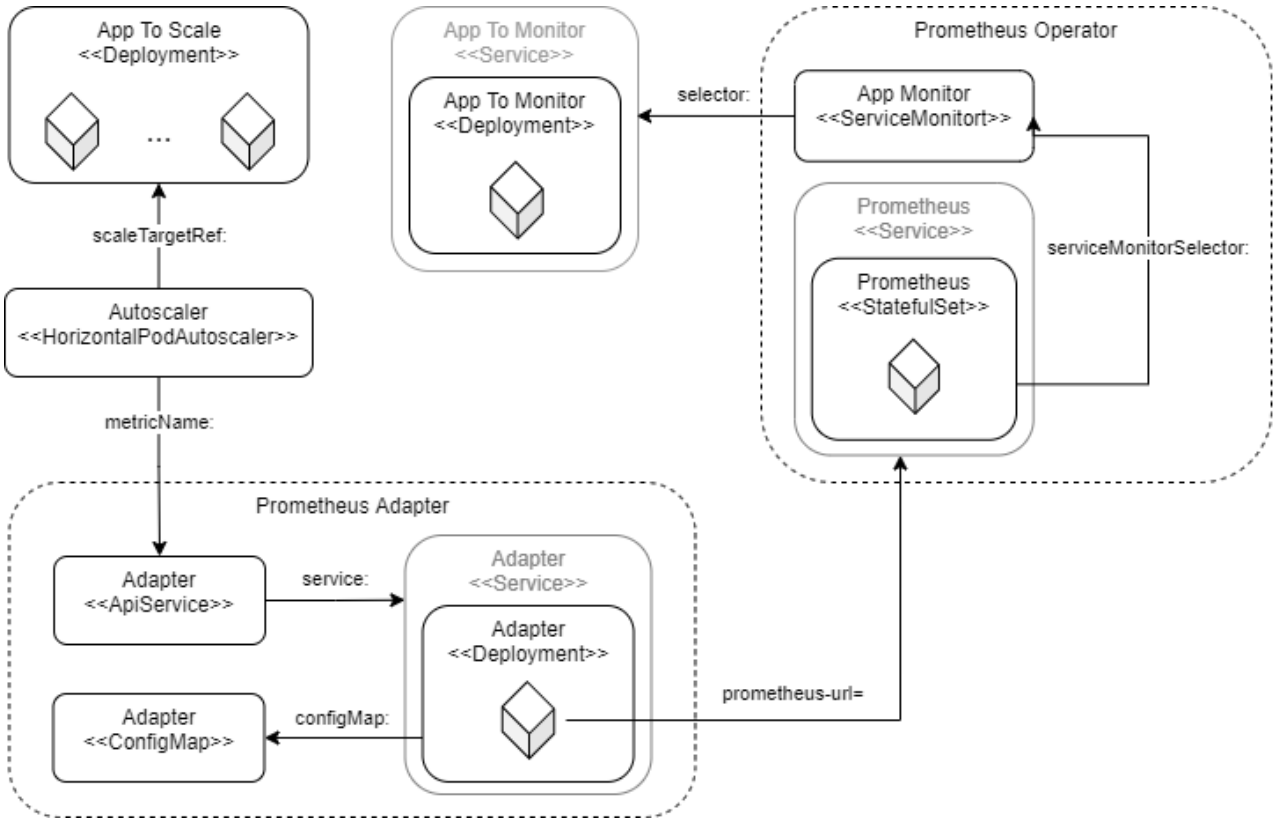
1. Custom metrics autoscaling

Openshift consente lo scaling automatico delle applicazioni dispiegate all'interno del suo Kubernetes Cluster, tramite l'utilizzo della risorsa *HorizontalPodAutoscaler* e di un *Metrics Api Server*.

Un modo per implementare questo controllo, vede l'impiego sia delle risorse *Prometheus* e *ServiceMonitor,* messe a disposizione dal *Prometheus Operator* per il monitoraggio dell'applicazione, sia di un *Metrics Api Server* realizzato tramite *Prometheus Adapter.*
Quest'ultimo, andrà a leggere le metriche memorizzate da *Prometheus* e le esporrà estendendo il comportamento del *Kubernetes API server.*

L' *HorizontalPodAutoscaler* verrà quindi configurato per l'utilizzo di una metrica esposta dall' *ApiService* del *Prometheus Adapter.*



Custom Metrics Autoscaling architecture

Il dispiegamento delle risorse necessarie alla realizzazione del meccanismo di controllo è tratto da un articolo presente sulla piattaforma *medium.com*:

https://medium.com/ibm-cloud/autoscaling-applications-on-openshift-container-platform-3-11-with-custom-metrics-6e9c14474de3

Un primo dispiegamento è stato fatto installando tutti i componenti all'interno del namespace *default* di Openshift, come descritto dall'articolo.
E' stato poi necessario apportare alcuni accorgimenti rispetto a quanto descritto, per superare problemi di compatibilità ed errori di deployment.

Installazione del *Prometheus Operator framework*

Piuttosto che installare *Prometheus Operator* tramite *Ansible playbooks,* come descritto nell'articolo, è stato utilizzato un *bundle* messo a disposizione dal progetto *prometheus-operator*, all'interno del repository GitHub

https://github.com/prometheus-operator/prometheus-operator

Nel *README.md* del progetto, viene descritto il componente e le risorse che mette a disposizione, tra cui *Prometheus* e *ServiceMonitor.*
L'installazione in Openshift deve essere eseguita da un *cluster admin,* tramite comando:

oc apply -f bundle.yaml

Per superare problemi di compatibilità, è stato necessario modificare il file *bundle.yaml* e utilizzare l'*apiVersion*

apiextensions.k8s.io/v1beta1

in sostituzione di

apiextensions.k8s.io/v1

per le risorse di tipo *CustomResourceDefinition.*

L'installazione del *bundle* potrebbe andare comunque in errore a causa di un *user id* non valido, perché non presente all'interno del range di *uid* a cui Openshift consente di effettuare il dispiegamento di un Pod (nel nostro caso del pod che a runtime svolge il ruolo di Operator).

Tra gli eventi di errore associati al Pod, sarà possibile visualizzare il corretto range di *uid* consentito. Bisognerà quindi modificare il campo

securityContext.runAsUser: <uid corrente>

presente all'interno della risorsa *Deployment* del file *bundle.yaml,* ed utilizzare un *uid* valido.

Setup di *Prometheus*

Una volta dispiegata l'applicazione da monitorare e installato l'Operator, l'articolo illustra come dispiegare un'istanza di *Prometeus*, in modo che monitori l'applicazione tramite la risorsa *ServiceMonitor.*

In sostanza, il *SeriveMonitor* viene agganciato all'applicazione tramite il campo

selector.matchLabels: <app labels>

mentre *Prometheus* viene collegato al *SeriveMonitor* tramite il campo

serviceMonitorSelector.matchLabels: <service monitor labels>

In questo modo *Prometheus* potrà collezionare di volta in volta i valori delle metriche esposte dall'applicazione.

Dispiegamento del *Prometheus Adapter*

L'articolo prosegue con il dispiegamento del *Prometheus Adapter, per il quale* sarà necessaria la creazione di risorse *RBAC* quali *ServiceAccount, ClusterRole, RoleBinding e ClusterRoleBinding,* che consentiranno all'adapter di lavorare correttamente.

Verrà poi definita la risorsa *ConfigMap,* attraverso la quale potrà essere specificata sia la query per la lettura delle metriche collezionate da *Prometheus*, sia il modo in cui queste metriche saranno "wrappate" per essere esposte sull'*ApiServer*.

Al fine di correggere alcuni errori riscontrati durante il deploy dell'adapter, si è reso necessario modificare il *ClusterRole* di nome *custom-metrics-resource-reader*, aggiungendo al campo

rules.resource: <resource list>

la voce

- configmaps

E aggiungendo al campo

rules.verbs: <operations type list>

la voce

- watch

In modo da consentire all'adapter di poter accedere correttamente alle risorse di cui ha bisogno.

Verrà infine creata una risorsa di tipo *APIService* per esporre la metrica e verrà dispiegato, tramite *Deployment,* il *Prometheus Adapter.*
Quest'ultimo sarà agganciato all'istanza *Prometheus,* dispiegata in precedenza, tramite il flag

--prometheus-url= <prometheus host>

definito in fase di deployment.
Sarà quindi possibile verificare il corretto funzionamento dell'adapter interrogando l'api server tramite il comando:

oc get --raw "/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods//<metric name>"*

Creazione di un *Horizontal Pod Autoscaler*

Step finale della configurazione, consiste nel dispiegamento di una risorsa di tipo *HorizontalPodAutoscaler*.
Questo verrà agganciato all'applicazione da scalare tramite il campo

spec.scaleTargetRef: <application deployment name>

e avrà come input la metrica esposta dal *Prometheus Adapter*, tramite configurazione del campo

metrics.pods.metricName: <metric name>

Ulteriori verifiche e configurazioni da effettuare

Per il dispiegamento di test che è stato fatto, l'*HorizontalPodAutoscaler* è stato configurato per scalare la stessa applicazione monitorata, come descritto dall'articolo, ed è stato osservato il corretto funzionamento dello scaling automatico al variare della metrica esposta.

Passaggio successivo sarà quello di verificare l'autoscaling, agganciando all' l'*HorizontalPodAutoscaler* un'applicazione diversa da quella monitorata.

Occorrerà poi configurare opportunamente i ruoli al'interno del cluster, in modo da consentire ad un utente diverso dal *cluster admin* di poter dispiegare le risorse necessarie all'autoscaling, all'interno di un namespace diverso da quello di default.

2. External metrics autoscaling

La verifica circa l'autoscaling di un'applicazione diversa da quella monitorata non ha dato esito positivo.
il problema principale è che le metriche *custom* non possono influire su pod appartenenti ad una applicazione diversa, poiché vengono raccolte solo per i pod dell'applicazione monitorata.

Per esporre il problema, supponiamo che la nostra applicazione monitorata sia *sample-app,* occorrerà quindi avere un istanza di *Prometheus* agganciata ad un *ServiceMonitor,* che a sua volta sarà collegata all'applicazione *sample-app.*
Avremo poi un istanza di *PrometheusAdapter* che andrà a leggere il valore di un ametrica (e.g. *nginx_http_requests_per_second*) da *Prometheus* e la esporrà come *custom metric api (custom.metrics.k8s.io).*

A questo punto, andando a creare l' *HorizontalPodAutoscaler:*

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: sample-app2-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample-app2
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Pods
      pods:
        metricName: nginx_http_requests_per_second
        targetAverageValue: 20m
```

Quest'ultimo andrà ad interrogare la metrica esposta dall'adapter, tramite la richiesta GET all'URI:

“/apis/custom.metrics.k8s.io/v1beta1/namespaces/tesi-delucia/pods/%2A/nginx_http_requests_per_second?labelSelector=app%3Dsample-app2”.

Rispetto a quest’uri, la risposta dell’api server del *PrometheusAdapter* è:


```
{
        "kind": "MetricValueList",
        "apiVersion": "custom.metrics.k8s.io/v1beta1",
        "metadata":
        {
                "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/tesi-
                             delucia/pods/%2A/nginx_http_requests_per_second"
        },
        "items": []
}
```


Abbiamo dunque una risposta vuota. Questo perché l’ *HorizontalPodAutoscaler,* configurato per utilizzare una metrica *custom*, formatta l’uri della *request* con il nome dell’applicazione da scalare “sample-app2”, mentre il *PrometheusAdapter* la espone per l’applicazione monitorata “sample-app”.

Se infatti proviamo ad effettuare la *request*:

```
oc get --raw "/apis/custom.metrics.k8s.io/v1beta1/namespaces/tesi-delucia/pods/%2A/nginx_http_requests_per_second?labelSelector=app%3Dsample-app"
```

otterremo dall’adapter la *response:*


```
{
        "kind": "MetricValueList",
        "apiVersion": "custom.metrics.k8s.io/v1beta1",
        "metadata":
        {
                "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/tesi-
                             delucia/pods/%2A/nginx_http_requests_per_second"
        },
        "items":
        [
                {
                        "describedObject":
                        {
                                "kind": "Pod",
                                "namespace": "tesi-delucia",
                                "name": "sample-app-674b6d4b6c-7ws6s",
                                "apiVersion": "/v1"
                        },
                        "metricName": "nginx_http_requests_per_second",
                        "timestamp": "2021-02-16T19:10:58Z",
                        "value": "33m",
                        "selector": null
                }
        ]
}
```


Avremmo quindi una risposta corretta se L’autoscaler interrogasse l’api server specificando nell’uri l’app monitorata piuttosto che quella da scalare, ma questo non accade.

Per risolvere questo problema sarà necessario che l’adapter esponga la metrica come *external metric api (external.metrics.k8s.io).*

Questo problema e la sua risoluzione sono descritti nell'articolo:

https://blog.kloia.com/kubernetes-hpa-externalmetrics-prometheus-acb1d8a4ed50

In sostanza, ciò che occorre fare è dispiegare un'istanza di *Prometheus Adapter* che esponga una *external metric api.*

Nell'articolo viene installato l'*helm chart*:

https://github.com/helm/charts/tree/master/stable/prometheus-adapter

Il file *values.yaml* del chart, dovrà essere opportunamente configurato affinchè l'adapter legga ed esponga la metrica desiderata. In particolare la sezione relativa al *ConfigMap* dell'adapter

```
...

rules:
  external:
    - seriesQuery: '{__name__="nginx_http_requests_total",namespace!="",pod!=""}'
      resources:
        overrides:
          namespace: {resource: "namespace"}
          pod: {resource: "pod"}
          service: {resource: "service"}
      name:
        matches: "^(.*)_total"
        as: "${1}_per_second"
      metricsQuery: 'sum(rate(<<.Series>>{<<.LabelMatchers>>}[2m])) by (<<.GroupBy>>)'

...
```

L'*HorizontalPodAutoscaler,* potrà essere configurato utilizzando il field *type: External,* piuttosto che *type: Pod* (che andava ad utilizzare le metriche custom)

```
...

metrics:
  - type: External
    external:
      metricName: nginx_http_requests_per_second
      targetValue: 70m

...
```

A questo punto, per l'autoscaling della nostra applicazione, potrà essere fatto utilizzando una metrica appartenente ad un'applicazione diversa.

Dispiegamenti di esempio, sia per *custom metrics* che per *external metrics*, si trovano all'url:

https://github.com/faber03/ProjectTesi/tree/main/Neo4j/Neo4j-Prometheus/AutoscalingSampleApp