

1. Introduction.....	4
1.1. Présentation du projet.....	4
1.2. Contexte et problématique.....	5
1.3. Objectifs du projet.....	5
1.4. Aproche méthodologique.....	6
1.5. Cadre méthodologique Agile.....	7
1.5.1. Principes de l'agilité appliqués au projet.....	7
1.5.2. Organisation en sprints et gestion du backlog.....	7
1.5.3. Rôles et responsabilités.....	7
1.5.4. Utilisation de Trello pour le suivi du projet.....	7
1.5.5. Rétrospectives et adaptation continue.....	8
2. Contexte et architecture existante.....	9
2.1. Présentation de l'entreprise et des besoins.....	9
2.2. Architecture actuelle de l'application.....	9
2.3. Enjeux liés à la scalabilité et à la haute disponibilité.....	9
2.4. Contraintes et défis techniques.....	10
3. Conception de la nouvelle infrastructure.....	11
3.1. Analyse des limites de l'existant.....	11
3.2. Définition des objectifs pour la nouvelle infrastructure.....	11
3.3. Choix des technologies et des outils.....	12
3.4. Architecture cible.....	13
3.4.1. Dépôt du code source sur GitHub.....	13
3.4.2. Provisionnement de l'infrastructure avec Terraform.....	13
3.4.3. Automatisation de la configuration avec Ansible.....	13
3.4.4. CI/CD avec Jenkins et Docker Hub.....	14
3.4.5. Déploiement dans Kubernetes.....	14
3.4.6. Cluster Galera MariaDB dans K3s via Helm.....	15
3.4.7. Schéma de l'architecture cible.....	16
4. Provisionnement de l'infrastructure avec Terraform.....	17
4.1. Introduction à Terraform.....	17
4.2. Définition des ressources (VMs, réseaux, sécurité).....	17
4.3. Déploiement de l'infrastructure cloud.....	21
4.4. Gestion de l'état et versionnement du code Terraform.....	21
5. Mise en place de l'infrastructure avec Ansible.....	22
5.1. Rôle et présentation d'Ansible.....	22
5.2. Installation et configuration d'Ansible.....	22
5.2.1 Automatisation de l'installation des services.....	25
5.2.2. Installation de Docker	26

5.2.3. Installation de K3s sur les nœuds.....	27
5.2.4. Installation de Jenkins.....	29
5.2.5. Installation de SonarQube.....	33
5.2.6. Installation de SonarScanner.....	35
5.3. Déploiement des configurations et gestion des mises à jour.....	36
5.4. Versionnement et suivi des playbooks sur GitHub.....	37
6. Déploiement des services applicatifs avec Helm.....	35
6.1. Introduction à Helm et aux charts.....	35
6.2. Déploiement de MetallLB pour la gestion des LoadBalancers.....	38
6.3. Déploiement de Galera MariaDB en haute disponibilité.....	40
6.4. Déploiement de phpMyAdmin pour l'administration de la base.....	43
6.5. Déploiement de Prometheus et Grafana dans le cluster K3s.....	46
7. Conteneurisation et packaging de l'application.....	50
7.1. Présentation de Docker et bonnes pratiques.....	50
7.2. Conteneurisation du service web Taskshare.....	51
7.3. Ajout de tests, fichiers Jenkinsfile et sonar.properties.....	53
7.4. Création d'un Helm chart personnalisé pour l'application.....	56
8. Automatisation CI/CD avec Jenkins et Docker.....	58
8.1. Présentation de l'intégration et du déploiement continu.....	58
8.2. Configuration de Jenkins et Webhooks GitHub.....	58
8.3. Création du pipeline CI/CD.....	69
8.3.1. Build de l'image Docker de l'application.....	69
8.3.2. Analyse statique avec SonarScanner.....	69
8.3.3. Push de l'image sur Docker Hub.....	70
8.3.4. Déploiement automatique vers l'environnement de test (K3s).....	70
8.3.5. Promotion vers l'environnement de production.....	70
8.4. Notifications, logs et suivi des pipelines.....	71
9. Supervision et monitoring des services.....	72
9.1. Introduction à Prometheus et Grafana.....	72
9.2. Surveillance du cluster K3s.....	72
9.3. Monitoring de l'application TaskShare.....	73
9.4. Monitoring de Galera MariaDB.....	74
9.5. Alerting et gestion proactive des incidents.....	74
9.6. Ajout d'un dashboard personnalisé dans Grafana.....	75
10. Sécurisation de l'infrastructure.....	77
10.1. Gestion des accès et permissions SSH.....	77
10.2. Mise en place de certificats TLS (Traefik, application).....	77
10.3. Audit de sécurité et centralisation des logs.....	77
10.4. Sécurisation des bases de données et du pipeline CI/CD.....	78
11. Tests, validation et mise en production.....	78
11.1. Tests fonctionnels et de performance.....	78

11.2. Validation complète du pipeline CI/CD.....	79
11.3. Optimisations finales et corrections.....	79
11.4. Mise en production et stratégie de rollback.....	79
11.5. Captures d'écran et démonstration du bon fonctionnement.....	80
12. Bilan et perspectives.....	83
12.1. Résultats obtenus et bénéfices pour l'entreprise.....	83
12.2. Défis rencontrés et solutions apportées.....	83
12.3. Perspectives d'évolution et axes d'amélioration.....	83
13. Annexes et références.....	84
13.1. Scripts, fichiers Terraform et playbooks Ansible.....	84
13.2. Helm charts et fichiers de configuration.....	84
13.3. Bonnes pratiques.....	85
13.4. Bibliographie et ressources utilisées.....	85
Conclusion.....	86
Remerciements.....	87

1. Introduction

Dans un contexte marqué par la transformation numérique, les entreprises cherchent à optimiser leurs processus de développement et de déploiement en adoptant des approches modernes telles que l'**agilité** et le **DevOps**. Ces méthodologies permettent non seulement d'accélérer les cycles de livraison, mais aussi de renforcer la stabilité, la résilience et la maintenabilité des systèmes d'information.

Le présent rapport s'inscrit dans le cadre d'un stage de fin d'études axé sur les pratiques DevOps, et a pour objectif principal la mise en place d'une infrastructure automatisée destinée à l'application **TaskShare**. Ce projet met en œuvre des technologies **d'Infrastructure as Code** (Terraform, Ansible), de **conteneurisation** (Docker), **d'orchestration** (Kubernetes), ainsi qu'un pipeline complet d'**intégration et de déploiement continu** (CI/CD) basé sur Jenkins. La supervision de l'environnement est assurée via des outils de monitoring tels que Prometheus et Grafana.

Réalisé au sein d'un environnement virtualisé (Proxmox), le projet illustre concrètement l'ensemble des étapes de déploiement d'une application cloud-native, depuis la définition de l'infrastructure jusqu'à son exploitation en production, dans un cadre aligné sur les standards actuels de l'ingénierie système et logicielle.

1.1. Présentation du projet

Le projet porte sur le déploiement et l'automatisation de l'infrastructure de **TaskShare**, une application web collaborative de gestion de tâches.

TaskShare permet à des équipes de créer des projets, d'ajouter des tâches, d'assigner des membres, de suivre l'avancement en temps réel et de recevoir des notifications automatiques. L'objectif de l'application est d'améliorer la productivité et la communication dans des environnements de travail dynamiques en centralisant la gestion de projet dans une interface intuitive.

L'application repose sur un backend développé en **Node.js**, une base de données **MySQL**, et une interface web responsive.

Ce projet s'inscrit dans le cadre d'un stage DevOps. Il vise à concevoir, automatiser et déployer l'infrastructure nécessaire à TaskShare, en s'appuyant sur des outils modernes (Terraform, Ansible, Docker, Kubernetes, Jenkins...) pour garantir la scalabilité, la haute disponibilité, et la résilience du système.

1.2. Contexte et problématique

Dans le cadre de ce stage, l'entreprise m'a confié la mission de mettre en œuvre un environnement DevOps complet pour héberger TaskShare.
L'entreprise fournit un serveur **Proxmox**, sur lequel l'ensemble de l'infrastructure devra être provisionnée et configurée.
L'objectif est de :

- mettre en place une infrastructure **automatisée** de bout en bout (infrastructure as code),
- permettre le **déploiement continu** de l'application,
- garantir la **haute disponibilité** des services,
- faciliter la **scalabilité horizontale** de l'application en cas d'augmentation du trafic.

1.3. Objectifs du projet

Les objectifs principaux de ce projet sont les suivants :

- Déployer l'infrastructure de la plateforme **TaskShare** sur un environnement virtualisé fourni via **Proxmox**.
- Moderniser le processus de mise en production grâce à des outils et pratiques DevOps.
- Automatiser le provisionnement de l'infrastructure avec Terraform, pour créer dynamiquement les machines virtuelles.
- Automatiser la configuration et l'installation des services nécessaires (Kubernetes, Docker, Jenkins, etc.) à l'aide d'**Ansible**.
- Conteneuriser l'application TaskShare avec **Docker**.
- Orchestrer les conteneurs avec **Kubernetes** pour assurer la scalabilité et la tolérance aux pannes.
- Implémenter un pipeline **CI/CD** avec **Jenkins**, pour automatiser les étapes de test, de build et de déploiement.

- Mettre en place un **cluster Galera MariaDB** pour assurer la redondance et la haute disponibilité des données.
- Assurer la **surveillance, la gestion des incidents et la sécurité** de l'infrastructure à l'aide de **Prometheus, Grafana** .

1.4. Approche méthodologique

L'approche méthodologique repose sur une démarche DevOps et l'automatisation des processus, structurée autour des étapes suivantes :

- **Analyse de l'existant et définition de l'architecture cible** : étude des besoins applicatifs de TaskShare, choix des technologies adaptées, et modélisation de l'architecture cible.
- **Provisionnement de l'infrastructure avec Terraform** : création automatique des machines virtuelles sur Proxmox à l'aide de fichiers de configuration déclaratifs, assurant cohérence, réplicabilité et traçabilité.
- **Automatisation avec Ansible** : installation, configuration et gestion des services (Kubernetes, Docker, Jenkins, MariaDB, etc.) de manière idempotente et automatisée.
- **Conteneurisation et orchestration** : encapsulation des composants de TaskShare dans des conteneurs Docker, puis orchestration à l'aide de Kubernetes pour garantir la scalabilité, la résilience et la facilité de gestion.
- **Implémentation du CI/CD** : automatisation du cycle de vie de l'application avec Jenkins, incluant l'intégration continue, les tests, le build, et le déploiement sur le cluster.
- **Tests et validation** : réalisation de tests de charge, de performance et de résilience pour valider la fiabilité de l'infrastructure.
- **Monitoring et alerting** : mise en place d'une stack de supervision avec Prometheus, Grafana et Nagios pour assurer un suivi continu de la plateforme et réagir rapidement aux incidents.

1.5. Cadre méthodologique Agile

Dans le cadre de ce projet, une démarche Agile a été adoptée afin de favoriser l'adaptation continue, l'amélioration incrémentale et l'implication active dans la réalisation des différentes étapes. Cette approche permet de structurer le travail en cycles courts, de s'adapter rapidement aux contraintes techniques ou changements de priorités, et de garantir une meilleure visibilité sur l'avancement du projet.

1.5.1. Principes de l'agilité appliqués au projet

L'approche Agile appliquée repose sur les valeurs fondamentales du Manifeste Agile : collaboration étroite avec les parties prenantes, livraisons incrémentales, réactivité au changement, et priorisation de la qualité logicielle. Ces principes ont été adaptés au contexte spécifique du stage DevOps, avec un focus particulier sur l'automatisation, l'intégration continue, et la livraison fréquente de versions fonctionnelles de l'infrastructure.

1.5.2. Organisation en sprints et gestion du backlog

Le projet a été découpé en **sprints hebdomadaires**, chacun ayant des objectifs précis liés aux étapes de provisionnement, d'automatisation ou de déploiement. Un **backlog** a été initialement défini, listant l'ensemble des tâches à réaliser (infrastructure, CI/CD, supervision, sécurité...). Ce backlog a été mis à jour régulièrement, en fonction de l'avancement réel, des blocages rencontrés et des priorités revues lors des rétrospectives.

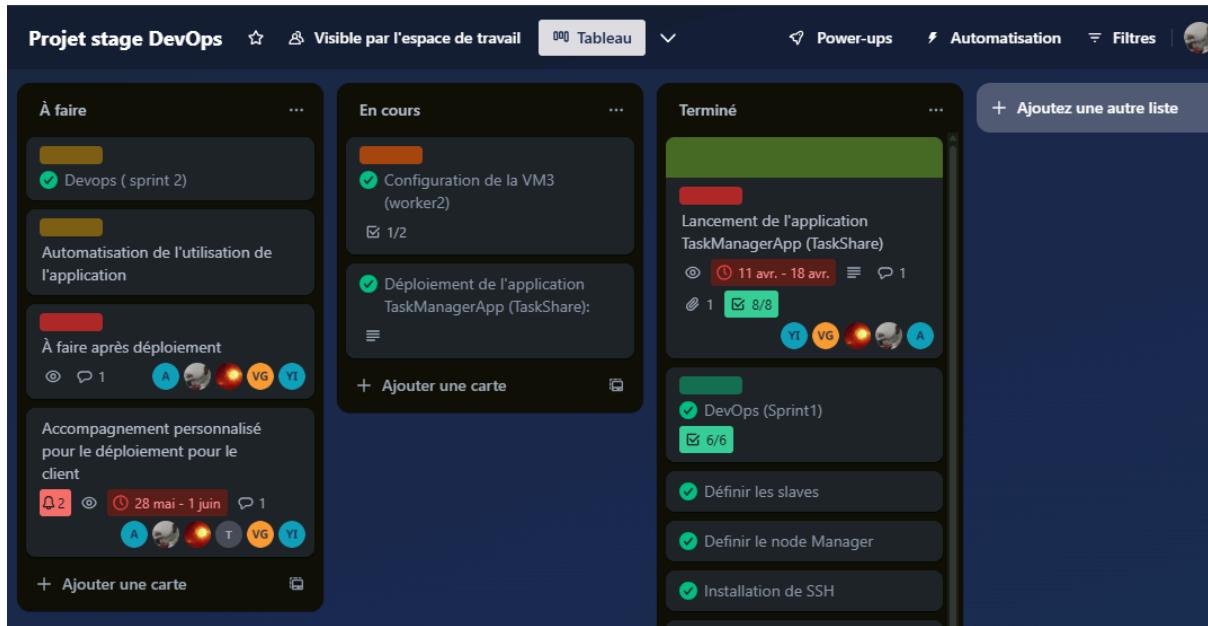
1.5.3. Rôles et responsabilités

Les rôles ont été adaptés au contexte du stage :

- **Product Owner (PO)** : tuteur technique de l'entreprise, garant des besoins exprimés et des priorités métier.
- **Scrum Master** : rôle assuré de manière informelle par le tuteur, facilitant la gestion du temps et des obstacles techniques.
- **DevOps Engineer** : le stagiaire, chargé de l'ensemble des livrables techniques, du développement à la mise en production de l'infrastructure.

1.5.4. Utilisation de Trello pour le suivi du projet

L'outil **Trello** a été utilisé comme tableau de bord Agile, structuré selon les colonnes : *Backlog, À faire, En cours, Revue, Fait*. Chaque carte représente une tâche précise, avec des checklists, des dates butoirs, et parfois des liens vers du code ou des ressources associées. Trello a permis un suivi clair et visuel de la progression du projet, tout en facilitant la communication avec le tuteur.



1.5.5. Rétrospectives et adaptation continue

À la fin de chaque sprint, une **rétrospective informelle** était menée pour évaluer les points forts, les difficultés rencontrées, et les pistes d'amélioration. Cette boucle de rétroaction a permis d'ajuster les priorités, d'optimiser l'organisation, et de garantir une meilleure efficacité dans les phases suivantes du projet. Cette pratique a renforcé l'autonomie du stagiaire et la qualité des livrables techniques.

2. Contexte et architecture existante

2.1. Présentation de l'entreprise et des besoins

L'entreprise dans laquelle se déroule ce stage est une structure spécialisée dans le développement de solutions web. Elle accorde une importance particulière à la modernisation de son infrastructure et à l'adoption des pratiques DevOps afin d'accroître l'efficacité de ses processus de développement, de déploiement et de maintenance.

Dans cette optique, le besoin principal identifié est de mettre en place une infrastructure automatisée et résiliente pour déployer **TaskShare**, une nouvelle application collaborative de gestion de tâches. Cette infrastructure devra garantir la haute disponibilité des services, être facilement scalable, et permettre un déploiement continu grâce à l'automatisation.

2.2. Architecture actuelle de l'application

Étant une nouvelle application en cours de développement, **TaskShare** ne dispose pas encore d'une infrastructure déployée en production. Actuellement, les différents composants sont testés localement ou dans des environnements de développement manuels, sans mécanisme d'orchestration ou d'automatisation.

L'architecture applicative de **TaskShare** se compose des éléments suivants :

- **Frontend** : Une interface web responsive, développée en React Native.
- **Backend** : Une API REST développée en Node.js, qui gère la logique métier et les interactions avec la base de données.
- **Base de données** : Un serveur MySQL assurant la persistance des données liées aux projets, tâches et utilisateurs.

Cette architecture repose sur une séparation claire des couches, mais elle n'est pour l'instant ni conteneurisée ni déployée dans un environnement de production stable.

2.3. Enjeux liés à la scalabilité et à la haute disponibilité

L'objectif de l'entreprise est de rendre TaskShare accessible à un large public, incluant des équipes de tailles variables dans différents fuseaux horaires. Cela implique les enjeux suivants :

- **Scalabilité** : L'infrastructure doit pouvoir évoluer horizontalement pour supporter un pic de trafic ou une croissance du nombre d'utilisateurs, sans impact sur les

performances.

- **Haute disponibilité** : Les services critiques (API, base de données) doivent rester accessibles en permanence. L'objectif est de réduire au maximum les temps d'indisponibilité, même en cas de défaillance d'un composant.
- **Équilibrage de charge** : Une distribution intelligente des requêtes est nécessaire pour éviter la saturation des ressources.
- **Résilience** : L'infrastructure doit pouvoir se rétablir automatiquement après un incident, grâce à la redondance et au monitoring.

2.4. Contraintes et défis techniques

Plusieurs contraintes techniques ont été identifiées dans le cadre de ce projet :

- **Ressources matérielles limitées** : Le projet repose sur un serveur Proxmox unique mis à disposition par l'entreprise, ce qui nécessite une gestion rigoureuse des ressources (CPU, RAM, stockage).
- **Automatisation complète exigée** : Le déploiement de l'infrastructure doit être entièrement automatisé via des outils comme Terraform et Ansible pour garantir la reproductibilité et éviter les erreurs humaines.
- **Intégration de multiples outils** : La mise en place du pipeline CI/CD nécessite l'interopérabilité entre Jenkins, Docker, Kubernetes, GitHub, etc.
- **Maîtrise des outils DevOps** : L'ensemble du projet exige une bonne connaissance des outils DevOps ainsi que de leur configuration dans un environnement virtualisé.

3. Conception de la nouvelle infrastructure

3.1. Analyse des limites de l'existant

L'environnement actuel repose sur des déploiements manuels, effectués localement ou sur des machines virtuelles non orchestrées. Cette approche présente plusieurs limitations :

- **Absence d'automatisation** : Le déploiement nécessite des interventions humaines, ce qui augmente le risque d'erreur et rend difficile la reproductibilité.
- **Manque d'orchestration** : Aucun orchestrateur de conteneurs n'est en place, ce qui limite la capacité à assurer la haute disponibilité ou la scalabilité.
- **Pas de CI/CD** : Aucun pipeline d'intégration ou de déploiement continu n'est configuré, rendant les mises à jour lentes et sujettes à erreur.
- **Surveillance et résilience limitées** : Le monitoring est rudimentaire voire inexistant, et la capacité de récupération en cas de panne est faible.

3.2. Définition des objectifs pour la nouvelle infrastructure

Pour pallier les limites de l'existant, la nouvelle infrastructure devra répondre aux objectifs suivants :

- **Automatisation complète** du provisionnement et de la configuration grâce à Terraform et Ansible.
- **Déploiement conteneurisé** de tous les composants de l'application.
- **Orchestration avec Kubernetes (K3s)** pour assurer la scalabilité et la résilience.
- **Mise en place d'un pipeline CI/CD** avec Jenkins pour assurer l'intégration et le déploiement continus.
- **Haute disponibilité de la base de données** grâce à Galera MariaDB déployé via Helm.
- **Exposition sécurisée des services** via Traefik (Ingress Controller).
- **Surveillance et observabilité** à l'aide de Prometheus et Grafana.

3.3. Choix des technologies et des outils

Domaine	Outil choisi	Rôle
Infrastructure	Proxmox	Virtualisation des machines
Provisionnement	Terraform	Création automatisée des VMs
Configuration	Ansible	Configuration des machines (OS, packages, accès SSH...)
Conteneurisation	Docker	Création des images applicatives
Orchestration	K3s	Déploiement des conteneurs avec Kubernetes allégé
Base de données	MariaDB Galera	RéPLICATION multi-nœud pour la haute disponibilité
CI/CD	Jenkins + Docker Hub	Intégration et déploiement automatisés
Ingress	Traefik	Routage HTTP/HTTPS, TLS
Monitoring	Prometheus + Grafana	Collecte de métriques et visualisation
Registre de code	GitHub	Hébergement du code et des workflows

3.4. Architecture cible

L'architecture cible repose sur une infrastructure entièrement automatisée et modulaire, déployée sur un cluster K3s orchestré dans des machines virtuelles Proxmox. Les composants clés sont :

3.4.1. Dépôt du code source sur GitHub

L'ensemble du code source de l'application (frontend, backend) ainsi que les fichiers de configuration et d'automatisation (Terraform, Ansible, Helm charts, etc.) sont versionnés sur GitHub. Cela permet de :

- Centraliser le code dans un espace collaboratif.
- Déclencher automatiquement les pipelines Jenkins.
- Faciliter la traçabilité des modifications.

3.4.2. Provisionnement de l'infrastructure avec Terraform

Terraform permet de décrire l'infrastructure sous forme de code (Infrastructure as Code) pour :

- Créer automatiquement les machines virtuelles sur Proxmox.
- Définir les réseaux, volumes et interfaces.
- Réduire les erreurs manuelles et améliorer la reproductibilité.

3.4.3. Automatisation de la configuration avec Ansible

Une fois les machines provisionnées, Ansible est utilisé pour :

- Configurer les systèmes (utilisateurs, SSH, packages...).
- Installer les dépendances nécessaires à K3s.
- Déployer le cluster K3s sur plusieurs nœuds (architecture HA).

3.4.4. CI/CD avec Jenkins et Docker Hub

Jenkins est configuré pour déclencher automatiquement les pipelines de build et de déploiement :

- **Build** des images Docker à partir du code source.
- **Push** vers Docker Hub.
- **Déploiement** automatisé dans le cluster K3s à l'aide de Helm Charts.
- **Tests** unitaires et de validation sont également intégrés.

3.4.5. Déploiement dans Kubernetes

L'ensemble des composants applicatifs sont déployés dans des pods Kubernetes, avec les avantages suivants :

- Facilité de scaling horizontal.
- Redondance des services critiques.
- Mise à jour continue grâce aux pipelines.
- Séparation logique via des namespaces et services.

L'exposition des services se fait via un **Ingress Controller Traefik**, configuré pour gérer les certificats TLS et le routage HTTP/HTTPS.

3.4.6. Cluster Galera MariaDB dans K3s via Helm

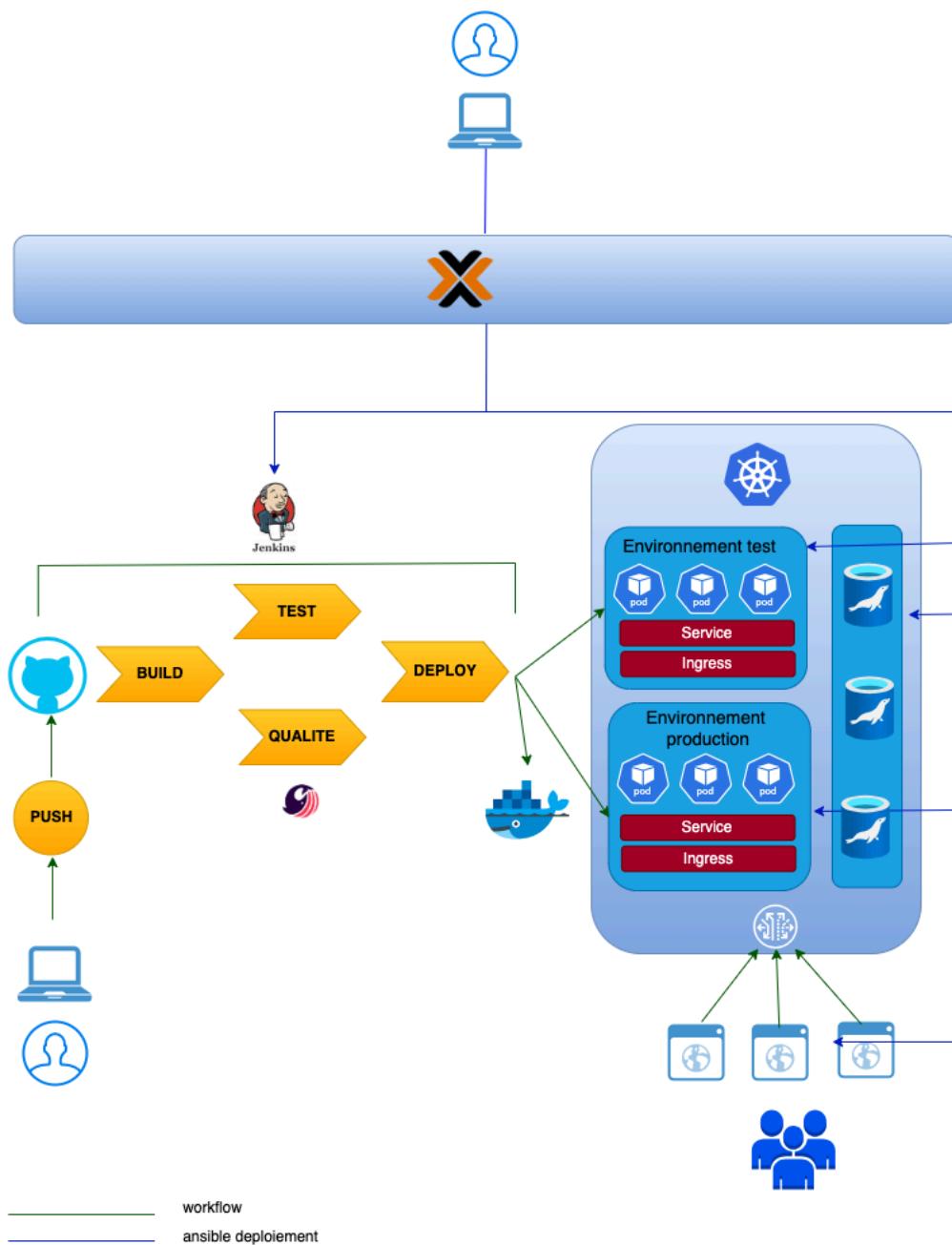
La base de données MariaDB est déployée en cluster Galera à l'aide du chart Helm `bitnami/mariadb-galera`. Ce choix permet d'assurer :

- **Haute disponibilité** : trois pods MariaDB répliqués assurent la continuité de service.
- **Résilience** : en cas de panne d'un pod, les autres prennent le relais automatiquement.
- **Intégration native** : les services sont accessibles via des DNS internes Kubernetes.
- **Stockage persistant** : les volumes sont gérés par le provisioner de K3s.

Contrairement à une architecture classique, aucun HAProxy externe n'est utilisé. La répartition des requêtes est gérée via les services Kubernetes.

3.4.7. Schéma de l'architecture cible

Voici le schéma illustrant l'ensemble de la nouvelle infrastructure déployée :



4. Provisionnement de l'infrastructure avec Terraform

4.1. Introduction à Terraform

Terraform est un outil d'Infrastructure as Code (IaC) développé par HashiCorp. Il permet de décrire et provisionner des infrastructures cloud, on-premise ou hybrides à l'aide de fichiers de configuration déclaratifs. Dans le cadre de ce projet, Terraform est utilisé pour automatiser la création de machines virtuelles sur un serveur Proxmox.

4.2. Définition des ressources (VMs, réseaux, sécurité)

a) Mise en place de l'environnement de travail

Pour centraliser le code Terraform, un dépôt GitHub dédié a été créé :

Dépôt GitHub : [terraform-taskshare](https://github.com/ibrahimbakayoko/terraform-taskshare.git)

Le dépôt est ensuite cloné localement :

>> git clone

<https://github.com/ibrahimbakayoko/terraform-taskshare.git>

>> cd [terraform-taskshare](#)

b) Création d'un template Cloud-Init

Pour automatiser le déploiement des machines sur Proxmox, un **template Debian 12**

Cloud-Init est généré à l'aide d'un script shell :

>> nano create-debian12-cloudinit-template.sh

```
● root@pve:~# cat create-debian12-template.sh
#!/bin/bash

# -----
# Paramètres à modifier
# -----
VMID=9000
VMNAME="debian12-cloudinit"
STORAGE="local-lvm"           # Ou autre stockage (ex: "local", "ssd", etc.)
BRIDGE="vmbr0"
ISO_URL="https://cloud.debian.org/images/cloud/bookworm/latest/debian-12-genericcloud-amd64.qcow2"
DISK_IMAGE="debian-12-genericcloud-amd64.qcow2"
SSH_KEY_PATH="$HOME/.ssh/id_rsa.pub" # Facultatif - injecte la clé dans le template

# -----
# Dépendance requise
# -----
echo "● Vérification des dépendances..."
if ! command -v virt-customize >/dev/null 2>&1; then
    echo "✗ 'virt-customize' manquant. Installer avec : sudo apt install libguestfs-tools"
    exit 1
fi
```

>> chmod +x create-debian12-cloudinit-template.sh

>> ./create-debian12-cloudinit-template.sh

```
unused0: successfully imported disk 'local-lvm:vm-9001-disk-0'
update VM 9001: -agent enabled=1 -boot c -bootdisk scsi0 -ide2 local-lvm:cloudinit -scsi0 local-lvm:vm-9001-disk-0 -scsihw virtio-scsi-pci -serial0 socket -vga serial0
WARNING: You have not turned on protection against thin pools running out of space.
WARNING: Set activation/thin_pool_autoextend_threshold below 100 to trigger automatic extension of thin pools before they get full.
Logical volume "vm-9001-cloudinit" created.
WARNING: Sum of all thin volume sizes (29.01 GiB) exceeds the size of thin pool pve/data and the size of whole volume group (<19.50 GiB).
ide2: successfully created disk 'local-lvm:vm-9001-cloudinit,media=cdrrom'
generating cloud-init ISO
Conversion en template...
Renamed "vm-9001-disk-0" to "base-9001-disk-0" in volume group "pve"
Logical volume pve/base-9001-disk-0 changed.
WARNING: Combining activation change with other commands is not advised.
✓ Template debian12-cloudinit (9001) prêt à être utilisé avec Proxmox ou Terraform ✨
root@pve:~/bin# [ ]
```

Résumé	Supprimer	Éditer	Régénérer l'image
Matériel			Par défaut
Cloud-Init			
Options			
Historique des tâches			
Sauvegarde			
Réplication			
Pare-feu			
Permissions			

Ce template permet à chaque VM déployée de disposer d'une configuration initiale automatisée (hostname, SSH, utilisateur, etc.).

c) Installation de Terraform

L'installation de Terraform s'effectue sur la machine locale avec les commandes

suivantes :

- Installer les dépendances nécessaire :

>> sudo apt update && sudo apt install -y curl gnupg lsb-release

- Ajouter la clé GPG de HashiCorp :

**>> curl -fsSL https://apt.releases.hashicorp.com/gpg | |
sudo gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg**

- Ajouter le dépôt officiel HashiCorp à APT :

**>> echo "deb
[signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \\\nhttps://apt.releases.hashicorp.com \$(lsb_release -cs) main" | |
sudo tee /etc/apt/sources.list.d/hashicorp.list**

- Installer Terraform :

>> sudo apt update && sudo apt install -y terraform

- Vérifier l'installation et la version terraform :

>> terraform -v

```
Last login: Sat May 17 14:48:19 on ttys001
bakayokobrahima@MacBook-Pro-de-BAKAYOKO ~ % terraform -version
Terraform v1.11.2
on darwin_amd64
```

d) Fichiers de configuration

Deux fichiers principaux composent la configuration Terraform :

- **main.tf** : définit les ressources à provisionner (VMs, réseaux, stockage...)
- **variables.tf** : centralise les variables (IP, noms, tailles, etc.)

Extrait de **main.tf** :

```
You, 1 second ago | 2 authors (BAKAYOKO BRAHIMA and one other)
1  terraform {
2    required_providers {
3      proxmox = {
4        source  = "bpg/proxmox"
5        version = ">= 0.39.0"
6      }
7    }
8    required_version = ">= 1.4.0"
9  }
10
11 provider "proxmox" {
12   endpoint  = var.pm_api_url
13   username  = var.pm_user
14   password  = var.pm_password
15   insecure  = true
16 }
17
18 # Liste des noms de VM à créer
19 locals {
20   vm_names = ["ansiblemaster", "cicdorchestrator", "monitoringhub"]
21   ip_addresses = [
22     "ansiblemaster" = "192.168.27.20/24"
23     "cicdorchestrator" = "192.168.21.66/24"
24     "monitoringhub" = "192.168.22.69/24"          You, 1 second ago • Uncommitted changes
25   ]
26 }
```

Extrait de **variables.tf** :

```
variables.tf
You, 20 minutes ago | 2 authors (BAKAYOKO BRAHIMA and one other)
1  variable "pm_api_url" {
2    type    = string
3    default = "https://10.8.200.100:8006/api2/json"
4  }
5
6  variable "pm_user" {
7    type    = string
```

4.3. Déploiement de l'infrastructure cloud

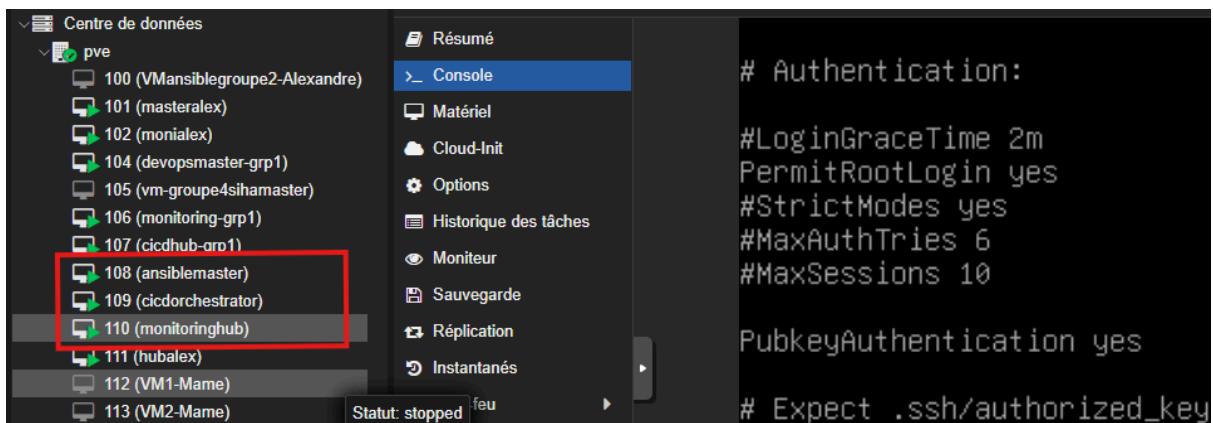
Une fois la configuration définie, le déploiement s'effectue en trois étapes :

>> **terraform init** (Initialise les plugins nécessaires)

>> **terraform plan** (Affiche les changements qui seront effectués)

>> **terraform apply** (Applique la configuration et crée les VMs)

```
Apply complete! Resources: 5 added, 1 changed, 2 destroyed.  
PS C:\Users\stagiaire\Desktop\projet-stage\infra\terrafom-taskshare>
```



4.4. Gestion de l'état et versionnement du code Terraform

L'état de l'infrastructure est suivi via le fichier **terraform.tfstate**, qui contient la représentation exacte des ressources déployées. Ce fichier permet à Terraform de savoir ce qui a été créé et de gérer les modifications futures.

5. Mise en place de l'infrastructure avec Ansible

5.1. Rôle et présentation d'Ansible

Ansible est un outil d'automatisation IT open-source utilisé pour le déploiement, la configuration et la gestion des systèmes. Il repose sur une architecture **agentless** : aucun agent n'est nécessaire sur les machines distantes, ce qui simplifie considérablement la maintenance. Son fonctionnement s'appuie sur SSH pour exécuter des tâches, et sur des fichiers YAML appelés *playbooks* pour décrire les opérations à effectuer.

Dans le cadre de ce projet, Ansible joue un rôle central dans la configuration automatisée des machines virtuelles créées par Terraform. Il permet d'installer et de configurer automatiquement les services suivants :

- Docker
- K3s (distribution légère de Kubernetes)
- Jenkins
- SonarQube
- prometheus
- SonarQube Scanner

5.2. Installation et configuration d'Ansible

Avant d'utiliser Ansible pour automatiser l'infrastructure, il a été nécessaire de préparer l'environnement d'exécution.

❖ Création d'un utilisateur dédié

Sur chaque machine, un utilisateur avec droits sudo a été créé :

```
>> sudo adduser ansible
>> sudo usermod -aG sudo ansible
>> su - ansible
>> sudo whoami
```

```
○ root@ansiblēmaster:~# su - ansible
ansible@ansiblēmaster:~$ su - ansible
```

❖ Environnement des machines

- **ansiblēmaster** : Ce serveur principal est utilisé pour piloter l'automatisation. Il contient Python 3, SSH et Ansible.

- **Node2, Node3** : Ces nœuds sont configurés manuellement avec Python 3 et SSH. Ansible n'y est pas installé.

❖ **Configuration de la connexion SSH entre entre noeud**

Installation de SSH : >> **apt install ssh**

Configuration SSH (/etc/ssh/sshd_config) pour autoriser la connexion.

❖ **Génération et déploiement des clés SSH**

Depuis **devopsmaster**, une clé SSH est générée :

>> **ssh-keygen -t rsa**

```
ansible@ansiblmaster:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ansible/.ssh/id_rsa):
Created directory '/home/ansible/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ansible/.ssh/id_rsa
Your public key has been saved in /home/ansible/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:PbzVNBfLGhsPpAFXj7eYQ0qoopWY5sEusJwz/ML2hcY ansible@ansiblmaster
The key's randomart image is:
+---[RSA 3072]----+
|       ..o . |
|       o + + o|
|       . o *o=.|
| . o . .o. oo@o.|
|. * + .S +..*.o|
|+.*= . + . |
|oB E . .
| +* .
| . oo
+---[SHA256]----+
```

Puis copiée vers les autres VMs :

>> **ssh-copy-id root@192.168.27.21**

```
ansible@ansiblmaster:~$ ssh-copy-id ansible@192.168.27.21
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/ansible/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new
ansible@192.168.27.21's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'ansible@192.168.27.21'"
and check to make sure that only the key(s) you wanted were added.
```

>> **ssh-copy-id root@192.168.27.22**

```
ansible@ansiblmaster:~$ ssh-copy-id ansible@192.168.27.22
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/ansible/.ssh/id_rsa"
The authenticity of host '192.168.27.22 (192.168.27.22)' can't be established.
ED25519 key fingerprint is SHA256:1LADtIaemsgg8dT9bPsuaNYmM7eDH+XJqML43Dfg7Vw.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out an
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now
ansible@192.168.27.22's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'ansible@192.168.27.22'"
and check to make sure that only the key(s) you wanted were added.
```

Toutes les VMs doivent pouvoir se ping mutuellement pour assurer la connectivité.

❖ Installation de l'environnement Python et Ansible

Sur **ansiblmaster** :

```
>> apt install python3.11-venv
>> python3 -m venv taskshareenv
>> source ansiblmaster /bin/activate
>> sudo apt update && sudo apt -y upgrade
>> apt install -y build-essential libssl-dev libffi-dev    python3-dev python3-pip
```

```
(taskshareenv) ansible@ansiblmaster:~$ ansible --version
ansible [core 2.18.6]
  config file = None
  configured module search path = ['/home/ansible/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /home/ansible/taskshareenv/lib/python3.11/site-packages/ansible
  ansible collection location = /home/ansible/.ansible/collections:/usr/share/ansible/collections
  executable location = /home/ansible/taskshareenv/bin/ansible
  python version = 3.11.2 (main, Apr 28 2025, 14:11:48) [GCC 12.2.0] (/home/ansible/taskshareenv/bin/python3)
  jinja version = 3.1.6
  libyaml = True
```

Test d'Ansible

```
(taskshareenv) ansible@ansiblēmaster:~/automatisation/taskshare-ansible$ ansible all -i inventory.ini -m ping
[WARNING]: Found both group and host with same name: monitoringhub
[WARNING]: Found both group and host with same name: cicdorchestrator
[WARNING]: Platform linux on host monitoringhub is using the discovered Python interpreter at /usr/bin/python3.11, but fun
core/2.18/reference_appendices/interpreter_discovery.html for more information.
monitoringhub | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.11"
    },
    "changed": false,
    "ping": "pong"
}
[WARNING]: Platform linux on host cicdorchestrator is using the discovered Python interpreter at /usr/bin/python3.11, but
fun
core/2.18/reference_appendices/interpreter_discovery.html for more information.
cicdorchestrator | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.11"
    },
    "changed": false,
    "ping": "pong"
}
(taskshareenv) ansible@ansiblēmaster:~/automatisation/taskshare-ansible$
```

5.2.1 Automatisation de l'installation des services

L'ensemble des playbooks a été centralisé dans un **dépôt GitHub dédié** contenant un fichier d'inventaire, des rôles Ansible et des playbooks organisés par service. Une copie de ce dépôt est présente sur un serveur dédié à Ansible, depuis lequel les déploiements sont exécutés.

```
ansible@ansiblēmaster:~/automatisation/taskshare-ansible$ sudo tree
.
├── README.md
├── dockerplay.yml
├── grafana-prometheus-playbook.yml
├── inventory.ini
├── inventory.ini.save
├── jenkins-playbook.yml
├── k3s-playbook.yml
└── requirements.yml
    └── sonarqube-playbook.yml
    └── sonarqube-scanner.yml

1 directory, 10 files
```

The screenshot shows a GitHub repository named 'taskshare-ansible'. It contains one branch ('main') and no tags. A single commit was made by 'BAKAYOKO BRAHIMA' and 'BAKAYOKO BRAHIMA' titled 'first commit' 2 days ago. The commit includes several files: README.md, dockerplay.yml, grafana-prometheus-playbook.yml, inventory.ini, jenkins-playbook.yml, k3s-playbook.yml, requirements.yml, sonarqube-playbook.yml, and sonarqube-scanner.yml, all of which are marked as 'first commit'.

5.2.2. Installation de Docker

Le premier service installé est Docker, utilisé pour conteneuriser les services applicatifs.

La commande suivante est exécutée depuis le serveur **Ansiblemaster** :

>> ansible-playbook -i inventory.ini dockerplay.yml --ask-become-pass

```
TASK [Afficher la version de Docker] ****
ok: [cicdorchestrator] => {
    "msg": "Docker installé : Docker version 28.2.2, build e6534b4"
}
ok: [monitoringhub] => {
    "msg": "Docker installé : Docker version 28.2.2, build e6534b4"
}

PLAY RECAP ****
cicdorchestrator      : ok=10   changed=7    unreachable=0   failed=0    skipped=0    rescued=0
monitoringhub         : ok=10   changed=7    unreachable=0   failed=0    skipped=0    rescued=0
```

5.2.3. Installation de K3s sur les nœuds

K3s est une distribution légère de Kubernetes conçue pour des environnements de type edge, IoT ou DevOps à ressources limitées. Dans le cadre de ce projet, il est utilisé pour orchestrer les services conteneurisés de manière centralisée et résiliente.

Le déploiement de K3s est automatisé via Ansible. Il est réalisé sur deux machines virtuelles distinctes : une jouant le rôle de **nœud maître (cicdorchestrator)**, et l'autre comme **nœud worker (monitoringhub)**.

Extrait du playbook

```
--  
- name: Install required packages (Debian)  
  hosts: all  
  become: yes  
  tasks:  
    - name: Update package list  
      ansible.builtin.apt:  
        update_cache: yes  
  
    - name: Install curl  
      ansible.builtin.apt:  
        name: curl  
        state: present  
  
- name: Install K3s on Master  
  hosts: cicdhub  
  become: yes  
  tasks:  
    - name: Download and install K3s  
      ansible.builtin.shell: curl -sfL https://get.k3s.io | sh -  
  
    - name: Get Node Token  
      ansible.builtin.shell: cat /var/lib/rancher/k3s/server/node-token  
      register: node_token  
  
- name: Install K3s on Worker  
  hosts: monitoring  
  become: yes  
  tasks:  
    - name: Download and install K3s with Master token  
      ansible.builtin.shell: >  
        curl -sfL https://get.k3s.io | K3S_URL=https://{{ hostvars['cicdhub']['ansible_host'] }}:6443 K3S_TOKEN={{ hostvars['cicdhub']['node_token'] }} | sh -
```

Le playbook **k3s-playbook.yml** se charge :

- d'installer les dépendances nécessaires,
- de télécharger et configurer K3s,
- et de lancer le service K3s selon le rôle de la machine (maître ou agent).

Fichier d'inventaire associé ([inventory.ini](#)) :

```

[all]
cicdorchestrator ansible_host=192.168.27.21 ansible_user=ansible
monitoringhub ansible_host=192.168.27.22 ansible_user=ansible
#devopsmaster ansible_host=192.168.27.63 ansible_user=root

[cicdorchestrator]
cicdorchestrator ansible_host=192.168.27.21 ansible_user=ansible

[monitoringhub]
monitoringhub ansible_host=192.168.27.22 ansible_user=ansible
  
```

Lancement de l'installation

La commande suivante permet de lancer le déploiement automatisé :

>> ansible-playbook -i inventory.ini k3s-playbook.yml --ask-become-pass

```

(taskshareenv) ansible@ansiblemaster:~/automatisation/taskshare-ansible$ ansible-playbook -i inventory.ini k3s-playbook.yml
BECOME password:
[WARNING]: Found both group and host with same name: monitoringhub
[WARNING]: Found both group and host with same name: cicdorchestrator

PLAY [Install required packages (Debian)] ****
TASK [Gathering Facts] ****
[WARNING]: Platform linux on host monitoringhub is using the discovered Python interpreter at /usr/bin/python3.11, but future core/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [monitoringhub]
[WARNING]: Platform linux on host cicdorchestrator is using the discovered Python interpreter at /usr/bin/python3.11, but future core/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [cicdorchestrator]

TASK [Update package list] ****
ok: [monitoringhub]
changed: [cicdorchestrator]

TASK [Install curl] ****
ok: [monitoringhub]
ok: [cicdorchestrator]
[WARNING]: Could not match supplied host pattern, ignoring: cicdhub

PLAY [Install K3s on Master] ****
skipping: no hosts matched
[WARNING]: Could not match supplied host pattern, ignoring: monitoring

PLAY [Install K3s on Worker] ****
skipping: no hosts matched

PLAY RECAP ****
cicdorchestrator      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
monitoringhub         : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
  
```

Vérification de l'installation

Une fois l'exécution du playbook terminée, plusieurs commandes permettent de s'assurer que K3s fonctionne correctement :

Vérification du statut du service :

>> **systemctl status k3s**

```
● root@cicdorchestrator:~# systemctl status k3s
● k3s.service - Lightweight Kubernetes
   Loaded: loaded (/etc/systemd/system/k3s.service; enabled; preset: enabled)
   Active: active (running) since Thu 2025-06-05 12:38:50 UTC; 55s ago
     Docs: https://k3s.io
   Process: 38118 ExecStartPre=/bin/sh -xc ! /usr/bin/systemctl is-enabled --quiet nm-cloud-setup.service 2>/dev/null (code=exited, status=0/SUCCESS)
   Process: 38120 ExecStartPre=/sbin/modprobe br_netfilter (code=exited, status=0/SUCCESS)
   Process: 38122 ExecStartPre=/sbin/modprobe overlay (code=exited, status=0/SUCCESS)
 Main PID: 38123 (k3s-server)
   Tasks: 83
  Memory: 896.6M
    CPU: 24.347s
   CGroup: /system.slice/k3s.service
           ├─38123 "/usr/local/bin/k3s server"
           ├─38150 └─38879 [Open file in editor (ctrl + click)] 10c229b8445ff9999c26495950a5c8b2e08fd13f1889d38f994edeebf026236/bin/containerd-shim-runc-v2 -na
```

Vérification de la connectivité entre les nœuds :

kubectl get nodes

```
● root@cicdorchestrator:~# kubectl get nodes
  NAME      STATUS   ROLES      AGE      VERSION
  cicdorchestrator   Ready    control-plane,master   3m17s   v1.32.5+k3s1
  monitoringhub     Ready    <none>     3m      v1.32.5+k3s1
● root@cicdorchestrator:~#
```

Un retour listant les deux nœuds (**cicdorchestrator** et **monitoringhub**) avec l'état **Ready** confirme le bon fonctionnement du cluster.

5.2.4. Installation de Jenkins

Jenkins est un outil d'intégration continue open source largement utilisé dans les workflows DevOps. Il permet d'automatiser les différentes étapes du cycle de vie logiciel, telles que le build, les tests, le déploiement ou encore la livraison continue. Dans notre infrastructure, Jenkins est installé sur une machine virtuelle dédiée afin de prendre en charge le pipeline CI/CD de l'application **TaskShare**.

L'installation de Jenkins a été automatisée via Ansible pour garantir une reproductibilité et une cohérence dans le déploiement.

Extrait du playbook

```
---
```

```
- name: Installer Jenkins sur Debian 12
  hosts: cicdorchestrator
  become: yes

  tasks:
    - name: Mettre à jour les paquets
      apt:
        update_cache: yes

    - name: Installer Java (OpenJDK 17)
      apt:
        name: openjdk-17-jdk
        state: present

    - name: Ajouter la clé GPG de Jenkins
      apt_key:
        url: https://pkg.jenkins.io/debian-stable/jenkins.io.key
        state: present

    - name: Ajouter le dépôt Jenkins
      apt_repository:
        repo: "deb https://pkg.jenkins.io/debian-stable binary/"
        state: present
        filename: jenkins

    - name: Mettre à jour les paquets après ajout du dépôt
      apt:
        update_cache: yes

    - name: Installer Jenkins
```

La commande suivante permet de lancer le playbook :

>> **ansible-playbook -i inventory.ini jenkins-playbook.yml --ask-become-pass**

```
(taskshareenv) ansible@ansibl...:~/automatisation/taskshare-ansible$ ansible-playbook -i inventory.ini jenkins-playbook.yml --ask-become-pass
BECOME password:
[WARNING]: Found both group and host with same name: monitoringhub
[WARNING]: Found both group and host with same name: cicdorchestrator

PLAY [Installer Jenkins sur Debian 12] ****
TASK [Gathering Facts] ****
[WARNING]: Platform linux on host cicdorchestrator is using the discovered Python interpreter at /usr/bin/python3.11, but core/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [cicdorchestrator]

TASK [Mettre à jour les paquets] ****
changed: [cicdorchestrator]

TASK [Installer Java (OpenJDK 17)] ****
ok: [cicdorchestrator]

TASK [Télécharger la clé GPG officielle de Jenkins] ****
changed: [cicdorchestrator]

TASK [Ajouter le dépôt Jenkins (compatible Debian 12)] ****
changed: [cicdorchestrator]

TASK [Mettre à jour la liste des paquets après ajout du dépôt] ****
ok: [cicdorchestrator]

TASK [Installer Jenkins] ****
changed: [cicdorchestrator]

TASK [Démarrer et activer Jenkins] ****
ok: [cicdorchestrator]

TASK [Attendre que Jenkins crée le mot de passe initial] ****
ok: [cicdorchestrator]

TASK [Afficher le mot de passe initial de Jenkins] ****
ok: [cicdorchestrator]

TASK [Mot de passe initial de Jenkins] ****
ok: [cicdorchestrator] => {
    "msg": "Mot de passe initial : eb4949fa57434fa1bd6de592c77d7939"
}

PLAY RECAP ****
cicdorchestrator      : ok=11   changed=4    unreachable=0   failed=0    skipped=0   rescued=0   ignored=0
```

Une fois l'installation terminée, l'interface de Jenkins devient accessible depuis un navigateur à l'adresse suivante :

<http://192.168.27.21:8080/>

Lors de la première connexion, Jenkins demande un mot de passe de déverrouillage. Celui-ci est affiché directement dans le terminal suite à l'installation ou peut être retrouvé dans le fichier suivant sur la VM Jenkins



Installer les plugins suggérés

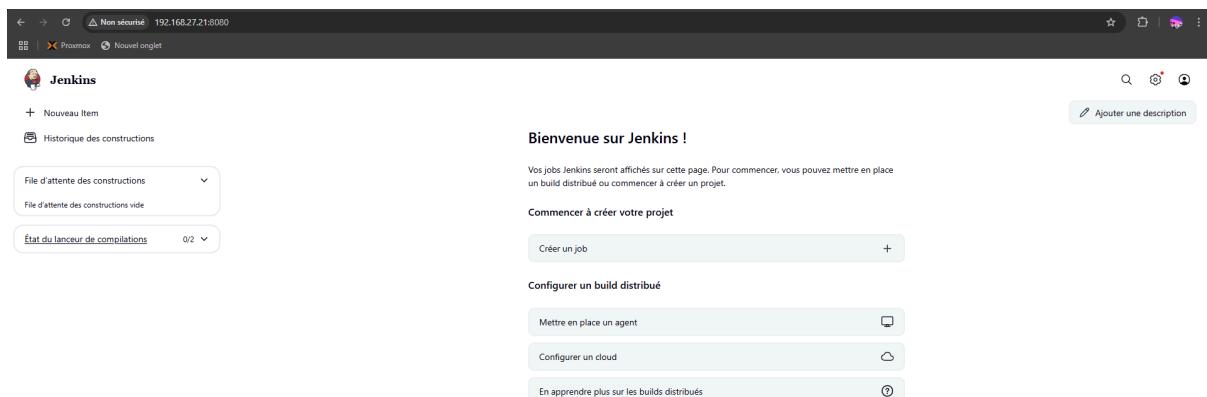


Une fois les plugins installés, créer un compte administrateur Jenkins en renseignant un nom d'utilisateur, mot de passe et email. L'instance Jenkins est ensuite prête à l'emploi.

L'IP par défaut peut être conservée, aucune configuration supplémentaire n'est nécessaire à ce stade.

L'installation se termine par l'affichage du tableau de bord Jenkins, prêt à exécuter des pipelines d'intégration et de déploiement continu.

tableau de bord jenkins



5.2.5. Installation de SonarQube

SonarQube est une plateforme d'analyse statique de code source permettant d'évaluer automatiquement la qualité d'un projet logiciel. Elle détecte des bugs, vulnérabilités, duplications, et problèmes de style ou de maintenabilité. Son rôle est central dans une démarche DevOps orientée qualité logicielle. Dans notre architecture, SonarQube est déployé sur une VM dédiée.

SonarQube ne doit **pas être installé en tant que root** pour des raisons de sécurité. Un utilisateur système spécifique a donc été créé préalablement à l'exécution du playbook Ansible.

Extrait du playbook

```
---
- name: Déployer SonarQube sans installer Java
  hosts: cicdorchestrator
  become: yes

  vars:
    sonarqube_version: "9.0.65466"
    sonarqube_url: "https://binaries.sonarsource.com/Distribution/sonarqube"
    sonarqube_home: "/opt/sonarqube"

  tasks:
    - name: Mettre à jour les paquets
      apt:
        update_cache: yes

    - name: Installer unzip
      apt:
        name: unzip
        state: present

    - name: Créer l'utilisateur sonarqube
      user:
        name: sonarqube
```

La commande suivante permet de lancer le playbook :

```
>> ansible-playbook -i inventory.ini sonarqube-playbook.yml --ask-become-pass
```

```
(taskshareenv) ansible@ansiblemaster:~/automatisation/taskshare-ansible$ ansible-playbook -i inventory.ini sonarqube-playbook.yml --ask-become-pass
BECOME password:
[WARNING]: Found both group and host with same name: monitoringhub
[WARNING]: Found both group and host with same name: cicdorchestrator

PLAY [Déployer SonarQube sans installer Java] ****
TASK [Gathering Facts] ****
[WARNING]: Platform linux on host cicdorchestrator is using the discovered Python interpreter at /usr/bin/python3.11, but future installation of and core/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [cicdorchestrator]

TASK [Mettre à jour les paquets] ****
ok: [cicdorchestrator]

TASK [Installer unzip] ****
changed: [cicdorchestrator]

TASK [Créer l'utilisateur sonarqube] ****
changed: [cicdorchestrator]

TASK [Créer un dossier pour SonarQube] ****
changed: [cicdorchestrator]

TASK [Télécharger SonarQube] ****
changed: [cicdorchestrator]

TASK [Vérifier l'intégrité du fichier SonarQube] ****
ok: [cicdorchestrator]

TASK [fail] ****
skipping: [cicdorchestrator]

TASK [Extraire SonarQube] ****
changed: [cicdorchestrator]
```

Une fois le déploiement terminé, le service peut être vérifié avec :

>> systemctl status sonarqube

```
root@cicdorchestrator:~$ systemctl status sonarqube
● sonarqube.service - file:///cicdorchestrator/etc/systemd/system/sonarqube.service (ctrl + click)
   Loaded: loaded (/etc/systemd/system/sonarqube.service; enabled; preset: enabled)
   Active: active (running) since Wed 2025-06-04 10:03:19 UTC; 2min 56s ago
     Process: 23794 ExecStart=/opt/sonarqube/current/bin/linux-x86-64/sonar.sh start (code=exited, status=0/SUCCESS)
    Main PID: 23847 (java)
       Tasks: 210 (limit: 4686)
      Memory: 1.8G
        CPU: 1min 34.238s
       CGroup: /system.slice/sonarqube.service
```

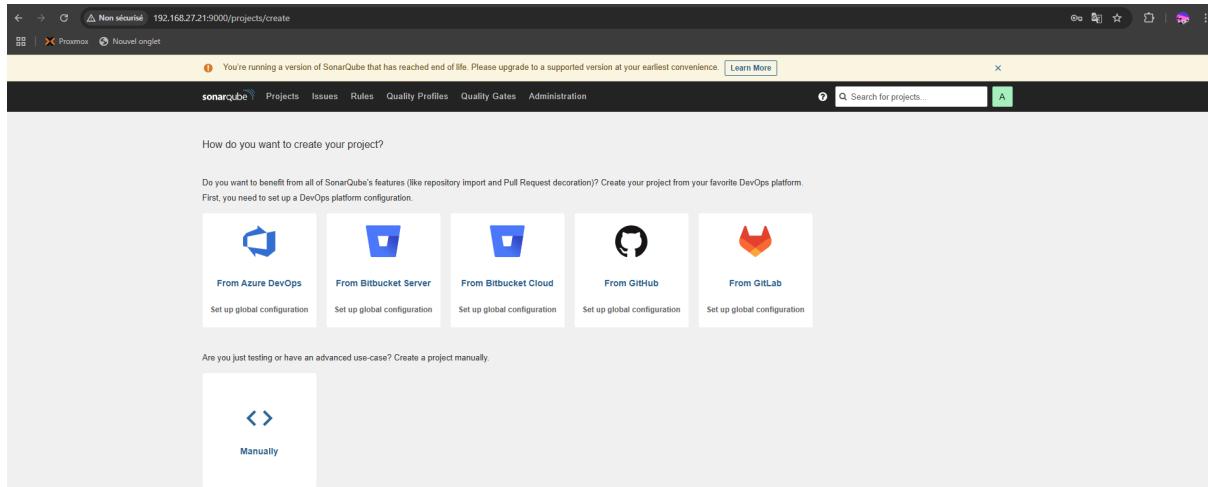
L'interface web de SonarQube devient accessible via l'adresse :

<http://192.168.27.21:9000/>

À la première connexion, les identifiants par défaut sont :

- Utilisateur : admin
- Mot de passe : admin

Un changement de mot de passe est requis à la première utilisation. La page d'accueil s'affiche ensuite avec les options de gestion des projets et des analyses.



5.2.6. Installation de SonarScanner

SonarScanner est l'outil en ligne de commande officiel permettant de lancer des analyses de code vers une instance SonarQube. Dans notre architecture, il est installé directement sur le serveur Jenkins afin d'être utilisé dans les jobs CI pour déclencher automatiquement l'analyse statique du code source.

La commande suivante permet de lancer le playbook :

```
>> ansible-playbook -i inventory.ini sonarqube-playbook.yml --ask-become-pass
```

```
(taskshareenv) ansible@ansiblemaster:~/automatisation/taskshare-ansible$ ansible-playbook -i inventory.ini sonarqube-scanner.yml --ask-become-pass
BECOME password:
[WARNING]: Found both group and host with same name: cicdorchestrator
[WARNING]: Found both group and host with same name: monitoringhub

PLAY [Installer et configurer SonarQube Scanner sur Debian 12 avec Java 17] *****

TASK [Gathering Facts] *****
[WARNING]: Platform linux on host cicdorchestrator is using the discovered Python interpreter at /usr/bin/python3.11, but future installation of ansible/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [cicdorchestrator]

TASK [Mettre à jour les paquets] *****
ok: [cicdorchestrator]

TASK [Installer unzip] *****
ok: [cicdorchestrator]

TASK [Créer le répertoire d'installation de SonarQube Scanner] *****
changed: [cicdorchestrator]

TASK [Télécharger SonarQube Scanner] *****
ok: [cicdorchestrator]

TASK [Extraire le fichier zip de SonarQube Scanner] *****
changed: [cicdorchestrator]

TASK [Créer un lien symbolique pour sonar-scanner dans /usr/local/bin] *****
changed: [cicdorchestrator]

TASK [Ajouter SonarScanner au PATH dans /etc/profile.d] *****
changed: [cicdorchestrator]

TASK [Vérifier l'installation de SonarQube Scanner] *****
ok: [cicdorchestrator]

TASK [Afficher la version de SonarQube Scanner installée] *****
ok: [cicdorchestrator] => {
  "msg": "SonarQube Scanner version: INFO: Scanner configuration file: /opt/sonar-scanner/sonar-scanner-4.8.0.2856-linux/conf/sonar-scanner.properties (64-bit)\nINFO: Linux 6.1.0-33-cloud-amd64 amd64"
}

PLAY RECAP *****
cicdorchestrator : ok=10  changed=4    unreachable=0   failed=0   skipped=0   rescued=0   ignored=0
```

Une fois le playbook terminé, la commande suivante permet de s'assurer que SonarScanner est bien installé :

>> sonar-scanner --version

```
● root@cicdorchestrator:~# sonar-scanner --version
INFO: Scanner configuration file: /opt/sonar-scanner/sonar-scanner-4.8.0.2856-linux/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: SonarScanner 4.8.0.2856
INFO: Java 11.0.17 Eclipse Adoptium (64-bit)
INFO: Linux 6.1.0-33-cloud-amd64 amd64
```

5.3. Déploiement des configurations et gestion des mises à jour

Ansible permet également de réappliquer les playbooks de manière idempotente. Cela signifie que les configurations sont vérifiées et corrigées sans causer de redondance ou d'erreurs, ce qui garantit une infrastructure toujours conforme à l'état souhaité.

5.4. Versionnement et suivi des playbooks sur GitHub

Tous les playbooks, fichiers d'inventaire et rôles sont versionnés dans un **dépôt GitHub** dédié. Cela garantit la traçabilité, le partage et la maintenabilité du code d'automatisation.

6. Déploiement des services applicatifs avec Helm

Le déploiement des services dans le cluster K3s repose sur l'utilisation de **Helm**, le gestionnaire de paquets pour Kubernetes. Dans ce projet, tous les *charts Helm* ont été personnalisés et versionnés dans des dépôts Git distincts. Cela permet une traçabilité complète, une meilleure gestion des mises à jour et une reproductibilité des déploiements. L'installation des services se fait en clonant les dépôts correspondants sur le serveur, puis en exécutant les commandes Helm avec les fichiers `values.yaml` associés.

6.1. Introduction à Helm et aux charts

Helm est un gestionnaire de packages pour Kubernetes qui permet de déployer, gérer et versionner des applications sous forme de *charts*. Ces charts contiennent toutes les ressources nécessaires pour installer un service sur un cluster Kubernetes, comme des **Deployments, Services, Ingress, ConfigMaps**, etc.

Dans ce projet, chaque service applicatif déployé (MetallLB, Galera MariaDB, phpMyAdmin, Prometheus & Grafana) dispose de son propre chart Helm personnalisé, versionné dans un dépôt Git distinct. Le processus de déploiement repose donc sur le clonage de chaque dépôt sur le serveur **cicdorchestrator**, suivi de l'exécution des commandes `helm install`.

Installation de Helm

Avant de pouvoir utiliser Helm, il est nécessaire de l'installer sur le serveur de contrôle (**cicdorchestrator**). Les étapes sont les suivantes :

```
>> sudo apt install apt-transport-https curl gnupg -y

>> curl https://baltocdn.com/helm/signing.asc | sudo gpg --dearmor -o
/usr/share/keyrings/helm.gpg

>> echo "deb [signed-by=/usr/share/keyrings/helm.gpg]
https://baltocdn.com/helm/stable/debian/ all main" | sudo tee
/etc/apt/sources.list.d/helm-stable.list

>> sudo apt update

>> sudo apt install helm -y
```

Une fois l'installation terminée, la commande suivante permet de vérifier que Helm est bien opérationnel :

>> helm version

```
root@cicdorchestrator:/home/taskshareapp/taskshare-backend/metalLB-taskmanager# helm version
version.BuildInfo{Version:"v3.18.1", GitCommit:"f6f8700a539c18101509434f3b59e6a21402a1b2", GitTreeState:"clean", GoVersion:"go1.24.3"}
```

6.2. Déploiement de MetalLB pour la gestion des LoadBalancers

Par défaut, Kubernetes (et donc K3s) ne fournit pas de solution pour exposer les services en type **LoadBalancer** sur une **infrastructure bare-metal**. Pour combler ce manque, **MetalLB** a été intégré afin d'attribuer dynamiquement des adresses IP aux services exposés.

Structure du chart MetalLB personnalisé

Conformément à la documentation officielle, un chart Helm personnalisé a été créé avec l'arborescence suivante :

```
root@cicdorchestrator:/home/taskshareapp/taskshare-backend# tree metalLB-taskmanager/
metalLB-taskmanager/
├── Chart.yaml
├── templates
│   ├── ipaddresspool.yaml
│   └── l2advertisement.yaml
└── values.yaml
```

- **ipaddresspool.yaml** définit la plage d'adresses IP utilisables par MetalLB.
- **l2advertisement.yaml** configure la publicité L2 pour l'annonce des IP sur le réseau local.

Ajout du dépôt Helm officiel de MetalLB

Avant toute installation, le dépôt officiel de MetalLB est ajouté :

>> helm repo add metallb https://metallb.github.io/metallb

>> helm repo update

Installation du composant principal MetalLB

La première étape consiste à déployer MetalLB dans le namespace **metallb-system** :

```
>> helm install metallb metallb/metallb --namespace metallb-system  
--create-namespace
```

```
root@cicdorchestrator:/home/taskshareapp/taskshare-backend/metalLB-taskmanager# helm install metallb metallb/m  
NAME: metallb  
LAST DEPLOYED: Thu Jun 5 14:08:17 2025  
NAMESPACE: metallb-system  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
NOTES:  
MetalLB is now running in the cluster.  
  
Now you can configure it via its CRs. Please refer to the metallb official docs  
on how to use the CRs.
```

Cette commande installe les composants nécessaires (contrôleurs, webhook, etc.).

Déploiement de la configuration spécifique (IP address pool + L2 advertisement)

Une fois MetalLB installé, la configuration réseau personnalisée est appliquée en déployant le chart local :

```
>> helm install matallb ./ -n metallb-system
```

```
root@cicdorchestrator:/home/taskshareapp/taskshare-backend/metalLB-taskmanager# helm install matallb  
NAME: matallb  
LAST DEPLOYED: Thu Jun 5 14:09:36 2025  
NAMESPACE: metallb-system  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
root@cicdorchestrator:/home/taskshareapp/taskshare-backend/metalLB-taskmanager#
```

Le fichier [values.yaml](#) contient notamment :

```
ipRange:  
- "192.168.27.23-192.168.27.25"  
- "192.168.27.31-192.168.27.35"
```

Cette configuration permet à MetalLB d'allouer dynamiquement une IP dans cette plage à chaque service de type **LoadBalancer**.

6.3. Déploiement de Galera MariaDB en haute disponibilité

Pour assurer une **haute disponibilité** et une **tolérance aux pannes** du système de gestion de base de données, MariaDB est déployée en **cluster Galera** dans l'environnement Kubernetes. Ce type de cluster permet une **réPLICATION SYNCHRONE** entre les nœuds, garantissant la cohérence des données en temps réel.

Installation du chart Bitnami

Le déploiement s'appuie sur le **chart Helm officiel de Bitnami** pour MariaDB Galera. Ce chart permet de configurer plusieurs aspects du cluster, notamment le nombre de répliques, les mots de passe root, le nom de la base, etc., via un fichier `galera-values.yaml`.

Ajouter le dépôt Bitnami

Avant de procéder à l'installation, il faut ajouter le dépôt contenant le chart :

```
>> helm repo add bitnami https://charts.bitnami.com/bitnami
```

```
>> helm repo update
```

Cloner les fichiers de configuration

Les fichiers nécessaires au déploiement, y compris `galera-values.yaml`, sont versionnés dans un dépôt Git :

The screenshot shows a GitHub repository named "galeradb". The repository is public and has one branch ("main") and no tags. It contains three files: ".DS_Store", "README.md", and "galera-values.yaml". All three files were committed by "BAKAYOKO BRAHIMA" 5 days ago. The commit message is "first commit". The repository has 2 commits in total.

File	Commit Message	Time Ago
.DS_Store	first commit	5 days ago
README.md	first commit	5 days ago
galera-values.yaml	first commit	5 days ago

galeradb

Déploiement automatisé de Galera Cluster via Helm sur un cluster K3s. Ce dépôt facilite l'installation, la configuration et la gestion de Galera dans un environnement Kubernetes léger.

>> git clone <https://github.com/ibrahimbakayoko/galeradb.git>

>> cd galeradb

Le fichier `galera-values.yaml` contient la configuration personnalisée du cluster :

```
persistence:
  enabled: true
  size: 5Gi

galera:
  bootstrap:
    forceSafeToBootstrap: true
    bootstrapFromNode: "my-galera-mariadb-galera-0" # Utilise un nœud comme point de démarrage

resources:
  requests:
    memory: 1Gi
    cpu: 500m
  limits:
    memory: 2Gi
    cpu: 1000m

service:
  type: ClusterIP
  name: my-galera-mariadb-galera
  ports:
    mysql: 3306
```

Déployer le cluster galera avec Helm

L'installation se fait ensuite avec la commande suivante :

```
>> helm install galeradb bitnami/mariadb-galera -f galera-values.yaml -n taskshare
--create-namespace
```

```
root@cicdorchestrator:/home/galeradb/galeradb# helm install galeradb bitnami/mariadb-galera -f galera-values.yaml
NAME: galeradb
LAST DEPLOYED: Tue Jun 10 10:07:39 2025
NAMESPACE: taskshare
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
  CHART NAME: mariadb-galera
  CHART VERSION: 14.2.7
  APP VERSION: 11.4.7
```

Cette commande crée un namespace dédié (`taskshare`) et déploie un cluster MariaDB Galera à 3 nœuds.

Vérification du déploiement

Une fois le chart installé, on peut vérifier que les trois pods MariaDB Galera sont bien en fonctionnement :

>> kubectl get pods -n taskshare

```
root@cicdorchestrator:/home/galeradb/galeradb# kubectl get pods -n taskshare
NAME                  READY   STATUS    RESTARTS   AGE
galeradb-mariadb-galera-0  1/1     Running   0          4m54s
galeradb-mariadb-galera-1  1/1     Running   0          3m43s
galeradb-mariadb-galera-2  1/1     Running   0          2m45s
```

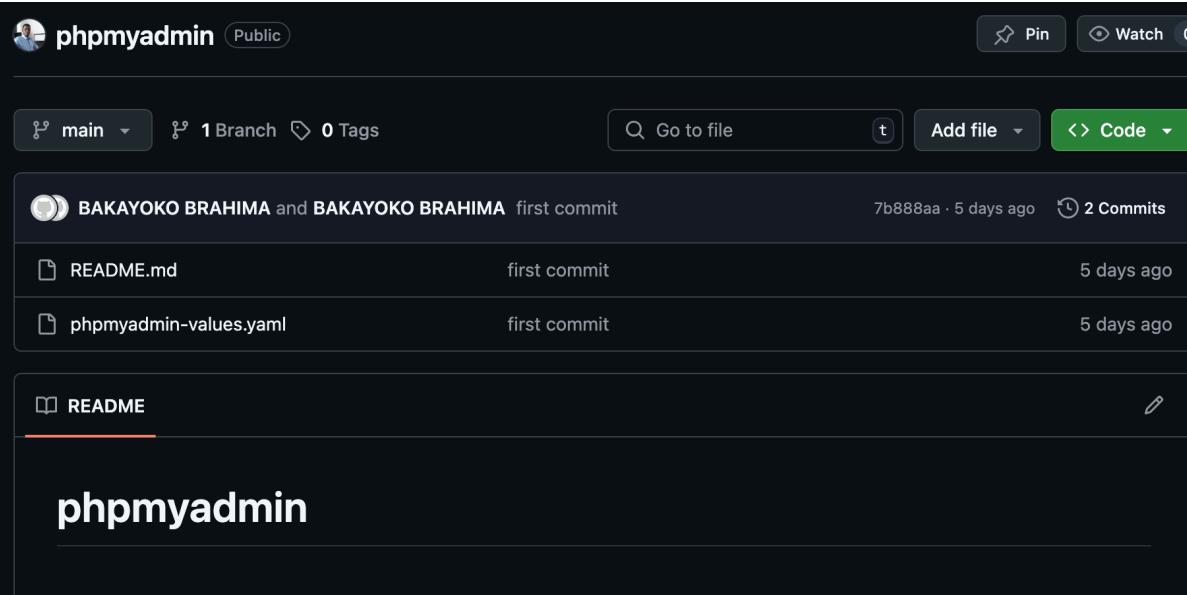
6.4. Déploiement de phpMyAdmin pour l'administration de la base

Afin de faciliter l'administration de la base de données MariaDB déployée en cluster Galera, l'outil **phpMyAdmin** a été intégré au cluster Kubernetes via Helm. phpMyAdmin offre une interface web conviviale pour gérer la base, les utilisateurs, les permissions, etc.

Préparation du chart et du fichier de configuration

Le déploiement repose sur le chart officiel Bitnami, configuré à l'aide d'un fichier personnalisé [phpmyadmin-values.yaml](#), versionné dans un dépôt Git dédié.

Cloner le dépôt contenant la configuration



The screenshot shows a GitHub repository named "phpmyadmin". The repository has 1 branch and 0 tags. The main branch has 2 commits:

- BAKAYOKO BRAHIMA and BAKAYOKO BRAHIMA first commit (7b888aa, 5 days ago)
- first commit (5 days ago)

The repository contains two files:

- README.md (first commit, 5 days ago)
- phpmyadmin-values.yaml (first commit, 5 days ago)

>> git clone https://github.com/ibrahimbakayoko/phpmyadmin.git

>> cd phpmyadmin

À l'intérieur du dépôt, on retrouve le fichier **phpmyadmin-values.yaml** qui contient la configuration personnalisée du service, notamment le service de type LoadBalancer et le lien vers la base de données Galera.

Extrait de **phpmyadmin-values.yaml** :

```

# Configuration du service PHPMyAdmin
service:
  type: LoadBalancer # Utiliser LoadBalancer pour exposer phpMyAdmin
  port: 8282           # Changer Le port d'exposition à 8080 (ou un autre port disponible)

# Configuration de l'Ingress (désactivé ici)
ingress:
  enabled: false      # Ingress désactivé, l'accès se fait via LoadBalancer

# Configuration de la base de données
database:
  host: galeraDB-mariadb-galera.taskshare.svc.cluster.local # Nom du service MariaDB Galera dans Le namespace taskman
  port: 3306           # Port de connexion de MariaDB
  username: root       # Nom d'utilisateur pour La connexion à MariaDB
  password: rootpassword # Mot de passe root de MariaDB (assure-toi qu'il corresponde)
  
```

Déployer phpMyAdmin avec Helm

>> helm install phpmyadmin bitnami/phpmyadmin -f phpmyadmin-values.yaml -n taskshare

Cette commande installe le service phpMyAdmin dans le namespace **taskshare** en se basant sur le fichier de valeurs spécifié.

Accès à l'interface web

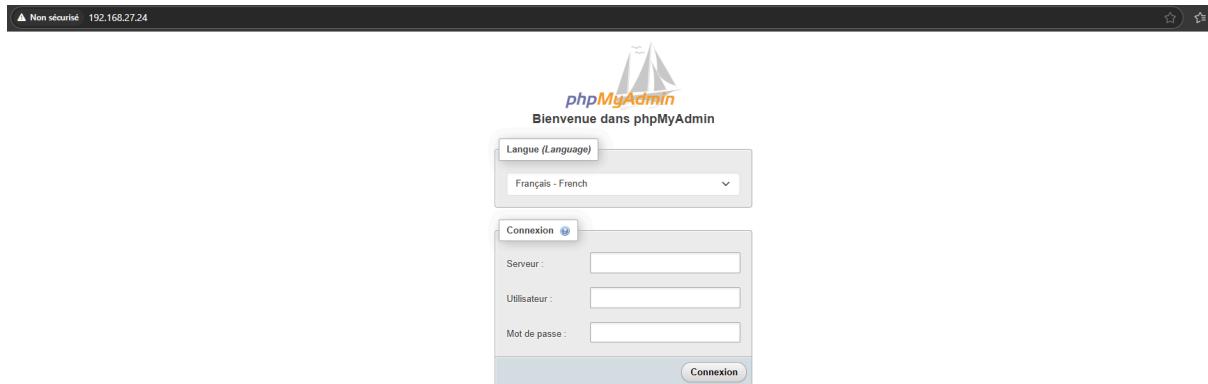
Une fois le pod en cours d'exécution, on peut obtenir l'adresse IP publique attribuée au service LoadBalancer via la commande :

>> kubectl get svc -n taskshare

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
galeraDB-mariadb-galera	ClusterIP	10.43.1.134	<none>	3306/TCP
galeraDB-mariadb-galera-headless	ClusterIP	None	<none>	4567/TCP,4568/TCP,4444/TCP
phpmyadmin	LoadBalancer	10.43.158.234	192.168.27.24	80:31364/TCP,443:31240/TCP

Il suffit ensuite d'ouvrir cette IP dans un navigateur pour accéder à l'interface phpMyAdmin. Les identifiants de connexion sont ceux définis dans MariaDB :

- Serveur : **galeradb-mariadb-galera.taskshare.svc.cluster.local**
- Utilisateur : **user**
- Mot de passe : **password**



Création manuelle de la base et des utilisateurs

Un pod temporaire peut être lancé pour interagir avec le cluster Galera via la ligne de commande :

```
>> kubectl run galeradb-mariadb-galera-client --rm -it --restart='Never' --namespace taskshare \
```

```
--image docker.io/bitnami/mariadb-galera:11.4.7-debian-12-r1 -- \
```

```
mysql -h galeradb-mariadb-galera -P 3306 -uroot -p$(kubectl get secret --namespace taskshare galeradb-mariadb-galera -o jsonpath="{.data.mariadb-root-password}" | base64 -d) my_database
```

Une fois connecté, on peut créer la base et un utilisateur :

```
>> CREATE DATABASE taskmanager;
```

```
>> CREATE USER 'user'@'%' IDENTIFIED BY 'password';
```

```
>> GRANT ALL PRIVILEGES ON taskmanager.* TO 'user'@'%';
```

```
>> FLUSH PRIVILEGES;
```

Pour vérifier les droits accordés :

>> SHOW GRANTS FOR 'user'@'%';

interface web de phpmyadmin après connexion

The screenshot shows the phpMyAdmin interface with the URL 192.168.27.24/index.php?route=/ highlighted in red. The left sidebar shows databases: information_schema and taskmanager, with taskmanager selected. The main panel has two sections: 'Paramètres généraux' and 'Paramètres d'affichage'. In 'Paramètres généraux', there is a 'Modifier le mot de passe' link, an 'Interclassement pour la connexion au serveur' dropdown set to utf8mb4_unicode_ci, and a 'Plus de paramètres' link. In 'Paramètres d'affichage', there is a 'Langue (Language)' dropdown set to Français - French, a 'Thème' dropdown set to pmahomme, and a 'Tout afficher' button.

6.5. Déploiement de Prometheus et Grafana dans le cluster K3s

Dans une architecture DevOps moderne, la supervision est essentielle pour garantir la disponibilité, la performance et la fiabilité des services. Pour cela, les outils **Prometheus** (collecte de métriques) et **Grafana** (visualisation) sont déployés dans le cluster K3s via Helm.

Préparation de l'installation avec Ansible

Le déploiement est automatisé à l'aide d'un playbook Ansible, ce qui garantit une installation reproductible, maintenable et versionnée.

Création du playbook Ansible

Un fichier [grafana-prometheus-playbook.yml](#) a été créé dans le dépôt d'automatisation pour lancer l'installation via Helm.

Extrait du playbook :

```
---
- name: Déploiement de kube-prometheus-stack avec Grafana en LoadBalancer
  hosts: ciдорхestrator
  become: true

  vars:
    helm_repo_name: prometheus-community
    helm_repo_url: https://prometheus-community.github.io/helm-charts
    release_name: kube-prometheus-stack
    release_namespace: monitoring
    chart_name: prometheus-community/kube-prometheus-stack
    chart_version: "56.6.2" # ou adapte à la version souhaitée

  tasks:
    - name: Installer pip3 si absent
      apt:
        name: python3-pip
        state: present

    - name: Installer la lib Python 'kubernetes'
      apt:
        name: python3-kubernetes
        state: present

    - name: Vérifier si Helm est installé
      command: helm version
      register: helm_check
      ignore_errors: true

    - name: Installer Helm si non présent
      shell: curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
      when: helm_check.rc != 0

    - name: Ajouter le repo Helm Prometheus Community
      kubernetes.core.helm_repository:
        name: "{{ helm_repo_name }}"
        repo_url: "{{ helm_repo_url }}"

    - name: Mettre à jour les dépôts Helm
      command: helm repo update
```

Gestion des dépendances Ansible

Un fichier **requirements.yml** permet d'installer les collections nécessaires au bon fonctionnement du playbook :

```
home > task-ansible > ! requierements.yml
1   collections:
2     - name: kubernetes.core
3       version: ">=2.4.0"
4     - name: community.kubernetes
5       version: ">=2.0.0"
6
```

Installer les dépendances :

>> ansible-galaxy collection install -r requirements.yml

```
(taskshareenv) ansible@ansiblemaster:~/automatisation/taskshare-ansible$ ansible-galaxy collection install -r requirements.yml
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Downloading https://galaxy.ansible.com/api/v3/plugin/ansible/content/published/collections/artifacts/community-kubernetes-2.0.1.
'kubernetes.core:5.3.0' is already installed, skipping.
Installing 'community.kubernetes:2.0.1' to '/home/ansible/.ansible/collections/ansible_collections/community/kubernetes'
community.kubernetes:2.0.1 was installed successfully
```

Exécution du playbook

Lancer le déploiement sur le cluster avec la commande :

**>> ansible-playbook -i inventory.ini grafana-prometheus-playbook.yml
--ask-become-pass**

```
TASK [Mettre à jour les dépôts Helm] ****
changed: [cicdorchestrator]

TASK [Créer le namespace monitoring s'il n'existe pas] ****
[WARNING]: kubernetes<24.2.0 is not supported or tested. Some features may not work.
ok: [cicdorchestrator]

TASK [Déployer kube-prometheus-stack avec valeurs personnalisées] ****
changed: [cicdorchestrator]

PLAY RECAP ****
cicdorchestrator : ok=8    changed=3    unreachable=0    failed=0    skipped=1    rescued=0
```

Accès à l'interface Grafana

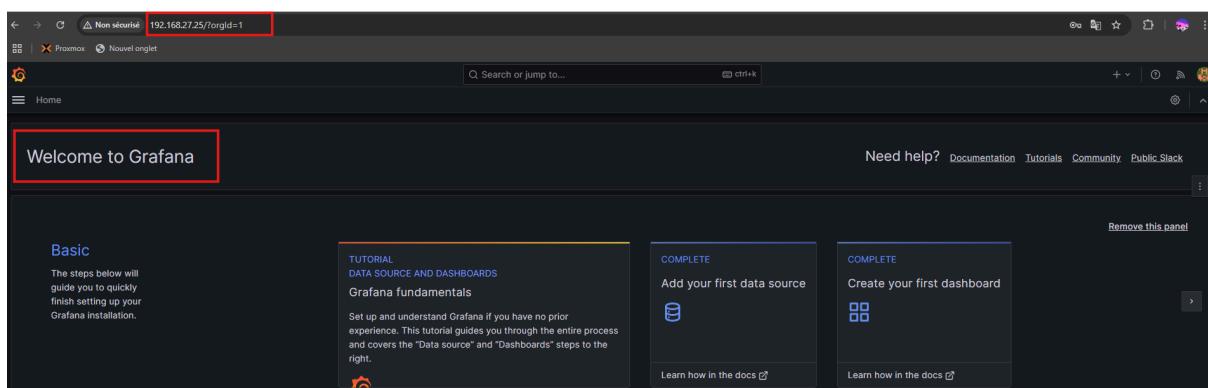
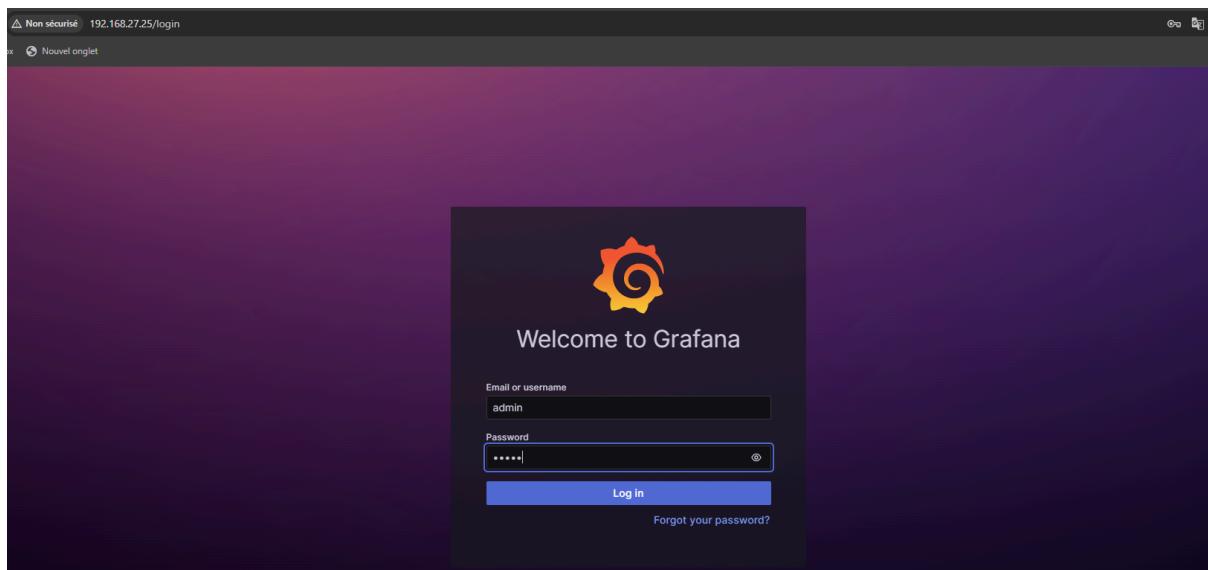
Une fois déployé, on peut vérifier l'IP publique du service Grafana avec la commande :

>> kubectl get svc -n monitoring

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
alertmanager-operated	ClusterIP	None	<none>	9093/TCP,9094/T
kube-prometheus-stack-alertmanager	ClusterIP	10.43.37.121	<none>	9093/TCP,8080/T
kube-prometheus-stack-grafana	LoadBalancer	10.43.176.47	192.168.27.25	80:30778/TCP
kube-prometheus-stack-kube-state-metrics	ClusterIP	10.43.164.189	<none>	8080/TCP
kube-prometheus-stack-operator	ClusterIP	10.43.41.207	<none>	443/TCP
kube-prometheus-stack-prometheus	ClusterIP	10.43.130.179	<none>	9090/TCP,8080/T
kube-prometheus-stack-prometheus-node-exporter	ClusterIP	10.43.210.204	<none>	9100/TCP
prometheus-operated	ClusterIP	None	<none>	9090/TCP

Il suffit ensuite d'ouvrir cette IP dans un navigateur pour accéder à Grafana :

- **Adresse IP : 192.168.27.25**
- **Identifiants par défaut :**
 - **Utilisateur : admin**
 - **Mot de passe : admin**



Une fois connecté, on peut :

- Visualiser les métriques collectées par Prometheus,
- Créer des dashboards personnalisés pour surveiller les services applicatifs, bases de données, et nœuds du cluster.

7. Conteneurisation et packaging de l'application

7.1. Présentation de Docker et bonnes pratiques

Docker est un outil permettant de créer, d'exécuter et de déployer des applications dans des conteneurs légers, portables et isolés. L'utilisation de Docker dans notre projet assure une cohérence entre les environnements de développement, de test et de production.

Bonnes pratiques adoptées :

- Utilisation d'une image Node.js officielle comme base
- Définition explicite du **WORKDIR**
- Copie ciblée des fichiers nécessaires
- Construction de l'image en multi-'layers' optimisés
- Exposition explicite du port
- Déclaration d'une commande de démarrage claire via **CMD**

Extrait du Dockerfile :

```
backend > 🐦 Dockerfile
          BAKAYOKO BRAHIMA, last week | 1 author (BAKAYOKO BRAHIMA)
1   # Utiliser une image de base officielle pour Node.js
2   FROM node:18
3
4   # Créer et définir le répertoire de travail
5   WORKDIR /app
6
7   # Copier les fichiers package.json et package-lock.json
8   COPY package.json package-lock.json ./
9
10  # Installer les dépendances
11  RUN npm install
12
13  # Copier le reste des fichiers du backend
14  COPY . .
15
16  # Exposer le port 5000 pour l'API
17  EXPOSE 5000
18
19  # Commande pour démarrer l'application
20  CMD ["npm", "start"]
21  |
```

7.2. Conteneurisation du service web Taskshare

L'application Taskshare a été conteneurisée avec Docker à partir du fichier [Dockerfile](#). Une image Docker sera construite, taguée et poussée sur Docker Hub pendant l'exécution du pipeline :

Des ajouts fonctionnels ont été réalisés pour permettre l'exposition des métriques à Prometheus, via le module [prom-client](#) :

>> **npm install prom-client**

Ajout dans [server.js](#) :

Modifier le **server.js** pour ajouter le Prometheus client :

```
import client from "prom-client"; // <-- Ajout Prometheus client
```

Ajouter la collecte du CPU, de la mémoire ...

```
client.collectDefaultMetrics();
```

Ajouter un **middleware** pour compter les requêtes :

```
app.use((req, res, next) => {
  res.on("finish", () => {
    httpRequestCounter.inc({
      method: req.method,
      route: req.route?.path || req.path,
      status: res.statusCode
    });
  });
  next();
});
```

Ajouter la route **/metrics** pour Prometheus

```
app.get("/metrics", async (req, res) => {
  res.set("Content-Type", client.register.contentType);
  res.end(await client.register.metrics());
});
```

Cette route **/metrics** permet à Prometheus de collecter les données d'observabilité.

7.3. Ajout de tests, fichiers Jenkinsfile et sonar.properties

Afin de garantir la qualité et la stabilité du code, des tests unitaires ont été mis en place, accompagnés d'une configuration pour l'analyse statique via SonarQube et l'intégration dans le pipeline Jenkins.

a. Mise en place des tests unitaires

Avant toute chose, Node.js doit être installé sur le serveur Jenkins :

```
>> sudo apt update  
>> sudo apt install nodejs  
>> sudo apt install npm
```

Installation des dépendances de test dans le dossier backend :

```
>> npm install --save-dev nyc mocha mocha-junit-reporter
```

Création d'un répertoire **test** contenant un fichier d'exemple **basic.test.js**. Celui-ci vérifie le bon fonctionnement de base de l'application :

```
backend > test > js basic.test.js > ...  
1 // backend/test/basic.test.js  
2 // const assert = require('assert');  
3  
4 import assert from 'assert';  
5  
6 describe('Test logique de base', function() {  
7   it('devrait valider que 1 + 1 égale 2', function() {  
8     assert.strictEqual(1 + 1, 2);  
9   });  
10  
11  it('devrait valider que "toto" est une chaîne de caractères', function() {  
12    assert.strictEqual(typeof "toto", 'string');  
13  });  
14});
```

Le test peut être exécuté avec la commande :

```
>> npx mocha
```

Des rapports de couverture peuvent également être générés via **nyc** :

```
>> npx nyc mocha
```

b. Jenkinsfile

Le fichier [Jenkinsfile](#) a été ajouté à la racine du projet Voici un extrait :

```
Jenkinsfile
You, 5 days ago | 2 authors (BAKAYOKO BRAHIMA and one other)
pipeline {
    agent any

    environment {
        DOCKER_IMAGE = 'dnais1210/taskshare'
        DOCKER_CREDENTIALS = 'dockerhub-taskshare'
        SONARQUBE_ENV = 'SonarQube'
        SONARQUBE_TOKEN = credentials('taskshare-token')
        SONAR_HOST_URL = 'http://192.168.27.21:9000/'
        KUBECONFIG_PATH = '/var/lib/jenkins/.kube/config'
    }

    stages {
        stage('Cloner le code') {
            steps {
                git branch: 'main', url: 'https://github.com/ibrahimbakayoko/taskshare-app.git'
            }
        }

        stage('Tests & couverture') {
            steps {
                dir('backend') {
                    sh 'npm install'
                    sh 'npx nyc --reporter=lcov --reporter=text mocha ./test/basic.test.js --re
                }
            }
        }
    }
}
```

c. Analyse statique avec SonarQube

Le fichier **sonar-project.properties** a été ajouté à la racine du projet :

```
❶ sonar-project.properties
 1 # Identifiant unique du projet sur SonarQube
 2 sonar.projectKey=taskmanager
 3
 4 # Nom affiché dans l'interface SonarQube
 5 sonar.projectName=TaskManager
 6
 7 # Version du projet (optionnelle)
 8 sonar.projectVersion=1.0
 9
10 # Répertoire contenant le code source à analyser
11 sonar.sources=backend
12
13 # Langue principale (Node.js = JavaScript)
14 sonar.language=js
15
16 # Exclure les fichiers générés ou non pertinents
17 sonar.exclusions=**/node_modules/**,**/test/**,**/*.spec.js
18
19 # Encodage des fichiers source
20 sonar.sourceEncoding=UTF-8
21
22 # Token (sera injecté via le pipeline, donc ne pas le mettre ici)
23 # sonar.login=<ne pas mettre ici si tu utilises Jenkins avec credentials>
```

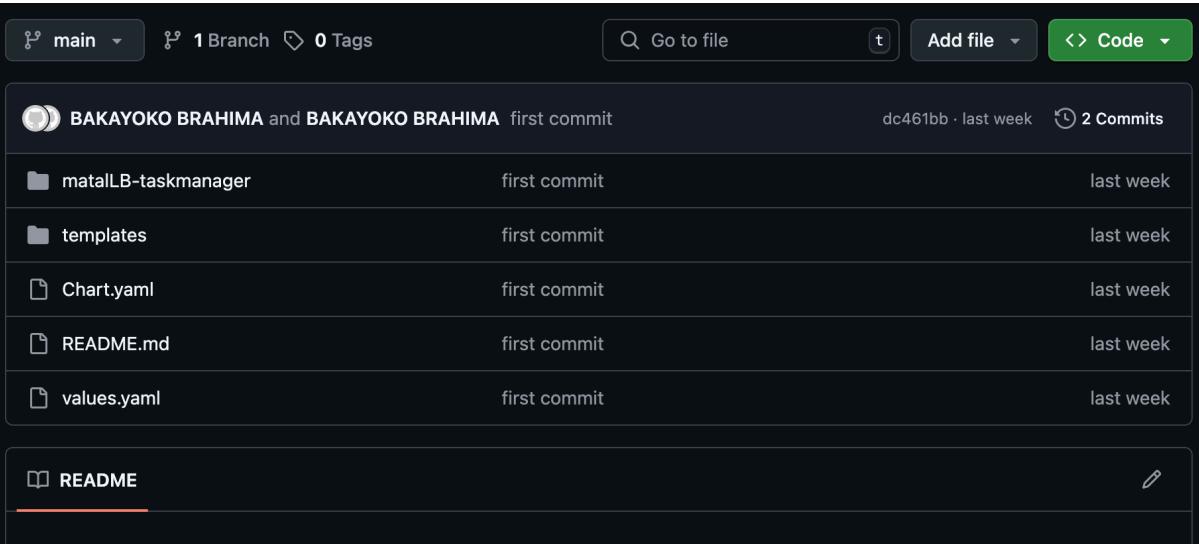
Dans le pipeline, l'analyse est déclenchée ainsi :

```
stage('Analyse SonarQube') {
    steps {
        withSonarQubeEnv('SonarQube') {
            sh 'sonar-scanner'
        }
    }
}
```

7.4. Création d'un Helm chart personnalisé pour l'application

Un chart Helm personnalisé a été développé pour faciliter le déploiement de l'application Taskshare dans un cluster Kubernetes K3s. Il est disponible dans un dépôt GitHub dédié :

Dépôt : <https://github.com/ibrahimbakayoko/taskshare-backend.git>



BAKAYOKO BRAHIMA and BAKAYOKO BRAHIMA first commit dc461bb · last week 2 Commits

File	Commit Message	Time
mataLB-taskmanager	first commit	last week
templates	first commit	last week
Chart.yaml	first commit	last week
README.md	first commit	last week
values.yaml	first commit	last week

README

Structure du chart :

```
● root@cicdorchestrator:/home/taskshare-backend/taskshare-backend# tree
.
├── Chart.yaml
├── README.md
└── matallLB-taskmanager
    ├── Chart.yaml
    ├── templates
    │   ├── ipaddresspool.yaml
    │   └── l2advertisement.yaml
    ├── values.yaml
    ├── templates
    │   ├── deployment.yaml
    │   ├── ingress.yaml
    │   ├── service.yaml
    │   └── servicemonitor.yaml
    ├── values-prod.yaml
    ├── values-test.yaml
    └── values.yaml
```

Fonctionnalités du chart :

- Configuration paramétrable de l'image, du nombre de réplicas, des variables d'environnement
- Exposition par service **LoadBalancer** ou **Ingress**
- Déploiement multi-environnements grâce à l'utilisation de fichiers de valeurs spécifiques (**values.yaml**, **values-test.yaml**, **values-prod.yaml**)
- Nom du namespace passé dynamiquement à l'installation

NB : Le pipeline Jenkins inclus les étapes pour :

- **Déploiement automatique en environnement de test** après validation des tests et de l'analyse SonarQube
- **Déploiement conditionnel en production** après une approbation manuelle

Cette stratégie garantit une séparation stricte entre les environnements, avec validation humaine avant la mise en production. Elle répond aux exigences d'un déploiement maîtrisé, sécurisé et conforme aux pratiques DevOps.

8. Automatisation CI/CD avec Jenkins et Docker

8.1. Présentation de l'intégration et du déploiement continus

L'intégration continue (CI) et le déploiement continu (CD) constituent un pilier fondamental de la méthodologie DevOps. Leur objectif est d'automatiser les phases de compilation, test, analyse, packaging, et déploiement d'une application, afin d'améliorer la fréquence et la fiabilité des mises en production.

Dans le cadre de notre projet Taskshare, Jenkins a été choisi comme serveur d'intégration continue pour orchestrer les différentes étapes du pipeline CI/CD. Docker est utilisé pour construire et gérer les conteneurs applicatifs.

8.2. Configuration de Jenkins et Webhooks GitHub

Jenkins a été déployé via un playbook Ansible. Une fois opérationnel, nous allons :

- Installer les plugins nécessaires : GitHub, Docker, Docker pipeline, SonarQube Scanner, etc.
- Créer des identifiants Jenkins pour se connecter à DockerHub et SonarQube
- Connecté le dépôt GitHub de l'application Taskshare via un webhook :
 - **Settings > Webhooks > http://<IP_JENKINS>:8080/github-webhook/**

Chaque push sur le dépôt devra déclencher automatiquement l'exécution du pipeline défini dans le **Jenkinsfile** du projet.

Configuration de SonarQube pour Jenkins

1. Installer le plugin SonarQube Scanner :

- Jenkins > Gérer Jenkins > Gestion des plugins > SonarQube Scanner

The screenshot shows the Jenkins plugin management interface. A search bar at the top contains the text "Nom ↓". Below it, a list item for "SonarQube Scanner for Jenkins 2.18" is shown. The description below the list item reads: "This plugin allows an easy integration of SonarQube, the open source platform for Continuous Inspection of code quality. Report an issue with this plugin".

2. Configurer SonarQube dans Jenkins :

- Jenkins > Gérer Jenkins > Configurer le système

Tableau de bord > Administrer Jenkins > System >

SonarQube servers

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

Environment variables

Installations de SonarQube

Liste des installations de SonarQube

Ajouter une installation SonarQube

-

- Section « SonarQube servers » > Ajouter une installation
- Nom : **SonarQube**
- URL : **http://192.168.27.21:9000**
- Authentification : aucune

Installations de SonarQube

Liste des installations de SonarQube

Nom

SonarQube

URL du serveur

Par défaut à http://localhost:9000

http://192.168.27.21:9000/

Server authentication token

SonarQube authentication token. Mandatory when anonymous access is disabled.

- aucun -

3. Générer un Token dans SonarQube :

- Se connecter à l'interface : <http://192.168.27.21:9000>
- Aller sur « My Account » > « Security »

Tokens

If you want to enforce security by not providing credentials of a real SonarQube user to run your code scan or to invoke web services, you can provide a User Token as a replacement of the user login. This will increase the security of your installation by not letting your analysis user's password going through your network.

Generate Tokens

Name	Type	Expires in
<input type="text" value="Enter Token Name"/>	<input type="button" value="Select Token Type"/>	<input type="button" value="30 days"/>
<input type="button" value="Generate"/>		

! New token "taskshare" has been created. Make sure you copy it now, you won't be able to see it again!

Name	Type	Project	Last use	Created	Expiration
taskshare	User		Never	June 10, 2025	-
<input type="button" value="Revoke"/>					

- Créer un **User Token**

- NB : les *users tokens* sont utilisés pour l'authentification d'utilisateur, contrairement aux *project analysis tokens* (spécifiques à un projet) ou *global analysis tokens* (pour analyses globales)

4. Ajouter le Token dans Jenkins :

- Jenkins > Gérer Jenkins > Credentials > Add Credentials
- Type : Secret Text
- Secret : coller le token
- ID : **taskshare-token**

New credentials

Type

Portée ?

Secret

ID ?

Description ?

Create

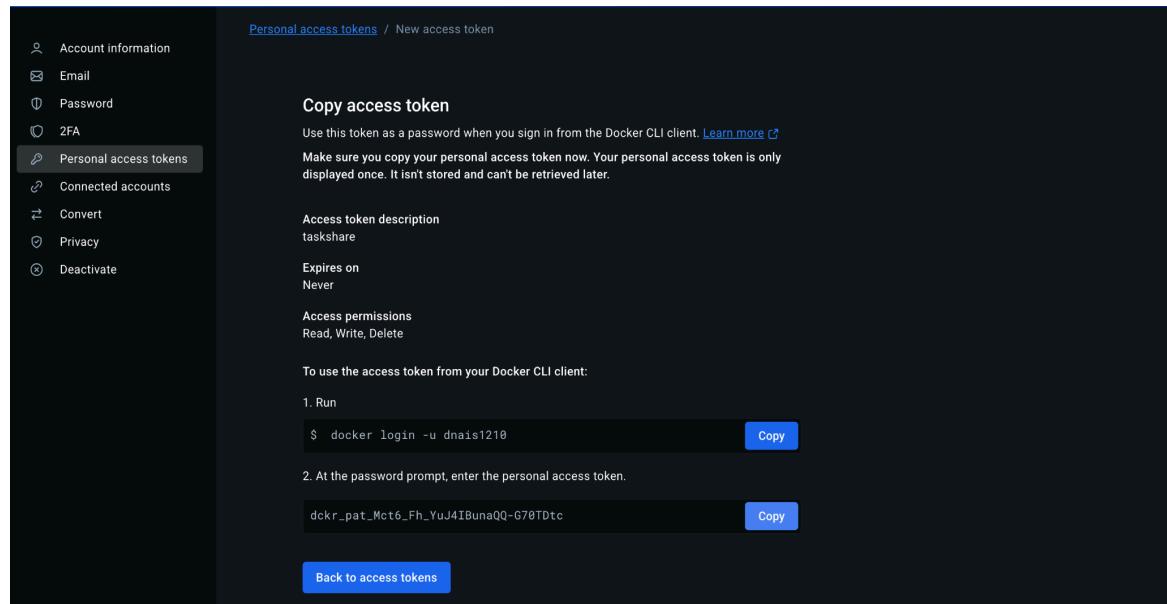
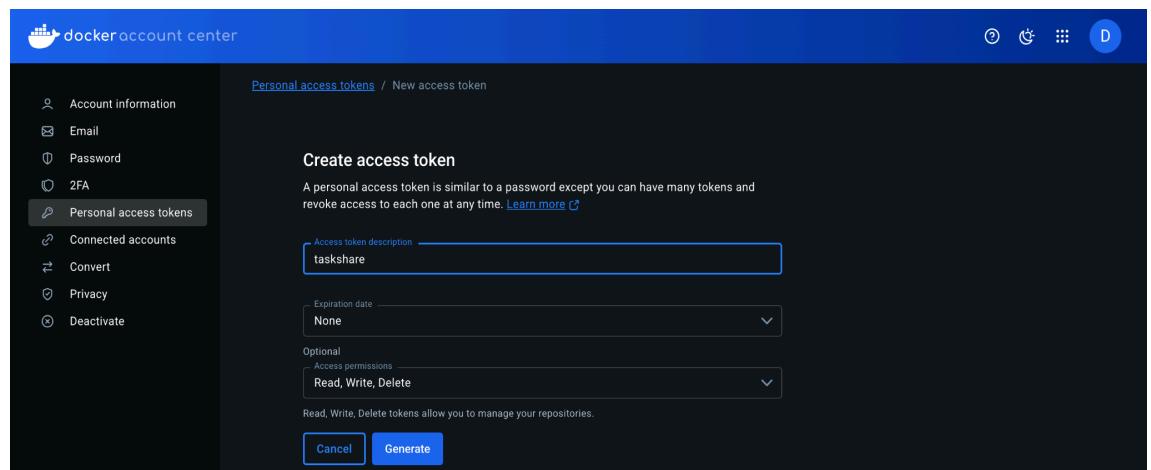
Installation et configuration de DockerHub

1. Créer un compte DockerHub et un dépôt nommé **taskshare**.

The screenshot shows the Docker Hub 'My Hub' interface. On the left, there's a sidebar with options like 'Repositories', 'Collaborations', 'Settings', 'Default privacy', 'Notifications', 'Billing', 'Usage', 'Pulls', and 'Storage'. The main area displays the 'dnais1210/taskshare' repository under 'General'. It shows the repository was last pushed about 1 month ago and has a size of 382.6 MB. There are tabs for 'General', 'Tags', 'Image Management (BETA)', 'Collaborators', 'Webhooks', and 'Settings'. Under 'General', there's a 'Tags' section showing one tag: 'latest'. A note says 'This repository contains 1 tag(s.)'. To the right, there's a 'Docker commands' section with the command 'docker push dnais1210/taskshare:tagname'. Below it is a 'buildcloud' advertisement.

2. Générer un token d'accès personnel :

- DockerHub > Settings > Security > Personal Access Tokens > Generate New Token



3. Ajouter les identifiants dans Jenkins :

- Jenkins > Gérer Jenkins > Credentials > Add Credentials
- Type : Username and Password
- Username : votre identifiant DockerHub
- Password : coller le token généré
- ID : [docker-hub](#)

New credentials

Type
Nom d'utilisateur et mot de passe

Portée ?
Global (Jenkins, agents, items, etc...)

Nom d'utilisateur ?
yacine78

Treat username as secret ?

Mot de passe ?
.....

ID ?
credentials-dockerhub

Description ?

[Create](#)

Intégration Webhook GitHub avec Jenkins via ngrok

Installer **ngrok** sur Debian 12

```
wget https://bin.equinox.io/c/bNyj1mQVY4c/ngrok-v3-stable-linux-amd64.tgz
```

```
tar -xvzf ngrok-v3-stable-linux-amd64.tgz
```

```
sudo mv ngrok /usr/local/bin/
```

```
sudo wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
sudo unzip ngrok-stable-linux-amd64.zip
sudo rm ngrok-stable-linux-amd64.zip
--2025-05-26 11:53:09-- https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
Resolving bin.equinox.io (bin.equinox.io)... 13.248.244.96, 99.83.220.108, 35.71.179.82, ...
Connecting to bin.equinox.io (bin.equinox.io)|13.248.244.96|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13921656 (13M) [application/octet-stream]
Saving to: 'ngrok-stable-linux-amd64.zip'

ngrok-stable-linux-amd64.zip                                              100%[=====]
2025-05-26 11:53:18 (1.81 MB/s) - 'ngrok-stable-linux-amd64.zip' saved [13921656/13921656]

Archive:  ngrok-stable-linux-amd64.zip
  inflating: ngrok
```

Lier le compte ngrok avec :

ngrok config add-authtoken <VOTRE_TOKEN>

Crée un compte gratuit sur <https://dashboard.ngrok.com>, puis récupère ton **authtoken**.

Aller dans “**Tunnel Agent Authtokens**”, et “**Add Tunnel Authtokens**”

Ensuite, configure ngrok :

ngrok config add-authtoken cr_2xdHz4c7K8TnZhuPaEQtaDjap0C

Exposer Jenkins localement via ngrok

>> ngrok http 8080

Version	3.22.1
Region	Europe (eu)
Latency	13ms
Web Interface	http://127.0.0.1:4040
Forwarding	https://05c8-80-118-148-154.ngrok-free.app -> http://localhost:8080
Connections	ttl opn rt1 rt5 p50 p90 0 0 0.00 0.00 0.00 0.00

Exemple d'URL publique fournie : <https://05c8-80-118-148-154.ngrok-free.app/>

Aller dans le pipeline Jenkins, dans configuration, puis trigger et cocher :

Triggers

Set up automated actions that start your build based on specific events, like code changes or scheduled times.

- Construire après le build sur d'autres projets ?
- Construire périodiquement ?
- GitHub hook trigger for GITScm polling ?
- Scrutation de l'outil de gestion de version ?
- Déclencher les builds à distance (Par exemple, à partir de scripts) ?

Étape 3 – Configurer le webhook GitHub

- GitHub > Repository > Settings > Webhooks > Add Webhook
- Payload URL : https://<ngrok_url>/github-webhook/
- Content Type : **application/json**
- Event : **Just the push event**

[Webhooks](#) / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

Content type *

Secret

SSL verification
 By default, we verify SSL certificates when delivering payloads.
 Enable SSL verification Disable (not recommended)

Which events would you like to trigger this webhook?

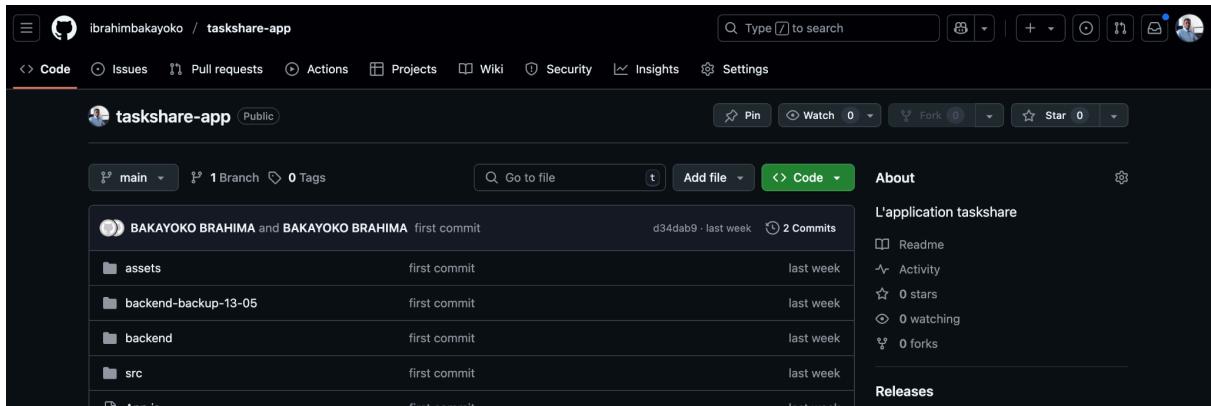
Just the push event.
 Send me everything.
 Let me select individual events.

Active
We will deliver event details when this hook is triggered.

Add webhook

Création et configuration du Jenkinsfile

- Créer un dépôt GitHub pour le projet (ex: <https://github.com/ibrahimbakayoko/taskshare-app.git>).



- Installer Git

Cloner le dépôt :

- git clone <https://github.com/ibrahimbakayoko/taskshare-app.git>
- Identifier le fichier **Jenkinsfile** à la racine du projet.
- Sur Jenkins :
 - Créer un nouvel item > **Pipeline**
 - Définir le pipeline via **Pipeline script from SCM**
 - SCM : Git
 - Repository URL : <https://github.com/ibrahimbakayoko/taskshare-app.git>
 - Branch Specifier : ***/main**
 - Script Path : **Jenkinsfile**

Rapport de stage

Brahima BAKAYOKO

 Jenkins / Tous / Nouveau Item

Nouveau Item

Saisissez un nom

taskshare

Select an item type



Construire un projet free-style

Job legacy polyvalent qui récupère l'état depuis un outil de gestion de version au plus, exécute les étapes de build en série, suivi d'étapes post-construction telles que l'archivage d'artefacts et l'envoi de notifications par e-mail.



Pipeline

Organise des activités de longue durée qui peuvent s'étendre sur plusieurs agents de construction. Adapté pour la création des pipelines (anciennement connues comme workflows) et/ou pour organiser des activités complexes qui ne s'adaptent pas facilement à des tâches de type libre.



Construire un projet multi-configuration

Adapté aux projets qui nécessitent un grand nombre de configurations différentes, comme des environnements de test multiples, des binaires spécifiques à une plateforme, etc.



Dossier

Crée un conteneur qui stocke des objets imbriqués. Utile pour grouper ensemble des éléments. Contrairement à une vue qui n'est qu'un filtre, un dossier crée un espace de nommage distinct, de sorte que vous pouvez avoir plusieurs éléments du même nom tout en ils se trouvent dans des dossiers différents.

Definition

Pipeline script from SCM

SCM ?

Git

Repositories ?

Repository URL ?

`https://github.com/ibrahimbakayoko/taskshare-app.git`

Credentials ?

- aucun -

+ Ajouter

Avancé ▾

Add Repository

Branches to build ?

Branch Specifier (blank for 'any') ?

`*/*main`

8.3. Création du pipeline CI/CD

Le fichier [Jenkinsfile](#) défini à la racine du dépôt GitHub décrit toutes les étapes d'automatisation.

8.3.1. Build de l'image Docker de l'application

```
stage('Construire l'image Docker') {
    steps {
        sh """
            docker build -t $DOCKER_IMAGE:$BUILD_NUMBER -t $DOCKER_IMAGE:latest backend/
        """
    }
}
```

8.3.2. Analyse statique avec SonarScanner

```
stage('Analyse SonarQube') {
    steps {
        withSonarQubeEnv("${SONARQUBE_ENV}") {
            sh """
                sonar-scanner \
                -Dsonar.projectKey=taskmanager \
                -Dsonar.sources=./backend \
                -Dsonar.javascript.lcov.reportPaths=./backend/coverage/lcov.info \
                -Dsonar.host.url=${SONAR_HOST_URL} \
                -Dsonar.login=${SONARQUBE_TOKEN}
            """
        }
    }
}
```

8.3.3. Push de l'image sur Docker Hub

```
stage('Pousser sur Docker Hub') {
    steps {
        withDockerRegistry([credentialsId: "$DOCKER_CREDENTIALS", url: ""]){
            sh """
                docker push $DOCKER_IMAGE:$BUILD_NUMBER
                docker push $DOCKER_IMAGE:latest
            """
        }
    }
}
```

8.3.4. Déploiement automatique vers l'environnement de test (K3s)

```
stage('Déploiement sur TEST') {
    steps {
        withEnv(["KUBECONFIG=$KUBECONFIG_PATH"]){
            sh """
                helm upgrade --install taskshare-backend /home/taskshare-backend/taskshare-backend \
                -f /home/taskshare-backend/taskshare-backend/values-test.yaml \
                --set image.repository=$DOCKER_IMAGE \
                --set image.tag=$BUILD_NUMBER \
                -n test --create-namespace
            """
        }
    }
}
```

8.3.5. Promotion vers l'environnement de production

```
stage('Approval for Production') {
    steps {
        input message: "Déployer en production ?"
    }
}
```

```
stage('Déploiement sur PRODUCTION') {
    steps {
        withEnv(["KUBECONFIG=$KUBECONFIG_PATH"]) {
            sh """
                helm upgrade --install taskshare-backend /home/taskshare-backend/taskshare-backend \
                -f /home/taskshare-backend/taskshare-backend/values-prod.yaml \
                --set image.repository=$DOCKER_IMAGE \
                --set image.tag=$BUILD_NUMBER \
                -n taskshare --create-namespace
            """
        }
    }
}
```

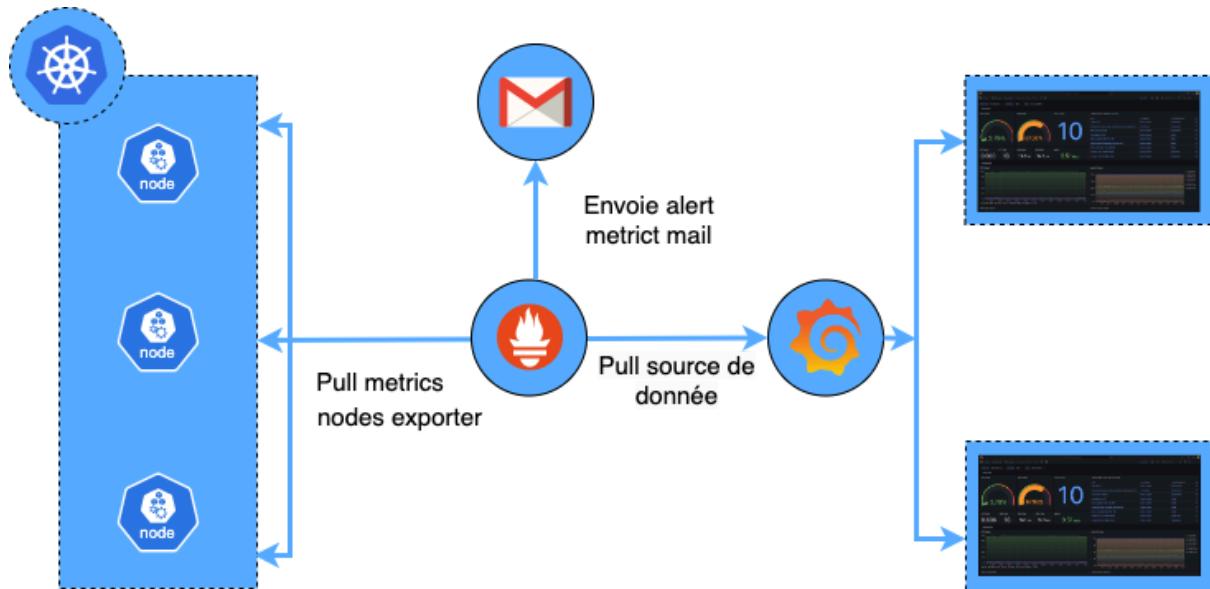
8.4. Notifications, logs et suivi des pipelines

Les pipelines Jenkins offrent une visibilité détaillée sur chaque étape :

- Affichage des logs en temps réel
- Suivi graphique
- Possibilité d'ajouter des notifications via plugins (email, Slack, etc.)

Cette automatisation complète assure une livraison continue, rapide et fiable de l'application Taskshare, tout en garantissant la qualité du code grâce à l'analyse SonarQube.

9. Supervision et monitoring des services



9.1. Introduction à Prometheus et Grafana

Dans une architecture distribuée et conteneurisée comme celle du projet Taskshare, la supervision des services est cruciale pour garantir la fiabilité, la performance et la résilience du système. Pour cela, nous avons adopté deux outils principaux :

- **Prometheus** : un système open-source de collecte de métriques et de surveillance, particulièrement adapté aux environnements cloud-native.
- **Grafana** : une plateforme de visualisation et d'analyse de données, utilisée pour afficher les métriques collectées par Prometheus à travers des tableaux de bord dynamiques et personnalisables.

L'association de Prometheus et Grafana permet d'avoir une vue temps réel de l'état des services et d'anticiper les défaillances.

9.2. Surveillance du cluster K3s

La stack de monitoring a été déployée dans le cluster K3s à l'aide de Helm charts communautaires :

- Le chart **kube-prometheus-stack** de Bitnami a été utilisé pour installer Prometheus Operator, Alertmanager, Grafana, node-exporter et kube-state-metrics.
- L'installation a été réalisée dans un namespace dédié : **monitoring**.

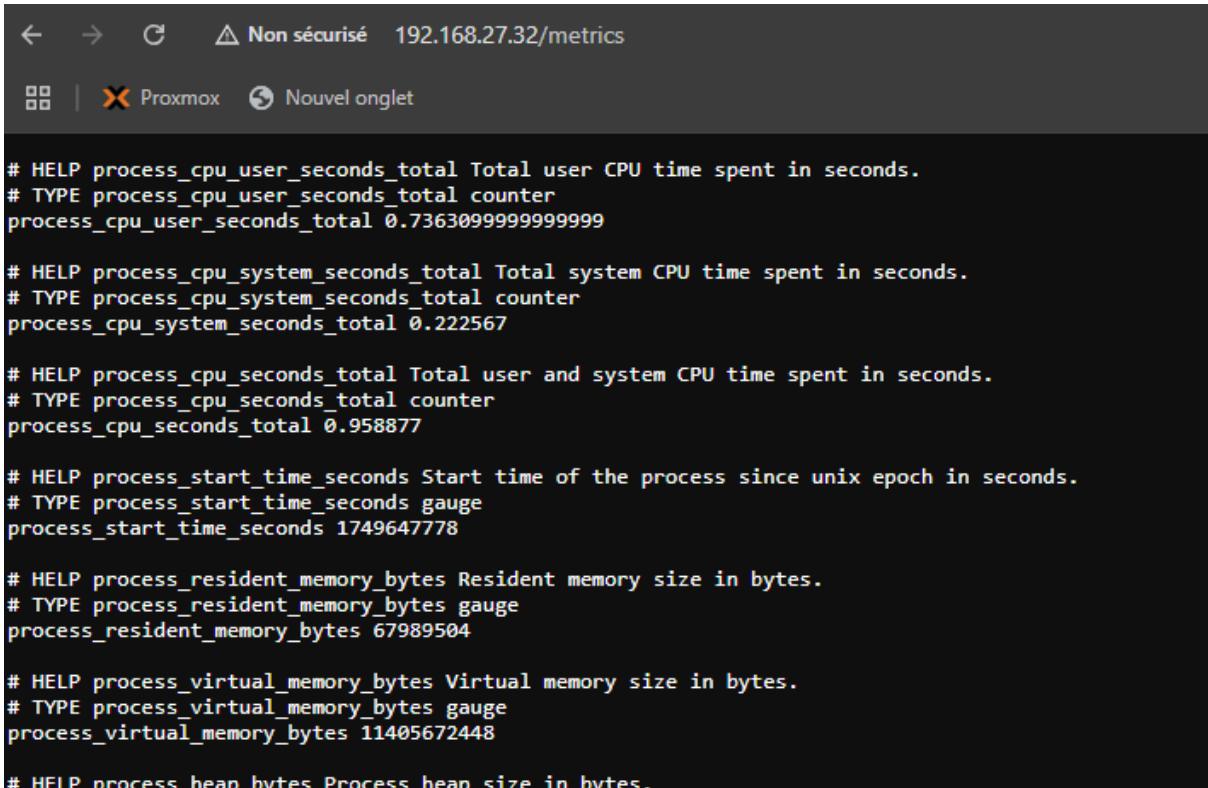
Cette stack permet de surveiller :

- L'état des nœuds du cluster
- La consommation des ressources CPU/mémoire
- Les statuts des pods, services et namespaces
- Les évènements Kubernetes

9.3. Monitoring de l'application TaskShare

L'application Node.js expose des métriques au format Prometheus via le module **prom-client** sur l'endpoint **/metrics**.

Étapes : voir le chapitre empaquetage de l'app



The screenshot shows a terminal window with the following content:

```
← → ⌂ Non sécurisé 192.168.27.32/metrics
[[{"process": "node", "cpu_user_seconds_total": 0.7363099999999999, "cpu_system_seconds_total": 0.222567, "cpu_seconds_total": 0.958877, "start_time_seconds": 1749647778, "resident_memory_bytes": 67989504, "virtual_memory_bytes": 11405672448}], [{"process": "node"}]]
```

The terminal window has a dark background and light-colored text. It displays the output of a curl command to the /metrics endpoint of a node process. The output is a JSON object containing two arrays: one for metrics and one for labels. The metrics array contains a single entry for the node process, with values for various CPU and memory metrics. The labels array also contains a single entry for the node process.

9.4. Monitoring de Galera MariaDB

La base de données Galera MariaDB a été déployée à l'aide du chart Helm [bitnami/mariadb-galera](#), qui inclut un exporter Prometheus.

Étapes :

1. Activer l'exporter dans le fichier de valeurs [galera-values.yaml](#) :

metrics:

enabled: true

serviceMonitor:

enabled: true

2. Déploiement avec Helm :

```
>> helm install galera bitnami/mariadb-galera -f galera-values.yaml -n database
```

3. Vérification de la détection par Prometheus et visualisation dans Grafana.

Des dashboards communautaires pour MariaDB sont disponibles sur le site officiel de Grafana et ont été importés pour surveiller :

- Le nombre de requêtes par seconde
- L'état du cluster Galera
- Le temps moyen d'exécution des requêtes

9.5. Alerting et gestion proactive des incidents

Le système d'alerting repose sur Alertmanager, intégré à la stack Prometheus. Des règles d'alerte ont été configurées pour détecter des anomalies :

- Pod en crashloop
- Utilisation CPU ou mémoire au-delà d'un seuil
- Service non disponible
- Nœud injoignable

Exemple de règle :

- alert: HighMemoryUsage

```
expr: node_memory_Active_bytes / node_memory_MemTotal_bytes > 0.9
```

for: 5m

labels:

```
severity: warning
```

annotations:

```
summary: "Utilisation mémoire > 90% sur {{ $labels.instance }}"
```

Les alertes sont redirigées par Alertmanager vers des canaux configurés :

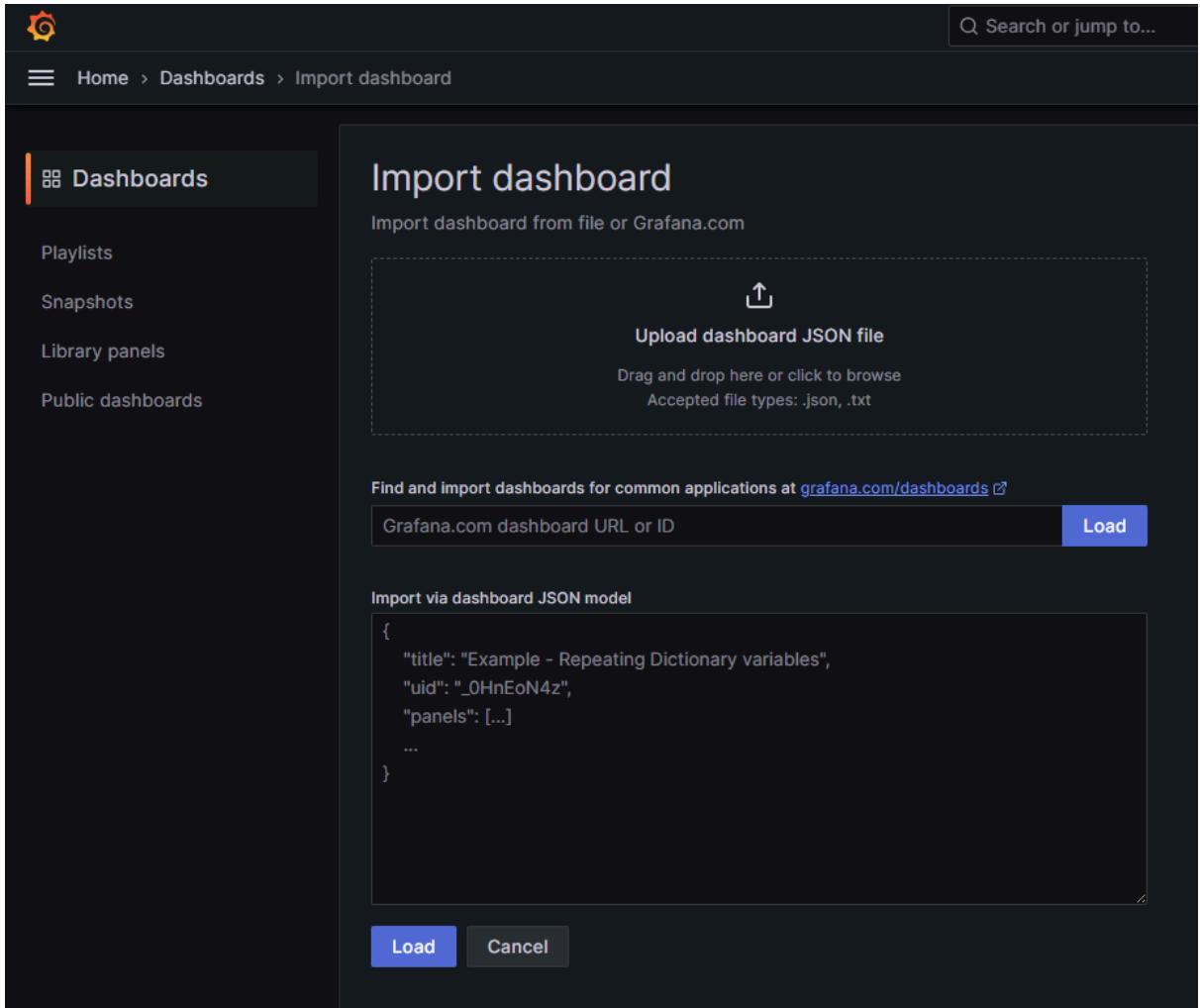
- Email

Cela permet une réaction rapide et proactive en cas d'incident, contribuant à la haute disponibilité du système.

9.6. Ajout d'un dashboard personnalisé dans Grafana

Pour une supervision plus fine, un tableau de bord personnalisé a été importé dans Grafana :

- Depuis l'interface Grafana, cliquer sur "+" > **Import**.
- Dans "**Import via dashboard JSON**", coller le contenu d'un fichier JSON de dashboard (préalablement exporté ou téléchargé depuis Grafana Labs).
- Sélectionner la source de données Prometheus.
- Cliquer sur **Import**.



The screenshot shows the 'Import dashboard' screen in Grafana. On the left, a sidebar menu includes 'Dashboards' (selected), 'Playlists', 'Snapshots', 'Library panels', and 'Public dashboards'. The main area has a title 'Import dashboard' and a sub-instruction 'Import dashboard from file or Grafana.com'. It features a dashed box for 'Upload dashboard JSON file' with an 'Upload' icon and the text 'Drag and drop here or click to browse Accepted file types: .json, .txt'. Below this is a search bar 'Grafana.com dashboard URL or ID' and a blue 'Load' button. At the bottom, there's a code editor for 'Import via dashboard JSON model' containing a JSON snippet:

```
{  
  "title": "Example - Repeating Dictionary variables",  
  "uid": "_0HnEoN4z",  
  "panels": [...]  
  ...  
}
```

At the very bottom are 'Load' and 'Cancel' buttons.

Cela permet d'obtenir un dashboard visuel complet regroupant :

- Les métriques de l'application Taskshare
- Les performances du cluster K3s
- Les statistiques de MariaDB Galera

10. Sécurisation de l'infrastructure



Accès & authentification

- SSH : Accès restreint par clé privée & root est désactivée
- Traefik TLS : Chiffrement HTTPS avec certificats

CI/CD & Base de donnée

- Secrets masqués dans Jenkins (credentials, tokens)
- MariaDB est uniquement accessible en interne au cluster .

Bonnes pratiques

- Suppression des ports inutiles
- Mise à jour régulière des images et dépendances

10.1. Gestion des accès et permissions SSH

La première étape dans la sécurisation d'une infrastructure consiste à restreindre et contrôler les accès via SSH. Voici les pratiques mises en place :

- Utilisation de clés SSH au lieu de mots de passe pour l'authentification.

Désactivation de l'accès root direct via SSH en modifiant `/etc/ssh/sshd_config` :

- PermitRootLogin no
- PasswordAuthentication no
- Gestion centralisée des accès avec Ansible pour déployer les clés publiques des utilisateurs autorisés sur l'ensemble des noeuds.

10.2. Mise en place de certificats TLS (Traefik, application)

La sécurité des communications est assurée par l'utilisation de certificats TLS. Nous avons :

- Intégré Traefik comme Ingress Controller avec support HTTPS via un certificat TLS auto-signé.
- Protégé l'interface de l'application TaskShare en forçant l'utilisation du protocole HTTPS, à l'aide d'un certificat auto-signé.

10.3. Audit de sécurité et centralisation des logs

Afin d'assurer un suivi continu de la sécurité, plusieurs mécanismes d'audit et de centralisation ont été mis en place :

- Des alertes personnalisées ont été intégrées dans Grafana en se basant sur les données issues de Prometheus.
- Cela permet de détecter en temps réel des comportements anormaux tels qu'un pic de charge CPU, des démarrages répétés de conteneurs ou l'absence de métriques. Les notifications sont envoyées automatiquement via mail assurant une réactivité immédiate face aux incidents.

10.4. Sécurisation des bases de données et du pipeline CI/CD

Bases de données (Galera MariaDB)

- Activation de l'authentification par mot de passe fort pour tous les comptes utilisateurs.
- Accès réservé au réseau interne K3s, sans exposition publique.

Pipeline CI/CD

- Stockage des **secrets** dans Jenkins via Credentials Manager (DockerHub, SonarQube, etc.)
- Utilisation de **tokens d'accès limités** pour GitHub et DockerHub (tokens personnels avec droits restreints).
- Intégration sécurisée de GitHub avec Jenkins via **webhook et ngrok**, sans exposer Jenkins publiquement de façon permanente.
- Restriction des permissions d'exécution des pipelines aux seuls développeurs autorisés.

Cette approche de sécurité par défaut, avec des accès minimaux et tracés, permet de réduire la surface d'attaque tout en assurant une exploitation sûre de l'infrastructure.

11. Tests, validation et mise en production

11.1. Tests fonctionnels et de performance

Avant toute mise en production, il est essentiel de vérifier le bon fonctionnement de l'application et de s'assurer de ses performances. Deux types de tests ont été mis en place dans le cadre du projet Taskshare :

- **Tests fonctionnels** : ils permettent de valider que toutes les fonctionnalités de l'application (création de tâches, authentification, collaboration, etc.) se comportent comme attendu.
 - Réalisés manuellement via l'interface web et mobile.

11.2. Validation complète du pipeline CI/CD

Le pipeline CI/CD a été vérifié à travers plusieurs cycles d'intégration et de déploiement :

- **Build de l'image Docker** à chaque push sur la branche **main**
- **Analyse de code** via SonarQube
- **Push sur Docker Hub**
- **Déploiement automatique en environnement de test** sur K3s
- **Déploiement manuel en production via étape d'approbation**

Des tests finaux ont été exécutés sur l'environnement de test pour valider la conformité du déploiement.

11.3. Optimisations finales et corrections

Durant les dernières itérations du projet, plusieurs optimisations ont été appliquées :

- **Optimisation du Jenkinsfile** pour améliorer les temps d'exécution
- **Amélioration des performances de la base de données Galera** optimisation des requêtes
- **Suppression des dépendances inutiles** dans le projet

11.4. Mise en production et stratégie de rollback

La mise en production de Taskshare a été effectuée après validation complète :

- Déploiement avec Helm sur le namespace **production** du cluster K3s
- Tests de bon fonctionnement réalisés en production
- Monitoring via Prometheus et Grafana immédiatement activé

Stratégie de rollback

En cas d'échec ou de bug critique en production, un plan de rollback est prévu :

Versionnage des charts Helm permet de revenir à une version stable précédemment déployée via :

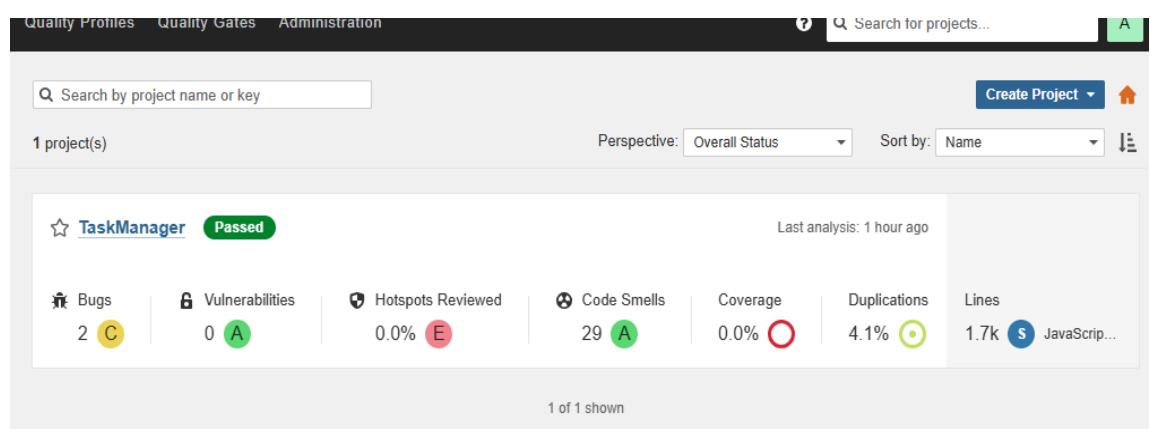
- **helm rollback taskshare <num_version> -n production**
- **Images Docker taguées** : chaque image déployée en production est associée à un tag unique , facilitant un retour à une version antérieure.

Cette stratégie garantit une mise en production sécurisée, avec la possibilité de réagir rapidement en cas d'incident.

11.5. Captures d'écran et démonstration du bon fonctionnement

Pour illustrer les résultats finaux et démontrer le bon fonctionnement global du système, plusieurs captures d'écran ont été intégrées :

- **SonarQube** :
 - Qualité du code source (coverage, bugs, duplications, ...)
 - Interface de tableau de bord projet Taskshare



- Jenkins :

- Vue d'ensemble des étapes du pipeline CI/CD
- Exécution complète (Build, Analyse, Push, Déploiement)

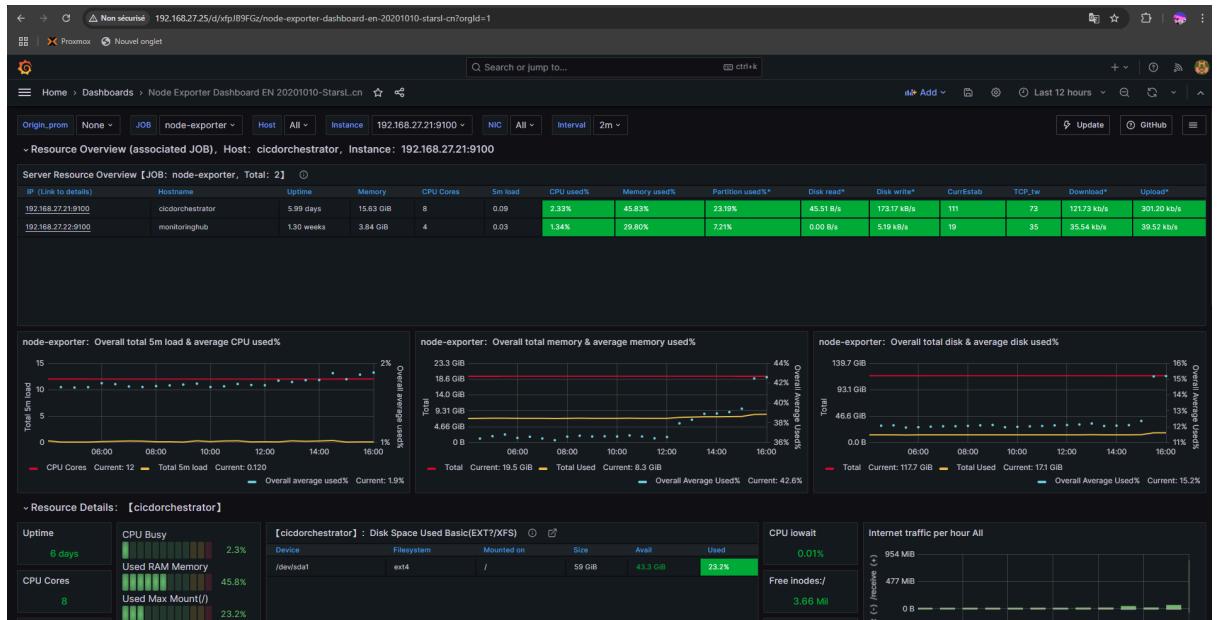
Stage View

	Declarative: Checkout SCM	Cloner le code	Tests & couverture	Analyse SonarQube	Construire l'image Docker	Pousser sur Docker Hub	Déploiement sur TEST	Approval for Production	Déploiement sur PRODUCTION	Declarative: Post Actions
Average stage times: (full run time: ~1min 33s)										
juin 13 13:59	1 commit	585ms	446ms	2s	12s	3s	12s	727ms	40ms	736ms
juin 11 15:14	1 commit	609ms	454ms	2s	13s	3s	12s	882ms (paused for 22s)	40ms	892ms

○

- Grafana :

- Monitoring de l'application (requêtes HTTP, taux d'erreur, CPU, mémoire)
- Dashboard personnalisé pour Galera MariaDB et Taskshare



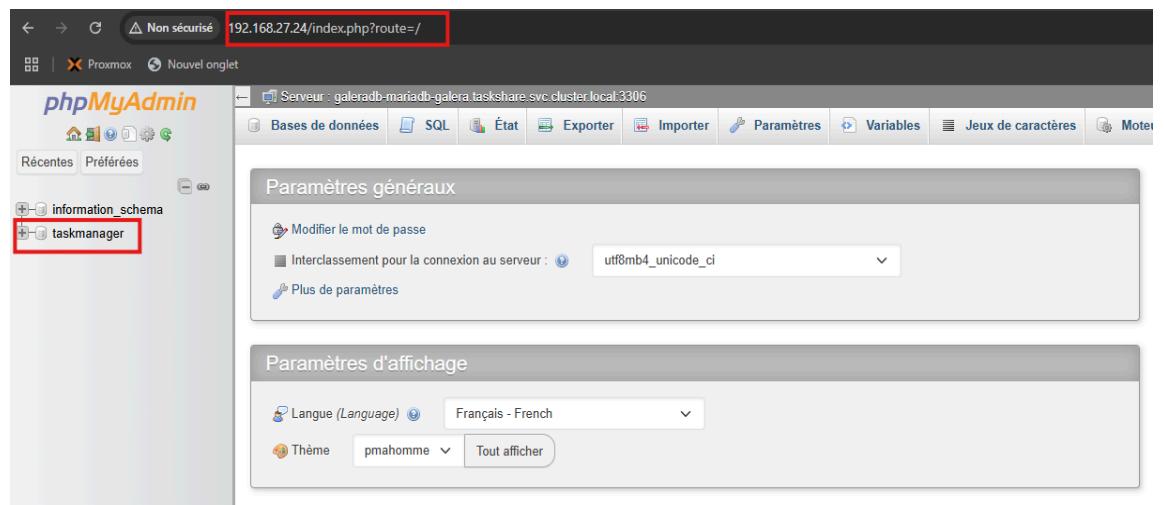
- **Application Taskshare :**

- Interface de connexion
- Interface de l'espace de travail et tâches partagées
- Affichage responsive sur desktop et mobile



- **phpMyAdmin :**

- Vue en temps réel de la base Galera MariaDB
- Statut des pods et services via la CLI K3s



Ces éléments visuels permettent de conclure sur le succès de la chaîne DevOps mise en place, garantissant un déploiement fluide, supervisé, sécurisé et fonctionnel de l'application Taskshare.

12. Bilan et perspectives

12.1. Résultats obtenus et bénéfices pour l'entreprise

Le projet Taskshare a permis de déployer une application collaborative robuste, sécurisée, et entièrement automatisée via une chaîne DevOps complète. Les bénéfices concrets sont :

- Une **infrastructure HA** basée sur K3s, MariaDB Galera, et Traefik
- Une **chaîne CI/CD automatisée avec Jenkins**, garantissant des livraisons rapides et fiables
- Une **surveillance proactive** avec Prometheus, Grafana, et alerting
- Une **qualité logicielle renforcée** grâce à SonarQube et aux tests automatisés
- Une **maîtrise des déploiements** via Helm et Docker

12.2. Défis rencontrés et solutions apportées

Au cours du projet, plusieurs défis ont été surmontés :

- **Problèmes de réseau dans Proxmox** : résolus via la configuration de bridges adaptés pour K3s et MetallLB
- **Erreurs d'analyse SonarQube** : corrigées via des ajustements de règles de qualité et exclusions
- **Problèmes d'accès à Jenkins local depuis GitHub** : solution apportée avec **ngrok** et webhooks

12.3. Perspectives d'évolution et axes d'amélioration

Plusieurs pistes d'amélioration ont été identifiées pour aller plus loin :

- Sauvegarde automatique du volume de la base de données sur un cloud public
- Ajout d'un **reverse proxy avec authentification** devant Jenkins
- Déploiement sur un **cloud public** pour élargir la scalabilité
- Intégration de **Vault** ou **Sealed Secrets** pour une gestion plus sécurisée des credentials
- Mise en place d'un **système de logs centralisés** (ex : ELK Stack ou Loki)
- **Sauvegarde automatique des volumes de données** (snapshot de Galera MariaDB via des scripts cron)

Ces perspectives ouvrent la voie à une industrialisation complète de l'infrastructure, tout en renforçant la sécurité, la fiabilité et la scalabilité de la plateforme Taskshare.

13. Annexes et références

13.1. Scripts, fichiers Terraform et playbooks Ansible

L'ensemble des scripts d'automatisation et de provisionnement ont été versionnés sur GitHub pour assurer une traçabilité complète :

- **Terraform** : création des machines virtuelles sur Proxmox, configuration réseau, intégration Cloud-Init
 - 🔗 *Repo : <https://github.com/ibrahimbakayoko/terrafom-taskshare.git>*
- **Ansible** : installation automatisée de Jenkins, Docker, SonarQube, et prérequis système
 - 🔗 *Repo : <https://github.com/ibrahimbakayoko/taskshare-ansible.git>*

13.2. Helm charts et fichiers de configuration

Tous les fichiers de déploiement sont également disponibles sur GitHub :

- **Helm Chart personnalisé de l'application Taskshare** (déploiement de l'image Docker, configuration des variables d'environnement, Ingress, etc.)
 - 🔗 *Repo : <https://github.com/ibrahimbakayoko/taskshare-backend.git>*
- **Fichiers values.yaml** utilisés pour :
 - MariaDB Galera (**galera-values.yaml**)
 - Prometheus et Grafana
 - phpMyAdmin et autres services auxiliaires
 - 🔗 *Repo : <https://github.com/ibrahimbakayoko/galeradb.git>*
- **Fichiers CI/CD et supervision** :
 - **Jenkinsfile** complet avec les étapes automatisées

- **sonar-project.properties** pour l'analyse de code
- **dockerfile.yml**

 *Repo : <https://github.com/ibrahimbakayoko/taskshare-app.git>*

13.3. Bonnes pratiques

Les **bonnes pratiques DevOps** appliquées :

- Modularité des scripts
- Séparation des environnements (dev, prod)
- Versionnage rigoureux
- Respect des standards Docker/Kubernetes

13.4. Bibliographie et ressources utilisées

Type	Ressources clés
Documentation officielle	k3s.io , helm.sh , terraform.io , grafana.com
Tutoriaux & guides	DigitalOcean Kubernetes Series, Medium DevOps articles, Bitnami Helm charts
Communauté & support	Stack Overflow, GitHub Discussions, Reddit (r/devops), Discord DevOps groups
Plateformes utilisées	GitHub, Docker Hub, Grafana Dashboards, Jenkins Plugins, Prometheus.io

Conclusion

Le projet Taskshare a permis la mise en œuvre complète d'une démarche DevOps moderne, alliant automatisation, conteneurisation, supervision et sécurisation de bout en bout. Grâce à une chaîne CI/CD robuste, une infrastructure K3s haute disponibilité, et une application entièrement monitorée et testée, l'entreprise dispose désormais d'un socle technique fiable et évolutif.

Les outils choisis (Terraform, Ansible, Docker, Helm, Jenkins, Prometheus, Grafana, SonarQube...) ont été intégrés de manière cohérente pour répondre aux exigences d'agilité, de qualité logicielle et de résilience.

Ce projet constitue un socle solide pour une industrialisation future, notamment avec des pistes comme l'hébergement cloud, la montée en charge, la gestion avancée des secrets ou l'intégration de logs centralisés.

Enfin, cette expérience a permis de mettre en pratique les grands principes du DevOps : **collaboration, automatisation, amélioration continue et responsabilisation**. Elle démontre concrètement les bénéfices d'une telle démarche pour les équipes de développement comme pour l'entreprise.

Remerciements

Je tiens à remercier chaleureusement toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce projet.

Tout d'abord, je remercie **Gilles Veronie**, mon tuteur au sein de l'entreprise Acfor, pour son accompagnement, sa disponibilité et la confiance qu'il m'a accordée tout au long de ce stage.

Je remercie également l'ensemble de l'équipe , pour leur accueil, leur esprit d'équipe et leurs précieux conseils qui m'ont permis d'évoluer dans un environnement stimulant et formateur.

Je n'oublie pas mes proches, amis et camarades de promotion, pour leur soutien moral et leur aide technique occasionnelle.

Ce stage a été une opportunité précieuse pour mettre en pratique mes compétences, découvrir des outils professionnels et adopter une véritable culture DevOps. Il marque une étape importante dans mon parcours .

