Rapport Technique - TP Final

Déploiement Automatisé d'une Application 3-Tiers sur Cluster Kubernetes

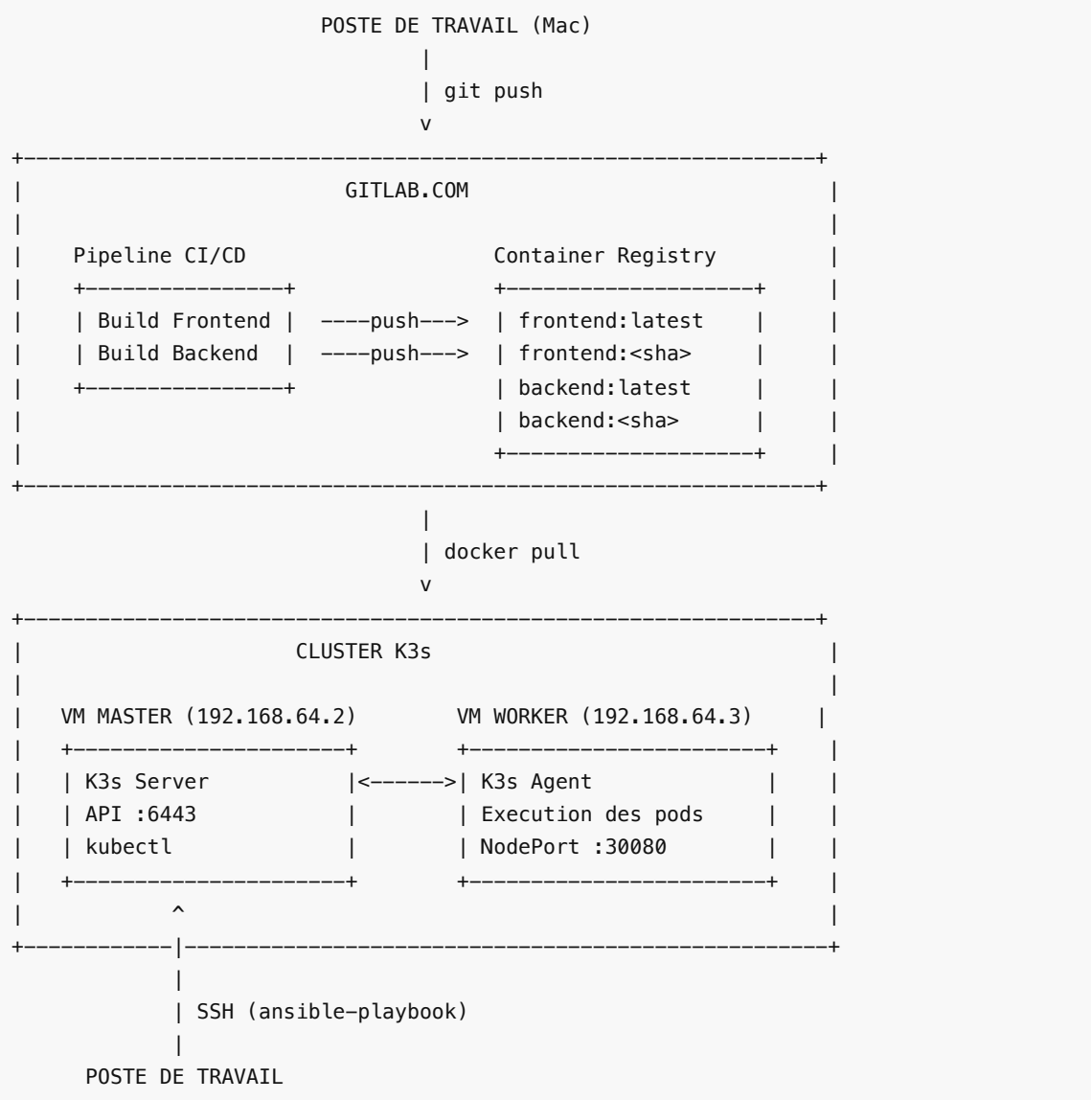
Module : INF4052 - Virtualisation et Conteneurisation
Date : Décembre 2025

---

1. Architecture et Choix Techniques

Schema d'architecture global

On a mis en place l'architecture suivante pour ce projet :



Architecture de l'application

```
FRONTEND (nginx)  --->  BACKEND (nodejs)  --->  POSTGRESQL
    :80                     :5000                  :5432
  NodePort                ClusterIP              ClusterIP
   30080
```

Nos choix techniques

Pour les images Docker :

On a pris des images Alpine partout parce que c'est beaucoup plus leger. Par exemple node:18-alpine fait 170MB au lieu de 900MB pour la version standard. Pareil pour nginx et postgres.

Pour le taggage des images :

On tag chaque image avec deux tags :

- Le SHA du commit pour pouvoir retrouver quelle version exacte tourne
- "latest" pour simplifier les deploiements de test

Pour l'orchestrateur :

On a choisi K3s parce que c'est plus leger que Kubernetes standard. Ca tourne bien sur des petites VMs avec 2GB de RAM. L'installation est rapide, une seule commande et c'est fait.

---

2. Démarche de Mise en Œuvre

Comment on a procede

Premiere etape : l'application

On a cree une appli de vote simple. Le frontend c'est une page HTML avec du JavaScript qui appelle une API. Le backend c'est du Node.js avec Express qui stocke les votes dans PostgreSQL.

On a d'abord fait tourner ca en local pour verifier que ca marchait.

Deuxieme etape : les Dockerfiles

Pour le frontend on a fait un multi-stage build. C'est peut-etre un peu exagere pour du HTML statique mais ca montre qu'on sait le faire :

```
FROM nginx:alpine AS build
COPY index.html /tmp/

FROM nginx:alpine
COPY --from=build /tmp/index.html /usr/share/nginx/html/
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Pour le backend pareil, le premier stage installe les dependances npm et le deuxieme ne garde que le necessaire.

Troisieme etape : docker-compose

On a ecrit un docker-compose.yml pour tester en local. Ca lance les 3 services et on peut tester sur localhost:8080.

Quatrieme etape : le pipeline GitLab

On a configure .gitlab-ci.yml pour builder et pusher les images automatiquement a chaque commit sur main. On utilise docker-in-docker pour pouvoir executer les commandes docker dans le runner GitLab.

Cinquieme etape : les VMs

On a cree 2 VMs Ubuntu avec UTM (on est sur Mac). On les a configurees en reseau bridge pour qu'elles puissent se parler.

Sur la premiere VM on a installe K3s en mode server. Sur la deuxieme on a installe K3s en mode agent en lui donnant l'adresse du master et le token.

Sixieme etape : les manifestes Kubernetes

On a ecrit les fichiers YAML pour deployer l'appli :

- Un namespace "vote" pour isoler nos ressources
- Un secret pour le mot de passe postgres
- Les deployments et services pour chaque composant

Le frontend a un service NodePort pour etre accessible de l'exterieur sur le port 30080. Les autres sont en ClusterIP.

Septieme etape : Ansible

On a ecrit un playbook qui se connecte au master en SSH, copie les fichiers YAML et execute kubectl apply. Comme ca on n'a pas besoin d'aller sur la VM a chaque fois.

---

3. Difficultés Rencontrées et Solutions

Probleme 1 : le backend plantait au demarrage

Ce qui se passait : Quand on lancait docker-compose, le backend crashait direct avec "connection refused" sur le port 5432.

Comment on a compris : On a regarde les logs et on a vu que le backend essayait de se connecter a postgres alors que postgres etait encore en train de demarrer. depends_on verifie juste que le conteneur est lance, pas que postgres est pret.

Ce qu'on a fait : On a ajoute un systeme de retry dans le code. Le backend essaie de se connecter 5 fois avec 3 secondes entre chaque essai. Si au bout de 5 fois ca marche pas, il s'arrete.

Probleme 2 : le pipeline rebuildait tout deux fois

Ce qui se passait : On avait mis le build dans un stage et le push dans un autre. Mais les images etaient construites deux fois.

Comment on a compris : On a realise que chaque stage repart de zero dans GitLab CI. Les images du premier stage n'existent plus dans le deuxieme.

Ce qu'on a fait : On a tout mis dans un seul stage. Build puis push direct.

Probleme 3 : erreurs YAML chelou

Ce qui se passait : kubectl refusait nos fichiers avec des messages d'erreur pas clairs genre "mapping values not allowed".

Comment on a compris : Apres avoir galere un moment on a fini par comprendre que c'etait des problemes d'indentation. En YAML faut etre precis.

Ce qu'on a fait : On a utilise un validateur YAML en ligne et on a fait plus attention. L'extension YAML de VS Code aide bien aussi.

Probleme 4 : le frontend trouvait pas le backend dans K8s

Ce qui se passait : Ca marchait en local avec docker-compose mais dans K8s le frontend recevait des erreurs quand il appelait /api.

Comment on a compris : On avait oublie de creer le Service pour le backend. Du coup le nom "backend" n'etait pas resolu dans le cluster.

Ce qu'on a fait : On a cree backend-service.yaml. Apres ca le DNS interne de K8s resolvait "backend" correctement.

---

4. Usage de l'IA Générative (Transparence)

Ce qu'on a utilise

On a utilise GitHub Copilot pour nous aider sur certaines parties du projet.

Pour quoi faire

- **Squelettes de code** : On a demande a l'IA de generer la base du server.js et des Dockerfiles. Ca evite de partir de zero.
- **Comprendre les erreurs** : Quand on avait des erreurs kubectl bizarres, on les copiait et l'IA nous expliquait ce que ca voulait dire.
- **Syntaxe YAML** : Pour les manifestes K8s, on decrivait ce qu'on voulait et l'IA generait le YAML.

Les erreurs qu'on a du corriger

L'IA fait des erreurs, on a du corriger plusieurs choses :

1. Elle generait du code trop complique. Genre le playbook Ansible faisait 80 lignes alors qu'on avait besoin que de 35.
2. Les chemins de fichiers etaient souvent faux. Elle mettait des chemins qui existaient pas chez nous.
3. Elle ajoutait des trucs inutiles comme des readinessProbe partout. C'est bien en prod mais pour un TP ca compliquait le debug.

Notre avis

C'est utile pour aller plus vite mais faut pas faire confiance aveuglement. On a du relire et corriger pas mal de choses.

---

5. Démonstration (Preuves de fonctionnement)

Test local

```
$ docker compose up -d --build
[+] Running 3/3
 ✔ Container postgres Started
 ✔ Container backend  Started
 ✔ Container frontend Started

$ curl http://localhost:8080/api/health
{"status":"OK"}

$ curl -X POST http://localhost:8080/api/vote -H 'Content-Type: application/json' -d
'{"option":"Python"}'
{"success":true}
```

[CAPTURE : Application fonctionnelle sur localhost:8080]

Pipeline CI/CD

[CAPTURE : Pipeline GitLab au vert avec le job "build" passe]

Container Registry

[CAPTURE : Registry GitLab montrant les images frontend et backend avec les tags latest et sha]

Cluster K3s

```
$ kubectl get nodes
NAME     STATUS   ROLES                  AGE   VERSION
master   Ready    control-plane,master   2d    v1.28.4+k3s1
worker   Ready    <none>                 2d    v1.28.4+k3s1
```

[CAPTURE : Resultat kubectl get nodes avec 2 noeuds Ready]

Pods deployes

```
$ kubectl get pods -n vote
NAME                        READY   STATUS    RESTARTS   AGE
postgres-xxx                1/1     Running   0          10m
backend-xxx                 1/1     Running   0          9m
frontend-xxx                1/1     Running   0          9m
```

[CAPTURE : Resultat kubectl get pods]

Application finale

[CAPTURE : Application de vote accessible sur http://192.168.64.2:30080]

Conclusion

On a reussi a deployer une application 3-tiers complete sur un cluster Kubernetes. Le plus formateur ca a ete de debugger les problemes de communication entre les services. On comprend mieux maintenant

comment fonctionne le reseau dans Docker et dans K8s.

Le projet montre toute la chaine : du code source jusqu'au deploiement automatise avec Ansible.

Depot Git : [URL a completer]