# Cloud Application Autoscaling

Kubernetes & Custom Metrics

Gaël LARGER

2019 / 2020

*Master thesis in*

*Telecommunications, Services and Uses Engineering*

Content

## Figures

# 1. Introduction

During the last years, some new trending topics emerged in the fields of applications and infrastructure: *containers, microservices, IT automation...* Applications are now made of several Docker containers, and their usage in production environments became an evidence. But the management of a containers-army can be very challenging for IT operators.

A very common approach of designing and running web-scale application is called "Site-Reliability Engineering" (term from Google Engineering teams), and it's aim at threating operations as if it's a software problem.

Kubernetes is the industry-standard, container orchestrator system. It provides a kind of Platform-as-a-Service (PaaS) for executing and managing containers over a set of nodes.

More of that, it provides a lot of functionalities helping the IT team to face today's challenges:

- Monitoring the containers' health, and restart them if needed;
- Monitoring the nodes' health, and reschedule the workload running of top of them in case of failure;
- Providing a standard API for describing how an application should work (in terms of dependencies, scale, redundancy, reliability, durability, networking, storage...) – then the Kubernetes Engine will deploy and maintain the application in desired state;
- Simplifying 0-downtime software upgrade (Rolling Update strategy) and rollbacks...

Also, Kubernetes is very helpful to get full power of cloud's "rapid-elasticity" and "pay-as-you-use" model, especially by providing a component enabling people to autoscale their application in function of several metrics.

If it offers by default scalability features based on CPU and RAM usage only, the possibilities can be extended: the Kubernetes API is very open, which enables people to extend the possibilities offered by the "stock"-Kubernetes.

Also, we observe an evolution of monitoring technics, with a paradigm shifting. Rather than monitoring only on the infrastructure (on specific control points with a tool such Nagios), IT teams now also focuses their elf on the application and the services delivered to the users. New tools such Prometheus and Grafana offers an open-source solution for designing a metric-based monitoring system. These changes also impact the application, which has to expose the metrics: developers need to instrument their code.

Wrapping all together, this document describes how to setup the autoscalability in Kubernetes based metrics exposed by the application itself. The document assumes the basic knowledge of a container engine (such Docker) as a prerequisite.

# 2. Technical Architecture Documentation

## 2.1 Lab overview

### 2.1.1 Big Picture

The following illustration (figure 1) highlights the main components of the architecture.

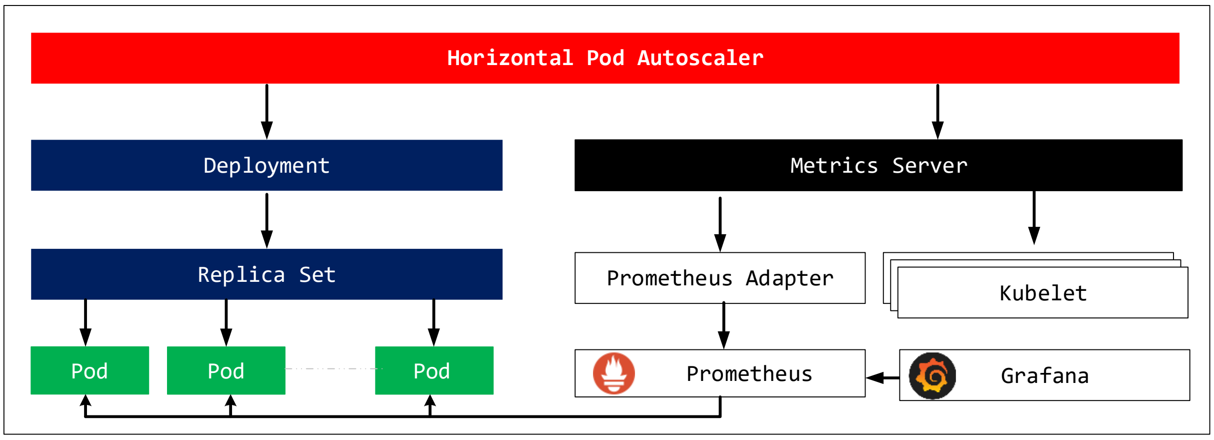


Figure 1: Lab scemantic diagram

### 2.1.2 Role of each element

#### 2.1.2.1. Pod

A pod is the smallest entity managed by Kubernetes. A pod can contain one or more container. In the case of this document, each applicative pod contains a single Java Runtime Environment (JRE) container with a Spring Boot Application.

#### 2.1.2.2. Replica Set

A replica set targets to maintain a stable set of pods. This object is used to guarantee the availability of a given number of identical pods.

We don't explicitly create a replica set, but we prefer using a Deployment (see below) that offer more functionalities and wrap the replica set object.

#### 2.1.2.3. Deployment

A deployment is a high-level object in the Kubernetes architecture. A deployment describes a desired state for the application, and a deployment controller will create, update or delete resources inside the cluster (Replica Set, Pods).

#### 2.1.2.4. Metrics server

The metric server aggregates the metrics of the Kubernetes cluster. By default, the metric server only exposes the following metrics:

- For nodes: CPU usage, Memory usage
- For pods: CPU usage, Memory usage

These metrics are gathered from the Kubelet agent, that is running on each node of a Kubernetes cluster.

It exposes some APIs to be extended – the most used is called "custom.metrics.k8s.io". Its purpose is to provide an access to any external metric from a single entry-point (the metric server). Custom metrics are provided by an adapter which needs to implement the interface contract to be compatible with the metric server.

#### 2.1.2.5. Prometheus Adapter

The Prometheus Adapter is one of these adapters. It's an Open-Source project that targets to make latest metrics gathered by the Prometheus engine available as custom metrics in the Kubernetes metrics server.

There is some commercial alternative:

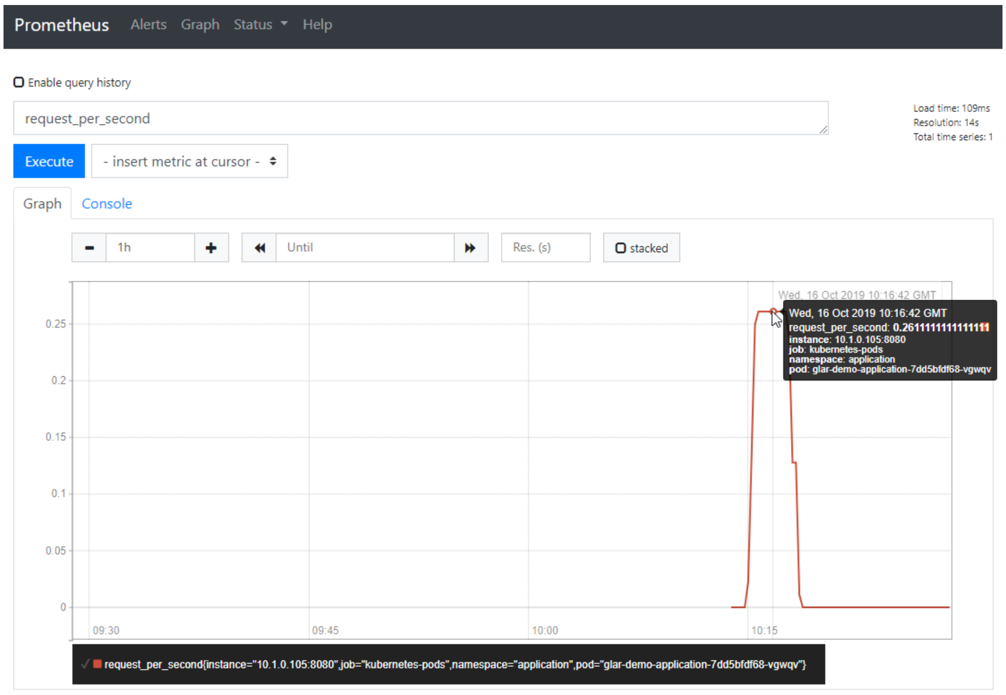
- Stack Driver on Google Cloud Platform (GCP) for Google Kubernetes Engine (GKE)
- DataDog metrics provider
- ...

The Kubernetes core developer team also gives a general framework (called Custom Metrics Adapter Server Boilerplate) to write your own adapter and make your metric system compatible with the Kubernetes metric server.

#### 2.1.2.6. Prometheus

Prometheus is an applicative monitoring engine. It's composed by 4 components:

- A metric scrapper, that will collect metrics on monitored applications
- A time-series database to store the metrics
- A web interface (figure 2) to visualize the metrics and realize queries (using the PromQL language)



request_per_second

Load time: 109ms
Resolution: 14s
Total time series: 1

Wed, 16 Oct 2019 10:16:42 GMT
request_per_second: **0.26111111111111**
**instance**: 10.1.0.105:8080
**job**: kubernetes-pods
**namespace**: application
**pod**: glar-demo-application-7dd5bfdf68-vgwqv

✓ 🟥 request_per_second{instance="10.1.0.105:8080",job="kubernetes-pods",namespace="application",pod="glar-demo-application-7dd5bfdf68-vgwqv"}

Figure 2: Prometheus WebUI overview

- A REST API, that is used by the external clients (such the adapter) to fetch latest metrics

#### 2.1.2.7. Grafana

Because the Prometheus GUI (Figure 3) offers a limited set of functionalities and a poor GUI, Grafana enables a better visualization of any metrics stored in the Prometheus Time Series database.

In this case, a single dashboard is used to display with the same time scale and reference:

- The total number of pods running in the cluster;
- The application metrics, per pod
- The application metrics, as a sum for all pods

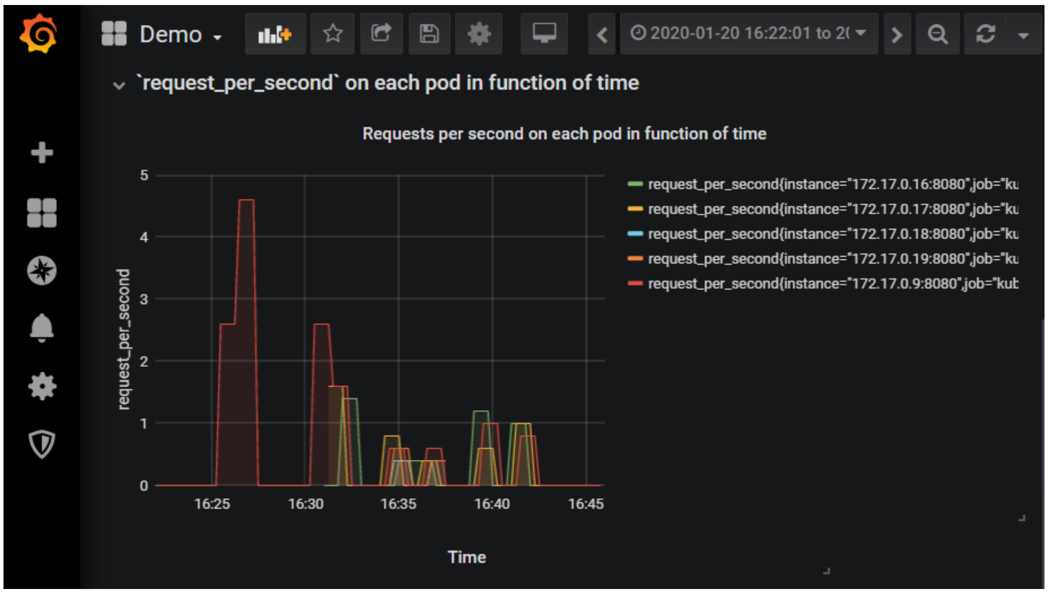

Demo
2020-01-20 16:22:01 to 2(
`request_per_second` on each pod in function of time
Requests per second on each pod in function of time
request_per_second
Time
request_per_second{instance="172.17.0.16:8080",job="ku
request_per_second{instance="172.17.0.17:8080",job="ku
request_per_second{instance="172.17.0.18:8080",job="ku
request_per_second{instance="172.17.0.19:8080",job="ku
request_per_second{instance="172.17.0.9:8080",job="kub

Figure 3: Example of Grafana dashboard

#### 2.1.2.8. Kubernetes Pod Horizontal Autoscaler

This component targets to scale-out and scale down the number of pods in a deployment depending on any one metric available through the Metrics Server[1]. So, by default, we can only scale the number of pods based on pods' CPU or Memory usage. But, because we have set up the external metrics pipeline, all metrics exposed by an application inside a pod can be used to scale the deployment.

## 2.2 Prerequisites

### 2.2.1 Lab diagram

This part describes the setup of prerequisites in a remote lab / server-room, already running, in this configuration (Figure 4).

[1] "Horizontal Pod Autoscaler." [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

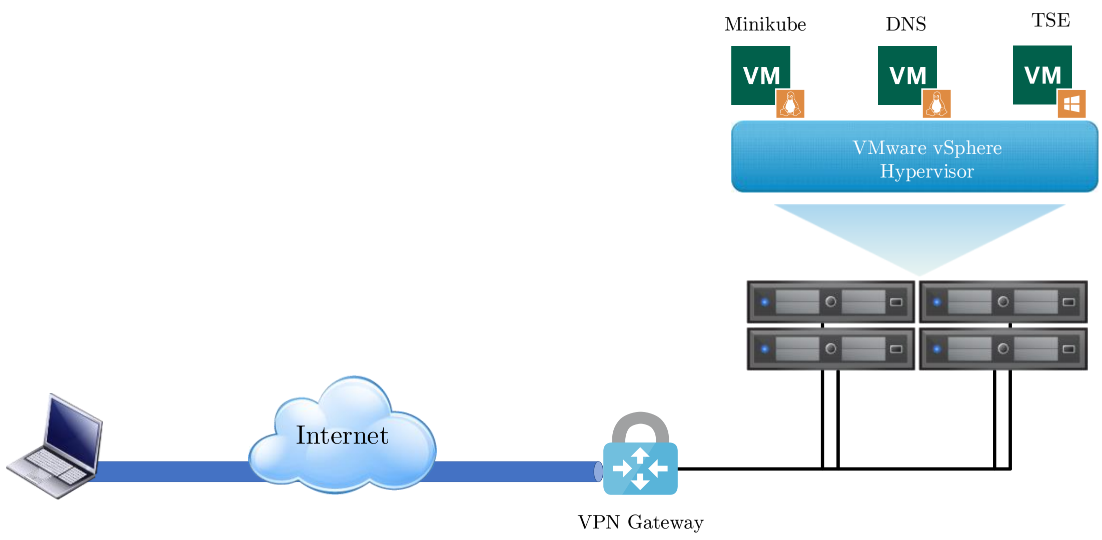


Figure 4: Lab network (layer 3) diagram

### 2.2.2 Choosing a Kubernetes distribution for development

There are many options to run a Kubernetes development cluster:

- On Windows or MacOS, the simplest way is to use Docker Desktop, and enable the Kubernetes feature. Because we will have to schedule a certain number of pods, this option requires to have enough RAM available on the local machine. This is the option I originally chose, but I was very quickly memory-bottlenecked.
- On Linux, Minikube works fine and there is a lot of resources on the Internet, that can help – if needed. It is very simple to install, even for a remote usage (by installing it on a dedicated VM running on a true hypervisor, with enough RAM for running many pods simultaneously, which makes it be an ideal solution for testing scalability mechanisms). This is the option that will be used in this document.

### 2.2.3 Installing Minikube

#### 2.2.3.1. Setup and configuration

This part describes the installation of Minikube in a dedicated VM. By default, Minikube will create itself a VM (over libvirt/KVM), by using the machine where the Minikube binary is executed as host.

To prevent nested virtualization (and the performance overhead), we need to install `docker`, `kubectl` and `minikube` on our own VM; then tell Minikube to use the current machine as the single-node Kubernetes cluster (with the option `-vm-driver=none`).

Running theses commands as root will correctly install all mandatory binaries on the system to be used with Minikube.

```
# Install Docker

apt install apt-transport-https ca-certificates curl software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -

add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
(lsb_release -cs) \
stable"

apt instal docker-ce

# Disable swap (unsupported with k8s)
swapoff -a
sed -i s%/swap.img%#/swap.img%g /etc/fstab
mount -a

# Install kubectl and socat (kubectl is dependent from socat)
apt install socat

mkdir -p /opt/kubernetes/bin

curl https://storage.googleapis.com/kubernetes-release/release/`curl -
s https://storage.googleapis.com/kubernetes-
release/release/stable.txt`/bin/linux/amd64/kubectl \
-o /opt/kubernetes/bin/kubectl

# Install Minikube
curl https://storage.googleapis.com/minikube/releases/latest/minikube-linux-
amd64 -o /opt/kubernetes/bin/minikube
```

As regular user (with sudoer rights), we can also run the command to belong to the docker group and be able to talk with the docker daemon (login in again is required, else we can use the `` `newgrp docker` `` command).

```
sudo usermod -aG docker $(whoami)
newgrp docker
```

For easier management, it can be good idea to update the `~/.bashrc` file, especially to add the minikube, helm and kubectl binaries folder to the path (source-ing the file is required to apply changes in the current shell):

```
alias k='kubectl'
PATH=/opt/kubernetes/bin:$PATH
```

#### 2.2.3.2. Bootstrapping the minikube cluster

This command will install a single-node Kubernetes cluster on the local VM using minikube:

```
minikube start --vm-driver=none --kubernetes-version='v1.16.0'
```

After that, we can export the .kube/config file to our local machine to directly run kubectl commands without SSH:

```
# On the local machine (Windows, Mac or Linux, with kubectl installed)

mkdir .kube .minikube
scp user@minikube.host:/home/user/.kube/config .kube/config
scp user@minikube.host:/home/user/.minikube/client* .minikube/
scp user@minikube.host:/home/user/.minikube/ca.crt .minikube/
```

### 2.2.4 Networking considerations

#### 2.2.4.1. Ingress overview

To access our applications running inside Kubernetes, we have many possibilities. A common one is to use an Ingress Controller, that will run at the edge between the Kubernetes networking and the external network (Figure 5):

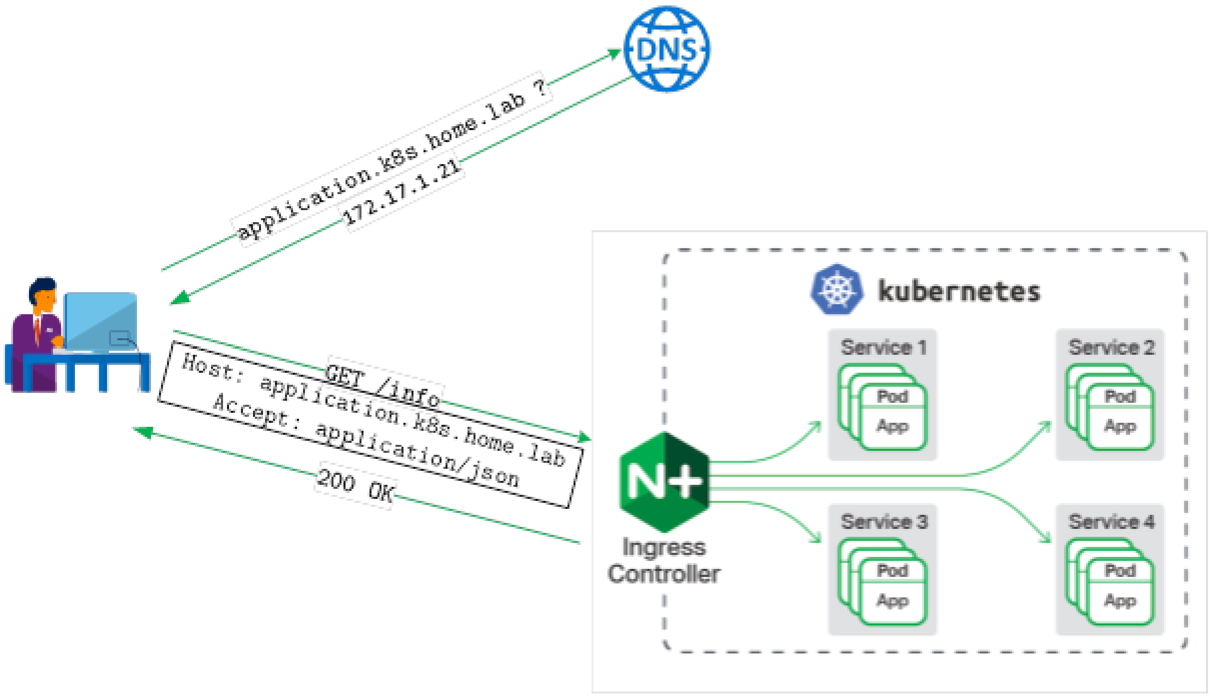


Figure 5: Nginx Ingress Controller overview

In fact, the Ingress Controller will act as reverse proxy, so the Host header passed inside the HTTP request will act as a discriminatory criterion to forward the request to the current service. So, we will need to resolve hostnames properly ouside of the cluster (all ingresses' hosnames has to be resolved to the Kubernetes master IP), and – one more time – that is showing that the DNS is a critical component.

#### 2.2.4.2. DNS configuration

I choose to use a subdomain of my existing local domain to be used for Ingresses hostnames:

- Existing domain: `home.lab`
- Domain for k8s ingresses: k8s.home.lab
- Ingresses' hostnames: `*.k8s.home.lab`

So, I just had to add a new record in my existing unbound DNS (172.17.1.21 is the IP of the Minikube VM). The `unbound.conf` looks like it:

```
server:
    interface: 0.0.0.0
    port: 53
    do-ip4: yes
    do-udp: yes
    access-control: 172.16.0.0/12 allow
    verbosity: 1

local-zone: "home.lab." static
local-data: "hlbdns01lv.home.lab A 172.17.1.5"
local-data-ptr: "172.17.1.5 hlbdns01lv.home.lab"
local-data: "hlbvct01lv.home.lab A 172.17.1.13"
local-data-ptr: "172.17.1.13 hlbvct01lv.home.lab"
[...]
local-data: "hlbk8s01lv.home.lab A 172.17.1.21"
local-data-ptr: "172.17.1.21 hlbk8s01lv.home.lab"
local-zone: "k8s.home.lab" redirect
local-data: "k8s.home.lab 86400 IN A 172.17.1.21"

forward-zone:
   name: "."
   forward-addr: 8.8.8.8
   forward-addr: 9.9.9.9
```

### 2.2.5 Helm

#### 2.2.5.1. Overview

Helm is a package manager for Kubernetes. Helm enables k8s administrators to deploy applications based on templating and dependencies mechanisms in order to prevent the complex management of yaml manifest usually used in Kubernetes.[2]

Main concepts of Helm are:

- Chart: the description of a Kubernetes package (chart.yml) and some templates of k8s manifest, that will be rendered and send to the Kubernetes API.
- Repository: Provide charts in a standardized and structed manner.
- Release: A deployed instance in the cluster.

Helm relies on a client-server model and the following architecture (Figure 6):

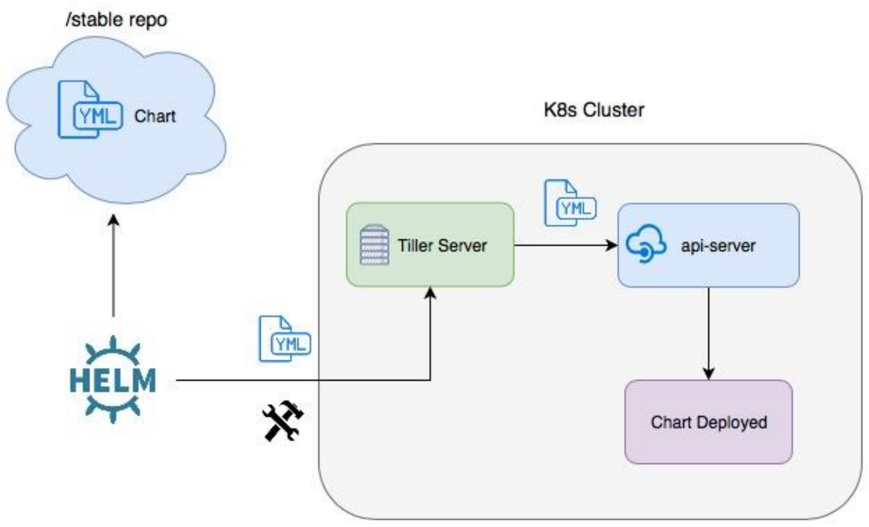


Figure 6: Helm architecture

The `helm` CLI is the client part. It is the interface between the administrator and Helm itself. `tiller` is the server part and runs inside the Kubernetes cluster. It actually communicates with the Kubernetes API.

[2] "Helm | Docs." [Online]. Available: https://helm.sh/docs/

#### 2.2.5.2. Installing Helm

##### 2.2.5.2.1. Getting the helm client

The client has to be downloaded from the helm website. It is shipped as a targz archive that we need to extract to get the binaries.

```
# Download and extract helm
curl https://get.helm.sh/helm-v2.16.1-linux-arm64.tar.gz -o /usr/local/src/helm-
v2.16.1-linux-arm64.tar.gz

cd /usr/local/src
tar xfvz helm-v2.16.1-linux-amd64.tar.gz
mv linux-amd64/{helm,tiller} /opt/kubernetes/bin/
cd -
```

If `` `/opt/kubernetes/bin/` `` is registered in the `$PATH` (see previous parts), the `` `helm` `` command should now be available.

##### 2.2.5.2.2. Installing tiller

If the kubeconfig is properly setup, the helm binary should be able to read it and so, communication with k8s.

The following command will install k8s-side components in the cluster (the –wait option will make the command to wait for the tiller pod to be ready before returning):

```
helm init --wait
```

## 2.3 Installing the lab itself

### 2.3.1 Filesytem / Git Repo overview

Some commands described in this document refers to the following file system hierarchy:

```
PS D:\master_thesis\SOURCES> tree
├───dist
│   ├───ext
│   │   ├───jmeter
│   │   ├───linux
│   │   │   ├───run-as-root
│   │   │   └───run-as-user
│   │   └───unbound
│   └───k8s
│       ├───application
│       │   ├───app
│       │   ├───code
│       │   │   └───project
│       │   └───hpa
│       ├───kube-system
│       ├───kubernetes-dashboard
│       └───monitoring
└───doc
```

The associated Git repo is [https://gitlab.com/Gaeel/k8s-app-autoscaling](https://gitlab.com/Gaeel/k8s-app-autoscaling) (public access to sources).

### 2.3.2 Kubernetes dashboard

For easier management, it could be interesting to deploy the Kubernetes Dashboard on the cluster. In the case of minikube, it's easy as running `` `minikube addons enable dashboard` ``.

Then, we can create an ingress for external access, without proxying the service via `kubectl`. In a `dashboard-ingress.yml` file, let's put this manifest:

```yaml
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubernetes-dashboard-ingress
  namespace: kubernetes-dashboard
spec:
  rules:
```

```
  - host: dashboard.k8s.home.lab
    http:
      paths:
      - path: /
        backend:
          serviceName: kubernetes-dashboard
          servicePort: 80
```

Then, we can create the resource by running the command `kubectl create -n kubernetes-dashboard -f ./dashboard-ingress.yml`.

It is pretty easy to understand. This manifest request the creation of an Ingress called "`kubernetes-dashboard-ingress`" in the namespace "`kubernetes-dashboard`", based on the host "`dashboard.k8s.home.lab`". Each HTTP(S) request will be forwarded to the service called "`kubernetes-dashboard`" and running on TCP80.

### 2.3.3 Creating namespaces

We need to create two namespaces to deploy Prometheus and the adapter on the one hand and our application on the other hand.

The file `namespaces.yaml` describes these two namespaces. Creating the namespaces is simple as running `kubectl create -f namespaces.yaml`.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring

---
apiVersion: v1
kind: Namespace
metadata:
  name: application
```

The monitoring namespaces will be used for Prometheus and the adapter, the application one for the Spring Boot application itself.

### 2.3.4 Creating a minimal application

#### 2.3.4.1. Spring Initializer

Using start.spring.io, let's create a basic Kotlin Application, using Gradle and Java 11 LTS. Required dependencies are Spring Web, H2 Database (in-memory mode), and Spring Data JPA. The full code is available on the git repository, or in appendix.

#### 2.3.4.2. The readiness controller

Let's create a minimal REST Controller, that will always answer `{‘‘status’’: ‘‘up’’}` when the app is started. It will be used by Kubernetes itself later.

```
$ curl -sk https://application.k8s.home.lab/health | jq

{

  "status": "up"

}
```

#### 2.3.4.3. Instrumenting the application

Instrumentation our application is one of the most important prerequisites to satisfy before setting up autoscaling: we need to have visibility on what is happening at the application layer.

So, we will setup a logging mechanism that will historized the requests processed by the application (like a logging database). In the case of this demo, we will store the request timestamp, HTTP status code, application latency...

Then, we should expose the metrics on a dedicated page, with a specific and standard format scrapable by Prometheus. Usually, these metrics are exposed on /metrics[3].

```
$ curl -sk https://application.k8s.home.lab/metrics
#
request_per_second 0.0
averaging_period 5.0
```

The associated code is given in the appendix.

[3] "Data model | Prometheus." [Online]. Available: https://prometheus.io/docs/concepts/data_model/.

#### 2.3.4.4. Deploying the application on Kubernetes

The first thing to do is to prepare a Docker image with a JRE and our application JAR. The following Dockerfile describe how to build the image, using an intermediate container to perform compilation of source code (a JDK is required for compilation, but a JRE is enough for execution):

```
# Build with gradle and JDK 11
FROM gradle:5.6.2-jdk11 as build

WORKDIR /app
COPY ./project/ .
RUN gradle build jar --no-daemon


# Run with a JRE
FROM openjdk:11-jre-slim

WORKDIR /app
EXPOSE 8080
COPY --from=build /app/build/libs/*.jar /app/spring-boot-application.jar

CMD ["java", "-jar", "spring-boot-application.jar"]
```

The image name should be demo-application (as configured in the Kubernetes Deployment later), so the command that can build the image is:

```
docker build -t demo-application
```

The image should now be ready, we can see it by running:

```
docker image ls
```

Pushing/pulling the image to a Docker Registry is outside of the scope of this document. So, the application will be deployed on Kubernetes with an imagePullPolicy set to IfNotPresent.

Because the image was built on the local machine, it will always be present, so it will never be pulled.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: glar-demo-application
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: glar-demo-application
      annotations:
        # Enable Prometheus scrapping
        prometheus.io/scrape: 'true'
        # Metrics are available on /metrics
        prometheus.io/path: '/metrics'
    spec:
      containers:
      - name: glar-demo-application
        # Use the built image
        image: demo-application
        # Do not pull it if it's already present
        imagePullPolicy: IfNotPresent
        # The app is listening on TCP:8080
        ports:
          - containerPort: 8080
        readinessProbe:
          # Wait 5 seconds before checking for readiness
          # (~ SpringBoot start time)
          initialDelaySeconds: 5
          # Check each 5 seconds if the app is already alive
          periodSeconds: 5
          # Heatlh check is realized via an HTTP GET on :8080/health
          httpGet:
            path: /health
            port: 8080
        resources:
          limits:
            memory: 256Mi
```

Then we will create a Kubernetes Service. What we need is just a ClusterIP service, so our service will only be available inside the cluster. For external access, we will use an Ingress that will forward requests to this service (check back the “Ingress Overview” schema if needed).

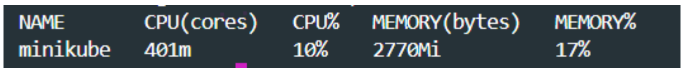
```
---
apiVersion: v1
kind: Service
metadata:
  name: glar-demo-application
spec:
  ports:
  - port: 80              # Internal service port
    targetPort: 8080      # Pods port (to send the traffic to)
  selector:
    app: glar-demo-application
  type: ClusterIP
```

### 2.3.5 Deploying the metric server

In minikube, the deployment of the metric-server is performed by enabling an addon:

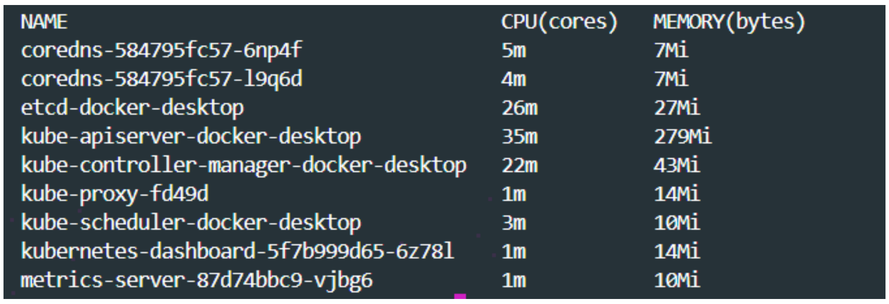
```
minikube addons enable metrics-server
```

If the deployment is successful, the command `` `kubectl top (pods | nodes)` `` should returns the CPU and Memory usage of the targeted entity :



```
NAME       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube   401m         10%    2770Mi          17%
```

Figure 7: Example of nodes metrics



```
NAME                                     CPU(cores)   MEMORY(bytes)
coredns-584795fc57-6np4f                 5m           7Mi
coredns-584795fc57-l9q6d                 4m           7Mi
etcd-docker-desktop                      26m          27Mi
kube-apiserver-docker-desktop            35m          279Mi
kube-controller-manager-docker-desktop   22m          43Mi
kube-proxy-fd49d                         1m           14Mi
kube-scheduler-docker-desktop            3m           10Mi
kubernetes-dashboard-5f7b999d65-6z78l    1m           14Mi
metrics-server-87d74bbc9-vjbg6           1m           10Mi
```

Figure 8: Example of pods metrics

The corresponding HTTP endpoint is `/apis/metrics.k8s.io/v1beta1/{pods,nodes}`.

### 2.3.6 Prometheus

#### 2.3.6.1. Installing Prometheus with Helm

First of all, we need to create a `prometheus-values.yml` file to put our custom configuration[4] values in:

```
# Very important part in order to make the interface with the prometheus-k8s-
metrics-adapter works
# scrape_interval <= k8s/prometheus adapter's discovery interval
# else, the metric will be ‘‘flapping’’ / appearing-disappearing
#
# scrape_timeout < scrape_interval
# to prevent another scrape to attend starting even if a previous one is still
```

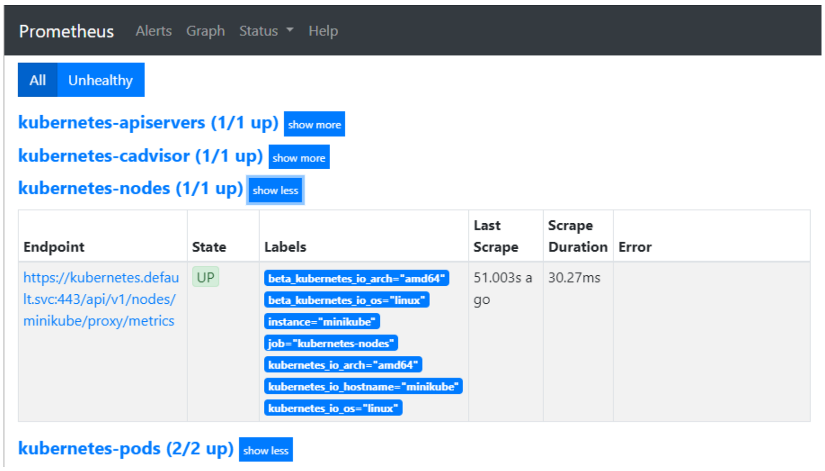
[4] “charts/stable/prometheus at master · helm/charts · GitHub.” [Online]. Available: https://github.com/helm/charts/tree/master/stable/prometheus

```
# running
global:
  scrape_interval: 15s
  scrape_timeout: 10s

server:
  # Enable ingress on the given host
  ingress:
    enabled: true
    hosts:
      - prometheus.k8s.home.lab

serverFiles:
  prometheus.yml:
       # Put the prometheus config file content here[5].
       # This one is a great base[6]
https://github.com/prometheus/prometheus/blob/
       # master/documentation/examples/prometheus-kubernetes.yml
```

Then, it can be deployed using this command:

```
helm install --values prometheus-server.yml --name prometheus \
             --namespace monitoring stable/prometheus
```

After creating the resources (`kubectl create -f monitoring/prometheus`), Prometheus is accessible on https://prometheus.k8s.home.lab. A summary of scrapped targets is available on https://prometheus.k8s.home.lab/targets.



kubernetes-apiservers (1/1 up) show more

kubernetes-cadvisor (1/1 up) show more

kubernetes-nodes (1/1 up) show less

| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|---|---|---|---|---|---|
| https://kubernetes.default.svc:443/api/v1/nodes/minikube/proxy/metrics | UP | beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="minikube" job="kubernetes-nodes" kubernetes_io_arch="amd64" kubernetes_io_hostname="minikube" kubernetes_io_os="linux" | 51.003s ago | 30.27ms | |

kubernetes-pods (2/2 up) show less

Figure 9: Checking the prometheus scaping targets status

[5] "Configuration | Prometheus." [Online]. Available: https://prometheus.io/docs/prometheus/latest/configuration/configuration/

[6] "Kubernetes & Prometheus Scraping Configuration." [Online]. Available: https://www.weave.works/docs/cloud/latest/tasks/monitor/configuration-k8s/

### 2.3.6.2. Custom Metrics Adapter for Prometheus

As for Prometheus, there is a lot of resources to deploy in order to create the adapter. Hopefully, the maintainers of the project propose the following tools to easily setup it:

- Helm Charts
- Kubernetes manifests (yaml resources files)

One more time, we will use Helm to install the adapter.

Installing the adapter is simpler. We just need to provide the prometheus-server service endpoint and port in `prometheus-adapter.yml`[7]:

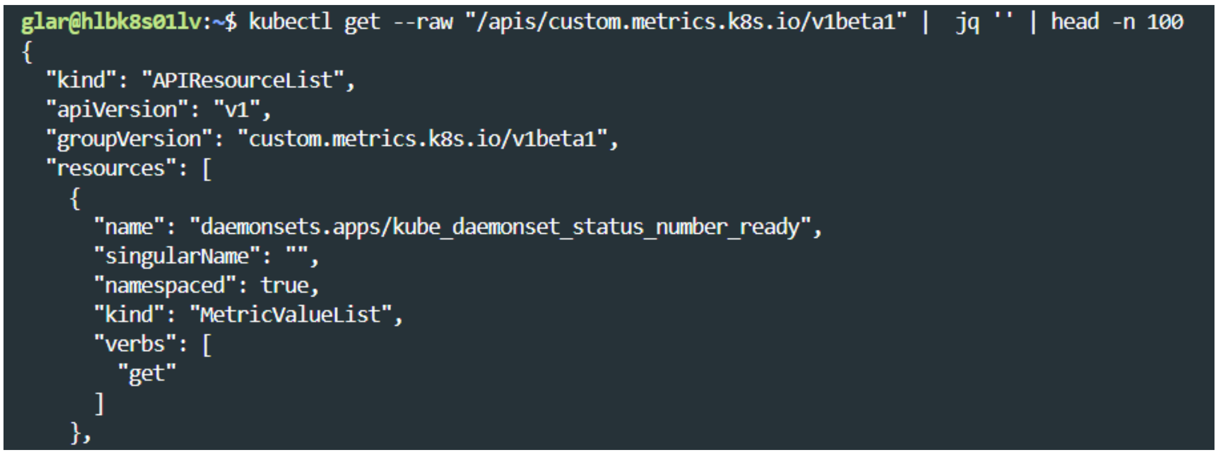
```
prometheus:
  url: http://prometheus-server.monitoring
  port: 80

# Change to 6 or 10 for debug
logLevel: 4
```

Then, we just have to wait for tiller to deploy, after running the command:

```
helm install --namespace monitoring --name prometheus-adapter \
      --values prometheus-adapter-values.yml stable/prometheus-adapter
```

After creating the resources, the command bellow should return all the metrics available in Prometheus:

```
kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1" | jq
```



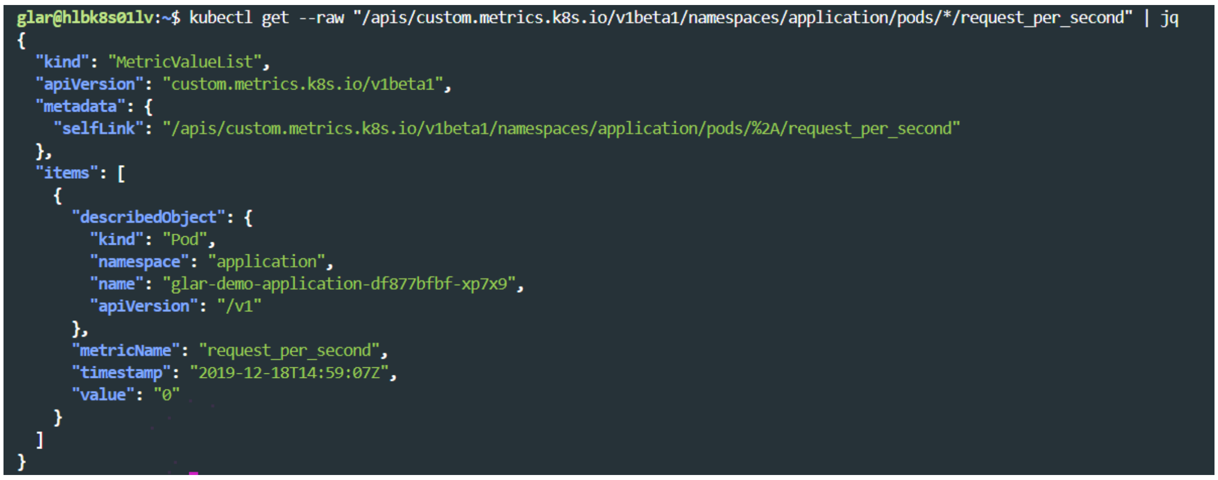
```
glar@hlbk8s01lv:~$ kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1" |  jq '' | head -n 100
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "daemonsets.apps/kube_daemonset_status_number_ready",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
```

Figure 10: Listing custom metrics

[7] "charts/stable/prometheus-adapter at master · helm/charts · GitHub." [Online]. Available: https://github.com/helm/charts/tree/master/stable/prometheus-adapter.

The application metrics are available on :

```
kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/namespaces
                                        /application/pods/*/request_per_second" | jq
```



```
glar@hlbk8s01lv:~$ kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/namespaces/application/pods/*/request_per_second" | jq
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/application/pods/%2A/request_per_second"
  },
  "items": [
    {
      "describedObject": {
        "kind": "Pod",
        "namespace": "application",
        "name": "glar-demo-application-df877bfbf-xp7x9",
        "apiVersion": "/v1"
      },
      "metricName": "request_per_second",
      "timestamp": "2019-12-18T14:59:07Z",
      "value": "0"
    }
  ]
}
```

Figure 11: Getting a custom metric value

### 2.3.7 Deploying Grafana in a fully automated way

In the same principle, we can deploy a Grafana Server for dashboarding and displaying in a better way the evolution of the metric's value in function of the time. Time is always the unit of the x-axis in Grafana graphs, because they are time-series oriented.

The following command command will deploy Grafana, setup a default datasource (Prometheus)[8] and create a first dashboard. Theses items are describe in the `grafana-values.yml` file, according to the chart specifications[9]:

```
helm install --namespace monitoring --name grafana \
      --values grafana-values.yml stable/grafana
```

This `grafana-values.yml` file contains all the configuration for this Grafana server instance:

```
# Enable ingress on the given host
```

[8] "Grafana | Prometheus." [Online]. Available: https://prometheus.io/docs/visualization/grafana/.

[9] "charts/stable/grafana at master · helm/charts · GitHub." [Online]. Available: https://github.com/helm/charts/tree/master/stable/grafana.

```yaml
ingress:
  enabled: true
  hosts:
    - grafana.k8s.home.lab

# Content of the datasources.yaml config file
datasources:
  datasources.yaml:
    # Declare a single datasource referring to our Prometheus server
    apiVersion: 1
    datasources:
    - name: Prometheus
      type: prometheus
      url: http://prometheus-server.monitoring
      access: proxy
      isDefault: true

# Declare a local dashboard provider
dashboardProviders:
  dashboardproviders.yaml:
    apiVersion: 1
    providers:
      - name: 'default'
        orgId: 1
        folder: ''
        type: file
        disableDeletion: true
        editable: false
        options:
          path: /var/lib/grafana/dashboards/default

# Configure dashbaord
dashboards:
  default:
    demo:
      json: |
        # Put the JSON config of the dashboard here
```

In this case, we don't need any persistant storage for the Grafana Server. It is fully stateless. The configuration is declared in a static way and all data displayed on the dashboard came from Prometheus.

The admin password of Grafana can be retrieved with the command:

```
kubectl get secret --namespace monitoring grafana \
      -o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

Grafana is reachable at the address https://grafana.k8s.home.lab/.

### 2.3.8 Creating an Horizontal Pod Autoscaler for the application

Now, it's possible to configure an HPA based on custom metrics.[10] Here is an example of configuration:

```
---
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: glar-demo-application-hpa
spec:
  # The scaled entity is the Deployment
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: glar-demo-application
  # We need at least one replica, and 10 pods max
  minReplicas: 1
  maxReplicas: 10
  # The scaling will be based on pods metrics
  metrics:
  - type: Pods
    pods:
      # The average of the request_per_second
      # metric for all pods should be <= to 1
      metricName: request_per_second
      targetAverageValue: 1
```

The HPA is now ready to be deployed in the application namespace:

```
kubectl create -n application -f .\application\deploy-hpa
```

The command `kubectl get hpa -n application` will return the current status of the HPA :

```
$ kubectl get hpa -n application
NAME                       REFERENCE                         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
glar-demo-application-hpa   Deployment/glar-demo-application   0/1       1         10        1          15h
```

[10] "Horizontal Pod Autoscaler Walkthrough." [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

## 2.4 Scaling tests

### 2.4.1 Prerequises, considerations and JMeter

#### 2.4.1.1. Overview

JMeter is an open-source performance and load testing tool written in Java, developed inside the Apache foundation.

In the case of this project, we will use Jmeter to describe different testing scenarios. These scenarios can be designed using a GUI application, then exported as `` `.jmx` `` files (which are very closed to XML) in ordered to be versioned in a source code manager in a proper way ; and to be run in headless mode, via the CLI.

#### 2.4.1.2. Installation

For my lab, I installed JMeter on a “management” Windows Server VM (the TSE one on the schema), with the GUI (this VM and the k8s one are running in the same hypervisors cluster, in order to not have side effects due to a possible network latency).

To install JMeter on Windows, we need to:

- Download and install a JRE (version 8 or later) – I used AdoptOpenJDK 8 with hotstop JVM
- Set the `$JAVA_HOME` environment variable
- Extract the tar.gz archive downloaded from, for example, http://mirror.nohup.it/apache//jmeter/binaries/apache-jmeter-5.2.tgz

Some of testing scenarios I designed required the plugin \`Ultimate Thread Group\` that can be installed via the plugin manager from Jmeter itself.

### 2.4.2 Test 1: Constant load

The first test will simulate a constant load. The HPA is configured to maintain an average of 1 request/second per pod.

The JMeter scenario's configuration is the following:

- 1 thread group
- 5 threads
- HTTP Requests on `https://application.k8s.home.lab/info`
- 1100ms Constant timer

So, this will simulate 5 users accessing the application and performing each 1 request each 1.1 second. Therically, we should get a total of 4.54 req/sec, so the ideal scale is 5 pods. In reality, and because we are not running a real-time Kernel / OS, there can be a slight difference between what is scheduled by Jmeter at the Application layer and what is really transmitted.

These first Grafana graph (Figure 12 and 13) presents the requests received. An absence of point for a given time value means that the pod doesn't exist at the associated timestamp:

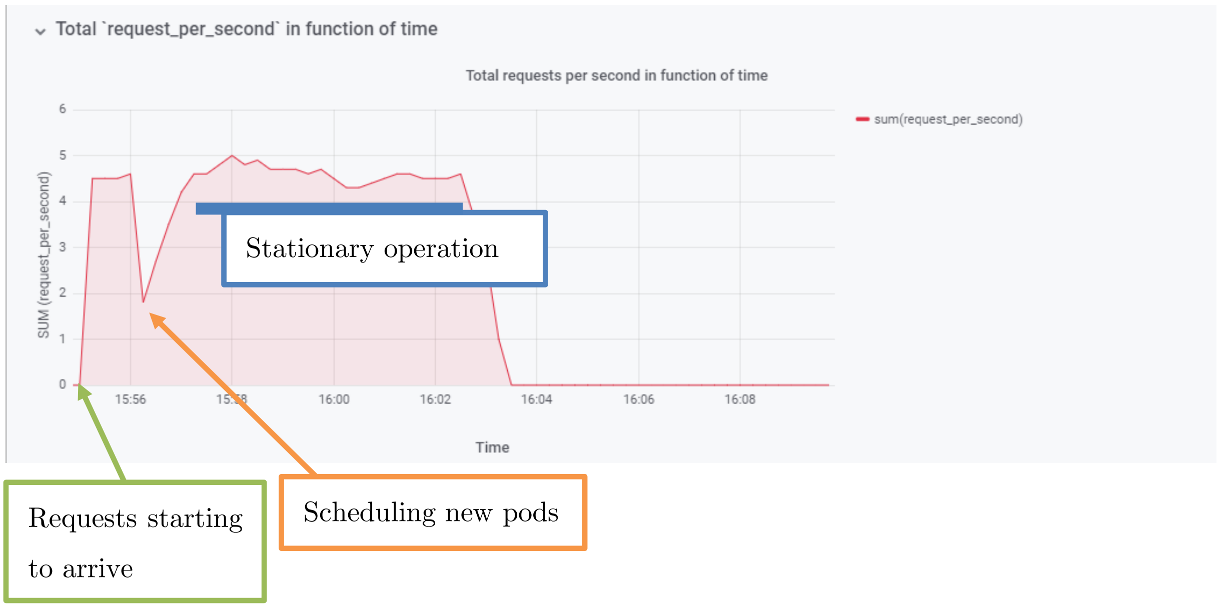


Figure 12: Scaling test with constant load pattern – HTTP requests total

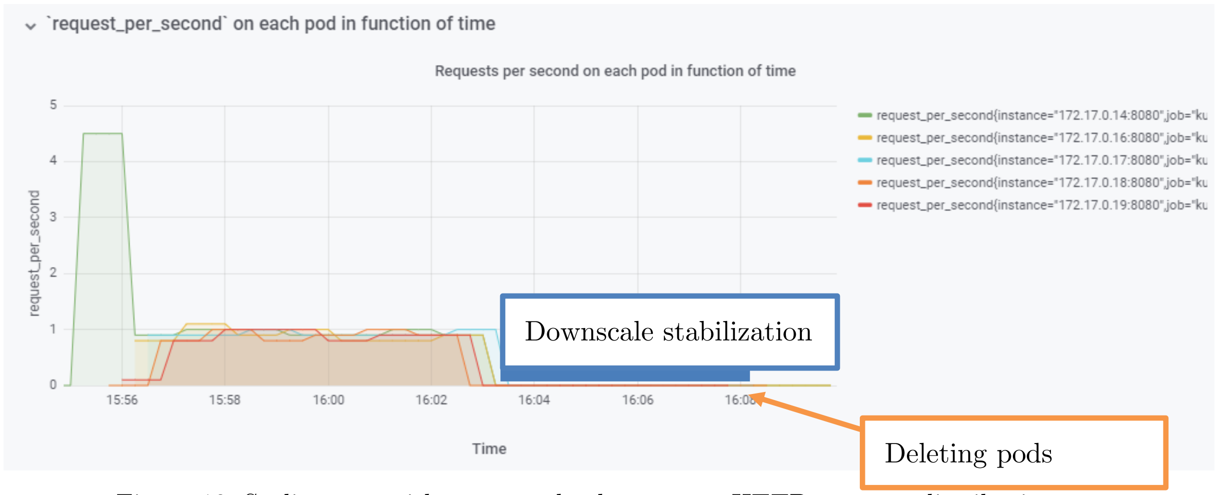


Figure 13: Scaling test with constant load pattern – HTTP requests distribution

We can see that as soon as the requests arrived to the application, the metric is raising up, and new pods are started in order to spread the load and respect the metric target. When the load peak is over, the deployement is downscaled after 5 minutes (configurable value, see below). That's what we can see on the following graph (same time scale) :

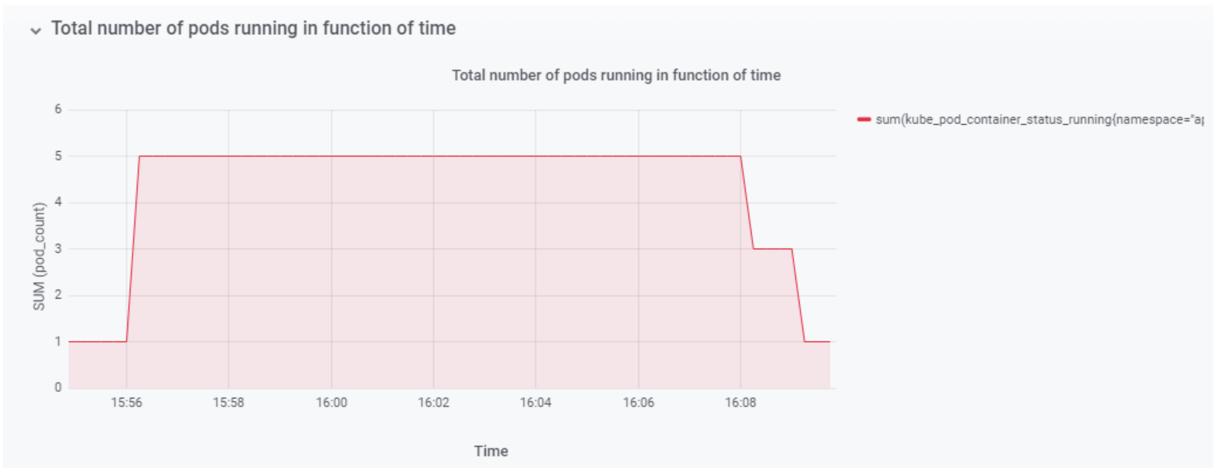
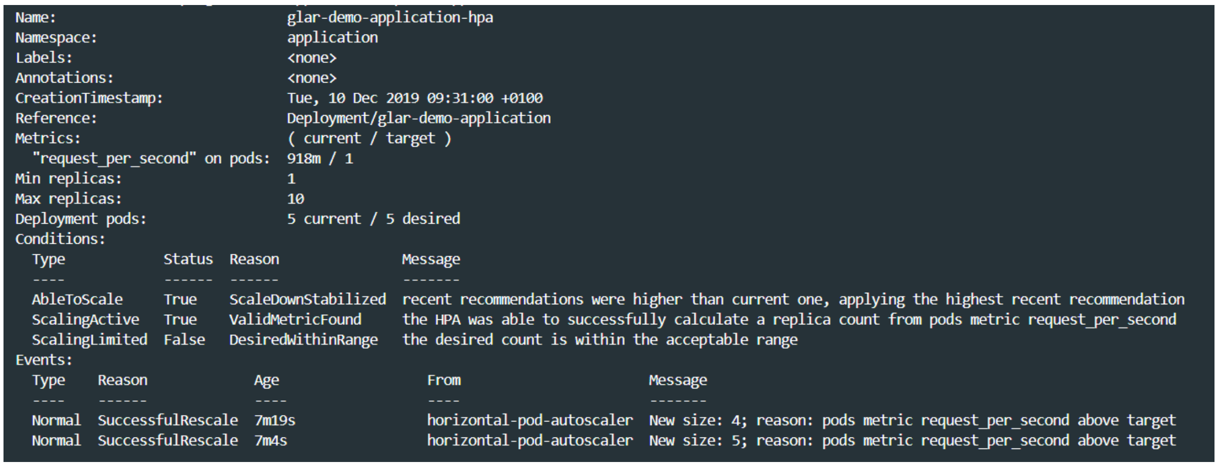


Figure 14: Scaling test with constant load pattern – Running pods

Now, let's check the logs. The `kubectl describe hpa/glar-demo-application-hpa -n application` is very helpful to better understand what is happening. The following output is returned by the command after the scale-up operation:

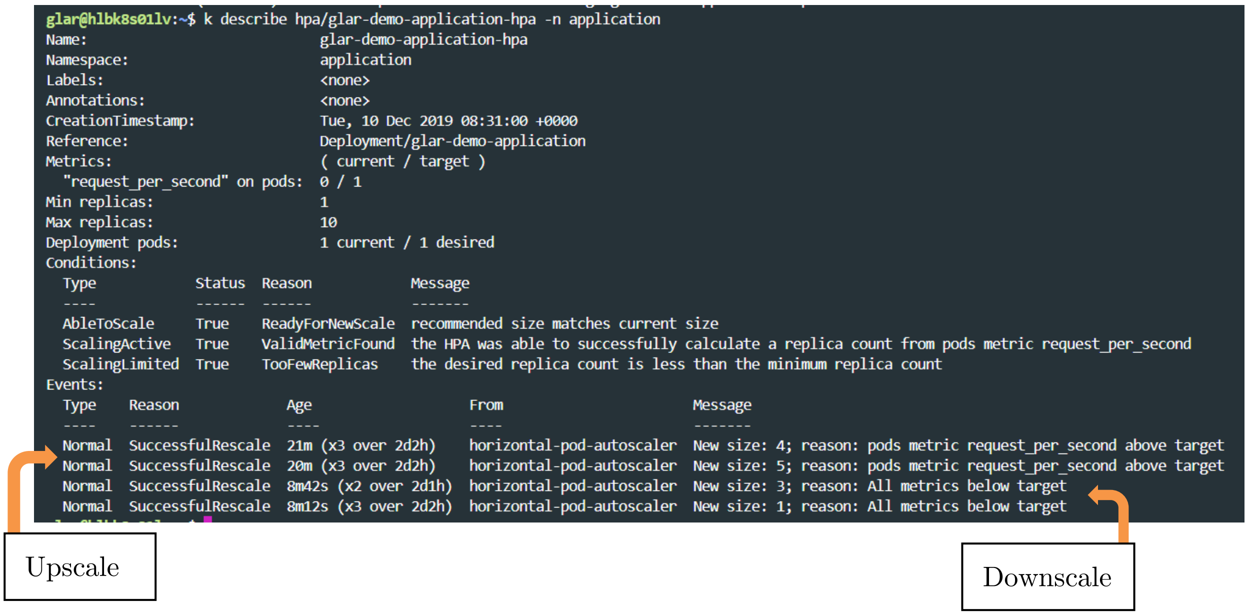


```
Name:                             glar-demo-application-hpa
Namespace:                        application
Labels:                           <none>
Annotations:                      <none>
CreationTimestamp:                Tue, 10 Dec 2019 09:31:00 +0100
Reference:                        Deployment/glar-demo-application
Metrics:                          ( current / target )
  "request_per_second" on pods:   918m / 1
Min replicas:                     1
Max replicas:                     10
Deployment pods:                  5 current / 5 desired
Conditions:
  Type            Status  Reason               Message
  ----            ------  ------               -------
  AbleToScale     True    ScaleDownStabilized  recent recommendations were higher than current one, applying the highest recent recommendation
  ScalingActive   True    ValidMetricFound     the HPA was able to successfully calculate a replica count from pods metric request_per_second
  ScalingLimited  False   DesiredWithinRange   the desired count is within the acceptable range
Events:
  Type    Reason             Age                  From                       Message
  ----    ------             ----                 ----                       -------
  Normal  SuccessfulRescale  7m19s                horizontal-pod-autoscaler  New size: 4; reason: pods metric request_per_second above target
  Normal  SuccessfulRescale  7m4s                 horizontal-pod-autoscaler  New size: 5; reason: pods metric request_per_second above target
```

Figure 15: Scaling test with constant load pattern – HPA rescale logs

We see that some events were triggered: *SuccessfulRescale*, New size N, because the metrics was above the target.

In the case of downscale, we can observe the same mechanism. The deployment was rescaled because the metric was bellow target.

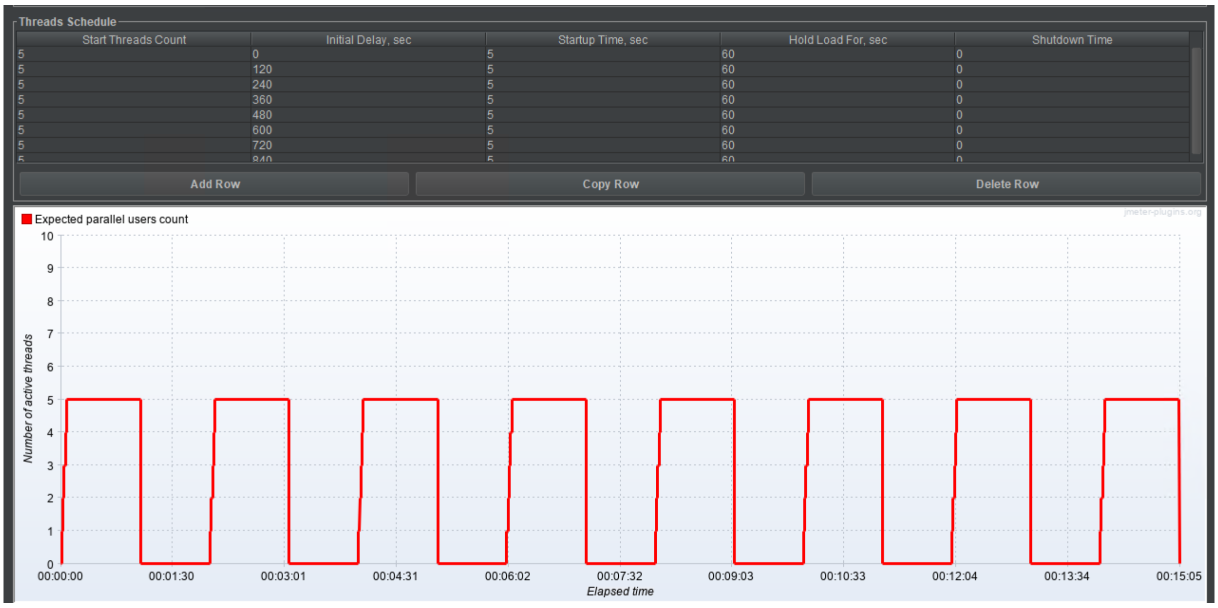


```
glar@hlbk8s01lv:~$ k describe hpa/glar-demo-application-hpa -n application
Name:                                glar-demo-application-hpa
Namespace:                           application
Labels:                              <none>
Annotations:                         <none>
CreationTimestamp:                   Tue, 10 Dec 2019 08:31:00 +0000
Reference:                           Deployment/glar-demo-application
Metrics:                             ( current / target )
  "request_per_second" on pods:      0 / 1
Min replicas:                        1
Max replicas:                        10
Deployment pods:                     1 current / 1 desired
Conditions:
  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    ReadyForNewScale  recommended size matches current size
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a replica count from pods metric request_per_second
  ScalingLimited  True    TooFewReplicas    the desired replica count is less than the minimum replica count
Events:
  Type    Reason             Age                From                       Message
  ----    ------             ----               ----                       -------
  Normal  SuccessfulRescale  21m (x3 over 2d2h)   horizontal-pod-autoscaler  New size: 4; reason: pods metric request_per_second above target
  Normal  SuccessfulRescale  20m (x3 over 2d2h)   horizontal-pod-autoscaler  New size: 5; reason: pods metric request_per_second above target
  Normal  SuccessfulRescale  8m42s (x2 over 2d1h) horizontal-pod-autoscaler  New size: 3; reason: All metrics below target
  Normal  SuccessfulRescale  8m12s (x3 over 2d2h) horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

Figure 16: Scaling test with constant load pattern – HPA downscale logs

### 2.4.3 Test 2: Periodic On Off – 50% duty cycle

Now, let's use a Custom Thread Group in Jmeter to create a load pattern:



Threads Schedule

| Start Threads Count | Initial Delay, sec | Startup Time, sec | Hold Load For, sec | Shutdown Time |
|---|---|---|---|---|
| 5 | 0 | 5 | 60 | 0 |
| 5 | 120 | 5 | 60 | 0 |
| 5 | 240 | 5 | 60 | 0 |
| 5 | 360 | 5 | 60 | 0 |
| 5 | 480 | 5 | 60 | 0 |
| 5 | 600 | 5 | 60 | 0 |
| 5 | 720 | 5 | 60 | 0 |
| 5 | 840 | 5 | 60 | 0 |

Figure 17: Descibing a periodic load pattern with JMeter and Ultimate Thread Group

As shown in the graph above the scenario is a loop made of the following actions:

- Start 5 Threads for 60 seconds, one request each 1.1s per Thread

- Stop all Threads for another 60 seconds

We obtain the following graphs:

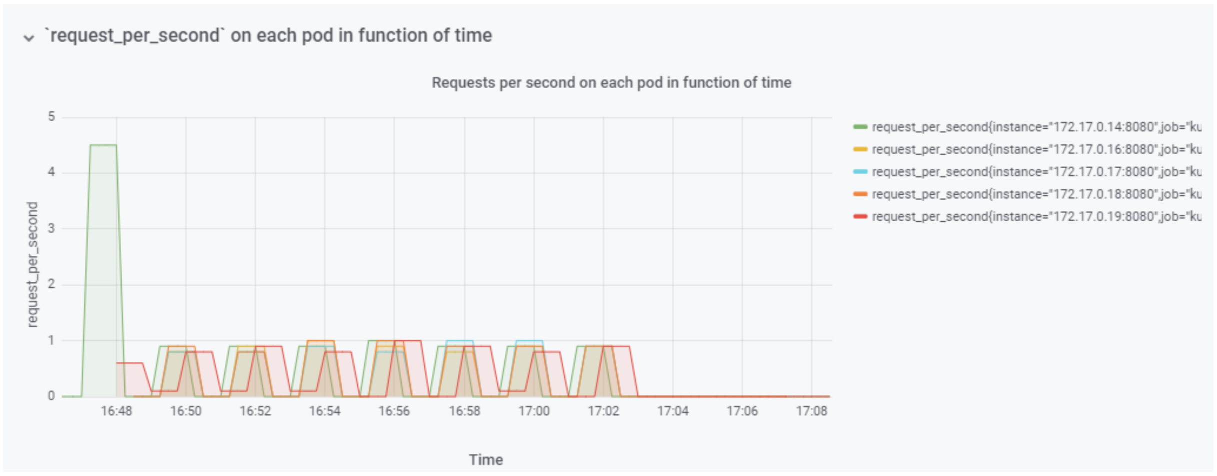


Figure 18: Scaling test with periodic load pattern – HTTP requets distribution

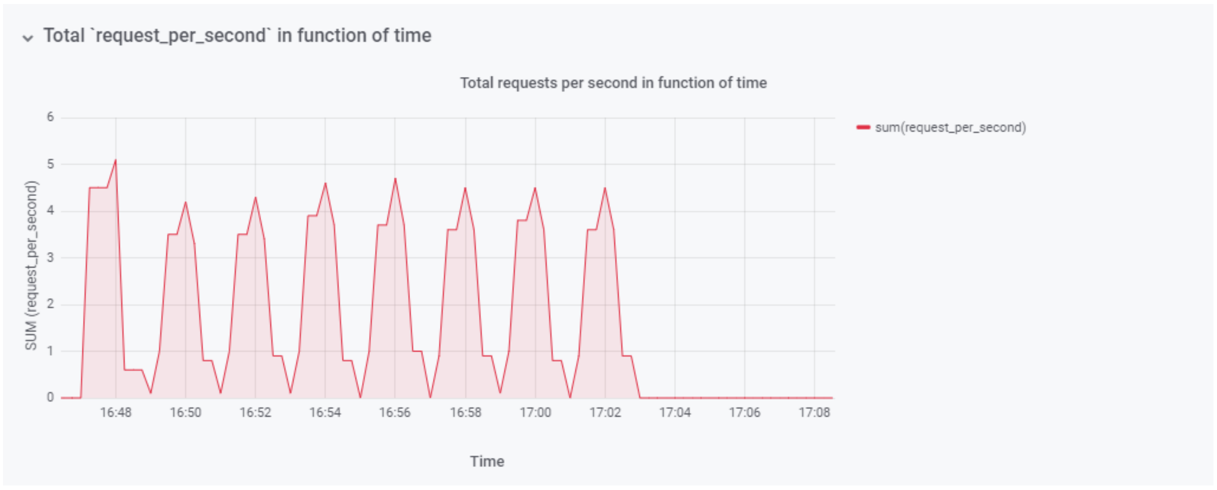


Figure 19: Scaling test with periodic load pattern – HTTP requets total

During the first load slot, we can see that all the load will be sent to the pod that was already running on the cluster. But, because the metric will raise above the target, and the HPA will schedule more pods in the cluster.

Due to the application boot time (Spring Boot with Spring Data and Tomcat needs approximatively 10 seconds to be up and ready), new pods will not be ready immediately, so not "heathy" from the point of view of k8s the readiness check and no load will be send to them before they are fully up and running. This appends during the first "no-load interval", so when the load start to increase again in the second period, they are ready, so receive traffic which is correctly spread between all pods.

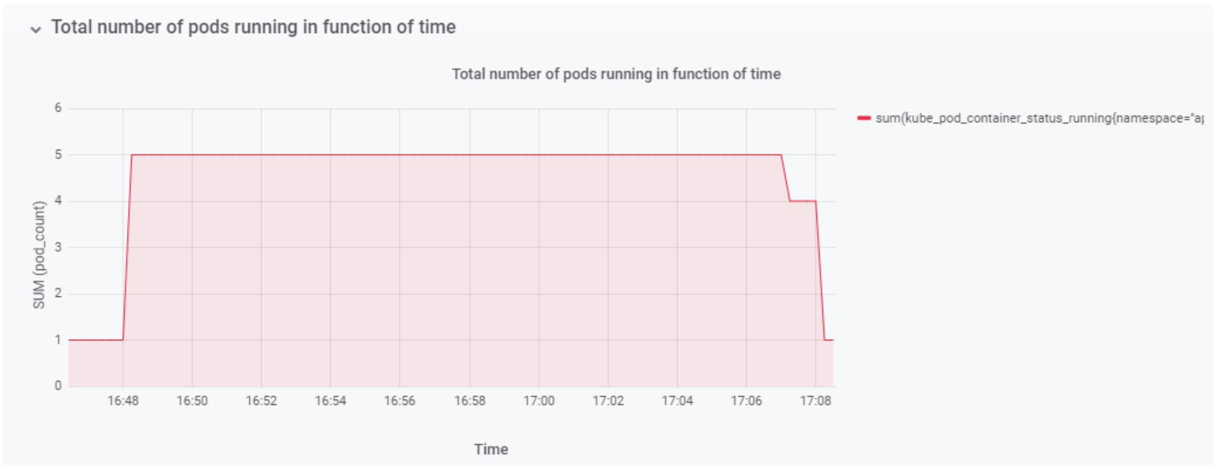


Figure 20: Scaling test with periodic load pattern – Running pods

This example is interesting. We remark that we have to take in consideration the following properties when choosing a downscale-stabilization $\tau$ value. Let's consider $\Delta_{peak}$ as the time elapsed between two load peaks (in a periodic case $\Delta_{peak}= DutyCyle * T$).

We need to have: $\boldsymbol{\tau} > \boldsymbol{\Delta_{peak}}$ , which mean that the downscale-stabilization $\boldsymbol{\tau}$ time should be longer than the duration between two load peaks, else, some pods can be deleted in the meantime between load peaks and k8s will have to re-schedule them at the next load peak.

This is not optimal, due to the delay between the load increase and the readiness of new pods schedule. Scaling-up is a long and expansible (in terms of time and resources operations). The delay for scaling up can be a bit long due to the process between the "metric-raise event" and new pods to be ready to receive traffic:

- The metric raises on pods
- Prometheus scrape metrics with a rate defined through the Prometheus Scrape Interval $\Delta_{promScrape}$. A delay between metric raise and Prometheus next scrape is defined as $\Delta_{loadToNextScrape}$
- The metrics is requested by the HPA with an interval $\Delta_{hpaPull}$ (the value is not given in the docs)
- The HPA scales the deployment and new pods are schedules. Application is booting in theses pods
- The application is started $\Delta_{appBoot}$
- The readiness check changed to green (status ready) for new pods that just booted : readinessProbe is configurable with two parameters (initialDelaySeconds $\Delta_{appReadiness}$ and periodSeconds $T_{appReadinessRecheck}$)

In the worst case, we have a total delay between HPA rescale order and new pods to be ready equal to:

$$\Delta_{totalDelay} = \Delta_{loadToNextScrape} + \Delta_{hpaPull} + \Delta_{appBoot} + N * T_{appReadinessRecheck}$$
$$\Delta_{totalDelay} = \Delta_1 + \Delta_2 + \Delta_3 + N * T_{ready}$$

with $N \geq 0$

Example:

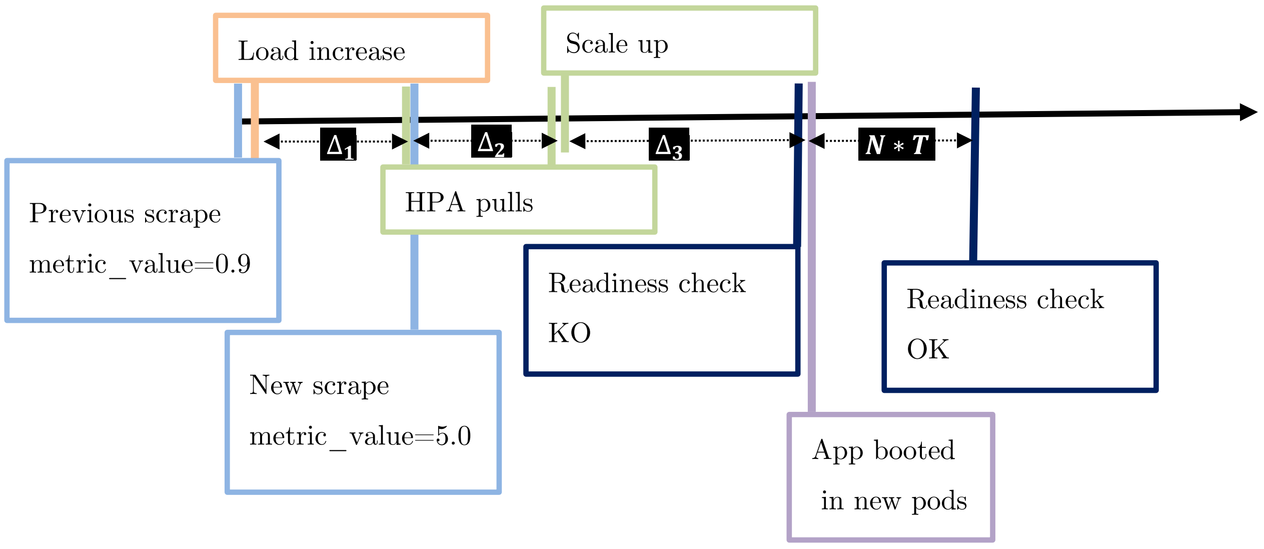


Figure 21: HPA latency

So, it is very important to keep the pods running between load peaks. That is what we will confirm in the next test.

### 2.4.4 Test 3: Aperiodic On Off – Variable duty cycle

We now describe the following scenario in JMeter:

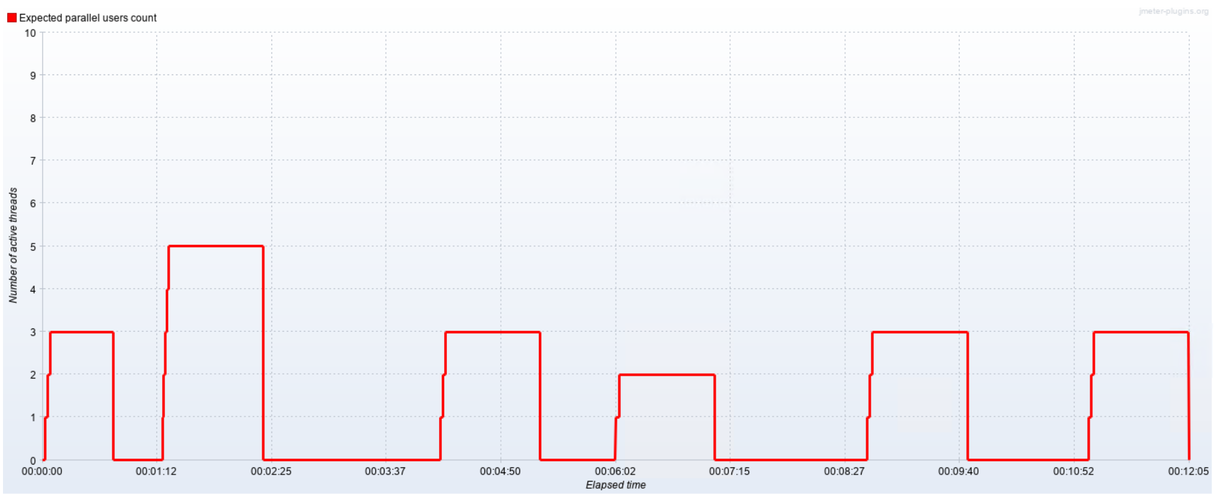


Figure 22: Descibing an aperiodic load pattern with JMeter and Ultimate Thread Group

By running the scenario, we get the following result:

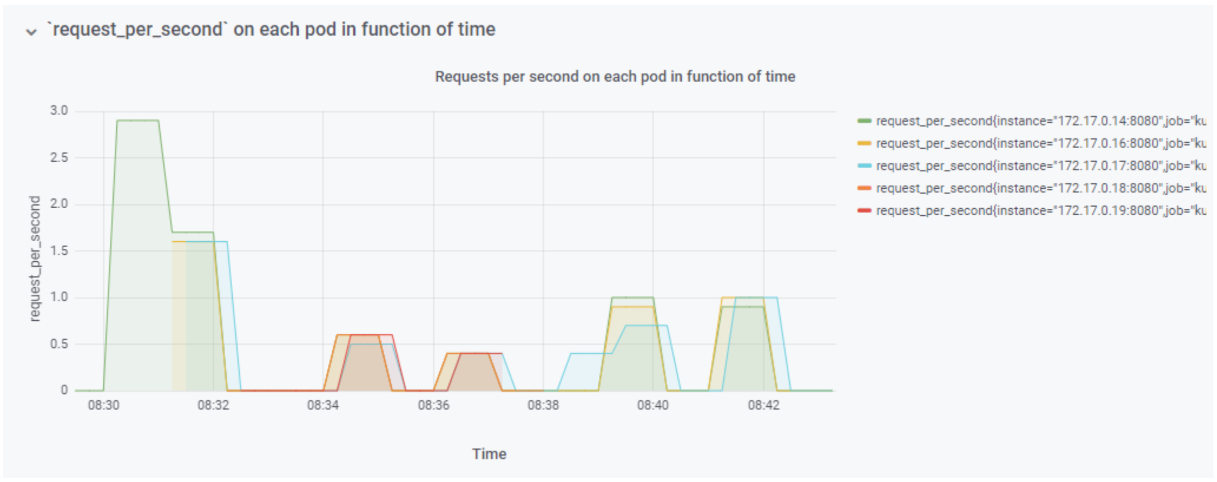


Figure 23: Scaling test with aperiodic load pattern – HTTP requets distribution

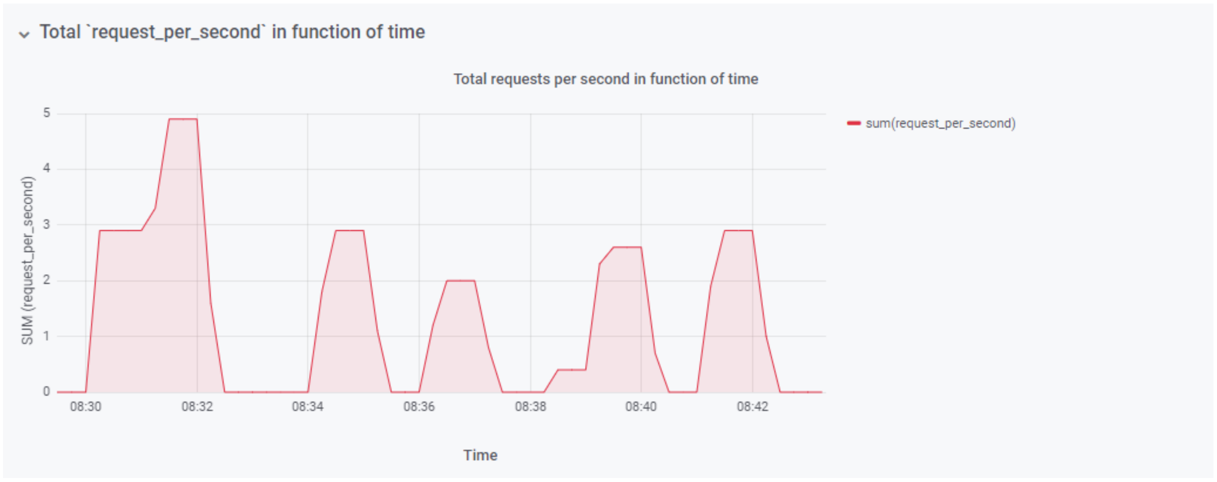


Figure 24: Scaling test with aperiodic load pattern – HTTP requets total

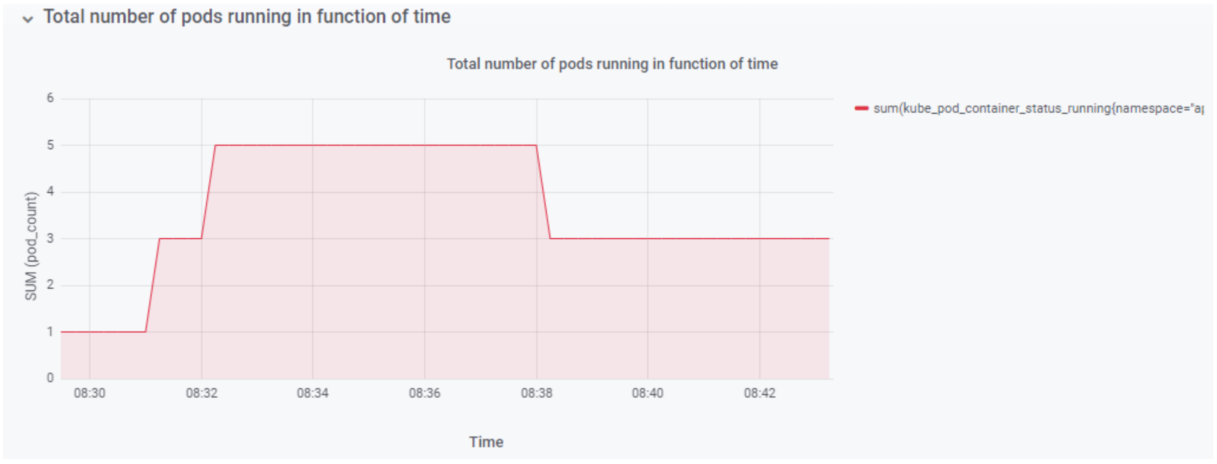


Figure 25: Scaling test with aperiodic load pattern – Running pods

What we observe:

- The first peak sets the metrics `request_per_second=3.0`, so the HPA rescale the Deployment with 3 pods. Due to the boot time (see the previous part), theses pods are not available until the next peak.
- The second peak pushed again the metric bellow the target, so the deployment is again rescaled. We now have 5 running pods.
- The two next peaks are correctly handled by the infrastructure and spread on all available pods. There is now too much pods running.
- The scaledown-stabilization interval acts as a sliding window. Because the metric didn't have been under 4.0 requests per second since more than 5 minutes, the HPA rescale the deployment to 4 pods.
- The penultimate peak is handled on 4 pods.
- Another time, and because the metric has been under the target for 5 minutes, the deployment is rescaled to 3 pods, so the last peak is handled on 3 pods.

Here, we can see that the behavior of the HPA is pretty convincing with this scaledown-stabilization interval. The condition set on the metric value is well respected:

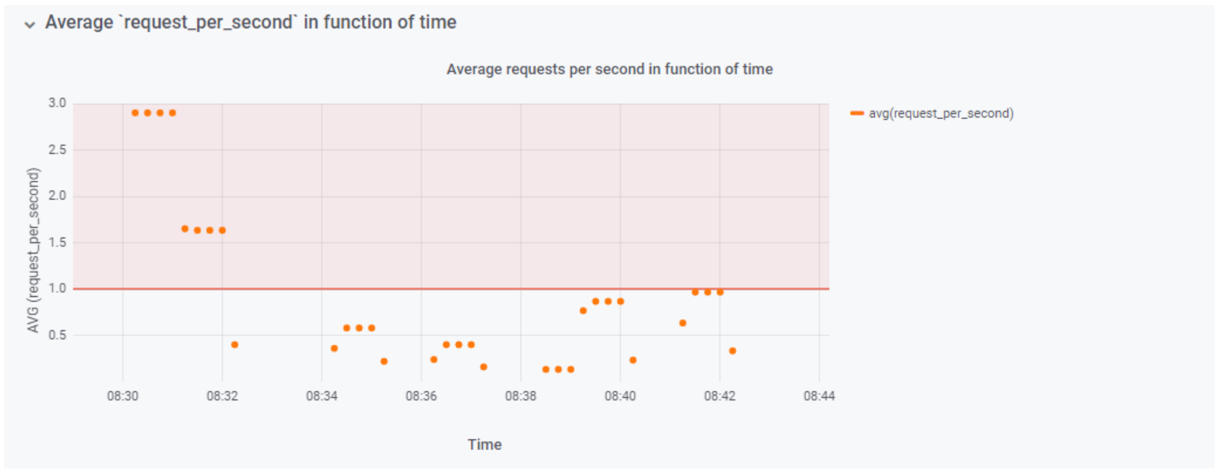


Figure 26: Scaling test with aperiodic load pattern – Average of HTTP request handeled per pod per second

## 2.5 Customizing the downscale-stabilization value

### 2.5.1 Goals

The objective of this part is to demonstrate the importance and the constraints linked to the choice of a value for the time waited by the HPA ($\boldsymbol{downscale - stabilization\,\tau}$) to downscale the Deployments.

We already determined that the optimal value is to have $\boldsymbol{\tau}$ as close as possible to $\Delta_{\boldsymbol{peak}}$, which can be very tricky to determine because load patterns are rarely periodic in reality. We will verify this by overriding the default value.

As a reminder, the default value is 5 minutes.

### 2.5.2 Configuration

This parameter is configured at the cluster level. We can't set it at the granularity of a single HPA, which is an important constraint when using a mutualized k8s cluster for many applications that can experience different load patterns.

The configuration is easy to do. It needs to be performed on the k8s master node (but we are running a single-node cluster with Minikube, so we just have an "all-in-one" node).

1. As root, edit the kube-controller-manager config file:

   ```
   vim /etc/kubernetes/manifests/kube-controller-manager.yaml
   ```

2. Change the container kube-controller-manager's command, and add the following parameter:

   ```
   spec:
     containers:
     - command:
       - kube-controller-manager
       - --authentication-kubeconfig=/etc/kubernetes/controller-manager.conf
       - --authorization-kubeconfig=/etc/kubernetes/controller-manager.conf
         […]
       - --horizontal-pod-autoscaler-downscale-stabilization=1m0s
   ```

   Here, it is set to one minute.

3. Save the file, exit vim and restart the kubelet service. Then check that the service is running / has correctly restarted.

   ```
   systemctl restart kubelet && systemctl status kubelet
   ```

### 2.5.3 Testing the new configuration

In this part, we will replay the last test, with new downscale stabilization values.

#### 2.5.3.1. 1 minutes downscale stabilization

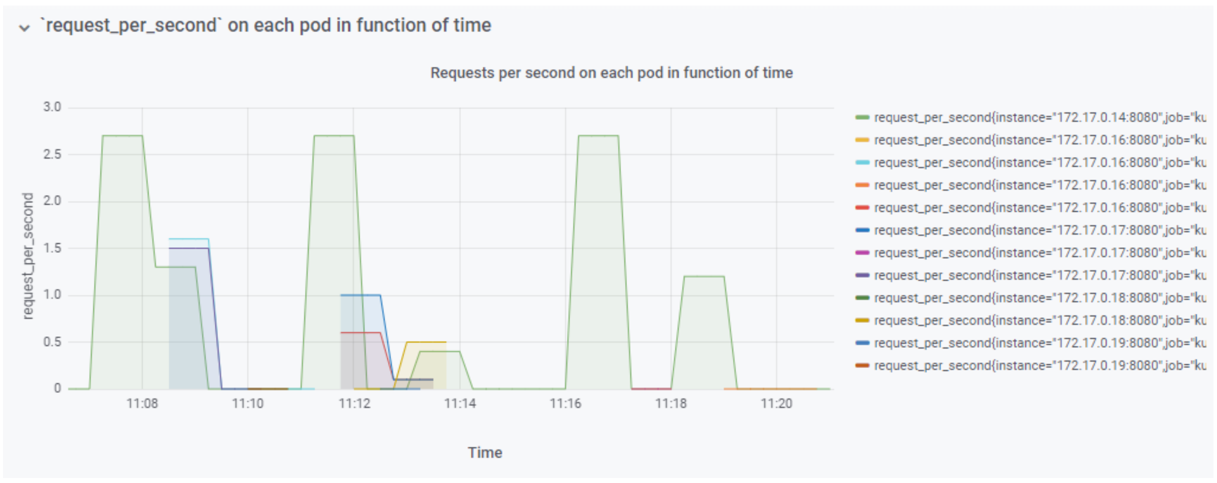


Figure 27: Scaling test with aperiodic load pattern, 1m downscale stabilization – HTTP requets distribution

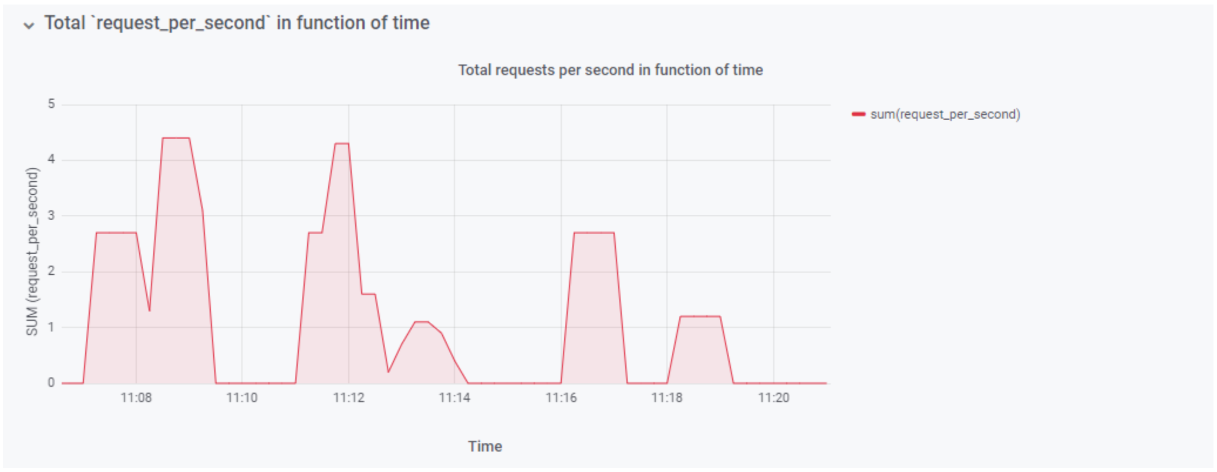


Figure 28: Scaling test with aperiodic load pattern, 1m downscale stabilization – HTTP requets total

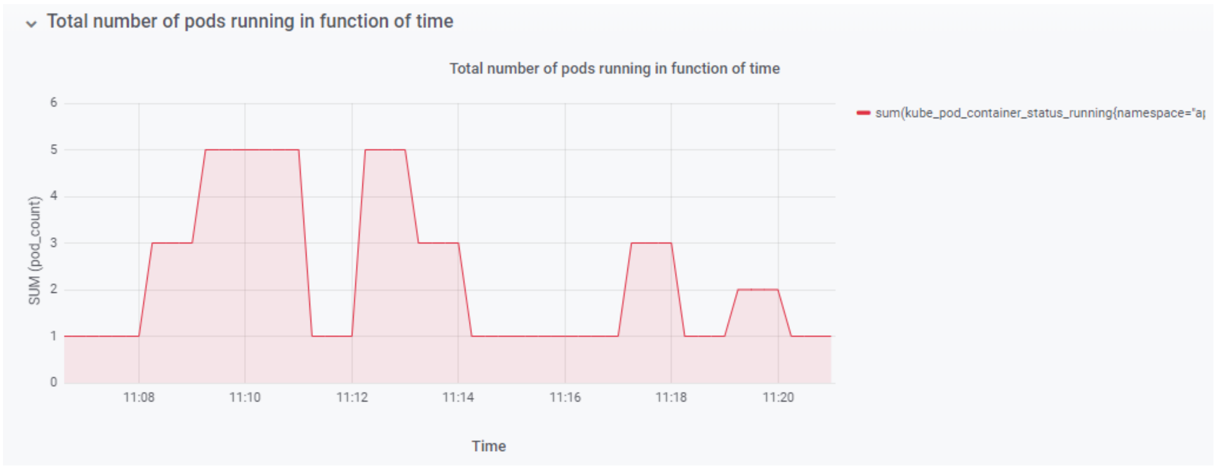


Figure 29: Scaling test with aperiodic load pattern, 1m downscale stabilization – Running pods

In this configuration, we can see that pods are very quickly deleted, then recreated when needed. But, due to the delay explained in previous parts, no load peak except the last one – very close to the penultimate – is handled on enough pods.

Kubernetes spend a huge amount of time on creating and deleting pods, for a not so convincing result; some pods has even been schedule to process any traffic (load peak ended when they were ready).

So, it appears that a longer downscale interval seems to be better to prevent these phenomena of "pods-flapping" ; but I think that it can be dependant of the application. For example, if we have an application able to boot very quickly, we can give a chance to a smaller interval, if we are sure that new pods will be ready in very little time.

Finaly, if we check the average value of the metric (that is used as target for scaling), we can see that the condition is rarely respected (zero-values are not dispolayed):

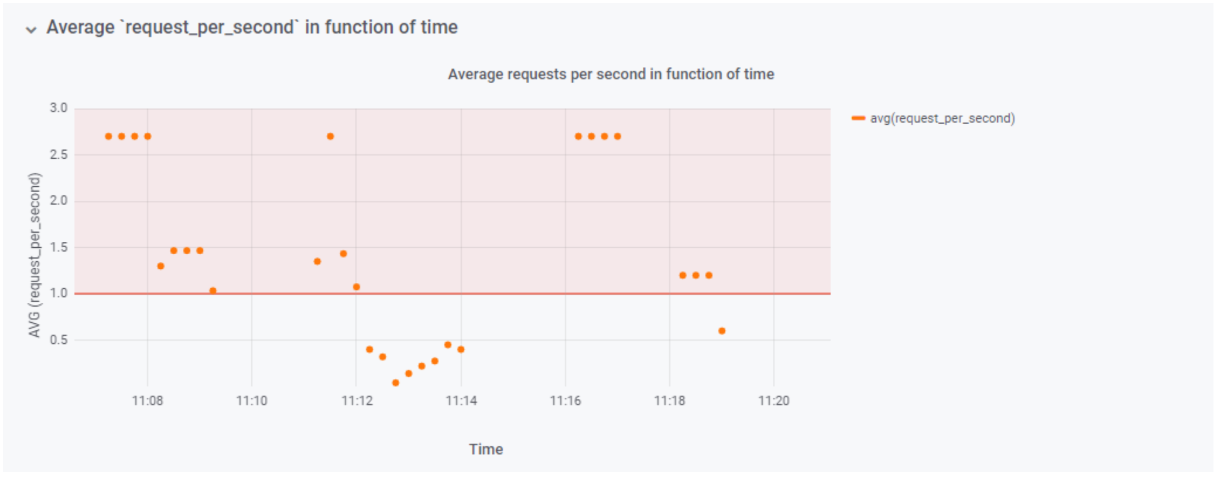


Figure 30: Scaling test with aperiodic load pattern, 1m downscale stabilization – Average HTTP requests handled per pod per second

#### 2.5.3.2. 3 minutes downscale stabilization

Let's try again with an intermediate value: 3 minutes.

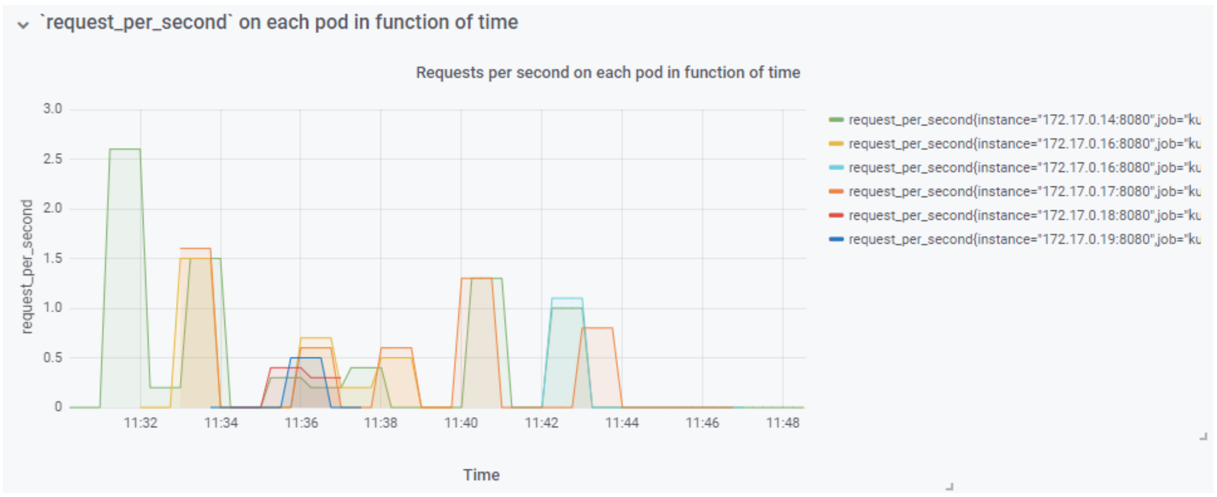


Figure 31: Scaling test with aperiodic load pattern, 3m downscale stabilization – HTTP requets distribution

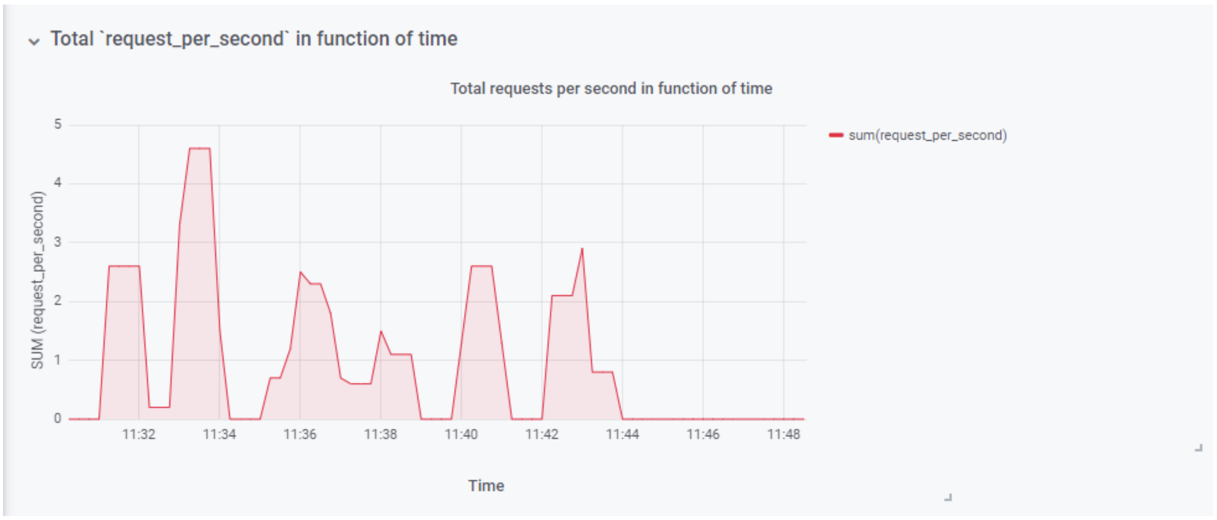


Figure 32: Scaling test with aperiodic load pattern, 3m downscale stabilization – HTTP requets total

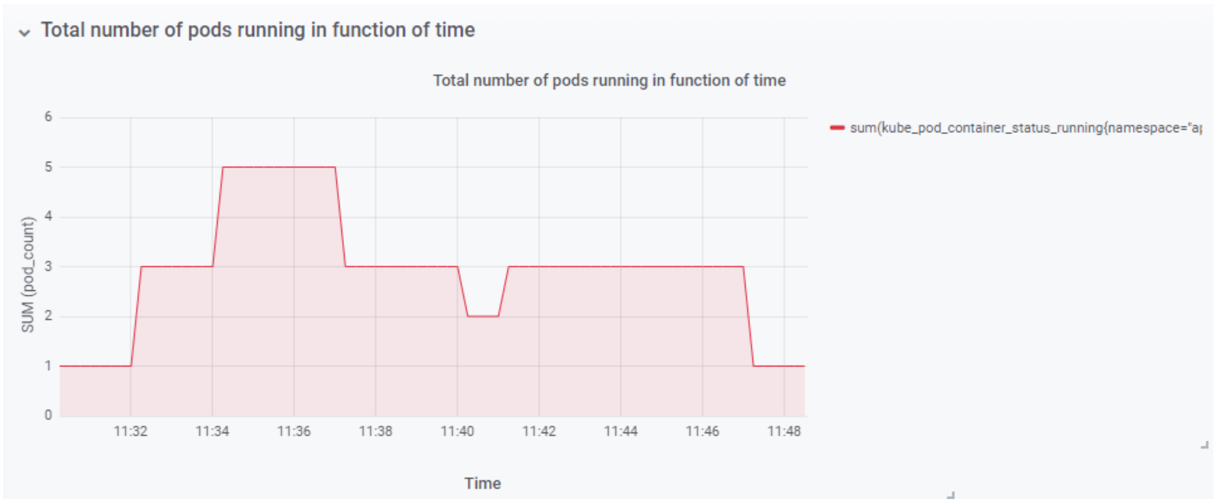


Figure 33: Scaling test with aperiodic load pattern, 3m downscale stabilization – Running pods

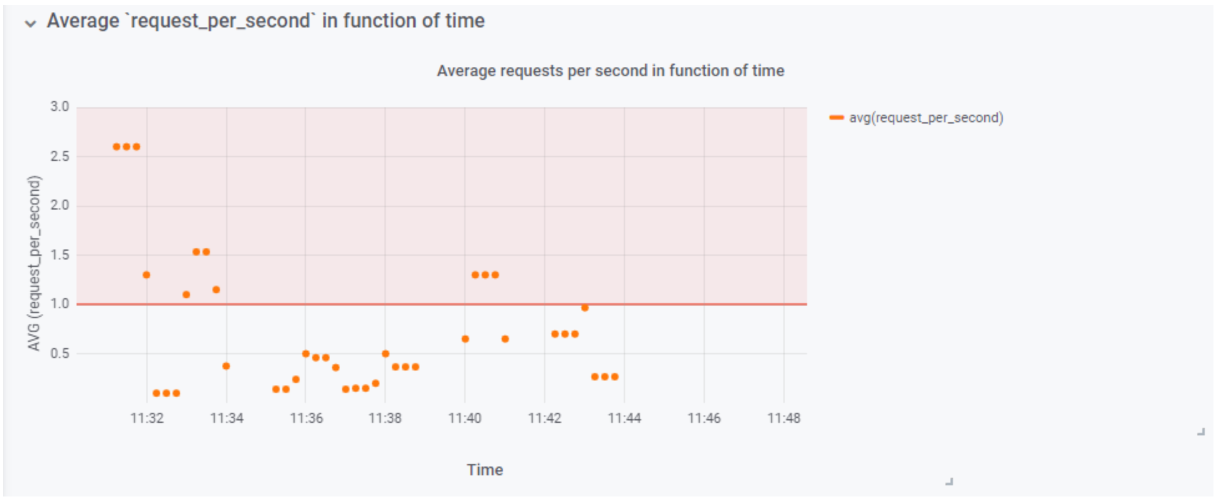


Figure 34: Scaling test with aperiodic load pattern, 3m downscale stabilization – Average HTTP requets handled per pod per second

This configuration is better than the previous one in term of load-spreading and respect with the scaling condition. We can see that there is less points violating this condition (excepting for the first load-peak, but this beahavior is common to each scenario, because there wasn't any traffic before).

#### 2.5.3.3. 7 minutes downscale stabilization

Finnaly, let's give a try to an higher value, 7 minutes:

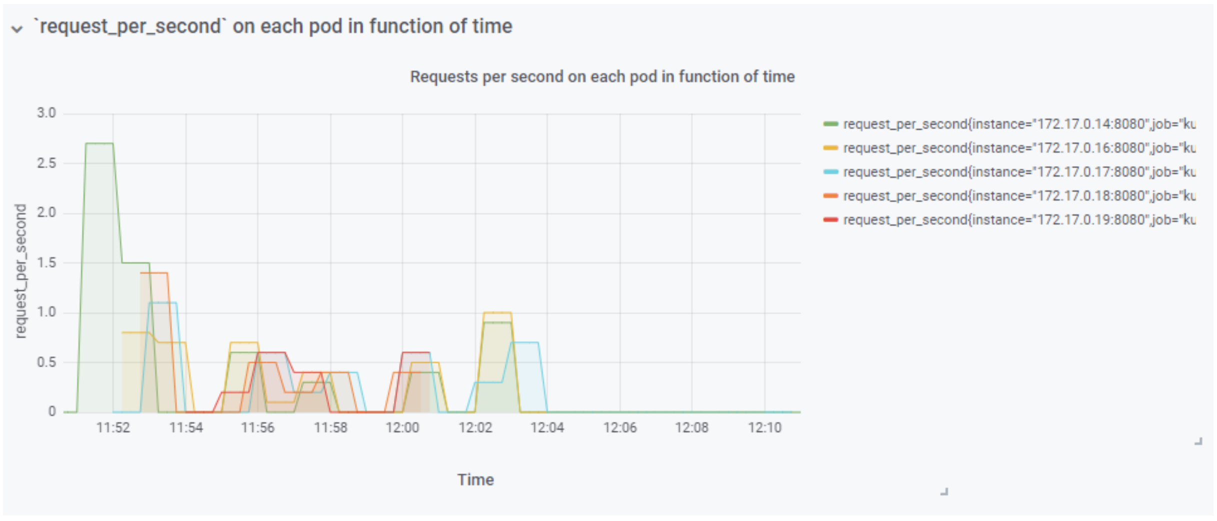


Figure 35: Scaling test with aperiodic load pattern, 7m downscale stabilization – HTTP requets distribution

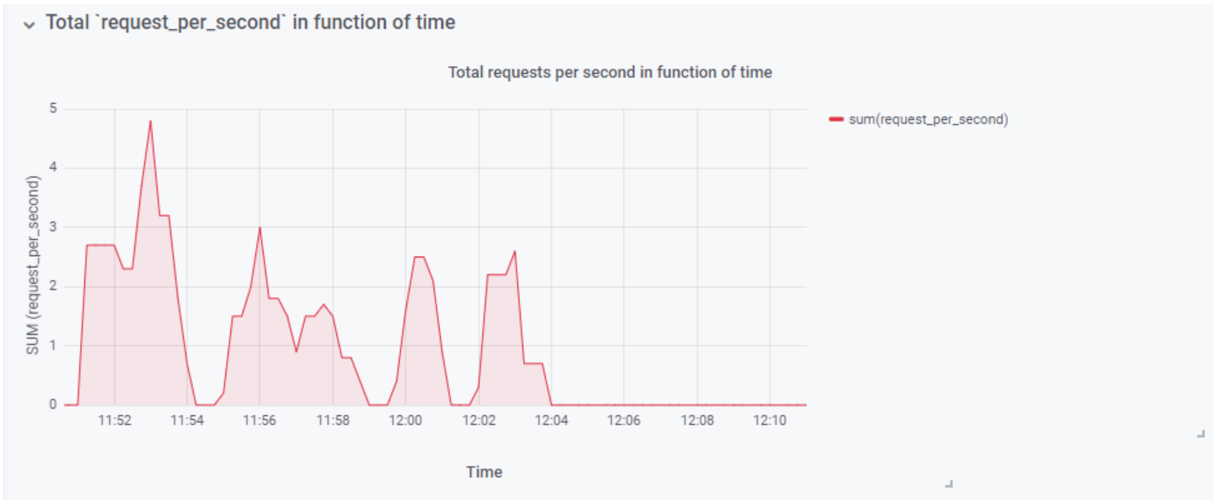


Figure 36: Scaling test with aperiodic load pattern, 7m downscale stabilization – total HTTP request

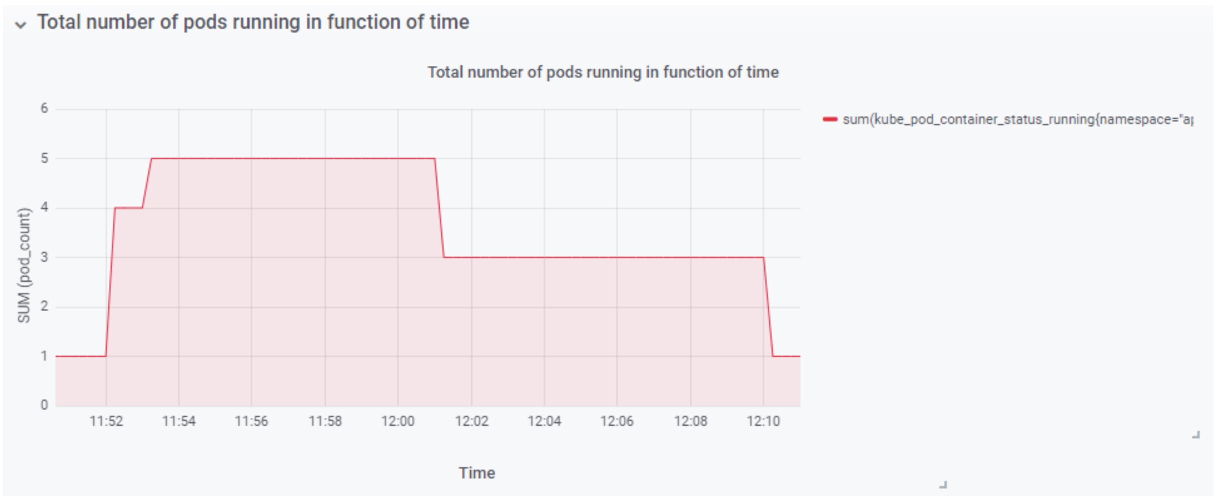


Figure 37: Scaling test with aperiodic load pattern, 7m downscale stabilization – Running pods

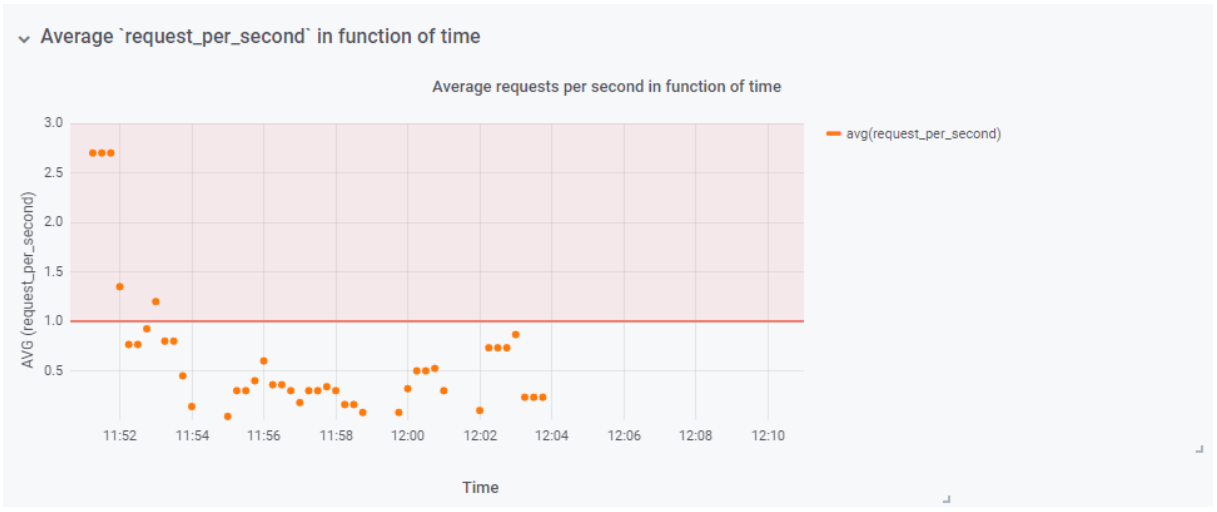


Figure 38: Scaling test with aperiodic load pattern, 7m downscale stabilization – Average HTTP requests handled per pod pod second

In this case, we can see that there isn't any improvement in comparison with the 5 minutes downscale stabilization. So, in this case of this scenario, we don't get any improvement in terms of Quality of Experience for the final user, but we consumed more resources, by keeping up some useless pods.

### 2.5.4 Summary

For each downscale value, let's compulet the average count of pods running, the percentage of pods overusage, and the percentage of pods underusage.

#### 2.5.4.1. Formulas

Let's define some formulas and variable:

- Total pod run_time. Sum of running durations of each pod involved in the scenario's execution [seconds]:

$$run\_time = \sum_{i=1}^{i=N} \mathrm{T}_{run\ i}\,, \qquad N \geq 1\ number\ of\ pods$$

- Average pods running is the run_time divided by the scenario duration + the downscale-stabilization [count]:

$$pod\_avg = \frac{rt}{\mathrm{T}_{scenario} + \tau}$$

- Overusage factor [no UOM]:

$$\alpha = \frac{metric\_value}{metric\_threshold}, \qquad metric\_value > metric\_threshold$$

- Pods overusage: Sum of durations that each pod has been in overusage, multiplied by the previous factor, in part of the total run_time [%]:

$$ovr\_us = \frac{\sum_{i=1}^{i=N}\sum_{j=1}^{j=M} \mathbf{T}_{overusage,\ j_{\ i}} \times \alpha}{run\_time} * 100$$

$$N \geq 1\ number\ of\ pods, \qquad M \geq 1\ number\ of\ overusage\ intervals$$

- Underusage factor [no UOM]:

$$\beta = 1 - \frac{metric\_value}{metric\_threshold}, \qquad metric\_value < metric\_threshold$$

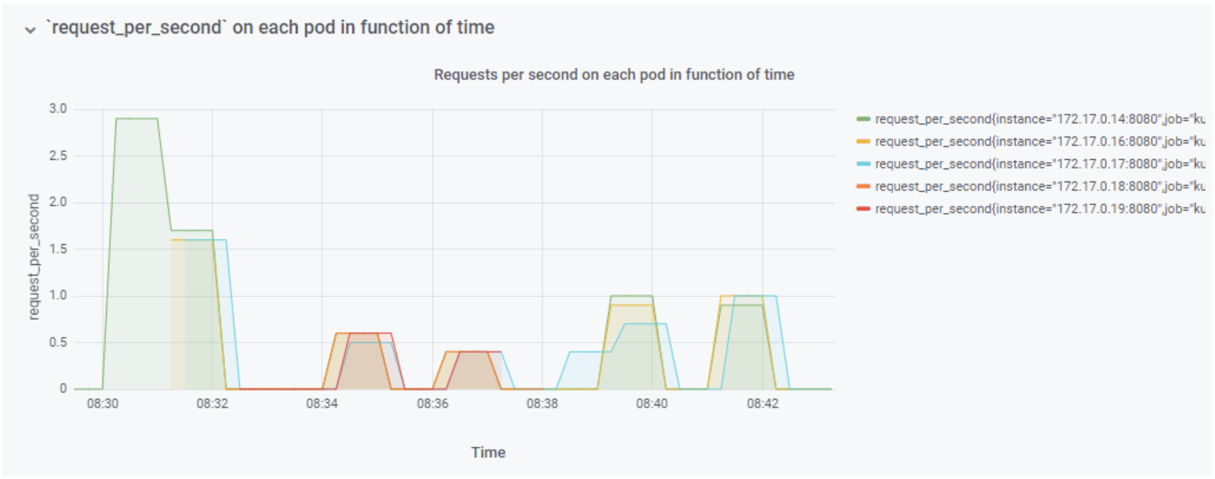
- Pods overusage: Sum of durations that each pod has been in underusage, multiplied by the previous factor, in part of the total run_time. [%]:

$$udr\_us = \frac{\sum_{i=1}^{i=N}\sum_{j=1}^{j=M} \mathbf{T}_{underusage,\ j_{\ i}}}{run\_time} * 100$$

$$N \geq 1\ number\ of\ pods, \qquad M \geq 1\ number\ of\ overusage\ intervals$$

#### 2.5.4.2. Manual calculation

As a reminder, the graph (Figure 23) is:



run_time = (17,5 + 17 + 17 + 6 + 6 ) * 60 = 3810s

pod_avg = 3810 / (17,5 * 60) = 3.63

ovr_us = ( ( 3*60 + 1,7 * 60 + 1,7 * 60 ) / 3810 ) * 100 = 10.08 %

udr_us = ( ( 0,4*50 + 0,6*50 + 0,1*50 + 1*420 + 0,4*50 + 0,6*50 + 0,1*50 + 1*420 + 0,5*60 + 0,6*60 + 0,4*60 + 0,7*60 + 1*420 + 0,4*50 + 0,6*50 +1*120 + 0,4*50 + 0,6*50 + 1*120 ) / 3810 ) * 100 = 48,34%

#### 2.5.4.3. Pseudo-automated calculation

As we can see, calculating manually the over and under usage percentage is time-consuming and can be a source of error.

It is possible to use a simple python3 script that can do the same thing. In the Git repo, there is a custom tool in `dist/ext/` called `ppUsgTool` (Prometheus Pods' Usage Tool).

The code is given in appendix.

##### 2.5.4.3.1. Installation

Insalling the tool is easy as running:

```
python3 -m pip install -r requirements.txt
```

##### 2.5.4.3.2. Using the tool

The tool offers a CLI enable the user to interact with it :

```
python3 .\dist\ext\ppUsgTool
usage: A simple tool to get over and under pod usage from Prometheus
       [-h] -p PROMETHEUS [-k] -q QUERY -v VALUE -s START -e END

optional arguments:
  -h, --help            show this help message and exit
  -p PROMETHEUS, --prometheus PROMETHEUS
                        Prometheus server url
  -k, --notlscheck      Disable cert check verification
  -q QUERY, --query QUERY
                        Prometheus query for the metric
  -v VALUE, --value VALUE
                        Metric target value
  -s START, --start START
                        Scenario start
  -e END, --end END     Scenario end
```

It is not required to give start and end date with a great precision; if there is zeroes values, theses will be automatically removed, so we can take some margin before and after to be sure the scenario has been running in the interval. For example, the script can be invoked like:

```
$ py -3 ./dist/ext/ppUsgTool  -p 'https://prometheus.k8s.home.lab' -k -v 1.0 -q
'request_per_second' -s '2020-01-20T15:30:00.000Z' -e '2020-01-20T15:43:00.000Z'
Average pods running: 3.63
Overusage: 10,08%
Underusage: 48,34%
```

#### 2.5.4.4. Synthesis table

| Downscale value | Average pods running | Pods overusage (%) | Pods underusage (%) |
|---|---|---|---|
| 1 minute | 2,69 | 39,00% | 31,05% |
| 3 minutes | 3,33 | 16,37% | 41,01% |
| 5 minutes | 3,63 | 10,08% | 48,34% |
| 7 minutes | 3,86 | 8,15% | 54,98% |

#### 2.5.4.5. Service-first or cost/resources-optimzed approach

We can see that, the higher the downscale-stabilization is, the more pods we have running. So, we have more resources up and ready to serve the requests received. But, this will result on more resources consumtions; which will affect directly the cost for running the application.

If we chose a small downscale-stabilization, we will optimize resources usage and billing, but we will degrade the service, because not enough pods will be ready at the begging of new load peaks.

# 3. Conclusion

This work highlights how Kubernetes helps IT teams to make their cloud applications running at scale, in a dynamic way. By performing scale-up dans scale-down operations based on any application-exposed metric, it enables people to configure exactly how they want the pods horizontal scalability.

By offering an open API, Kubernetes is extensible: the integration of Prometheus via a third-party adapter is an example of this. This is very great because Prometheus is probabaly the industry standard application performance monitoring solution (based on metrics), so every one who is already using Prometheus to monitor applications don't has to redesign his monitoring stack for enabling the custom-metrics-based Horizontal Pod Autoscaler.

# 4. Appendix

## 4.1 Application code

### 4.1.1 Readiness controller

```
package tech.glar.hellok8s.health

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

@RestController
class HealthController {

    @GetMapping("/health")
    fun health(): HealthData {
        return HealthData("up")
    }
}
```

```
package tech.glar.hellok8s.health

data class HealthData (
    val status: String
)
```

### 4.1.2 Application instrumentation

Before exposing metrics inside the application, we need to set up some instrumentation. Let's start by setting up an embedded database to store logs :

```
// application.properties: defined the application configuration

spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:appdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=passwd
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

```
// data.sql: this file will init the in-memory database at application startup

DROP TABLE IF EXISTS LOG_DATA;

CREATE TABLE LOG_DATA (
     id LONG AUTO_INCREMENT PRIMARY KEY,
     timestamp TIMESTAMP,
     endpoint TEXT,
     code INT,
     resp_time DOUBLE
)
```

```
// LogData.kt: A model of the table used by the Object Relationship Manager
(ORM) to generate requests

package tech.glar.hellok8s.logs

import org.hibernate.annotations.CreationTimestamp
import java.util.*
import javax.persistence.*


@Entity
data class LogData (

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private val id: Long = 0,

    @Column(nullable = false, updatable = false)
    @CreationTimestamp
    private val timestamp: Date = Date(),

    @Column(nullable = false)
    private val endpoint: String,

    @Column(nullable = false)
    private val code: Int,

    @Column(nullable = false)
    private val respTime: Long

)
```

```
// LogRepository.kt: Extends ths CrudRepository class with new methods. The
method's name is used to automatically generate the SQL request.

package tech.glar.hellok8s.logs

import org.springframework.data.repository.CrudRepository
import org.springframework.stereotype.Repository
import java.util.Date

@Repository
interface LogRepository : CrudRepository<LogData, Long> {

    fun countLogDataByTimestampAfter(timestamp: Date): Long

}
```

```kotlin
// LogService.kt: A Sping Boot Service Interface

package tech.glar.hellok8s.logs

import java.util.Date

interface LogService {

    fun saveOrUpdate(log: LogData)
    fun getRequestCount(timeStamp: Date): Long

}
```

```kotlin
// LogServiceImpl.kt: The implementation of LogService

package tech.glar.hellok8s.logs

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service
import java.util.Data

@Service
class LogServiceImpl : LogService{
    @Autowired
    lateinit var logRepository: LogRepository

    override fun saveOrUpdate(log: LogData){
        logRepository.save(log)
    }
    override fun getRequestCount(timeStamp: Date): Long{
        return logRepository.countLogDataByTimestampAfter(timeStamp)
    }
}
```

We can now use the LogService to save logs in any Controller :

```kotlin
package tech.glar.hellok8s.info

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController
import tech.glar.hellok8s.logs.LogData
import tech.glar.hellok8s.logs.LogServiceImpl
import java.net.InetAddress
import java.util.Random

```

```
@RestController
class InfoController {

    @Autowired
    lateinit var logService: LogService

    val address: InetAddress = InetAddress.getLocalHost()
    val random =  Random()

    @GetMapping("/info")
    fun info(): InfoData {

        val latency = (random.nextFloat() * 10).toLong()
        Thread.sleep(latency)

        val log = LogData(endpoint = "/info",
                          code = 200, respTime = latency)
        logService.saveOrUpdate(log)

        return InfoData(address.hostName, latency)

    }

}
```

```
package tech.glar.hellok8s.info

data class InfoData(
    val hostname: String,
    val latency: Long
)
```

### 4.1.3 Metrics page

We can now create a REST Controller for /metrics:

```
package tech.glar.hellok8s.metrics

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.beans.factory.annotation.Value
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController
import tech.glar.hellok8s.logs.LogService
import java.util.Calendar


@RestController
class MetricsController {
```

```kotlin
    @Autowired
    lateinit var logService: LogService

    // default value is set, but declared value in
    // application.properties (if any) is injected
    @Value("\${metrics.averagingperiod}")
    val averagingPeriod: Double = 60.0

    @GetMapping("/metrics", produces = ["text/plain"])
    fun metrics(): String {

        val calendar = Calendar.getInstance()
        calendar.add(Calendar.SECOND, - averagingPeriod.toInt() )
        val startTime = calendar.time

        val rate = logService.getRequestCount(startTime)
                             .div(averagingPeriod)
        return """#
            |request_per_second $rate
            |averaging_period $averagingPeriod
            |#
        """.trimMargin()
    }
}
```

The averaging period is set in a config file, called application.properties:

```
metrics.averagingperiod=5.0
```

## 4.2 Prometheus Pods Usage Tool code

### 4.2.1 CLI

```
import argparse
import sys

def parse_args():
    parser = argparse.ArgumentParser("A simple tool to get over and under
                                                  pod usage from Prometheus")
    parser.add_argument("-p", "--prometheus",
                              help="Prometheus server url", required=True, type=str)
    parser.add_argument("-k", "--notlscheck",
                       help="Disable cert check verification", action='store_true')
    parser.add_argument("-q", "--query",
                   help="Prometheus query for the metric", required=True, type=str)
    parser.add_argument("-v", "--value",
                              help="Metric target value", required=True, type=float)
    parser.add_argument("-s", "--start",
                                     help="Scenario start", required=True, type=str)
    parser.add_argument("-e", "--end",
                                       help="Scenario end", required=True, type=str)

    if len(sys.argv) <= 1:
        parser.print_help()
        sys.exit(-1)

    return parser.parse_args()
```

### 4.2.2 Main

```
from lib.args import parse_args
import atexit
import sys
import requests

def main():
    args = parse_args()

    s = requests.session()
    s.verify = False if args.notlscheck else True
    atexit.register(lambda : s.close())
    r = s.get(f'{args.prometheus}/api/v1/query_range?query={args.query}
                            &start={args.start}Z&end={args.end}&step=1s')
    r.raise_for_status()

    over = 0
    under = 0
    total_nonnull = 0
    run_time = 0
    min_tstp = sys.maxsize
    max_tstp = - sys.maxsize

    for pod in r.json()['data']['result']:
        for v in pod['values']:

            # Time bound detection
            if v[0] < min_tstp:
                min_tstp = v[0]
            if v[0] > max_tstp:
                max_tstp = v[0]


            m = float(v[1]) # Value
            if m > args.value:
                over += (m - args.value)

            elif (m < args.value) and (m != 0) :
                under += (args.value - m)

            if m != 0:
                total_nonnull += 1

            run_time += 1

    user_time_duration = max_tstp - min_tstp

    print("Average pods running: %.3f"%(run_time /  user_time_duration))
    print("Overusage: %d%%"%(over*100/total_nonnull))
    print("Underusage: %d%%"%(under*100/total_nonnull))

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

# 5. Bibliography

1. “Horizontal Pod Autoscaler.” [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

2. “Helm | Docs.” [Online]. Available: https://helm.sh/docs/

3. “Data model | Prometheus.” [Online]. Available: https://prometheus.io/docs/concepts/data_model/

4. “charts/stable/prometheus at master · helm/charts · GitHub.” [Online]. Available: https://github.com/helm/charts/tree/master/stable/prometheus

5. “Configuration | Prometheus.” [Online]. Available: https://prometheus.io/docs/prometheus/latest/configuration/configuration/

6. “Kubernetes & Prometheus Scraping Configuration.” [Online]. Available: https://www.weave.works/docs/cloud/latest/tasks/monitor/configuration-k8s/

7. “charts/stable/prometheus-adapter at master · helm/charts · GitHub.” [Online]. Available: https://github.com/helm/charts/tree/master/stable/prometheus-adapter

8. “Grafana | Prometheus.” [Online]. Available: https://prometheus.io/docs/visualization/grafana/

9. “charts/stable/grafana at master · helm/charts · GitHub.” [Online]. Available: https://github.com/helm/charts/tree/master/stable/grafana

10. “Horizontal Pod Autoscaler Walkthrough.” [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/