

# Kubernetes, mise en oeuvre

Pierre Baconnier

2022-01-28

## Contents

<b>Kubernetes, mise en oeuvre</b>	<b>3</b>
Gestion avancée des conteneurs . . . . .	3
Qu'est-ce qu'un container ? . . . . .	3
Dockerfile . . . . .	9
Création et automatisation d'images personnalisées . . . . .	10
Déploiement d'une image personnalisée . . . . .	11
Un conteneur et plusieurs services . . . . .	13
Création et automatisation d'images personnalisées (TP) . . . . .	14
Introduction à kubernetes . . . . .	15
De la virtualisation à la conteneurisation . . . . .	17
Solutions d'installation (Minikube, On-Premise, etc.) . . . . .	17
Installation et configuration de docker . . . . .	20
Accéder au cluster Kubernetes: CLI (kubectl), GUI (dashboard) et APIs . . . . .	20
Déploiement, publication manuelle et introspection du déploiement . . . . .	23
Déploiement d'une plateforme de test (TP) . . . . .	26
Les fichiers descriptifs . . . . .	26
Syntaxe YAML . . . . .	26
Scalabilité d'un déploiement . . . . .	30
Stratégie de mise à jour sans interruption . . . . .	32
Revenir à une révision précédente . . . . .	34
Suppression d'un déploiement . . . . .	36
Publication et analyse d'un déploiement (TP) . . . . .	36
Architecture Kubernetes . . . . .	36
Composants du Control Plane . . . . .	37
Concepts: objets stateful, stateless . . . . .	41
Quelques exemples d'architecture cloud hybrides ou on-premise . . . . .	43
Quelques exemples d'architecture cloud public . . . . .	46
Exploiter Kubernetes . . . . .	49
Types de services . . . . .	49
Labels et choix d'un node pour le déploiement . . . . .	50
Affinité et anti-affinité . . . . .	52
Daemons set, health check, config map et secrets . . . . .	54
Persistent Volumes et Persistent Volumes Claim . . . . .	60
Déploiement d'une base de données et d'une application (TP) . . . . .	63
TP Version 2 (on K3s instead of AWS) . . . . .	64
Kubernetes en production . . . . .	65

Frontal administrable Ingress . . . . .	65
Limitation de ressources . . . . .	67
Service Discovery (env, DNS) . . . . .	69
Les namespaces et les quotas . . . . .	70
Gestion des accès (RBAC) . . . . .	72
Haute disponibilité et mode maintenance . . . . .	76
Gestion des droits user, sa et mise en place de services exposés (TP) . . . . .	81
Déploiement d'un cluster Kubernetes . . . . .	82
Déploiement d'un master-nodeadm, d'un master-node, d'un worker-node . . . . .	82
Mise en place du dashboard et du réseau . . . . .	84
Déploiement d'un cluster (TP) . . . . .	86
Annexe A: Kubectl Command CheatSheet . . . . .	89

# Kubernetes, mise en oeuvre

## Gestion avancée des conteneurs

### Qu'est-ce qu'un container ?



---

**“Caisson métallique parallélépipédique conçu pour le transport de marchandise par différents mode de transports. Ses dimensions ont été standardisées au niveau international et il est muni à tous les angles de pièces de préemption permettant de l’arrimer ou de le transborder d’un véhicule à l’autre.”** *Définition d’un container maritime*

Nous allons voir que ces propriétés se retrouvent dans la conception informatique du container.

C'est tout simplement faire tourner un ensemble de processus systèmes ou métier au sein d'un même environnement, isolé.

Une des problématique de la virtualisation par hyperviseur est celle de la boîte noire. Difficile de recenser l'ensemble des composants, ce qui peut être critique en cas de panne de la VM, impossible donc d'en déterminer l'état.

Comment faire également lorsque les ressources ne sont pas partagées, comme le

cache des applications par exemple ?

Physiquement, un container est un système de fichiers contenant l'ensemble des binaires dont on a besoin pour faire tourner notre application.

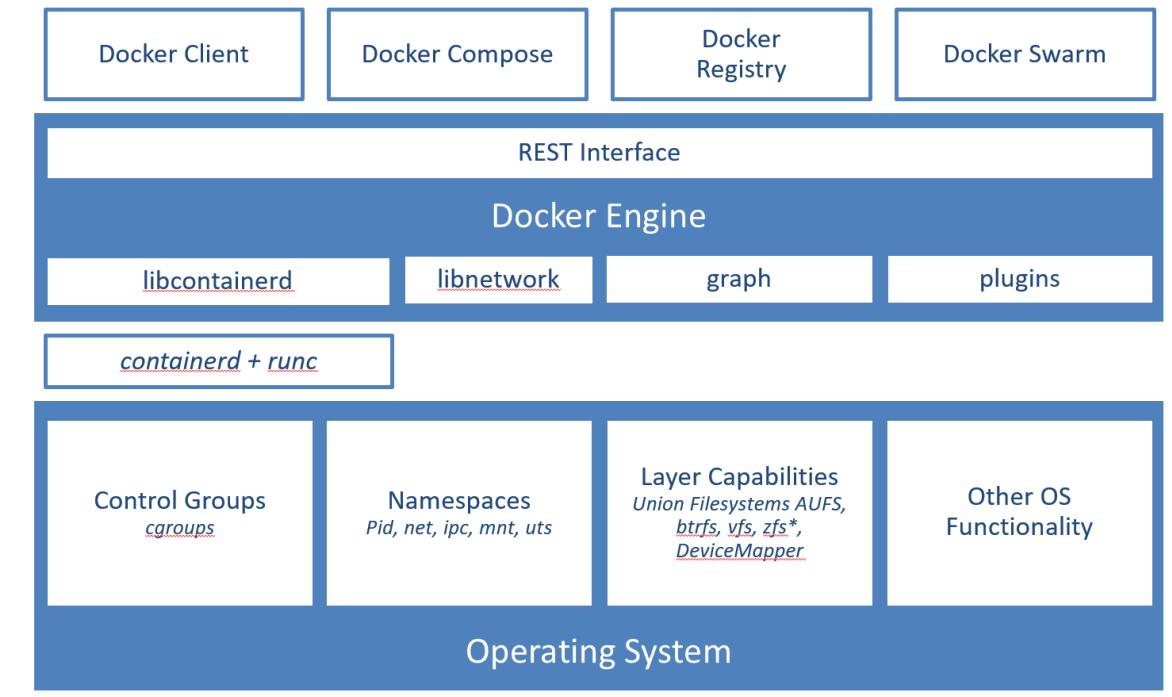
L'image apporte également des métadonnées rattachées aux layers dont on peut ensuite déterminer la composition exacte.

A la fin j'ai une brique atomique, je n'ai pas besoin de savoir quelles sont les couches successivement empilées du container.

Cela permet de résoudre la problématique du packaging OS (.deb, .rpm, etc.), non agnostique.

*Docker on Linux*

## Architecture In Linux



---

Les containers sont immutables, ils ne laissent rien derrière eux, ont un format autosufisant.

Les développeurs peuvent passer d'un langage à l'autre pour les applications sans jamais impacter l'infrastructure de PROD.

C'est toujours un 'runner' de container tel containerd qui execute le processus sur l'hôte mais on se fiche désormais de savoir quel est le launcher sous le container (npm, shell, cargo, etc.).

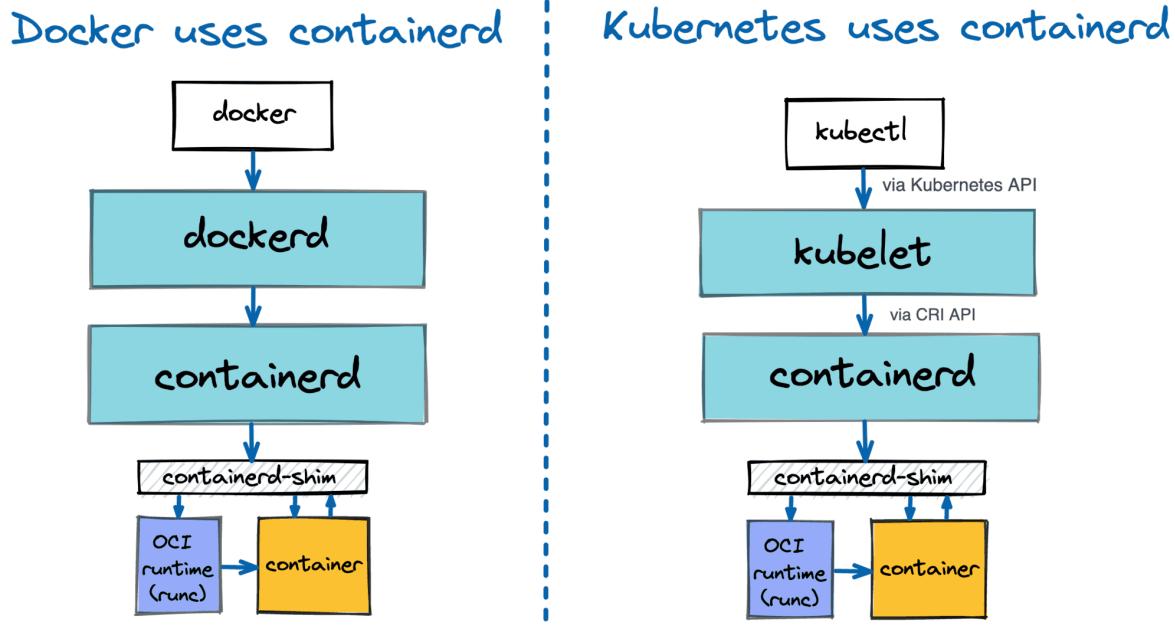
Les containers amènent de l'auditabilité en production.

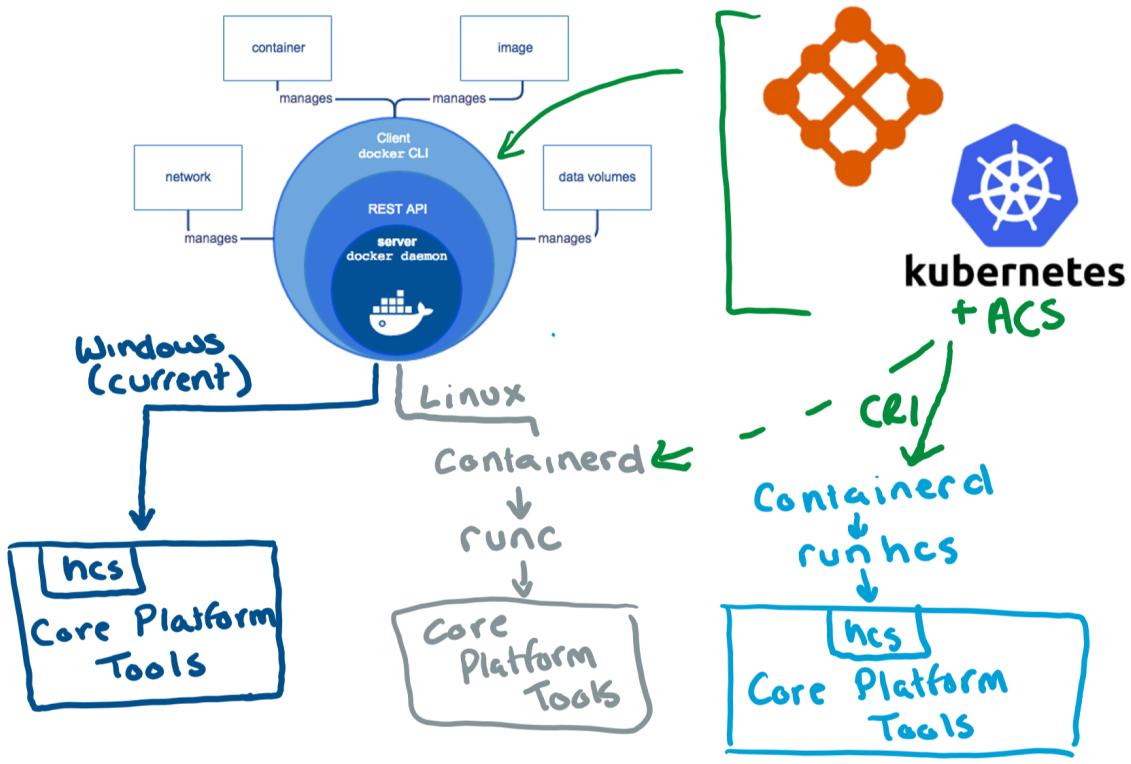
Autour de cette infrastructure, d'autres composants, tel un orchestrateur peuvent être intégrés.

### Le container runtime est le logiciel responsable de l'exécution des conteneurs

Kubernetes est compatible avec plusieurs containers runtime: **Docker, containerd, cri-o, rktlet ainsi que toute implémentation de Kubernetes CRI (Container Runtime Interface)**

*Docker and Kubernetes use containerd*





### Ressources supplémentaires

Image disposant d'une binary toolbox minimaliste: [Busybox](#)

Outil d'introspection des layers des images: [Dive](#)

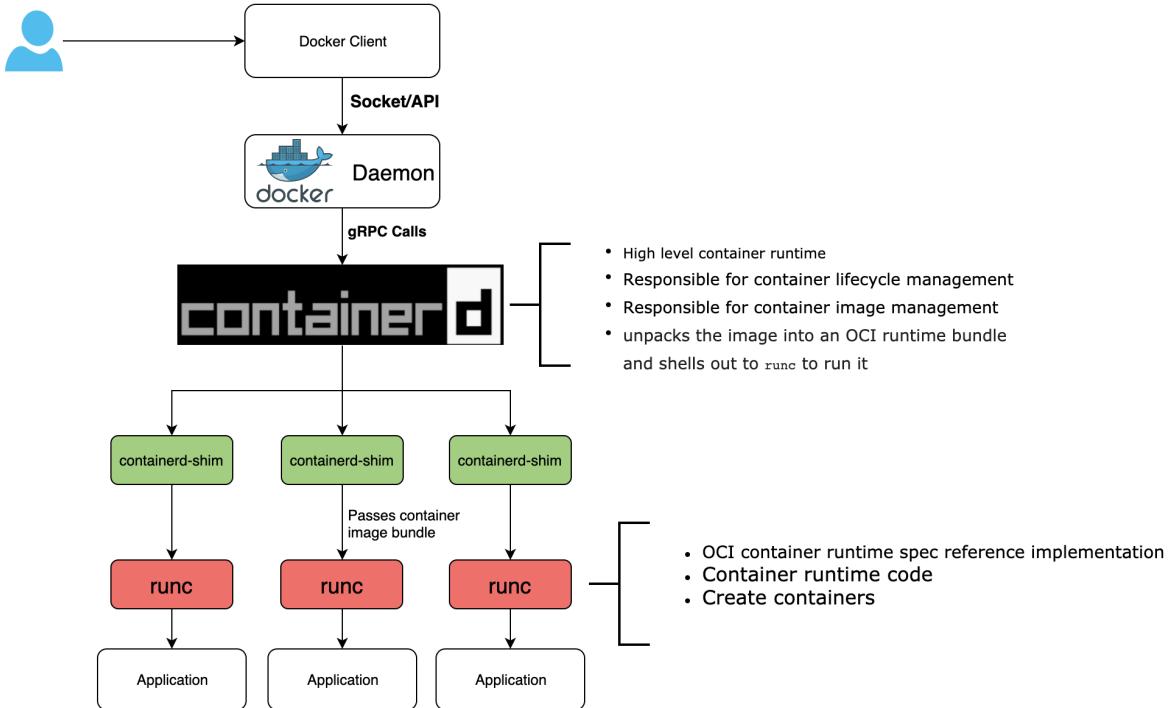
*Demo CI dive*

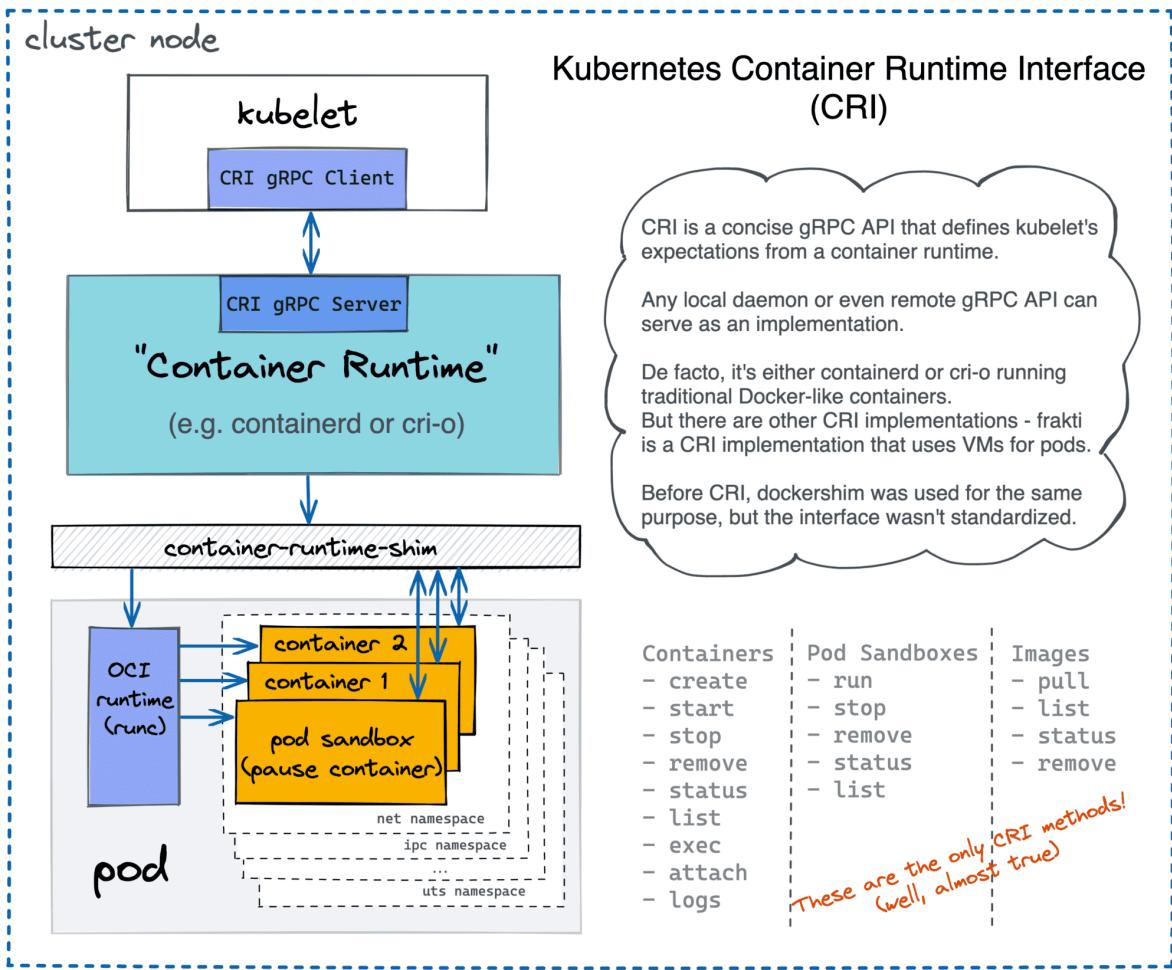
```
[wagoodman@kiwi dive] $ ci-integration $ CI=true build/dive dive-test
Using config file: /home/wagoodman/.dive.yaml
Fetching image...
Parsing image...
└── [layer: 1] 1871059774abe69 : [=====] 100 % (1/1)
└── [layer: 2] 28cfe03618aa2e9 : [=====] 100 % (415/415)
└── [layer: 3] 3d4ad907517a021 : [=====] 100 % (2/2)
└── [layer: 4] 461885fc2258915 : [=====] 100 % (4/4)
└── [layer: 5] 49fe2a475548bfa : [=====] 100 % (4/4)
└── [layer: 6] 5eca617bdc3bc06 : [=====] 100 % (3/3)
└── [layer: 7] 80cd2ca1ffc8996 : [=====] 100 % (3/3)
└── [layer: 8] 81b1b002d4b4c13 : [=====] 100 % (3/3)
└── [layer: 9] a10327f68ffed4a : [=====] 100 % (2/2)
└── [layer: 10] aad36d0b05e71c7 : [=====] 100 % (4/4)
└── [layer: 11] c99e2f8d3f62826 : [=====] 100 % (3/3)
└── [layer: 12] cfb35bb5c127d84 : [=====] 100 % (2/2)
└── [layer: 13] f07c3eb88757239 : [=====] 100 % (3/3)
└── [layer: 14] f2fc54e25cb7966 : [=====] 100 % (2/2)

Analyzing image...
efficiency: 98.4421 %
wastedBytes: 32025 bytes (32 kB)
userWastedPercent: 2.6376 %
Run CI Validations...
Using CI config: .dive-ci
PASS: highestUserWastedPercent
SKIP: highestWastedBytes: rule disabled
FAIL: lowestEfficiency: image efficiency is too low (efficiency=0.9844212134184309 < threshold=0.99)
Result:FAIL [Total:3] [Passed:1] [Failed:1] [Warn:0] [Skipped:1]
X:1 [wagoodman@kiwi dive] $ ci-integration $
```

---

## Container runtime de docker: Containerd





```

NAME:
  ctr - 

██████

containerd CLI

USAGE:
  ctr [global options] command [command options] [arguments...]

VERSION:
  1.5.2-0ubuntu1~20.04.2

DESCRIPTION:
ctr is an unsupported debug and administrative client for interacting
with the containerd daemon. Because it is unsupported, the commands,
options, and operations are not guaranteed to be backward compatible or
stable from release to release of the containerd project.

```

---

## Dockerfile

**Les Dockerfiles contiennent des instructions pour créer une image Docker. Chaque instruction est écrite sur une ligne et est donnée sous la forme <argument(s)>. Les fichiers Docker sont utilisés pour créer des images Docker à l'aide de la commande `docker build`**

Les principales directives pouvant être utilisées dans un Dockerfile sont:

- **FROM** - Il s'agit de la première instruction. Elle est indispensable, car elle définit l'image de base avec le système d'exploitation et l'application portant le reste des couches. **Vous pouvez également spécifier une image de base minimale en utilisant l'instruction `FROM scratch`.**
- **COPY** - Cette commande ajoute dans votre image des fichiers locaux ou distants tels le code source d'une application ou un fichier de configuration. Il existe aussi une commande ADD qui peut être très utile lorsque l'on veut extraire à la volée un tar.gz. ADD permet de faire ceci en une instruction alors que COPY ne permet que de copier la source vers une destination.
- **RUN** - Exécuter avec RUN toute commande souhaitée pour créer une nouvelle couche sur l'image.
- **CMD** - Pour spécifier une commande à exécuter seulement lors du lancement du conteneur.

**HelloWorld Dockerfile** Un fichier Dockerfile minimal pourrait ressembler à ceci:

```
FROM alpine
CMD ["echo", "Hello Classroom!"]
```

Cela indiquera à Docker de créer une image basée sur Alpine (FROM), une distribution minimale pour les conteneurs et d'exécuter une commande spécifique (CMD) lors de l'exécution de l'image résultante.

Construisez et exécutez-le:

```
docker build -t hello .
docker run --rm hello
```

```
# Output
Hello Classroom!
```

## Création et automatisation d'images personnalisées

Exemple de code applicatif go:

```
package main

import (
    "net/http"
    "os"

    "github.com/labstack/echo/v4"
    "github.com/labstack/echo/v4/middleware"
)

func main() {

    e := echo.New()

    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    e.GET("/", func(c echo.Context) error {
        return c.HTML(http.StatusOK, "Hello, Docker! <3")
    })

    e.GET("/ping", func(c echo.Context) error {
        return c.JSON(http.StatusOK, struct{ Status string }{Status: "OK"})
    })

    httpPort := os.Getenv("HTTP_PORT")
    if httpPort == "" {
        httpPort = "8080"
    }

    e.Logger.Fatal(e.Start(": " + httpPort))
}
```

*Dockerfile*

```
# syntax=docker/dockerfile:1
```

```
FROM golang:1.16-alpine
```

```
WORKDIR /app
```

```
COPY go.mod ./
```

```
COPY go.sum ./
```

```
RUN go mod download
```

```
COPY *.go ./
```

```
RUN go build -o /docker-gs-ping
```

```
EXPOSE 8080
```

```
CMD [ "/docker-gs-ping" ]
```

Golang build images: [Golang build images](#)

Docker multistage building: [Multistage building](#)

## Déploiement d'une image personnalisée

Init du package applicatif golang:

```
go mod init example.com/echo
```

```
go: creating new go.mod: module example.com/echo  
go: to add module requirements and sums:  
    go mod tidy
```

Installation des dépendances:

```
go mod tidy
```

```
go: finding module for package github.com/labstack/echo/v4/middleware  
go: finding module for package github.com/labstack/echo/v4  
go: downloading github.com/labstack/echo/v4 v4.6.3  
go: downloading github.com/labstack/echo v3.3.10+incompatible  
go: found github.com/labstack/echo/v4 in github.com/labstack/echo/v4 v4.6.3  
go: found github.com/labstack/echo/v4/middleware in github.com/labstack/echo/v4 v4.6.3  
go: downloading golang.org/x/crypto v0.0.0-20210817164053-32db794688a5  
go: downloading golang.org/x/net v0.0.0-20210913180222-943fd674d43e  
go: downloading github.com/labstack/gommon v0.3.1  
go: downloading github.com/golang-jwt/jwt v3.2.2+incompatible  
go: downloading github.com/valyala/fasttemplate v1.2.1  
go: downloading golang.org/x/time v0.0.0-20201208040808-7e3f01d25324  
go: downloading github.com/mattn/go-isatty v0.0.14  
go: downloading github.com/mattn/go-colorable v0.1.11  
go: downloading github.com/valyala/bytebufferpool v1.0.0  
go: downloading golang.org/x/sys v0.0.0-20211103235746-7861aae1554b
```

Build de l'image personnalisée:

```

docker build --tag docker-gs-ping .

[+] Building 19.3s (12/12) FINISHED

=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 220B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/golang:1.16-alpine
=> [internal] load build context
=> => transferring context: 5.75kB
=> [1/7] FROM docker.io/library/golang:1.16-alpine@sha256:c4b36cff558405c8368be606325261a21a9a3ae9f79
=> => resolve docker.io/library/golang:1.16-alpine@sha256:c4b36cff558405c8368be606325261a21a9a3ae9f79
=> => sha256:c4b36cff558405c8368be606325261a21a9a3ae9f79dc1bd3fff9f831a0411f2 1.65kB / 1.65kB
=> => sha256:802b7b395fac3d9b0ed552be257a3f19739486f6e712b7f5276f76214ad1efaf1 1.36kB / 1.36kB
=> => sha256:4bcb0d501de3d5c9d95c65a8ef94ef2d169244bf06970413780748ac6fc7536c 5.20kB / 5.20kB
=> => sha256:59bf1c3509f33515622619af21ed55bbe26d24913cedbca106468a5fb37a50c3 2.82MB / 2.82MB
=> => sha256:666ba61612fd7c93393f9a5bc1751d8a9929e32d51501dba691da9e8232bc87b 282.16kB / 282.16kB
=> => sha256:8ed8ca4862056a130f714accb3538decfa0663fec84e635d8b5a0a3305353dee 155B / 155B
=> => sha256:fa6e261410ce0cebd518eb5ee56b9bb56fa66883992dc54ab7bd25afdfa2b521 105.85MB / 105.85MB
=> => sha256:033ff68f5bd26b7f1a7567e74101649886332f6bffed2928025b2dc88a49a909 155B / 155B
=> => extracting sha256:59bf1c3509f33515622619af21ed55bbe26d24913cedbca106468a5fb37a50c3
=> => extracting sha256:666ba61612fd7c93393f9a5bc1751d8a9929e32d51501dba691da9e8232bc87b
=> => extracting sha256:8ed8ca4862056a130f714accb3538decfa0663fec84e635d8b5a0a3305353dee
=> => extracting sha256:fa6e261410ce0cebd518eb5ee56b9bb56fa66883992dc54ab7bd25afdfa2b521
=> => extracting sha256:033ff68f5bd26b7f1a7567e74101649886332f6bffed2928025b2dc88a49a909
=> [2/7] WORKDIR /app
=> [3/7] COPY go.mod .
=> [4/7] COPY go.sum .
=> [5/7] RUN go mod download
=> [6/7] COPY *.go .
=> [7/7] RUN go build -o /docker-gs-ping
=> exporting to image
=> => exporting layers
=> => writing image sha256:d6282f70c0eed3c1f1f5c705c99c64c0aa1ac703d240630a9a0be1b437dc2025
=> => naming to docker.io/library/docker-gs-ping

```

Run de l'image:

```
docker run -it docker.io/library/docker-gs-ping
```

```

  _/----/----/----\_
 / _/ / _/ \_ \_ \_ \
 /__/\_/_/ / / \_/_/ v4.6.3
High performance, minimalist Go web framework
https://echo.labstack.com
----- 0/-----
0\_
-> http server started on [::]:8080

```

## Un conteneur et plusieurs services

**Le processus principal en cours d'exécution d'un conteneur est le ENTRYPOINT et/ou le CMD à la fin du Dockerfile. Il est généralement recommandé de séparer les zones de préoccupation en utilisant un service par conteneur. Ce service peut être divisé en plusieurs processus (par exemple, le serveur Web Apache lance plusieurs processus de travail). Il est acceptable d'avoir plusieurs processus, mais pour tirer le meilleur parti de Docker, évitez qu'un conteneur soit responsable de plusieurs aspects de votre application globale. Vous pouvez connecter plusieurs conteneurs à l'aide de réseaux définis par l'utilisateur et de volumes partagés.**

Le processus principal du conteneur est responsable de la gestion de tous les processus qu'il démarre. Dans certains cas, le processus principal n'est pas bien conçu et ne gère pas la "récolte" (l'arrêt) des processus enfants de manière élégante lorsque le conteneur sort. Si votre processus entre dans cette catégorie, vous pouvez utiliser l'option –init lorsque vous exécutez le conteneur. L'option –init insère un minuscule processus d'initialisation dans le conteneur en tant que processus principal, et gère la récupération de tous les processus à la sortie du conteneur. La gestion de ces processus de cette manière est supérieure à l'utilisation d'un processus init complet tel que sysvinit, upstart, ou systemd pour gérer le cycle de vie des processus dans votre conteneur.

Si vous devez exécuter plus d'un service dans un conteneur, vous pouvez le faire de plusieurs manières différentes.

Placez toutes vos commandes dans un script wrapper, avec des informations de test et de débogage. Exécutez le script wrapper comme votre CMD. Voici un exemple très naïf.

Tout d'abord, le script wrapper :

```
#!/bin/bash

# Start the first process
./my_first_process &

# Start the second process
./my_second_process &

# Wait for any process to exit
wait -n

# Exit with status of process that exited first
exit $?
```

Puis, le dockerfile:

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest
COPY my_first_process my_first_process
COPY my_second_process my_second_process
COPY my_wrapper_script.sh my_wrapper_script.sh
```

**CMD** ./my\_wrapper\_script.sh

Si vous avez un processus principal qui doit démarrer en premier et rester en cours d'exécution, mais que vous avez temporairement besoin de lancer d'autres processus (peut-être pour interagir avec le processus principal), vous pouvez utiliser le contrôle des tâches de bash pour faciliter cela.

```
#!/bin/bash

# turn on bash's job control
set -m

# Start the primary process and put it in the background
./my_main_process &

# Start the helper process
./my_helper_process

# the my_helper_process might need to know how to wait on the
# primary process to start before it does its work and returns

# now we bring the primary process back into the foreground
# and leave it there
fg %1

# syntax=docker/dockerfile:1
FROM ubuntu:latest
COPY my_main_process my_main_process
COPY my_helper_process my_helper_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

Notons toutefois que cette façon de procéder n'est que peu adaptée à une utilisation dans Kubernetes où dans une approche orientée micro-services l'on préférera 1 container pour 1 service exposé.

*Sources*

[Multi Service container](#)

## Création et automatisation d'images personnalisées (TP)

**Partie 1** Dans ce TP vous aurez à charge de réaliser une image personnalisée sur la base d'un autre code applicatif (Golang/Java/Python,etc.) que vous souhaitez containeriser.

L'image devra être buildée et run sans erreurs.

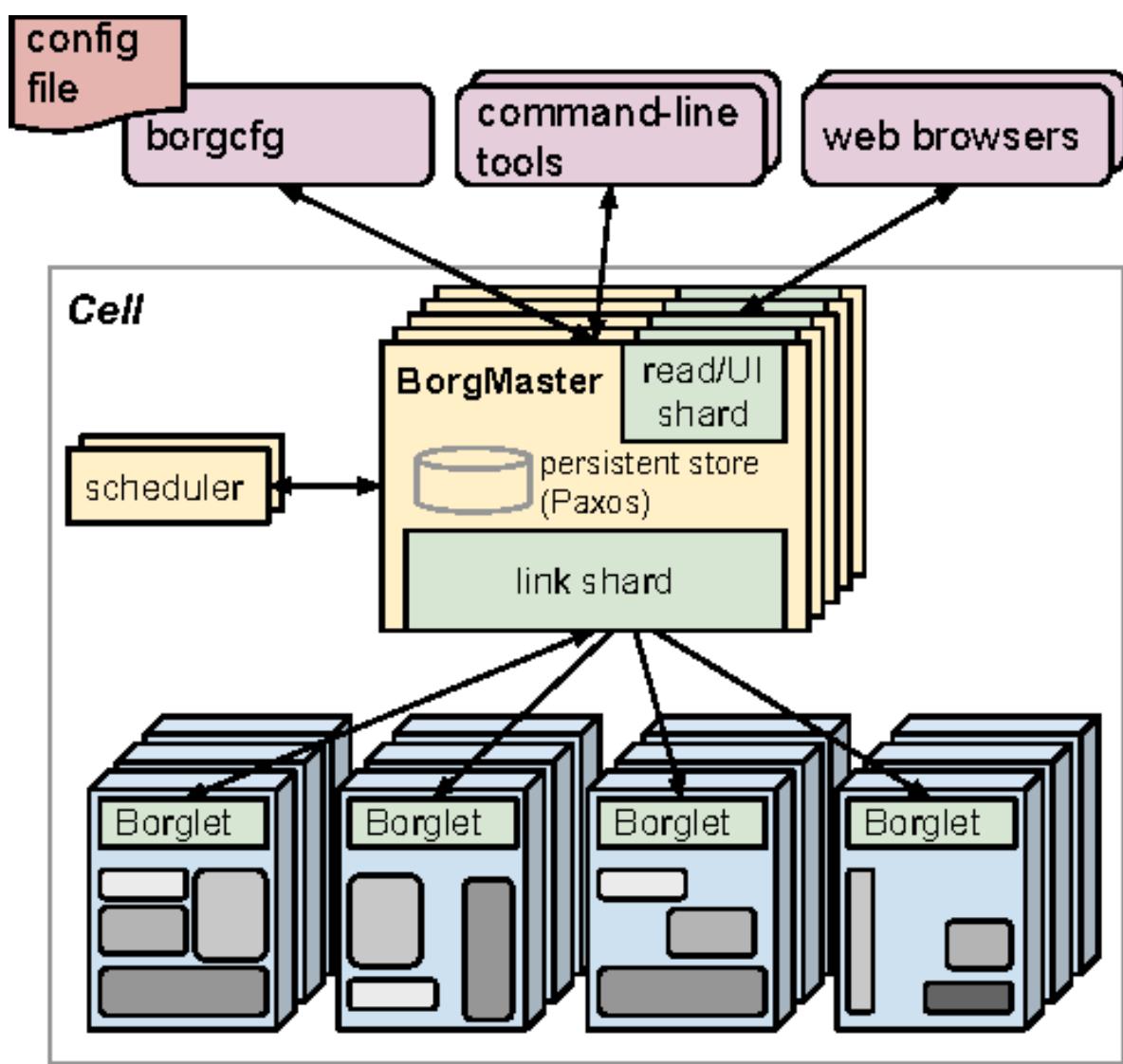
**Partie 2 (Facultatif)** Étoffer votre image en rajoutant plusieurs services comme vu dans la section “Un conteneur et plusieurs services”.

**Partie 3 (Facultatif)** Étoffer à nouveau votre image en la construisant cette fois selon les règles du multi-stage building (<https://docs.docker.com/develop/develop>-

images/multistage-build) qui visera à séparer le build du ou des binaires des processus qui seront run par l'image finale.

## Introduction à kubernetes

Kubernetes, souvent abrégé k8s, est un orchestrateur de conteneurs conçu par Google et sorti en 2014. C'est un outil Open Source (code source disponible sur github), écrit en Go, qui trouve ses racines dans Borg le système interne de Google qui gère l'infrastructure du géant de Mountain View.



Ce n'est pas le seul orchestrateur de conteneurs sur le marché, il en existe d'autres comme **Docker Swarm**, **Apache Marathon**, **Nomad** ou **Kontena**. Cependant, aujourd'hui K8s apparaît comme hégémonique sur le marché et ses concurrents ne

se divisent que peu de parts.

**Le nom Kubernetes tire son origine du grec ancien, signifiant capitaine ou pilote ou encore timonier et est la racine de gouverneur et cybernetic. K8s est l'abréviation dérivée par le remplacement des 8 lettres "ubernete" par "8".**

Soit celui qui tient la barre d'un bateau. Si l'on reprend l'analogie de Docker avec les conteneurs maritimes, Kubernetes est le capitaine qui dirige le porte-conteneurs.

Aujourd'hui, en 2022, force est de constater que Kubernetes est devenu incontournable et la plupart des cloud providers proposent désormais des solutions Kubernetes hébergées.

Il faudra attendre **2013 pour que Google lance sa première offre IAAS** en ayant été ralenti par l'échec de son PAAS AppEngine.

**Un des enjeux de K8S fut de réussir à mixer la flexibilité du IAAS et la facilité d'accès, la descriptabilité du PAAS, ce que l'on pourrait appeler l'IPAAS.**

Il fallait donc un standard pour l'orchestration, une API qui puisse gérer l'ensemble de ces caractéristiques.

Kubernetes est strictement impératif, nous demandons un état sans avoir à connaître les étapes intermédiaires.

Par exemple:

- “J'aimerai un espace isolé dont les applications n'utilisent que 4Go de RAM et 2 CPU”
- “J'aimerai une base de données répliquée 2 fois”
- “J'aimerai un storage avec une rétention de 6 mois”
- “J'aimerai que des mises a jour soit déployées mais n'impactent que les applications du ns ‘web’ ou correspondant au tag ‘web’”

Pour utiliser Kubernetes, vous utilisez les objets de l'API Kubernetes pour décrire l'état souhaité de votre cluster: quelles applications ou autres processus que vous souhaitez exécuter, quelles images de conteneur elles utilisent, le nombre de réplicas, les ressources réseau et disque que vous mettez à disposition, et plus encore. Vous définissez l'état souhaité en créant des objets à l'aide de l'API Kubernetes, généralement via l'interface en ligne de commande, kubectl. Vous pouvez également utiliser l'API Kubernetes directement pour interagir avec le cluster et définir ou modifier l'état souhaité.

Une fois que vous avez défini l'état souhaité, le plan de contrôle Kubernetes (control plane en anglais) permet de faire en sorte que l'état actuel du cluster corresponde à l'état souhaité. Pour ce faire, Kubernetes effectue automatiquement diverses tâches, telles que le démarrage ou le redémarrage de conteneurs, la mise à jour du nombre de réplicas d'une application donnée, etc.

**L'orchestration est gérée par des contrôleurs.** Ces contrôleurs sont compilés dans le kube-controller-manager.

Kubernetes est container-centric et va nous permettre de déployer et de gérer des conteneurs. Les conteneurs ne sont pas gérés individuellement. Au lieu de cela ils font partie d'un ensemble plus grand appelé **Pod**.

## **Un Pod se compose d'un ou de plusieurs conteneurs qui partagent une adresse IP, un accès au stockage et un espace de nommage.**

On va distinguer la notion de spec de la notion de statut.

S'il réussit à converger à l'issue de sa boucle de réconciliation, l'état est appliqué et le résultat retourné à l'api server.

Il était très compliqué d'atteindre l'indempotence avec Puppet ou Chef par exemple, car outre l'effet snowflake, certains composants tel le réseau on-premise, n'étaient pas prévus pour être indomptables.

Deux machines, 2 versions ne vont pas nécessairement réagir de la même façon. C'est l'effet snowflake car tous les flocons se ressemblent et il devient presque impossible distinguer/troubleshooter les différences sur des milliers de machines de production.

Kubernetes répond à ce problème car il maintient l'état et le retourne après action. On a donc un *avant* et un *après*.

Les pratiques de Kube sont basées sur Google mais également sur beaucoup d'autres suggestions de la communauté.

L'organe de certification de la CNCF permet d'assurer que toutes les solutions qui sont vendues sont compatibles.

On a un cluster ETCD qui sert à la persistance de l'état.

L'API server est en somme un "gardien du temple" qui s'assure de la validité des requêtes émises.

## **De la virtualisation à la conteneurisation**

L'ancienne façon (old way) de déployer des applications consistait à installer les applications sur un hôte en utilisant les systèmes de gestions de paquets natifs. Cela avait pour principale inconvénient de lier fortement les exécutables, la configuration, les bibliothèques et le cycle de vie de chacun avec l'OS. Il est bien entendu possible de construire une image de machine virtuelle (VM) immuable pour arriver à produire des publications (rollouts) ou retours arrières (rollbacks), mais les VMs sont lourdes et non-portables.

La nouvelle façon (new way) consiste à déployer des conteneurs basés sur une virtualisation au niveau du système d'opération (operation-system-level) plutôt que de la virtualisation hardware. Ces conteneurs sont isolés les uns des autres et de l'hôte : ils ont leurs propres systèmes de fichiers, ne peuvent voir que leurs propres processus et leur usage des ressources peut être contraint. Ils sont aussi plus faciles à construire que des VMs, et vu qu'ils sont décorrélés de l'infrastructure sous-jacente et du système de fichiers de l'hôte, ils sont aussi portables entre les différents fournisseurs de Cloud et les OS.

## **Solutions d'installation (Minikube, On-Premise, etc.)**

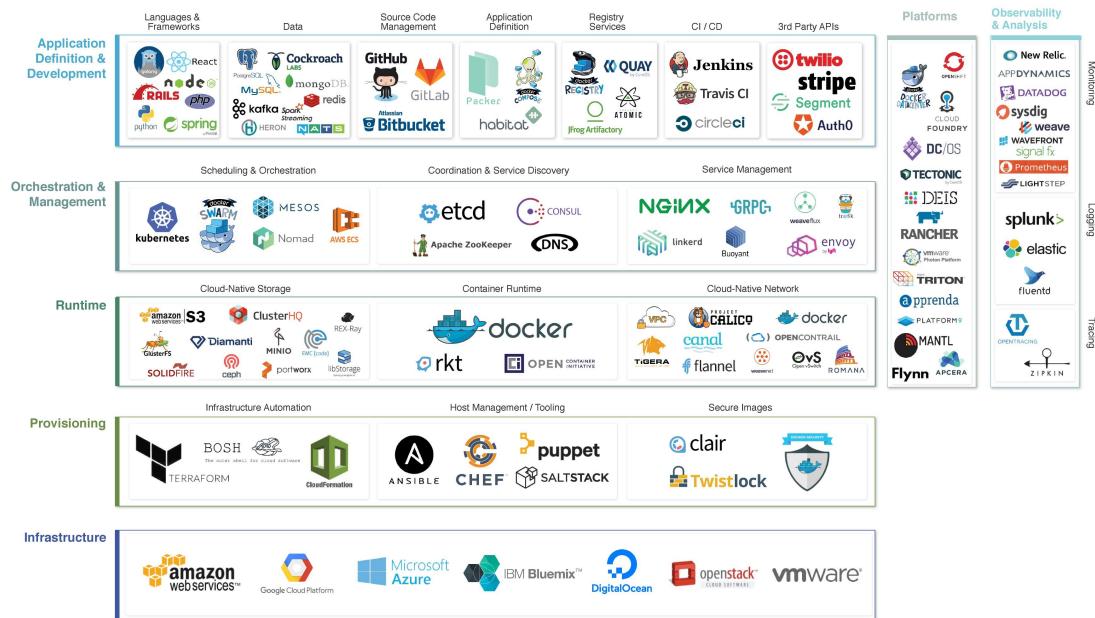
**K8S certified providers** [Marketplace Solutions](#)

[Kubernetes Training Partners](#)

Cloud Native Landscape v0.9.2



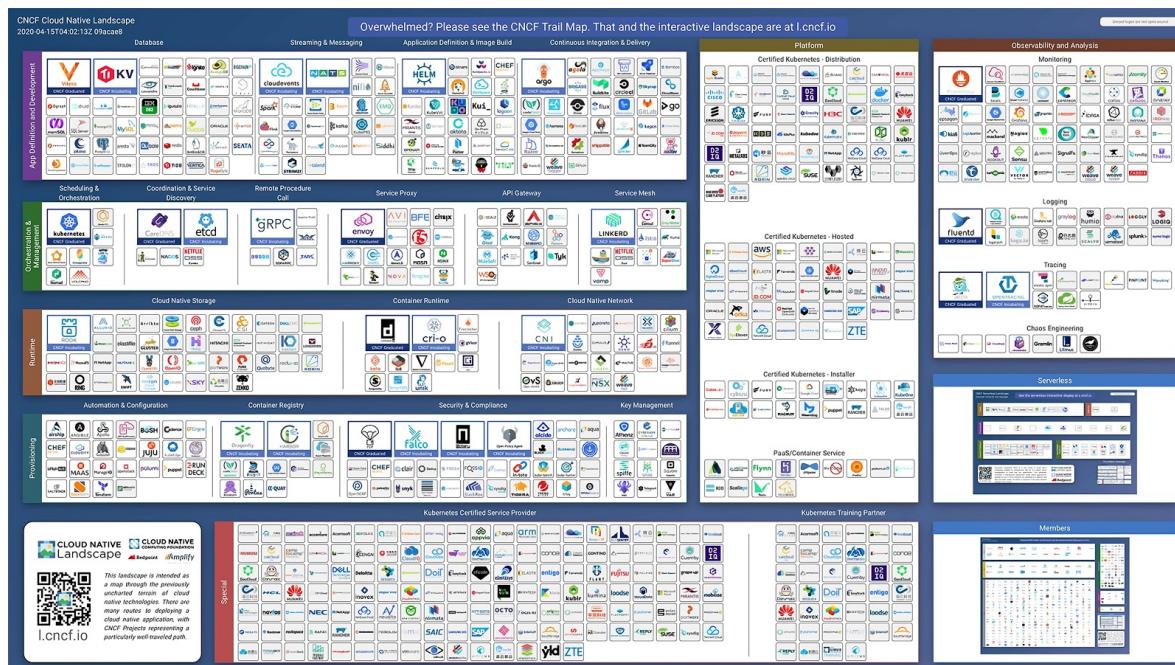
CLOUD NATIVE COMPUTING FOUNDATION



 <http://github.com/cncf/landscape>

 @dankohn1

@lennypauss



---

**Managed Solutions** Quelques solutions parmi les plus représentées:

- GCP GKE
- AWS EKS
- Azure AKS
- Platform9 (KUBE2GO)
- Pivotal
- Rancher
- Tectonic by CoreOS
- Stackpoint.io

**Le cas OVH: implémentation d'offre KAAS basée sur OpenStack et Platform9 solution** Dans le cadre de la Kubecon 2019, OVH annonce le déploiement de Kubernetes sur ses serveurs dédiés suite à un partenariat avec **Platform9**. Par conséquent, il est désormais le fournisseur européen qui propose le choix le plus diversifié en matière de déploiement Kubernetes.

**Le cas Bare-metal** En bare-metal l'expérience montre qu'avec CoreOs racheté par Redhat et intégré à Openshift, cette solution ainsi que Pivotal et Rancher sont les plus représentées sur le marché français.

## Solutions locales

**kind: Kubernetes dans Docker** Sous licence Apache, le projet kind (« Kubernetes in Docker») est un peu différent. Il déploie les nodes sous forme de conteneurs – chaque cluster étant identifié par une étiquette Docker. Son usage est donc plutôt orienté sur le test que le développement.

Le cœur fonctionnel est codé en Go. L'installation de base se fonde sur une image minimale d'Ubuntu. Elle contient les outils statiques nécessaires à Kubernetes, ajoute un point d'entrée sur chaque nœud pour réaliser des actions avant l'amorçage et paramètre un service systemd pour gérer la journalisation. Il en existe des variantes « étendues », contenant des tarballs (archives d'images) que le cluster chargera avant d'exécuter kubeadm.

L'installation est nettement plus lourde qu'avec les options sus-évoquées. Sur Mac comme sur Windows, il faut au minimum 6 Go de RAM (8 recommandés) pour la VM Docker. Il n'existe pas encore d'images Arm.

Parmi les autres limites, kind ne gère pas les applications à état. Il ne fonctionne par ailleurs pas dans la sandbox Linux de Chrome OS et ne prend pas en charge les conteneurs Windows. Depuis peu, il permet d'utiliser, côté hôte, docker et podman en rootless. L'implémentation réseau par défaut repose sur un module propre au projet (kindnetd). Associable à MetalLB pour la répartition de charge, il supporte l'IPv6, y compris en mode dual-stack.

**Minikube et MicroK8** Pas encore d'IPv6 sur Minikube, le « Kubernetes light officiel ». Sa première release remonte à 2016. Il couvre aujourd’hui Windows, Mac et Linux, sur x86-64. Son principe : exécuter des clusters à nœud unique dans une VM. Ses exigences : au minimum 2 CPU, 2 Go de RAM et 20 Go de disque. On peut aussi l'utiliser en remplacement de Docker Desktop, sans lancer Kubernetes.

Comme k0s, MicroK8s (AMD64, ARM64) peut faire l'objet d'un support commercial – 10 ans de maintenance – par son éditeur. Au-delà du développement local, il cible l'IoT et les environnements minimaux d'intégration continue. Canonical le distribue sous forme de Snap (et d'exécutable sur Windows) qui exécute nativement l'orchestrateur (sans VM).

De base, MicroK8s inclut le serveur d'API, le planificateur, le kubelet, l'interface CNI et kube-proxy. On peut l'étendre avec une trentaine d'add-on dont Istio, Kube-flow, OpenFaaS, Prometheus et l'opérateur NVIDIA. Configuration minimale recommandée: 4 Go de RAM et 20 Go de disque sur Linux avec l'installation par défaut (40 Go de disque sur Windows et Mac). Autre possibilité: installer MicroK8s dans un conteneur LXD. Comme Minikube, on est sur du déploiement de clusters à un seul nœud. Mais il est possible d'en ajouter et d'activer la haute disponibilité – avec DQLite en datastore.

## Installation et configuration de docker

**Ce guide part du postulat que les machines clientes pour cette formation qui seront amenées à utiliser un runtime docker tournent sous Windows, tandis que les outils seront installés dans un container linux.**

L'installation de docker référera donc au docker-desktop.

Cliquez sur le lien de téléchargement ci-dessous et suivez les étapes mentionnées:  
[Install Docker Desktop](#)

## Accéder au cluster Kubernetes: CLI (kubectl), GUI (dashboard) et APIs

Pour accéder au cluster nous aurons besoin de récupérer au préalable le **KUBE-CONFIG** de celui-ci. La récupération des données propres à générer, reconstruire ou accéder à un kubeconfig existant dépendra de la solution via laquelle le cluster a été déployé.

Voici un ensemble d'outils nécessaires sinon indispensables à une utilisation de Kubernetes en ligne de commande:

**Kubectl** `kubectl` est le client CLI écrit en go pour Kubernetes.

```
curl -LO \
"https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

Valider le téléchargement du binaire via la checksum:

```
curl -LO \
"https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
echo "$(cat <kubectl.sha256)  kubectl" | sha256sum --check
```

Si la vérification est conforme:

```
kubectl: OK
```

Si la vérification échoue, sha256 se termine avec un statut non nul et imprime une sortie similaire à:

```
kubectl: FAILED
```

```
sha256sum: WARNING: 1 computed checksum did NOT match
```

Installer kubectl:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Si vous n'avez pas l'accès root sur le système cible, vous pouvez quand même installer kubectl dans le répertoire `~/local/bin`:

```
chmod +x kubectl
mkdir -p ~/.local/bin/kubectl
mv ./kubectl ~/.local/bin/kubectl
# and then add ~/.local/bin/kubectl to $PATH
```

Vérifier l'installation:

```
kubectl version --client
```

## Krew, plugin manager for Kubernetes

```
(
set -x; cd "$(mktemp -d)" &&
OS="$(uname | tr '[:upper:]' '[:lower:]')" &&
ARCH="$(uname -m | sed -e 's/x86_64/amd64/' -e 's/^(arm\)(64)\?.*$/\1\2/' -e 's/aarch64$/arm64/')"
KREW="krew-$OS_${ARCH}" &&
curl -fsSLO "https://github.com/kubernetes-sigs/krew/releases/latest/download/${KREW}.tar.gz" &&
tar zxvf "${KREW}.tar.gz" &&
./"${KREW}" install krew
)

cat << EOF >> ~/.bashrc
export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"
EOF
```

## Kubectx + Kubens

```
kubectl krew install ctx
kubectl krew install ns
```

**Konfig** Le plugin konfig va nous servir à exporter une configuration cluster présente dans un fichier de structure kubeconfig et la copier dans `~/kube/config`. De cette façon, les binaires kubectx ou kubie auront accès au contexte et nous pourrons ainsi passer d'un contexte à l'autre sans jamais passer la variable d'environnement **KUBECONFIG**.

```
kubectl krew install konfig
konfig import /tmp/custom-cluster-KuBeConFig
```

```
konfig import --save /tmp/custom-cluster-KuBeConFig
kubie ctx custom-cluster
kubectl get po -A
```

**Kubie** Réunit les commandes `kubectx` + `kubens` dans un seul binaire (et peut permettre par exemple d'exécuter des commandes extérieures au contexte kube actuel).

```
curl -sS \
https://github.com/sbstp/kubie/releases/download/v0.16.0/kubie-linux-amd64 \
| bash
```

```
chmod +x kubie-linux-amd64
mv kubie-linux-amd64 /usr/local/bin/kubie
```

Quelques exemples concernant l'utilisation de Kubie:

Changer de contexte courant:

```
kubie ctx kolo@kolorado.us-east-2.eksctl.io
```

Afficher les namespaces du contexte courant:

```
kubie ns
```

Exécuter une commande kubectl sur un contexte accessible:

```
kubie exec kolo@kolorado.us-east-2.eksctl.io kube-system kubectl get pods
```

**K9s** Visualiseur en shell console pour Kubernetes:

```
curl -sS https://webinstall.dev/k9s | bash

echo "export PATH='/home/pbackz/.local/bin:$PATH'" >> ~/.bashrc

# List all available CLI options
k9s help
# To get info about K9s runtime (logs, configs, etc..)
k9s info
# To run K9s in a given namespace
k9s -n mycoolns
# Start K9s in an existing KubeConfig context
k9s --context coolCtx
# Start K9s in readonly mode - with all cluster modification commands disabled
k9s --readonly
```

**GUI** Kubernetes fournit nativement des dashboards.

Cependant, l'interface utilisateur du tableau de bord n'est pas déployée par défaut. Pour la déployer, exécutez la commande suivante:

```
kubectl apply -f \
https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended.yaml
```

Pour protéger vos données dans le cluster, le tableau de bord se déploie avec une configuration RBAC minimale par défaut. Actuellement, le tableau de bord prend uniquement en charge la connexion avec un token de support.

Vous pouvez accéder au tableau de bord à l'aide de la commande suivante:

```
kubectl proxy
```

Kubectl mettra le tableau de bord à disposition à l'adresse suivante: <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>

Vous ne pouvez accéder à l'interface utilisateur que depuis la machine sur laquelle la commande est exécutée. Voir kubectl `proxy --help` pour plus d'options.

*N.B.* La méthode d'authentification Kubeconfig ne prend pas en charge les fournisseurs d'identité externes ni l'authentification basée sur un certificat x509.

Nous y préférerons aujourd'hui l'outil Kubernetes Lens [k8slens.dev](https://k8slens.dev), disposant de plus de fonctionnalités.

**APIs** De nombreux sdks conçus pour la majorité des langages permettent également d'accéder à un cluster Kubernetes, d'en visualiser, lister ou manipuler les ressources via des apis.

Toutefois, nous ne saurions que trop conseiller l'utilisation du langage Golang dans lequel Kubernetes est écrit pour manipuler ces APIs ou créer des opérateurs.

## Déploiement, publication manuelle et introspection du déploiement

**Un Deployment fournit des mises à jour déclaratives pour Pods et ReplicaSets. Vous décrivez un état désiré dans un déploiement et le contrôleur déploiement change l'état réel à l'état souhaité à un rythme contrôlé. Vous pouvez définir des Deployments pour créer de nouveaux ReplicaSets, ou pour supprimer des déploiements existants et adopter toutes leurs ressources avec de nouveaux déploiements.**

Voici un exemple de déploiement. Il crée un ReplicaSet pour faire apparaître trois pods nginx:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
```

```

labels:
  app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.7.9
      ports:
        - containerPort: 80

```

Dans cet exemple:

Un déploiement nommé nginx-deployment est créé, indiqué par le champ **.meta-data.name**.

Le déploiement crée trois pods répliqués, indiqués par le champ replicas.

Le champ selector définit comment le déploiement trouve les pods à gérer. Dans ce cas, vous sélectionnez simplement un label défini dans le template de pod (app:nginx). Cependant, des règles de sélection plus sophistiquées sont possibles, tant que le modèle de pod satisfait lui-même la règle.

*N.B. Le champ matchLabels est une table de hash [clé, valeur]. Une seule {clé, valeur} dans la table matchLabels est équivalente à un élément de matchExpressions, dont le champ clé est "clé", l'opérateur est "In" et le tableau de valeurs contient uniquement "valeur". Toutes les exigences, à la fois de matchLabels et de matchExpressions, doivent être satisfaites pour correspondre.*

Les Pods reçoivent le label **app:nginx** dans le champ labels.

La spécification du template de pod dans le champ **.template.spec**, indique que les pods exécutent un conteneur, nginx, qui utilise l'image nginx Docker Hub à la version 1.7.9.

Créez un conteneur et nommez-le nginx en utilisant le champ **name**.

Créez le déploiement en exécutant la commande suivante:

*N.B. Vous pouvez spécifier l'indicateur **-record** pour écrire la commande exécutée dans l'annotation de ressource kubernetes.io/change-cause.*

C'est utile pour une future introspection. Par exemple, pour voir les commandes exécutées dans chaque révision de déploiement.

```
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
```

Exécutez **kubectl get deployments** pour vérifier si le déploiement a été créé.

Si le déploiement est toujours en cours de création, la sortie est similaire à:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	0/3	0	0	1s

Lorsque vous inspectez les déploiements de votre cluster, les champs suivants s'affichent:

- **NAME** - répertorie les noms des déploiements dans le cluster.
- **DESIRED** - affiche le nombre souhaité de répliques de l'application, que vous définissez lorsque vous créez le déploiement. C'est l'état désiré.

- **CURRENT** - affiche le nombre de réplicas en cours d'exécution.
- **UP-TO-DATE** - affiche le nombre de réplicas qui ont été mises à jour pour atteindre l'état souhaité.
- **AVAILABLE** - affiche le nombre de réplicas de l'application disponibles pour vos utilisateurs.
- **AGE** - affiche la durée d'exécution de l'application.

Notez que le nombre de réplicas souhaitées est de 3 selon le champ **.spec.replicas**

Pour voir l'état du déploiement, exécutez:

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

Exécutez à nouveau kubectl get deployments quelques secondes plus tard. La sortie est similaire à ceci:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	18s

Notez que le déploiement a créé les trois répliques et que toutes les répliques sont à jour (elles contiennent le dernier modèle de pod) et disponibles.

Pour voir le ReplicaSet (**rs**) créé par le déploiement, exécutez `kubectl get rs`.

La sortie est similaire à ceci:

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-75675f5897	3	3	3	18s

Notez que le nom du ReplicaSet est toujours formaté comme ceci: “[**DEPLOYMENT-NAME**]-[**RANDOM-STRING**]”

La chaîne aléatoire est générée aléatoirement et utilise le pod-template-hash.

Pour voir les labels générées automatiquement pour chaque Pod, exécutez:

```
kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
nginx-deployment-75675f5897-7ci7o	1/1	Running	0	18s	app=nginx, pod-template-hash=31231
nginx-deployment-75675f5897-kzszej	1/1	Running	0	18s	app=nginx, pod-template-hash=31231
nginx-deployment-75675f5897-qqcnn	1/1	Running	0	18s	app=nginx, pod-template-hash=31231

Le ReplicaSet créé garantit qu'il y a trois pods nginx.

*N.B. Vous devez spécifier un sélecteur approprié et des labels de template de pod dans un déploiement (dans ce cas, app: nginx). Ne superposez pas les étiquettes ou les sélecteurs avec d'autres contrôleurs (y compris d'autres déploiements et StatefulSets). Kubernetes n'empêche pas les chevauchements de noms, et si plusieurs contrôleurs ont des sélecteurs qui se chevauchent, ces contrôleurs peuvent entrer en conflit et se comporter de façon inattendue.*

## Déploiement d'une plateforme de test (TP)

Dans ce TP vous devrez mettre en place une plateforme de test locale via l'une des solutions suivantes: Minikube, Kind, MicroK8S ou encore Docker Desktop.

Vous créerez des pods disposant d'un label 'test'.

Vous devrez pouvoir lister les pods ainsi que leurs labels et accéder au fichier de configuration du cluster (kubeconfig).

## Les fichiers descriptifs

### Syntaxe YAML

Le YAML, qu'est-ce que c'est ?

**YAML, acronyme de Yet Another Markup Language dans sa version 1.01, devient l'acronyme récursif de YAML Ain't Markup Language (« YAML n'est pas un langage de balisage ») dans sa version 1.12, est un format de représentation de données par sérialisation Unicode. Il reprend des concepts d'autres langages comme XML, ou encore du format de message électronique tel que documenté par RFC 2822. YAML a été proposé par Clark Evans en 20013, et implémenté par ses soins ainsi que par Brian Ingerson et Oren Ben-Kiki. Son objet est de représenter des informations plus élaborées que le simple CSV en gardant cependant une lisibilité presque comparable, et bien plus grande en tout cas que du XML.**

Source [YAML definition](#)

L'essentiel de la syntaxe YAML:

```
-- # document start

# Comments in YAML look like this.

#####
# SCALAR TYPES #
#####

# Our root object (which continues for the entire document) will be a map,
# which is equivalent to a dictionary, hash or object in other languages.
key: value
another_key: Another value goes here.
a_number_value: 100
scientific_notation: 1e+12
# The number 1 will be interpreted as a number, not a boolean. if you want
# it to be interpreted as a boolean, use true
boolean: true
null_value: null
key with spaces: value
# Notice that strings don't need to be quoted. However, they can be.
however: 'A string, enclosed in quotes.'
'Keys can be quoted too.': "Useful if you want to put a ':' in your key."
```

```

single quotes: 'have ''one'' escape pattern'
double quotes: "have many: \", \0, \t, \u263A, \x0d\x0a == \r\n, and more."
# UTF-8/16/32 characters need to be encoded
Superscript two: \u00B2

# Multiple-line strings can be written either as a 'literal block' (using /),
# or a 'folded block' (using '>').
literal_block: |
    This entire block of text will be the value of the 'literal_block' key,
    with line breaks being preserved.

The literal continues until de-dented, and the leading indentation is
stripped.

Any lines that are 'more-indented' keep the rest of their indentation -
these lines will be indented by 4 spaces.

folded_style: >
    This entire block of text will be the value of 'folded_style', but this
    time, all newlines will be replaced with a single space.

Blank lines, like above, are converted to a newline character.

'More-indented' lines keep their newlines, too -
this text will appear over two lines.

#####
# COLLECTION TYPES #
#####

# Nesting uses indentation. 2 space indent is preferred (but not required).

a_nested_map:
    key: value
    another_key: Another Value
    another_nested_map:
        hello: hello

# Maps don't have to have string keys.
0.25: a float key

# Keys can also be complex, like multi-line objects
# We use ? followed by a space to indicate the start of a complex key.
? |
    This is a key
    that has multiple lines
    : and this is its value

# YAML also allows mapping between sequences with the complex key syntax
# Some language parsers might complain
# An example
? - Manchester United
  - Real Madrid

```

```

: [2001-01-01, 2002-02-02]

# Sequences (equivalent to lists or arrays) look like this
# (note that the '-' counts as indentation):
a_sequence:
  - Item 1
  - Item 2
  - 0.5 # sequences can contain disparate types.
  - Item 4
  - key: value
    another_key: another_value
  -
    - This is a sequence
    - inside another sequence
    - -- Nested sequence indicators
      - can be collapsed

# Since YAML is a superset of JSON, you can also write JSON-style maps and
# sequences:
json_map: {"key": "value"}
json_seq: [3, 2, 1, "takeoff"]
and quotes are optional: {key: [3, 2, 1, takeoff]}

#####
# EXTRA YAML FEATURES #
#####

# YAML also has a handy feature called 'anchors', which let you easily duplicate
# content across your document. Both of these keys will have the same value:
anchored_content: &anchor_name This string will appear as the value of two keys.
other_anchor: *anchor_name

# Anchors can be used to duplicate/inherit properties
base: &base
  name: Everyone has same name

# The regexp << is called Merge Key Language-Independent Type. It is used to
# indicate that all the keys of one or more specified maps should be inserted
# into the current map.

foo:
  <<: *base
  age: 10

bar:
  <<: *base
  age: 20

# foo and bar would also have name: Everyone has same name

# YAML also has tags, which you can use to explicitly declare types.

```

```

explicit_string: !!str 0.5
# Some parsers implement language specific tags, like this one for Python's
# complex number type.
python_complex_number: !!python/complex 1+2j

# We can also use yaml complex keys with language specific tags
? !!python/tuple [5, 7]
: Fifty Seven
# Would be {(5, 7): 'Fifty Seven'} in Python

#####
# EXTRA YAML TYPES #
#####

# Strings and numbers aren't the only scalars that YAML can understand.
# ISO-formatted date and datetime literals are also parsed.
datetime: 2001-12-15T02:59:43.1Z
datetime_with_spaces: 2001-12-14 21:59:43.10 -5
date: 2002-12-14

# The !!binary tag indicates that a string is actually a base64-encoded
# representation of a binary blob.
gif_file: !!binary |
R0lGODlhDAAMAIQAAAP//9/X17unp5WZmZgAAAOfn515eXvPz7Y60juDg4J+fn5
OTk6enp56enmlpaWNjY60jo4SEhP/++f/++f/++f/++f/++f/++f/++f/+
+f/++f/++f/++f/++f/++SH+Dk1hZGUgd210aCBHSU1QACwAAAAADAAMAAFLC
AgjoEwnuNAFOhpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcggoBADs=

# YAML also has a set type, which looks like this:
set:
? item1
? item2
? item3
or: {item1, item2, item3}

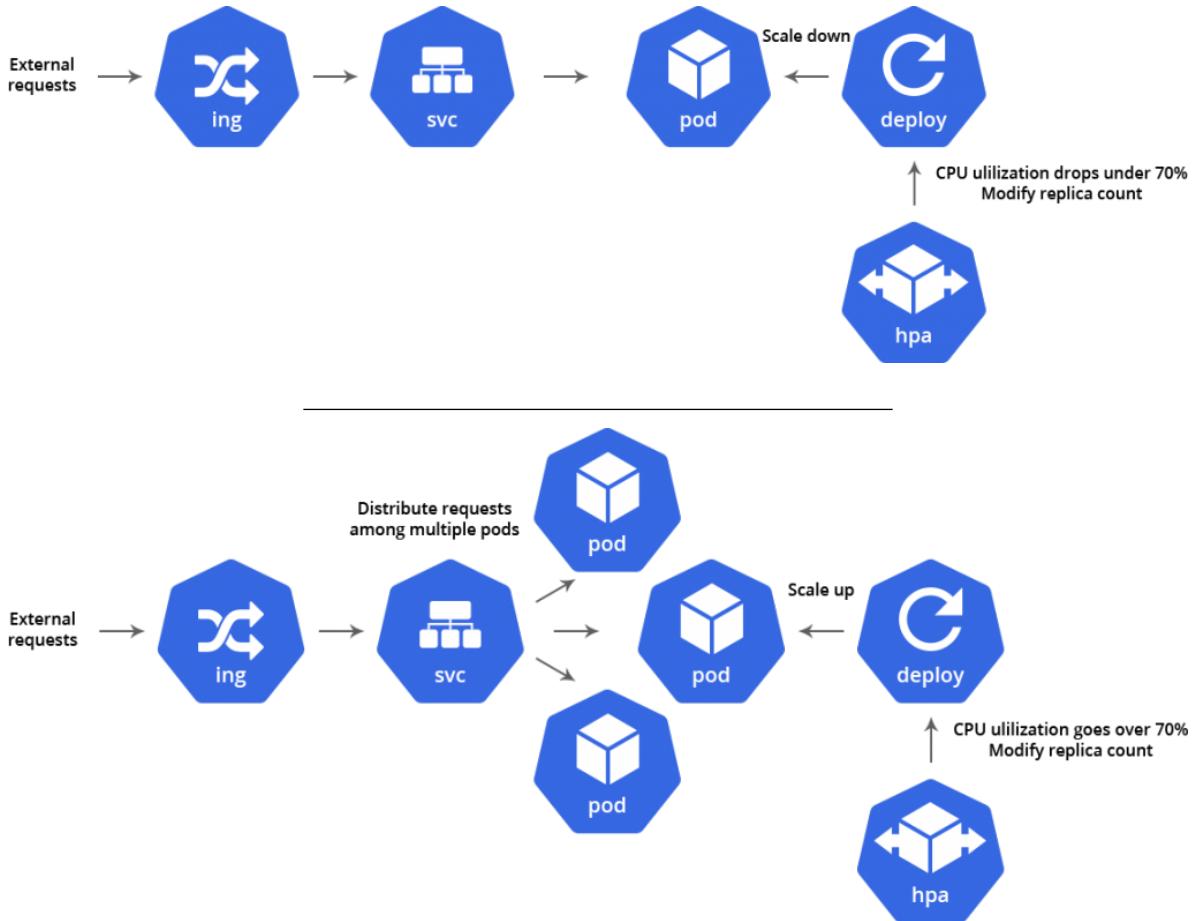
# Sets are just maps with null values; the above is equivalent to:
set2:
item1: null
item2: null
item3: null

... # document end

```

*Source* [Learn YAML in 10 minutes](#)

## Scalabilité d'un déploiement



Vous pouvez mettre à l'échelle un déploiement en utilisant la commande suivante:

```
kubectl scale deployment/nginx-deployment --replicas=10
```

```
deployment.apps/nginx-deployment scaled
```

En supposant que l'autoscaling horizontal des pods est activé dans votre cluster, **vous pouvez configurer un autoscaler pour votre déploiement** et choisir le nombre minimum et maximum de pods que vous souhaitez exécuter en fonction de l'utilisation du CPU de vos pods existants.

```
kubectl autoscale deployment/nginx-deployment --min=10 --max=15 --cpu-percent=80
```

```
deployment.apps/nginx-deployment scaled
```

Les déploiements RollingUpdate permettent d'exécuter plusieurs versions d'une application en même temps. Lorsque vous ou un autoscaler mettez à l'échelle un déploiement RollingUpdate qui est au milieu d'un déploiement (en cours ou en pause), le contrôleur de déploiement équilibre les replicas supplémentaires dans les ReplicaSets actifs existants (ReplicaSets avec Pods) afin d'atténuer le risque.

C'est ce qu'on appelle la mise à l'échelle proportionnelle.

Par exemple, vous exécutez un déploiement avec 10 replicas, maxSurge=3 et maxUnavailable=2.

Assurez-vous que les 10 replicas de votre déploiement sont running.

```
kubectl get deploy
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	10	10	10	10	50s

Vous effectuez une mise à jour vers une nouvelle image qui se trouve être non résoluble depuis l'intérieur du cluster.

```
kubectl set image deployment/nginx-deployment nginx=nginx:sometag
```

```
deployment.apps/nginx-deployment image updated
```

La mise à jour de l'image lance un nouveau déploiement avec ReplicaSet nginx-deployment-1989198191, mais il est bloqué en raison de l'exigence maxUnavailable que vous avez mentionnée ci-dessus.

Vérifiez l'état du déploiement.

```
kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1989198191	5	5	0	9s
nginx-deployment-618515232	8	8	8	1m

Puis une nouvelle demande de mise à l'échelle pour que le déploiement arrive. L'autoscaler incrémente les replicas du déploiement à 15. Le contrôleur de déploiement doit décider où ajouter ces 5 nouveaux replicas.

Si vous n'utilisiez pas la mise à l'échelle proportionnelle, les 5 replicas seraient ajoutés dans le nouveau ReplicaSet. Avec la mise à l'échelle proportionnelle, vous répartissez les replicas supplémentaires sur tous les ReplicaSets. Les proportions les plus importantes vont aux ReplicaSets ayant le plus de replicas et les proportions les plus faibles vont aux ReplicaSets ayant moins de replicas. Les restes sont ajoutés au ReplicaSet ayant le plus de replicas.

Les ReplicaSets avec zéro replica ne sont pas mis à l'échelle.

Dans notre exemple ci-dessus, 3 replicas sont ajoutées à l'ancien ReplicaSet et 2 replicas sont ajoutées au nouveau ReplicaSet. Le processus de déploiement devrait finalement déplacer toutes les replicas vers le nouveau ReplicaSet, en supposant que les nouveaux replicas soient valides.

Pour confirmer cela, exécutez:

```
kubectl get deploy
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	15	18	7	8	7m

Le statut de déploiement confirme comment les replicas ont été ajoutés à chaque ReplicaSet.

```
kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1989198191	7	7	0	7m
nginx-deployment-618515232	11	11	11	7m

## Stratégie de mise à jour sans interruption

*N.B.* Le re-déploiement d'un déploiement est déclenché si et seulement si le modèle de pod du déploiement (c'est-à-dire **.spec.template**) est modifié, par exemple si les labels ou les images de conteneur du template sont mis à jour. D'autres mises à jour, telles que la mise à l'échelle du déploiement, ne déclenchent pas de rollout.

Suivez les étapes ci-dessous pour mettre à jour votre déploiement:

Mettons à jour les pods nginx pour utiliser l'image nginx: 1.9.1 au lieu de l'image nginx: 1.7.9.

```
kubectl -record deployment.apps/nginx-deployment set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1
```

ou utilisez la commande suivante:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1 --record  
deployment.apps/nginx-deployment image updated
```

Alternativement, vous pouvez éditer le déploiement et changer **.spec.template.spec.containers[0].image** de 'nginx: 1.7.9' à 'nginx: 1.9.1':

```
kubectl edit deployment.v1.apps/nginx-deployment
```

```
deployment.apps/nginx-deployment edited
```

Pour voir l'état du déploiement, exécutez:

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

# OR

```
deployment "nginx-deployment" successfully rolled out
```

Obtenez plus de détails sur votre déploiement mis à jour:

Une fois le déploiement réussi, vous pouvez afficher le déploiement en exécutant **kubectl get deployments**. La sortie est similaire à ceci:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	36s

Exécutez kubectl get rs pour voir que le déploiement a mis à jour les pods en créant un nouveau ReplicaSet et en le redimensionnant jusqu'à 3 replicas, ainsi qu'en réduisant l'ancien ReplicaSet à 0 replicas.

```
kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1564180365	3	3	3	6s
nginx-deployment-2035384211	0	0	0	36s

L'exécution de `kubectl get pods` ne devrait désormais afficher que les nouveaux pods:

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1564180365-khku8	1/1	Running	0	14s
nginx-deployment-1564180365-nacti	1/1	Running	0	14s
nginx-deployment-1564180365-z9gth	1/1	Running	0	14s

La prochaine fois que vous souhaitez mettre à jour ces pods, il vous suffit de mettre à jour le modèle de pod de déploiement à nouveau.

**Le déploiement garantit que seul un certain nombre de pods sont en panne pendant leur mise à jour. Par défaut, il garantit qu'au moins 75% du nombre souhaité de pods sont en place (25% max indisponible).**

Le déploiement garantit également que seul un certain nombre de pods sont créés au-dessus du nombre souhaité de pods. **Par défaut, il garantit qu'au plus 125% du nombre de pods souhaité sont en hausse (surtension maximale de 25%).**

Par exemple, si vous regardez attentivement le déploiement ci-dessus, vous verrez qu'il a d'abord créé un nouveau pod, puis supprimé certains anciens pods et en a créé de nouveaux. Il ne tue pas les anciens Pods tant qu'un nombre suffisant de nouveaux Pods n'est pas apparu, et ne crée pas de nouveaux Pods tant qu'un nombre suffisant de Pods anciens n'a pas été tué. Il s'assure qu'au moins 2 pods sont disponibles et qu'au maximum 4 pods au total sont disponibles.

Obtenez les détails de votre déploiement:

```
kubectl describe deployments
```

```
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Thu, 30 Nov 2021 10:56:25 +0000
Labels:          app=nginx
Annotations:    deployment.kubernetes.io/revision=2
Selector:        app=nginx
Replicas:       3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
    nginx:
      Image:      nginx:1.9.1
      Port:       80/TCP
      Environment: <none>
      Mounts:     <none>
```

```

Volumes:           <none>
Conditions:
Type          Status  Reason
----          -----  -----
Available      True    MinimumReplicasAvailable
Progressing    True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  nginx-deployment-1564180365 (3/3 replicas created)
Events:
Type  Reason          Age   From            Message
----  -----          ----  ---            -----
Normal ScalingReplicaSet 2m    deployment-controller  Scaled up replica set nginx-deployment-2035384211
Normal ScalingReplicaSet 24s   deployment-controller  Scaled up replica set nginx-deployment-1564180365
Normal ScalingReplicaSet 22s   deployment-controller  Scaled down replica set nginx-deployment-2035384211
Normal ScalingReplicaSet 22s   deployment-controller  Scaled up replica set nginx-deployment-1564180365
Normal ScalingReplicaSet 19s   deployment-controller  Scaled down replica set nginx-deployment-2035384211
Normal ScalingReplicaSet 19s   deployment-controller  Scaled up replica set nginx-deployment-1564180365
Normal ScalingReplicaSet 14s   deployment-controller  Scaled down replica set nginx-deployment-2035384211

```

Ici, vous voyez que lorsque vous avez créé le déploiement pour la première fois, il a créé un ReplicaSet (nginx-deployment-2035384211) et l'a mis à l'échelle directement jusqu'à 3 replicas.

Lorsque vous avez mis à jour le déploiement, il a créé un nouveau ReplicaSet (nginx-deployment-1564180365) et l'a mis à l'échelle jusqu'à 1, puis a réduit l'ancien ReplicaSet à 2, de sorte qu'au moins 2 pods étaient disponibles et au plus 4 pods ont été créés à chaque fois.

Il a ensuite poursuivi la montée en charge du nouveau et de l'ancien ReplicaSet, avec la même stratégie de mise à jour continue.

Enfin, vous aurez 3 replicas disponibles dans le nouveau ReplicaSet, et l'ancien ReplicaSet est réduit à 0.

## Revenir à une révision précédente

Suivez les étapes ci-dessous pour restaurer le déploiement de la version actuelle à la version précédente, qui est la version 2.

Vous avez maintenant décidé d'annuler le déploiement actuel et le retour à la révision précédente:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

```
deployment.apps/nginx-deployment
```

Alternativement, vous pouvez revenir à une révision spécifique en la spécifiant avec **--to-revision**:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment \
--to-revision=2
```

```
deployment.apps/nginx-deployment
```

Pour plus de détails sur les commandes liées au déploiement, lisez `kubectl rollout`

Le déploiement est maintenant rétabli à une précédente révision stable. Comme vous pouvez le voir, **un événement DeploymentRollback pour revenir à la révision 2 est généré à partir du contrôleur de déploiement.**

Vérifiez si la restauration a réussi et que le déploiement s'exécute comme prévu, exécutez:

```
kubectl get deployment nginx-deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	30m

Obtenez la description du déploiement:

```
kubectl describe deployment nginx-deployment
```

```
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Sun, 02 Sep 2021 18:17:55 -0500
Labels:          app=nginx
Annotations:    deployment.kubernetes.io/revision=4
                 kubernetes.io/change-cause=kubectl set image deployment.v1.apps/nginx-deployment
Selector:        app=nginx
Replicas:       3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
    nginx:
      Image:      nginx:1.9.1
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type  Status  Reason
    ----  -----  -----
    Available  True   MinimumReplicasAvailable
    Progressing  True   NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet:  nginx-deployment-c4747d96c (3/3 replicas created)
Events:
  Type  Reason          Age   From            Message
  ----  -----          ----  ----            -----
  Normal ScalingReplicaSet  12m   deployment-controller  Scaled up replica set nginx-deployment-756
  Normal ScalingReplicaSet  11m   deployment-controller  Scaled up replica set nginx-deployment-c47
  Normal ScalingReplicaSet  11m   deployment-controller  Scaled down replica set nginx-deployment-756
  Normal ScalingReplicaSet  11m   deployment-controller  Scaled up replica set nginx-deployment-c47
```

Normal	ScalingReplicaSet	11m	deployment-controller	Scaled down replica set nginx-deployment-7
Normal	ScalingReplicaSet	11m	deployment-controller	Scaled up replica set nginx-deployment-c47
Normal	ScalingReplicaSet	11m	deployment-controller	Scaled down replica set nginx-deployment-7
Normal	ScalingReplicaSet	11m	deployment-controller	Scaled up replica set nginx-deployment-595
Normal	DeploymentRollback	15s	deployment-controller	Rolled back deployment " <a href="#">nginx-deployment</a> "
Normal	ScalingReplicaSet	15s	deployment-controller	Scaled down replica set nginx-deployment

## Suppression d'un déploiement

```
kubectl delete deploy deployment.apps/nginx-deployment
```

# or

```
kubectl delete deploy nginx-deployment
```

## Publication et analyse d'un déploiement (TP)

**Partie 1** Vous devrez publier un fichier deployment contenant la spécification de 3 replicas web servers nginx.

Vous devrez décrire ce deployment, afficher son état et vous assurez qu'il soit dans un état sain.

**Partie 2** Scaler le nombre de replicas à 10. Assurez-vous que l'état des replicas est sain.

Revenir ensuite à une révision précédente de votre spécification.

Détruirez ensuite ce deployment.

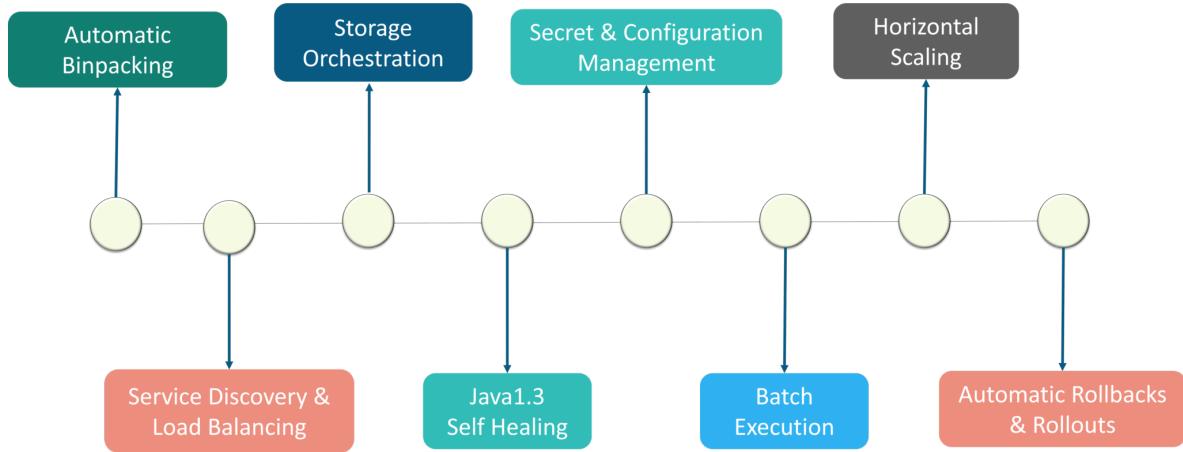
**Partie 3 (Facultatif)** Reproduisez l'étape 1 pour une base de données mysql en lieu et place des serveurs nginx. Ce deployment ne devra disposer que d'un seul replica.

La persistence des données n'est requise.

Détruirez ensuite ce deployment.

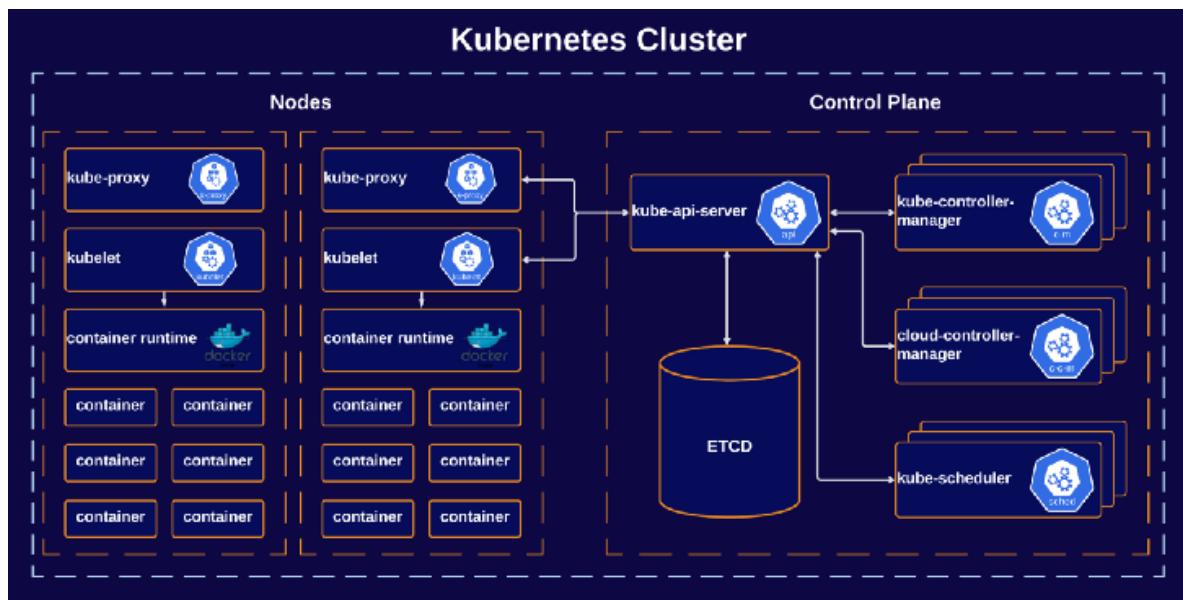
## Architecture Kubernetes

Fonctionnalités:

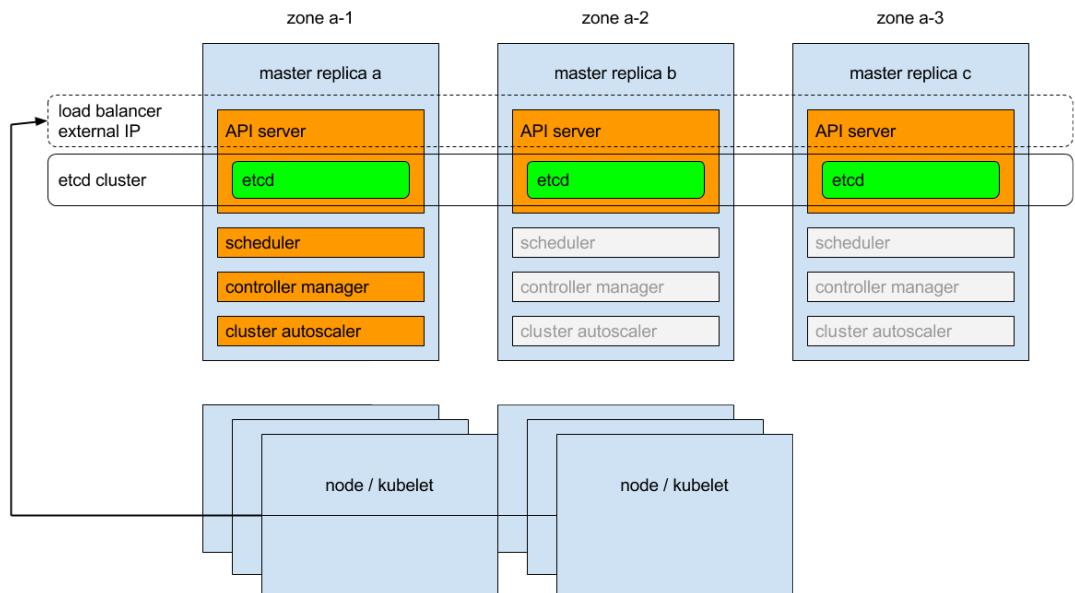
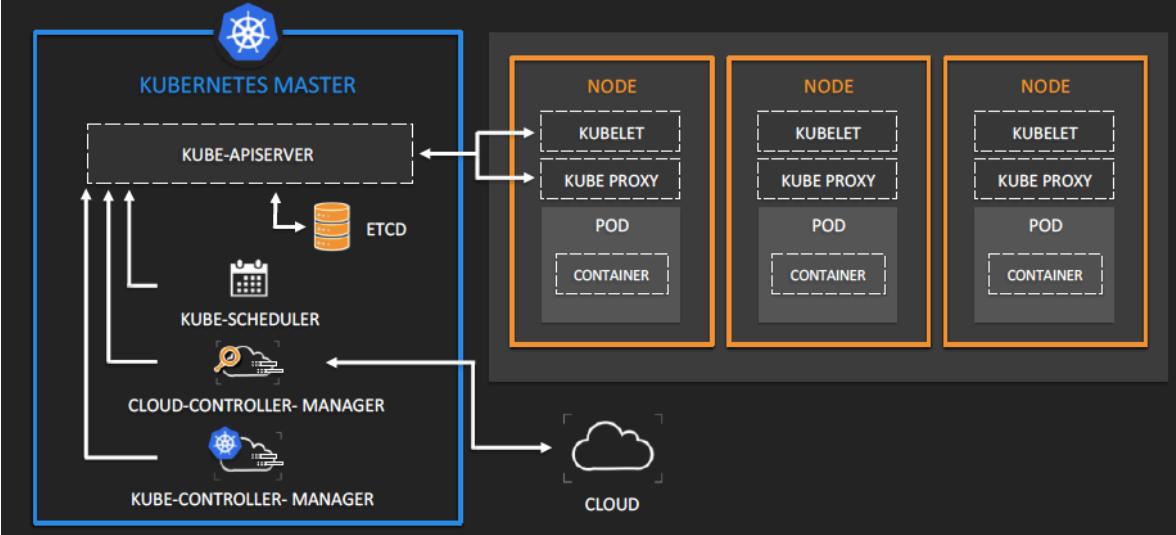


## Composants du Control Plane

*Vue macro*



## BASIC KUBERNETES ARCHITECTURE



Le master Kubernetes qui est un ensemble de trois processus qui s'exécutent sur un

seul node de votre cluster, désigné comme master node. Ces processus sont: kube-apiserver, kube-controller-manager et kube-scheduler. Chaque node non master de votre cluster exécute deux processus: - kubelet, qui communique avec le Kubernetes master. - kube-proxy, un proxy réseau reflétant les services réseau Kubernetes sur chaque node.

Les différentes parties du control plane Kubernetes, telles que les processus Kubernetes master et kubelet, déterminent la manière dont Kubernetes communique avec votre cluster. Le control plane conserve un enregistrement de tous les objets Kubernetes du système et exécute des boucles de contrôle continues pour gérer l'état de ces objets. À tout moment, les boucles de contrôle du control plane répondent aux modifications du cluster et permettent de faire en sorte que l'état réel de tous les objets du système corresponde à l'état souhaité que vous avez fourni.

Par exemple, lorsque vous utilisez l'API Kubernetes pour créer un objet Deployment, vous fournissez un nouvel état souhaité pour le système. Le control plane Kubernetes enregistre la création de cet objet et exécute vos instructions en lançant les applications requises et en les planifiant vers des nodes de cluster, afin que l'état actuel du cluster corresponde à l'état souhaité.

## Composants du master node

---

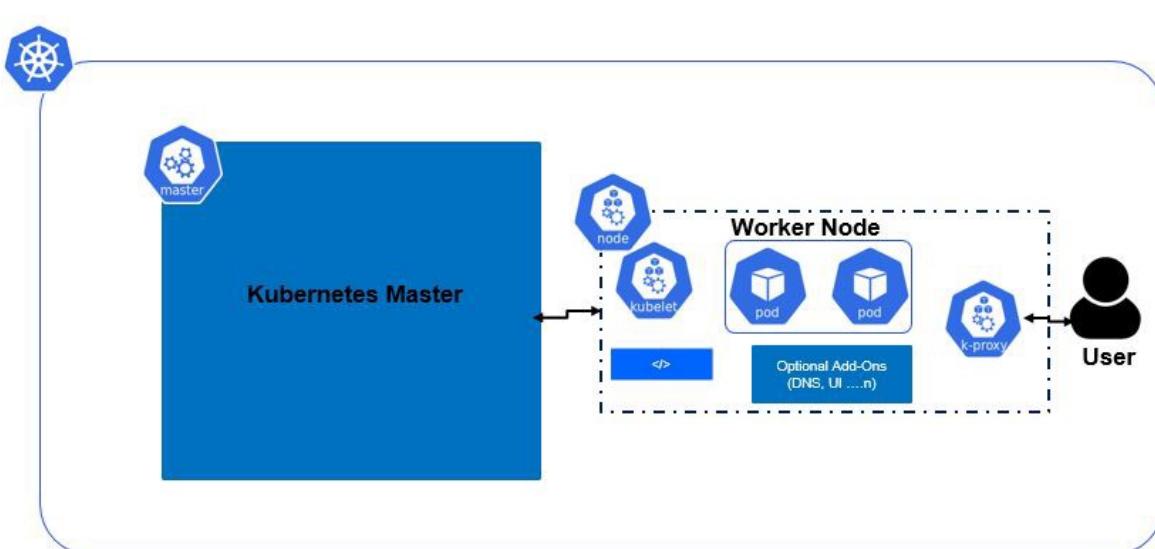
- cluster etcd > **Un stockage simple et distribué de valeurs de clés qui est utilisé pour stocker les données du cluster Kubernetes (telles que le nombre de pods, leur état, l'espace de noms, etc), les objets API et les détails de la découverte de services.** Pour des raisons de sécurité, il n'est accessible qu'à partir du serveur d'API. etc. etcd permet de notifier au cluster les changements de configuration à l'aide de surveillants. Les notifications sont des requêtes API sur chaque node du cluster etcd pour déclencher la mise à jour des informations dans le stockage du node.
- kube-apiserver > **Le serveur API Kubernetes est l'entité de gestion centrale qui reçoit toutes les demandes REST de modifications (aux pods, services, ensembles de réplication/contrôleurs et autres), servant de front-end au cluster.** C'est également le seul composant qui communique avec le cluster etcd, s'assurant que les données sont stockées dans etcd et sont en accord avec les détails des services des pods déployés.
- kube-controller-manager > **Exécute un certain nombre de processus de contrôle distincts en arrière-plan (par exemple, le contrôleur de réplication contrôle le nombre de replica dans un pod, le contrôleur de endpoints remplit les objets endpoints comme les services et les pods)** pour réguler l'état partagé du cluster et effectuer des tâches de routine. Lorsqu'un changement dans la configuration d'un service se produit (par exemple, le remplacement de l'image à partir de laquelle les pods sont exécutés, ou la modification des paramètres dans le fichier yaml de configuration), le contrôleur repère le changement et commence à travailler vers le nouvel état souhaité.
- cloud-controller-manager > **Le responsable de la gestion des processus du**

contrôleur qui dépendent du cloud provider sous-jacent (le cas échéant). Par exemple, lorsqu'un contrôleur doit vérifier si un node a été résilié ou configurer des routes, des équilibriseurs de charge ou des volumes dans l'infrastructure du cloud provider, tout cela est géré par le cloud provider.

- kube-scheduler > Aide à planifier les pods (un groupe de conteneurs co-localisés à l'intérieur desquels nos processus d'application sont exécutés) sur les différents nodes en fonction de l'utilisation des ressources. Il lit les exigences opérationnelles du service et le planifie sur le node le mieux adapté. Par exemple, si l'application a besoin de 1 Go de mémoire et de 2 cœurs CPU, les pods de cette application seront planifiés sur un node disposant au moins de ces ressources. Le planificateur s'exécute chaque fois qu'il est nécessaire de planifier des pods. Le planificateur doit connaître les ressources totales disponibles ainsi que les ressources allouées aux charges de travail existantes sur chaque node.

**Architecture d'un minion** Un node est une machine de travail dans Kubernetes, connue auparavant sous le nom de minion. Un node peut être une machine virtuelle ou une machine physique, selon le cluster.

Chaque node contient les services nécessaires à l'exécution de pods et est géré par les composants du master. **Les services présents sur un node incluent le container runtime, kubelet et kube-proxy.**



Pour effectuer l'auto-enregistrement des nodes:

Lorsque l'indicateur de kubelet **-register-node est à true** (valeur par défaut), le kubelet tente de s'enregistrer auprès du serveur d'API. C'est le modèle préféré, utilisé par la plupart des distributions Linux.

Pour l'auto-enregistrement (self-registration en anglais), le kubelet est lancé avec les options suivantes:

- `--kubeconfig` - Chemin d'accès aux informations d'identification pour s'authentifier auprès de l'apiserver.
- `--cloud-provider` - Comment lire les métadonnées d'un fournisseur de cloud sur lui-même.
- `--register-node` - Enregistrement automatique avec le serveur API.
- `--register-with-taints` - Enregistrez le noeud avec la liste donnée de marques (séparés par des virgules =:). Sans effet si register-node est à false.
- `--node-ip` - Adresse IP du noeud.
- `--node-labels` - Labels à ajouter lors de l'enregistrement du noeud dans le cluster (voir Restrictions des labels appliquées par le plugin NodeRestriction admission dans les versions 1.13+).
- `--node-status-update-frequency` - Spécifie la fréquence à laquelle kubelet publie le statut de nœud sur master.

Quand le mode autorisation de node et plugin NodeRestriction admission sont activés, les kubelets sont uniquement autorisés à créer / modifier leur propre ressource de node.

## Concepts: objets stateful, stateless

Un objet dit stateful est capable de maintenir l'état d'un processus ou d'une transaction.

StatefulSet est l'objet de l'API de charge de travail utilisé pour gérer des applications stateful.

Il gère le déploiement et la mise à l'échelle d'un ensemble de Pods, et fournit des garanties sur l'ordre et l'unicité de ces Pods.

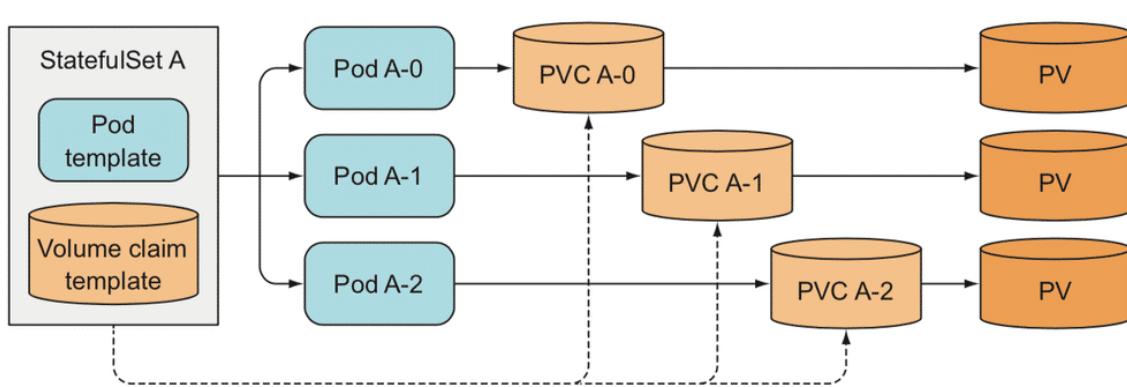
Comme un Déploiement, un StatefulSet gère des Pods qui sont basés sur une même spécification de conteneur. Contrairement à un Deployment, un StatefulSet maintient une identité pour chacun de ces Pods. Ces Pods sont créés à partir de la même spec, mais ne sont pas interchangeables: **chacun a un identifiant persistant qu'il garde à travers tous ses re-scheduling**.

Si vous voulez utiliser des volumes de stockage pour fournir de la persistance à votre charge de travail, vous pouvez utiliser un StatefulSet comme partie de la solution. Même si des Pods individuels d'un StatefulSet sont susceptibles d'échouer, les identifiants persistants des Pods rendent plus facile de faire correspondre les volumes existants aux nouveaux Pods remplaçant ceux ayant échoué.

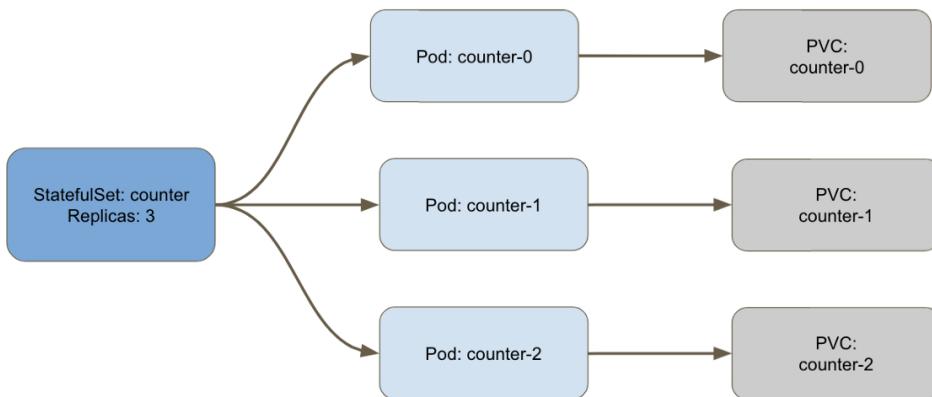
Ce qui caractérise la persistance de notre objet stateful reste toutefois lié à notre politique de storage. Un StatefulSet dont le stockage n'est pas persistant n'aura que peu de tolérance à la panne sur des applications critiques ou en hautes disponibilités. C'est dans cette perspective que nous introduirons ensuite les concepts de PersistentVolume et de PersistentVolumeClaim qui permettent de tirer parti de toutes les fonctionnalités du StatefulSet.

A l'inverse, les processus pour lesquels il n'est pas souhaité que l'état soit maintenu sont dits stateless. Il s'agit par défaut dans Kubernetes de tout objet qui ne soit pas un StatefulSet ou ne disposant pas d'un storage persistant.

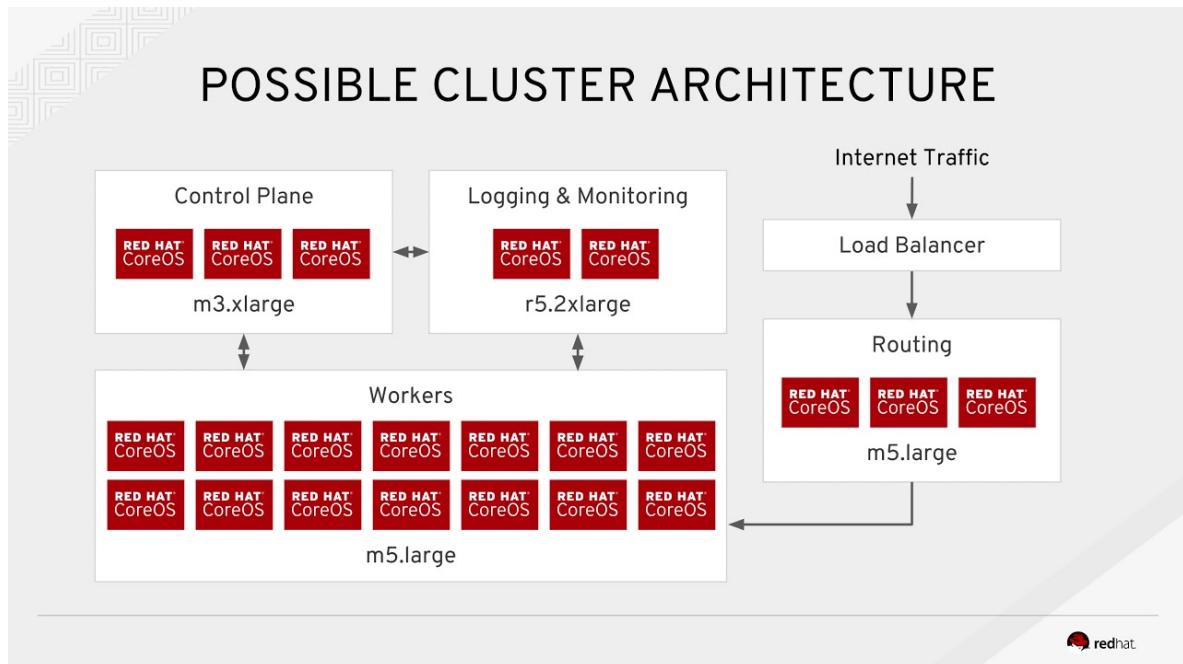
Ces concepts seront revus et pratiqués dans le cadre du TP du chapitre “Exploiter Kubernetes”.

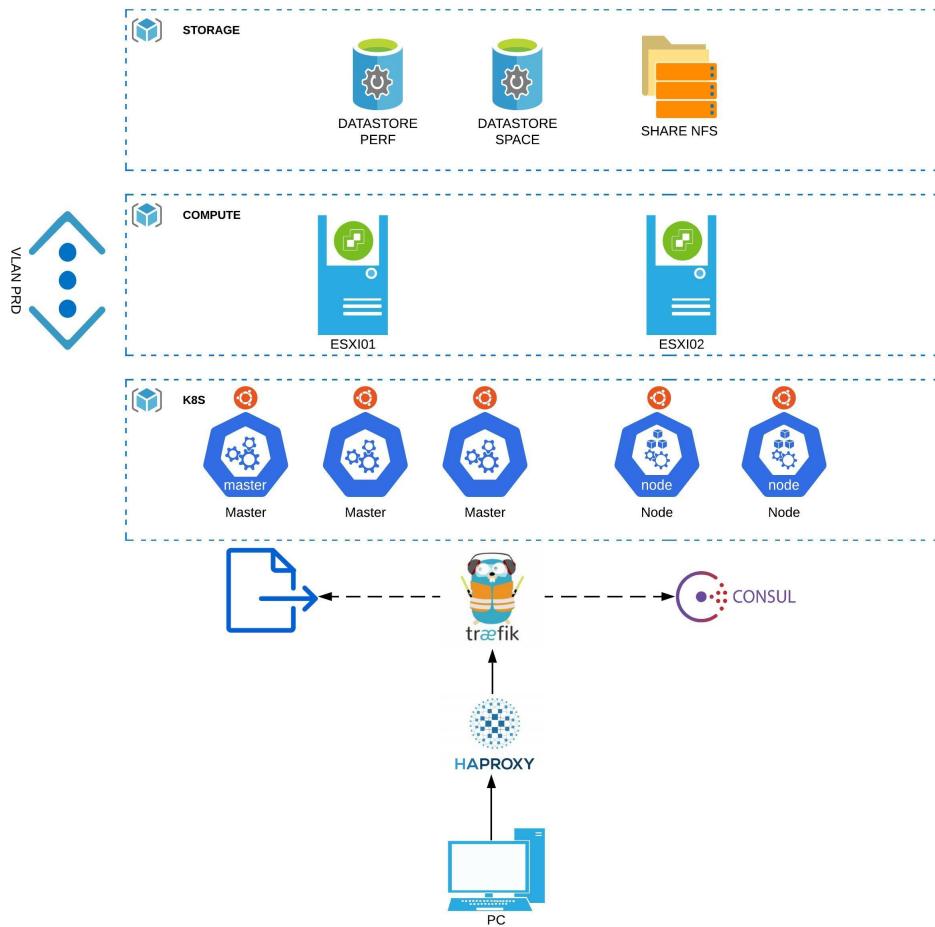


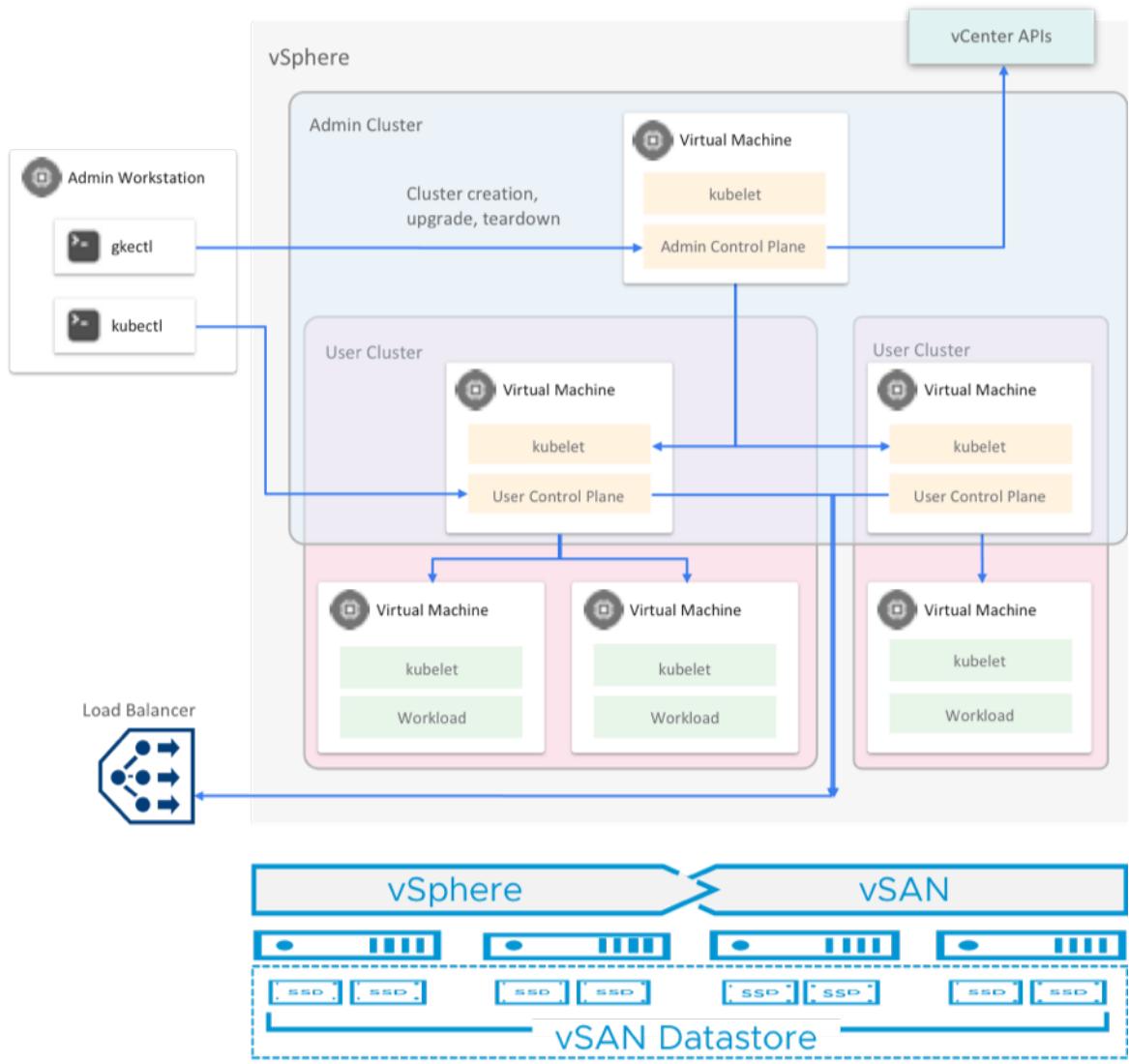
## Persistence in Statefulsets



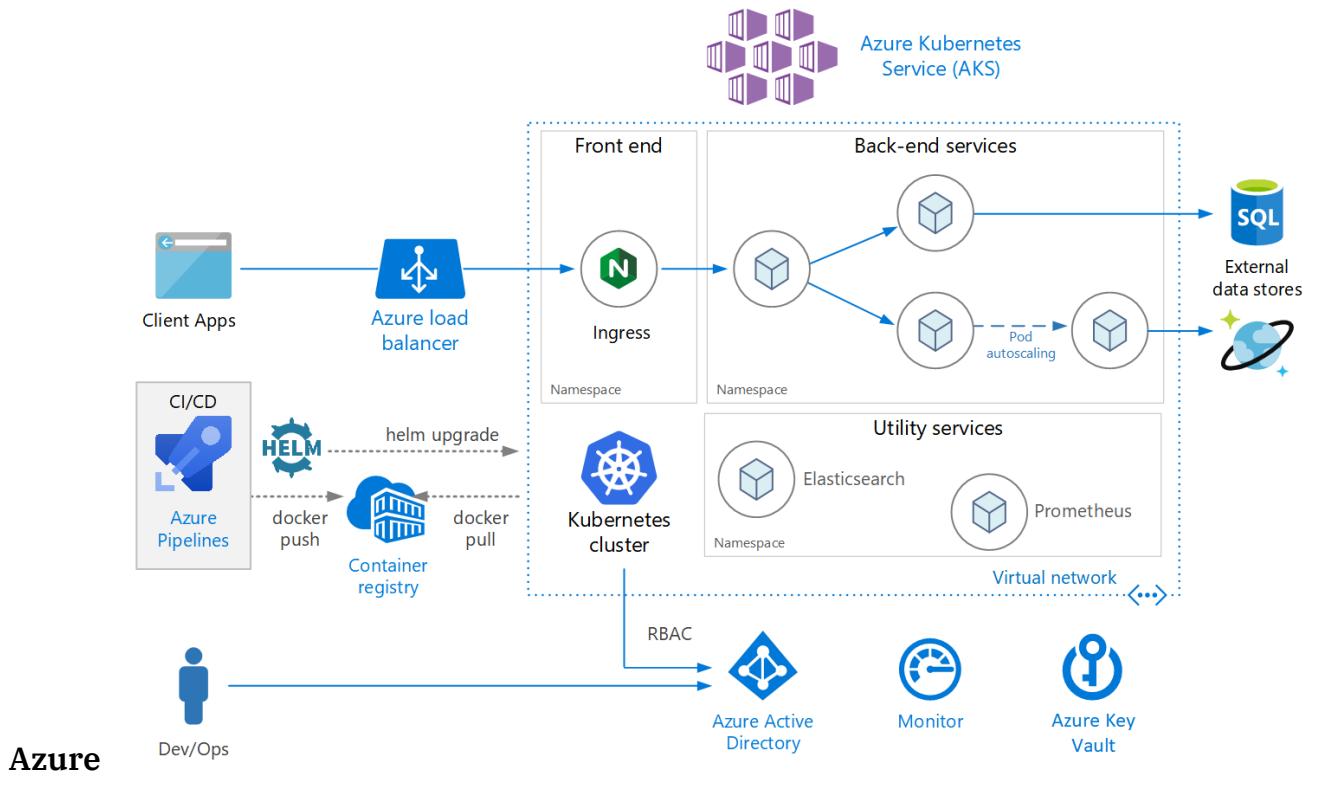
## Quelques exemples d'architecture cloud hybrides ou on-premise

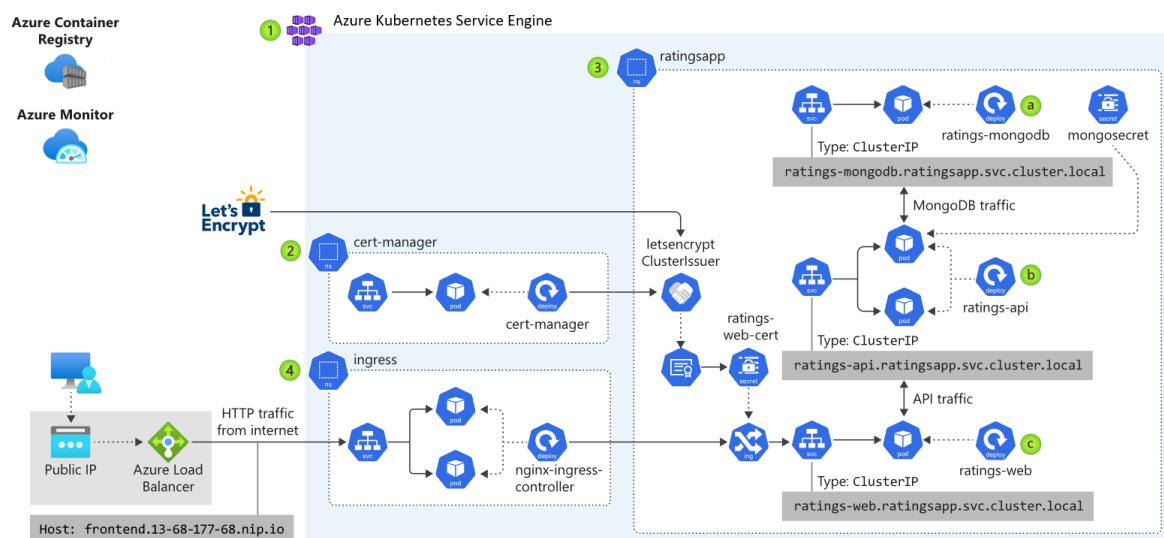
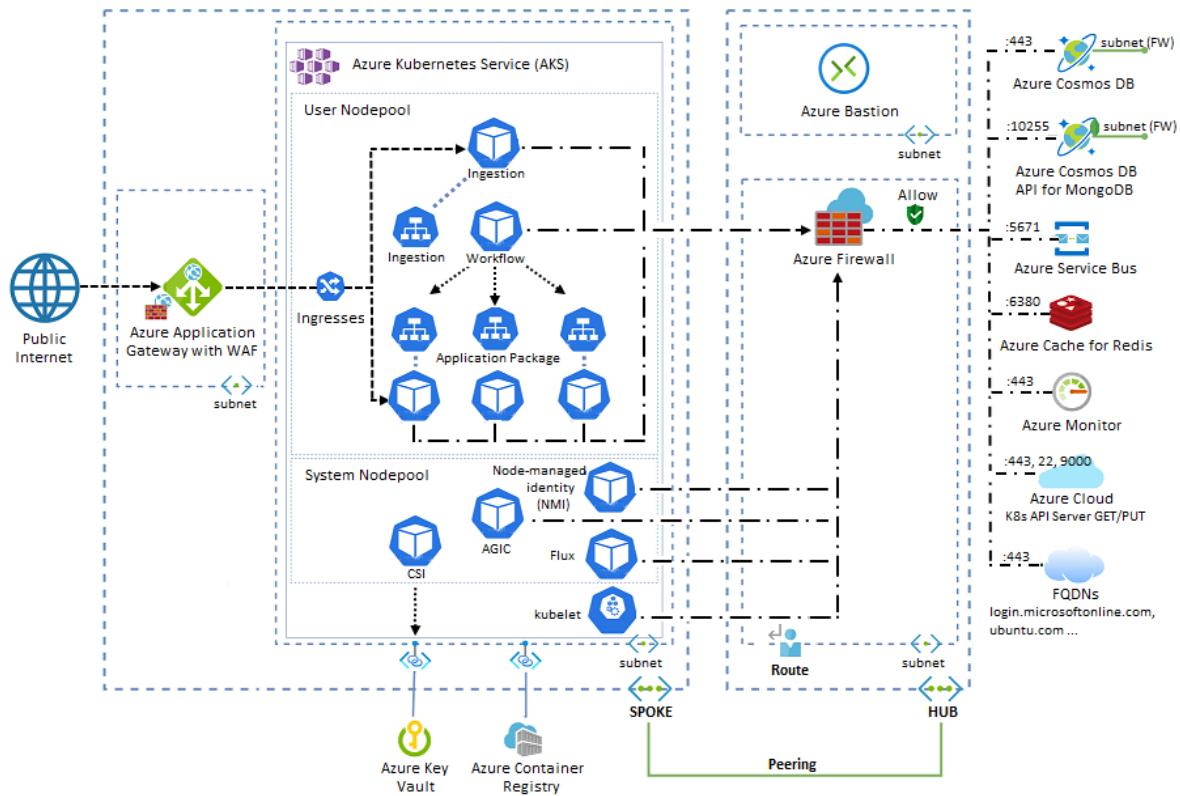




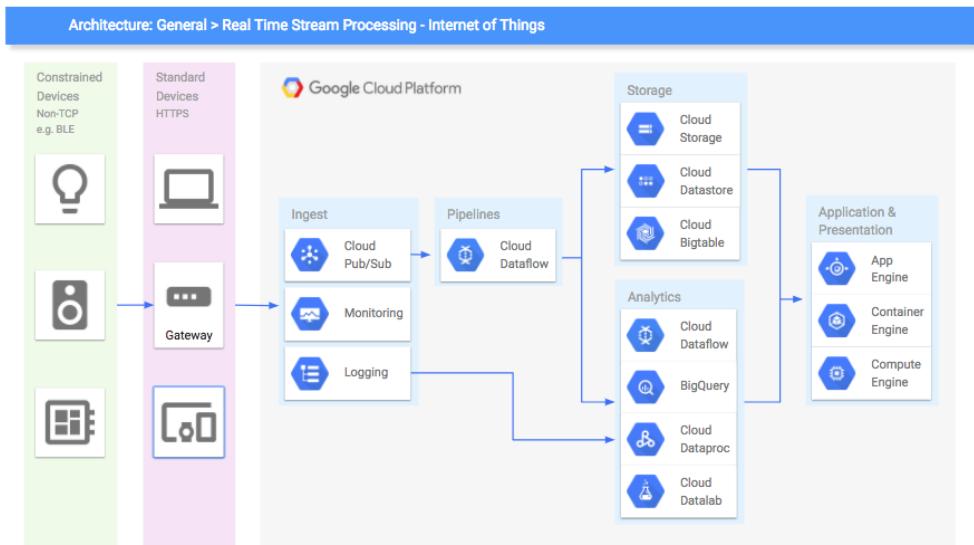


## Quelques exemples d'architecture cloud public

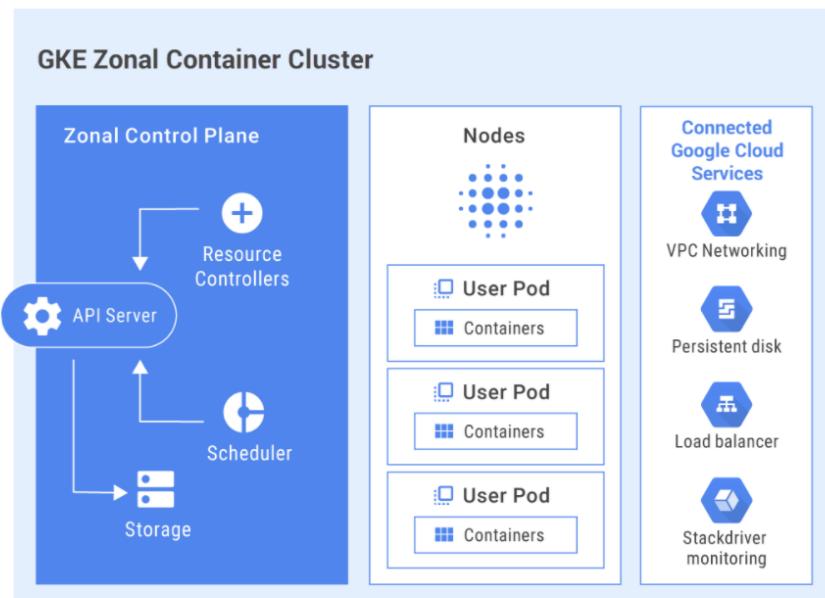




## General > Real Time Stream Processing IoT



GCP



Color Legend:

GKE provisions, maintains, and operates.

GKE provisions. User optionally maintains and operates.

# Exploiter Kubernetes

## Types de services

**Un service est une abstraction qui définit un ensemble logique de Pods exposé à travers un endpoint.**

Il existe différents types de services :

- ClusterIP expose l'IP interne du cluster.
- NodePort expose le Service sur l'IP de chaque nœud sur un port statique.
- LoadBalancer expose le Service à l'extérieur en utilisant l'équilibrEUR de charge d'un fournisseur de cloud computing.
- ExternalName associe le Service à un nom externe, tel que abc.toto.com.

Le service de type ClusterIP est le service par défaut de Kubernetes. Il donne un service à l'intérieur du cluster. Les pods à l'intérieur du même cluster peuvent utiliser ce service pour y accéder. Le service de type ClusterIP ne propose pas d'accès externe.

Spec pour un service de type **ClusterIP**:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: toto
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 8080
    protocol: TCP
```

Tous les pods à l'intérieur du cluster peuvent atteindre le pod “toto” sur leur port 8080 via `http://my-service:80`. Il est aussi possible d'atteindre le pod\_b en passant par l'adresse IP du clusterIP, mais ceci n'est pas conseillé. Si il y a plusieurs pod qui ont un label “app” égale à “toto”, my-service distribue les requêtes selon une approche aléatoire.

Un service NodePort est le moyen le plus simple d'aiguiller du trafic externe directement vers un Pod. NodePort, comme son nom l'indique, ouvre un port spécifique sur tous les nodes, et tout trafic envoyé vers ce port est transféré vers le service.

Spec pour un service de **NodePort**:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - port: 80
    protocol: TCP
```

```

targetPort: 8080
nodePort: 32016
selector:
  app: toto
  type: NodePort

```

Un service LoadBalancer est le moyen standard d'exposer un service via un load balancer. Ce type de service est particulièrement adapté aux clusters Kubernetes managés par un fournisseur cloud. Par exemple sur l'infrastructure Cloud de Google, dans un cluster GKE, cela fera tourner un loadbalancer qui donnera une adresse IP unique qui transférera tout le trafic vers votre service.

Spec pour un service de type **LoadBalancer**:

```

apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    run: toto
  type: LoadBalancer

```

## Labels et choix d'un node pour le déploiement

Pour nous intéresser plus tard à l'affinité nous devons auparavant nous intéresser au **nodeSelector**.

Le nodeSelector est une des formes de contrainte de sélection de node disponible dans Kubernetes. C'est un champ qu'on rajoute dans la spec des pods, et où on spécifie une paire de clé-valeur correspondant au label du node sur lequel on souhaite recevoir notre pod. Passons en revue un exemple d'utilisation de nodeSelector.

Avant toute chose, pour assigner un label à un node:

```
kubectl label node <NODE_NAME> <key>=<value>
```

Dans cet exemple je vais affecter le label ntype:toto pour mon node worker-1.

```
kubectl label node worker-1 ntype=toto
```

Vous pouvez vérifier que cela a fonctionné en exécutant la commande suivante:

```
kubectl get nodes worker-1 --show-labels
```

```
# Confirmons que l'on retrouve bien notre label
NAME     STATUS   ROLES   AGE    VERSION   LABELS
worker-1  Ready    <none>  82m   v1.14.0  beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linu
```

L'étape suivante sera d'ajouter le champ nodeSelector à la configuration YAML de votre pod.

```

apiVersion: v1
kind: pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    ntype: toto

```

Créons notre pod avec la commande suivante:

```
kubectl create -f pod.yaml
```

Une fois la commande lancée, le pod est alors planifié sur le node auquel vous avez attaché le label. Constatons cela avec la commande suivante:

```
kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
nginx	1/1	Running	0	2m52s	10.44.0.1	worker-1	<node>	<node>

Vous pouvez supprimer le label de votre node en rajoutant le signe ‘-’ (moins) à la fin de la commande kubectl label sans spécifier la valeur de la clé. Par exemple si je souhaite supprimer le label crée précédemment, je vais utiliser la commande suivante:

```
kubectl label node worker-1 ntype-
```

Nous pouvons également ordonner un pod sur un node spécifique via le paramètre nodeName.

Par exemple:

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeName: foo-node # schedule pod to specific node
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent

```

Nous pourrions par exemple souhaiter lancer un pod sur plusieurs nodes avec différents labels ou encore lancer un pod sur tous les nodes qui ne contiennent pas de tel ou tel label.

Néanmoins le nodeSelector ou le nodeName possèdent quelques limites. Dans notre exemple nous n'avions utilisé qu'un seul label de sélection pour atteindre notre objectif.

Mais que se passe-t-il si notre exigence est beaucoup plus complexe ?

Pour résoudre ces problématiques complexes, nous utiliserons une autre fonctionnalité proposée par Kubernetes qui est l'affinité ou anti-affinité de node, que nous

aborderons dans la section suivante.

## Affinité et anti-affinité

**La fonctionnalité d'affinité de node nous fournit des fonctionnalités avancées pour limiter le placement de pods sur des nodes spécifiques. Dans Kubernetes, les directives relatives aux "Affinités" contrôlent comment les Pods sont programmés - plus regroupés ou plus dispersés.**

**Pour PodAffinity, vous pouvez essayer de regrouper un certain nombre de Pods dans des domaines de topologie qualifiés.**

**Pour PodAntiAffinity, seulement un Pod peut être programmé dans un domaine de topologie unique.**

La fonctionnalité “EvenPodsSpread” fournit des options flexibles pour distribuer des Pods uniformément sur différents domaines de topologie - pour mettre en place de la haute disponibilité ou réduire les coûts. Cela peut aussi aider au rolling update des charges de travail et à la mise à l'échelle de répliques.

Il existe actuellement deux types d'affinité de node, appelés:

**requiredDuringSchedulingIgnoredDuringExecution preferredDuringSchedulingIgnoredDuringExecution**

On peut distinguer ces types avec les états suivants:

During Scheduling	During Execution
required	ignored
preferred	ignored

Nous avons ainsi:

**During Scheduling:** c'est l'état où un pod n'existe pas encore et qu'il est créé pour la première fois.

Cet état peut bénéficier de deux valeurs:

**required:** exige que le pod soit placé sur un node respectant les règles d'affinité. S'il ne parvient pas à en trouver un, le pod ne sera alors pas planifié.

**preferred:** quand aucun node correspondant n'est trouvé. Le scheduler ignorera simplement les règles d'affinité du pod et le placera sur n'importe quel node disponible.

C'est une façon de dire au scheduler: > **“Fais de ton mieux pour placer le pod sur la correspondance d'affinité mais si tu ne peux vraiment pas en trouver un, place-le sur n'importe quel autre pod.”**

**During Execution:** c'est l'état lorsqu'un pod est déjà exécuté et qu'un changement a été apporté à l'environnement qui affecte l'affinité du node, tel qu'un changement dans le label du node.

Comme vous pouvez le constater, les deux types d'affinité disponibles aujourd'hui sont à l'état ignored, ce qui signifie que les pods continueront à fonctionner et que toute modification de l'affinité des nodes, n'aura aucun impact une fois qu'ils sont planifiés.

L'affinité de node est spécifiée dans le champ nodeAffinity du champ affinity dans la spécification du pod. Voici un exemple d'un pod utilisant une affinité de node:

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: ntype
                operator: NotIn
                values:
                  - virus
                  - malware
  containers:
  - name: nginx
    image: nginx
```

Cette règle d'affinité de node indique que le pod ne peut pas être placé (**grâce à l'opérateur NotIn**) sur un node portant un label dont la clé est ntype et dont la valeur est virus ou malware.

Voici les différents opérateurs avec leur description pris en charge par les règles d'affinité de node:

- **In** - utiliser les nodes avec les mêmes clés et valeurs à ceux qui sont définies dans le spec des pods.
- **NotIn** - ignorer les nodes avec les mêmes clés et valeurs à ceux qui sont définies dans le spec des pods.
- **Exists** - utiliser les nodes avec les mêmes clés à ceux qui sont définies dans le spec des pods.
- **DoesNotExist** - ignorer les nodes avec les mêmes clés à ceux qui sont définies dans le spec des pods.
- **Gt** - utiliser les nodes avec des valeurs numériques supérieur à ceux qui sont définies dans le spec du pod.
- **Lt** - utiliser les nodes avec des valeurs numériques inférieur à ceux qui sont définies dans le spec du pod.

Les opérateurs **NotIn** et **DoesNotExist** provoquent ce que l'on appelle un comportement anti-affinité de node afin de respawn les pods des nodes que vous avez spécifiés.

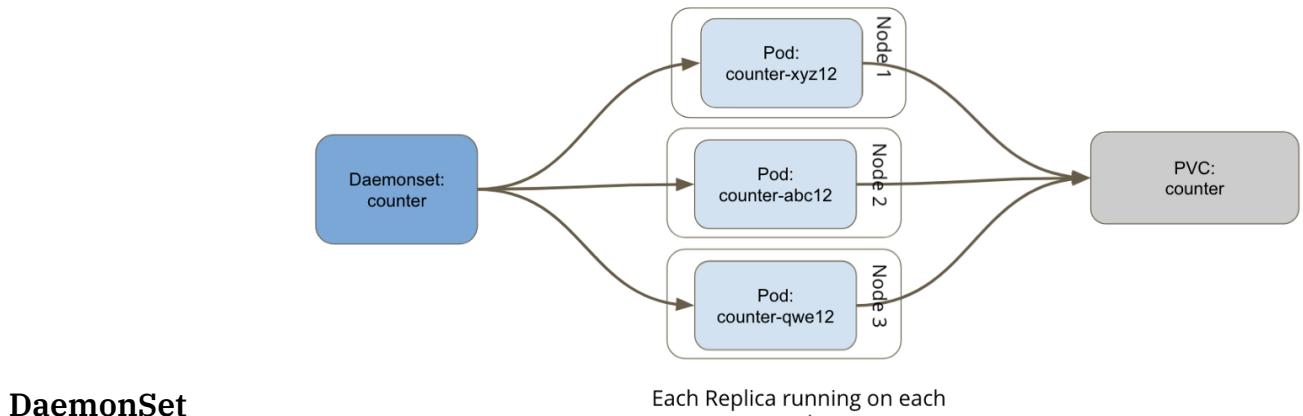
*Sources*

[Assigning Pods to Nodes](#)

[Manipuler le scheduler Kubernetes](#)

## Daemons set, health check, config map et secrets

### Persistence in Daemonsets



DaemonSet

Each Replica running on each

**DaemonSet - Un DaemonSet est un contrôleur qui va s'assurer qu'un seul et unique pod s'exécute sur un node. C'est utile pour faire du monitoring serveur ou collecter des logs par exemple. Ainsi quand un node est ajouté au cluster, le DaemonSet va lancer lui même le pod qu'il définit.**

Vous pouvez décrire un DaemonSet dans un fichier YAML. Par exemple, le fichier daemonset.yaml ci-dessous décrit un DaemonSet qui exécute l'image Docker fluentd-elasticsearch:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
  spec:
    tolerations:
      # this toleration is to have the daemonset runnable on master nodes
      # remove it if your masters can't run pods
      - key: node-role.kubernetes.io/master
```

```

operator: Exists
effect: NoSchedule
containers:
- name: fluentd-elasticsearch
  image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
  resources:
    limits:
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 200Mi
  volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
volumes:
- name: varlog
  hostPath:
    path: /var/log
- name: varlibdockercontainers
  hostPath:
    path: /var/lib/docker/containers

```

Puis, appliquer la spec:

```
kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml
```

Pour détruire le DaemonSet:

```
kubectl delete -f https://k8s.io/examples/controllers/daemonset.yaml
```

## HealthCheck

**Afin de vérifier si un conteneur dans un pod est sain et prêt à servir le trafic, Kubernetes prévoit une série de mécanismes de contrôle de santé. Les contrôles de santé, ou sondes comme on les appelle dans Kubernetes, sont effectués par la kubelet pour déterminer quand redémarrer un conteneur (sondes de lavidité) et utilisés par les services et les déploiements pour déterminer si un pod doit recevoir du trafic (sondes de disponibilité). Dans ce qui suit, nous nous concentrerons sur les contrôles de santé HTTP.**

Notez qu'il est de la responsabilité du développeur de l'application d'exposer une URL que le kubelet peut utiliser pour déterminer si le conteneur est sain (et potentiellement prêt).

Créons un pod qui expose un endpoint /health, répondant avec un code d'état HTTP 200:

```
kubectl apply -f \
https://raw.githubusercontent.com/openshift-evangelists/kbe/main/specs/healthz/pod.yaml
```

Dans la spec du pod nous avons défini ce qui suit:

```
livenessProbe:  
initialDelaySeconds: 2  
periodSeconds: 5  
httpGet:  
path: /health  
port: 9876
```

La configuration ci-dessus indique à Kubernetes de commencer à vérifier le point de terminaison **/health**, après avoir initialement attendu 2 secondes, toutes les 5 secondes.

Si nous regardons maintenant le pod, nous pouvons voir qu'il est considéré comme sain:

```
kubectl describe pod hc  
  
Name:      hc  
Namespace:  default  
Priority:   0  
Node:       minikube/192.168.39.51  
...  
Containers:  
  sise:  
    Container ID:  docker://2cfe4187808a89ae4731abfe242ac42611e1f658505691f540ac31ca8f6ce86f  
    Image:        quay.io/openshiftlabs/simpleservice:0.5.0  
    ...  
    Ready:        True  
    Restart Count: 0  
    Liveness:     http-get http://:9876/health delay=2s timeout=1s period=5s #success=1 #failure=3  
    Environment:  <none>  
Conditions:  
  Type        Status  
  Initialized  True  
  Ready        True  
  ContainersReady  True  
  PodScheduled  True  
...
```

[Source HealthChecks Kube By Example](#)

## ConfigMap

**Une ConfigMap est un objet d'API utilisé pour stocker des données non confidentielles dans des paires clé-valeur. Les pods peuvent utiliser les ConfigMaps comme variables d'environnement, arguments de ligne de commande ou fichiers de configuration dans un volume. Un ConfigMap vous permet de découpler la configuration spécifique à l'environnement de vos images de conteneur, afin que vos applications soient facilement portables.**

Typiquement, nous pourrions renseigner en tant que ConfigMap les fichiers de configuration de nos bases de données.

Utilisez une ConfigMap pour définir les données de configuration séparément du code de l'application.

Par exemple, imaginez que vous développez une application que vous pouvez exécuter sur votre propre ordinateur (pour le développement) et dans le cloud (pour gérer le trafic réel). Vous écrivez le code pour chercher dans une variable d'environnement appelée DATABASE\_HOST. Localement, vous attribuez à cette variable la valeur localhost. Dans le nuage, vous la définissez pour faire référence à un service Kubernetes qui expose le composant de base de données à votre cluster. Cela vous permet de récupérer une image de conteneur exécutée dans le nuage et de déboguer le même code localement si nécessaire.

Une ConfigMap n'est pas conçue pour contenir de grandes quantités de données. Les données stockées dans une ConfigMap ne peuvent pas dépasser 1 Mo. Si vous devez stocker des paramètres dont la taille dépasse cette limite, vous pouvez envisager de monter un volume ou d'utiliser une base de données ou un service de fichiers distinct.

Créer un objet ConfigMap et définissez la spec d'un pod utilisant cette ConfigMap:

```
# cm.yaml
# ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
  ---

# Pod
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
```

```

        # from the key name in the ConfigMap.

    valueFrom:
      configMapKeyRef:
        name: game-demo          # The ConfigMap this value comes from.
        key: player_initial_lives # The key to fetch.
  - name: UI_PROPERTIES_FILE_NAME
    valueFrom:
      configMapKeyRef:
        name: game-demo
        key: ui_properties_file_name
  volumeMounts:
  - name: config
    mountPath: "/config"
    readOnly: true
volumes:
  # You set volumes at the Pod level, then mount them into containers inside that Pod
  - name: config
    configMap:
      # Provide the name of the ConfigMap you want to mount.
      name: game-demo
      # An array of keys from the ConfigMap to create as files
      items:
      - key: "game.properties"
        path: "game.properties"
      - key: "user-interface.properties"
        path: "user-interface.properties"

```

kubectl apply -f cm.yaml

Pour détruire les ressources:

kubectl delete -f cm.yaml

## Secrets

**Les objets secret de Kubernetes vous permettent de stocker et de gérer des informations sensibles, telles que les mots de passe, les tokens OAuth et les clés ssh. Mettre ces informations dans un secret est plus sûr et plus flexible que de le mettre en dur dans la définition d'un Pod ou dans une image de container.**

Un secret est un objet qui contient une petite quantité de données sensibles telles qu'un mot de passe, un token ou une clé. De telles informations pourraient autrement être placées dans une spécification de pod ou dans une image; le placer dans un objet secret permet de mieux contrôler la façon dont il est utilisé et réduit le risque d'exposition accidentelle.

Les utilisateurs peuvent créer des secrets et le système crée également des secrets.

Pour utiliser un secret, un pod doit référencer le secret. Un secret peut être utilisé avec un pod de deux manières: sous forme de fichiers dans un volume monté sur un ou plusieurs de ses conteneurs, ou utilisé par kubelet lorsque vous récupérez des images pour le pod.

Vous pouvez également créer un secret dans un fichier d'abord, au format json ou yaml, puis créer cet objet. Le secret contient deux table de hachage: data et stringData. Le champ data est utilisé pour stocker des données arbitraires, encodées en base64. Le champ stringData est fourni pour plus de commodité et vous permet de fournir des données secrètes sous forme de chaînes non codées.

Par exemple, pour stocker deux chaînes dans un secret à l'aide du champ data, convertissez-les en base64 comme suit:

```
echo -n 'admin' | base64  
YWRtaW4=  
echo -n '1f2d1e2e67df' | base64  
MWYyZDFlMmU2N2Rm
```

Écrivez un secret qui ressemble à ceci:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
data:  
  username: YWRtaW4=  
  password: MWYyZDFlMmU2N2Rm
```

Maintenant, créez le secret en utilisant `kubectl apply` :

```
kubectl apply -f ./secret.yaml
```

```
secret "mysecret" created
```

Pour certains scénarios, vous pouvez utiliser le champ stringData à la place. Ce champ vous permet de mettre une chaîne non codée en base64 directement dans le secret, et la chaîne sera codée pour vous lorsque le secret sera créé ou mis à jour.

Un exemple pratique de cela pourrait être le suivant: vous déployez une application qui utilise un secret pour stocker un fichier de configuration. Vous souhaitez remplir des parties de ce fichier de configuration pendant votre processus de déploiement.

Si votre application utilise le fichier de configuration suivant:

```
apiUrl: "https://my.api.com/api/v1"  
username: "user"  
password: "password"
```

Vous pouvez stocker cela dans un secret en utilisant ce qui suit:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
stringData:  
  config.yaml: |-  
    apiUrl: "https://my.api.com/api/v1"  
    username: {{username}}  
    password: {{password}}
```

Votre outil de déploiement pourrait alors remplacer les variables de modèle {{username}} et {{password}} avant d'exécuter `kubectl apply`.

stringData est un champ de commodité en écriture seule. Il n'est jamais affiché lors de la récupération des secrets. Par exemple, si vous exécutez la commande suivante:

```
kubectl get secret mysecret -o yaml
```

L'output généré sera alors:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2021-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
data:
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXN1cm5hbWU6IHt7dXN1cm5hbWV9fQpwYXNzd2
```

Pour décoder un secret nous aurons besoin de récupérer le secret créé via la commande `kubectl get secret`.

```
kubectl get secret mysecret -o yaml
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2021-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Décodez le champ du mot de passe:

```
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

```
1f2d1e2e67df
```

## Persistent Volumes et Persistent Volumes Claim

La gestion du stockage est un problème distinct de la gestion du compute.

Le sous-système PersistentVolume fournit aux utilisateurs et aux administrateurs une API qui fait abstraction des détails sur la façon dont le stockage est fourni et sur la façon dont il est consommé.

Pour ce faire, nous introduisons deux nouvelles ressources API: **PersistentVolume** et **PersistentVolumeClaim**.

**Un PersistentVolume (PV) est un élément de stockage en réseau dans le cluster qui a été provisionné par un administrateur.** Il s'agit d'une ressource du cluster, tout comme un nœud est une ressource du cluster. Les PV sont des plugins de volume comme les volumes, mais ont un cycle de vie indépendant de tout pod individuel qui utilise le PV. Cet objet API capture les détails de l'implémentation du stockage, qu'il s'agisse de NFS, d'iSCSI ou d'un système de stockage spécifique au fournisseur de cloud.

**Une PersistentVolumeClaim (PVC) est une demande de stockage par un utilisateur.** Elle est similaire à un pod. Les pods consomment des ressources de nodes et les PVC des ressources de PV. Les pods peuvent demander des niveaux spécifiques de ressources (CPU et mémoire). Les revendications peuvent demander une taille et des modes d'accès spécifiques (par exemple, peuvent être montées une fois en lecture/écriture ou plusieurs fois en lecture seule).

Alors que les PersistentVolumeClaims permettent à un utilisateur de consommer des ressources de stockage abstraites, il est courant que les utilisateurs aient besoin de PersistentVolumes avec des propriétés variables, telles que la performance, pour différents problèmes.

Les administrateurs de clusters doivent être en mesure d'offrir une variété de PersistentVolumes qui diffèrent en plus de la taille et des modes d'accès, sans exposer les utilisateurs aux détails de l'implémentation de ces volumes.

Pour ces besoins, il existe la ressource StorageClass.

**Une StorageClass permet aux administrateurs de décrire les "classes" de stockage qu'ils proposent.** Les différentes classes peuvent correspondre à des niveaux de qualité de service, à des politiques de sauvegarde ou à des politiques arbitraires déterminées par les administrateurs du cluster. Kubernetes lui-même ne se prononce pas sur ce que représentent les classes. Ce concept est parfois appelé "profils" dans d'autres systèmes de stockage.

Quelques types de plugins pour persistent volumes:

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- FC (Fibre Channel)
- Flocker
- NFS
- iSCSI
- RBD (Ceph Block Device)
- CephFS
- Cinder (OpenStack block storage)
- Glusterfs
- VsphereVolume

- Quobyte Volumes
- HostPath (tester sur un seul nœud uniquement - le stockage local n'est en aucun cas pris en charge et ne fonctionnera pas dans un cluster multi-nœuds)
- VMware Photon
- Portworx Volumes
- ScaleIO Volumes

Les plug-ins de volume suivants prennent en charge les volumes de blocs bruts, y compris l'approvisionnement dynamique, le cas échéant:

- AWSElasticBlockStore
- AzureDisk
- FC (Fibre Channel)
- GCEPersistentDisk
- iSCSI
- Local volume
- RBD (Ceph Block Device)
- VsphereVolume (alpha)

**Seuls les volumes FC et iSCSI prennent en charge les volumes de blocs bruts dans Kubernetes 1.9. La prise en charge des plugins supplémentaires a été ajoutée dans 1.10.**

Se référer à la documentation spécifique à chaque plugin pour déterminer la spec permettant leur utilisation.

Pour des volumes persistants utilisant un volume de bloc brut:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cf1"]
    lun: 0
    readOnly: false
```

Revendication de volume persistant demandant un volume de bloc brut:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
```

```
  requests:  
    storage: 10Gi
```

Spécification de pod ajoutant le chemin du périphérique de bloc brut dans le conteneur:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: pod-with-block-volume  
spec:  
  containers:  
    - name: fc-container  
      image: fedora:26  
      command: ["/bin/sh", "-c"]  
      args: [ "tail -f /dev/null" ]  
      volumeDevices:  
        - name: data  
          devicePath: /dev/xvda  
  volumes:  
    - name: data  
      persistentVolumeClaim:  
        claimName: block-pvc
```

N.B. Lorsque vous ajoutez un périphérique de bloc brut pour un pod, vous spécifiez le chemin de périphérique dans le conteneur au lieu d'un chemin de montage.

#### Sources

[Unofficial K8s: persistent volumes](#)

[Concepts: Persistent Volumes](#)

## Déploiement d'une base de données et d'une application (TP)

Vos disposez de la spec suivante qui définit respectivement:

- un PVC qui crée automatiquement le PV associé via la storage-class local-path

Vous devrez définir:

- un service mysql
- une configmap pour un fichier de configuration de votre choix
- un secret pour notre db
- le deployment de notre db

Vous aurez à charge d'écrire la spec du deployment du service et du secret mysql correspondant. N'hésitez pas pour ce faire à vous reportez à la section “Syntaxe YAML”.

*mysql-full-resources.yaml*

```
apiVersion: storage.k8s.io/v1  
kind: StorageClass  
metadata:  
  name: ebs-storage-class  
provisioner: kubernetes.io/aws-ebs
```

```

parameters:
  type: io1
  iopsPerGB: '10'
  fsType: xfs
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: ebs-pvc
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: ebs-storage-class
---
apiVersion: apps/v1
kind: Deployment
# Compléter ce qui suit
---
apiVersion: v1
kind: ConfigMap
# Compléter ce qui suit
---
apiVersion: v1
kind: Service
# Compléter ce qui suit
---
apiVersion: v1
kind: Secret
# Compléter ce qui suit
kubectl apply mysql-full-resources.yaml

```

*Facultatif:* Réécrivez les specs en utilisant une autre classe de storage que celle définie précédemment (kubernetes.io/aws-ebs) et créer le PV et le PVC en conséquence.

## TP Version 2 (on K3s instead of AWS)

Vos disposez de la spec suivante qui définit respectivement:

- un PVC qui crée automatiquement le PV associé via la storage-class local-path

Vous devrez définir:

- un service mysql
- un secret pour notre db
- le deployment de notre db

Vous aurez à charge d'écrire la spec du deployment du service et du secret mysql correspondant. N'hésitez pas pour ce faire à vous reportez à la section "Syntaxe

YAML".

```
mysql-full-resources.yaml

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-path-pvc
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 2Gi
---
apiVersion: apps/v1
kind: Deployment
# Completer ce qui suit
---
apiVersion: v1
kind: ConfigMap
# Completer ce qui suit
---
apiVersion: v1
kind: Service
# Completer ce qui suit
---
apiVersion: v1
kind: Secret
# Completer ce qui suit
kubectl apply mysql-full-resources.yaml
```

*Facultatif:* Réécrivez les specs en utilisant une autre classe de storage que celle définie précédemment (local-path) et créer le PV et le PVC en conséquence.

## Kubernetes en production

### Frontal administrable Ingress

**Ingress (ou entrée réseau), ajouté à Kubernetes v1.1, est un objet Kubernetes qui expose les routes HTTP et HTTPS de l'extérieur du cluster à des services au sein du cluster. Le routage du trafic est contrôlé par des règles définies sur la ressource Ingress.**

```
internet
  |
[ Ingress ]
--|----|--
[ Services ]
```

*FEATURE STATE: Kubernetes v1.1 [beta]* Avant de commencer à utiliser un Ingress, vous devez comprendre qu'un Ingress est une ressource en "version Beta".

Un Ingress peut être configuré pour donner aux services des URLs accessibles de l'extérieur, un équilibrage du trafic de charge externe, la terminaison SSL/TLS et un hébergement virtuel basé sur le nom. Un contrôleur d'Ingress est responsable de l'exécution de l'Ingress, généralement avec un load-balancer (équilibrEUR de charge), bien qu'il puisse également configurer votre routeur périphérique ou des interfaces supplémentaires pour aider à gérer le trafic.

Un Ingress n'expose pas de ports ni de protocoles arbitraires. **Exposer des services autres que HTTP et HTTPS à Internet généralement utilise un service de type Service.Type=NodePort ou Service.Type=LoadBalancer.**

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

La spécification de la ressource Ingress dispose de toutes les informations nécessaires pour configurer un loadbalancer ou un serveur proxy. Plus important encore, il contient une liste de règles d'appariement de toutes les demandes entrantes. La ressource Ingress ne supporte que les règles pour diriger le trafic HTTP.

Pour 2 vhosts sur un ingress:

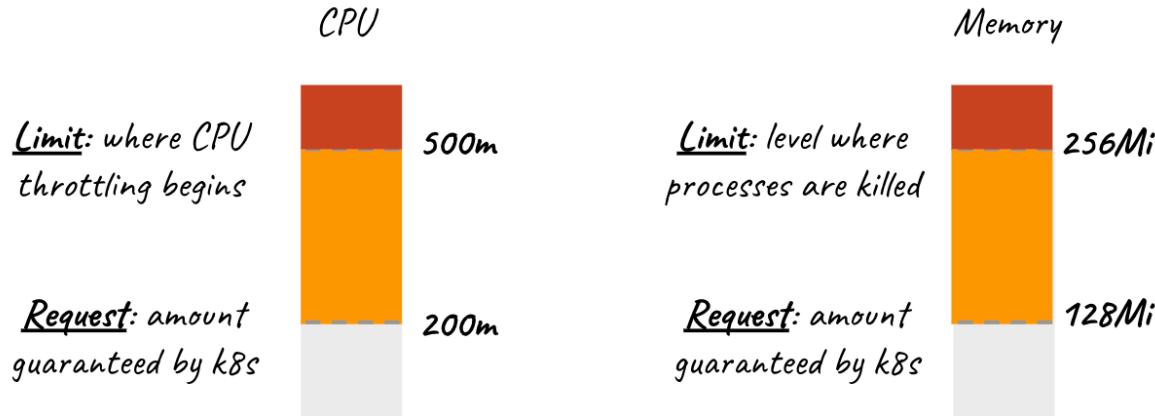
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: service-a.example.com
    http:
      paths:
      - backend:
          serviceName: service-a
          servicePort: 80
  - host: service-b.example.com
    http:
      paths:
```

```

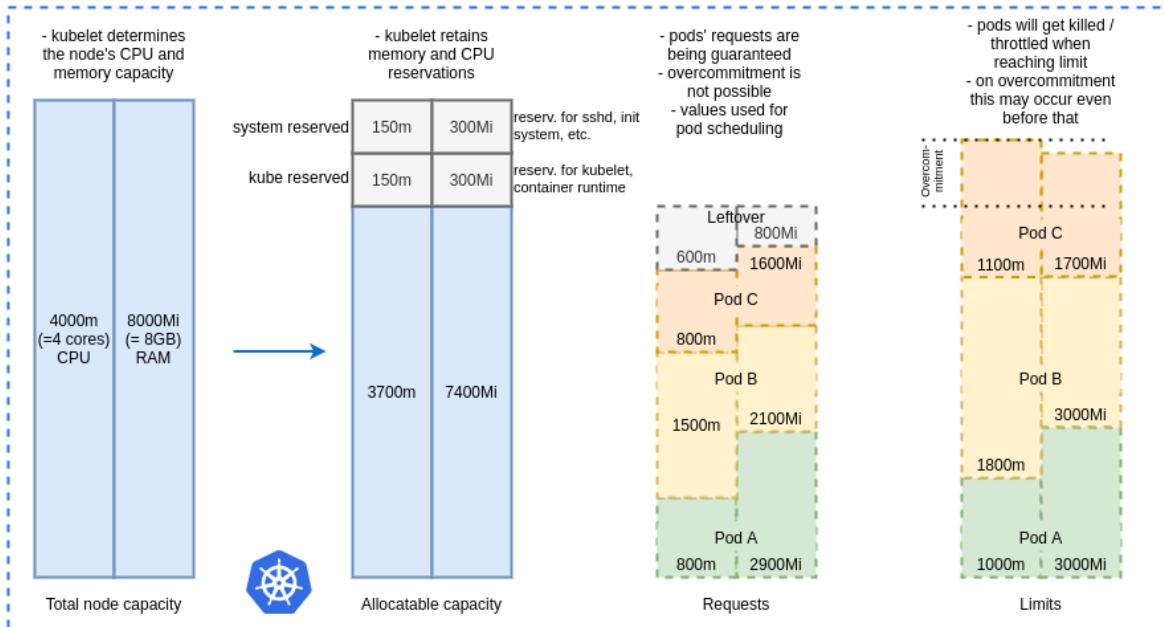
- backend:
  serviceName: service-a
  servicePort: 80

```

## Limitation de ressources



## Kubernetes Resource Requests and Limits



Lorsque Kubernetes planifie un Pod, il est important que les conteneurs disposent de suffisamment de ressources pour fonctionner. Si vous planifiez une grosse application sur un node aux ressources limitées, il est possible que le node soit à court de mémoire ou de ressources CPU et que les pods cessent de fonctionner.

**Les requêtes et les limites sont les mécanismes utilisés par Kubernetes pour contrôler les ressources telles que le CPU et la mémoire.  
Les requêtes sont ce que le conteneur est assuré d'obtenir.**

Si un conteneur demande une ressource, Kubernetes ne le programmera que sur un node qui peut lui fournir cette ressource. Les limites, quant à elles, garantissent qu'un conteneur ne dépasse jamais une certaine valeur. Le conteneur est seulement autorisé à aller jusqu'à la limite, et ensuite il est restreint.

Il est important de se rappeler que la limite ne peut jamais être inférieure à la demande. Si vous essayez de le faire, Kubernetes émettra une erreur et ne vous laissera pas exécuter le conteneur.

Les demandes et les limites sont établies par conteneur. Si les pods ne contiennent généralement qu'un seul conteneur, il est courant de voir des pods avec plusieurs conteneurs. Chaque conteneur du pod reçoit sa propre limite et sa propre demande, mais comme les pods sont toujours planifiés en tant que groupe, vous devez additionner les limites et les demandes de chaque conteneur pour obtenir une valeur globale pour le pod.

Il existe deux types de ressources :

- Le processeur et la mémoire.

Le scheduler de Kubernetes les utilisent pour déterminer où exécuter vos pods.

Une spécification typique de ressources pour un pod peut ressembler à ceci. Ce pod possède deux conteneurs :

```
containers:  
  - name: container1  
    image: busybox  
    resources:  
      requests:  
        memory: "32Mi"  
        cpu: "200m"  
      limits:  
        memory: "64Mi"  
        cpu: "200m"  
  - name: container2  
    image: busybox  
    resources:  
      requests:  
        memory: "96Mi"  
        cpu: "300m"  
      limits:  
        memory: "192Mi"  
        cpu: "750m"
```

Chaque conteneur du pod peut définir ses propres demandes et limites, qui sont toutes additives. Ainsi, dans l'exemple ci-dessus, le Pod a une demande totale de 500 m de CPU et 128 MiB de mémoire, et une limite totale de 1 CPU et 256MiB de mémoire.

*CPU* Les ressources CPU sont définies en millicores. Si votre conteneur a besoin de deux coeurs complets pour fonctionner, vous mettez la valeur "2000m". Si votre

conteneur n'a besoin que de ¼ de cœur, vous mettrez une valeur de "250m".

Une chose à garder à l'esprit concernant les demandes de CPU est que si vous mettez une valeur supérieure au nombre de cœurs de votre plus gros node, votre pod ne sera jamais programmé. Disons que vous avez un pod qui a besoin de quatre cœurs, mais que votre cluster Kubernetes est composé de VM à double cœur - votre pod ne sera jamais programmé !

À moins que votre application ne soit spécifiquement conçue pour tirer parti de plusieurs cœurs (l'informatique scientifique et certaines bases de données viennent à l'esprit), il est généralement préférable de maintenir la demande de CPU à '1' ou moins, et d'exécuter plus de répliques pour la faire évoluer. Cela donne au système plus de flexibilité et de fiabilité.

*RAM* Les ressources mémoire sont définies en octets. Normalement, vous donnez une valeur de mebibytes pour la mémoire (c'est en fait la même chose qu'un mégaoctet), mais vous pouvez donner n'importe quelle valeur en octets.

Comme pour le CPU, si vous introduisez une demande de mémoire supérieure à la quantité de mémoire disponible sur vos nodes, le pod ne sera jamais programmé.

Contrairement aux ressources du CPU, la mémoire ne peut pas être compressée. Comme il n'y a aucun moyen de limiter l'utilisation de la mémoire, si un conteneur dépasse sa limite de mémoire, il sera terminé. Si votre pod est géré par un Deployment, StatefulSet, DaemonSet ou un autre type de contrôleur, le contrôleur lance un remplacement.

*Nodes* Il est important de se rappeler que vous ne pouvez pas définir des demandes plus importantes que les ressources fournies par vos nodes. Par exemple, si vous avez un cluster de machines à deux cœurs, un Pod avec une demande de 2,5 cœurs ne sera jamais scheduler.

## Service Discovery (env, DNS)

**Dans Kubernetes, la détection de services est mise en œuvre avec des noms de service générés automatiquement correspondant à l'adresse IP du service. Les noms de service suivent la spécification standard suivante: my-svc.my-namespace.svc.cluster-domain.example. Les pods peuvent également accéder à des services externes, tels que example.com, via leur nom. Pour en savoir plus sur le fonctionnement du DNS dans Kubernetes, consultez la page [DNS pour les services et les pods](#).**

Kubernetes fournit les options DNS de cluster suivantes pour résoudre les noms de service et les noms externes :

kube-dns: module complémentaire de cluster déployé par défaut. Cloud DNS: infrastructure DNS de cluster gérée dans le cloud exploitant Cloud DNS et ne nécessitant pas la gestion des serveurs DNS dans les clusters, comme kube-dns. Vous pouvez également enregistrer vos services dans l'annuaire des services pour votre cloud provider.

GKE fournit également le module complémentaire facultatif NodeLocal DNSCache pouvant être utilisé avec kube-dns ou Cloud DNS.

*kube-dns* > **kube-dns** est le fournisseur DNS par défaut des clusters. Il s'exécute comme un déploiement qui programme les pods **kube-dns** sur les nodes du cluster.

*Cloud DNS* > Cloud DNS fournit la résolution DNS des pods et des services, sans nécessiter de fournisseur DNS hébergé par le cluster tel que **kube-dns**. Le contrôleur Cloud DNS provisionne automatiquement les enregistrements DNS des pods et des services dans Cloud DNS pour les adresses ClusterIP, les services sans adresse IP de cluster et les services de noms externes.

*NodeLocal DNSCache* > NodeLocal DNSCache s'exécute en tant que DaemonSet qui programme un pod de cache DNS sur chaque node de cluster. Ce cache DNS améliore la latence de la résolution DNS, harmonise les délais des résolutions DNS, et peut réduire le nombre de requêtes DNS adressées à **kube-dns** ou **Cloud DNS**.

**Détection de services en dehors d'un cluster** Vous pouvez configurer la détection de services sur plusieurs clusters à l'aide de l'une des méthodes suivantes.

*VPC Cloud DNS Scope* Un cluster qui utilise Cloud DNS pour les services DNS de cluster doit s'exécuter dans l'un des deux modes disponibles: champ d'application de cluster ou champ d'application de cloud privé virtuel (VPC).

Lorsque vous configurez un cluster avec un champ d'application de cluster, les enregistrements DNS ne peuvent être résolus que dans le cluster à l'aide du schéma `<svc>. <ns>. svc.cluster.local`. **Ce comportement est le même que celui de kube-dns.**

Lorsque vous configurez un cluster avec un champ d'application de VPC, les enregistrements DNS des services du cluster peuvent être résolus dans l'ensemble du VPC. Cela signifie que les clients du même VPC ou connectés à celui-ci via Cloud VPN ou Cloud Interconnect peuvent résoudre directement les enregistrements DNS des services du cluster.

*Services multiclouds* > **Les services multiclouds fournissent la détection de services et l'équilibrage de charge multiclouds pour GKE qui exploite l'objet Service existant. Les services multiclouds sont visibles et accessibles sur n'importe quel cluster GKE doté d'une seule adresse IP virtuelle. Ce comportement est identique à celui d'un service ClusterIP accessible dans un seul cluster.**

Les services multiclouds agrègent les services entre les clusters et les rendent adressables sous la forme d'un seul enregistrement DNS multicloud à l'aide du schéma `<svc>. <ns>. svc.clusterset.local`.

*Sources*

[GKE Service Discovery](#)

## Les namespaces et les quotas

Après avoir créé des namespaces, vous pouvez les verrouiller à l'aide de Resource-Quotas. Les quotas de ressources sont très puissants, mais voyons simplement comment les utiliser pour limiter l'utilisation des ressources CPU et mémoire.

Un quota de ressource peut ressembler à ceci:

```
apiVersion: v1
kind: ResourceQuotas
metadata:
  name: sample
spec:
  hard:
    requests.cpu: 500m
    requests.memory: 100MiB
    limits.cpu: 700m
    limits.memory: 500MiB
```

En regardant cet exemple, vous pouvez voir qu'il y a quatre sections. La configuration de chacune de ces sections est facultative.

*requests.cpu* > **Nombre maximum de demandes combinées de CPU en millicores pour tous les conteneurs de le namespace.** Dans l'exemple ci-dessus, vous pouvez avoir 50 conteneurs avec des demandes de 10m, cinq conteneurs avec des demandes de 100m, ou même un conteneur avec une demande de 500m. Tant que le total des demandes de CPU dans le namespace est inférieur à 500m!

*requests.memory* > **Demandes maximales combinées de mémoire pour tous les conteneurs dans le namespace.** Dans l'exemple ci-dessus, vous pouvez avoir 50 conteneurs avec des demandes de 2MiB, cinq conteneurs avec des demandes de CPU de 20MiB, ou même un seul conteneur avec une demande de 100MiB. Tant que la mémoire totale demandée dans le namespace est inférieure à 100MiB!

*limits.cpu* > **Limites maximales combinées du CPU pour tous les conteneurs dans le namespace.** C'est comme *requests.cpu* mais pour la limite.

*limits.memory* > **Limites maximales de mémoire combinées pour tous les conteneurs dans le namespace.** C'est comme *requests.memory* mais pour la limite.

Si vous utilisez un Namespace de production et de développement (par opposition à un Namespace par équipe ou service), un modèle commun est de ne pas mettre de quota sur le Namespace de production et des quotas stricts sur le Namespace de développement. Cela permet à la production de prendre toutes les ressources dont elle a besoin en cas de pic de trafic.

*LimitRange* > **Vous pouvez également créer une LimitRange dans votre Namespace.** Contrairement à un quota, qui concerne le namespace dans son ensemble, une LimitRange s'applique à un conteneur individuel. Cela peut aider à empêcher les gens de créer des conteneurs super petits ou super grands dans le namespace.

Une **LimitRange** peut ressembler à ceci :

```
apiVersion: v1
kind: LimitRange
metadata:
  name: sample
spec:
```

```

limits:
- default:
    cpu: 600m
    memory: 100Mib
defaultRequest:
    cpu: 100Mib
    memory: 50Mib
max:
    cpu: 1000m
    memory: 200Mib
min:
    cpu: 10m
    memory: 100Mib
type: Container

```

En regardant cet exemple, vous pouvez voir qu'il y a quatre sections. Là encore, la définition de chacune de ces sections est facultative

*Section default > La section default définit les limites par défaut pour un conteneur dans un pod. Si vous définissez ces valeurs dans la section limitRange, tous les conteneurs qui ne les définissent pas explicitement se verront attribuer les valeurs par défaut.*

*Section defaultRequest > La section defaultRequest définit les demandes par défaut pour un conteneur dans un pod. Si vous définissez ces valeurs dans l'intervalle limitRange, les valeurs par défaut seront attribuées à tous les conteneurs qui ne les définissent pas explicitement.*

*Section max > La section max définit les limites maximales qu'un conteneur dans un pod peut fixer. La section par défaut ne peut être supérieure à cette valeur. De même, les limites définies pour un conteneur ne peuvent être supérieures à cette valeur. Il est important de noter que si cette valeur est définie et que la section par défaut ne l'est pas, tous les conteneurs qui ne définissent pas explicitement ces valeurs eux-mêmes se verront attribuer les valeurs max comme limite.*

*Section min > La section min définit les requêtes minimales qu'un conteneur dans un Pod peut définir. La section defaultRequest ne peut être inférieure à cette valeur. De même, les demandes définies sur un conteneur ne peuvent être inférieures à cette valeur. Il est important de noter que si cette valeur est définie et que la section defaultRequest ne l'est pas, la valeur min devient également la valeur defaultRequest.*

## Gestion des accès (RBAC)

**Le contrôle d'accès basé sur les rôles (RBAC) est une méthode de régulation de l'accès aux ressources informatiques ou réseau en fonction des rôles des utilisateurs individuels au sein de votre organisation.**

L'autorisation RBAC utilise le groupe d'API `rbac.authorization.k8s.io`

**pour piloter les décisions d'autorisation, ce qui vous permet de configurer dynamiquement les politiques via l'API Kubernetes.**

Pour activer RBAC, démarrez le serveur API avec l'indicateur `--authorization-mode` défini sur une liste séparée par des virgules qui inclut RBAC; par exemple:

```
kube-apiserver --authorization-mode=Example,RBAC --other-options --more-options
```

**L'API RBAC déclare quatre types d'objets Kubernetes: Role, ClusterRole, RoleBinding et ClusterRoleBinding.** Vous pouvez décrire les objets, ou les modifier, à l'aide d'outils tels que kubectl, comme tout autre objet Kubernetes.

*Attention!* Ces objets, par conception, imposent des restrictions d'accès. Si vous apportez des modifications à un cluster au fur et à mesure de votre apprentissage, consultez la section Prévention de l'escalade des priviléges et amorçage pour comprendre comment ces restrictions peuvent vous empêcher d'effectuer certaines modifications.

**Rôle et ClusterRole** Un rôle RBAC ou un ClusterRole contient des règles qui représentent un ensemble de permissions. Les permissions sont purement additives (il n'existe pas de règles de "refus").

Un rôle définit toujours les permissions dans un espace de nom particulier; lorsque vous créez un rôle, vous devez spécifier le namespace auquel il appartient.

**ClusterRole**, en revanche, est une ressource sans namespaces. Les ressources ont des noms différents (Role et ClusterRole) parce qu'un objet Kubernetes doit toujours être soit namespaced soit non namespaced; il ne peut pas être les deux.

Les ClusterRoles ont plusieurs usages. Vous pouvez utiliser un ClusterRole pour:

- Définir des autorisations sur des ressources à espace de noms et les accorder au sein d'un ou de plusieurs espaces de noms individuels.
- Définir des autorisations sur des ressources d'espace de noms et les accorder dans tous les espaces de noms
- Définir des permissions sur des ressources à l'échelle du cluster

Si vous souhaitez définir un rôle au sein d'un espace de noms, utilisez un Role ; si vous souhaitez définir un rôle à l'échelle du cluster, utilisez un ClusterRole.

#### *Exemple de rôle*

Voici un exemple de rôle dans l'espace de nom "default" qui peut être utilisé pour accorder un accès en lecture aux pods:

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pods-svc-reader
rules:
- apiGroups: []
  resources: ["pods", "services"]
  verbs: ["get", "list"]
```

#### *Exemple de ClusterRole*

**Un ClusterRole peut être utilisée pour accorder les mêmes permissions qu'un rôle.**

Étant donné que les ClusterRoles sont adaptés aux clusters, vous pouvez également les utiliser pour accorder l'accès à:

- des ressources à l'échelle du cluster (comme les nodes)
- des points de terminaison non liés aux ressources (comme /healthz sur un healthcheck)
- des ressources namespaces (comme les pods), dans tous les namespaces.

Par exemple: vous pouvez utiliser une ClusterRole pour autoriser un utilisateur particulier à exécuter `kubectl get pods --all-namespaces`.

Voici un exemple de ClusterRole qui peut être utilisé pour accorder un accès en lecture aux secrets dans un espace de noms particulier ou dans tous les espaces de noms (selon la façon dont il est lié) :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

Le nom d'un objet Role ou ClusterRole doit être un nom de segment de chemin valide.

#### *RoleBinding et ClusterRoleBinding*

**Un role binding accorde les permissions définies dans un rôle à un utilisateur ou à un ensemble d'utilisateurs. Il contient une liste de sujets (utilisateurs, groupes ou comptes de service) et une référence au rôle accordé. Un RoleBinding accorde des permissions dans un espace de nom spécifique, tandis qu'un ClusterRoleBinding accorde cet accès à l'échelle du cluster.**

**Un RoleBinding peut faire référence à n'importe quel rôle dans le même espace de noms. Par ailleurs, un RoleBinding peut faire référence à un ClusterRole et lier ce ClusterRole à l'espace de nom du RoleBinding. Si vous souhaitez lier un ClusterRole à tous les espaces de noms de votre cluster, vous utilisez un ClusterRoleBinding.**

Le nom d'un objet RoleBinding ou ClusterRoleBinding doit être un nom de segment de chemin valide.

#### *Exemples de RoleBinding*

Voici un exemple de RoleBinding qui accorde le rôle “pod-reader” à l'utilisateur “jane” dans l'espace de nom “default”. Ceci permet à “jane” de lire les pods dans l'espace de noms “default”.

```

apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io

```

**Un RoleBinding peut également faire référence à un ClusterRole pour accorder les permissions définies dans ce ClusterRole aux ressources de le namespace du RoleBinding. Ce type de référence vous permet de définir un ensemble de rôles communs à l'ensemble de votre cluster, puis de les réutiliser dans plusieurs namespaces.**

Par exemple, même si le RoleBinding suivant fait référence à un ClusterRole, “dave” (le sujet, sensible à la casse) ne pourra lire que les Secrets dans l'espace de nom “development”, car le namespace du RoleBinding (dans ses métadonnées) est “development”.

```

apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "dave" to read secrets in the "development" namespace.
# You need to already have a ClusterRole named "secret-reader".
kind: RoleBinding
metadata:
  name: read-secrets
  #
# The namespace of the RoleBinding determines where the permissions are granted.
# This only grants permissions within the "development" namespace.
  namespace: development
subjects:
- kind: User
  name: dave # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io

```

#### *ClusterRoleBinding example*

Pour accorder des permissions à l'ensemble d'un cluster, vous pouvez utiliser un ClusterRoleBinding. Le ClusterRoleBinding suivant permet à tout utilisateur du groupe “manager” de lire les secrets dans n'importe quel namespace.

```

apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "dave" to read secrets in the "development" namespace.
# You need to already have a ClusterRole named "secret-reader".
kind: RoleBinding
metadata:
  name: read-secrets
  #
  # The namespace of the RoleBinding determines where the permissions are granted.
  # This only grants permissions within the "development" namespace.
  namespace: development
subjects:
- kind: User
  name: dave # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io

```

Après avoir créé un lien, vous ne pouvez pas modifier le rôle ou le ClusterRole auquel il fait référence. Si vous essayez de modifier le roleRef d'une liaison, vous obtenez une erreur de validation. Si vous souhaitez modifier le roleRef d'un binding, vous devez supprimer l'objet binding et en créer un autre.

Il y a deux raisons pour cette restriction:

- Rendre le roleRef immuable permet d'accorder à quelqu'un la permission de mettre à jour un objet de liaison existant, afin qu'il puisse gérer la liste des sujets, sans pouvoir changer le rôle qui est accordé à ces sujets.

L'utilitaire de ligne de commande kubectl auth reconvoie crée ou met à jour un fichier manifeste contenant les objets RBAC, et gère la suppression et la recréation des objets de liaison si nécessaire pour modifier le rôle auquel ils se réfèrent. Voir l'utilisation de la commande et les exemples pour plus d'informations.

## Haute disponibilité et mode maintenance

Cette page explique deux approches différentes pour configurer un Kubernetes à haute disponibilité cluster utilisant kubeadm:

Avec des nodes de control plane empilés. Cette approche nécessite moins d'infrastructure. Les membres etcd et les nodes du control plane sont co-localisés.

Avec un cluster etcd externe cette approche nécessite plus d'infrastructure. Les nodes du control plane et les membres etcd sont séparés.

Avant de poursuivre, vous devez déterminer avec soin quelle approche répond le mieux aux besoins de vos applications et de l'environnement. Cette comparaison décrit les avantages et les inconvénients de chacune.

Vos clusters doivent exécuter Kubernetes version 1.12 ou ultérieure. Vous devriez aussi savoir que la mise en place de clusters HA avec kubeadm est toujours expérimentale et sera simplifiée davantage dans les futures versions. Vous pouvez par exemple rencontrer des problèmes lors de la mise à niveau de vos clusters. Nous

vous encourageons à essayer l'une ou l'autre approche et à nous faire part de vos commentaires dans Suivi des problèmes Kubeadm.

Avertissement: Cette page ne traite pas de l'exécution de votre cluster sur un fournisseur de cloud. Dans un environnement Cloud, les approches documentées ici ne fonctionnent ni avec des objets de type load balancer, ni avec des volumes persistants dynamiques.

**Pré-requis** Pour les deux méthodes, vous avez besoin de cette infrastructure:

- Trois machines qui répondent aux pré-requis des exigences de kubeadm pour les maîtres (masters)
- Trois machines qui répondent aux pré-requis des exigences de kubeadm pour les workers
- Connectivité réseau complète entre toutes les machines du cluster (public ou réseau privé)
- Privilèges sudo sur toutes les machines
- Accès SSH d'une machine à tous les nodes du cluster
- kubeadm et une kubelet installés sur toutes les machines. kubectl est optionnel.

Pour le cluster etcd externe uniquement, vous avez besoin également de:

- Trois machines supplémentaires pour les membres etcd

*N.B.* Les exemples suivants utilisent Calico en tant que fournisseur de réseau de Pod. Si vous utilisez un autre CNI, pensez à remplacer les valeurs par défaut si nécessaire.

*N.B.* Toutes les commandes d'un control plane ou d'un noeud etcd doivent être exécutées en tant que root. Certains plugins réseau CNI tels que Calico nécessitent un CIDR tel que 192.168.0.0 / 16 et certains comme Weave n'en ont pas besoin. Voir la Documentation du CNI réseau.

Pour ajouter un CIDR de pod, définissez le champ podSubnet: 192.168.0.0 / 16 sous l'objet networking de ClusterConfiguration.

**Créez un load balancer pour kube-apiserver** Note: Il existe de nombreuses configurations pour les équilibreurs de charge (load balancers). L'exemple suivant n'est qu'un exemple. Vos exigences pour votre cluster peuvent nécessiter une configuration différente.

### **Créez un load balancer kube-apiserver avec un nom résolu en DNS.**

Dans un environnement cloud, placez vos nodes du control plane derrière un load balancer TCP. Ce load balancer distribue le trafic à tous les nodes du control plane sains dans sa liste. La vérification de la bonne santé d'un apiserver est une vérification TCP sur le port que kube-apiserver écoute (valeur par défaut: 6443).

Il n'est pas recommandé d'utiliser une adresse IP directement dans un environnement cloud.

Le load balancer doit pouvoir communiquer avec tous les nodes du control plane sur le port apiserver. Il doit également autoriser le trafic entrant sur son réseau de port d'écoute.

HAProxy peut être utilisé comme load balancer.

Assurez-vous que l'adresse du load balancer correspond toujours à l'adresse de **ControlPlaneEndpoint** de kubeadm.

Ajoutez les premiers nodes du control plane au load balancer et testez la connexion:

```
nc -v LOAD_BALANCER_IP PORT
```

Une erreur connection refused est attendue car l'apiserver n'est pas encore en fonctionnement. Cependant, un timeout signifie que le load balancer ne peut pas communiquer avec le node du control plane. Si un timeout survient, reconfigurez le load balancer pour communiquer avec le node du control plane.

Ajouter les nodes du control plane restants au groupe cible du load balancer.

**Configurer SSH** SSH est requis si vous souhaitez contrôler tous les nodes à partir d'une seule machine.

Activer ssh-agent sur votre machine ayant accès à tous les autres nodes du cluster:

```
eval $(ssh-agent)
```

Ajoutez votre clé SSH à la session:

```
ssh-add ~/.ssh/path_to_private_key
```

SSH entre les nodes pour vérifier que la connexion fonctionne correctement.

Lorsque vous faites un SSH sur un node, assurez-vous d'ajouter l'option **-A**:

```
ssh -A 10.0.0.7
```

Lorsque vous utilisez sudo sur n'importe quel node, veillez à préserver l'environnement afin que le SSH forwarding fonctionne:

```
sudo -E -s
```

**Control plane empilé et nodes etcd** Sur le premier node du control plane, créez un fichier de configuration appelé **kubeadm-config.yaml**:

```
apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: stable
apiServer:
  certSANs:
  - "LOAD_BALANCER_DNS"
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT"
```

kubernetesVersion doit représenter la version de Kubernetes à utiliser. Cet exemple utilise stable. **controlPlaneEndpoint** doit correspondre à l'adresse ou au DNS et au port du load balancer. Il est recommandé que les versions de kubeadm, kubelet, kubectl et kubernetes correspondent.

Assurez-vous que le node est dans un état sain puis exécuter:

```
sudo kubeadm init --config=kubeadm-config.yaml
```

Vous pouvez à présent joindre n'importe quelle machine au cluster en lancant la commande suivante sur chaque nœud en tant que root:

```
kubeadm join 192.168.0.200:6443 \
--token j04n3m.octy8zely83cy2ts \
--discovery-token-ca-cert-hash \
sha256:84938d2a22203a8e56a787ec0c6ddad7bc7dbd52ebabc62fd5f4dbea72b14d1f
```

Copiez ce jeton dans un fichier texte. Vous en aurez besoin plus tard pour joindre d'autres nodes du control plane au cluster.

Activez l'extension CNI Weave:

```
kubectl apply -f \
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

Tapez ce qui suit et observez les pods des composants démarrer:

```
kubectl get pod -n kube-system -w
```

Il est recommandé de ne joindre les nouveaux nodes du control plane qu'après l'initialisation du premier node.

Copiez les fichiers de certificat du premier node du control plane dans les autres:

Dans l'exemple suivant, remplacez **CONTROL\_PLANE\_IPS** par les adresses IP des autres nodes du control plane.

```
USER=ubuntu # customizable
CONTROL_PLANE_IPS="10.0.0.7 10.0.0.8"
for host in ${CONTROL_PLANE_IPS}; do
    scp /etc/kubernetes/pki/ca.crt "${USER}"@$host:
    scp /etc/kubernetes/pki/ca.key "${USER}"@$host:
    scp /etc/kubernetes/pki/sa.key "${USER}"@$host:
    scp /etc/kubernetes/pki/sa.pub "${USER}"@$host:
    scp /etc/kubernetes/pki/front-proxy-ca.crt "${USER}"@$host:
    scp /etc/kubernetes/pki/front-proxy-ca.key "${USER}"@$host:
    scp /etc/kubernetes/pki/etcd/ca.crt "${USER}"@$host:etcd-ca.crt
    scp /etc/kubernetes/pki/etcd/ca.key "${USER}"@$host:etcd-ca.key
    scp /etc/kubernetes/admin.conf "${USER}"@$host:
done
```

Avertissement: N'utilisez que les certificats de la liste ci-dessus. kubeadm se chargera de générer le reste des certificats avec les SANs requis pour les instances du control plane qui se joignent. Si vous copiez tous les certificats par erreur, la création de noeuds supplémentaires pourrait échouer en raison d'un manque de SANs requis.

**Étapes pour le reste des nodes du control plane** Déplacer les fichiers créés à l'étape précédente où scp était utilisé:

```
USER=ubuntu # customizable
mkdir -p /etc/kubernetes/pki/etcd
mv /home/${USER}/ca.crt /etc/kubernetes/pki/
mv /home/${USER}/ca.key /etc/kubernetes/pki/
mv /home/${USER}/sa.pub /etc/kubernetes/pki/
mv /home/${USER}/sa.key /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.crt /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.key /etc/kubernetes/pki/
```

```
mv /home/${USER}/etcd-ca.crt /etc/kubernetes/pki/etcd/ca.crt
mv /home/${USER}/etcd-ca.key /etc/kubernetes/pki/etcd/ca.key
mv /home/${USER}/admin.conf /etc/kubernetes/admin.conf
```

Ce processus écrit tous les fichiers demandés dans le dossier **/etc/kubernetes**.

Lancez kubeadm join sur ce node en utilisant la commande de join qui vous avait été précédemment donnée par kubeadm init sur le premier noeud. Ça devrait ressembler à quelque chose comme ça:

```
sudo kubeadm join 192.168.0.200:6443 \
--token j04n3m.octy8zely83cy2ts \
--experimental-control-plane \
--discovery-token-ca-cert-hash \
sha256:84938d2a22203a8e56a787ec0c6ddad7bc7dbd52ebabc62fd5f4dbea72b14d1f
```

Remarquez l'ajout de l'option --experimental-control-plane. Ce paramètre automatisé l'adhésion au control plane du cluster. Tapez ce qui suit et observez les pods des composants démarrer:

```
kubectl get pod -n kube-system -w
```

Répétez ces étapes pour le reste des nodes du control plane.

## Noeuds etcd externes

**Configurer le cluster etcd** Suivez ces instructions pour configurer le cluster etcd.

- Configurer le premier node du control plane
- Copiez les fichiers suivants de n'importe quel node du cluster etcd vers ce node.:

```
export CONTROL_PLANE="ubuntu@10.0.0.7"
+scp /etc/kubernetes/pki/etcd/ca.crt "${CONTROL_PLANE}":
+scp /etc/kubernetes/pki/apiserver-etcd-client.crt "${CONTROL_PLANE}":
+scp /etc/kubernetes/pki/apiserver-etcd-client.key "${CONTROL_PLANE}":
```

Remplacez la valeur de CONTROL\_PLANE par l'utilisateur@hostname de cette machine.

Créez un fichier YAML appelé kubeadm-config.yaml avec le contenu suivant:

```
apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: stable
apiServer:
  certSANs:
    - "LOAD_BALANCER_DNS"
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT"
etcd:
  external:
    endpoints:
      - https://ETCD_0_IP:2379
      - https://ETCD_1_IP:2379
      - https://ETCD_2_IP:2379
  caFile: /etc/kubernetes/pki/etcd/ca.crt
  certFile: /etc/kubernetes/pki/apiserver-etcd-client.crt
```

```
keyFile: /etc/kubernetes/pki/apiserver-etcd-client.key
```

La différence entre etcd empilé et externe, c'est que nous utilisons le champ external pour etcd dans

Remplacez les variables suivantes dans le modèle (template) par les valeurs appropriées pour votre cluster:

```
LOAD_BALANCER_DNS  
LOAD_BALANCER_PORT  
ETCD_0_IP  
ETCD_1_IP  
ETCD_2_IP
```

Lancez `kubeadm init --config kubeadm-config.yaml` sur ce node.

Ecrivez le résultat de la commande de join dans un fichier texte pour une utilisation ultérieure.

Appliquer le plugin CNI Weave:

```
kubectl apply -f \  
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

Pour ajouter le reste des nodes du control plane, suivez ces instructions. Les étapes sont les mêmes que pour la configuration etcd empilée, à l'exception du fait qu'un membre etcd local n'est pas créé.

Pour résumer:

- Assurez-vous que le premier node du control plane soit complètement initialisé.
- Copier les certificats entre le premier node du control plane et les autres nodes du control plane.
- Joignez chaque node du control plane à l'aide de la commande de join que vous avez enregistrée dans un fichier texte, puis ajoutez l'option –experimental-control-plane.

Tâches courantes après l'amorçage du control plane

- Installer un réseau de pod Suivez ces instructions afin d'installer le réseau de pod. Assurez-vous que cela correspond au pod CIDR que vous avez fourni dans le fichier de configuration principal.
- Installer les workers Chaque node worker peut maintenant être joint au cluster avec la commande renvoyée à partir du résultat de n'importe quelle commande kubeadm init. L'option –experimental-control-plane ne doit pas être ajouté aux nodes workers.

## Gestion des droits user, sa et mise en place de services exposés (TP)

Dans ce TP vous aurez à charge de définir un ensemble de services de type clusterIP ou nodePort ainsi que la gestion des accès des pods.

Les pods derrière les services devront pouvoir être joints et listés dans l'ensemble du cluster pour un service account donné. Vous utiliserez pour cela un clusterRole et clusterRoleBinding.

2 services devront exposés un port sur le node, 2 autres devront permettre aux pods de communiquer entre eux à l'intérieur du cluster.

# Déploiement d'un cluster Kubernetes

## Déploiement d'un master-nodeadmd, d'un master-node, d'un worker-node

Choisissez un add-on réseau pour les pods et vérifiez s'il nécessite des arguments à passer à l'initialisation de kubeadm. **Selon le fournisseur tiers que vous choisissez, vous devrez peut-être définir le –pod-network-cidr sur une valeur spécifique au fournisseur.**

Maintenant, lancez:

```
kubeadm init <args>
```

Pour plus d'informations sur les arguments de `kubeadm init`, voir le guide de référence kubeadm.

Pour une liste complète des options de configuration, voir la documentation du fichier de configuration.

Pour personnaliser les composants du control plane, y compris l'affectation facultative d'IPv6 à la sonde liveness, pour les composants du control plane et du serveur etcd, fournissez des arguments supplémentaires à chaque composant, comme indiqué dans les arguments personnalisés.

Pour lancer encore une fois `kubeadm init`, vous devez d'abord détruire le cluster.

*N.B. Si vous joignez un node avec une architecture différente par rapport à votre cluster, créez un Déploiement ou DaemonSet pour kube-proxy et kube-dns sur le node. C'est nécessaire car les images Docker pour ces composants ne prennent actuellement pas en charge la multi-architecture.*

`kubeadm init` exécute d'abord une série de vérifications préalables pour s'assurer que la machine est prête à exécuter Kubernetes. Ces vérifications préalables exposent des avertissements et se terminent en cas d'erreur. Ensuite kubeadm init télécharge et installe les composants du control plane du cluster.

Cela peut prendre plusieurs minutes. l'output devrait ressembler à:

```
[init] Using Kubernetes version: vX.Y.Z
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.e
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Activating the kubelet service
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [kubeadm-master localhost] and IPs [10.138.0.4]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [kubeadm-master localhost] and IPs [10.138.0.4]
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
```

```

[certs] apiserver serving cert is signed for DNS names [kubeadm-master kubernetes kubernetes.default]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from directory /etc/kubernetes/manifests
[apiclient] All control plane components are healthy after 31.501735 seconds
[uploadconfig] storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config-X.Y" in namespace kube-system with the configuration for node kubeadm-master
[patchnode] Uploading the CRI Socket information "/var/run/dockershim.sock" to the Node API object "kubeadm-master"
[mark-control-plane] Marking the node kubeadm-master as control-plane by adding the label "node-role.kubernetes.io/control-plane"
[mark-control-plane] Marking the node kubeadm-master as control-plane by adding the taints [node-role.kubernetes.io/control-plane:NoSchedule]
[bootstrap-token] Using token: <token>
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstraptoken] configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for nodes to self-register
[bootstraptoken] configured RBAC rules to allow the csrapprover controller automatically approve CSRs from tokens
[bootstraptoken] configured RBAC rules to allow certificate rotation for all node client certificates
[bootstraptoken] creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

```

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

```

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

```

You should now deploy a pod network to the cluster.

Run "`kubectl apply -f [podnetwork].yaml`" with one of the options listed at:  
<https://kubernetes.io/fr/docs/concepts/cluster-administration/addons/>

You can now join any number of machines by running the following on each node as root:

```

kubeadm join <master-ip>:<master-port> --token <token> --discovery-token-ca-cert-hash sha256:<hash>

```

Pour que kubectl fonctionne pour votre utilisateur non root, exécuter ces commandes, qui font également partie du résultat de la commande `kubeadm init` :

```

mkdir -p $HOME/.kube

```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Alternativement, si vous êtes root, vous pouvez exécuter:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Faites un enregistrement du retour de la commande `kubeadm join` que `kubeadm init` génère. Vous avez besoin de cette commande pour joindre des nœuds à votre cluster.

Le token est utilisé pour l'authentification mutuelle entre le master et les nœuds qui veulent le rejoindre. Le token est secret. Gardez-le en sécurité, parce que n'importe qui avec ce token peut ajouter des nœuds authentifiés à votre cluster. Ces tokens peuvent être listés, créés et supprimés avec la commande `kubeadm token`.

## Mise en place du dashboard et du réseau

Vous devez installer un add-on réseau pour pod afin que vos pods puissent communiquer les uns avec les autres.

Le réseau doit être déployé avant toute application. De plus, CoreDNS ne démarra pas avant l'installation d'un réseau. `kubeadm` ne prend en charge que les réseaux basés sur un CNI (et ne prend pas en charge `kubenet`).

Plusieurs projets fournissent des réseaux de pod Kubernetes utilisant CNI, dont certains supportent les network policies. Allez voir la page des add-ons pour une liste complète des add-ons réseau disponibles.

**Le support IPv6 a été ajouté dans CNI v0.6.0. CNI bridge et local-ipam sont les seuls plug-ins de réseau IPv6 pris en charge dans Kubernetes version 1.9.**

**Notez que `kubeadm` configure un cluster sécurisé par défaut et impose l'utilisation de RBAC. Assurez-vous que votre manifeste de réseau prend en charge RBAC.**

Veuillez également à ce que votre réseau Pod ne se superpose à aucun des réseaux hôtes, car cela pourrait entraîner des problèmes. Si vous constatez une collision entre le réseau de pod de votre plug-in de réseau et certains de vos réseaux hôtes, vous devriez penser à un remplacement de CIDR approprié et l'utiliser lors de `kubeadm init` avec `--pod-network-cidr` et en remplacement du YAML de votre plugin réseau.

Vous ne pouvez installer qu'un seul réseau de pod par cluster.

Choisissez-en un parmi les suivants:

- Calico
- Canal
- Cilium
- Flannel
- Kube-router
- Romana
- Weave Net
- JuniperContrail/TungstenFabric

Pour notre part nous ferons le choix de Calico qui est majoritairement utilisé.

Pour plus d'informations sur l'utilisation de Calico, voir Guide de démarrage rapide de Calico sur Kubernetes, Installation de Calico pour les netpols (network policies) et le réseau, ainsi que d'autres resources liées à ce sujet.

[Quickstart Calico on K8s](#)

[Calico Home](#)

Pour que Calico fonctionne correctement, vous devez passer **-pod-network-cidr = 192.168.0.0 / 16 à kubeadm init ou mettre à jour le fichier calico.yaml pour qu'il corresponde à votre réseau de Pod.**

Notez que Calico fonctionne uniquement sur amd64, arm64, ppc64le et s390x.

```
kubectl apply -f \
https://docs.projectcalico.org/v3.8/manifests/calico.yaml
```

**Une fois qu'un réseau de pod a été installé, vous pouvez vérifier qu'il fonctionne en vérifiant que le pod CoreDNS est en cours d'exécution** dans l'output de `kubectl get pods --all-namespaces`. Et une fois que le pod CoreDNS est opérationnel, vous pouvez continuer en joignant vos nœuds.

Si votre réseau ne fonctionne pas ou si CoreDNS n'est pas en cours d'exécution, vérifiez notre documentation de dépannage.

Par défaut, votre cluster ne déploie pas de pods sur le master pour des raisons de sécurité. Si vous souhaitez pouvoir déployer des pods sur le master, par exemple, pour un cluster Kubernetes mono-machine pour le développement, exécutez:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
node "test-01" untainted
taint "node-role.kubernetes.io/master:" not found
taint "node-role.kubernetes.io/master:" not found
```

Cela supprimera la marque **node-role.kubernetes.io/master** de tous les nodes qui l'ont, y compris du node master, ce qui signifie que le scheduler sera alors capable de déployer des pods partout.

Les nodes sont ceux sur lesquels vos workloads (conteneurs, pods, etc.) sont exécutés. Pour ajouter de nouveaux nodes à votre cluster, procédez comme suit pour chaque machine:

- SSH vers la machine
- Devenir root (par exemple, `sudo su-`)
- Exécutez la commande qui a été récupérée sur l'output de `kubeadm init`.

Par exemple:

```
kubeadm join --token <token> <master-ip>:<master-port> \
--discovery-token-ca-cert-hash sha256:<hash>
```

Si vous n'avez pas le jeton, vous pouvez l'obtenir en exécutant la commande suivante sur le node master:

```
kubeadm token list
```

Par défaut, les jetons expirent après 24 heures. Si vous joignez un node au cluster après l'expiration du jeton actuel, vous pouvez créer un nouveau jeton en exécutant la commande suivante sur le node master:

```
kubeadm token create
```

```
5didvk.d09sbcov8ph2amjw
```

Si vous n'avez pas la valeur **-discovery-token-ca-cert-hash**, vous pouvez l'obtenir en exécutant la suite de commande suivante sur le node master:

```
openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | \
openssl rsa -pubin -outform der 2>/dev/null | \
openssl dgst -sha256 -hex | sed 's/^.* //'
```

```
8cb2de97839780a412b93877f8507ad6c94f73add17d5d7058e91741c9d5ec78
```

```
[preflight] Running pre-flight checks
```

```
... (log output of join workflow) ...
```

```
Node join complete:
* Certificate signing request sent to master and response
  received.
* Kubelet informed of new secure connection details.
```

```
Run 'kubectl get nodes' on the master to see this machine join.
```

Quelques secondes plus tard, vous remarquerez ce node dans l'output de `kubectl get nodes`.

Enfin, nous allons installer la dashboard UI:

```
kubectl apply -f \
https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml
```

## Déploiement d'un cluster (TP)

Dans ce TP vous aurez à charge de spawner un cluster avec kubeadm/eksctl ou les 2 sur un environnement AWS.

Nous devrons au préalable nous assurer de l'existence de 3 VM accessibles.

**Installation manuelle avec kubeadm** Suivez les étapes de la documentation ci-dessous [Create cluster from scratch with kubeadm](#)

Puis, valider le bon fonctionnement suite à l'installation.

```
controlplane $ kubectl cluster-info
```

```
Kubernetes master is running at https://172.17.0.10:6443
KubeDNS is running at
https://172.17.0.10:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

```
# Validate resources visibility after installation
kubectl get pods --all-namespaces -o wide
```

Enfin, détruisez le cluster:

Pour annuler ce que kubeadm a fait, vous devez d'abord drainer le node et assurez-vous que le node est vide avant de l'arrêter.

En communiquant avec le master en utilisant les informations d'identification appropriées, exécutez:

```
kubectl drain <node name> --delete-local-data --force --ignore-daemonsets
kubectl delete node <node name>
```

Ensuite, sur le node en cours de suppression, réinitialisez l'état de tout ce qui concerne kubeadm:

```
kubeadm reset
```

Le processus de réinitialisation ne réinitialise pas et ne nettoie pas les règles iptables ni les tables IPVS. Si vous souhaitez réinitialiser iptables, vous devez le faire manuellement:

```
iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X
```

Si vous souhaitez réinitialiser les tables IPVS, vous devez exécuter la commande suivante:

```
ipvsadm -C
```

Si vous souhaitez recommencer il suffit de lancer `kubeadm init` ou `kubeadm join` avec les arguments appropriés.

**Installation sur AWS EKS avec eksctl** Plusieurs types de configuration pour les nodes qui seront spawn pour l'installation du cluster:

- Fargate - Linux - Sélectionnez ce type de node si vous souhaitez exécuter des applications Linux sur AWS Fargate. Fargate est un moteur de calcul sans serveur qui vous permet de déployer des pods Kubernetes sans gérer les instances Amazon EC2.
- nodes gérés - Linux - sélectionnez ce type de node si vous souhaitez exécuter des applications Amazon Linux sur des instances Amazon EC2. Bien que cela ne soit pas abordé dans ce guide, vous pouvez également ajouter des nodes autogérés et Bottlerocket Windows à votre cluster.

## Installation d'eksctl

```
curl --silent --location \
"https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" \
| tar xz -C /tmp

sudo mv /tmp/eksctl /usr/local/bin
```

## Création d'un cluster avec eksctl sur des nodes de type fargate

```
eksctl create cluster --name kolorado --region us-east-2 --fargate

2021-12-14 02:35:13 [i] eksctl version 0.76.0
2021-12-14 02:35:13 [i] using region us-east-2
2021-12-14 02:35:13 [i] setting availability zones to [us-east-2a us-east-2b us-east-2c]
2021-12-14 02:35:13 [i] subnets for us-east-2a - public:192.168.0.0/19 private:192.168.96.0/19
2021-12-14 02:35:13 [i] subnets for us-east-2b - public:192.168.32.0/19 private:192.168.128.0/19
2021-12-14 02:35:13 [i] subnets for us-east-2c - public:192.168.64.0/19 private:192.168.160.0/19
2021-12-14 02:35:13 [i] using Kubernetes version 1.21
2021-12-14 02:35:13 [i] creating EKS cluster "kolorado" in "us-east-2" region with Fargate profile
2021-12-14 02:35:13 [i] if you encounter any issues, check CloudFormation console or try 'eksctl utils update-cluster-logging --enable-types=...'
2021-12-14 02:35:13 [i] CloudWatch logging will not be enabled for cluster "kolorado" in "us-east-2"
2021-12-14 02:35:13 [i] you can enable it with 'eksctl utils update-cluster-logging --enable-types=...'
2021-12-14 02:35:13 [i] Kubernetes API endpoint access will use default of {publicAccess=true, priv...
2021-12-14 02:35:13 [i]

2 sequential tasks: { create cluster control plane "kolorado",
  2 sequential sub-tasks: {
    wait for control plane to become ready,
    create fargate profiles,
  }
}
2021-12-14 02:35:13 [i] building cluster stack "eksctl-kolorado-cluster"
2021-12-14 02:35:14 [i] deploying stack "eksctl-kolorado-cluster"
2021-12-14 02:35:44 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:36:14 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:37:15 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:38:15 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:39:16 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:40:16 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:41:16 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:42:17 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:43:17 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:44:18 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:45:18 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:46:19 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:47:19 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:48:19 [i] waiting for CloudFormation stack "eksctl-kolorado-cluster"
2021-12-14 02:50:22 [i] creating Fargate profile "fp-default" on EKS cluster "kolorado"
2021-12-14 02:54:41 [i] created Fargate profile "fp-default" on EKS cluster "kolorado"
2021-12-14 02:57:13 [i] "coredns" is now schedulable onto Fargate
2021-12-14 02:58:17 [i] "coredns" is now scheduled onto Fargate
2021-12-14 02:58:17 [i] "coredns" pods are now scheduled onto Fargate
2021-12-14 02:58:17 [i] waiting for the control plane availability...
2021-12-14 02:58:17 [OK] saved kubeconfig as "/home/pbackz/.kube/config"
2021-12-14 02:58:17 [i] no tasks
2021-12-14 02:58:17 [OK] all EKS cluster resources for "kolorado" have been created
2021-12-14 02:58:19 [i] kubectl command should work with "/home/pbackz/.kube/config", try 'kubectl g...
2021-12-14 02:58:19 [OK] EKS cluster "kolorado" in "us-east-2" region is ready

# Validate resources visibility after installation
```

```

kubectl get pods --all-namespaces -o wide

# Show config
kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://D574D5657EED6B65990285B60D420EBB.gr7.us-east-2.eks.amazonaws.com
    name: kolorado.us-east-2.eksctl.io
contexts:
- context:
    cluster: kolorado.us-east-2.eksctl.io
    user: kolo@kolorado.us-east-2.eksctl.io
    name: kolo@kolorado.us-east-2.eksctl.io
current-context: kolo@kolorado.us-east-2.eksctl.io
kind: Config
preferences: {}
users:
- name: kolo@kolorado.us-east-2.eksctl.io
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1alpha1
      args:
      - eks
      - get-token
      - --cluster-name
      - kolorado
      - --region
      - us-east-2
      command: aws
      env:
      - name: AWS_STS_REGIONAL_ENDPOINTS
        value: regional
      interactiveMode: IfAvailable
      provideClusterInfo: false

```

Puis détruire le cluster:

```

# Delete cluster
eksctl delete cluster --name kolorado --region us-east-2
eksctl examples

```

## Annexe A: Kubectl Command CheatSheet

Pour accéder aux contextes de votre KUBECONFIG:

```
kubectl config get-contexts
```

Pour changer de contexte:

```
kubectl config use-context minikube
```

Pouyr récupérer le nom des containers running sur le pod:

```
kubectl get pod MYPOD -o 'jsonpath={.spec.containers[*].name}'
```

Pour récupérer la valeur de la clé d'un secret:

```
kubectl -n mynamespace get secret MYSECRET \
-o 'jsonpath={.data.DB_PASSWORD}' | base64 -d
SuperSecretPassword
```

Si vous n'avez pas la commande base64 vous pouvez utiliser go-template:

```
kubectl -n mynamespace get secret MYSECRET \
-o 'go-template={{.data.DB_PASSWORD | base64decode}}'
```

Un exemple de filtre de recherche dans jsonpath:

```
kubectl get pod nginx \
-o 'jsonpath={.spec.containers[?(@.name=="nginx")].image}'
nginx:1.9.1
```

Pour créer un secret depuis vos credentials Docker actuels pour récupérer les images depuis un registry privé:

```
kubectl create secret generic SECRETNAME \
--from-file=.dockerconfigjson=$HOME/.docker/config.json \
--type=kubernetes.io/dockerconfigjson
```

Pour forward le port 8080 sur le port local 8888 du poste de travail:

```
kubectl port-forward MYPOD 8888:8080
```

Pour tester des règles RBAC:

```
kubectl --as=system:serviceaccount:MYNS:MYSA auth can-i get configmap/MYCM
yes
```