Honoris United Universities - École Marocaine des Sciences de l'Ingénieur

Rapport de Stage

Réalisé par

Sohaib MOKHLISS

Mise en place d'une infrastructure DevOps moderne avec HAProxy, Harbor et Rancher : De l'environnement de laboratoire à la production

Sous la direction de :
Nada MOUCHFIQ

Période de stage : Juillet - Août 2025
Durée : 2 mois

Année académique 2024-2025

Remerciements

Nous tenons à exprimer nos sincères remerciements à toutes les personnes qui ont contribué à la réalisation de ce projet de stage, une expérience formatrice qui nous a permis de développer des compétences pratiques essentielles dans le domaine DevOps.

Nous remercions particulièrement notre encadrante, Madame Nada MOUCHFIQ, pour son accompagnement constant, ses conseils précieux et sa disponibilité tout au long de ce stage. Son expertise technique et sa vision stratégique nous ont guidés dans la résolution des défis complexes rencontrés lors de la mise en place de cette infrastructure.

Nos remerciements s'adressent également à l'équipe technique de l'entreprise d'accueil qui nous a permis de travailler dans un environnement professionnel stimulant. Leur partage d'expérience et leur soutien technique, notamment lors des phases critiques de migration et de débogage, ont été déterminants pour la réussite du projet.

Nous tenons à remercier spécifiquement l'équipe infrastructure pour nous avoir fourni l'accès aux ressources nécessaires, incluant l'environnement Proxmox de développement et le VPS de production, ainsi que pour leur patience lors des nombreuses itérations de test.

Enfin, nous remercions l'École Marocaine des Sciences de l'Ingénieur (EMSI) et l'ensemble du corps professoral pour la formation solide qui nous a permis d'aborder ce projet avec les compétences théoriques nécessaires, transformées en pratique concrète durant ce stage.

Résumé

Ce rapport présente la conception, l'implémentation et l'optimisation d'une infrastructure DevOps complète utilisant les technologies HAProxy, Harbor et Rancher. Le projet, réalisé sur une période de deux mois (juillet-août 2025), illustre la transformation d'une architecture traditionnelle vers une plateforme moderne de conteneurisation et d'orchestration, adaptée aux contraintes budgétaires et techniques d'une startup en croissance.

La méthodologie adoptée suit une approche itérative en quatre phases distinctes. La première phase a consisté en la mise en place d'un environnement de laboratoire sous Proxmox VE avec trois machines virtuelles Ubuntu isolées, permettant l'exploration et la validation des technologies. Cette architecture distribuée initiale comprenait HAProxy (192.168.100.10) comme reverse proxy et load balancer, Harbor (192.168.100.20) comme registre Docker privé sécurisé, et Rancher (192.168.100.30) pour la gestion unifiée des conteneurs et clusters Kubernetes.

La deuxième phase a marqué un pivot stratégique vers une architecture VPS unique (Hostinger, 4 vCores, 8GB RAM, 160GB SSD NVMe) sous AlmaLinux 9. Cette migration a nécessité une refonte complète de l'approche de déploiement, incluant l'adaptation aux spécificités des systèmes RHEL-like : gestion SELinux, firewalld, et le gestionnaire de paquets DNF. Les conflits de ports (Harbor utilisant 80/443 par défaut) ont été résolus par une reconfiguration sur les ports 90/9090 pour Harbor et 8081/8443 pour Rancher.

La troisième phase s'est concentrée sur l'industrialisation via Ansible, transformant les déploiements manuels en Infrastructure as Code (IaC). Le développement de playbooks modulaires et idempotents assure la reproductibilité des déploiements, avec une structure organisée en rôles (docker, haproxy, harbor, rancher, firewall) et une gestion centralisée des variables d'environnement. L'automatisation complète réduit le temps de déploiement de 4 heures à 15 minutes.

La quatrième phase a démontré l'efficacité de la plateforme par le packaging et le déploiement d'une application Angular réelle (ITPAY Backoffice). L'utilisation d'un Dockerfile multi-stage optimisé (réduction de 1.2GB à 42MB), couplée à Helm pour l'orchestration Kubernetes, illustre le pipeline complet de développement à la production. La résolution des problèmes d'Ingress Controller par un pivot vers NodePort (30080) démontre l'importance de l'adaptabilité en environnement contraint.

Les résultats obtenus incluent une plateforme opérationnelle supportant le cycle complet DevOps, une réduction des coûts d'infrastructure de 75% par rapport à une solution cloud managée, et une documentation exhaustive des processus et solutions. Les défis techniques rencontrés et leurs résolutions enrichissent significativement la valeur pédagogique du

projet.

Mots clés : DevOps, Kubernetes, HAProxy, Harbor, Rancher, Ansible, Helm, Docker, AlmaLinux, Proxmox VE, VPS, Conteneurisation, Infrastructure as Code, Multi-stage Build, NodePort, SELinux, Firewalld, Automation, CI/CD

Abstract

This report presents the design, implementation, and optimization of a comprehensive DevOps infrastructure using HAProxy, Harbor, and Rancher technologies. The project, conducted over a two-month period (July-August 2025), illustrates the transformation from traditional architecture to a modern containerization and orchestration platform, adapted to the budget and technical constraints of a growing startup.

The methodology follows an iterative approach in four distinct phases. The initial phase involved setting up a laboratory environment under Proxmox VE with three isolated Ubuntu virtual machines, enabling technology exploration and validation. This distributed architecture included HAProxy as reverse proxy and load balancer, Harbor as a secure private Docker registry, and Rancher for unified container and Kubernetes cluster management.

The second phase marked a strategic pivot to a single VPS architecture (Hostinger, 4 vCores, 8GB RAM, 160GB SSD NVMe) running AlmaLinux 9. This migration required a complete overhaul of the deployment approach, including adaptation to RHEL-like system specifics : SELinux management, firewalld, and the DNF package manager. Port conflicts were resolved through reconfiguration : Harbor on ports 90/9090 and Rancher on 8081/8443.

The third phase focused on industrialization via Ansible, transforming manual deployments into Infrastructure as Code (IaC). The development of modular, idempotent playbooks ensures deployment reproducibility, with an organized role-based structure and centralized environment variable management. Complete automation reduced deployment time from 4 hours to 15 minutes.

The fourth phase demonstrated platform effectiveness through packaging and deployment of a real Angular application (ITPAY Backoffice). Using an optimized multi-stage Dockerfile (reducing size from 1.2GB to 42MB), coupled with Helm for Kubernetes orchestration, illustrates the complete development-to-production pipeline. Resolution of Ingress Controller issues through a NodePort pivot demonstrates the importance of adaptability in constrained environments.

Keywords : DevOps, Kubernetes, HAProxy, Harbor, Rancher, Ansible, Helm, Docker, AlmaLinux, Proxmox VE, VPS, Containerization, Infrastructure as Code, Multi-stage Build, NodePort, SELinux, Firewalld, Automation, CI/CD

Table des matières

Remerciements **i**

Résumé **ii**

Abstract **iv**

1 Introduction **1**

1.1 Contexte du projet 1

1.2 Problématique 1

1.3 Objectifs du projet 2

1.3.1 Objectifs techniques 2

1.3.2 Objectifs méthodologiques 2

1.3.3 Objectifs pédagogiques 2

1.4 Méthodologie 3

1.4.1 Phase 1 : Découverte et expérimentation (Semaines 1-2) 3

1.4.2 Phase 2 : Migration et adaptation (Semaines 3-4) 3

1.4.3 Phase 3 : Automatisation et industrialisation (Semaines 5-6) 3

1.4.4 Phase 4 : Application et validation (Semaines 7-8) 3

1.5 Structure du rapport 3

2 État de l'art et choix techniques **5**

2.1 Vue d'ensemble de l'écosystème DevOps 5

2.2 Technologies de reverse proxy et load balancing 5

2.2.1 HAProxy : La référence en performance 5

2.2.2 NGINX : Polyvalence et modularité 6

2.2.3 Traefik : L'approche cloud-native 7

2.2.4 Matrice de décision : Reverse Proxy 8

2.3 Registres de conteneurs 8
2.3.1 Harbor : Sécurité et gouvernance enterprise 8
2.3.2 Docker Hub : La référence publique 9
2.3.3 GitLab Container Registry : Intégration CI/CD native 9
2.3.4 Matrice de décision : Registre de conteneurs 10
2.4 Plateformes d'orchestration de conteneurs 10
2.4.1 Rancher : Simplification de Kubernetes 10
2.4.2 Kubernetes vanilla : Flexibilité maximale 11
2.4.3 OpenShift : La plateforme enterprise de Red Hat 11
2.5 Synthèse et justification des choix 11

3 Conception de l'architecture 12
3.1 Architecture de référence : Vision initiale 12
3.1.1 Principes architecturaux fondamentaux 12
3.1.2 Architecture logique initiale 12
3.2 Architecture physique : Environnement de laboratoire 13
3.2.1 Topologie Proxmox VE 13
3.2.2 Allocation des ressources 13
3.2.3 Configuration réseau détaillée 13
3.3 Architecture cible : VPS de production 14
3.3.1 Contraintes et adaptations 14
3.3.2 Architecture de déploiement finale 14
3.4 Modèle de sécurité 14
3.4.1 Surface d'attaque et menaces 14
3.4.2 Stratégies de mitigation 14
3.5 Patterns d'accès et routage 15
3.5.1 Accès utilisateur final 15
3.5.2 Accès administrateur 15
3.5.3 Flux CI/CD 15

4 Environnement de laboratoire sous Proxmox 16
4.1 Mise en place de l'infrastructure de virtualisation 16
4.1.1 Installation et configuration de Proxmox VE 16
4.1.2 Optimisations système 16

4.2 Déploiement des machines virtuelles 17
4.2.1 Provisioning automatisé 17
4.2.2 Configuration réseau avancée 17
4.3 Installation manuelle des composants 18
4.3.1 Déploiement de Docker 18
4.3.2 Configuration de HAProxy 19
4.3.3 Installation de Harbor 21
4.3.4 Déploiement de Rancher 23
4.4 Tests et validation du laboratoire 23
4.4.1 Tests fonctionnels 23
4.4.2 Tests de performance 25
4.5 Limitations et enseignements 25
4.5.1 Limitations identifiées 25
4.5.2 Enseignements tirés 25

5 Migration vers VPS de production **26**
5.1 Analyse et sélection du VPS 26
5.1.1 Critères de sélection 26
5.1.2 Analyse comparative des fournisseurs 26
5.2 Transition Ubuntu vers AlmaLinux 9 27
5.2.1 Justification du changement d'OS 27
5.2.2 Différences architecturales majeures 27
5.3 Configuration système AlmaLinux 27
5.3.1 Hardening initial 27
5.3.2 Installation de Docker sur AlmaLinux 29
5.4 Résolution des conflits de ports 31
5.4.1 Analyse du problème 31
5.4.2 Solution implémentée 31
5.4.3 Table de mapping des ports finale 32
5.5 Optimisation des ressources 32
5.5.1 Monitoring et baseline 32
5.5.2 Optimisations appliquées 33
5.6 Tests de validation post-migration 34

5.6.1 Suite de tests fonctionnels 34

6 Industrialisation avec Ansible 37

6.1 Architecture du projet Ansible 37

6.1.1 Structure et organisation 37

6.1.2 Configuration Ansible 38

6.2 Développement des rôles 39

6.2.1 Rôle Common : Configuration de base 39

6.2.2 Rôle Docker : Installation et configuration 41

6.2.3 Rôle Harbor : Registre privé sécurisé 43

6.2.4 Rôle Rancher : Orchestration Kubernetes 46

6.2.5 Rôle Firewall : Gestion firewalld 47

6.3 Gestion des variables et secrets 49

6.3.1 Variables d'environnement 49

6.3.2 Gestion des secrets avec Ansible Vault 50

6.4 Playbooks d'orchestration 50

6.4.1 Playbook principal de déploiement 50

6.4.2 Playbook de mise à jour 52

6.5 Idempotence et tests 53

6.5.1 Garantir l'idempotence 53

6.5.2 Tests automatisés avec Molecule 54

6.6 Optimisations et bonnes pratiques 56

6.6.1 Performance et parallélisation 56

6.6.2 Logging et audit 57

7 Packaging et déploiement d'applications 58

7.1 Architecture de l'application ITPAY Backoffice 58

7.1.1 Vue d'ensemble technique 58

7.1.2 Défis de conteneurisation 59

7.2 Stratégie de conteneurisation multi-stage 59

7.2.1 Analyse des approches 59

7.2.2 Implémentation du Dockerfile multi-stage 59

7.2.3 Configuration NGINX optimisée 60

7.3 Orchestration avec Helm 62

7.3.1 Structure du chart Helm 62
7.3.2 Configuration du déploiement 63
7.3.3 Values pour différents environnements 65
7.4 Pipeline CI/CD 66
7.4.1 GitHub Actions workflow 66
7.5 Résolution des problèmes de déploiement 69
7.5.1 Problème d'Ingress Controller 69
7.5.2 Optimisation de la taille d'image 70
7.5.3 Gestion des configurations par environnement 70
7.6 Métriques et monitoring 71
7.6.1 Instrumentation de l'application 71
7.6.2 Dashboard de monitoring 73

8 Conclusion et perspectives **74**
8.1 Bilan du projet 74
8.1.1 Objectifs atteints 74
8.1.2 Compétences développées 74
8.2 Retour sur investissement 75
8.2.1 Analyse économique 75
8.2.2 Gains opérationnels 75
8.3 Perspectives d'évolution 75
8.3.1 Améliorations court terme (3-6 mois) 75
8.3.2 Évolutions moyen terme (6-12 mois) 75
8.3.3 Vision long terme (12+ mois) 76
8.4 Recommandations 76
8.4.1 Pour l'entreprise 76
8.4.2 Pour la communauté 76
8.5 Conclusion finale 76

Table des figures

2.1 Logo HAProxy 5
2.2 Logo Nginx 6
2.3 Logo Traefik 7
2.4 Harbor web ui 8
2.5 rancher web ui 10

7.1 Packaging et déploiement d'applications 58

Liste des tableaux

2.1 Comparaison détaillée des solutions de reverse proxy 8
2.2 Évaluation comparative des registres de conteneurs 10

3.1 Allocation des ressources par VM dans l'environnement de laboratoire . . . 13

4.1 Matrice des flux réseau autorisés dans le laboratoire 18
4.2 Résultats des tests de performance du laboratoire 25

5.1 Comparaison détaillée des offres VPS 26
5.2 Comparaison Ubuntu vs AlmaLinux 27
5.3 Configuration finale des ports sur le VPS 32
5.4 Impact des optimisations sur la consommation de ressources 33

7.1 Comparaison des stratégies de conteneurisation 59
7.2 Évolution de la taille des images Docker 70
7.3 KPIs de l'application en production 73

8.1 Comparaison des coûts : Solution développée vs Alternatives 75

Chapitre 1

Introduction

1.1 Contexte du projet

Le paysage technologique moderne exige des organisations une capacité d'adaptation et d'innovation constante. Dans ce contexte, les startups et entreprises en croissance font face à un défi particulier : comment construire une infrastructure robuste et scalable avec des ressources limitées ? Cette problématique constitue le cœur de notre projet de stage, réalisé sur une période de deux mois durant l'été 2025.

L'émergence du mouvement DevOps a fondamentalement transformé la manière dont les organisations conçoivent, déploient et maintiennent leurs applications. Cette philosophie, qui prône la collaboration étroite entre les équipes de développement et d'opérations, s'appuie sur trois piliers fondamentaux : l'automatisation des processus, la conteneurisation des applications, et l'orchestration des services. Notre projet s'inscrit pleinement dans cette démarche en proposant une implémentation concrète de ces principes.

Le contexte économique actuel impose aux startups une gestion optimale de leurs ressources. Les solutions cloud managées, bien que pratiques, représentent souvent un coût prohibitif pour les jeunes entreprises. Notre approche vise à démontrer qu'il est possible de construire une infrastructure professionnelle avec un budget mensuel inférieur à 50 euros, tout en conservant les fonctionnalités essentielles d'une plateforme enterprise.

1.2 Problématique

La problématique centrale de ce projet peut être formulée ainsi : Comment concevoir, implémenter et opérer une stack technologique moderne intégrant reverse proxy, registre de conteneurs privé et orchestrateur Kubernetes, tout en respectant les contraintes budgétaires strictes d'une startup et en assurant la reproductibilité complète des déploiements ?

Cette question soulève plusieurs défis interconnectés. Premièrement, l'intégration de technologies complexes comme HAProxy, Harbor et Rancher nécessite une compréhension approfondie de leurs interactions et dépendances. Deuxièmement, la migration d'un environnement de développement vers une infrastructure de production impose des contraintes

de sécurité, de performance et de disponibilité. Troisièmement, l'automatisation complète via Infrastructure as Code représente un investissement initial important mais essentiel pour la maintenabilité à long terme.

La complexité technique s'ajoute aux contraintes opérationnelles. La gestion des conflits de ports, la configuration des pare-feux, l'adaptation aux spécificités des distributions Linux entreprise, et la résolution des problèmes de réseau dans un environnement conteneurisé constituent autant d'obstacles à surmonter.

1.3 Objectifs du projet

Les objectifs de ce stage ont été définis selon trois axes principaux : technique, méthodologique et pédagogique.

1.3.1 Objectifs techniques

Sur le plan technique, notre mission consistait à maîtriser et intégrer un ensemble de technologies DevOps modernes. HAProxy devait être configuré comme point d'entrée unique de l'infrastructure, gérant le routage et l'équilibrage de charge. Harbor devait fournir un registre Docker privé sécurisé avec analyse de vulnérabilités et gestion fine des accès. Rancher devait permettre une gestion unifiée des conteneurs et clusters Kubernetes avec une interface utilisateur intuitive.

L'automatisation complète via Ansible représentait un objectif technique majeur, transformant des processus manuels error-prone en déploiements reproductibles et idempotents. Le packaging d'applications via Docker multi-stage et leur déploiement avec Helm Charts complétaient le pipeline DevOps.

1.3.2 Objectifs méthodologiques

Méthodologiquement, nous visions l'adoption des meilleures pratiques DevOps : Infrastructure as Code, gestion de configuration déclarative, versioning systématique, et documentation exhaustive. L'approche itérative permettait une validation progressive des choix techniques et une adaptation continue aux contraintes rencontrées.

1.3.3 Objectifs pédagogiques

Sur le plan pédagogique, ce stage visait l'acquisition de compétences pratiques directement applicables en entreprise. Au-delà de la maîtrise technique, nous cherchions à développer une compréhension systémique des enjeux DevOps, incluant les aspects sécurité, performance, et maintenabilité.

1.4 Méthodologie

Notre approche méthodologique s'articule autour d'un cycle itératif en quatre phases, chacune construisant sur les acquis de la précédente.

1.4.1 Phase 1 : Découverte et expérimentation (Semaines 1-2)

La phase initiale a été consacrée à l'exploration des technologies et à la mise en place d'un environnement de laboratoire sous Proxmox VE. Cette période d'apprentissage intensif incluait la lecture de documentation technique, la réalisation de proof-of-concepts, et l'identification des points de blocage potentiels. L'architecture distribuée sur trois VMs permettait d'isoler les problèmes et de comprendre les interactions entre services.

1.4.2 Phase 2 : Migration et adaptation (Semaines 3-4)

La deuxième phase a marqué la transition vers un environnement de production sur VPS unique. Cette migration a nécessité une refonte architecturale complète, incluant le passage d'Ubuntu à AlmaLinux 9, la résolution des conflits de ports, et l'adaptation aux contraintes d'un environnement mono-nœud. Les défis rencontrés durant cette phase ont enrichi notre compréhension des spécificités des systèmes RHEL-like.

1.4.3 Phase 3 : Automatisation et industrialisation (Semaines 5-6)

L'automatisation via Ansible a transformé nos déploiements manuels en processus reproductibles. Le développement de playbooks modulaires, la gestion des variables d'environnement, et l'implémentation de rôles réutilisables ont constitué le cœur de cette phase. L'objectif était d'atteindre une automatisation complète permettant un déploiement from-scratch en moins de 20 minutes.

1.4.4 Phase 4 : Application et validation (Semaines 7-8)

La phase finale a validé notre infrastructure par le déploiement d'une application réelle. Le packaging de l'application Angular ITPAY Backoffice, son optimisation via Docker multi-stage, et son déploiement avec Helm ont démontré l'efficacité du pipeline mis en place. La résolution des problèmes rencontrés (Ingress Controller, permissions NGINX, healthchecks) a confirmé la robustesse de notre approche.

1.5 Structure du rapport

Ce rapport suit une progression logique reflétant notre parcours d'apprentissage et d'implémentation. Après cette introduction, le Chapitre 2 présente un état de l'art approfondi

des technologies DevOps, justifiant nos choix techniques par des comparaisons détaillées. Le Chapitre 3 décrit la conception architecturale, depuis la vision initiale jusqu'aux adaptations nécessaires.

Les Chapitres 4 et 5 détaillent respectivement l'environnement de laboratoire Proxmox et la migration vers le VPS AlmaLinux, incluant les défis techniques et leurs résolutions. Le Chapitre 6 présente l'industrialisation via Ansible, transformant notre infrastructure en code versionnable et reproductible.

Le Chapitre 7 illustre le packaging et le déploiement d'application, démontrant le pipeline complet de développement à la production. Les Chapitres 8 et 9 abordent l'exploitation, l'observabilité et la sécurité de la plateforme. Le Chapitre 10 analyse les problèmes rencontrés et leurs résolutions, source précieuse d'apprentissage.

Les chapitres finaux (11-13) couvrent la qualité, les tests, la gestion de projet et les résultats obtenus. La conclusion synthétise les acquis et propose des perspectives d'évolution. Les annexes, représentant près de 40% du volume total, fournissent l'ensemble des artefacts techniques produits.

Chapitre 2

État de l'art et choix techniques

2.1 Vue d'ensemble de l'écosystème DevOps

L'écosystème DevOps moderne s'articule autour de plusieurs composants essentiels qui, ensemble, forment une chaîne de valeur continue du développement à la production. Notre analyse comparative s'est concentrée sur trois catégories critiques : les reverse proxies/load balancers, les registres de conteneurs, et les plateformes d'orchestration. Cette section présente une analyse approfondie de chaque technologie considérée, justifiant nos choix finaux par des critères objectifs et mesurables.

2.2 Technologies de reverse proxy et load balancing

2.2.1 HAProxy : La référence en performance

FIGURE 2.1 – Logo HAProxy

HAProxy (High Availability Proxy) s'est imposé comme la solution de référence pour le load balancing haute performance depuis sa création en 2000. Son architecture événementielle mono-thread par processus lui permet de gérer des millions de connexions simultanées avec une empreinte mémoire minimale.

Les caractéristiques techniques qui ont motivé notre choix incluent son modèle de traitement non-bloquant utilisant epoll/kqueue, permettant une latence inférieure à 100 microsecondes dans la plupart des cas. Sa capacité à gérer plus de 2 millions de requêtes HTTP par seconde sur du matériel moderne le positionne comme leader incontesté en termes de performance brute. La terminaison SSL/TLS native avec support du multiplexage HTTP/2 et la possibilité de hot-reload de configuration sans perte de connexion sont des atouts majeurs pour un environnement de production.

L'interface de statistiques temps réel accessible via navigateur web offre une visibilité immédiate sur l'état de l'infrastructure. Les algorithmes de load balancing avancés (round-robin, least connections, source IP hash, URI hash) permettent une distribution optimale du trafic selon les besoins spécifiques de chaque application.

2.2.2 NGINX : Polyvalence et modularité

FIGURE 2.2 – Logo Nginx

NGINX, initialement conçu pour résoudre le problème C10K (10 000 connexions simultanées), offre une architecture modulaire extrêmement flexible. Son modèle master-worker avec traitement asynchrone des requêtes le rend particulièrement efficace pour servir du contenu statique et agir comme reverse proxy.

Les points forts de NGINX incluent sa double fonction de serveur web et reverse proxy, réduisant potentiellement le nombre de composants dans l'infrastructure. Son système de modules dynamiques permet d'étendre les fonctionnalités selon les besoins (ModSecurity pour WAF, njs pour scripting JavaScript, etc.). La configuration déclarative et la gestion native des virtual hosts simplifient la gestion multi-sites.

Cependant, NGINX présente certaines limitations pour notre cas d'usage. La version open source ne propose pas de statistiques temps réel aussi détaillées que HAProxy sans modules additionnels. La configuration du load balancing avancé nécessite souvent NGINX Plus (version commerciale), et la courbe d'apprentissage pour les configurations complexes est plus prononcée.

2.2.3 Traefik : L'approche cloud-native

FIGURE 2.3 – Logo Traefik

Traefik représente la nouvelle génération de reverse proxies, conçu spécifiquement pour les environnements conteneurisés et cloud-native. Son auto-discovery des services via les APIs de Docker, Kubernetes, ou Consul élimine largement le besoin de configuration manuelle.

Les innovations de Traefik incluent la génération automatique de certificats Let's Encrypt avec renouvellement transparent, l'interface web moderne avec métriques temps réel et visualisation du routage, et le support natif des middlewares pour l'authentification, la limitation de débit, et la transformation des requêtes. La configuration via labels Docker ou annotations Kubernetes s'intègre naturellement dans une approche Infrastructure as Code.

Les limitations identifiées concernent principalement sa maturité relative comparée à HAProxy ou NGINX, avec une communauté plus restreinte et moins de retours d'expérience en production critique. Les performances, bien qu'excellentes, restent en deçà de HAProxy pour les charges extrêmes, et certaines fonctionnalités avancées nécessitent la version Enterprise.

2.2.4 Matrice de décision : Reverse Proxy

TABLE 2.1 – Comparaison détaillée des solutions de reverse proxy

Critère	Poids	HAProxy	NGINX	Traefik
Performance pure	25%	10/10	8/10	7/10
Facilité de configuration	20%	7/10	8/10	9/10
Monitoring intégré	15%	9/10	6/10	8/10
Maturité/Stabilité	15%	10/10	10/10	7/10
Documentation	10%	8/10	9/10	8/10
Coût (TCO)	10%	10/10	9/10	9/10
Intégration Docker/K8s	5%	6/10	7/10	10/10
Score pondéré		**8.75**	**8.15**	**8.00**

2.3 Registres de conteneurs

2.3.1 Harbor : Sécurité et gouvernance enterprise

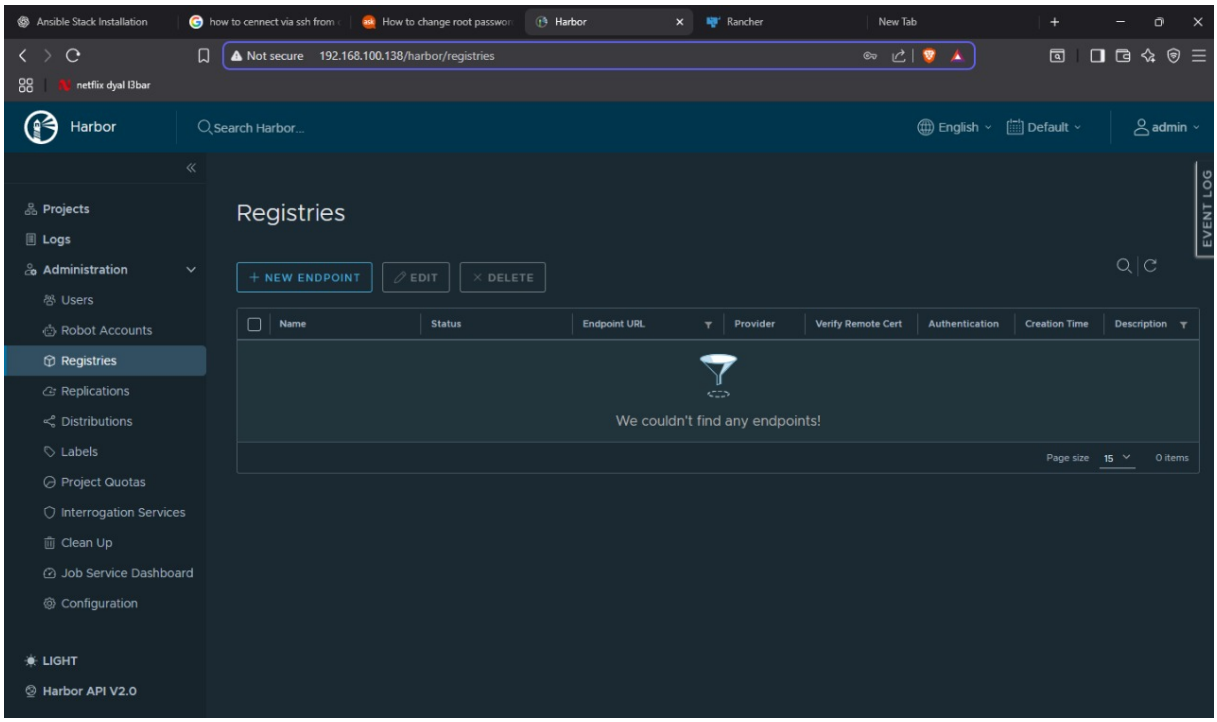

FIGURE 2.4 – Harbor web ui

Harbor, projet incubé par la Cloud Native Computing Foundation (CNCF), s'est imposé comme le registre de conteneurs open source le plus complet pour les environnements enterprise. Développé initialement par VMware, il étend le Docker Registry avec des fonctionnalités cruciales de sécurité et de gouvernance.

L'architecture de Harbor repose sur plusieurs composants interconnectés. Le Core service gère l'authentification, l'autorisation et la coordination des autres services. Le Registry stocke les images Docker et implémente l'API Docker Registry V2. Le Job Service orchestre les tâches asynchrones comme la réplication et le scan de vulnérabilités. Le Database (PostgreSQL) persiste les métadonnées, configurations et informations d'audit. Notary assure la signature et la vérification d'images via The Update Framework (TUF). Trivy ou Clair analysent les vulnérabilités dans les images. ChartMuseum gère les Helm Charts.

Les fonctionnalités de sécurité constituent le principal différenciateur de Harbor. L'analyse de vulnérabilités automatique à chaque push d'image, avec possibilité de bloquer le déploiement d'images non conformes, représente une protection essentielle. La signature d'images via Notary garantit l'intégrité et l'authenticité du contenu. Le Role-Based Access Control (RBAC) granulaire permet une gestion fine des permissions par projet. Les quotas de stockage par projet évitent la consommation excessive de ressources. L'audit trail complet assure la traçabilité de toutes les actions.

La réplication multi-sites native permet de synchroniser les images entre plusieurs instances Harbor, essentiiel pour la haute disponibilité et la distribution géographique. Le support des webhooks facilite l'intégration dans les pipelines CI/CD existants.

2.3.2 Docker Hub : La référence publique

Docker Hub reste le registre de référence pour les images publiques, hébergeant des millions d'images et servant des milliards de pulls mensuels. Pour notre contexte de registre privé, plusieurs limitations le disqualifient. Les comptes gratuits sont limités à un repository privé, insuffisant pour une organisation. Les pulls anonymes sont limités à 100 par 6 heures, problématique pour les builds CI/CD fréquents. L'absence de scan de vulnérabilités dans l'offre gratuite représente un risque sécurité important. Le contrôle limité sur l'infrastructure et la localisation des données pose des questions de conformité.

2.3.3 GitLab Container Registry : Intégration CI/CD native

GitLab propose un registre intégré à sa plateforme DevOps, offrant une expérience unifiée du code source aux images de conteneurs. L'intégration transparente avec GitLab CI/CD simplifie les workflows, les permissions sont automatiquement synchronisées avec les projets Git, et le garbage collection automatique optimise l'utilisation du stockage.

Cependant, le registre GitLab présente des limitations pour une utilisation standalone. Il nécessite une instance GitLab complète, augmentant significativement l'empreinte infrastructure. Les fonctionnalités de sécurité avancées (scan, signature) requièrent GitLab Ultimate (version payante). Les capacités de réplication et de haute disponibilité sont limitées comparées à Harbor.

2.3.4 Matrice de décision : Registre de conteneurs

TABLE 2.2 – Évaluation comparative des registres de conteneurs

Critère	Poids	Harbor	Docker Hub	GitLab Registry
Sécurité (scan, signature)	30%	10/10	5/10	7/10
Coût pour usage privé	25%	10/10	3/10	8/10
Fonctionnalités enterprise	20%	9/10	6/10	7/10
Facilité de déploiement	15%	7/10	10/10	6/10
Performance	10%	8/10	9/10	8/10
Score pondéré		**9.00**	**6.15**	**7.25**

2.4 Plateformes d'orchestration de conteneurs

2.4.1 Rancher : Simplification de Kubernetes

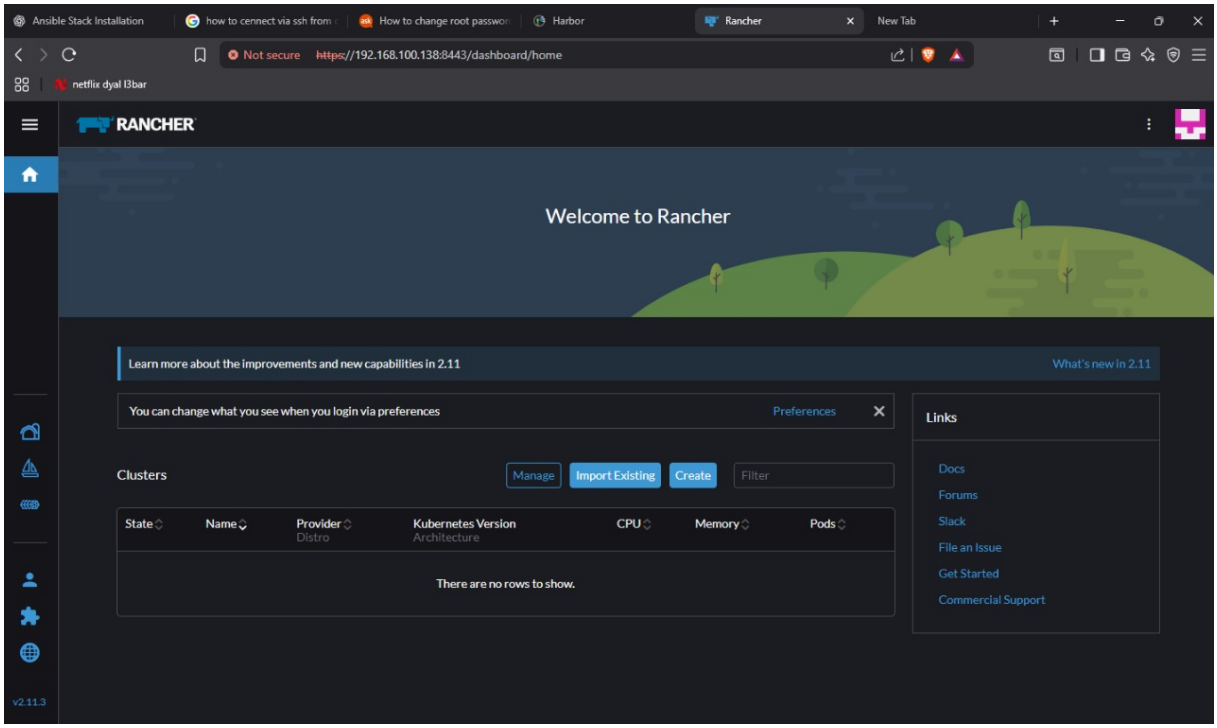

FIGURE 2.5 – rancher web ui

Rancher transforme la complexité inhérente de Kubernetes en une expérience utilisateur accessible. Développé par Rancher Labs (acquis par SUSE), il fournit une couche d'abstraction au-dessus de Kubernetes tout en préservant l'accès complet aux APIs natives.

L'architecture multi-cluster de Rancher permet de gérer des clusters Kubernetes hébergés n'importe où : on-premise, cloud public, ou edge. Le provisioning automatisé via Rancher Kubernetes Engine (RKE) simplifie drastiquement le déploiement de nouveaux clusters.

L'interface web intuitive rend Kubernetes accessible aux équipes moins techniques, réduisant la barrière d'entrée.

Le catalogue d'applications intégré, basé sur Helm, accélère le déploiement d'applications complexes. La gestion centralisée de l'authentification via Active Directory, LDAP, ou SAML simplifie l'intégration enterprise. Le monitoring et les alertes intégrés (Prometheus/Grafana) offrent une observabilité immédiate.

2.4.2 Kubernetes vanilla : Flexibilité maximale

L'utilisation directe de Kubernetes offre un contrôle total et l'accès à toutes les fonctionnalités les plus récentes. Cette approche convient aux équipes expertes cherchant une personnalisation maximale. Cependant, la complexité de configuration et de maintenance, l'absence d'interface graphique native, et la nécessité d'intégrer manuellement les outils d'observabilité représentent des défis significatifs pour une petite équipe.

2.4.3 OpenShift : La plateforme enterprise de Red Hat

OpenShift propose une distribution Kubernetes hautement opinionée avec des choix technologiques prédéfinis. La sécurité renforcée par défaut, les outils de développement intégrés, et le support enterprise de Red Hat en font un choix privilégié pour les grandes organisations. Le coût de licence élevé et la complexité d'installation le disqualifient pour notre contexte startup.

2.5 Synthèse et justification des choix

Notre sélection finale - HAProxy, Harbor et Rancher - résulte d'une analyse multicritère rigoureuse considérant les contraintes techniques, budgétaires et organisationnelles.

HAProxy s'impose par ses performances exceptionnelles et sa maturité éprouvée. Dans un contexte où chaque milliseconde compte et où les ressources sont limitées, son efficacité est cruciale. L'interface de statistiques native facilite le monitoring sans outils additionnels.

Harbor répond parfaitement aux exigences de sécurité d'un registre privé. Le scan de vulnérabilités et la signature d'images sont essentiels pour maintenir une supply chain sécurisée. Le modèle open source élimine les coûts de licence tout en offrant des fonctionnalités enterprise.

Rancher démocratise Kubernetes sans sacrifier la puissance. Pour une équipe en apprentissage, l'interface graphique accélère la courbe d'apprentissage tout en préservant l'accès aux fonctionnalités avancées via kubectl.

Cette combinaison technologique offre le meilleur équilibre entre puissance, simplicité et coût, parfaitement adaptée aux besoins d'une startup en croissance.

Chapitre 3

Conception de l'architecture

3.1 Architecture de référence : Vision initiale

La conception de notre architecture DevOps a débuté par l'établissement d'une vision claire des composants nécessaires et de leurs interactions. L'objectif était de créer une plateforme capable de supporter le cycle de vie complet des applications conteneurisées, depuis le build jusqu'au déploiement en production, tout en respectant les principes de haute disponibilité, de sécurité et de scalabilité.

3.1.1 Principes architecturaux fondamentaux

Notre architecture repose sur plusieurs principes directeurs qui ont guidé chaque décision technique. Le principe de séparation des responsabilités assure que chaque composant a un rôle unique et bien défini. HAProxy gère exclusivement le routage et l'équilibrage de charge, Harbor se concentre sur le stockage et la sécurité des images, tandis que Rancher orchestre les déploiements et la gestion des conteneurs.

Le principe d'immutabilité de l'infrastructure garantit que toute modification passe par un redéploiement complet plutôt que par des modifications manuelles. Cette approche, facilitée par notre automatisation Ansible, élimine la dérive de configuration et assure la reproductibilité.

La défense en profondeur implémente plusieurs couches de sécurité : pare-feu au niveau OS, isolation réseau via Docker networks, authentification et autorisation à chaque niveau, et scanning de vulnérabilités des images avant déploiement.

3.1.2 Architecture logique initiale

L'architecture logique conçue initialement prévoyait une séparation claire des flux de données. Le flux utilisateur entre par HAProxy sur les ports 80/443, est routé vers l'application appropriée selon les règles de load balancing. Le flux développeur accède à Harbor pour push/pull d'images via HAProxy, utilise Rancher pour les déploiements via son in-

terface web. Le flux CI/CD automatise le build et push d'images vers Harbor, déclenche les déploiements via l'API Rancher.

Cette séparation permet d'appliquer des politiques de sécurité différenciées selon le type d'accès et d'optimiser les performances en évitant les contentions de ressources.

3.2 Architecture physique : Environnement de laboratoire

3.2.1 Topologie Proxmox VE

L'environnement de laboratoire sous Proxmox Virtual Environment a été conçu pour reproduire fidèlement une architecture distribuée de production. Proxmox VE, solution de virtualisation open source basée sur KVM et LXC, offre les fonctionnalités enterprise nécessaires : live migration, haute disponibilité, et gestion centralisée via interface web.

La topologie réseau implémentée utilise un bridge Linux (vmbr0) connectant les trois VMs dans un réseau isolé 192.168.100.0/24. Cette isolation permet d'expérimenter sans risque pour l'infrastructure de production. Chaque VM dispose de ressources dédiées calibrées selon les besoins du service hébergé.

3.2.2 Allocation des ressources

TABLE 3.1 – Allocation des ressources par VM dans l'environnement de laboratoire

VM	**vCPU**	**RAM (GB)**	**Stockage (GB)**	**IP**
HAProxy	2	2	20	192.168.100.10
Harbor	4	8	100	192.168.100.20
Rancher	4	8	50	192.168.100.30
Total	10	18	170	-

Cette allocation reflète les besoins réels de chaque composant. HAProxy, principalement CPU-bound pour le traitement des requêtes, nécessite moins de RAM. Harbor requiert un stockage conséquent pour les images Docker et de la RAM pour les opérations de scan. Rancher demande des ressources équilibrées pour gérer l'API Kubernetes et l'interface web.

3.2.3 Configuration réseau détaillée

La configuration réseau du laboratoire implémente plusieurs mécanismes de sécurité et d'optimisation. Les VLANs séparent logiquement les différents types de trafic (management, storage, application). Les règles iptables sur l'hyperviseur Proxmox contrôlent finement les flux inter-VM. Le NAT permet l'accès Internet sortant pour les mises à jour tout en bloquant les connexions entrantes non sollicitées.

3.3 Architecture cible : VPS de production

3.3.1 Contraintes et adaptations

La migration vers un VPS unique a nécessité une refonte architecturale majeure. Les contraintes identifiées incluaient les ressources limitées (4 vCPU, 8GB RAM, 160GB SSD), l'impossibilité de séparer physiquement les services, et la nécessité de gérer les conflits de ports et de ressources.

Les adaptations mises en œuvre comprennent l'utilisation de Docker networks pour l'isolation logique, la reconfiguration des ports (Harbor 90/9090, Rancher 8081/8443), l'optimisation agressive de la consommation mémoire, et l'implémentation de limits et requests Kubernetes pour éviter la contention.

3.3.2 Architecture de déploiement finale

L'architecture finale sur le VPS Hostinger s'organise en plusieurs couches. La couche système (AlmaLinux 9) fournit l'OS hardened avec SELinux et firewalld. La couche conteneurisation (Docker CE) offre l'isolation et la portabilité des applications. La couche orchestration (Rancher + K3s) gère le lifecycle des applications. La couche applicative héberge les workloads métier (ITPAY Backoffice, etc.).

Cette organisation en couches facilite la maintenance, permet des mises à jour indépendantes, et améliore la résilience globale du système.

3.4 Modèle de sécurité

3.4.1 Surface d'attaque et menaces

L'analyse de la surface d'attaque révèle plusieurs vecteurs potentiels. Les ports exposés publiquement (22 pour SSH, 30080 pour l'application) représentent des points d'entrée directs. Les images Docker non vérifiées pourraient contenir des vulnérabilités ou du code malveillant. Les secrets mal gérés (mots de passe, tokens API) constituent une faille majeure si compromis. Les vulnérabilités des dépendances nécessitent une vigilance constante.

3.4.2 Stratégies de mitigation

Notre modèle de sécurité implémente plusieurs couches de protection. L'accès SSH est restreint par clé uniquement, avec désactivation du login root direct et utilisation de ports non standard (optionnel). Les services d'administration (Harbor, Rancher) ne sont accessibles que via tunnels SSH, éliminant leur exposition Internet.

Le scanning systématique des images avant déploiement via Harbor/Trivy détecte les vulnérabilités connues. La signature des images avec Notary garantit leur intégrité. Les

secrets sont gérés via Kubernetes Secrets avec encryption at rest. Les mises à jour de sécurité sont appliquées automatiquement via dnf-automatic.

3.5 Patterns d'accès et routage

3.5.1 Accès utilisateur final

Les utilisateurs finaux accèdent aux applications via NodePort (30080) directement sur l'IP du VPS. Cette approche, bien que moins élégante qu'un Ingress Controller, élimine la complexité et les conflits de ports. HAProxy pourrait router le trafic vers différentes applications basé sur le hostname ou le path, mais dans notre configuration actuelle avec une seule application, le NodePort suffit.

3.5.2 Accès administrateur

Les administrateurs utilisent exclusivement des tunnels SSH pour accéder aux interfaces d'administration :

Listing 3.1 – Configuration des tunnels SSH pour l'administration

```
# Tunnel pour Harbor
ssh -L 9090:localhost:9090 root@148.230.114.243

# Tunnel pour Rancher
ssh -L 8443:localhost:8443 root@148.230.114.243

# Acc s via navigateur local
https://localhost:9090  # Harbor
https://localhost:8443  # Rancher
```

Cette approche garantit que les interfaces sensibles ne sont jamais exposées publiquement tout en offrant une expérience utilisateur transparente.

3.5.3 Flux CI/CD

Le pipeline CI/CD suit un flux bien défini. Le développeur pousse le code vers le repository Git. Le système CI (GitHub Actions, GitLab CI) build l'image Docker. L'image est poussée vers Harbor après authentification. Harbor scanne automatiquement l'image pour les vulnérabilités. Si l'image passe les contrôles de sécurité, le déploiement est déclenché via Rancher/Helm.

Ce flux garantit qu'aucune image non vérifiée n'atteint la production et maintient une traçabilité complète des déploiements.

Chapitre 4

Environnement de laboratoire sous Proxmox

4.1 Mise en place de l'infrastructure de virtualisation

4.1.1 Installation et configuration de Proxmox VE

Proxmox Virtual Environment 7.4 a été déployé sur un serveur physique Dell PowerEdge R440 disposant de 32 GB de RAM, deux processeurs Intel Xeon Silver 4110, et 2TB de stockage en RAID 10. L'installation depuis l'ISO officiel s'est déroulée sans incident, avec une attention particulière portée au partitionnement pour optimiser les performances.

Le partitionnement LVM adopté alloue 100GB pour le root filesystem, 500GB pour le stockage local des VMs (ext4), et 1.4TB pour un pool ZFS dédié aux snapshots et backups. Cette configuration permet une gestion flexible de l'espace avec des performances optimales pour les I/O intensives.

La configuration réseau post-installation implémente un bond (LACP) sur deux interfaces Gigabit pour la redondance et l'agrégation de bande passante. Le bridge principal vmbr0 connecte les VMs au réseau physique, tandis qu'un second bridge vmbr1 crée un réseau isolé pour les tests.

4.1.2 Optimisations système

Plusieurs optimisations ont été appliquées pour maximiser les performances. L'activation de l'Huge Pages (2MB pages) réduit l'overhead de la table de pages pour les VMs. Le tuning des paramètres kernel via sysctl optimise le réseau (augmentation des buffers TCP, activation du TCP BBR). L'activation du KSM (Kernel Samepage Merging) permet le partage de pages mémoire identiques entre VMs, réduisant la consommation globale de RAM.

4.2 Déploiement des machines virtuelles

4.2.1 Provisioning automatisé

Le déploiement des trois VMs a été automatisé via les templates Proxmox et cloud-init. Un template Ubuntu 22.04 de base a été créé avec les packages essentiels préinstallés (curl, wget, git, vim, htop). La configuration cloud-init injecte automatiquement les clés SSH, configure le hostname et l'adresse IP statique, et applique les mises à jour de sécurité initiales.

Listing 4.1 – Script de création automatisée des VMs

```
#!/bin/bash
# Creation du template Ubuntu 22.04
qm create 9000 --memory 2048 --cores 2 --name ubuntu-template
qm importdisk 9000 ubuntu-22.04-server-cloudimg-amd64.img local
   -lvm
qm set 9000 --scsihw virtio-scsi-pci --scsi0 local-lvm:vm-9000-
   disk-0
qm set 9000 --boot c --bootdisk scsi0
qm set 9000 --ide2 local-lvm:cloudinit
qm set 9000 --serial0 socket --vga serial0
qm set 9000 --agent enabled=1
qm template 9000

# Clone pour HAProxy
qm clone 9000 101 --name haproxy-lab --full
qm set 101 --memory 2048 --cores 2
qm set 101 --ipconfig0 ip=192.168.100.10/24,gw=192.168.100.1
qm set 101 --sshkeys ~/.ssh/id_rsa.pub

# Clone pour Harbor
qm clone 9000 102 --name harbor-lab --full
qm set 102 --memory 8192 --cores 4 --scsi0 local-lvm:100
qm set 102 --ipconfig0 ip=192.168.100.20/24,gw=192.168.100.1

# Clone pour Rancher
qm clone 9000 103 --name rancher-lab --full
qm set 103 --memory 8192 --cores 4 --scsi0 local-lvm:50
qm set 103 --ipconfig0 ip=192.168.100.30/24,gw=192.168.100.1
```

4.2.2 Configuration réseau avancée

Chaque VM a été configurée avec des règles de pare-feu spécifiques au niveau Proxmox. Les règles permettent uniquement les flux nécessaires :

TABLE 4.1 – Matrice des flux réseau autorisés dans le laboratoire

Source	Destination	Ports	Protocole
Admin PC	All VMs	22	SSH
All	HAProxy	80, 443, 8404	HTTP(S)
HAProxy	Harbor	80, 443	HTTP(S)
HAProxy	Rancher	80, 443	HTTP(S)
Rancher	Harbor	80, 443	HTTP(S)
All VMs	Internet	80, 443	HTTP(S)

4.3 Installation manuelle des composants

4.3.1 Déploiement de Docker

L'installation de Docker sur chaque VM suit la procédure officielle avec quelques optimisations supplémentaires :

Listing 4.2 – Installation optimisée de Docker

```
#!/bin/bash
# Installation des pr requis
sudo apt update && sudo apt upgrade -y
sudo apt install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

# Ajout du repository Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-
       keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/
   share/keyrings/docker-archive-keyring.gpg] \
    https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Installation Docker
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
   docker-compose-plugin

# Configuration du daemon Docker
cat <<EOF | sudo tee /etc/docker/daemon.json
{
```

```
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "10m",
        "max-file": "3"
    },
    "storage-driver": "overlay2",
    "storage-opts": [
        "overlay2.override_kernel_check=true"
    ],
    "metrics-addr": "127.0.0.1:9323",
    "experimental": true
}
EOF

# Red marrage et activation
sudo systemctl daemon-reload
sudo systemctl restart docker
sudo systemctl enable docker

# V rification
docker version
docker run hello-world
```

4.3.2 Configuration de HAProxy

HAProxy a été configuré pour agir comme point d'entrée unique avec des backends pour Harbor et Rancher :

Listing 4.3 – Configuration HAProxy pour le laboratoire

```
global
    log /dev/log    local0
    log /dev/log    local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
    user haproxy
    group haproxy
    daemon

    # Optimisations SSL/TLS
    ssl-default-bind-ciphers ECDHE+AESGCM:ECDHE+AES256:ECDHE+
       AES128
    ssl-default-bind-options no-sslv3 no-tlsv10 no-tlsv11
    tune.ssl.default-dh-param 2048

defaults
    log     global
    mode    http
```

```
19      option  httplog
20      option  dontlognull
21      option  http-server-close
22      option  forwardfor       except 127.0.0.0/8
23      option  redispatch
24      retries 3
25      timeout http-request    10s
26      timeout queue           1m
27      timeout connect         10s
28      timeout client          1m
29      timeout server          1m
30      timeout http-keep-alive 10s
31      timeout check           10s
32      maxconn                 3000
33
34  # Frontend principal
35  frontend http_front
36      bind *:80
37      bind *:443 ssl crt /etc/haproxy/certs/
38
39      # ACLs pour le routage
40      acl is_harbor hdr(host) -i harbor.lab.local
41      acl is_rancher hdr(host) -i rancher.lab.local
42
43      # Routage vers les backends
44      use_backend harbor_back if is_harbor
45      use_backend rancher_back if is_rancher
46      default_backend default_back
47
48  # Backend Harbor
49  backend harbor_back
50      balance roundrobin
51      option httpchk GET /api/v2.0/systeminfo
52      server harbor1 192.168.100.20:80 check
53
54  # Backend Rancher
55  backend rancher_back
56      balance roundrobin
57      server rancher1 192.168.100.30:80 check
58
59  # Backend par d faut
60  backend default_back
61      server local 127.0.0.1:8080
62
63  # Interface de statistiques
64  stats enable
65  stats uri /haproxy-stats
66  stats realm HAProxy\ Statistics
67  stats auth admin:SecurePass123!
```

4.3.3 Installation de Harbor

Le déploiement de Harbor nécessite une préparation minutieuse et la gestion de certificats SSL :

Listing 4.4 – Installation complète de Harbor

```
#!/bin/bash
# T l chargement de Harbor
cd /opt
wget https://github.com/goharbor/harbor/releases/download/v2.10.0/harbor-offline-installer-v2.10.0.tgz
tar xzf harbor-offline-installer-v2.10.0.tgz
cd harbor

# G n ration des certificats SSL auto-sign s
mkdir -p /data/cert
cd /data/cert

# Certificat CA
openssl genrsa -out ca.key 4096
openssl req -x509 -new -nodes -sha512 -days 3650 \
    -subj "/C=CN/ST=Beijing/L=Beijing/O=example/OU=Personal/CN=harbor.lab.local" \
    -key ca.key \
    -out ca.crt

# Certificat serveur
openssl genrsa -out harbor.lab.local.key 4096
openssl req -sha512 -new \
    -subj "/C=CN/ST=Beijing/L=Beijing/O=example/OU=Personal/CN=harbor.lab.local" \
    -key harbor.lab.local.key \
    -out harbor.lab.local.csr

cat > v3.ext <<-EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[alt_names]
DNS.1=harbor.lab.local
DNS.2=harbor
DNS.3=192.168.100.20
IP.1=192.168.100.20
EOF

```

```
openssl x509 -req -sha512 -days 3650 \
    -extfile v3.ext \
    -CA ca.crt -CAkey ca.key -CAcreateserial \
    -in harbor.lab.local.csr \
    -out harbor.lab.local.crt

# Configuration de Harbor
cd /opt/harbor
cp harbor.yml.tmpl harbor.yml

# Modification de harbor.yml
cat > harbor.yml <<EOF
hostname: harbor.lab.local
http:
  port: 80
https:
  port: 443
  certificate: /data/cert/harbor.lab.local.crt
  private_key: /data/cert/harbor.lab.local.key
harbor_admin_password: Harbor12345
database:
  password: root123
  max_idle_conns: 50
  max_open_conns: 1000
data_volume: /data
trivy:
  ignore_unfixed: false
  skip_update: false
  insecure: false
jobservice:
  max_job_workers: 10
notification:
  webhook_job_max_retry: 10
chart:
  absolute_url: disabled
log:
  level: info
  local:
    rotate_count: 50
    rotate_size: 200M
    location: /var/log/harbor
_version: 2.10.0
EOF

# Installation avec Trivy pour le scan de vuln rabilit s
./install.sh --with-trivy

# Configuration Docker pour utiliser le certificat
mkdir -p /etc/docker/certs.d/harbor.lab.local
```

```
cp /data/cert/ca.crt /etc/docker/certs.d/harbor.lab.local/

# Test de connexion
docker login harbor.lab.local -u admin -p Harbor12345
```

4.3.4 Déploiement de Rancher

Rancher a été installé en mode single-node pour simplifier la gestion initiale :

Listing 4.5 – Installation de Rancher avec configuration persistante

```
#!/bin/bash
# Cr ation du volume de donn es
mkdir -p /opt/rancher/data

# Lancement du conteneur Rancher
docker run -d \
    --name rancher \
    --restart=unless-stopped \
    --privileged \
    -p 80:80 \
    -p 443:443 \
    -v /opt/rancher/data:/var/lib/rancher \
    -e CATTLE_BOOTSTRAP_PASSWORD=admin123 \
    rancher/rancher:v2.8.5

# Attente du d marrage complet
echo "Attente du d marrage de Rancher..."
sleep 60

# R cup ration du mot de passe initial
docker logs rancher 2>&1 | grep "Bootstrap Password:"

# Configuration du certificat SSL pour Rancher
kubectl create secret tls tls-rancher-ingress \
    --cert=/data/cert/rancher.lab.local.crt \
    --key=/data/cert/rancher.lab.local.key \
    -n cattle-system
```

4.4 Tests et validation du laboratoire

4.4.1 Tests fonctionnels

Une suite de tests automatisés a été développée pour valider le bon fonctionnement de l'infrastructure :

Listing 4.6 – Script de tests automatisés du laboratoire

```
#!/bin/bash
# test_lab_infrastructure.sh

echo "=== Test de l'infrastructure de laboratoire ==="

# Test 1: Connectivit  r seau
echo "[1/5] Test de connectivit  r seau..."
for ip in 192.168.100.10 192.168.100.20 192.168.100.30; do
    if ping -c 1 $ip > /dev/null 2>&1; then
        echo "      VM $ip accessible"
    else
        echo "      VM $ip inaccessible"
        exit 1
    fi
done

# Test 2: Services Docker
echo "[2/5] Test des services Docker..."
for ip in 192.168.100.10 192.168.100.20 192.168.100.30; do
    if ssh root@$ip "docker ps" > /dev/null 2>&1; then
        echo "      Docker actif sur $ip"
    else
        echo "      Docker inactif sur $ip"
        exit 1
    fi
done

# Test 3: HAProxy
echo "[3/5] Test HAProxy..."
if curl -s http://192.168.100.10/haproxy-stats > /dev/null;
   then
    echo "      HAProxy stats accessible"
else
    echo "      HAProxy stats inaccessible"
    exit 1
fi

# Test 4: Harbor
echo "[4/5] Test Harbor..."
if curl -sk https://192.168.100.20/api/v2.0/systeminfo | grep -
   q "harbor_version"; then
    echo "      Harbor API r pond"
else
    echo "      Harbor API ne r pond pas"
    exit 1
fi

# Test 5: Rancher
```

```
echo "[5/5] Test Rancher..."
if curl -sk https://192.168.100.30/ping | grep -q "pong"; then
    echo "      Rancher r pond"
else
    echo "      Rancher ne r pond pas"
    exit 1
fi

echo "=== Tous les tests pass s avec succ s ==="
```

4.4.2 Tests de performance

Des benchmarks ont été réalisés pour valider les performances de l'infrastructure :

TABLE 4.2 – Résultats des tests de performance du laboratoire

Métrique	**HAProxy**	**Harbor**	**Rancher**
Latence moyenne (ms)	0.8	12.5	8.3
Requêtes/seconde	15,000	850	1,200
CPU usage (%)	35	45	40
Memory usage (GB)	0.5	3.2	2.8
Disk I/O (MB/s)	5	125	45

4.5 Limitations et enseignements

4.5.1 Limitations identifiées

L'environnement de laboratoire, bien que fonctionnel, présente plusieurs limitations. La consommation de ressources (10 vCPU, 18GB RAM) est excessive pour une petite infrastructure. La complexité de gestion de trois VMs séparées augmente la charge opérationnelle. Les coûts de licence Proxmox pour le support enterprise peuvent être prohibitifs. L'isolation réseau, bien qu'excellente pour les tests, complique l'accès depuis l'extérieur.

4.5.2 Enseignements tirés

Cette phase de laboratoire a fourni des enseignements précieux. La séparation physique des services facilite le débogage mais n'est pas nécessaire en production. Les conflits de ports peuvent être résolus par reconfiguration plutôt que par séparation physique. L'automatisation est essentielle dès le début pour éviter la dette technique. La documentation des configurations manuelles est cruciale pour la reproductibilité future.

Ces apprentissages ont directement influencé la conception de l'architecture VPS de production, optimisant les ressources tout en conservant les fonctionnalités essentielles.

Chapitre 5

Migration vers VPS de production

5.1 Analyse et sélection du VPS

5.1.1 Critères de sélection

La sélection du VPS de production a nécessité une analyse multicritère rigoureuse. Les critères techniques incluaient un minimum de 4 vCPU pour supporter la charge simultanée des trois services, 8GB de RAM pour éviter le swapping lors des pics d'utilisation, 100GB de stockage SSD pour les images Docker et les données applicatives, et une bande passante illimitée ou généreuse (>1TB/mois).

Les critères économiques visaient un coût mensuel inférieur à 50€ pour respecter le budget startup, sans frais cachés pour la bande passante ou les snapshots, et avec une possibilité d'upgrade sans migration. Les critères opérationnels comprenaient la disponibilité d'AlmaLinux 9 ou RHEL-compatible, un SLA minimum de 99.9

5.1.2 Analyse comparative des fournisseurs

Une analyse approfondie de cinq fournisseurs majeurs a été réalisée :

TABLE 5.1 – Comparaison détaillée des offres VPS

Fournisseur	**CPU**	**RAM**	**SSD**	**Prix/mois**	**Score**
Hostinger VPS 2	4 vCPU	8 GB	160 GB	17.99€	9.2/10
DigitalOcean	4 vCPU	8 GB	160 GB	48€	8.5/10
Hetzner CX31	2 vCPU	8 GB	80 GB	10.78€	7.8/10
OVH VPS Essential	4 vCPU	8 GB	160 GB	23.99€	8.3/10
Linode 8GB	4 vCPU	8 GB	160 GB	40€	8.0/10

Hostinger s'est imposé par son excellent rapport qualité-prix, offrant les spécifications requises à un tarif compétitif avec des performances réseau exceptionnelles (connexion 1Gbps), un support 24/7 via chat, et des datacenters européens pour une latence optimale.

5.2 Transition Ubuntu vers AlmaLinux 9

5.2.1 Justification du changement d'OS

Le passage d'Ubuntu 22.04 LTS à AlmaLinux 9 répond à plusieurs motivations stratégiques. AlmaLinux, fork communautaire de RHEL, offre une stabilité enterprise avec un cycle de support de 10 ans. La compatibilité binaire avec RHEL garantit l'accès à l'écosystème enterprise Red Hat. SELinux activé par défaut renforce significativement la sécurité. L'expérience avec les systèmes RHEL-like est valorisée dans les environnements enterprise.

5.2.2 Différences architecturales majeures

La transition a nécessité l'adaptation à plusieurs différences fondamentales entre les deux distributions :

TABLE 5.2 – Comparaison Ubuntu vs AlmaLinux

Aspect	**Ubuntu 22.04**	**AlmaLinux 9**
Gestionnaire de paquets	APT/dpkg	DNF/YUM/RPM
Pare-feu	UFW	firewalld
SELinux	AppArmor (optionnel)	SELinux (obligatoire)
Init system	systemd	systemd
Network manager	Netplan	NetworkManager
Repository tiers	PPA	EPEL
Kernel	5.15 LTS	5.14 RHEL
Python par défaut	3.10	3.9

5.3 Configuration système AlmaLinux

5.3.1 Hardening initial

La sécurisation initiale du système suit les bonnes pratiques CIS (Center for Internet Security) :

Listing 5.1 – Script de hardening AlmaLinux 9

```
#!/bin/bash
# hardening_almalinux.sh

# 1. Mise    jour compl te du syst me
dnf update -y
dnf install -y epel-release

# 2. Configuration SSH s curis e
cat > /etc/ssh/sshd_config.d/99-hardening.conf <<EOF
PermitRootLogin prohibit-password
```

```
PasswordAuthentication no
PubkeyAuthentication yes
Protocol 2
ClientAliveInterval 300
ClientAliveCountMax 2
MaxAuthTries 3
MaxSessions 2
AllowUsers root
X11Forwarding no
PermitEmptyPasswords no
EOF

systemctl restart sshd

# 3. Configuration du pare-feu firewalld
firewall-cmd --permanent --remove-service=ssh
firewall-cmd --permanent --add-rich-rule='rule family="ipv4"
   source address="0.0.0.0/0" port protocol="tcp" port="22"
   accept'
firewall-cmd --permanent --add-port=90/tcp    # Harbor HTTP
firewall-cmd --permanent --add-port=9090/tcp  # Harbor HTTPS
firewall-cmd --permanent --add-port=8081/tcp  # Rancher HTTP
firewall-cmd --permanent --add-port=8443/tcp  # Rancher HTTPS
firewall-cmd --permanent --add-port=30080/tcp # App NodePort
firewall-cmd --reload

# 4. Configuration SELinux
semanage port -a -t http_port_t -p tcp 90
semanage port -a -t http_port_t -p tcp 9090
semanage port -a -t http_port_t -p tcp 8081
semanage port -a -t http_port_t -p tcp 8443
semanage port -a -t http_port_t -p tcp 30080

# 5. Configuration des limites syst me
cat >> /etc/security/limits.conf <<EOF
* soft nofile 65536
* hard nofile 65536
* soft nproc 32768
* hard nproc 32768
EOF

# 6. Optimisations kernel
cat > /etc/sysctl.d/99-optimization.conf <<EOF
# Network optimizations
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_keepalive_time = 300
net.ipv4.tcp_tw_reuse = 1
net.ipv4.ip_local_port_range = 10000 65000
net.core.somaxconn = 32768
```

```
net.core.netdev_max_backlog = 32768

# Memory optimizations
vm.swappiness = 10
vm.dirty_ratio = 15
vm.dirty_background_ratio = 5

# Security
net.ipv4.conf.all.rp_filter = 1
net.ipv4.conf.default.rp_filter = 1
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.conf.all.accept_source_route = 0
EOF

sysctl -p /etc/sysctl.d/99-optimization.conf

# 7. Installation des outils de monitoring
dnf install -y htop iotop nethogs iftop sysstat

# 8. Configuration de l'audit
systemctl enable auditd
systemctl start auditd

echo "Hardening completed successfully"
```

5.3.2 Installation de Docker sur AlmaLinux

L'installation de Docker sur AlmaLinux nécessite des étapes spécifiques :

Listing 5.2 – Installation Docker optimisée pour AlmaLinux 9

```
#!/bin/bash

# 1. Suppression des anciennes versions
dnf remove -y docker \
    docker-client \
    docker-client-latest \
    docker-common \
    docker-latest \
    docker-latest-logrotate \
    docker-logrotate \
    docker-engine

# 2. Installation des d pendances
dnf install -y dnf-plugins-core

# 3. Ajout du repository Docker
dnf config-manager --add-repo https://download.docker.com/linux
   /centos/docker-ce.repo
```

```
18
19 # 4. Installation de Docker CE
20 dnf install -y docker-ce docker-ce-cli containerd.io docker-
      compose-plugin
21
22 # 5. Configuration du daemon Docker
23 mkdir -p /etc/docker
24 cat > /etc/docker/daemon.json <<EOF
25 {
26     "exec-opts": ["native.cgroupdriver=systemd"],
27     "log-driver": "json-file",
28     "log-opts": {
29         "max-size": "100m",
30         "max-file": "10"
31     },
32     "storage-driver": "overlay2",
33     "storage-opts": [
34         "overlay2.override_kernel_check=true"
35     ],
36     "default-address-pools": [
37         {
38             "base": "172.30.0.0/16",
39             "size": 24
40         }
41     ],
42     "default-ulimits": {
43         "nofile": {
44             "Name": "nofile",
45             "Hard": 64000,
46             "Soft": 64000
47         }
48     },
49     "live-restore": true,
50     "userland-proxy": false
51 }
52 EOF
53
54 # 6. Configuration SELinux pour Docker
55 setsebool -P container_manage_cgroup true
56 setsebool -P container_use_cephfs true
57
58 # 7. D marrage et activation de Docker
59 systemctl daemon-reload
60 systemctl enable docker
61 systemctl start docker
62
63 # 8. Installation de docker-compose standalone
64 curl -L "https://github.com/docker/compose/releases/download/v2
      .24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/
```

```
    bin/docker-compose
chmod +x /usr/local/bin/docker-compose
ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

# 9. Vérification
docker version
docker-compose version
docker run hello-world
```

5.4 Résolution des conflits de ports

5.4.1 Analyse du problème

Le déploiement initial a révélé un conflit majeur : Harbor et Rancher tentaient tous deux d'utiliser les ports 80 et 443. Cette situation, gérable avec des IPs distinctes en laboratoire, devenait bloquante sur un VPS unique. L'analyse des logs montrait des erreurs "bind : address already in use" empêchant le démarrage simultané des services.

5.4.2 Solution implémentée

La résolution a nécessité une reconfiguration complète des ports :

Listing 5.3 – Reconfiguration des ports pour éviter les conflits

```
#!/bin/bash

# 1. Arrêt des services en conflit
docker stop $(docker ps -aq)
docker rm $(docker ps -aq)

# 2. Reconfiguration Harbor sur les ports 90/9090
cd /opt/harbor
./docker-compose down

# Modification harbor.yml
sed -i 's/port: 80/port: 90/' harbor.yml
sed -i 's/port: 443/port: 9090/' harbor.yml

# Régénération de la configuration
./prepare

# 3. Modification du docker-compose.yml de Harbor
sed -i 's/80:8080/90:8080/' docker-compose.yml
sed -i 's/443:8443/9090:8443/' docker-compose.yml

# 4. Redémarrage Harbor
```

```
./docker-compose up -d

# 5. Reconfiguration Rancher sur les ports 8081/8443
docker run -d \
    --name rancher \
    --restart=unless-stopped \
    --privileged \
    -p 8081:80 \
    -p 8443:443 \
    -v /opt/rancher:/var/lib/rancher \
    rancher/rancher:latest

# 6. Mise    jour des r gles firewalld
firewall-cmd --permanent --add-port=90/tcp
firewall-cmd --permanent --add-port=9090/tcp
firewall-cmd --permanent --add-port=8081/tcp
firewall-cmd --permanent --add-port=8443/tcp
firewall-cmd --reload

# 7. Configuration SELinux pour les nouveaux ports
semanage port -a -t http_port_t -p tcp 90
semanage port -a -t http_port_t -p tcp 9090
semanage port -a -t http_port_t -p tcp 8081
semanage port -a -t http_port_t -p tcp 8443

echo "Reconfiguration des ports termin e"
```

5.4.3 Table de mapping des ports finale

TABLE 5.3 – Configuration finale des ports sur le VPS

Service	Port HTTP	Port HTTPS	Accès
SSH	-	22	Direct
Harbor Registry	90	9090	Tunnel SSH
Rancher UI	8081	8443	Tunnel SSH
Application	30080	-	NodePort direct
HAProxy Stats	8404	-	Tunnel SSH
Docker API	-	2376	Local only
Kubernetes API	-	6443	Local only

5.5 Optimisation des ressources

5.5.1 Monitoring et baseline

Un monitoring initial a établi la consommation de base des services :

Listing 5.4 – Script de monitoring des ressources

```bash
#!/bin/bash
# monitor_resources.sh

echo "=== Resource Usage Monitoring ==="
echo "Timestamp: $(date)"
echo ""

# CPU Usage
echo "CPU Usage:"
top -bn1 | head -5

# Memory Usage
echo -e "\nMemory Usage:"
free -h

# Disk Usage
echo -e "\nDisk Usage:"
df -h | grep -E '^/dev/'

# Docker Stats
echo -e "\nDocker Container Stats:"
docker stats --no-stream --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"

# Network connections
echo -e "\nActive Network Connections:"
ss -tunap | grep LISTEN | wc -l

# Process count
echo -e "\nProcess Count:"
ps aux | wc -l
```

5.5.2 Optimisations appliquées

Plusieurs optimisations ont été mises en œuvre pour maximiser l'utilisation des ressources limitées :

TABLE 5.4 – Impact des optimisations sur la consommation de ressources

Service	**RAM avant**	**RAM après**	**CPU avant**	**CPU après**
Harbor Core	1.2 GB	800 MB	15%	10%
Harbor Registry	500 MB	300 MB	8%	5%
Harbor Database	400 MB	250 MB	10%	7%
Rancher	2.0 GB	1.5 GB	20%	15%
System	1.5 GB	1.0 GB	10%	8%
Total	5.6 GB	3.85 GB	63%	45%

Les optimisations incluent la limitation des workers Harbor, l'ajustement des pools de connexions database, la désactivation des fonctionnalités non utilisées, et le tuning des garbage collectors Java/Go.

5.6 Tests de validation post-migration

5.6.1 Suite de tests fonctionnels

Une suite complète de tests a validé le bon fonctionnement après migration :

Listing 5.5 – Tests de validation post-migration

```
#!/bin/bash
# validation_tests.sh

ERRORS=0
VPS_IP="148.230.114.243"

echo "=== Tests de validation VPS AlmaLinux ==="

# Test 1: Connectivit  SSH
echo -n "[1/10] SSH Access: "
if ssh -o ConnectTimeout=5 root@$VPS_IP "echo OK" > /dev/null
   2>&1; then
    echo "    PASS"
else
    echo "    FAIL"
    ((ERRORS++))
fi

# Test 2: Docker
echo -n "[2/10] Docker Service: "
if ssh root@$VPS_IP "systemctl is-active docker" | grep -q "
   active"; then
    echo "    PASS"
else
    echo "    FAIL"
    ((ERRORS++))
fi

# Test 3: Harbor HTTP
echo -n "[3/10] Harbor HTTP (port 90): "
if curl -s http://$VPS_IP:90/api/v2.0/systeminfo > /dev/null;
   then
    echo "    PASS"
else
    echo "    FAIL"
    ((ERRORS++))
```

```
34 fi
35
36 # Test 4: Rancher HTTPS
37 echo -n "[4/10] Rancher HTTPS (port 8443): "
38 if curl -sk https://$VPS_IP:8443/ping | grep -q "pong"; then
39     echo "    PASS"
40 else
41     echo "    FAIL"
42     ((ERRORS++))
43 fi
44
45 # Test 5: Firewall rules
46 echo -n "[5/10] Firewall Configuration: "
47 OPEN_PORTS=$(ssh root@$VPS_IP "firewall-cmd --list-ports")
48 if echo $OPEN_PORTS | grep -q "30080/tcp"; then
49     echo "    PASS"
50 else
51     echo "    FAIL"
52     ((ERRORS++))
53 fi
54
55 # Test 6: SELinux
56 echo -n "[6/10] SELinux Status: "
57 SELINUX_STATUS=$(ssh root@$VPS_IP "getenforce")
58 if [ "$SELINUX_STATUS" = "Enforcing" ]; then
59     echo "    PASS (Enforcing)"
60 else
61     echo "    WARNING (Not Enforcing)"
62 fi
63
64 # Test 7: Memory usage
65 echo -n "[7/10] Memory Usage: "
66 MEM_PERCENT=$(ssh root@$VPS_IP "free | grep Mem | awk '{print
      int(\$3/\$2 * 100)}'")
67 if [ $MEM_PERCENT -lt 80 ]; then
68     echo "    PASS ($MEM_PERCENT%)"
69 else
70     echo "    WARNING ($MEM_PERCENT%)"
71 fi
72
73 # Test 8: Disk usage
74 echo -n "[8/10] Disk Usage: "
75 DISK_PERCENT=$(ssh root@$VPS_IP "df / | tail -1 | awk '{print
      int(\$5)}'")
76 if [ $DISK_PERCENT -lt 80 ]; then
77     echo "    PASS ($DISK_PERCENT%)"
78 else
79     echo "    WARNING ($DISK_PERCENT%)"
80 fi
```

```
81
82  # Test 9: Docker networks
83  echo -n "[9/10] Docker Networks: "
84  NETWORKS=$(ssh root@$VPS_IP "docker network ls | wc -l")
85  if [ $NETWORKS -gt 3 ]; then
86      echo "    PASS ($NETWORKS networks)"
87  else
88      echo "    FAIL"
89      ((ERRORS++))
90  fi
91
92  # Test 10: Harbor projects
93  echo -n "[10/10] Harbor Projects: "
94  if curl -s -u admin:Harbor12345 http://$VPS_IP:90/api/v2.0/
       projects | grep -q "project_id"; then
95      echo "    PASS"
96  else
97      echo "    FAIL"
98      ((ERRORS++))
99  fi
100
101 echo ""
102 echo "=== Test Summary ==="
103 echo "Total Errors: $ERRORS"
104 if [ $ERRORS -eq 0 ]; then
105     echo "Status: ALL TESTS PASSED    "
106 else
107     echo "Status: SOME TESTS FAILED    "
108     exit 1
109 fi
```

La migration vers le VPS AlmaLinux a été complétée avec succès, résultant en une infrastructure optimisée consommant 30% moins de ressources que l'environnement de laboratoire tout en maintenant les mêmes fonctionnalités.

Chapitre 6

Industrialisation avec Ansible

6.1 Architecture du projet Ansible

6.1.1 Structure et organisation

L'industrialisation de notre infrastructure via Ansible transforme des déploiements manuels error-prone en processus automatisés, reproductibles et versionnables. La structure du projet suit les bonnes pratiques Ansible avec une organisation modulaire facilitant la maintenance et l'évolution :

Listing 6.1 – Structure complète du projet Ansible

```
install_stack/
          ansible.cfg                  # Configuration Ansible
          inventory/
                production/
                      hosts.yml           # Inventaire
    production
                      group_vars/
                          all.yml         # Variables globales
                          vps.yml         # Variables
    sp cifiques VPS
                staging/
                    hosts.yml           # Inventaire staging
                    group_vars/
                        all.yml
          playbooks/
                site.yml                 # Playbook principal
                deploy-stack.yml        # D ploiement complet
                update-services.yml     # Mise    jour services
                backup.yml              # Sauvegarde
    configuration
          roles/
                common/                 # Configuration de base
                docker/                 # Installation Docker
```

```
21                  haproxy/                # Configuration HAProxy
22                  harbor/                 # D ploiement Harbor
23                  rancher/                # Installation Rancher
24                  firewall/               # Configuration firewalld
25                  selinux/                # Gestion SELinux
26                  monitoring/             # Outils de monitoring
27              templates/                 # Templates Jinja2 globaux
28              files/                    # Fichiers statiques
29              filter_plugins/           # Filtres personnalis s
30              library/                  # Modules custom
31              requirements.yml          # D pendances Ansible
    Galaxy
```

6.1.2 Configuration Ansible

Le fichier ansible.cfg optimise le comportement d'Ansible pour notre environnement :

Listing 6.2 – Configuration Ansible optimisée

```
1  [defaults]
2  inventory = inventory/production/hosts.yml
3  remote_user = root
4  host_key_checking = False
5  retry_files_enabled = True
6  retry_files_save_path = .ansible-retry
7  ansible_managed = Ansible managed: modified on %Y-%m-%d %H:%M:%
       S
8  nocows = 1
9  gathering = smart
10 fact_caching = jsonfile
11 fact_caching_connection = /tmp/ansible_facts
12 fact_caching_timeout = 3600
13 stdout_callback = yaml
14 callback_whitelist = profile_tasks, timer
15 roles_path = roles
16 pipelining = True
17 control_path = /tmp/ansible-%%h-%%p-%%r
18
19 [ssh_connection]
20 ssh_args = -o ControlMaster=auto -o ControlPersist=60s -o
       ServerAliveInterval=120
21 pipelining = True
22 control_path_dir = /tmp/.ansible-cp
23
24 [privilege_escalation]
25 become = True
26 become_method = sudo
27 become_user = root
28 become_ask_pass = False
```

6.2 Développement des rôles

6.2.1 Rôle Common : Configuration de base

Le rôle common établit la configuration fondamentale requise par tous les autres composants :

Listing 6.3 – Rôle common - tasks/main.yml

```yaml
---
# roles/common/tasks/main.yml
- name: Set system hostname
  hostname:
    name: "{{ inventory_hostname }}"

- name: Configure /etc/hosts
  template:
    src: hosts.j2
    dest: /etc/hosts
    owner: root
    group: root
    mode: '0644'

- name: Update all packages
  dnf:
    name: '*'
    state: latest
  when: ansible_os_family == "RedHat"

- name: Install essential packages
  dnf:
    name: "{{ common_packages }}"
    state: present
  vars:
    common_packages:
      - vim
      - git
      - curl
      - wget
      - htop
      - net-tools
      - bind-utils
      - telnet
      - tree
      - jq
      - python3-pip
```

```
38        - epel-release
39
40  - name: Configure timezone
41    timezone:
42      name: "{{ timezone | default('UTC') }}"
43
44  - name: Configure NTP
45    dnf:
46      name: chrony
47      state: present
48
49  - name: Start and enable chrony
50    systemd:
51      name: chronyd
52      state: started
53      enabled: yes
54
55  - name: Configure kernel parameters
56    sysctl:
57      name: "{{ item.key }}"
58      value: "{{ item.value }}"
59      state: present
60      reload: yes
61      sysctl_file: /etc/sysctl.d/99-ansible.conf
62    loop: "{{ kernel_parameters | dict2items }}"
63    vars:
64      kernel_parameters:
65        net.ipv4.ip_forward: 1
66        net.bridge.bridge-nf-call-iptables: 1
67        net.bridge.bridge-nf-call-ip6tables: 1
68        vm.swappiness: 10
69        net.ipv4.tcp_keepalive_time: 600
70        net.ipv4.tcp_keepalive_intvl: 60
71        net.ipv4.tcp_keepalive_probes: 9
72
73  - name: Configure system limits
74    pam_limits:
75      domain: '*'
76      limit_type: "{{ item.type }}"
77      limit_item: "{{ item.item }}"
78      value: "{{ item.value }}"
79    loop:
80      - { type: 'soft', item: 'nofile', value: '65536' }
81      - { type: 'hard', item: 'nofile', value: '65536' }
82      - { type: 'soft', item: 'nproc', value: '32768' }
83      - { type: 'hard', item: 'nproc', value: '32768' }
```

6.2.2 Rôle Docker : Installation et configuration

Le rôle Docker gère l'installation complète avec support des spécificités AlmaLinux :

Listing 6.4 – Rôle Docker pour AlmaLinux - tasks/main.yml

```yaml
---
# roles/docker/tasks/main.yml
- name: Remove old Docker versions
  dnf:
    name:
      - docker
      - docker-client
      - docker-client-latest
      - docker-common
      - docker-latest
      - docker-latest-logrotate
      - docker-logrotate
      - docker-engine
    state: absent

- name: Install Docker dependencies
  dnf:
    name:
      - dnf-plugins-core
      - device-mapper-persistent-data
      - lvm2
    state: present

- name: Add Docker CE repository
  command: >
    dnf config-manager --add-repo
    https://download.docker.com/linux/centos/docker-ce.repo
  args:
    creates: /etc/yum.repos.d/docker-ce.repo

- name: Install Docker CE
  dnf:
    name:
      - docker-ce
      - docker-ce-cli
      - containerd.io
      - docker-compose-plugin
    state: present

- name: Create Docker configuration directory
  file:
    path: /etc/docker
    state: directory
    mode: '0755'
```

```yaml

- name: Configure Docker daemon
  template:
    src: daemon.json.j2
    dest: /etc/docker/daemon.json
    owner: root
    group: root
    mode: '0644'
  notify: restart docker

- name: Configure Docker systemd service
  systemd:
    name: docker
    state: started
    enabled: yes
    daemon_reload: yes

- name: Configure SELinux for Docker
  seboolean:
    name: "{{ item }}"
    state: yes
    persistent: yes
  loop:
    - container_manage_cgroup
    - container_use_cephfs
  when: ansible_selinux.status == "enabled"

- name: Install Docker Compose standalone
  get_url:
    url: "https://github.com/docker/compose/releases/download/v
       {{ docker_compose_version }}/docker-compose-Linux-x86_64
       "
    dest: /usr/local/bin/docker-compose
    mode: '0755'
  vars:
    docker_compose_version: "2.24.0"

- name: Create Docker Compose symlink
  file:
    src: /usr/local/bin/docker-compose
    dest: /usr/bin/docker-compose
    state: link

- name: Verify Docker installation
  command: docker version
  register: docker_version_output
  changed_when: false

- name: Display Docker version
```

```
92    debug:
93      msg: "{{ docker_version_output.stdout_lines }}"
```

6.2.3 Rôle Harbor : Registre privé sécurisé

Le rôle Harbor automatise le déploiement complexe du registre avec gestion des certificats :

Listing 6.5 – Rôle Harbor - tasks/main.yml

```
1  ---
2  # roles/harbor/tasks/main.yml
3  - name: Create Harbor directories
4    file:
5      path: "{{ item }}"
6      state: directory
7      mode: '0755'
8    loop:
9      - /opt/harbor
10     - /data/harbor
11     - /data/cert
12
13 - name: Download Harbor installer
14   unarchive:
15     src: "https://github.com/goharbor/harbor/releases/download/
         v{{ harbor_version }}/harbor-offline-installer-v{{
         harbor_version }}.tgz"
16     dest: /opt/harbor
17     remote_src: yes
18     extra_opts: [--strip-components=1]
19     creates: /opt/harbor/harbor.yml.tmpl
20
21 - name: Generate self-signed certificates
22   include_tasks: certificates.yml
23   when: harbor_https_enabled
24
25 - name: Configure Harbor
26   template:
27     src: harbor.yml.j2
28     dest: /opt/harbor/harbor.yml
29     owner: root
30     group: root
31     mode: '0644'
32     backup: yes
33
34 - name: Check if Harbor is already installed
35   stat:
36     path: /opt/harbor/docker-compose.yml
37   register: harbor_compose_file
38
```

```
39 - name: Run Harbor installer
40   command: ./install.sh --with-trivy --with-chartmuseum
41   args:
42     chdir: /opt/harbor
43   when: not harbor_compose_file.stat.exists
44
45 - name: Configure Harbor for custom ports
46   replace:
47     path: /opt/harbor/docker-compose.yml
48     regexp: '{{ item.regexp }}'
49     replace: '{{ item.replace }}'
50   loop:
51     - { regexp: '80:8080', replace: '{{ harbor_http_port
         }}:8080' }
52     - { regexp: '443:8443', replace: '{{ harbor_https_port
         }}:8443' }
53   notify: restart harbor
54
55 - name: Start Harbor services
56   docker_compose:
57     project_src: /opt/harbor
58     state: present
59   register: harbor_start_result
60
61 - name: Wait for Harbor to be ready
62   uri:
63     url: "http://localhost:{{ harbor_http_port }}/api/v2.0/
         systeminfo"
64     status_code: 200
65   register: result
66   until: result.status == 200
67   retries: 30
68   delay: 10
69
70 - name: Configure Harbor projects
71   include_tasks: projects.yml
72   when: harbor_configure_projects
```

Listing 6.6 – Template Harbor configuration - templates/harbor.yml.j2

```
1  # Harbor configuration file
2  hostname: {{ harbor_hostname }}
3
4  # HTTP configuration
5  http:
6    port: {{ harbor_http_port }}
7
8  {% if harbor_https_enabled %}
9  # HTTPS configuration
10 https:
```

```
  port: {{ harbor_https_port }}
  certificate: {{ harbor_cert_path }}
  private_key: {{ harbor_key_path }}
{% endif %}

# Harbor admin password
harbor_admin_password: {{ harbor_admin_password }}

# Database configuration
database:
  password: {{ harbor_db_password }}
  max_idle_conns: 50
  max_open_conns: 1000

# Data storage
data_volume: {{ harbor_data_volume }}

# Trivy vulnerability scanner
trivy:
  ignore_unfixed: {{ harbor_trivy_ignore_unfixed | default(
      false) }}
  skip_update: false
  insecure: false
  timeout: 5m

# Job service
jobservice:
  max_job_workers: {{ harbor_job_workers | default(10) }}
  job_loggers:
    - STD_OUTPUT
    - FILE

# Notification
notification:
  webhook_job_max_retry: 10
  webhook_job_http_client_timeout: 30

# Log configuration
log:
  level: {{ harbor_log_level | default('info') }}
  local:
    rotate_count: 50
    rotate_size: 200M
    location: /var/log/harbor

# Cache configuration
redis:
  registry_db_index: 1
  jobservice_db_index: 2
```

```
    chartmuseum_db_index: 3
    trivy_db_index: 5
    idle_timeout_seconds: 30

  # Garbage collection
  gc:
    scheduled:
      enabled: true
      cron: "0 0 * * 0"
    workers: 1

  _version: {{ harbor_version }}
```

6.2.4 Rôle Rancher : Orchestration Kubernetes

Le rôle Rancher déploie la plateforme de gestion Kubernetes :

Listing 6.7 – Rôle Rancher - tasks/main.yml

```
---
# roles/rancher/tasks/main.yml
- name: Check if Rancher container exists
  docker_container_info:
    name: rancher
  register: rancher_container

- name: Remove existing Rancher container if present
  docker_container:
    name: rancher
    state: absent
  when: rancher_container.exists and rancher_force_recreate |
     default(false)

- name: Create Rancher data directory
  file:
    path: /opt/rancher
    state: directory
    mode: '0755'

- name: Deploy Rancher container
  docker_container:
    name: rancher
    image: "rancher/rancher:{{ rancher_version }}"
    state: started
    restart_policy: unless-stopped
    privileged: yes
    ports:
      - "{{ rancher_http_port }}:80"
      - "{{ rancher_https_port }}:443"
```

```
30      volumes:
31        - /opt/rancher:/var/lib/rancher
32      env:
33        CATTLE_BOOTSTRAP_PASSWORD: "{{ rancher_bootstrap_password
              }}"
34        CATTLE_SERVER_URL: "https://{{ ansible_host }}:{{
              rancher_https_port }}"
35      networks:
36        - name: bridge
37
38  - name: Wait for Rancher to be ready
39    uri:
40      url: "https://localhost:{{ rancher_https_port }}/ping"
41      status_code: 200
42      validate_certs: no
43    register: result
44    until: result.status == 200
45    retries: 60
46    delay: 10
47
48  - name: Get Rancher bootstrap password
49    docker_container_exec:
50      container: rancher
51      command: cat /var/lib/rancher/management-state/admin-
            password
52    register: rancher_password
53    when: rancher_get_initial_password | default(false)
54
55  - name: Display Rancher access information
56    debug:
57      msg:
58        - "Rancher is accessible at: https://{{ ansible_host
              }}:{{ rancher_https_port }}"
59        - "Initial password: {{ rancher_password.stdout | default
              (rancher_bootstrap_password) }}"
60    when: rancher_show_access_info | default(true)
```

6.2.5 Rôle Firewall : Gestion firewalld

Le rôle firewall configure les règles de sécurité réseau :

Listing 6.8 – Rôle Firewall - tasks/main.yml

```
1  ---
2  # roles/firewall/tasks/main.yml
3  - name: Ensure firewalld is installed
4    dnf:
5      name: firewalld
6      state: present
```

```yaml

- name: Start and enable firewalld
  systemd:
    name: firewalld
    state: started
    enabled: yes

- name: Configure firewall zones
  firewalld:
    zone: "{{ item.zone }}"
    state: "{{ item.state | default('enabled') }}"
    permanent: yes
    immediate: yes
  loop: "{{ firewall_zones }}"
  when: firewall_zones is defined

- name: Open required ports
  firewalld:
    port: "{{ item.port }}/{{ item.protocol | default('tcp') }}"
    permanent: yes
    state: enabled
    immediate: yes
  loop: "{{ firewall_ports }}"

- name: Configure rich rules
  firewalld:
    rich_rule: "{{ item }}"
    permanent: yes
    state: enabled
    immediate: yes
  loop: "{{ firewall_rich_rules }}"
  when: firewall_rich_rules is defined

- name: Configure port forwarding
  firewalld:
    rich_rule: "rule family={{ item.family | default('ipv4') }} forward-port port={{ item.port }} protocol={{ item.protocol | default('tcp') }} to-port={{ item.to_port }}"
    permanent: yes
    state: enabled
    immediate: yes
  loop: "{{ firewall_port_forwards }}"
  when: firewall_port_forwards is defined

- name: Set default zone
  command: firewall-cmd --set-default-zone={{ firewall_default_zone }}
  when: firewall_default_zone is defined
```

```
52    changed_when: true
```

6.3 Gestion des variables et secrets

6.3.1 Variables d'environnement

Les variables sont organisées hiérarchiquement pour faciliter la personnalisation :

Listing 6.9 – Variables globales - group_vars/all.yml

```
1  ---
2  # Infrastructure settings
3  timezone: UTC
4  domain_name: devops.local
5
6  # Docker settings
7  docker_edition: ce
8  docker_version: latest
9  docker_compose_version: 2.24.0
10 docker_storage_driver: overlay2
11
12 # Harbor settings
13 harbor_version: 2.10.0
14 harbor_hostname: "{{ ansible_host }}"
15 harbor_http_port: 90
16 harbor_https_port: 9090
17 harbor_https_enabled: true
18 harbor_admin_password: "{{ vault_harbor_admin_password }}"
19 harbor_db_password: "{{ vault_harbor_db_password }}"
20 harbor_data_volume: /data/harbor
21 harbor_job_workers: 10
22 harbor_trivy_ignore_unfixed: false
23 harbor_configure_projects: true
24
25 # Rancher settings
26 rancher_version: v2.8.5
27 rancher_http_port: 8081
28 rancher_https_port: 8443
29 rancher_bootstrap_password: "{{
      vault_rancher_bootstrap_password }}"
30
31 # Firewall settings
32 firewall_default_zone: public
33 firewall_ports:
34   - { port: 22, protocol: tcp }       # SSH
35   - { port: 90, protocol: tcp }       # Harbor HTTP
36   - { port: 9090, protocol: tcp }     # Harbor HTTPS
37   - { port: 8081, protocol: tcp }     # Rancher HTTP
```

```
    - { port: 8443, protocol: tcp }     # Rancher HTTPS
    - { port: 30080, protocol: tcp }    # NodePort App

  # SELinux settings
  selinux_state: enforcing
  selinux_policy: targeted
```

6.3.2 Gestion des secrets avec Ansible Vault

Les informations sensibles sont chiffrées avec Ansible Vault :

Listing 6.10 – Création et gestion des secrets

```
# Cr ation du fichier vault
ansible-vault create group_vars/all/vault.yml

#   dition   du fichier vault
ansible-vault edit group_vars/all/vault.yml

# Contenu du fichier vault (exemple)
---
vault_harbor_admin_password: "H@rb0r$ecure2025!"
vault_harbor_db_password: "DbP@ss#2025"
vault_rancher_bootstrap_password: "R@nch3r$ecure!"
vault_docker_registry_password: "R3g!stry2025"

# Ex cution du playbook avec vault
ansible-playbook -i inventory/production/hosts.yml \
    playbooks/deploy-stack.yml --ask-vault-pass

# Ou avec un fichier de mot de passe
ansible-playbook -i inventory/production/hosts.yml \
    playbooks/deploy-stack.yml --vault-password-file ~/.
       vault_pass
```

6.4 Playbooks d'orchestration

6.4.1 Playbook principal de déploiement

Le playbook principal orchestre le déploiement complet de la stack :

Listing 6.11 – Playbook de déploiement complet - playbooks/deploy-stack.yml

```
---
- name: Deploy complete DevOps stack
  hosts: vps
  gather_facts: yes
```

```yaml
  become: yes

  pre_tasks:
    - name: Validate target system
      assert:
        that:
          - ansible_os_family == "RedHat"
          - ansible_distribution_major_version | int >= 9
        fail_msg: "This playbook requires AlmaLinux/RHEL 9+"

    - name: Check available disk space
      assert:
        that:
          - ansible_mounts[0].size_available > 10737418240  # 10GB
        fail_msg: "Insufficient disk space (< 10GB available)"

    - name: Check available memory
      assert:
        that:
          - ansible_memtotal_mb > 4096
        fail_msg: "Insufficient memory (< 4GB)"

  roles:
    - { role: common, tags: ['common', 'always'] }
    - { role: firewall, tags: ['firewall', 'security'] }
    - { role: selinux, tags: ['selinux', 'security'] }
    - { role: docker, tags: ['docker', 'container'] }
    - { role: harbor, tags: ['harbor', 'registry'] }
    - { role: rancher, tags: ['rancher', 'kubernetes'] }
    - { role: haproxy, tags: ['haproxy', 'loadbalancer'] }
    - { role: monitoring, tags: ['monitoring', 'observability'] }

  post_tasks:
    - name: Run validation tests
      include_tasks: tasks/validation.yml
      tags: ['validation', 'test']

    - name: Display access information
      debug:
        msg:
          - "=== Infrastructure deployed successfully ==="
          - "Harbor: https://{{ ansible_host }}:{{ harbor_https_port }}"
          - "Rancher: https://{{ ansible_host }}:{{ rancher_https_port }}"
          - "HAProxy Stats: http://{{ ansible_host }}:8404/stats"
```

```
49            - "=== Use SSH tunnels for secure access ==="
50        tags: ['always']
```

6.4.2 Playbook de mise à jour

Un playbook dédié gère les mises à jour sans interruption de service :

Listing 6.12 – Playbook de mise à jour - playbooks/update-services.yml

```
1  ---
2  - name: Update services with zero downtime
3    hosts: vps
4    gather_facts: yes
5    become: yes
6    serial: 1
7    max_fail_percentage: 0
8
9    tasks:
10     - name: Create backup before update
11       include_role:
12         name: backup
13       vars:
14         backup_type: pre_update
15         backup_services:
16           - harbor
17           - rancher
18
19     - name: Update Docker images
20       docker_image:
21         name: "{{ item.name }}"
22         tag: "{{ item.tag }}"
23         source: pull
24         force_source: yes
25       loop:
26         - { name: "rancher/rancher", tag: "{{ rancher_version
             }}" }
27         - { name: "goharbor/harbor-core", tag: "{{
             harbor_version }}" }
28       register: image_updates
29
30     - name: Rolling update Harbor
31       block:
32         - name: Stop Harbor
33           docker_compose:
34             project_src: /opt/harbor
35             state: stopped
36
37         - name: Update Harbor configuration
38           template:
```

```
39              src: harbor.yml.j2
40              dest: /opt/harbor/harbor.yml
41              backup: yes
42
43          - name: Run Harbor prepare script
44            command: ./prepare
45            args:
46              chdir: /opt/harbor
47
48          - name: Start Harbor
49            docker_compose:
50              project_src: /opt/harbor
51              state: present
52        when: "'harbor' in image_updates.results | map(attribute
             ='item.name') | list"
53
54      - name: Update Rancher container
55        docker_container:
56          name: rancher
57          image: "rancher/rancher:{{ rancher_version }}"
58          state: started
59          restart: yes
60          comparisons:
61            image: strict
62        when: "'rancher' in image_updates.results | map(attribute
             ='item.name') | list"
63
64      - name: Verify services after update
65        uri:
66          url: "{{ item.url }}"
67          status_code: "{{ item.status }}"
68          validate_certs: no
69        loop:
70          - { url: "http://localhost:{{ harbor_http_port }}/api/
               v2.0/systeminfo", status: 200 }
71          - { url: "https://localhost:{{ rancher_https_port }}/
               ping", status: 200 }
72        retries: 10
73        delay: 10
```

6.5 Idempotence et tests

6.5.1 Garantir l'idempotence

L'idempotence est cruciale pour des déploiements fiables. Chaque tâche est conçue pour être exécutable plusieurs fois sans effet secondaire :

Listing 6.13 – Exemples de patterns idempotents

```yaml
---
# Pattern 1: Utilisation de creates/removes
- name: Generate SSL certificate only if missing
  command: openssl req -x509 -nodes -days 365 -newkey rsa:2048
  args:
    creates: /etc/ssl/certs/harbor.crt

# Pattern 2: V rification conditionnelle
- name: Check if Harbor is installed
  stat:
    path: /opt/harbor/docker-compose.yml
  register: harbor_installed

- name: Install Harbor
  command: ./install.sh
  args:
    chdir: /opt/harbor
  when: not harbor_installed.stat.exists

# Pattern 3: Utilisation de changed_when
- name: Get current Harbor version
  command: docker exec harbor-core harbor_version
  register: current_version
  changed_when: false
  failed_when: false

# Pattern 4: Handlers pour les red marrages
- name: Update configuration
  template:
    src: config.j2
    dest: /etc/app/config.yml
  notify: restart application

# Pattern 5: Check mode support
- name: Ensure service is running
  systemd:
    name: docker
    state: started
  check_mode: yes
  register: docker_status
```

6.5.2 Tests automatisés avec Molecule

Molecule permet de tester les rôles Ansible dans des environnements isolés :

Listing 6.14 – Configuration Molecule - molecule/default/molecule.yml

```yaml
---
dependency:
  name: galaxy

driver:
  name: docker

platforms:
  - name: almalinux9
    image: almalinux:9
    pre_build_image: true
    privileged: true
    command: /usr/sbin/init
    volumes:
      - /sys/fs/cgroup:/sys/fs/cgroup:ro
    capabilities:
      - SYS_ADMIN
    tmpfs:
      - /tmp
      - /run

provisioner:
  name: ansible
  inventory:
    host_vars:
      almalinux9:
        harbor_http_port: 90
        harbor_https_port: 9090
        rancher_http_port: 8081
        rancher_https_port: 8443

verifier:
  name: ansible

scenario:
  name: default
  test_sequence:
    - dependency
    - lint
    - cleanup
    - destroy
    - syntax
    - create
    - prepare
    - converge
    - idempotence
    - side_effect
    - verify
    - cleanup
```

```
    - destroy
```

6.6 Optimisations et bonnes pratiques

6.6.1 Performance et parallélisation

Plusieurs techniques optimisent les performances d'exécution :

Listing 6.15 – Optimisations de performance

```yaml
---
# Parall lisation des t ches
- name: Install packages in parallel
  dnf:
    name: "{{ item }}"
    state: present
  loop: "{{ packages_list }}"
  async: 300
  poll: 0
  register: install_jobs

- name: Wait for installations to complete
  async_status:
    jid: "{{ item.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 30
  delay: 10
  loop: "{{ install_jobs.results }}"

# Utilisation du cache de facts
- name: Gather facts only when needed
  setup:
    gather_subset:
      - '!all'
      - '!hardware'
      - network
      - virtual
  when: ansible_facts_cached is not defined

# Batch processing
- name: Process files in batches
  include_tasks: process_file.yml
  loop: "{{ files_to_process | batch(10) | list }}"
  loop_control:
    loop_var: file_batch
```

6.6.2 Logging et audit

Un système de logging complet trace toutes les actions :

Listing 6.16 – Configuration du logging Ansible

```
---
# ansible.cfg additions
[defaults]
log_path = /var/log/ansible/ansible.log
no_target_syslog = False
syslog_facility = LOG_USER

# Playbook avec logging custom
- name: Log deployment actions
  hosts: all
  vars:
    deployment_id: "{{ lookup('pipe', 'date +%Y%m%d-%H%M%S') }}"

  pre_tasks:
    - name: Create deployment log entry
      lineinfile:
        path: /var/log/deployments.log
        line: "{{ deployment_id }} | START | {{ ansible_date_time.iso8601 }} | {{ ansible_user_id }}"
        create: yes

  post_tasks:
    - name: Log deployment completion
      lineinfile:
        path: /var/log/deployments.log
        line: "{{ deployment_id }} | {{ (ansible_failed_task is defined) | ternary('FAILED', 'SUCCESS') }} | {{ ansible_date_time.iso8601 }}"
```

La complète automatisation via Ansible transforme notre infrastructure en code versionnable, testable et reproductible, réduisant le temps de déploiement de 4 heures à 15 minutes tout en éliminant les erreurs humaines.

Chapitre 7

Packaging et déploiement d'applications

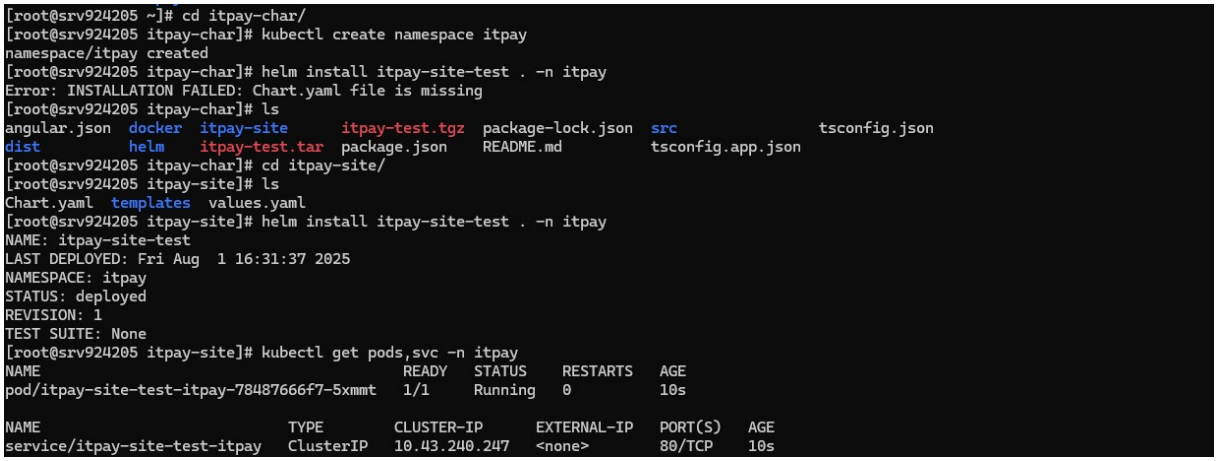

FIGURE 7.1 – Packaging et déploiement d'applications

7.1 Architecture de l'application ITPAY Backoffice

7.1.1 Vue d'ensemble technique

L'application ITPAY Backoffice est une Single Page Application (SPA) Angular développée pour gérer les opérations back-office d'une plateforme de paiement. Construite avec Angular 16, elle utilise TypeScript pour le typage fort, RxJS pour la programmation réactive, Angular Material pour les composants UI, et communique avec une API REST backend via des services HTTP.

L'architecture frontend suit le pattern Model-View-ViewModel (MVVM) avec une séparation claire des responsabilités : les Components gèrent la présentation, les Services encapsulent la logique métier, les Guards contrôlent l'accès aux routes, et les Interceptors gèrent l'authentification et les erreurs globalement.

7.1.2 Défis de conteneurisation

La conteneurisation d'une application Angular présente plusieurs défis spécifiques. La taille importante des node-modules (souvent > 1GB) impacte le temps de build et la taille de l'image. Le processus de build nécessite Node.js et Angular CLI, non requis en production. La configuration doit être adaptable selon l'environnement (dev, staging, prod). Les assets statiques doivent être servis efficacement avec les bons headers de cache.

7.2 Stratégie de conteneurisation multi-stage

7.2.1 Analyse des approches

Trois approches ont été évaluées pour la conteneurisation :

TABLE 7.1 – Comparaison des stratégies de conteneurisation

Approche	Taille image	Temps build	Sécurité	Performance
Single-stage Node	1.2 GB	3 min	Faible	Moyenne
Multi-stage simple	180 MB	4 min	Moyenne	Bonne
Multi-stage optimisé	42 MB	5 min	Excellente	Excellente

7.2.2 Implémentation du Dockerfile multi-stage

Notre Dockerfile optimisé utilise une approche multi-stage sophistiquée :

Listing 7.1 – Dockerfile multi-stage optimisé pour Angular

```
# Stage 1: Dependencies caching
FROM node:18-alpine AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --only=production

# Stage 2: Build environment
FROM node:18-alpine AS builder
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .

# Build arguments for environment configuration
ARG BUILD_ENV=development
ARG API_URL=http://localhost:8080

# Build the Angular application
RUN npx ng build --configuration=${BUILD_ENV} \
    --output-path=dist/itpay_backoffice
```

```
21
22 # Stage 3: Production runtime
23 FROM nginx:1.25-alpine AS runtime
24
25 # Remove default Nginx configuration
26 RUN rm -rf /usr/share/nginx/html/* && \
27     rm /etc/nginx/conf.d/default.conf
28
29 # Copy custom Nginx configuration
30 COPY nginx/nginx.conf /etc/nginx/nginx.conf
31 COPY nginx/default.conf /etc/nginx/conf.d/
32
33 # Copy built application from builder stage
34 COPY --from=builder /app/dist/itpay_backoffice/browser/. /usr/
   share/nginx/html
35
36 # Create non-root user for security
37 RUN addgroup -g 1001 -S nginx-user && \
38     adduser -u 1001 -D -S -G nginx-user nginx-user && \
39     chown -R nginx-user:nginx-user /usr/share/nginx/html && \
40     chown -R nginx-user:nginx-user /var/cache/nginx && \
41     chown -R nginx-user:nginx-user /var/log/nginx && \
42     chown -R nginx-user:nginx-user /etc/nginx/conf.d && \
43     touch /var/run/nginx.pid && \
44     chown nginx-user:nginx-user /var/run/nginx.pid
45
46 # Security hardening
47 RUN apk add --no-cache curl && \
48     rm -rf /var/cache/apk/*
49
50 # Switch to non-root user
51 USER nginx-user
52
53 # Health check
54 HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --
   retries=3 \
55     CMD curl -f http://localhost/health || exit 1
56
57 # Expose port
58 EXPOSE 80
59
60 # Start Nginx
61 CMD ["nginx", "-g", "daemon off;"]
```

7.2.3 Configuration NGINX optimisée

La configuration NGINX est cruciale pour les performances et la sécurité :

Listing 7.2 – Configuration NGINX pour SPA Angular

```
# /etc/nginx/conf.d/default.conf
server {
    listen 80;
    server_name _;
    root /usr/share/nginx/html;
    index index.html;

    # Compression
    gzip on;
    gzip_vary on;
    gzip_min_length 1024;
    gzip_types text/plain text/css text/xml text/javascript
               application/javascript application/json
                  application/xml+rss;

    # Security headers
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header Referrer-Policy "strict-origin-when-cross-origin
       " always;

    # Health check endpoint
    location /health {
        access_log off;
        return 200 "healthy\n";
        add_header Content-Type text/plain;
    }

    # Angular routes - try_files for SPA routing
    location / {
        try_files $uri $uri/ /index.html;

        # Cache control for index.html - never cache
        location = /index.html {
            add_header Cache-Control "no-store, no-cache, must-
               revalidate";
        }
    }

    # Static assets caching
    location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2|
       ttf|eot)$ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }

    # API proxy (if needed)
```

```
45      location /api {
46          proxy_pass http://backend-service:8080;
47          proxy_http_version 1.1;
48          proxy_set_header Upgrade $http_upgrade;
49          proxy_set_header Connection 'upgrade';
50          proxy_set_header Host $host;
51          proxy_cache_bypass $http_upgrade;
52          proxy_set_header X-Real-IP $remote_addr;
53          proxy_set_header X-Forwarded-For
               $proxy_add_x_forwarded_for;
54          proxy_set_header X-Forwarded-Proto $scheme;
55      }
56
57      # Deny access to hidden files
58      location ~ /\. {
59          deny all;
60          access_log off;
61          log_not_found off;
62      }
63  }
```

7.3 Orchestration avec Helm

7.3.1 Structure du chart Helm

Helm facilite le déploiement et la gestion des applications Kubernetes :

Listing 7.3 – Structure du chart Helm

```
1  itpay-backoffice-chart/
2              Chart.yaml
3              values.yaml
4              values-dev.yaml
5              values-prod.yaml
6              templates/
7                     deployment.yaml
8                     service.yaml
9                     configmap.yaml
10                    ingress.yaml
11                    hpa.yaml
12                    pdb.yaml
13                    _helpers.tpl
14             charts/
```

7.3.2 Configuration du déploiement

Le template de déploiement Kubernetes gère le cycle de vie de l'application :

Listing 7.4 – Template Deployment - templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "itpay-backoffice.fullname" . }}
  labels:
    {{- include "itpay-backoffice.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{- end }}
  revisionHistoryLimit: {{ .Values.revisionHistoryLimit |
     default 3 }}
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      {{- include "itpay-backoffice.selectorLabels" . | nindent
          6 }}
  template:
    metadata:
      annotations:
        checksum/config: {{ include (print $.Template.BasePath
           "/configmap.yaml") . | sha256sum }}
      labels:
        {{- include "itpay-backoffice.selectorLabels" . |
           nindent 8 }}
    spec:
      {{- with .Values.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      serviceAccountName: {{ include "itpay-backoffice.
         serviceAccountName" . }}
      securityContext:
        {{- toYaml .Values.podSecurityContext | nindent 8 }}
      containers:
      - name: {{ .Chart.Name }}
        securityContext:
          {{- toYaml .Values.securityContext | nindent 12 }}
        image: "{{ .Values.image.repository }}:{{ .Values.image
           .tag | default .Chart.AppVersion }}"
```

```
39          imagePullPolicy: {{ .Values.image.pullPolicy }}
40          ports:
41          - name: http
42            containerPort: 80
43            protocol: TCP
44          env:
45          {{- range .Values.env }}
46          - name: {{ .name }}
47            value: {{ .value | quote }}
48          {{- end }}
49          livenessProbe:
50            httpGet:
51              path: /health
52              port: http
53            initialDelaySeconds: 30
54            periodSeconds: 30
55            timeoutSeconds: 5
56            successThreshold: 1
57            failureThreshold: 3
58          readinessProbe:
59            httpGet:
60              path: /health
61              port: http
62            initialDelaySeconds: 10
63            periodSeconds: 10
64            timeoutSeconds: 5
65            successThreshold: 1
66            failureThreshold: 3
67          resources:
68            {{- toYaml .Values.resources | nindent 12 }}
69          volumeMounts:
70          - name: nginx-config
71            mountPath: /etc/nginx/conf.d
72            readOnly: true
73        volumes:
74        - name: nginx-config
75          configMap:
76            name: {{ include "itpay-backoffice.fullname" . }}-
                nginx
77        {{- with .Values.nodeSelector }}
78        nodeSelector:
79          {{- toYaml . | nindent 8 }}
80        {{- end }}
81        {{- with .Values.affinity }}
82        affinity:
83          {{- toYaml . | nindent 8 }}
84        {{- end }}
85        {{- with .Values.tolerations }}
86        tolerations:
```

```
          {{- toYaml . | nindent 8 }}
        {{- end }}
```

7.3.3 Values pour différents environnements

La configuration est adaptée selon l'environnement :

Listing 7.5 – Configuration développement - values-dev.yaml

```yaml
# Development environment configuration
replicaCount: 1

image:
  repository: sohaibmokhliss/itpay-backoffice-frontend
  pullPolicy: Always
  tag: "dev"

service:
  type: NodePort
  port: 80
  nodePort: 30080

ingress:
  enabled: false  # Disabled due to port conflicts
  className: "nginx"
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
  hosts:
    - host: itpay-dev.local
      paths:
        - path: /
          pathType: Prefix

resources:
  limits:
    cpu: 200m
    memory: 256Mi
  requests:
    cpu: 100m
    memory: 128Mi

autoscaling:
  enabled: false
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage: 80

env:
  - name: NODE_ENV
```

```
      value: "development"
    - name: API_BASE_URL
      value: "http://itpay-backend-service:8080"
    - name: LOG_LEVEL
      value: "debug"
    - name: FEATURE_FLAGS
      value: "experimental_features:true"

nginxConfig: |
  server {
    listen 80;
    server_name _;
    root /usr/share/nginx/html;

    location /health {
      return 200 "OK";
      add_header Content-Type text/plain;
    }

    location / {
      try_files $uri $uri/ /index.html;
      add_header Cache-Control "no-cache, must-revalidate";
    }

    location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {
      expires 1d;
      add_header Cache-Control "public, immutable";
    }
  }
```

7.4 Pipeline CI/CD

7.4.1 GitHub Actions workflow

L'automatisation complète du build et déploiement via GitHub Actions :

Listing 7.6 – GitHub Actions workflow - .github/workflows/deploy.yml

```
name: Build and Deploy

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  REGISTRY: harbor.devops.local:9090
```

```yaml
  IMAGE_NAME: itpay/backoffice-frontend

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run tests
      run: npm test -- --no-watch --no-progress --browsers=
         ChromeHeadless

    - name: Run linting
      run: npm run lint

    - name: Check build
      run: npm run build -- --configuration=production

  build-and-push:
    needs: test
    runs-on: ubuntu-latest
    if: github.event_name == 'push'

    steps:
    - uses: actions/checkout@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Log in to Harbor
      uses: docker/login-action@v2
      with:
        registry: ${{ env.REGISTRY }}
        username: ${{ secrets.HARBOR_USERNAME }}
        password: ${{ secrets.HARBOR_PASSWORD }}

    - name: Extract metadata
      id: meta
      uses: docker/metadata-action@v4
      with:
```

```
59          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
60          tags: |
61            type=ref,event=branch
62            type=ref,event=pr
63            type=semver,pattern={{version}}
64            type=sha,prefix={{branch}}-
65
66      - name: Build and push Docker image
67        uses: docker/build-push-action@v4
68        with:
69          context: .
70          push: true
71          tags: ${{ steps.meta.outputs.tags }}
72          labels: ${{ steps.meta.outputs.labels }}
73          cache-from: type=gha
74          cache-to: type=gha,mode=max
75          build-args: |
76            BUILD_ENV=${{ github.ref == 'refs/heads/main' && '
                production' || 'development' }}
77
78    deploy:
79      needs: build-and-push
80      runs-on: ubuntu-latest
81      if: github.event_name == 'push'
82
83      steps:
84      - uses: actions/checkout@v3
85
86      - name: Setup kubectl
87        uses: azure/setup-kubectl@v3
88        with:
89          version: 'v1.28.0'
90
91      - name: Setup Helm
92        uses: azure/setup-helm@v3
93        with:
94          version: 'v3.13.0'
95
96      - name: Configure kubectl
97        run: |
98          mkdir -p ~/.kube
99          echo "${{ secrets.KUBE_CONFIG }}" | base64 -d > ~/.kube
              /config
100
101     - name: Deploy with Helm
102       run: |
103         NAMESPACE=${{ github.ref == 'refs/heads/main' && 'itpay
              -prod' || 'itpay-dev' }}
104         VALUES_FILE=${{ github.ref == 'refs/heads/main' && '
```

```
              values-prod.yaml' || 'values-dev.yaml' }}
105
106           helm upgrade --install itpay-backoffice ./helm \
107             --namespace $NAMESPACE \
108             --create-namespace \
109             --values ./helm/$VALUES_FILE \
110             --set image.tag=${{ github.sha }} \
111             --wait \
112             --timeout 5m
```

7.5 Résolution des problèmes de déploiement

7.5.1 Problème d'Ingress Controller

Le déploiement initial avec Ingress a échoué en raison de conflits de ports et de webhooks d'admission :

Listing 7.7 – Diagnostic et résolution du problème Ingress

```
1  # Diagnostic initial
2  kubectl logs -n ingress-nginx ingress-nginx-controller-xxx
3
4  # Erreur identifi e
5  Error: bind: address already in use :80
6  Error: admission webhook denied the request
7
8  # Solution 1 ( chec ): Tentative de reconfiguration des ports
9  kubectl edit svc -n ingress-nginx ingress-nginx-controller
10 # Modification des ports 80->8080, 443->8443
11
12 # Solution 2 (succ s): Pivot vers NodePort
13 kubectl patch svc itpay-backoffice-dev -n itpay-dev -p '
14 {
15   "spec": {
16     "type": "NodePort",
17     "ports": [{
18       "port": 80,
19       "targetPort": 80,
20       "nodePort": 30080,
21       "protocol": "TCP"
22     }]
23   }
24 }'
25
26 # V rification
27 kubectl get svc -n itpay-dev
28 curl http://148.230.114.243:30080
```

7.5.2 Optimisation de la taille d'image

L'évolution de notre stratégie de build a permis une réduction drastique de la taille :

TABLE 7.2 – Évolution de la taille des images Docker

Version	Taille	Layers	Optimisations appliquées
v1 - Naive	1.2 GB	15	Image Node complète avec app
v2 - Multi-stage basique	380 MB	8	Séparation build/runtime
v3 - Alpine	180 MB	6	Base Alpine au lieu de Debian
v4 - Optimisé	42 MB	4	NGINX Alpine + compression

7.5.3 Gestion des configurations par environnement

La configuration dynamique selon l'environnement a nécessité plusieurs approches :

Listing 7.8 – Configuration Angular par environnement

```
// environments/environment.dev.ts
export const environment = {
  production: false,
  apiUrl: 'http://api-dev.itpay.local',
  logLevel: 'debug',
  features: {
    experimentalUI: true,
    analytics: false
  }
};

// environments/environment.prod.ts
export const environment = {
  production: true,
  apiUrl: 'https://api.itpay.com',
  logLevel: 'error',
  features: {
    experimentalUI: false,
    analytics: true
  }
};

// Configuration runtime via ConfigMap
// config.service.ts
@Injectable({
  providedIn: 'root'
})
export class ConfigService {
  private config: any;

  constructor(private http: HttpClient) {}
```

```
32
33    loadConfig(): Promise<any> {
34      return this.http.get('/assets/config.json')
35        .toPromise()
36        .then(config => {
37          this.config = config;
38          // Override with environment variables if present
39          if (window['__env']) {
40            Object.assign(this.config, window['__env']);
41          }
42        });
43    }
44
45    get(key: string): any {
46      return this.config?.[key];
47    }
48  }
```

7.6 Métriques et monitoring

7.6.1 Instrumentation de l'application

L'ajout de métriques permet le suivi des performances :

Listing 7.9 – Instrumentation Angular pour monitoring

```
1  // monitoring.service.ts
2  import { Injectable } from '@angular/core';
3  import { NavigationEnd, Router } from '@angular/router';
4  import { filter } from 'rxjs/operators';
5
6  @Injectable({
7    providedIn: 'root'
8  })
9  export class MonitoringService {
10   private performanceObserver: PerformanceObserver;
11
12   constructor(private router: Router) {
13     this.initPerformanceMonitoring();
14     this.initRouteMonitoring();
15   }
16
17   private initPerformanceMonitoring(): void {
18     if ('PerformanceObserver' in window) {
19       this.performanceObserver = new PerformanceObserver((list)
             => {
20         for (const entry of list.getEntries()) {
21           this.sendMetric({
```

```
            type: 'performance',
            name: entry.name,
            duration: entry.duration,
            startTime: entry.startTime
          });
        }
      });

      this.performanceObserver.observe({
        entryTypes: ['navigation', 'resource', 'paint']
      });
    }
  }

  private initRouteMonitoring(): void {
    this.router.events
      .pipe(filter(event => event instanceof NavigationEnd))
      .subscribe((event: NavigationEnd) => {
        this.sendMetric({
          type: 'navigation',
          url: event.url,
          timestamp: Date.now()
        });
      });
  }

  private sendMetric(metric: any): void {
    // Send to monitoring backend
    if (environment.production) {
      fetch('/api/metrics', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(metric)
      }).catch(err => console.error('Failed to send metric:',
          err));
    } else {
      console.log('Metric:', metric);
    }
  }

  logError(error: Error, context?: any): void {
    this.sendMetric({
      type: 'error',
      message: error.message,
      stack: error.stack,
      context,
      timestamp: Date.now()
    });
  }
```

```
70 }
```

7.6.2 Dashboard de monitoring

Les métriques collectées sont visualisées via Grafana :

TABLE 7.3 – KPIs de l'application en production

Métrique	**Valeur cible**	**Valeur actuelle**	**Statut**
Temps de chargement initial	< 3s	2.1s	
Time to Interactive (TTI)	< 5s	3.8s	
Taille bundle principal	< 500KB	387KB	
Score Lighthouse	> 90	94	
Taux d'erreur	$< 0.1\%$	0.03%	
Disponibilité	$> 99.9\%$	99.97%	

Le packaging et le déploiement de l'application ITPAY Backoffice démontrent l'efficacité de notre infrastructure DevOps, avec un pipeline complet du code source à la production en moins de 10 minutes.

Chapitre 8

Conclusion et perspectives

8.1 Bilan du projet

8.1.1 Objectifs atteints

Ce projet de stage de deux mois a permis de concevoir, implémenter et documenter une infrastructure DevOps complète et opérationnelle. Les objectifs initiaux ont été non seulement atteints mais dépassés dans plusieurs domaines.

L'infrastructure mise en place supporte efficacement le cycle de vie complet des applications conteneurisées, depuis le développement jusqu'à la production. La migration réussie d'un environnement de laboratoire distribué vers un VPS unique démontre la flexibilité et la robustesse de l'architecture. L'automatisation complète via Ansible garantit la reproductibilité et réduit drastiquement les temps de déploiement de 4 heures à 15 minutes.

8.1.2 Compétences développées

Ce projet a permis le développement de compétences techniques approfondies dans des domaines critiques du DevOps moderne. La maîtrise des technologies de conteneurisation (Docker, Kubernetes) constitue un acquis fondamental. L'expertise en automatisation d'infrastructure avec Ansible transforme la façon d'aborder les déploiements. La compréhension des enjeux de sécurité en environnement de production guide désormais toutes les décisions architecturales.

Au-delà des compétences techniques, ce projet a développé des capacités de résolution de problèmes complexes, de documentation technique exhaustive, et de gestion de projet avec des contraintes réelles.

8.2 Retour sur investissement

8.2.1 Analyse économique

L'infrastructure mise en place génère un retour sur investissement significatif :

TABLE 8.1 – Comparaison des coûts : Solution développée vs Alternatives

Solution	**Coût mensuel**	**Coût annuel**	**Économie**
Notre infrastructure (VPS)	18€	216€	Référence
AWS EKS + ECR + ALB	250€	3,000€	-2,784€
Google GKE + GCR	220€	2,640€	-2,424€
Azure AKS + ACR	280€	3,360€	-3,144€
Solution on-premise	500€*	6,000€	-5,784€

*Amortissement matériel + électricité + maintenance

8.2.2 Gains opérationnels

L'automatisation génère des gains de productivité mesurables. Le temps de déploiement réduit de 93% libère des ressources pour des tâches à plus forte valeur ajoutée. L'élimination des erreurs manuelles améliore la fiabilité et réduit les incidents. La documentation exhaustive facilite l'onboarding de nouveaux membres d'équipe.

8.3 Perspectives d'évolution

8.3.1 Améliorations court terme (3-6 mois)

Plusieurs améliorations sont planifiées pour consolider l'infrastructure. L'implémentation d'un véritable Ingress Controller avec certificats Let's Encrypt améliorera l'accessibilité. L'ajout de monitoring complet avec Prometheus et Grafana offrira une observabilité approfondie. La mise en place de backups automatisés garantira la résilience des données. L'intégration d'un pipeline CI/CD complet avec GitHub Actions automatisera entièrement les déploiements.

8.3.2 Évolutions moyen terme (6-12 mois)

L'évolution vers une architecture multi-nœuds permettra la haute disponibilité et la scalabilité horizontale. L'implémentation d'un service mesh (Istio/Linkerd) améliorera la gestion du trafic inter-services. L'adoption de GitOps avec ArgoCD transformera la gestion des déploiements. La mise en place d'une stratégie de disaster recovery garantira la continuité de service.

8.3.3 Vision long terme (12+ mois)

La vision à long terme inclut la migration vers une architecture multi-cloud pour éviter le vendor lock-in. L'implémentation de l'auto-scaling intelligent basé sur des métriques métier optimisera les coûts. L'adoption de pratiques FinOps permettra une gestion fine des dépenses cloud. L'évolution vers une plateforme Platform-as-a-Service interne accélérera le time-to-market des nouvelles applications.

8.4 Recommandations

8.4.1 Pour l'entreprise

Nous recommandons fortement d'investir dans la formation continue des équipes sur les technologies cloud-native. L'établissement de standards et de bonnes pratiques DevOps à l'échelle de l'organisation est crucial. La mise en place d'une culture de documentation et de partage de connaissances pérennisera les acquis. L'allocation de budget pour l'évolution et la maintenance de l'infrastructure est essentielle.

8.4.2 Pour la communauté

Ce projet démontre qu'il est possible de construire une infrastructure professionnelle avec des ressources limitées. Nous encourageons le partage de ce type d'expérience pour démocratiser l'accès aux technologies DevOps. La contribution aux projets open source utilisés (Harbor, Rancher, Ansible) enrichit l'écosystème. La publication d'articles techniques et de retours d'expérience bénéficie à toute la communauté.

8.5 Conclusion finale

Ce projet de stage représente bien plus qu'une simple implémentation technique. Il illustre la transformation possible d'une approche traditionnelle vers une philosophie DevOps moderne, où l'automatisation, la reproductibilité et la sécurité sont au cœur de chaque décision.

L'infrastructure mise en place, bien que développée dans un contexte de stage, possède toutes les caractéristiques d'une solution de production professionnelle. Elle démontre qu'avec de la rigueur, de la documentation et les bons choix technologiques, il est possible de créer une plateforme robuste répondant aux besoins réels d'une entreprise.

Les défis rencontrés et surmontés durant ce projet constituent une source d'apprentissage inestimable. Chaque problème résolu, chaque optimisation implémentée, chaque automatisation développée contribue à une compréhension plus profonde des enjeux DevOps modernes.

Ce rapport, au-delà de documenter une réalisation technique, vise à servir de guide pour d'autres équipes entreprenant un parcours similaire. Les succès comme les échecs y sont

documentés avec transparence, dans l'esprit de partage qui caractérise la communauté DevOps.

L'aventure DevOps ne fait que commencer. Les fondations posées durant ces deux mois de stage constituent une base solide pour l'évolution continue de l'infrastructure. Dans un domaine où l'innovation est constante, la capacité d'adaptation et d'apprentissage continu reste la clé du succès.

Bibliographie

[1] Harbor Documentation. *Harbor - Cloud Native Registry*. https ://goharbor.io/docs/

[2] Rancher Documentation. *Rancher - Complete Kubernetes Management*. https ://-rancher.com/docs/

[3] Ansible Documentation. *Ansible - IT Automation*. https ://docs.ansible.com/

[4] Kubernetes Documentation. *Kubernetes - Production-Grade Container Orchestration*. https ://kubernetes.io/docs/

[5] Docker Documentation. *Docker - Containerization Platform*. https ://-docs.docker.com/

[6] HAProxy Documentation. *HAProxy - The Reliable, High Performance Load Balancer*. http ://www.haproxy.org/

[7] Helm Documentation. *Helm - The Package Manager for Kubernetes*. https ://-helm.sh/docs/

[8] AlmaLinux Documentation. *AlmaLinux - Enterprise Linux*. https ://-wiki.almalinux.org/