Cloud Application Autoscaling

With Kubernetes and Custom Metrics

Gaël LARGER – 2020

Politecnico di Bari

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

kubernetes

$ whoami

- Gaël LARGER, 22

- Telecommunications, Services and Uses engineering @ INSA Lyon, France
- Erasmus+ @ Politecnico di Bari, Italy

- Areas of interest:
  - IT automation
  - Infrastructure as Code
  - Cloud applications and architecture
  - Software Defined *
  - IoT cloud platforms

Outline

Introduction

1. Architecture overview
2. Testing & Modelization of HPA operation
3. HPA parameters optimization

Conclusion

Introduction

From business needs to technical challenges

- Business requirements:
  - No service/application interruption
  - Transforming ideas to commercial products with shorter delays
  - Constant Quality of Experience at any time
  - Paying for the resources they really consumes (“pay-as-you-use”)
- Technical consequences:
  - We need to autoscale our applications
  - We need to deploy faster
  - Maintenances, IT incidents and load peaks should be transparent for the users (not perceptible)

Trends in IT

- Reliability, scalability, agility, continuous-delivery
    - *Solutions?* Multi/Hybrid clouds, containers, microservices, CI/CD
    - *Challenges?* Management of “containers armies” in the cloud or on premises

- Kubernetes: Industry-Standard container orchestration system

- Google “Site Reliability Engineering” approach: Threating operations as if it’s a software problem

K8s promises

- To be an engine for resolving state by converging actual and the desired state of the system. “You describe. It does.”
  - Deploying and maintaining the application in desired state
  - Monitoring the containers’ health, and restart them if needed
  - Monitoring the nodes’ health, and reschedule workloads running of top of them in case of failure
  - Simplifying 0-downtime software upgrade (Rolling Update strategy) and rollbacks…
- To offer the same API across bare metal and every cloud provider

Monitoring: a paradigm shift

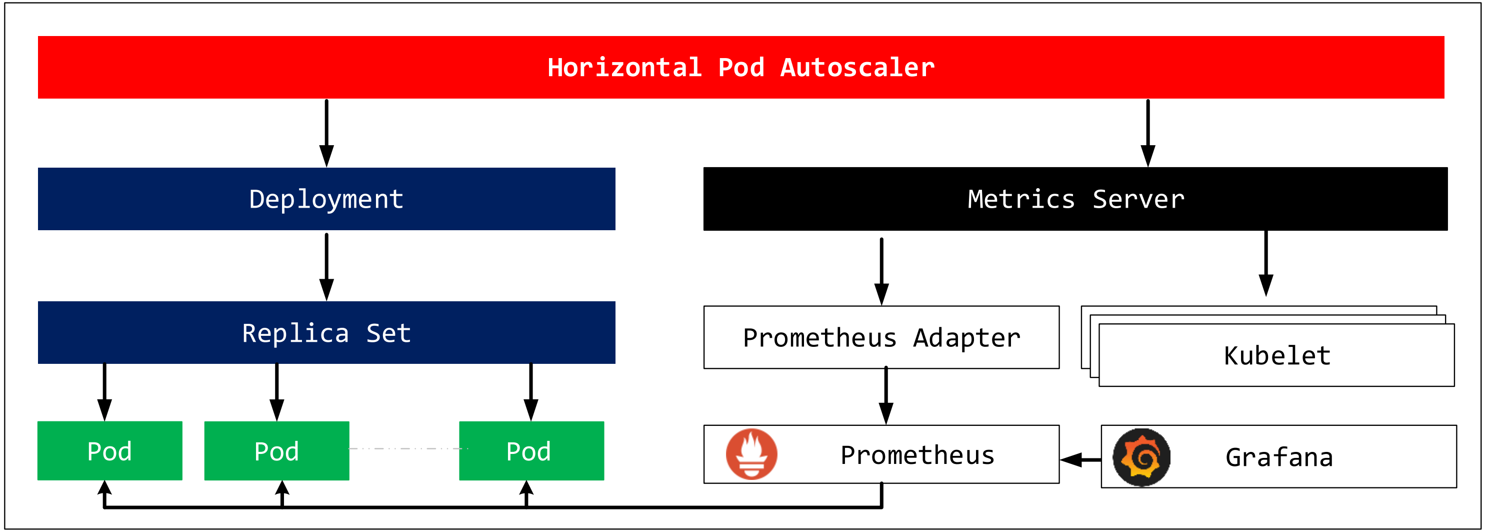
- … from infrastructure monitoring
    - We define control points to be checked on the infrastructure
    - Nagios, Zabbix…

- … to service monitoring
    - Focusing on applications and services delivered to users
    - A lot of open-source tools, usually metrics and/or logs based with time-series storage
    - Prometheus, Telegraf/InfluxDB (InfluxData stack), Wrap10, Grafana…
    - Applications changes: developers have to instrument their code

- Complementary approaches!

Architecture overview

The big picture (core)



Core components (1)

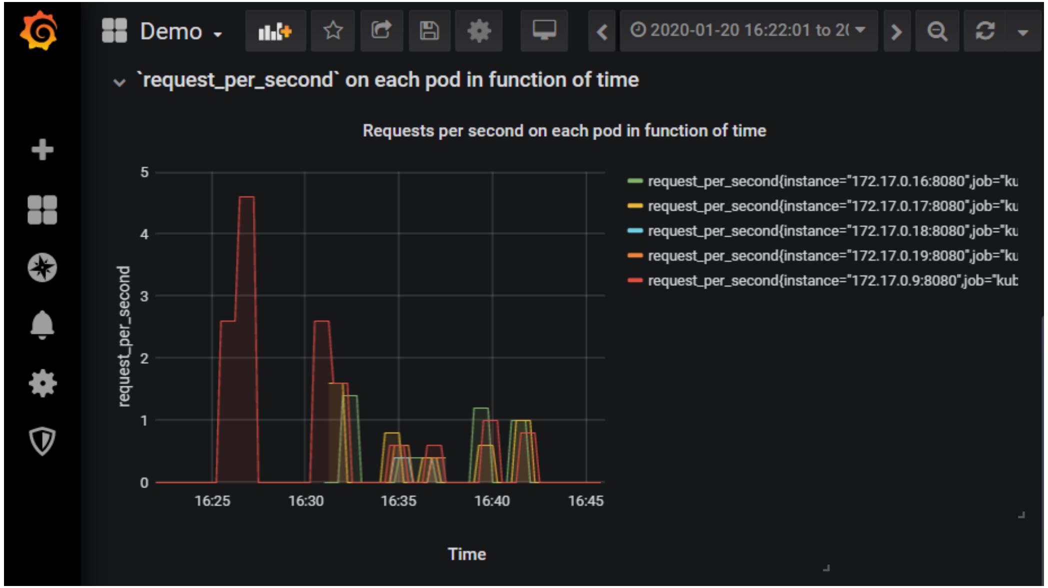
- Pod
  - The smallest entity managed by Kubernetes
  - One or more containers, e.g. a JRE running a Spring Boot Application
- Replica Set
  - Maintains a stable set of pods
  - Guarantee the availability of a given number of identical pods
  - Not directly defined by the k8s admin
- Deployment
  - High-level object
  - Desired state for an application

Core components (2)

- Metrics server
  - Aggregates the metrics of the Kubernetes cluster
  - Build-in: CPU and RAM usage for both pods and nodes
  - Gather these metrics from the Kubelet agent
- Prometheus server
  - Application monitoring engine
  - 4 components: Metrics scrapper, TSDB, WebUI, REST API
- Prometheus adapter
  - Interface making the metrics stored in Prometheus available for the Metrics Server
  - Acts as an interface
  - Registered as “customs.metrics.k8s.io” provider on the Metrics server

Core components (3)

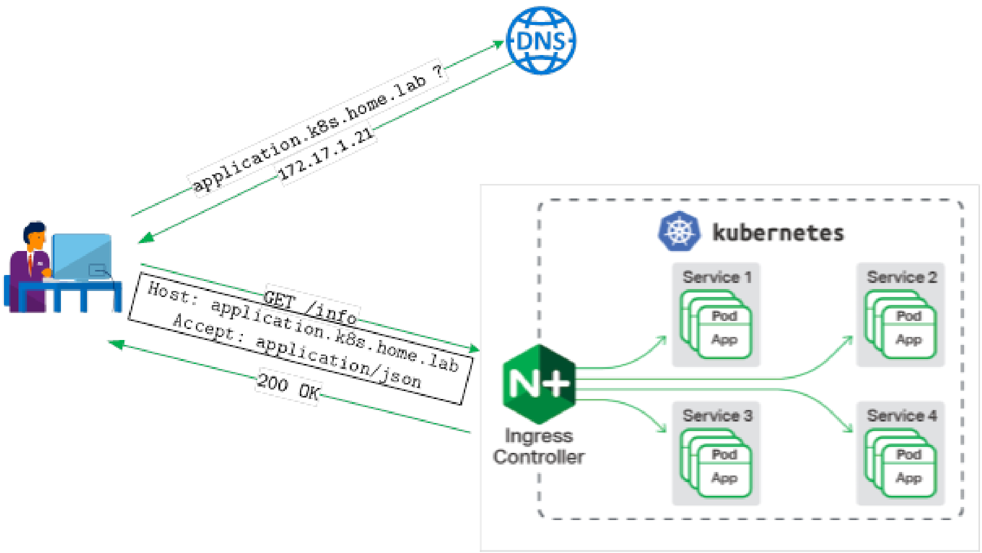
- Grafana
  - Nice WebUI for plotting time-series metrics
  - Can also perform alerting (requires an additional service, called “Alert Manager”)



Networking components

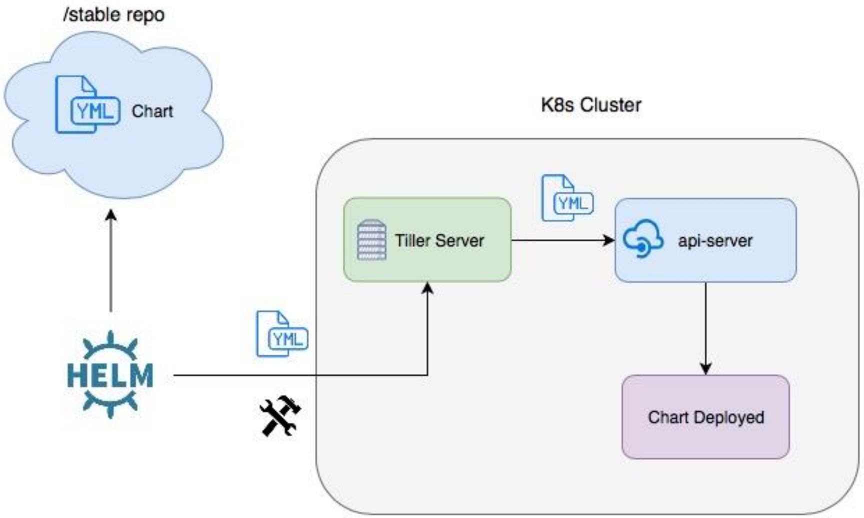
- Nginx Ingress controller
  - Acts as a reverse proxy
  - Runs at the edge between the cluster “internal network” and the “external network”

- DNS requirements
  - Can integrate with DNS providers (such Route53)
  - For development, using a wildcard “Zone” can be a good idea, e.g. `*.k8s.home.lab`



Deployment component: Helm

- Package manager for Kubernetes
  - Based on YAML templating
  - Dependencies management
- Concepts:
  - Chart: Description of a package
  - Repository: Provides charts in a structured and standard manner
  - Release: An instance deployed on k8s
- Client-Server model



Testing components (1)

- JMeter
  - A load testing tool running in the JVM
  - GUI mode: Helps designing testing scenarios (.jmx files) through a graphical interface; can also run the scenarios
  - Headless mode: Can run the jmx scenarios, possible integration with a CI/CD if needed

  - Important note: It can be some differences between the flow you designed in JMeter and what is effectively received by the tested application (latency, buffering…)

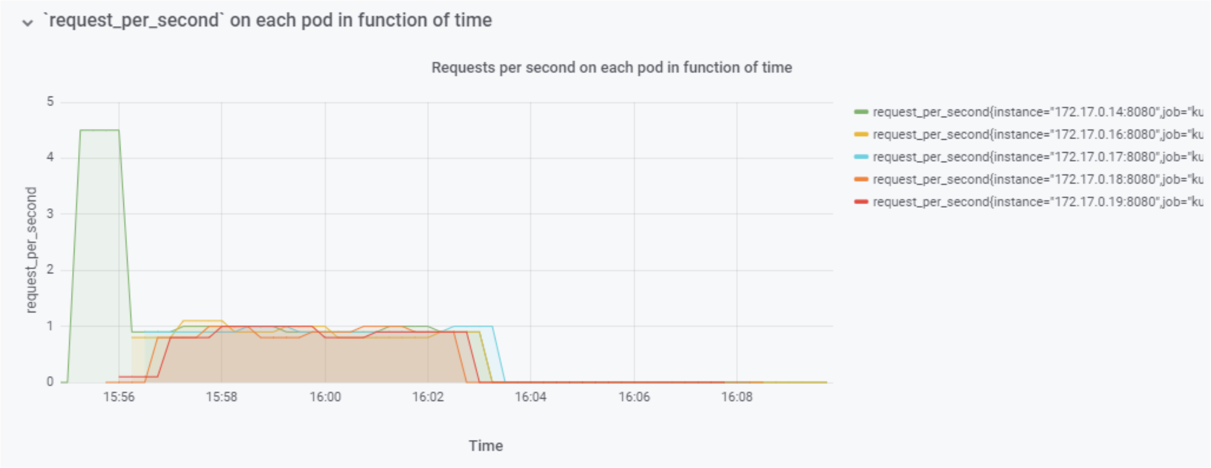
Testing components (2)

- Ultimate Thread Group
    - A plugin for JMeter
    - The most advanced load pattern designing tool for JMeter
    - Key features:
        - Infinite number of schedule records
        - Separate ramp-up time, shutdown time, flight time for each schedule record
        - Load preview graph

Testing & Modelization

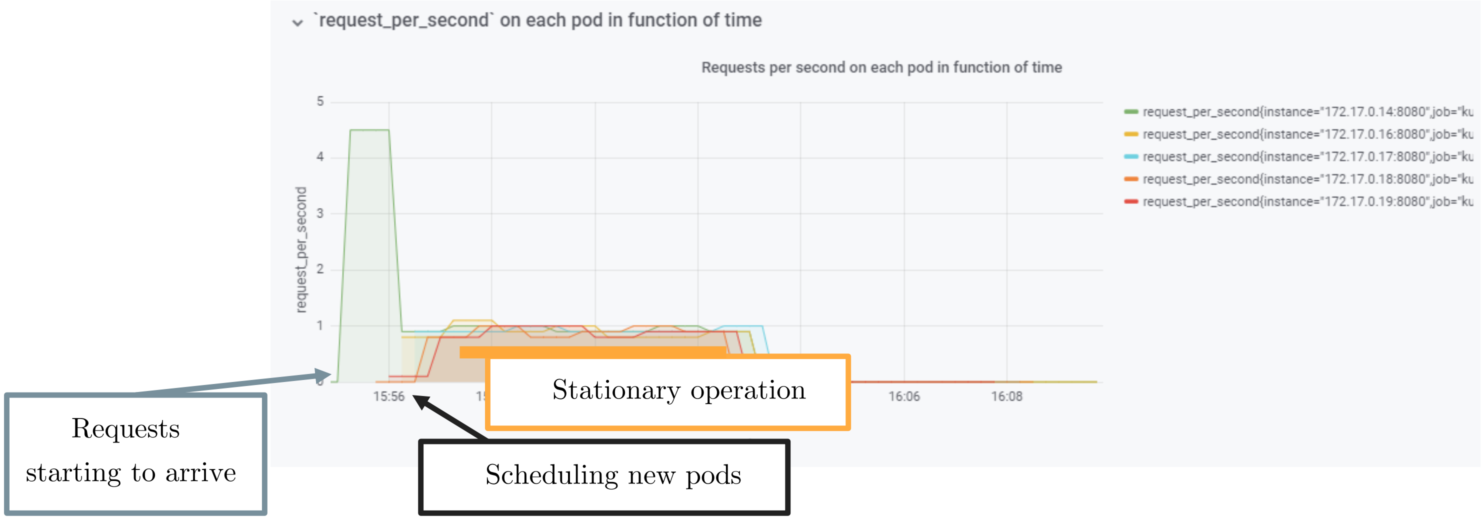
Pattern 0 : Constant load

- We will use JMeter to send HTTPs requests to the pod, and obeserve what happen
    - 1 thread group
    - 5 threads
    - HTTP Requests on https://application.k8s.home.lab/info
    - 1100ms Constant timer



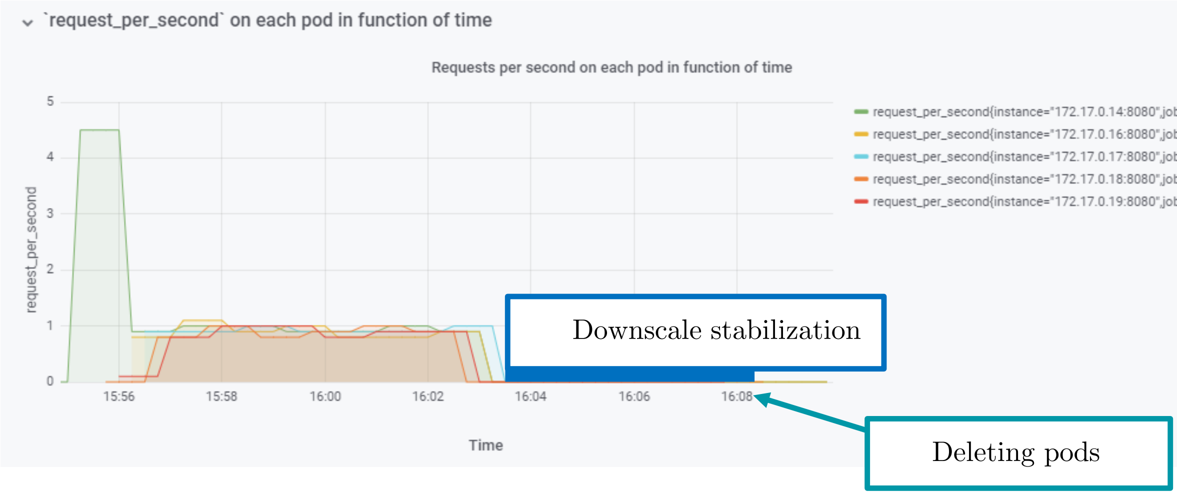
Pattern 0 : Constant load

- We will use JMeter to send HTTPs requests to the pod, and obeserve what happen
  - 1 thread group
  - 5 threads
  - HTTP Requests on https://application.k8s.home.lab/info
  - 1100ms Constant timer



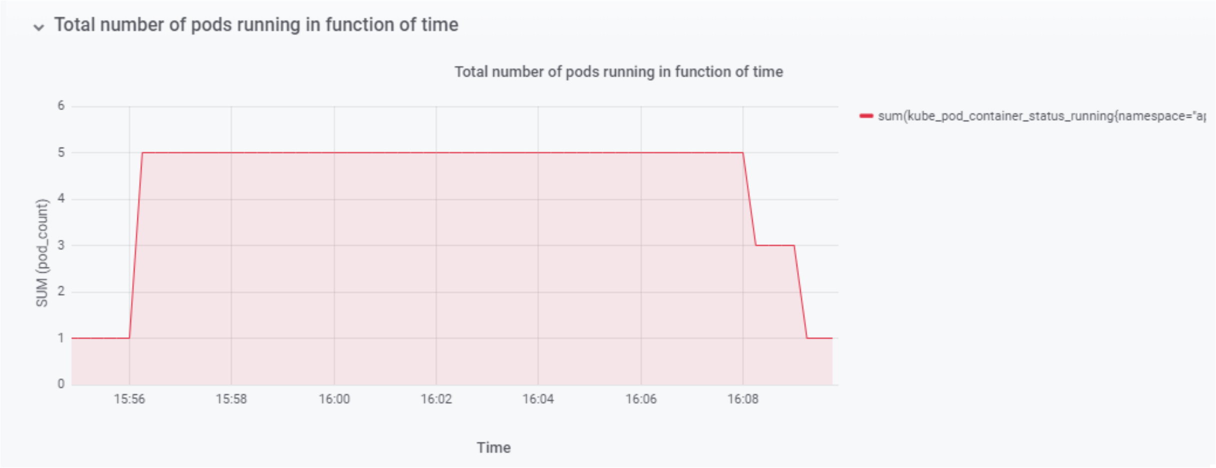
Pattern 0 : Constant load

- We will use JMeter to send HTTPs requests to the pod, and obeserve what happen
  - 1 thread group
  - 5 threads
  - HTTP Requests on https://application.k8s.home.lab/info
  - 1100ms Constant timer



Pattern 0 : Constant load

- We will use JMeter to send HTTPs requests to the pod, and obeserve what happen
    - 1 thread group
    - 5 threads
    - HTTP Requests on https://application.k8s.home.lab/info
    - 1100ms Constant timer



Pattern 0 : Constant load

- Let’s check the logs :

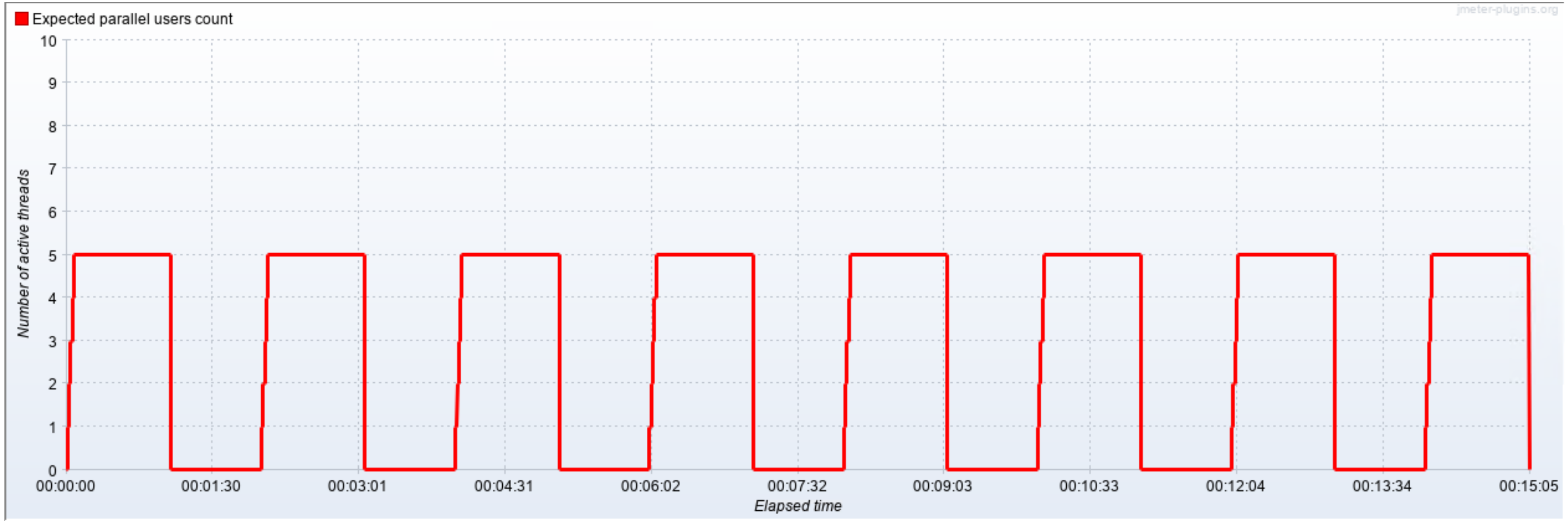
```
glar@hlbk8s01lv:~$ k describe hpa/glar-demo-application-hpa -n application
Name:                           glar-demo-application-hpa
Namespace:                      application
Labels:                         <none>
Annotations:                    <none>
CreationTimestamp:              Tue, 10 Dec 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
```

Upscale

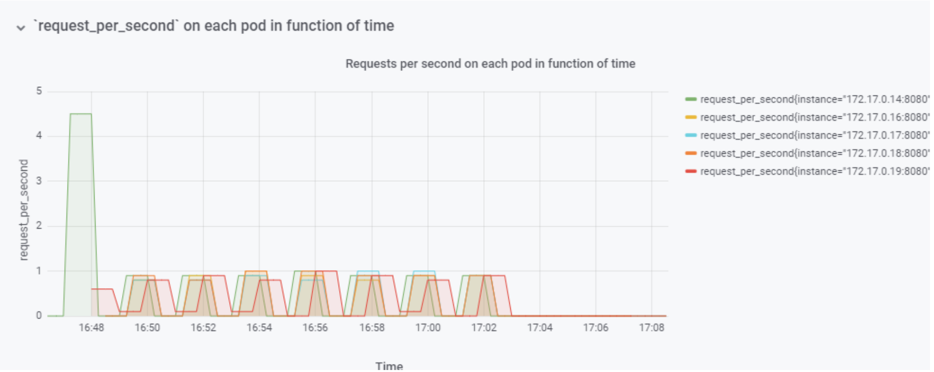
Downscale

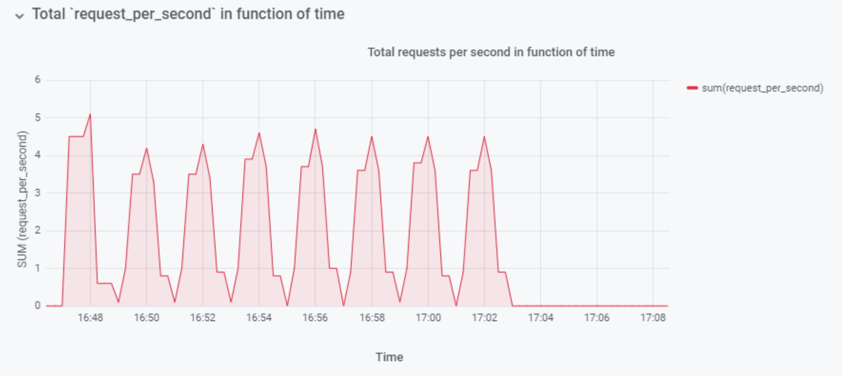
Pattern 1: On/Off, 50% duty cycle

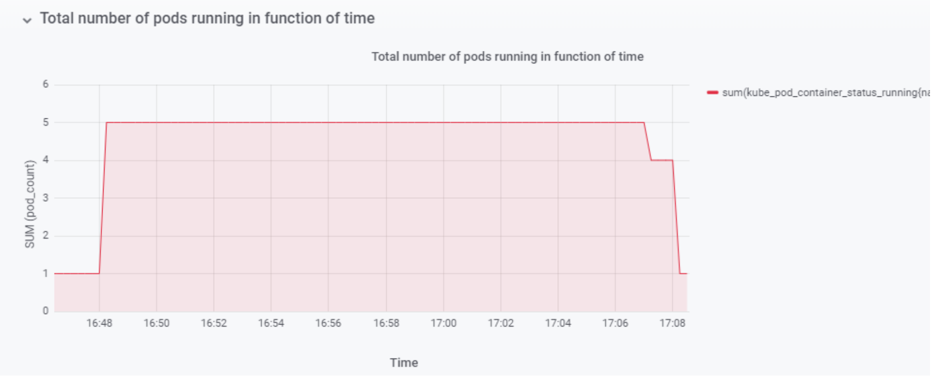
- Scenario:



Pattern 1: On/Off, 50% duty cycle



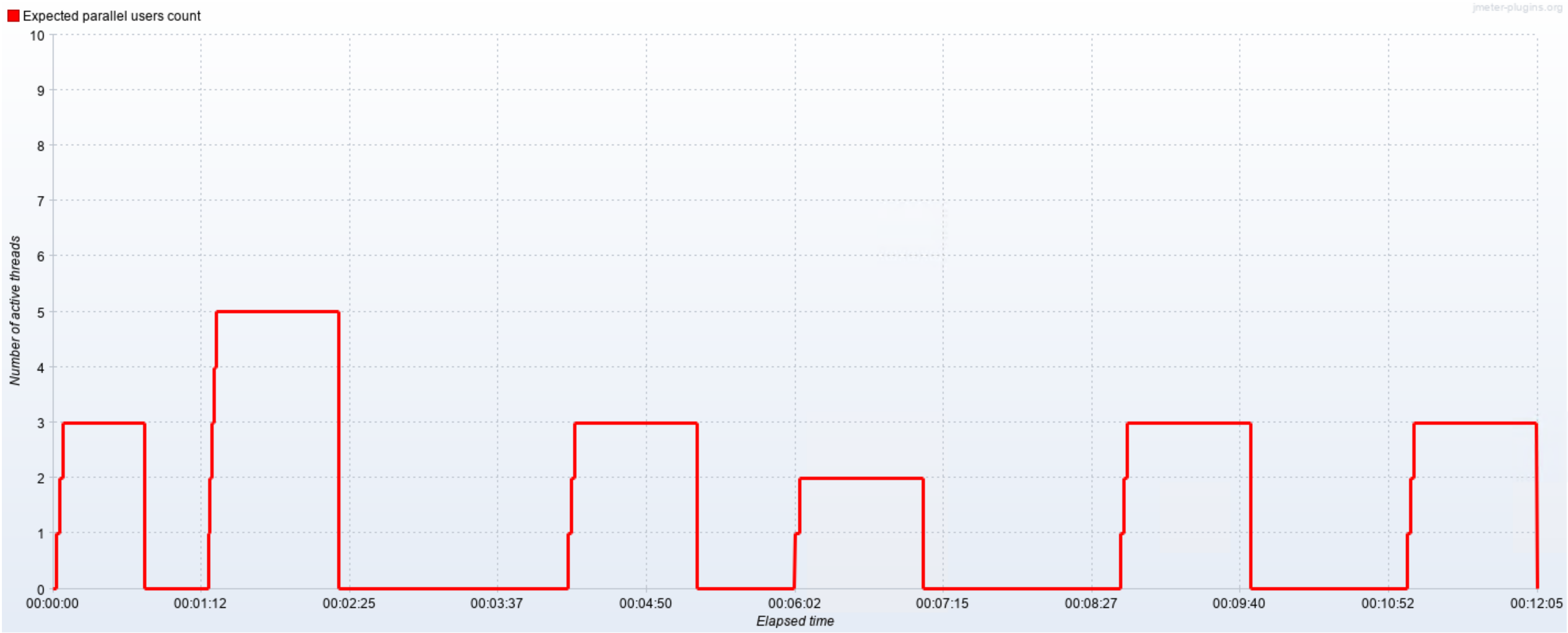




Pods stays scheduled from a load peak to another!

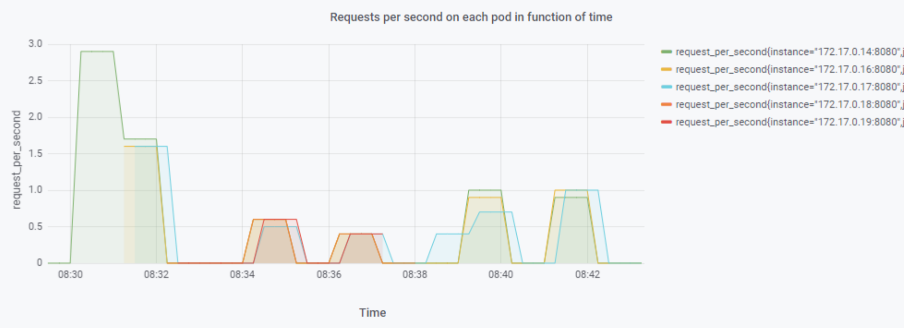
Pattern 2: Aperiodic On/Off

- Scenario:

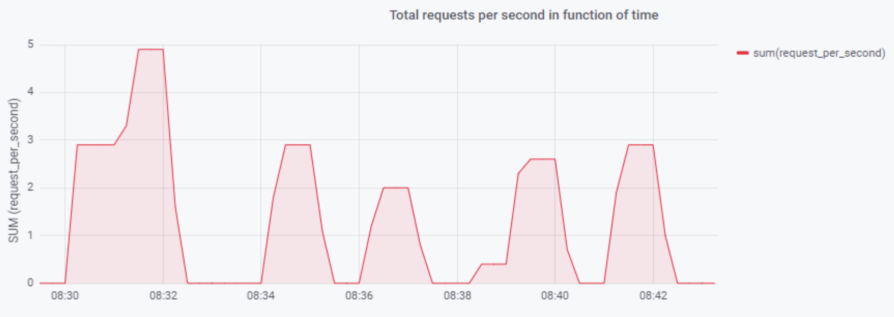


Pattern 2: Aperiodic On/Off

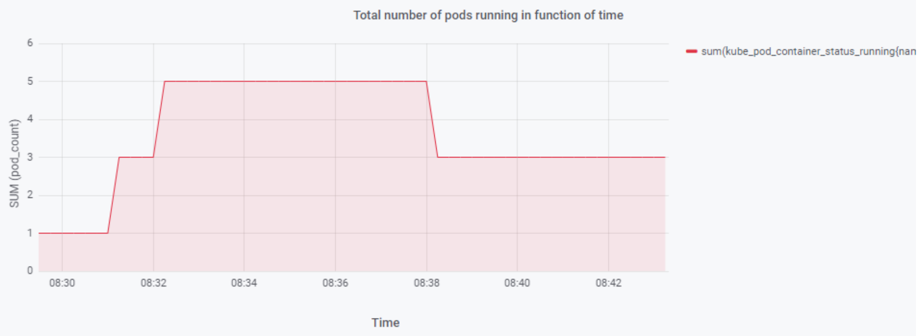
⌄ `request_per_second` on each pod in function of time



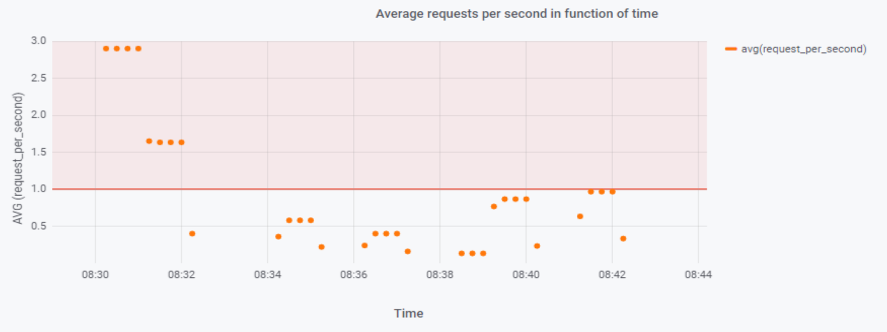
⌄ Total `request_per_second` in function of time



⌄ Total number of pods running in function of time



⌄ Average `request_per_second` in function of time



HPA operation modelization

- Let's define:
  - τ : Downscale-stabilization, time waited by k8s for deleting pods, after that the metric values indicates that the load received can be process by less pods
  - Δ_{peak} : Time between two load peaks, in a periodic case $\Delta_{peak} = DutyCycle * T$

- To prevent "pod flapping", we need to have: $\boldsymbol{\tau} > \boldsymbol{\Delta_{peak}}$

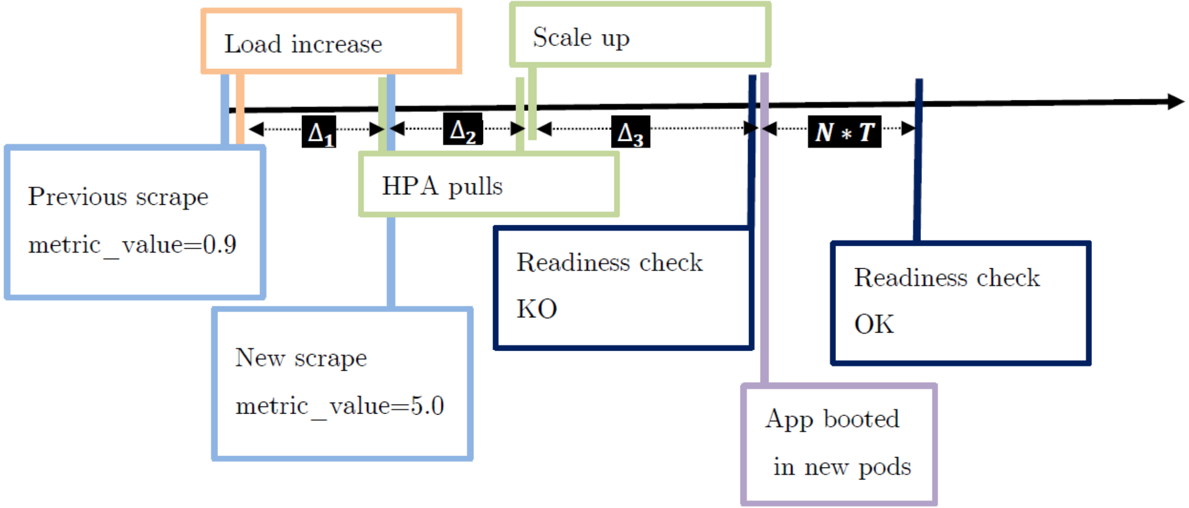
- To optimize resources consumption, we need to have: Δ_{peak} as close as possible to $\boldsymbol{\tau}$

HPA latency modelization (1)

- More of that, each component pulls the metrics from another one, and the application has a delay for booting and being ready:
    - $T_{promScrape}$: Prometheus Scrape interval
    - $T_{hpaPull}$: HPA query interval
    - $\Delta_{appBoot}$: Application boot time
    - $\Delta_{appReadiness}$: Initial readiness check delay (after scheduling)
    - $T_{appReadinessRecheck}$: Readiness recheck interval

HPA latency modelization (2)

- In the wort case we have a total delay equal to:
  - $\Delta_{totalDelay} = T_{promScrape} + T_{hpaPull} + \Delta_{appBoot} + N * T_{appReadinessRecheck}$
  - $\Delta_{totalDelay} = \Delta_1 + \Delta_2 + \Delta_3 + N * T$



Optimization

Goals

- Illustrates how the choice of a downscale-stabilization value (default is 5 minutes) influence the behavior of the HPA

- Real world constraints:
    - The traffic received by an application don’t match a perfect model
    - It’s complex to find the best value
    - Tweaking downscale stabilization value is needed to improve the behavior of the HPA in function of real traffic (as observed)

Configuration

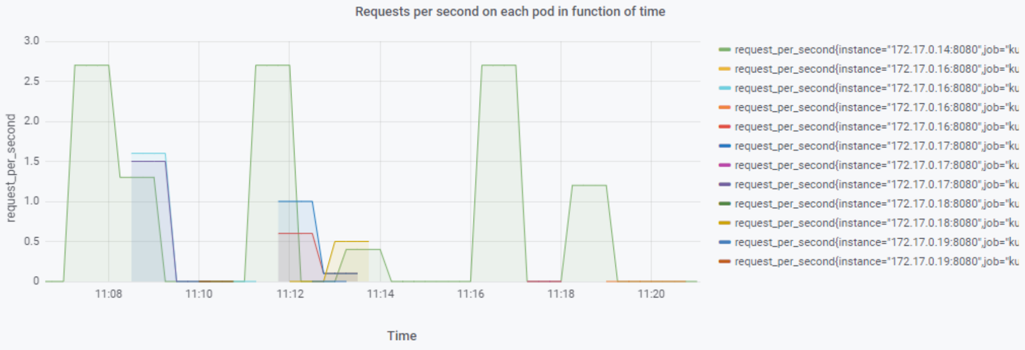
- Cluster-wide parameter (not namepaced!)
- Config file: `/etc/kubernetes/manifests/kube-controller-manager.yaml`

```
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
```

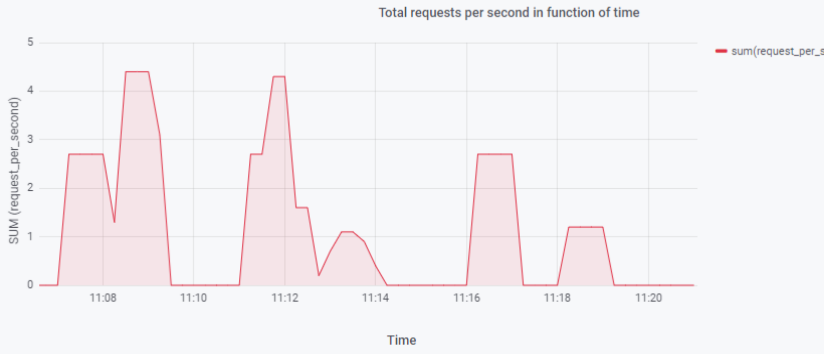
- SystemD service to restart: `kubelet.service`

1 minute Downscale Stabilization

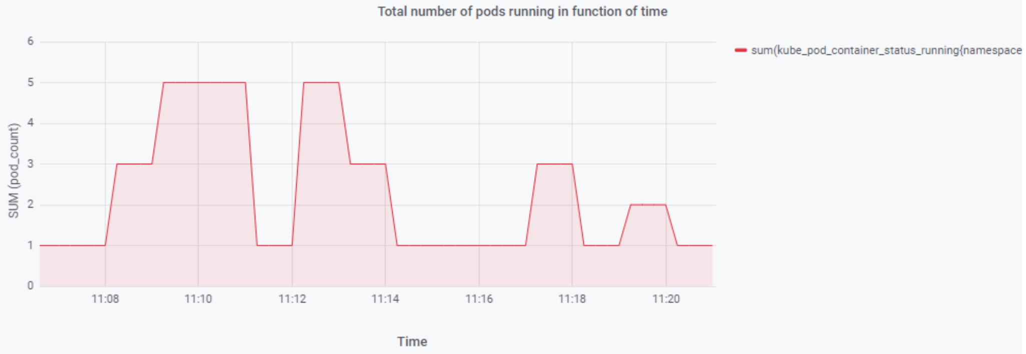
`request_per_second` on each pod in function of time



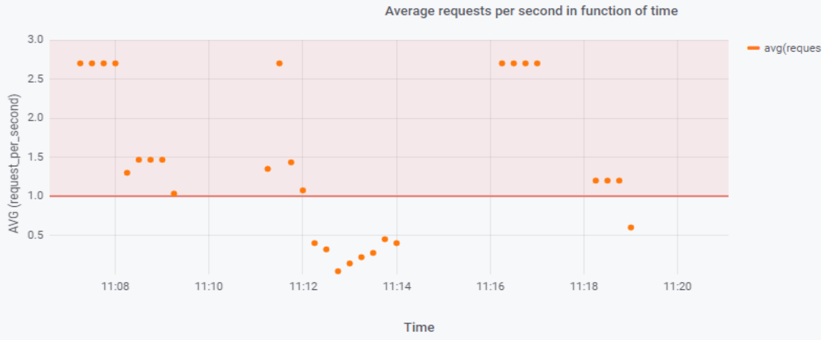
Total `request_per_second` in function of time



Total number of pods running in function of time

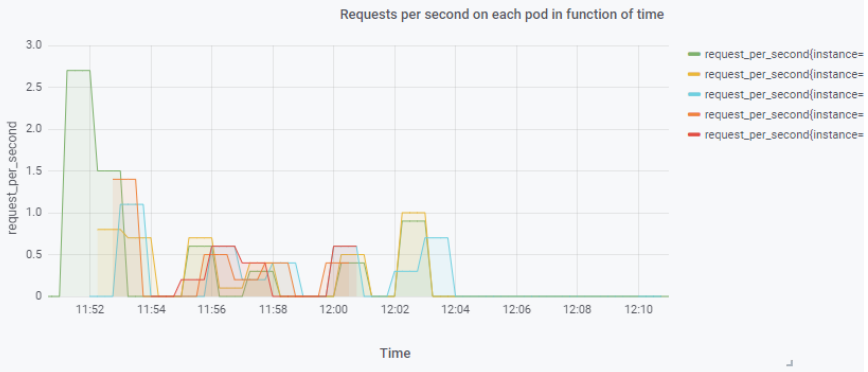


Average `request_per_second` in function of time

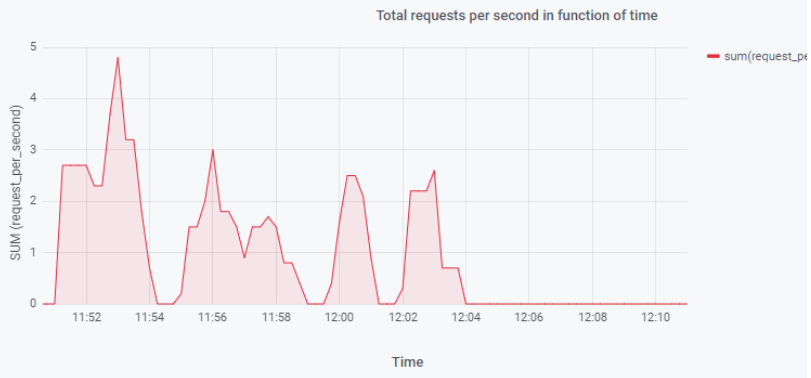


7 minutes Downscale Stabilization

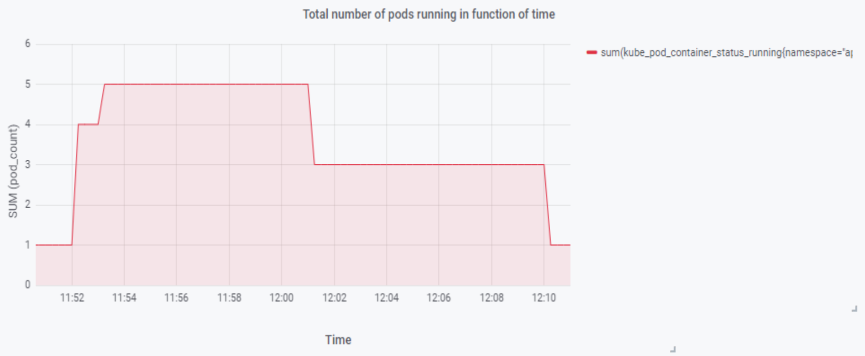
⌄ `request_per_second` on each pod in function of time



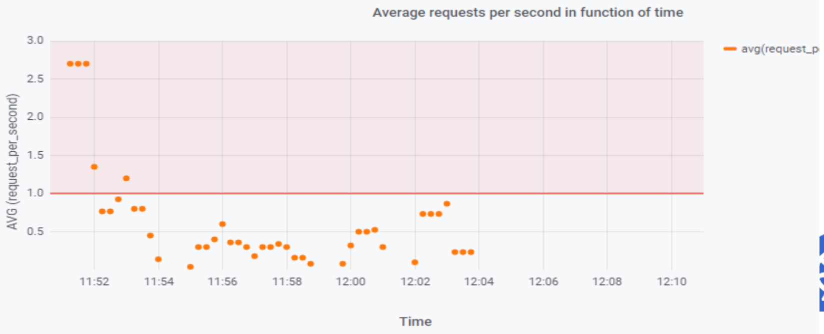
⌄ Total `request_per_second` in function of time



⌄ Total number of pods running in function of time



⌄ Average `request_per_second` in function of time



Analysis (run time)

- Total pod run_time. Sum of running durations of each pod involved in the scenario's execution [seconds]:
  - $run\_time = \sum_{i=1}^{i=N} \mathbf{T}_{run\ i}\ ,\ N \geq 1\ number\ of\ pods$

- Average pods running is the run_time divided by the scenario duration + the downscale-stabilization [count]:
  - $pod\_avg = \frac{run\_time}{\mathbf{T}_{scenario} + \tau}$

Analysis (over usage)

- Overusage factor [no UOM]:
  - $\alpha = \frac{metric\_value}{metric\_threshold}$
  - $metric_{value} > metric_{threshold}$

- Pods overusage: Sum of durations that each pod has been in overusage, multiplied by the previous factor, in part of the total run_time [%]:
  - $ovr\_us = \frac{\sum_{i=1}^{i=N} \sum_{j=1}^{j=M} \mathrm{T}_{overusage,\, j_i} \times \alpha_{j_i}}{run\_time} * 100$
  - $N \geq 1\ number\ of\ pods$
  - $M \geq 1\ number\ of\ overusage\ intervals$

Analysis (under-usage)

- Underusage factor [no UOM]:
  - $\beta = 1 - \frac{metric_{value}}{metric_{threshold}}$
  - $metric\_value < metric\_threshold$

- Pods overusage: Sum of durations that each pod has been in underusage, multiplied by the previous factor, in part of the total run_time. [%]:
  - $udr\_us = \frac{\sum_{i=1}^{i=N} \sum_{j=1}^{j=M} \tau_{underusage,\, j_{\,i}} \times \beta_{j_i}}{run\_time} * 100$
  - $N \geq 1\, number\ of\ pods,$
  - $M \geq 1\, number\ of\ overusage\ intervals$

Automated {over,under}-usage calculation

- Let’s wrap these formulas on a python script

```
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
```

Automated {over,under}-usage calculation

- Example:

```
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%
```

Summary

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%

➔ The challenge is to find a compromise between an high Downscale Stabilization which gives less over usage and a lower one which degrade the QoE. SLAs are important for choosing a value.

Conclusion

Conclusion

- Autoscaling is a technical need to answer 2020’s business challenges
- Applications has to be properly designed and instrumented to support autoscaling with custom metrics
- Setting up advanced autoscaling is “easy”, and it can be quickly plugged in an existing application monitoring system (Prometheus, StackDriver, DataDog…)
- Even if Kubernetes is not the only product to propose this kind of features, it makes it reliable and open by providing a set of standards APIs.

Thank you!

Gaël LARGER

/in/gael-larger
(+33 | 0) 6 06 85 23 56
gaellarger@gmail.com
Git{Lab,Hub}: @Gaeel

Deep Dive: Practical implementation (Optional part)

Prerequisites

- A Kubernetes cluster, Minikube is fine for development
- kubectl and helm binaries on the development / admin machine
- Tiler is deployed on the cluster
- If using Ingresses, the Ingress controller is installed (for Minikube, the plugin should be activated) and the DNS configured
- Optionally, the k8s dashboard can be deployed

A sample application

- Stack for the demo:
  - Kotlin
  - Spring Boot, Data JPA
  - OpenJDK (JRE) 11
- Needs to expose:
  - Readiness and health status (it can be different routes)
  - Metrics, Prometheus format

```
curl -sk https://application.k8s.home.lab/health | jq
{
  "status": "up"
}
```

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

Containerizing the application

- Enables Docker to build containers from a serie of instructions, written in a Dockerfile

```
# 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"]
```

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

Namespaces

- A namespace is a “virtual k8s cluster” running on a k8s cluster
  - 1 namespace = 1 project, 1 team
  - Resources names can be re-utilized over namespaces, not in the same namespace
- Let’s define two:

```
kubectl create -f namespaces.yaml
```

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

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

Deploying the application (1)

- We can write our own Helm chart if we need templatising or just use a K8s manifest:

```yaml
---
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
```

```yaml
        # 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 an HTTP GET on :8080/health
          httpGet:
            path: /health
            port: 8080
        resources:
          limits:
            memory: 256Mi
```

Deploying the application (2)

- Service and Ingress declaration

```yaml
---
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
```

```yaml
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: application-ingress
  namespace: application
spec:
  rules:
  - host: application.k8s.home.lab
    http:
      paths:
      - path: /
        backend:
          serviceName: glar-demo-application
          servicePort: 80
```

Enabling the metrics server

- A plugin in Minikube:

`minikube addons enable metrics-server`

- We can now get pods and nodes usage metrics:
  - `kubectl top nodes`

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

  - `kubectl top pods`

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

Prometheus

- Easy to deploy with Helm
- Single config file, which are the values used by the templating engine of Helm to generate the YAML sent to the k8s API for deploying the release:

```yaml
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:
      # https://github.com/prometheus/prometheus/blob/
      # master/documentation/examples/prometheus-kubernetes.yml
```

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

Metrics adapter

- Easy to deploy with Helm

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

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

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


```
{
  "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"
}
```

Creating an HPA for the application

```yaml
---
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
```

```
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
```

Template

Title only

Section Header

Title and body

Title and two columns

Title Slide #2

One Column Text

- Text
    - Text
        - Text

Main Point

Caption

Custom Layout