contenuto.

```
{
    "kind":"PodMetrics",
    "apiVersion":"metrics.k8s.io/v1beta1",
    "metadata":{
        "name":"metrics-server-7746886d4f-9btkj",
        "namespace":"kube-system",
        "creationTimestamp":"2023-09-11T16:32:15Z",
        "labels":{"k8s-app":"metrics-server",
        "pod-template-hash":"7746886d4f"
        }
        },
            "timestamp":"2023-09-11T16:32:10Z",
            "window":"1m0.008s",
            "containers":[{
                "name":"metrics-server",
                "usage":{
                    "cpu":"2172060n",
                    "memory":"61588Ki"
                    }
                    }
                    ]
                    }
```


- `**kind**` e `**apiVersion**`: Questi campi indicano il tipo di risorsa restituita e la versione dell'API a cui appartiene.
- `**metadata**`: Questo campo contiene metadati relativi al pod. Nel caso specifico, i metadati includono il nome del pod ("metrics-server-7746886d4f-9btkj"), il namespace a cui appartiene ("kube-system"), un timestamp di creazione e alcune etichette (labels) associate al pod. Le etichette forniscono informazioni sul pod, come l'applicazione associata e un hash del modello del pod.
- `**timestamp**`: Questo campo indica il momento in cui sono state registrate le metriche del pod.
- `**window**`: Questo campo rappresenta la finestra temporale all'interno della quale sono state raccolte le metriche.
- `**containers**`: Questo campo contiene le metriche di utilizzo delle risorse per i containers all'interno del pod. Nel caso specifico, c'è un solo container chiamato "metrics-server", e le metriche riportate sono le seguenti:
    - `**cpu**`: Indica l'utilizzo della CPU del container in nanocores (n).
    - `**memory**`: Indica l'utilizzo della memoria del container in kibibytes (Ki).

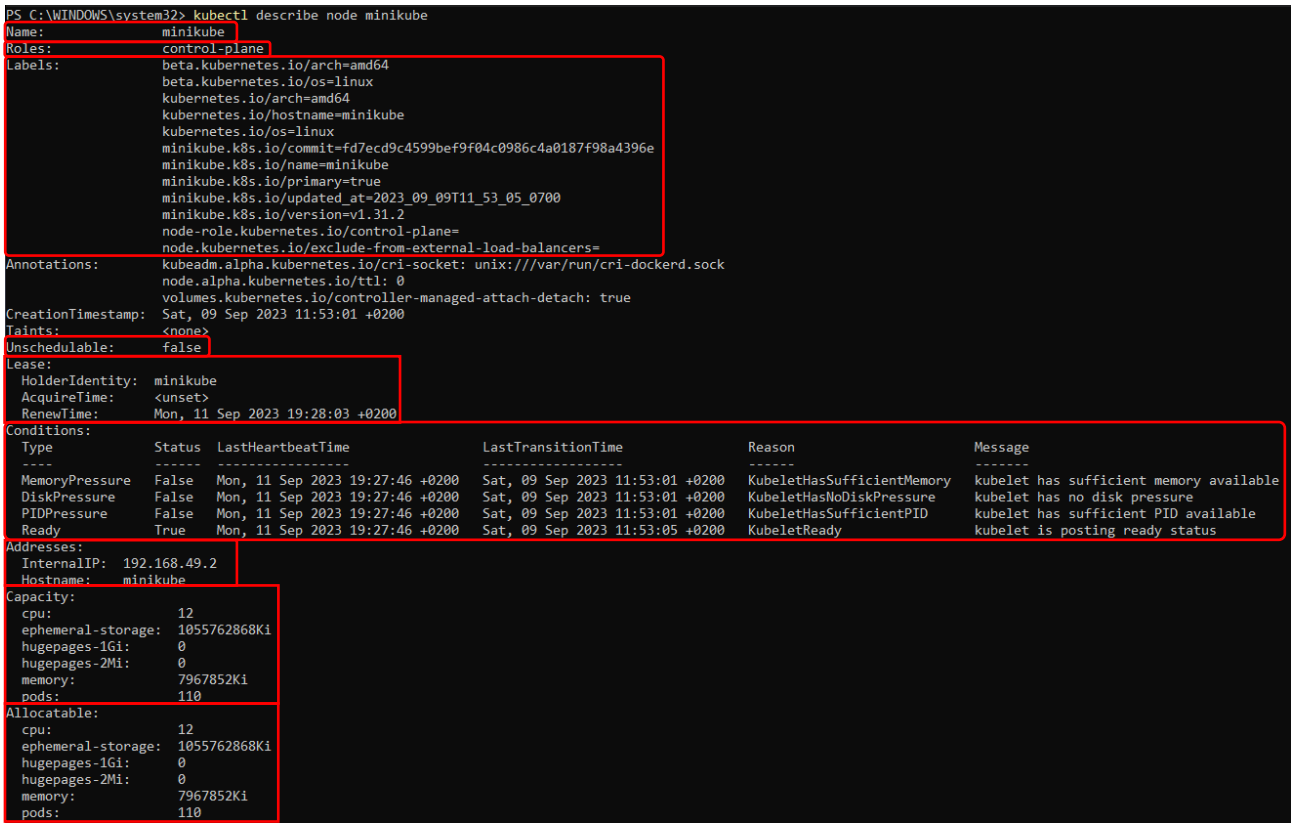
In alternativa, è possibile utilizzare il seguente comando che fornisce l'uso delle risorse:

```
PS C:\WINDOWS\system32> kubectl top pod metrics-server-7746886d4f-9btkj -n kube-system
NAME                              CPU(cores)   MEMORY(bytes)
metrics-server-7746886d4f-9btkj   2m           70Mi
```

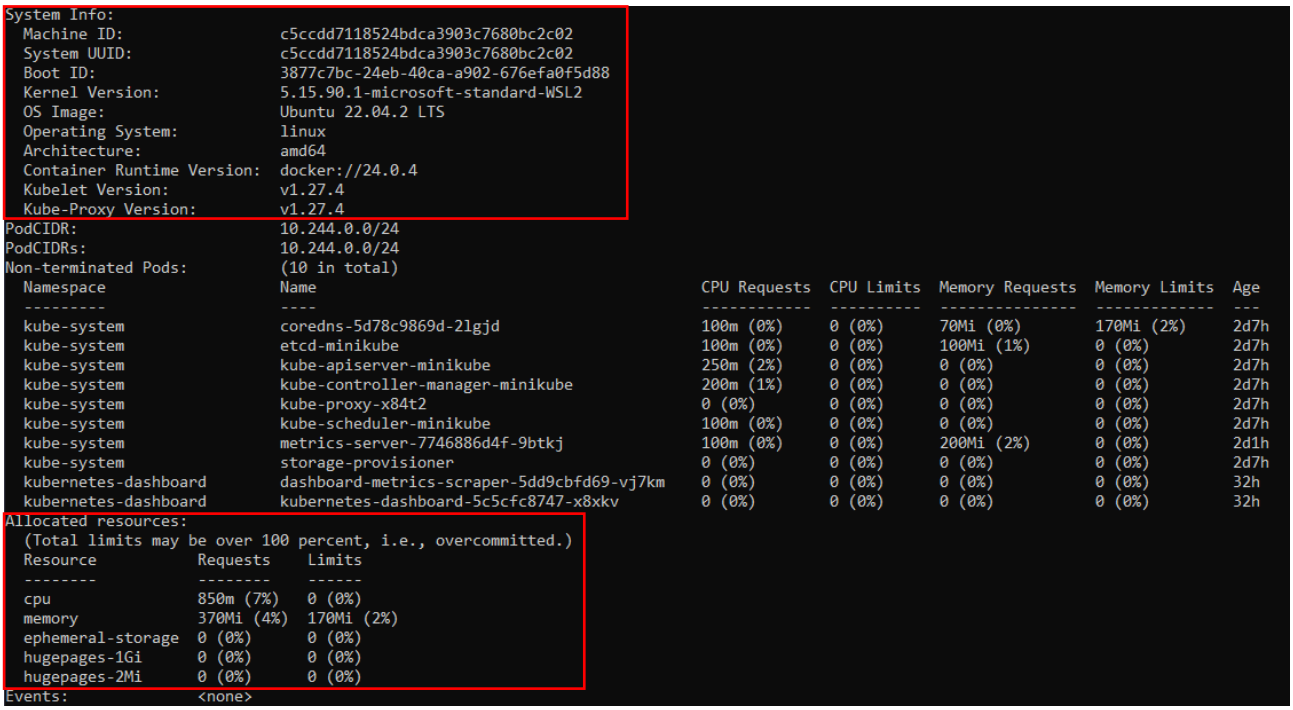
Queste metriche sono utili per monitorare le prestazioni dei pod e dei containers nel cluster Kubernetes. Consentono di verificare quanto CPU e memoria vengono utilizzati da un determinato pod o container, il che può aiutare nella risoluzione dei problemi di prestazioni e nell'ottimizzazione delle risorse.

Node information

Il comando mostrato di seguito è utilizzato per ottenere una descrizione dettagliata del nodo "*minikube*" nel cluster Kubernetes. Questo comando fornisce molte informazioni sul nodo stesso, come la capacità delle risorse, lo stato del nodo, le condizioni attuali e altro ancora. Di seguito, viene presentato l'output e ne vengono spiegati i vari punti:



```
PS C:\WINDOWS\system32> kubectl describe node minikube
Name:               minikube
Roles:              control-plane
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=minikube
                    kubernetes.io/os=linux
                    minikube.k8s.io/commit=fd7ecd9c4599bef9f04c0986c4a0187f98a4396e
                    minikube.k8s.io/name=minikube
                    minikube.k8s.io/primary=true
                    minikube.k8s.io/updated_at=2023_09_09T11_53_05_0700
                    minikube.k8s.io/version=v1.31.2
                    node-role.kubernetes.io/control-plane=
                    node.kubernetes.io/exclude-from-external-load-balancers=
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: unix:///var/run/cri-dockerd.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Sat, 09 Sep 2023 11:53:01 +0200
Taints:             <none>
Unschedulable:      false
Lease:
  HolderIdentity:  minikube
  AcquireTime:     <unset>
  RenewTime:       Mon, 11 Sep 2023 19:28:03 +0200
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------                ------                       -------
  MemoryPressure   False   Mon, 11 Sep 2023 19:27:46 +0200   Sat, 09 Sep 2023 11:53:01 +0200   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Mon, 11 Sep 2023 19:27:46 +0200   Sat, 09 Sep 2023 11:53:01 +0200   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Mon, 11 Sep 2023 19:27:46 +0200   Sat, 09 Sep 2023 11:53:01 +0200   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Mon, 11 Sep 2023 19:27:46 +0200   Sat, 09 Sep 2023 11:53:05 +0200   KubeletReady                 kubelet is posting ready status
Addresses:
  InternalIP:  192.168.49.2
  Hostname:    minikube
Capacity:
  cpu:                12
  ephemeral-storage:  1055762868Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             7967852Ki
  pods:               110
Allocatable:
  cpu:                12
  ephemeral-storage:  1055762868Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             7967852Ki
  pods:               110
```



```
System Info:
  Machine ID:                 c5ccdd7118524bdca3903c7680bc2c02
  System UUID:                c5ccdd7118524bdca3903c7680bc2c02
  Boot ID:                    3877c7bc-24eb-40ca-a902-676efa0f5d88
  Kernel Version:             5.15.90.1-microsoft-standard-WSL2
  OS Image:                   Ubuntu 22.04.2 LTS
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  docker://24.0.4
  Kubelet Version:            v1.27.4
  Kube-Proxy Version:         v1.27.4
PodCIDR:                      10.244.0.0/24
PodCIDRs:                     10.244.0.0/24
Non-terminated Pods:          (10 in total)
  Namespace                   Name                                CPU Requests  CPU Limits  Memory Requests  Memory Limits  Age
  ---------                   ----                                ------------  ----------  ---------------  -------------  ---
  kube-system                 coredns-5d78c9869d-21gjd            100m (0%)     0 (0%)      70Mi (0%)        170Mi (2%)     2d7h
  kube-system                 etcd-minikube                       100m (0%)     0 (0%)      100Mi (1%)       0 (0%)         2d7h
  kube-system                 kube-apiserver-minikube             250m (2%)     0 (0%)      0 (0%)           0 (0%)         2d7h
  kube-system                 kube-controller-manager-minikube    200m (1%)     0 (0%)      0 (0%)           0 (0%)         2d7h
  kube-system                 kube-proxy-x84t2                    0 (0%)        0 (0%)      0 (0%)           0 (0%)         2d7h
  kube-system                 kube-scheduler-minikube             100m (0%)     0 (0%)      0 (0%)           0 (0%)         2d7h
  kube-system                 metrics-server-7746886d4f-9btkj     100m (0%)     0 (0%)      200Mi (2%)       0 (0%)         2d1h
  kube-system                 storage-provisioner                 0 (0%)        0 (0%)      0 (0%)           0 (0%)         2d7h
  kubernetes-dashboard        dashboard-metrics-scraper-5dd9cbfd69-vj7km  0 (0%)  0 (0%)      0 (0%)           0 (0%)         32h
  kubernetes-dashboard        kubernetes-dashboard-5c5cfc8747-x8xkv       0 (0%)  0 (0%)      0 (0%)           0 (0%)         32h
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                850m (7%)   0 (0%)
  memory             370Mi (4%)  170Mi (2%)
  ephemeral-storage  0 (0%)      0 (0%)
  hugepages-1Gi      0 (0%)      0 (0%)
  hugepages-2Mi      0 (0%)      0 (0%)
Events:              <none>
```

- **Name**: Il nome del nodo.
- **Roles**: Il nodo è etichettato come "*control-plane*".

- **Labels**: Queste etichette forniscono informazioni sul nodo, come l'architettura, il sistema operativo, il nome del nodo e altre informazioni pertinenti.
- **Unschedulable**: Questo campo indica se il nodo è schedulabile o meno. Se fosse impostato su "true", il nodo non accetterebbe nuovi pod.
- **Lease**: Questa sezione mostra informazioni sulla *lease* del nodo, come l'identità del possessore, il tempo di acquisizione e il tempo di rinnovo.
- **Conditions**: Questa sezione elenca le condizioni attuali del nodo, come "MemoryPressure", "DiskPressure", "PIDPressure" e "Ready". Ogni condizione ha uno stato ("Status") che indica se la condizione è soddisfatta o meno.
- **Addresses**: Mostra gli indirizzi associati al nodo, come l'indirizzo IP interno e il nome host.
- **Capacity**: Questa sezione mostra la capacità totale del nodo per le risorse come CPU, memoria, storage, e il numero di pod supportati.
- **Allocatable**: Questa sezione mostra le risorse effettivamente allocabili, che tengono conto delle risorse riservate per il sistema.
- **System Info**: Fornisce informazioni sulla macchina fisica o virtuale su cui il nodo è in esecuzione, come l'ID della macchina, l'UUID del sistema, il kernel, l'immagine del sistema operativo, l'architettura e la versione del runtime del container.
    - La **presenza di "wsl"** è dovuta al fatto che **Docker utilizza Windows Subsystem for Linux (WSL 2) come backend** per l'esecuzione dei container Docker. In sostanza, Docker può essere configurato per eseguire container all'interno di un'istanza WSL 2, anche se è installato su Windows. Questa configurazione è comune ed è spesso utilizzata per garantire una migliore integrazione tra Docker e Windows.
- **Non-terminated Pods**: Questa sezione elenca i pod in esecuzione sul nodo, insieme alle loro richieste di risorse e limiti, nonché all'età dei pod.
- **Allocated resources**: Questa sezione mostra le risorse allocate per il nodo, inclusi limiti e richieste di CPU e memoria.

In generale, il comando `*kubectl describe node*` è utile per ottenere una visione dettagliata delle risorse di un nodo nel cluster Kubernetes, il che può essere utile per il monitoraggio e la gestione del cluster.

Istanziazione pod: Nginx

Con lo scopo di creare un contesto di test e sviluppo, si è scelto di **implementare Nginx come pod all'interno di Minikube**. Nginx è un potente server web e proxy inverso open-source ampiamente utilizzato nella gestione delle richieste HTTP e HTTPS. La sua scalabilità, prestazioni elevate e versatilità lo rendono una scelta popolare per ottimizzare l'erogazione di servizi web. Utilizzando l'ambiente Kubernetes locale si ha la possibilità di eseguire una replica di Nginx e testare le richieste di homepage in un ambiente controllato.

Istanze di Nginx

Il file YAML presentato di seguito rappresenta un oggetto di tipo "*Deployment*" in Kubernetes, il quale viene utilizzato per **definire la gestione delle applicazioni nei cluster Kubernetes**.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: test-apps
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Di seguito una breve spiegazione del contenuto del file YAML:

- **apiVersion**: Specifica la versione dell'API di Kubernetes utilizzata per definire l'oggetto. In questo caso, è "*apps/v1*", che è la versione delle API per i Deployment.
- **kind**: Specifica il tipo di oggetto che si sta creando, che è un "Deployment". I Deployment in Kubernetes sono utilizzati per definire come un'applicazione distribuita e gestita.
- **metadata**: Contiene i metadati associati all'oggetto Deployment. In particolare, si è definito il nome del Deployment come "*nginx-deployment*" e specificato il namespace come "*test-apps*".
- **spec**: Contiene le specifiche per il Deployment che determinano come l'applicazione dovrebbe essere eseguita e gestita:
    - **replicas**: Specifica il numero desiderato di repliche dell'applicazione. In questo caso, è stato impostato "*replicas*" su 1, il che significa che si vuole una singola istanza dell'applicazione.
    - **selector**: Definisce un selettore basato su etichette (*labels*) che verrà utilizzato per identificare le repliche del pod gestite da questo Deployment. Il selettore qui indica che le repliche avranno un'etichetta "*app*" con valore "*nginx*".
    - **template**: Specifica il modello per la creazione dei pod che costituiranno l'applicazione:
        1. **metadata**: Contiene le etichette da applicare ai pod creati da questo template. In questo caso, viene applicata un'etichetta "*app*" con valore "*nginx*".
        2. **spec**: Definisce le specifiche per il pod stesso:
            1. **containers**: Elenco dei container all'interno del pod. Nel caso in esame, c'è un solo container chiamato "*nginx*" che utilizza l'immagine Docker "*nginx:latest*". Questo container eseguirà il server web Nginx sulla porta 80.

In breve, questo file YAML definisce un Deployment chiamato "*nginx-deployment*" nel namespace "*test-apps*" che gestisce una singola istanza di un pod contenente il server web Nginx.

È possibile applicare il file al cluster Kubernetes per creare e gestire il pod Nginx con il seguente comando:

```
PS C:\WINDOWS\system32> kubectl apply -f C:\Users\idm95\Desktop\kubernetes-deployments\nginx.yaml -n test-apps
deployment.apps/nginx-deployment configured
```

Come è possibile notare dalla figura seguente, richiedendo tramite il seguente comando i pods presenti all'interno del namespace test-apps, è presente l'istanza di nginx.

```
PS C:\WINDOWS\system32> kubectl get pods -n test-apps
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-57d84f57dc-nkprn   1/1     Running   0          6m13s
```

Nginx-service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

A questo punto è stato necessario l'uso di un ulteriore file YAML rappresentante un oggetto di tipo "*Service*" in Kubernetes. I servizi in Kubernetes sono utilizzati per esporre le applicazioni all'interno di un cluster o all'esterno di esso, consentendo la comunicazione con i pod sottostanti. In questo caso viene utilizzato per **creare una porta per la comunicazione interna tra tutti i pod.**

Ecco una spiegazione del contenuto del file YAML:

- **apiVersion**: Specifica la versione dell'API di Kubernetes utilizzata per definire l'oggetto. In questo caso, è "*v1*", che è la versione di base delle API di Kubernetes.
- **kind**: Specifica il tipo di oggetto che si sta creando, che è un "*Service*".
- **metadata**: Contiene i metadati associati al servizio. In particolare, si è definito il nome del servizio come "*nginx-service*".
- **spec**: Contiene le specifiche per il servizio, che definiscono come il servizio dovrebbe indirizzare il traffico verso i pod sottostanti:
    - **selector**: Definisce un selettore basato su *labels* usato per individuare i pod destinatari del traffico. In questo caso, seleziona i pod che hanno un'etichetta "*app*" con valore "*nginx*".
    - **ports**: Specifica le porte attraverso le quali il servizio ascolterà e indirizzerà il traffico:
        - **protocol**: Specifica il protocollo utilizzato, che è **TCP** in questo caso.
        - **port**: Specifica la porta su cui il servizio ascolterà le richieste in ingresso.
        - **targetPort**: Specifica la porta di destinazione sui pod selezionati. È configurato su 80, il che significa che il traffico in ingresso verrà instradato ai pod sulla porta 80.

In sintesi, questo file YAML definisce un servizio chiamato "*nginx-service*" che instraderà il traffico alle istanze dei pod con l'etichetta "*app: nginx*" sulla porta 80.

Di seguito il comando per applicare questo servizio al pod:

```
PS C:\WINDOWS\system32> kubectl apply -f C:\Users\idm95\Desktop\kubernetes-deployments\nginx-service.yaml -n test-apps
service/nginx-service configured
```

Nginx-nodeport

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Infine, è stato creato un altro file YAML per un altro oggetto "Service", ma con una differenza importante rispetto al primo: ha il campo **type** impostato su "*NodePort*", il quale cambia il comportamento del servizio.

Ecco una spiegazione di come cambia il comportamento del servizio:

- **type: NodePort**: Quando il tipo è impostato su "*NodePort*", il servizio **esporrà una porta aperta su ciascun nodo del cluster**. Questa porta aperta è la stessa su tutti i nodi e può essere utilizzata per accedere al servizio.

Sostanzialmente, quindi si ha che:

- Il servizio precedente crea una **modalità di rete all'interno del cluster Kubernetes**. È utilizzato principalmente per **consentire a un insieme di pod all'interno del cluster di comunicare tra loro** in modo agevole ma non espone direttamente il servizio all'esterno del cluster. È un servizio interno utilizzato per l'orchestrazione interna delle applicazioni.
- Il servizio presentato in questo paragrafo **espone il servizio all'esterno del cluster Kubernetes**. Crea un "*NodePort*" su tutti i nodi del cluster. Ciò significa che è possibile accedere al servizio da fuori del cluster utilizzando l'IP di qualsiasi nodo e la porta specificata.

Di seguito la figura mostra il comando utilizzato:

```
PS C:\WINDOWS\system32> kubectl apply -f C:\Users\idm95\Desktop\kubernetes-deployments\nginx-service-nodeport.yaml -n test-apps
service/nginx-service configured
```

Apache JMeter

Con lo scopo di creare un **carico di lavoro verso il pod Nginx** istanziato, richiedendo più volte la homepage del server web, è stato installato e settato Apache JMeter.

Apache JMeter è una sofisticata e affidabile piattaforma open-source progettata per il testing delle prestazioni, il carico e l'analisi delle applicazioni e dei servizi. Questo strumento versatile è ampiamente utilizzato per valutare le performance di applicazioni web, servizi RESTful, server FTP, database, e altri servizi basati su protocolli di comunicazione. Con la sua ricca gamma di funzionalità e la capacità di simulare un ampio spettro di scenari di utilizzo, JMeter si è affermato come uno degli strumenti preferiti dalle organizzazioni per garantire che le loro applicazioni rimangano performanti, scalabili e affidabili.

Automatic Horizontal Scaling of Pods Using Metrics-Server

Nel seguente capitolo, verrà presentato lo sviluppo di un componente con una duplice finalità all'interno di un cluster Kubernetes: **monitoraggio** e **scalabilità**.

Questo componente sarà progettato per offrire un'analisi dettagliata sull'utilizzo delle risorse di CPU da parte dei pod in esecuzione all'interno del cluster Minikube. Allo stesso tempo, verranno usate queste informazioni di monitoraggio per implementare una logica di scalabilità intelligente.

Il **monitoraggio** delle risorse è un aspetto fondamentale nell'ambito delle infrastrutture basate su container, in quanto consente di tenere traccia delle prestazioni dei servizi e identificare eventuali situazioni di sovra utilizzo o sottoutilizzo delle risorse. Queste informazioni sono essenziali per garantire che le applicazioni siano eseguite in modo efficiente e affidabile.

La **scalabilità** è un altro pilastro cruciale dell'orchestrazione dei container. Con una logica di scalabilità ben progettata, è possibile adattare dinamicamente il numero di istanze dei pod per far fronte alle fluttuazioni del carico di lavoro.

Di seguito si entrerà nei dettagli di come è stato implementare questo componente di monitoraggio e scalabilità nell'ambiente Kubernetes, esaminando ciascun modulo in modo approfondito.

Componente di scaling

Il seguente paragrafo fornisce una panoramica completa dello script di controllo per la scalabilità, implementato in linguaggio Python su *PyCharm*.

Il programma è suddiviso in **tre moduli**: "*Main*," "*Scalability Controller*," e "*Metric Module*." Ogni modulo svolge un ruolo specifico nel processo complessivo di controllo di scalabilità.

- Il modulo "*Main*" costituisce il punto di partenza del programma e implementa il ciclo di monitoraggio continuo.
- Il "*Main*" richiama il modulo "*Metric Module*" per ottenere le metriche dei pod e passa queste informazioni al modulo "*Scalability Controller*" per prendere decisioni di scalabilità.
- Il modulo "*Scalability Controller*" è responsabile per il controllo di scalabilità in base alle metriche ricevute e aggiorna il numero di repliche di un deployment Kubernetes in tempo reale.
- Il modulo "*Metric Module*" si occupa del recupero delle metriche dei pod dal cluster Kubernetes.

Di seguito verrà esaminato ciascuno di questi moduli in modo completo, descrivendo le loro funzioni, le interazioni e l'importanza all'interno del contesto del programma di controllo di scalabilità.

Modulo "*Main*"

Il modulo "*Main*" ha come scopo principale quello di eseguire un ciclo continuo in cui monitora le metriche dei pod all'interno del namespace "*test-apps*" e applicare il controllo di scalabilità in base a queste metriche.

```python
import time
from scalability_controller import scale_deployment_based_on_pod_metrics
from your_metrics_module import get_pod_metrics
from kubernetes import config

# Configura la connessione da variabili d'ambiente o kubeconfig
config.load_kube_config()  # Utilizza il kubeconfig predefinito

# Codice principale per il controllo di scalabilità
1 usage
def main():
    while True:
        try:
            pod_metrics = get_pod_metrics("test-apps")
            print("\nMetriche dei pod nel namespace 'test-apps':")
            for metric in pod_metrics:
                print(f"Pod: {metric['metadata']['name']}, CPU Usage: {metric['containers'][0]['usage']['cpu']}")

            scale_deployment_based_on_pod_metrics( namespace: "test-apps", pod_metrics)
            print("Eseguito il controllo di scalabilità.")
        except Exception as e:
            print(f"Errore durante il controllo di scalabilità: {str(e)}")

        time.sleep(60)

if __name__ == "__main__":
    main()
```

Ecco una spiegazione più dettagliata:

- **Configurazione Kubernetes**: Carica la configurazione di Kubernetes dal *kubeconfig* predefinito. Questo consente al programma di comunicare con il cluster Kubernetes.
- **Funzione "main"**: Questa è la funzione principale che verrà eseguita quando il modulo viene avviato come script autonomo. La funzione fa quanto segue:
    - **Ciclo Continuo**: Avvia un ciclo continuo con un loop "*while True*" per eseguire il controllo di scalabilità in modo periodico.
    - **Recupero delle Metriche dei Pod**: Chiama la funzione "*get_pod_metrics*" per ottenere le metriche dei pod all'interno del namespace "*test-apps*" e stampa queste metriche a scopo di monitoraggio.
    - **Controllo di Scalabilità**: Chiama la funzione "*scale_deployment_based_on_pod_metrics*" per eseguire il controllo di scalabilità in base alle metriche dei pod ottenute precedentemente. Questa funzione può aumentare o diminuire il numero di repliche del deployment in base alle metriche della CPU.
    - **Gestione delle Eccezioni**: Gestisce eventuali eccezioni che possono verificarsi durante l'esecuzione del controllo di scalabilità e le registra in caso di errori.
    - **Intervallo di Attesa**: Mette in pausa l'esecuzione per un minuto prima di eseguire un nuovo ciclo di monitoraggio.

Modulo "*Scalability Controller*"

Il modulo "Scalability Controller" contiene la logica per il controllo di scalabilità in base alle metriche ottenute.

```python
import os
import time
from kubernetes import config, client

# Configura la connessione da variabili d'ambiente o kubeconfig
config.load_kube_config()  # Utilizza il kubeconfig predefinito

# Funzione per scalare il deployment in base alle metriche dei pod
2 usages
def scale_deployment_based_on_pod_metrics(namespace, pod_metrics):
    max_replicas = 10  # Numero massimo di repliche consentite
    api_instance = client.AppsV1Api()
    deployment = api_instance.read_namespaced_deployment(name="nginx-deployment", namespace=namespace)

    for metric in pod_metrics:
        pod_name = metric['metadata']['name']
        cpu_usage_str = metric['containers'][0]['usage']['cpu']

        # Confronta l'uso della CPU del pod con una soglia desiderata
        if cpu_usage_str > "2000000n":  # Soglia CPU in nanocore
            if deployment.spec.replicas < max_replicas:
                deployment.spec.replicas += 1  # Aumenta il numero di repliche se sotto il limite massimo
        else:
            if deployment.spec.replicas > 1:
                deployment.spec.replicas -= 1  # Diminuisci il numero di repliche se sopra il minimo

    # Applica le modifiche al deployment
    api_instance.patch_namespaced_deployment(name="nginx-deployment", namespace=namespace, body=deployment)
```

Ecco una spiegazione dettagliata:

- **Configurazione Kubernetes**: Carica la configurazione di Kubernetes dal *kubeconfig* predefinito per consentire la comunicazione con il cluster.
- **Funzione di Scalabilità "*scale_deployment_based_on_pod_metrics*"**: Questa funzione esegue il controllo di scalabilità in base alle metriche dei pod. Ecco cosa fa:
    - **Lettura del Deployment**: Utilizza il client Kubernetes per leggere il deployment denominato "*nginx-deployment*" nel namespace specificato.
    - **Iterazione sulle Metriche dei Pod**: Itera attraverso le metriche dei pod ottenute come input dalla funzione principale. Per ogni metrica, controlla l'uso della CPU del pod rispetto a una soglia specifica (nel caso in esame, 2000000 nanocore).
    - **Aumento o Diminuzione delle Repliche**: Se l'uso della CPU supera la soglia e il numero di repliche del deployment è inferiore al limite massimo consentito, aumenta il numero di repliche di 1. Se l'uso della CPU è al di sotto della soglia e il numero di repliche è superiore al minimo consentito, diminuisce il numero di repliche di 1.
    - **Applicazione delle Modifiche**: Dopo aver effettuato le modifiche, applica il nuovo stato del deployment utilizzando la funzione "*patch_namespaced_deployment*" del client Kubernetes.

Modulo "*Metric Module*"

Il modulo "Metric Module" contiene la logica per ottenere le metriche dei pod dal cluster Kubernetes.

```python
from kubernetes import client

2 usages
def get_pod_metrics(namespace):
    metrics_api = client.CustomObjectsApi()
    group = "metrics.k8s.io"
    version = "v1beta1"
    plural = "pods"

    try:
        response = metrics_api.list_namespaced_custom_object(
            group=group,
            version=version,
            namespace=namespace,
            plural=plural
        )

        pod_metrics = response.get("items", [])


        for metric in pod_metrics:
            pod_name = metric['metadata']['name']

            # Ottieni l'utilizzo effettivo della CPU in formato nanoCore
            cpu_usage = metric['containers'][0]['usage']['cpu']

        return pod_metrics
    except Exception as e:
        print(f"Errore durante il recupero delle metriche dei pod: {str(e)}")
        return []
```

Ecco una spiegazione dettagliata:

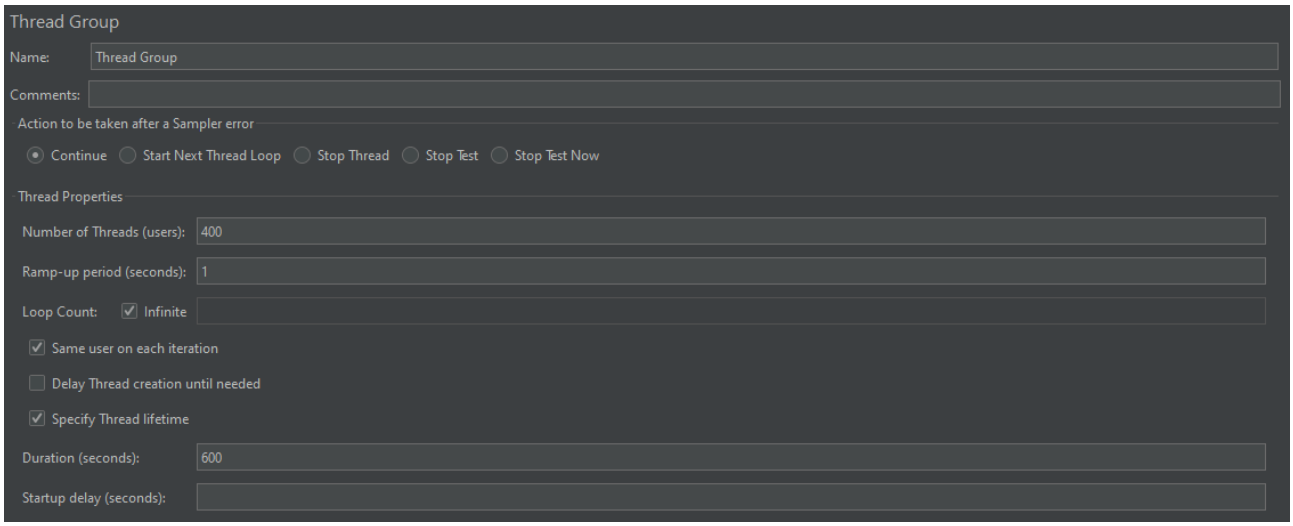
- **Funzione per Ottenere le Metriche dei Pod "*get_pod_metrics*"**: Questa funzione recupera le metriche dei pod dal cluster Kubernetes. Ecco cosa fa:
    - **Utilizzo della API Metric-Server**: Utilizza il client Kubernetes per chiamare l'API di "Metric-Server" per ottenere le metriche dei pod. Questa API fa riferimento all'addon abilitato in Minikube, presentato in precedenza.
    - **Iterazione sulle Metriche dei Pod**: Itera attraverso le metriche ottenute, estrae il nome del pod e l'uso effettivo della CPU in formato nanocore.
    - **Restituzione delle Metriche**: Restituisce le metriche dei pod, che verranno utilizzate dalla funzione di controllo di scalabilità per prendere decisioni.
    - **Gestione delle Eccezioni**: Gestisce eventuali eccezioni che possono verificarsi durante il recupero delle metriche dei pod e le registra in caso di errori.

Test plan

A questo punto, è stato necessario definite un “Test Plan” su Apache JMeter. Il "Test Plan" è un elemento fondamentale nell'ambiente di testing delle prestazioni di JMeter. Rappresenta il cuore del progetto di test e fornisce una struttura organizzativa per definire, configurare e orchestrare i test delle prestazioni delle applicazioni o servizi web. Il Test Plan è il luogo in cui si definiscono tutti gli aspetti del test, dai thread virtuali utilizzati agli scenari di utilizzo simulati e alle metriche monitorate. Di seguito viene presentata una spiegazione generale dei componenti e delle funzionalità del Test Plan utilizzato.

Thread Group

Un Thread Group è un controller di base in JMeter che definisce il comportamento dei thread virtuali (utenti simulati) durante il test.



Ecco alcuni aspetti chiave dei Thread Group:

- **Numero di Thread (Utenti Virtuali)**: Il campo specifica quante istanze virtuali degli utenti verranno simulate durante il test. Ogni thread rappresenta un utente simulato. Si è scelto di settarlo a 400.
- **Ramp-Up Period**: Il campo determina il tempo in cui verranno avviati tutti i thread definiti.
- **Duration**: Il campo definisce la durata, in secondi, del test.

HTTP Request

L'HTTP Request rappresenta una richiesta HTTP da parte dell'utente virtuale al server sotto test.

Ecco alcuni dettagli importanti:

- **Server Name or IP**: In questo campo viene inserito il nome del server o l'indirizzo IP del server di destinazione per la richiesta HTTP. Nel caso in esame, verrà avviato il web server sul “localhost”.
- **Port Number:** indica la porta a cui accedere per effettuare la richiesta HTTP
- **Protocol**: Specifica il protocollo (HTTP o HTTPS) per la richiesta.
- **Metodo**: Seleziona il metodo HTTP appropriato (GET, POST, ecc.) per la richiesta. Nel caso in esame viene utilizzato GET in quanto verrà richiesta la homepage del web server.
- **Path**: indica il path in cui la richiesta HTTP richiederà la risorsa specificata. In particolare, nel caso in esame, è stata settata a “/” che indica l’homepage del web server.

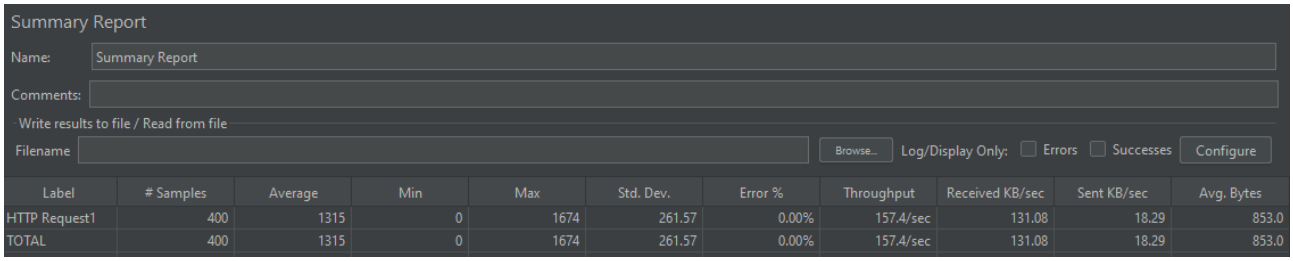
Constant Throughput Timer

Il Constant Throughput Timer è un timer utilizzato per impostare un carico costante sulla richiesta.

Nel caso in esame, è stato configurato ad un valore pari a 600 e rappresenta, appunto, il numero di richieste al minuto generate. Questo determinerà il ritmo con cui verranno effettuate le richieste durante il test.

Summary Report

Il Summary Report è un *listener* che fornisce un riepilogo dei risultati del test, inclusi i dati relativi alle richieste HTTP. È importante che la colonna “error” si esente da errori. Inoltre, fornisce una stima formale anche dei dati ricevuti come risposta.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request1	400	1315	0	1674	261.57	0.00%	157.4/sec	131.08	18.29	853.0
TOTAL	400	1315	0	1674	261.57	0.00%	157.4/sec	131.08	18.29	853.0

Testing

A questo punto parte la fase di testing vera e propria, con lo scopo di valutare lo script per lo scaling.

Port forwarding

Per prima cosa viene eseguito il seguente comando.

```
PS C:\WINDOWS\system32> kubectl port-forward nginx-deployment-57d84f57dc-nkprn 8080:80 -n test-apps
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Si tratta di un comando di *kubectl* utilizzato per eseguire il *port forwarding* all'interno di un cluster Kubernetes da una porta locale a una porta interna di un pod. Ecco una spiegazione dettagliata di cosa fa questo comando:

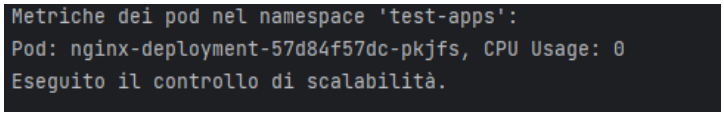
- **kubectl port-forward**: Questo è il comando principale di Kubernetes utilizzato per avviare il port forwarding tra l’ambiente locale e un pod all'interno di un cluster Kubernetes.

- **nginx-deployment-57d84f57dc-nkprn**: Questo è il nome del pod Kubernetes con cui si desidera stabilire la connessione tramite port forwarding.
- **8080:80**: Questa parte del comando specifica la configurazione del port forwarding. In pratica, avverte Kubernetes di inoltrare tutte le richieste che arrivano alla porta locale 8080 locale alla porta 80 all'interno del pod. Questo significa che qualsiasi richiesta inviata a **localhost:8080** sul computer verrà inoltrata al pod a **localhost:80** all'interno del pod specificato.
- **-n test-apps**: Questo è l'argomento opzionale che specifica il namespace in cui si trova il pod di destinazione. In questo caso, il pod si trova nel namespace "*test-apps*".

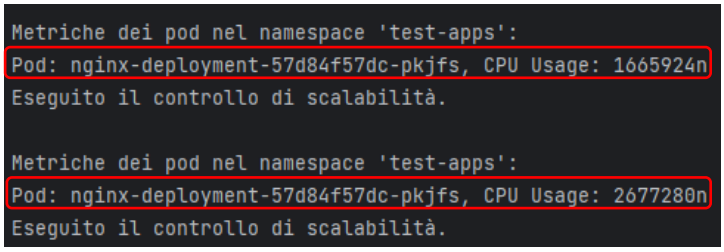
È possibile notare come, utilizzando un browser e settando l'indirizzo localhost:8080, è possibile ottenere la homepage di nginx.

Scaling

A questo punto è possibile avviare lo script Python per monitorare e scalare i pod all'interno del cluster. Il primo output ottenuto è il seguente:

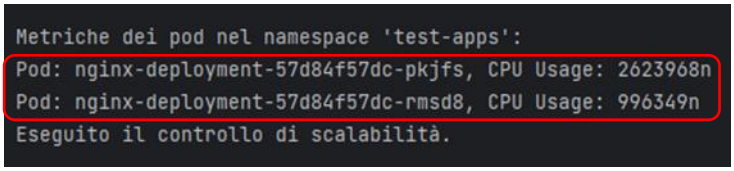


L'output garantisce che la connessione sia avvenuta con successo verso il pod nginx e permette la lettura dell'utilizzo della CPU, attualmente pari a 0 in quanto non utilizzata. Successivamente viene avviato il test tramite Apache Jmeter utilizzando il test plan configurato. La figura seguente mostra i successivi due output:



Come è possibile notare, nel primo ciclo l'utilizzo della CPU è ancora sotto la soglia di scalabilità impostata a 2000000 nanocore. Nel secondo, però, viene effettivamente superata la soglia.

Nel prossimo ciclo ci si aspetta, dunque, **l'aggiunta di un nuovo pod all'interno del cluster**. Di seguito i risultati del ciclo successivo:



È possibile apprezzare dalla figura che è stato effettivamente scalato il cluster con l'aggiunta di un nuovo pod. Da notare, però, che il pod iniziale ha effettivamente superato ancora una volta la soglia. Anche in questo caso, nel prossimo output, ci si aspetta un **ulteriore incremento di pod** nel cluster. Di seguito i risultati:

```
Metriche dei pod nel namespace 'test-apps':
Pod: nginx-deployment-57d84f57dc-75nóc, CPU Usage: 1008318n
Pod: nginx-deployment-57d84f57dc-pkjfs, CPU Usage: 2273676n
Pod: nginx-deployment-57d84f57dc-ttv7s, CPU Usage: 946959n
Eseguito il controllo di scalabilità.
```

Ancora una volta, a causa delle intensive richieste da parte dei clients di JMeter, si è superata la soglia di scalabilità. Di seguito l'output dei successivi due cicli.

```
Metriche dei pod nel namespace 'test-apps':
Pod: nginx-deployment-57d84f57dc-75nóc, CPU Usage: 0
Pod: nginx-deployment-57d84f57dc-mffmb, CPU Usage: 980311n
Pod: nginx-deployment-57d84f57dc-pkjfs, CPU Usage: 2649149n
Pod: nginx-deployment-57d84f57dc-ttv7s, CPU Usage: 0
Eseguito il controllo di scalabilità.
```

```
Metriche dei pod nel namespace 'test-apps':
Pod: nginx-deployment-57d84f57dc-75nóc, CPU Usage: 0
Pod: nginx-deployment-57d84f57dc-mffmb, CPU Usage: 980311n
Pod: nginx-deployment-57d84f57dc-pkjfs, CPU Usage: 2649149n
Pod: nginx-deployment-57d84f57dc-s9x25, CPU Usage: 1212953n
Pod: nginx-deployment-57d84f57dc-ttv7s, CPU Usage: 0
Eseguito il controllo di scalabilità.
```

È molto interessante notare come, negli outputs mostrati, siano avvenute con successo due ulteriori incrementi di pod nel cluster, ma che **si siano ottenute metriche di utilizzo della CPU pari a 0 per alcuni di essi**.

In generale, quando avviene una operazione di scaling in un cluster, è comune osservare che alcuni pod abbiano un utilizzo della CPU iniziale pari a zero. Questo fenomeno è una conseguenza naturale del processo di **scheduling e bilanciamento del carico all'interno del cluster**. Di seguito, vengono esplorate due possibili cause di questo fenomeno:

1. **Fasi dell'Avvio del Pod**: Dopo che un pod è stato schedulato su un nodo, questo passa attraverso diverse fasi di avvio tra cui: l'avvio del container, l'inizializzazione dell'applicazione e l'assegnazione delle risorse. Durante le prime fasi di avvio, l'utilizzo della CPU può essere nullo o molto basso. Questo è normale poiché il pod si sta preparando per diventare operativo.

2. **L'Equilibratura del Carico**: Un altro aspetto importante è l'equilibratura del carico tra i nodi del cluster. Kubernetes e altri sistemi di orchestrazione cercano di distribuire in modo equo i carichi di lavoro tra i nodi disponibili per sfruttare al meglio le risorse del cluster. Questo può comportare l'avvio di nuovi pod su nodi con un carico inferiore.

In sintesi, il fatto che alcuni nuovi pod abbiano un utilizzo iniziale della CPU pari a zero è una parte naturale del ciclo di vita dei container e del processo di scalabilità. È importante comprendere che il sistema di orchestrazione fa del suo meglio per distribuire i carichi di lavoro in modo efficiente, ma ciò può richiedere un po' di tempo durante le fasi di avvio.

A testimonianza di quanto detto, nel prossimo output è possibile notare che si è arrivati ad una situazione di equilibrio generale del carico di lavoro all'interno del cluster. Ogni pod ora ha un uso di CPU sotto la soglia di scalabilità e il cluster lavora in modo corretto servendo le varie richieste.