S5.B.01 Phase 4
Déploiement de services

Maxence Lagourgue

16 janvier 2026

Table des matières

I Introduction

I.A Outils

Dans cette partie, les outils utilisés seront :

- Rancher pour la gestion des clusters
- RKE2 pour la mise en œuvre Kubernetes des nœuds de travail
- k3s pour le cluster qui accueillera le déploiement Rancher
- kubectl pour la gestion des ressources
- Helm pour la gestion des applications
- Calico pour la gestion du réseau et son outil calicoctl

Plus tard, si nous avons le temps, nous utiliserons Ansible pour automiser la chaîne de production Rancher avec une pipeline Gitlab CI/CD.

I.B Machines

Les machines utilisées au cours de ce projet seront :

Machine	IP	Kubernetes	Rôle	Service
Applicatif	10.0.1.3	k3s	Master	Rancher Dashboard
K8SA2	10.0.1.4	RKE2 Cluster	Master	Gitlab, Helm, Kube-System
K8SB2	10.0.1.5	RKE2 Cluster	Worker	Gitlab, Docker Registry, local-path-provisioner, Nailloux-Front
K8SC2	10.0.1.6	RKE2 Cluster	Worker	Docker-Registry, Gitlab, Gitlab-runner, Database

I.C Configuration générale

Afin de pouvoir déployer efficacement notre cluster, nous devons configurer les machines correctement.

I.C.1 Nom de domaines

La première configuration réside dans le fait que rancher utilise un ingress, et donc pour accéder au service et aux endpoints API, il faut utiliser le nom de domaine et donc être en capacité de le résoudre.

I.C.1.a Fichier temporaire /etc/hosts

Dans un premier temps, nous ferons cela via les fichiers /etc/hosts.

Listing I.1 – /etc/hosts

```
10.0.1.3        rancher.rancher
10.0.1.4        master.rancher
10.0.1.5        worker.rancher
10.0.1.6        gitlab.rancher nailloux.gitlab.com nailloux.registry.com
```

I.C.1.b Zone DNS Unbound

Afin d'avoir une solution qualitative et évolutive, il est mieux de créer une zone DNS dans le service DNS approprié.
Dans le fichier principal on ajoute :

```
include: "/etc/unbound/unbound.conf.d/*.conf"
```

On crée un fichier nommée rancher.conf dans unbound.conf.d

Listing I.2 – /etc/unbound/unbound.conf.d/rancher.conf

```
server:
    local-zone: "rancher." static
    local-zone: "nailloux.gitlab.com." redirect
    local-zone: "registry.com." static

    local-data: "dns.rancher. IN A 10.0.1.2"
    local-data: "rancher.rancher. IN A 10.0.1.3"
    local-data: "master.rancher. IN A 10.0.1.4"
    local-data: "worker.rancher. IN A 10.0.1.5"
    local-data: "gitlab.rancher. IN A 10.0.1.6"
#   local-data: "nailloux.gitlab.com. IN CNAME gitlab.rancher."
    local-data: "nailloux.gitlab.com. IN A 10.0.1.4"
#   local-data: "nailloux.registry.com. IN CNAME gitlab.rancher."
    local-data: "nailloux.registry.com. IN A 10.0.1.4"
```

A la base, je souhaitais faire des Canonical NAME afin d'avoir une configuration flexible. Malheureusement le déploiement CoreDNS de Kubernetes supporte mal cette configuration et refuse de faire des requêtes DNS récursives.
J'ai donc dû changer cela.

I.C.2 Contrôle des ressources

Kubernetes, de part la multitude d'applications que nous allons déployer sera très énergivore.
Au vu de la configuration des VMs et de la désactivation du guest agent, nous ne pouvons pas nous fier aux données affichées par Proxmox en termes de RAM.
Je conseille donc l'outil simple en ligne de commande fastfetch.

Listing I.3 – Installation de fastfetch

```
wget https://github.com/fastfetch-cli/fastfetch/releases/download/2.56.1/fastfetch-linux-
    amd64.deb && dpkg -i fastfetch-linux-amd64.deb
```

II Installation de Rancher dans un cluster k3s

Pour utiliser Rancher, plusieurs méthodes d'installation s'offrent à nous. L'une avec docker, l'autre en tant que noeud Kubernetes. Les autres installations reposent sur l'utilisation d'un Cloud Provider ainsi que Terraform donc inutile dans notre cas.
Exemple de tutorial : Tutorial Rancher 2025
Dans notre cas, nous avons décidé de l'installer en tant que déploiement k3s.

II.A Installation de k3s

Listing II.1 – Installation de k3s

```
curl -sfL https://get.k3s.io | INSTALL_K3S_VERSION="v1.34.3+k3s1" sh -s - server --resolv-conf /run/systemd/resolve/resolv.conf
```

Il est possible de définir les paramètres suivants mais ils sont susceptibles de générer des bugs car nous avons deux interfaces sur les VMs et que celle locale n'est pas celle par défaut. Il vaut donc mieux ne pas définir ces options.

```
--bind-address 10.0.1.3 --advertise-address 10.0.1.3 --node-ip 10.0.1.3
```

II.A.1 Accès distant au cluster (Optionnel)

Si l'on veut se connecter au cluster avec un accès distant (sans SSH), on peut faire les manips suivantes. Je ne les ai pas testées mais je les ai mises au cas où.
<IP_OF_LINUX_MACHINE> est l'IP de la machine distante sur laquelle se trouve le cluster.

```
scp root@<IP_OF_LINUX_MACHINE>:/etc/rancher/k3s/k3s.yaml ~/.kube/config
```

```
nano ~/.kube/config
```

II.A.2 Installation d'helm

Helm est un gestionnaire de paquets (comme apt, pacmen, brew) pour Kubernetes. Il sera utile par la suite pour installer des applications en réduisant la complexité et surtout les erreurs.

```
sudo apt-get install curl gpg apt-transport-https --yes

curl -fsSL https://packages.buildkite.com/helm-linux/helm-debian/gpgkey | gpg --dearmor |
sudo tee /usr/share/keyrings/helm.gpg > /dev/null

echo "deb [signed-by=/usr/share/keyrings/helm.gpg]
https://packages.buildkite.com/helm-linux/helm-debian/any/ any main" | sudo tee
/etc/apt/sources.list.d/helm-stable-debian.list
```

```
sudo apt-get update
sudo apt-get install helm
```

II.A.3 Installation de kubectl

Kubectl est l'outil de gestion de ressources kubernetes.

```
sudo apt-get install -y apt-transport-https ca-certificates curl gnupg

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/Release.key | sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-apt-keyring.gpg

sudo chmod 644 /etc/apt/keyrings/kubernetes-apt-keyring.gpg

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /' | sudo tee
/etc/apt/sources.list.d/kubernetes.list
sudo chmod 644 /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update
sudo apt-get install -y kubectl
```

II.A.4 Installation de Calicoctl

Calicoctl est l'outil pour gérer la ressource réseau calico.

```
curl -L https://github.com/projectcalico/calico/releases/download/v3.30.4/calicoctl-linux-
    amd64 -o calicoctl

chmod +x ./calicoctl

mv ./calicoctl /usr/bin/calicoctl
```

II.A.5 Déploiement de l'application Rancher sur le cluster k3s

Dans cette partie, nous allons voir comment installer Rancher sur le cluster k3s.
Cela se fait avec Helm, comme cité précédemment.
<Hostname> correspond au nom de domaine utilisé pour contacter le pod rancher. <password> correspond au mot de passe pour se connecter.

```
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml



kubectl create namespace cattle-system
kubectl config set-context --current --namespace=cattle-system

kubectl apply -f
https://github.com/cert-manager/cert-manager/releases/download/v1.19.2/cert-manager.crds.yaml
```

```
helm repo add rancher-latest https://releases.rancher.com/server-charts/latest

helm repo add jetstack https://charts.jetstack.io

helm repo update

helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace

helm install rancher rancher-latest/rancher \
  --namespace cattle-system \
  --set hostname=rancher.rancher \
  --set replicas=1 \
  --set bootstrapPassword=<password> \
  --set additionalTrustedCAs=true
```

Il faut maintenant attendre car l'installation nécessite quelques minutes. On peut vérifier avec `kubectl␣get␣pods␣-n␣cattle-system`
Une fois fait, on se connecte à la page et on récupère le mot de passe :

```
kubectl get secret --namespace cattle-system bootstrap-secret -o
go-template='{{.data.bootstrapPassword|base64decode}}{{"\n"}}'
```

On définit un nouveau mot de passe (si demandé) qui est `qJHiA@wwaagi46U`.

II.A.6 Pour arrêter/pauser Rancher

```
kubectl scale --replicas=0 deployment/rancher -n cattle-system
```

Cela permet d'arrêter temporairement le pod Rancher.

II.A.7 Suppression de rancher

Etant donné que nous avons installé Rancher avec helm, la désinstallation se fait aussi avec.

```
helm delete --namespace cattle-system rancher
```

II.A.8 Ajout d'un certificat Root

```
kubectl -n cattle-system create secret generic tls-ca-additional --from-file=ca-additional
    .pem=./ca-additional.pem
```

II.B Réinitialisation du cluster

Pour revenir à l'état 0 du cluster, il est possible de :

```
rm -rf /var/lib/rancher/k3s/server/db/etcd
/usr/local/bin/k3s-killall.sh
systemctl restart k3s.service
```

III Création d'un cluster RKE2

Une fois l'installation de rancher faite, il faut maintenant créer un cluster et enrôler des machines afin d'avoir de la *workforce*.

III.A Enrôlement de machines

III.A.1 Variables d'environnement

Parce que rancher nécessite des variables d'environnements qui ne sont pas forcément définis de base :

```
echo "CRI_CONFIG_FILE=/var/lib/rancher/rke2/agent/etc/crictl.yaml
CONTAINERD_ADDRESS=unix:///run/k3s/containerd/containerd.sock
KUBECONFIG=/etc/rancher/rke2/rke2.yaml" >> /etc/environment

echo "export PATH=$PATH:/var/lib/rancher/rke2/bin" > /etc/profile.d/rancher.sh
```

III.A.2 Ajout d'une mâchine Master

Pour cela catil faut aller dans la section **Clusters** → **<Cluster>** → **Registration**

Listing III.1 – Machine master

```
curl --insecure -fL https://rancher.rancher/system-agent-install.sh | sudo  sh -s - --
    server https://rancher.rancher --label 'cattle.io/os=linux' --token
    b6rh9zgx8m8jgwg6q6nm7h5frb7d6v87wgt8scppljfvqbvjt85kqk --ca-checksum
    b5ede295e2fdd2b453ae8cee700b60185f393914281b4bc90008c1e3f4eb8e5a --etcd --controlplane
    --worker --address 10.0.1.4 --internal-address 10.0.1.4
```

III.A.3 Ajout d'une machine Worker only

Listing III.2 – Machine worker

```
curl --insecure -fL https://rancher.rancher/system-agent-install.sh | sudo  sh -s - --
    server https://rancher.rancher --label 'cattle.io/os=linux' --token
    tvg69x5vkm9szzlzsj6qqkx7wggzrgvt2grc755nth29h2ncjbgthz --ca-checksum
    fdc9c50ea58442994213e96883b6a5ca39227fc7d4116e60fa1026c123f56583 --worker --address
    10.0.1.5 --internal-address 10.0.1.5
```

Si souci, il faut aller voir la section Troubleshooting.

IV Replication du Noeud Master

Partie non faite, bloat d'IA

```
In the context of RKE2/K3s (the tech stack you are currently using), "replicating" a
    master node means adding additional Control Plane + Etcd nodes to form a High
    Availability (HA) cluster.

Since you are working across different subnets, the process requires a specific
    configuration to ensure the nodes can communicate and that the certificates are valid
    for your cross-subnet IPs.
1. Requirements for Master Replication

    An odd number of masters: To maintain "Quorum" (the ability to make decisions), you
        need 3, 5, or 7 master nodes. 3 is the standard for HA.

    The Shared Token: You need the token from your first master (found at /var/lib/rancher
        /rke2/server/node-token or /var/lib/rancher/k3s/server/node-token).

    A Fixed Registration Address: As discussed, this should be a DNS name pointing to your
         master IPs.

2. Configure the First Master (Initial Node)

If you already installed the first master, ensure its /etc/rancher/rke2/config.yaml (or
    k3s) has the tls-san parameter. This ensures the SSL certificates allow the "
    Registration DNS" you'll use.
YAML

token: my-shared-secret-token
tls-san:
  - k8s-api.example.com  # The DNS name for your masters
  - 10.0.1.3             # IP of Master 1
  - 10.0.2.3             # IP of Master 2 (Subnet B)

3. Join the Second Master (The Replication)

On your server in the second subnet, install the same version of RKE2/K3s, but point it to
     the first master.

The config.yaml for Master #2:
YAML

server: https://k8s-api.example.com:9345  # Point to the first node
token: my-shared-secret-token             # Must match node #1
```

```
tls-san:
  - k8s-api.example.com
# Important: Ensure this node knows where the "real" DNS is
resolv-conf: "/run/systemd/resolve/resolv.conf"

4. Handling the "No Load Balancer" Problem

Since you cannot use a load balancer across your subnets, you must handle the Initial Join
     carefully.

    Direct Join: For the very first join of Master #2, you can temporarily point the
        server parameter directly to the IP of Master #1: server: https://10.0.1.3:9345.

    DNS Update: Once Master #2 is up, add its IP to your k8s-api.example.com DNS record.

    Master #3: When you join the third master, point its server address to the DNS name.
        It will now be able to reach either Master #1 or Master #2 to join.

5. Verification

Once the replication is complete, check that all masters are participating in the etcd
    cluster and the control plane:
Bash

# Check node status
kubectl get nodes

# Check etcd health (RKE2 example)
/var/lib/rancher/rke2/bin/kubectl \
  --kubeconfig /etc/rancher/rke2/rke2.yaml \
  get componentstatuses

Summary of HA across Subnets

    Etcd: Will automatically replicate data from Master 1 to Master 2 over the network.

    API Server: Will be active on both nodes.

    Workers: Will automatically switch between Master 1 and Master 2 if one subnet goes
        offline.

Would you like the specific shell commands to extract the node-token from your first
    master to prepare for the join?
```

V Deploiement de l'application

V.A Changement de spécification avec migration Docker à Kubernetes

Dans la phase 2, nous avions déployé une application docker au moyen d'un docker-compose.yml.
Durant cette phase, nous devons la déployer avec Kubernetes, ce qui nécessite d'utiliser une spécification des ressources différentes à l'aide de manifests.yml.
Nous pouvons effectuer une première approche de cette traduction avec l'outil kompose qui s'installe comme ceci :

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.38.0/kompose-linux-
    amd64 -o kompose
chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

On convertit donc notre docker-compose avec la commande suivante

```
kompose convert docker-compose.yaml
```

De là va se créer un répertoire contenant tous les manifests mobilisés pour déployer l'application. Il faudra bien évidemment les revisiter car ce n'est pas un outil magique. Notamment au niveau des volumes.

V.B Stockage local de volumes Kubernetes

Dans notre situation, nous utilisons une solution locale avec un cluster local. Les données des volumes sont donc stockées sur les systèmes de fichiers des machines. Pour cela, il faut faire appel à un Provider capable de provisionner et de répondre à cette demande.
C'est un déploiement Kubernetes à installer dans le cluster comme ceci :

```
kubectl apply -f https://raw.githubusercontent.com/rancher/local-path-provisioner/master/
    deploy/local-path-storage.yaml

kubectl patch storageclass local-path -p '{"metadata": {"annotations":{"storageclass.
    kubernetes.io/is-default-class":"true"}}}'
```

La commande kubectl patch permet de définir la classe de stockage comme par défaut.

VI Déploiement de composants annexes

Dans le même temps que le déploiement de l'application du club photo nailloux, l'infrastructure de l'entreprise nécessite d'autres services, qui seront détaillés et déployés ici, afin de répondre aux besoins de développement par exemple.
Dans cette section nous verrons comment déployer Gitlab, Gitlab-Runner, un Docker Registry et tout cela via Kubernetes et le cluster RKE2.

VI.A Gitlab

VI.A.1 Phase 1 : Creation des volumes

Comme vu plus haut, dans Kubernetes il existe ce qu'on appelle des **Persistent Volume Claims**, qui ont pour vocation de se substituer aux volumes docker.

Néanmoins, ils possèdent quelques spécificités que n'ont pas les volumes docker, notamment que le volume docker monte un dossier hôte vers le conteneur alors que les volumes Kubernetes, sont initialisés totalement vides.

Donc là où Docker permettait d'avoir des volumes avec des fichiers, Kubernetes ne permet pas cela.

Volume	Taille	Point de montage	Rôle
gitlab-etc	2Gi	/etc/gitlab	Configuration
gitlab-opt	20Gi	/var/opt/gitlab	Données de fonctionnement Gitlab
gitlab-log	5Gi	/var/log/gitlab	Fichiers de journalisation

VI.A.2 Phase 2 : Creation de la ConfigMap

Pour définir le nom de domaine qu'on va utiliser pour accéder au serveur Gitlab, ainsi que d'autres paramètres, on créé une **ConfigMap** avec la procédure suivante :
• Cluster → Resources → ConfigMaps.
On créé une nouvelle configuration nommée **gitlab-config** et on définit une clé **gitlab.rb** avec comme valeur :

```
    external_url 'http://nailloux.gitlab.com'
    gitlab_rails['time_zone'] = 'Europe/Paris'
```

Ce fichier sera utilisé par gitlab au moment de son initialisation.

VI.A.3 Phase 3 : Deploiement des pods

Dans la section • Workloads -> Deployment on créé un Déploiement comme suivant :

VI.A.3.a Variables d'environnments

On peut aussi passer des paramètres via les variables d'environnements avec

```
GITLAB_ROOT_EMAIL="admin@example.com" GITLAB_ROOT_PASSWORD="strongpassword"
```

VI.A.3.b Conteneur

Nom	Image
Gitlab	gitlab/gitlab-ce :latest

VI.A.3.c Ouverture de ports

Il faut ensuite ouvrir des ports, qui impliqueront la création d'un service, en l'occurence ClusterIP et NodePort.

Type	Nom	Port	Protocole
ClusterIP	http	80	TCP
NodePort	ssh	22 :22	TCP

VI.A.3.d Storage (Volume Mounts)

Il faut ensuite monter les volumes créés et la ConfigMap donc les pods.

Volume	Type	Point de montage
gitlab-etc	PVC	/etc/gitlab
gitlab-opt	PVC	/var/opt/gitlab
gitlab-log	PVC	/var/log/gitlab
gitlab-config	ConfigMap	/etc/gitlab/gitlab.rb

VI.A.3.e Security Context

Il faut savoir que Rancher, par défaut, cloisonne l'utilisateur chargé de lancer le service. Cela mène à plusieurs problèmes de permissions si le service nécessite de lire des fichiers protégés.
C'est le cas ici avec Gitlab. Dans la section Security Context, il faut donc modifier les paramètres suivantes

Privileged	Run as User
True	0 (Root)

VI.A.4 Phase 4 : Attente

GitLab prend 5 à 10 minutes pour démarrer la première fois.
Il ne faut donc surtout pas ajouter de "Health Check" car cela mettra le pod/déploiement en défaut.
On peut accéder aux logs avec

```
kubectl logs -f deployment/gitlab
```

VI.A.5 Phase 5 : Création d'une redirection de paquets avec un Ingress

Dans • Service Discovery → Ingresses → Create :

- Host : nailloux.gitlab.com
- Path : / → Service : gitlab → Port : 80.

On peut définir le HTTPS pour l'Ingress dans la section Certificates en ajoutant un.
On ajoute un certificat en créant un Secret de type kubernetes.io/tls avec une clé tls.key et tls.crt contenant respectivement en base64 la clé et le certificat.

VI.A.6 Phase 6 : Récupération du mot de passe

Une fois que le serveur Gitlab est lancé, (environ 5 minutes), on peut récupérer le mot de passe généré par défaut comme ceci :

```
kubectl exec gitlab-54d7466479-vcd9w -- cat /etc/gitlab/initial_root_password
# WARNING: This password is only valid if ALL of the following are true:
# •          You set it manually via the GITLAB_ROOT_PASSWORD environment variable
#            OR the gitlab_rails['initial_root_password'] setting in /etc/gitlab/gitlab.rb
# •          You set it BEFORE the initial database setup (typically during first
    installation)
# •          You have NOT changed the password since then (via web UI or command line)
#
#          If this password doesn't work, reset the admin password using:
#          https://docs.gitlab.com/security/reset_user_password/#reset-the-root-password

Password: JKk4H/BOVuXOFgi899ZRsN4HBnLzmWLQAveHOkmtAyE=

# NOTE: This file is automatically deleted after 24 hours on the next reconfigure run.
```

Pour interagir avec le pod :

```
kubectl exec -it gitlab-54d7466479-vcd9w -- bash
```

Vous venez donc d'installer et de déployer une application Gitlab.

VI.B Gitlab avec Helm

Précédemment, a été évoqué l'installation manuelle de Gitlab avec des manifests. Cependant, cette installation s'est révélée complexe et propice aux erreurs. A donc été prise comme décision d'utiliser Helm pour installer Gitlab, qui a été de suite plus efficace et moins complexe.
Tout d'abord a été définie un fichier values.yaml, qui est utilisée pour spécifier des paramètres à Helm lors de l'installation d'un paquet.

Listing VI.1 – values.yaml

```
global:
  hosts:
    domain: nailloux.gitlab.com  # Your domain
    gitlab:
      name: nailloux.gitlab.com
    externalIP: 10.0.1.4
  ingress:
    configureCertmanager: true
  # If you want to use your existing external PostgreSQL/Redis, configure here.
```

```
    # Otherwise, it will install them inside the cluster.

certmanager-issuer:
  email: root@nailloux.lan

gitlab:
  webservice:
    # This solves your previous "unmapped file" error
    extraVolumes:
      - name: dshm
        emptyDir:
          medium: Memory
    extraVolumeMounts:
      - name: dshm
        mountPath: /dev/shm
    resources:
      requests:
        memory: 4Gi
      limits:
        memory: 8Gi
```

Une fois le fichier créé, on peut exécuter les commandes suivantes pour récupérer Gitlab et l'installer sur le namespace gitlab.

```
helm repo add gitlab https://charts.gitlab.io/

kubectl create namespace gitlab

helm repo update
helm upgrade --install gitlab gitlab/gitlab \
  --timeout 600s \
  --set global.hosts.externalIP=10.0.1.4 \
  --set global.edition=ce \
  --namespace gitlab \
  -f values.yaml
```

Une fois l'installation faite, on récupère le mot de passe dans le secret associé.

```
kubectl get secret <name>-gitlab-initial-root-password -ojsonpath='{.data.password}' |
    base64 --decode ; echo
```

On remarque donc que l'installation de gitlab via Helm est plus simple, mais aussi plus complète.

VI.C Gitlab runner

Maintenant que nous avons installé Gitlab et afin de pouvoir exécuter la pipeline CI/CD, il faut installer ce qu'on appelle un runner, un processus capable d'exécuter les jobs Gitlab.
Nous allons utiliser, (encore), le gestionnaire de paquets Helm.
Il faut d'ailleurs aussi créer un fichier values.yaml.

Listing VI.2 – values.yaml

```
image:
  registry: registry.gitlab.com
  image: gitlab-org/gitlab-runner
```

```
imagePullPolicy: IfNotPresent
useTini: false
unregisterRunners: true
terminationGracePeriodSeconds: 3600

concurrent: 10

shutdown_timeout: 0

checkInterval: 3

logLevel: debug

gitlabUrl: https://nailloux.gitlab.com  # Use your Proxy/GitLab URL
#secret: gitlab-runner-token             # The secret we created in Step 2

runnerToken: "glrt-_amtT5NiPBnvwO7U6niGBG86MQpwOjEKdDozCnU6MQ8.01.170ht4ipd"
unregisterRunners: false

rbac:
  create: true                      # Allows the runner to create job Pods


certsSecretName: gitlab-runner-crt

## Ensure the Manager has permission to read the mounted secret
securityContext:
  fsGroup: 65533

runners:
  config: |
    [[runners]]
      # This path is where the Helm chart automatically mounts 'certsSecretName'
      tls-ca-file = "/home/gitlab-runner/.gitlab-runner/certs/nailloux.gitlab.com.crt"
      clone_url = "https://nailloux.gitlab.com"

      [runners.kubernetes]
        namespace = "gitlab-runner"
        image = "debian:latest"
        privileged = true

        # Request more memory for the build pod
        memory_limit = "4Gi"
        memory_request = "2Gi"
        # CPU can also affect build speed/stability
        cpu_limit = "2"
        cpu_request = "1"

        # If the error persists, increase the helper memory too
        helper_memory_limit = "1Gi"

        # Injects the CA into the build pod's trust store
        ca_certificates_injection = true
```

```
          # --- MOUNT FOR THE BUILD CONTAINER ---
          # [[runners.kubernetes.volumes.secret]]
          #   name = "gitlab-runner-crt"
          #   mount_path = "/etc/ssl/certs"
          #   read_only = true
          #   # 420 is octal 0644 (readable by all)
          #   [runners.kubernetes.volumes.secret.items]
          #     "nailloux.gitlab.com.crt" = "420"

          #   [[runners.kubernetes.volumes.empty_dir]]
          #     name = "docker-certs"
          #     mount_path = "/certs/client"
          #     medium = "Memory"

        # 3. Force Git to use this specific file
        [[runners.kubernetes.env]]
          name = "GIT_SSL_CAINFO"
          value = "/etc/ssl/certs/nailloux.gitlab.com.crt"
        [[runners.kubernetes.env]]
          name = "DOCKER_HOST"
          value = "tcp://docker:2375"
        [[runners.kubernetes.env]]
          name = "DOCKER_TLS_CERTDIR"
          value = ""
        [[runners.kubernetes.env]]
          name = "DOCKER_TLS_VERIFY"
          value = "0"
        [[runners.kubernetes.env]]
          name = "DOCKER_CERT_PATH"
          value = ""
```

Et on installe le runner avec la commande associée

```
kubectl create namespace gitlab-runner

kubectl create secret generic runner-secret \
  --from-literal=runner-registration-token="glrt-
      _amtT5NiPBnvwO7U6niGBG86MQpwOjEKdDozCnU6MQ8.01.170ht4ipd" \
  -n gitlab-runner

helm repo add gitlab https://charts.gitlab.io
helm repo update
helm install --namespace gitlab-runner gitlab-runner -f values.yaml gitlab/gitlab-runner
```

Pour le désinstaller

```
helm delete --namespace gitlab-runner gitlab-runner
```

VI.D LoadBalancer avec MetalLB

Cette partie est optionnelle car elle sera effectuée seulement si nous disposons du temps nécessaire.
Cela n'a pas été le cas durant la SAE.

```
helm repo add metallb https://metallb.github.io/metallb
helm install metallb metallb/metallb
```

VI.E Docker Registry

Dans le cadre du déploiement de notre application, nous utilisons des Dockerfile pour créer des images personnalisées chargées de copier des fichiers hôtes. Plutôt que d'utiliser DockerHub, nous avons décidé d'utiliser un registre docker personnel.
Le serveur gitlab possède déjà une option pour disposer d'un serveur Docker Registry mais on ne va pas utiliser cette méthode ici.

VI.E.1 Phase 1 : Creation des PVC

Afin de stocker les images docker (volumineuses), nous allons créer des volumes persistants (PVC).
Il faut aller dans Storage > PersistentVolumeClaims.
On créé un volume registry-data avec les paramètres suivants :

- Size : 20GiB
- Classe : local-path (créée précédemment)

VI.E.2 Phase 2 : Deploiement des pods

Dans la section Workload > Deployments puis Create.

1. Nom : docker-registry.
2. Image : registry :3
3. Stockage : Monter le PVC registry-data dans /var/lib/registry.

Listing VI.3 – Variables d'environnements :

```
REGISTRY_HTTP_ADDR: :0.0.0.0:5000 (The internal port).

REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /var/lib/registry.

REGISTRY_HTTP_HEADERS_Access-Control-Allow-Origin : '[''*'']'
REGISTRY_HTTP_HEADERS_Access-Control-Allow-Methods : '[''HEAD'', ''GET'', ''OPTIONS'', ''
    DELETE'']'
REGISTRY_HTTP_HEADERS_Access-Control-Allow-Headers : '[''Authorization'', ''Accept'', ''
    Cache-Control'']'
```

Sera ensuite exposé le port 5000 en tcp, avec un service ClusterIP, nommé https.

VI.E.3 Phase 3 : UI

Afin d'interfacer proprement notre serveur Docker-Registry, nous allons faire appel à une autre image docker **joxit/docker-registry-ui** qui implique un nouveau déploiement chargé d'afficher une UI.
Il faudra définir les paramètres suivants :

Port	Protocole	Type	Nom
80	TCP	ClusterIP	http
443	TCP	ClusterIP	https

Il faut par la suite définir des variables d'environnements pour que le serveur UI puisse communiquer avec le serveur registre.

```
NGINX_PROXY_PASS_URL: http://docker-registry:5000
DELETE_IMAGES: true
SINGLE_REGISTRY: true
```

VI.E.4 Phase 4 : Création d'un Ingress avec HTTPS

Docker, par défaut, refuse de communiquer avec une API/Serveur n'utilisant pas HTTPS ou n'ayant pas un certificat SSL valide. Il est plutôt aisé de mettre en place un service HTTPS avec Rancher/Kubernetes au moyen des **Ingress**. Mais il sera de notre ressort de faire avec l'erreur de certificat auto-signé.

```
export KEY_FILE=key.pem
export CERT_FILE=cert.pem
export HOST=nailloux.registry.com

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout $KEY_FILE -out $CERT_FILE -
    subj "/CN=$HOST/O=$HOST" -addext "subjectAltName = DNS:$HOST"
```

Une fois le certificat généré, il suffit d'aller dans la section • Secret → Create → TLS Certificate et de renseigner la clé privée et le certificat.
Maintenant, il faut aller dans • **Service Discovery** → Ingresses → Create.

Request Host	Service	Port	Label et Annotations	Certificats :Secret-Name	Certificats : Host
nailloux.registry.com	docker-registry-ui	80	”nginx.ingress.kubernetes.io/proxy-body-size” = 0	registry.tls	nailloux.registry.com

VI.E.5 Phase 5 : Docker push

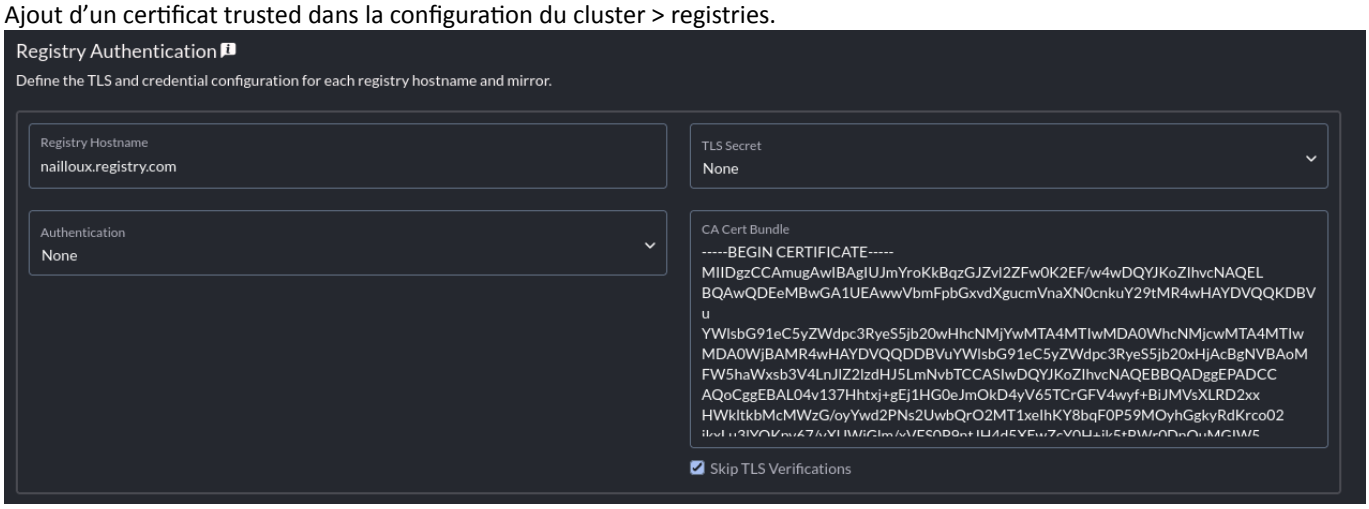
Maintenant il est possible de publier des images dans le registre en taggant les images de la manière suivante :

```
docker tag nginx:latest nailloux.registry.com/nginx:latest
docker push nailloux.registry.com/nginx:latest
```

VII Troubleshooting

VII.A Problème de certificats SSL

Ajout d'un certificat trusted dans la configuration du cluster > registries.



VII.B Cannot allocate new block due to per host block limit

Lorsque cette erreur survient c'est que calico ne dispose plus d'IPs restantes, ce qui est largement du à des IPs unitilisées n'étant pas recyclées. On peut forcer ce recyclage avec les commandes suivantes.

```
calicoctl datastore migrate lock
calicoctl ipam check -o report.json
calicoctl ipam release --from-report report.json
calicoctl datastore migrate unlock
```

VII.C DNS

Lorsqu'on utilise systemd-resolved, le fichier /etc/resolv.conf utilise une adresse locale inutilisable par le pod **coreDNS**. Il faut donc spécifier le fichier où systemd-resolved garde ses serveurs DNS.

```
echo 'services:
  kubelet:
    extra_args:
      resolv-conf: "/run/systemd/resolve/resolv.conf"' > /etc/rancher/rke2/config.yaml
```

On peut aussi faire la méthode suivante, moins flexible, en ajoutant en dur un hôte comme on le ferait dans le fichier **/etc/hosts**. Solution à ne pas faire car on ne résoud pas réellement le souci DNS.

```
  kubectl -n cattle-system logs -l app=cattle-cluster-agent

  export KUBE_EDITOR="nano"
  kubectl edit configmap rke2-coredns-rke2-coredns -n kube-system

Corefile: |-
  .:53 {
      errors
      health {
          lameduck 10s
      }
      ready
      kubernetes  cluster.local  cluster.local in-addr.arpa ip6.arpa {
          pods insecure
          fallthrough in-addr.arpa ip6.arpa
          ttl 30
      }
      hosts {
          10.0.1.3 rancher.rancher
          fallthrough
      }
      prometheus  0.0.0.0:9153
      forward  . /etc/resolv.conf
      cache  30
      loop
      reload
      loadbalance
  }


  kubectl rollout restart deployment rke2-coredns-rke2-coredns -n kube-system
  kubectl scale deployments/cattle-cluster-agent -n cattle-system --replicas=0
  kubectl scale deployments/cattle-cluster-agent -n cattle-system --replicas=1
```