Dédicace

A ... merci ...

Remerciements

Au terme de ce travail, je tiens à exprimer ma profonde gratitude à

Résumé

Ce projet a pour objectif de

Liste des abréviations et des termes

FSS	Faculté de sciences de Sfax
OTel	OpenTelemetry
PVE	Proxmox Virtual Environment
PBS	Proxmox Backup Server
QEMU	Quick Emulator
SPICE	Simple Protocol for Independent Computing Environments
SCSI	Small Computer System Interface
LXC	Linux containers
SSh	Secure Shell
VM	Virtual Machine

Table des matières

Liste des figures

Liste des tableaux

1. Introduction générale

1.1 Introduction

Dans un contexte technologique en constante évolution, marqué par une forte concurrence et une accélération des cycles de développement, l'importance des concepts que nous avons mis en œuvre est devenue capitale. L'automatisation, qui était autrefois perçue comme un avantage optionnel, s'impose désormais comme une nécessité incontournable pour garantir la fiabilité, la sécurité et la rapidité des processus.

1.2 Présentation de l'organisme d'accueil

Oneex est une entreprise française basée à Clermont-Ferrand, avec un établissement secondaire à Paris. Elle est spécialisée dans la conception et le développement de solutions logicielles et matérielles dédiées à la vérification d'identité et à l'analyse documentaire. Grâce à des technologies avancées telles que l'intelligence artificielle et des capteurs avancés tel que les lecteurs NFC , infrarouge , hyper resolution pour une analyse approfondie des documents , qui n'est autrement pas possible a l'oeil nu, Oneex propose des solutions innovantes pour lutter contre la fraude documentaire.

1.2.1 Historique de l'entreprise

- **Création de la société (2017)** Le concept Oneex est né de l'initiative de son fondateur, confronté à la problématique de l'analyse des documents d'identité. N'ayant trouvé aucune solution souveraine respectant les contraintes réglementaires sur les données personnelles, il a décidé de créer et de développer Oneex.
- **Recherche et développement (2018-2020)** Pendant deux années, l'entreprise s'est consacrée à la recherche et au développement. Le logiciel ScanApp a ainsi été créé pour reproduire la vision humaine grâce à une intelligence artificielle capable d'analyser avec précision le pays, le format et les spécificités techniques d'un document.
- **Déploiement de la solution Desktop (2020)** Forte de ce développement, Oneex a lancé une offre complète associant hardware et software, donnant naissance à la solution Desktop.
- **Reconnaissance et impact (2021-2023)** L'entreprise a rapidement acquis une reconnaissance dans son domaine, obtenant des distinctions et certifications de la part de leaders de l'industrie. Elle poursuit aujourd'hui sa croissance à l'international.
- **Percées technologiques et évolution (2023-2024)** Oneex a enrichi ses produits de nouvelles fonctionnalités, notamment le monitoring à distance, l'accès à des statistiques détaillées, la gestion autonome du parc matériel et un accompagnement expert en fraude

documentaire. Après une levée de fonds importante en 2024, l'entreprise développe de nouveaux produits pour renforcer son positionnement en tant que leader de l'analyse documentaire.

1.2.2 Domaine d'activité

A travers ses systemes de detection de faux documents , capables a operer en mode online ainsi qu'offline Oneex propose des solutions transversales adaptées à de nombreux secteurs, notamment la santé, les banques, la sécurité et le contrôle d'accès, la location de véhicules, ainsi que les aéroports et compagnies aériennes.

1.2.3 Organisation interne

La direction et les équipes de Oneex rassemblent des profils pluridisciplinaires aux parcours variés :

Alexandre Casagrande Fondateur et Président Directeur Général. Après 12 ans au sein du Ministère des Armées et plusieurs années dans la sûreté de grands groupes, il a fondé Oneex avec la volonté de développer une solution souveraine et innovante.

François-Xavier Hauet Directeur Général. Ancien haut fonctionnaire, il a piloté la transformation numérique du Centre Interministériel de Crise puis de la Présidence de la République avant de rejoindre Oneex en 2025.

Julien Otal CTO. Développeur spécialisé dans le multiplateforme, il possède une solide expérience dans la sécurisation de systèmes critiques et la traçabilité des flux.

Xavier Matton Directeur des Opérations. Ingénieur et ancien officier au Ministère de l'Intérieur, il est expert en contrôle des flux et lutte contre la fraude documentaire.

Sébastien Kowalczuk Directeur des Opérations Sud-Ouest. Ancien enquêteur en contre-ingérence économique, il apporte une expertise forte en sécurité des données sensibles.

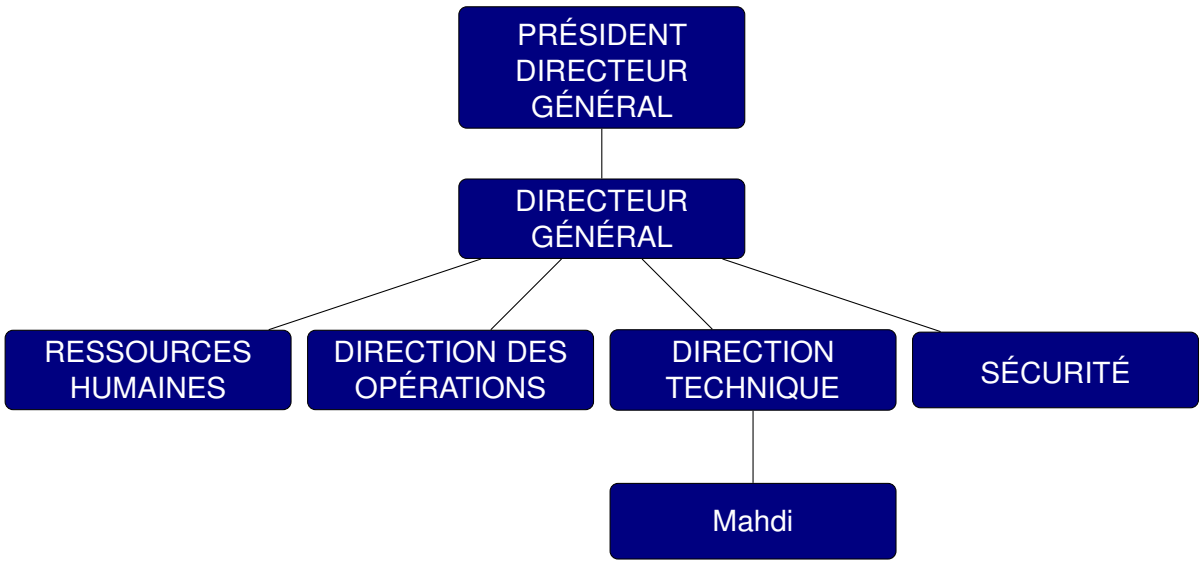

Figure 1. *Organisation interne simplifiée de l'entreprise*

1.2.4 Produits et services de l'entreprise

Oneex propose une gamme de solutions matérielles et logicielles, parmi lesquelles trois produits phares :

- **Oneex Desktop** : une station de vérification d'identité clé en main simple et intuitive pour un accueil sécurisé , cette derniere effectue chaque scan en toute confiance. Le desktop Oneex reconnaît instantanément tous les documents d'identité de 197 pays — garantissant une vérification sûre et précise à chaque fois.

Figure 2. *Oneex Desktop*

- **Oneex Suitcase** : une valise mobile permettant de réaliser des contrôles sur le terrain , portable et robuste , elle laisse les utilisateurs se profiter d'un système mobile de vérification d'identité fiable et sécurisé.

Figure 3. *Oneex Suitcase*

- **Oneex Kiosk** : un kiosque en libre-service pour l'accueil et le contrôle des visiteurs. Permet Accueiller vos visiteurs sans assistance grâce à une vérification rapide et un accès direct.Une solution pensée pour vos opérations, combinant biométrie avancée et automatisation complète de l'accès visiteurs.

Figure 4. *Oneex Kiosk*

Ces produits sont complétés par la suite logicielle Oneex Cloud et ScanApp, garantissant un pilotage centralisé et une intégration fluide dans les environnements clients.

Son application *Oneex Cloud*, une plateforme offre un contrôle centralisé et un suivi avancé des vérifications d'identité.

Cette plateforme permet un :

- **Suivi des documents scannés** : historique complet, résultats détaillés et traçabilité optimale.
- **Demande d'expertise** : sollicitation d'experts pour garantir des analyses approfondies et fiables.

- **Statistiques avancées** : exploitation des tendances de la fraude pour optimiser la gestion des incidents.

Ces solutions permettent aux entreprises de contrôler efficacement les accès, de réduire les fraudes et de fluidifier les processus d'intégration dans le respect des réglementations RGPD.

1.2.5 Services informatiques et outils internes

Les services informatiques

Oneex ScanApp constitue le cœur opérationnel de l'entreprise. Il assure l'analyse documentaire et la vérification d'identité, disponible sur postes fixes comme sur mobiles, avec une interface ergonomique et des fonctionnalités avancées. Il est constitué par une une interface utilisateur intuitive pour les ecrans des solutions que oneex offre et un moteur d'analyse de documents qui tourne localement , et est chargé par la lecture des documents , et l'execution d'une vingtaine d'algorithmes pour s'assurer de l'authenticité du document scanné. Il est composé de plusieurs modules , repartis entre le frontend et le backend, qui communiquent via des API sécurisées. Le code de ces dernieres est hebergé sur un serveur gitlab interne, garantissant ainsi la sécurité et la confidentialité des données et il est repartit en <a completer> **Oneex Cloud** complète cet écosystème en permettant :

- la gestion des vérifications,
- le suivi historique et la traçabilité,
- la sollicitation d'experts en cas de doute,
- l'analyse statistique des fraudes détectées.

de meme ceci est hébergé sur un serveur gitlab interne, garantissant ainsi la sécurité et la confidentialité des données. Ces services sont accessibles via une interface web sécurisée, permettant aux utilisateurs de gérer les opérations de vérification d'identité de manière centralisée et efficace. L'entreprise utilise également des outils de gestion de projet et de suivi des tâches, tels que You Track, pour assurer une coordination optimale entre les équipes et garantir la qualité des livrables. Le système de gestion des versions est basé sur GitLab, permettant une collaboration fluide et un suivi rigoureux des modifications apportées au code source des applications. Le code de ces dernieres est hebergé sur un serveur gitlab interne, garantissant ainsi la sécurité et la confidentialité des données et il est repartit en <a completer>

Les outils internes

L'entreprise utilise plusieurs outils collaboratifs et techniques, parmi lesquels : GitLab, Harbor, Nextcloud, Jitsi, Label Studio, YouTrack, Vault, SSO Keycloak. Les bases de données sont hébergées sur des serveurs dédiés et sécurisés, garantissant la confidentialité et la disponibilité des informations sensibles ainsi que la conformité aux réglementations et assurer un controle et une gestion des accès rigoureux.

infrastructures internes

Afin de garentir un controle total et une idependence aux fournisseurs de cloud, Oneex a opté pour une infrastructure IAAS (Infrastructure as a Service) hébergée dans des serveurs dont proxmox est installé et utilisé pour gerer toutes les ressources. Cette infrastructure est composée de serveurs physiques et virtuels, permettant de déployer les applications et services nécessaires à l'activité de l'entreprise. Oneex prefere aussi garder toutes les machines virtuelles dans un sous reseau privé interne a proxmox , en exposant les services a travers un reverse proxy Nginx. Pour toute communication inter-proxmox entre les machines virtuelles, Oneex utilise un VPN IP-Sec réseau interne dédié, garantissant ainsi la sécurité et la performance des échanges.

Actuellement, l'entreprise dispose de plusieurs serveurs physiques hebergés chez sclaeway

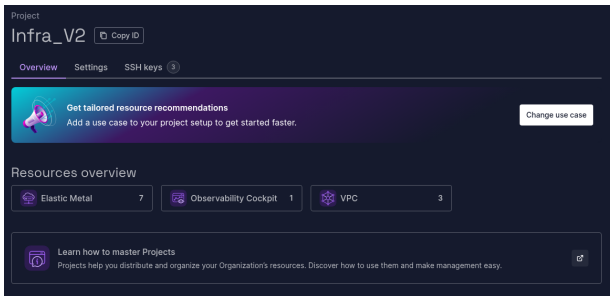

Figure 5. *Ressources cloud de Oneex*

1.3 Les projets informatiques de la société

<a completer> Oneex développe plusieurs projets informatiques stratégiques qui répondent à différents besoins métiers et techniques

1.3.1 Oneex Front

Application frontend permettant la gestion des opérations, l'affichage des données et l'interaction avec les utilisateurs finaux.

1.3.2 Oneex Back

Backend exposant les API et orchestrant les processus métiers critiques.

1.3.3 Oneex Scanner

Solution logicielle dédiée à l'acquisition et à l'analyse des documents d'identité.

1.3.4 Oneex ScanApp

Application mobile ou desktop facilitant le scan et la vérification en temps réel des documents.

1.3.5 Oneex CSharp

Projet spécifique développé en C# destiné à répondre à des besoins d'intégration ou d'outillage interne.

2. Objectifs et contexte du projet

2.1 Introduction

Dans un environnement où les systèmes d'information deviennent de plus en plus complexes et interconnectés, la gestion manuelle des infrastructures techniques pose de nombreux défis. Les entreprises doivent faire face à des exigences croissantes en matière de sécurité, de disponibilité et de performance, tout en cherchant à optimiser leurs coûts et à réduire les délais de mise en production. Cette évolution rend la gestion des infrastructures non seulement complexe, mais parfois inefficace, voire impossible à grande échelle.

Dans ce contexte, l'automatisation des processus d'infrastructure et l'adoption de solutions DevOps sont devenues des priorités stratégiques. La maîtrise de l'infrastructure et des processus de déploiement passe alors d'un luxe à une nécessité afin de soutenir rapidement et efficacement la croissance continue des besoins de l'entreprise.

Cependant, automatiser le déploiement des services ne suffit plus. Il est essentiel de garantir à tout moment leur bon fonctionnement grâce à des mécanismes de supervision et de contrôle rigoureux. La mise en place de solutions de *monitoring*, de *logging* et de *tracing* permet de disposer d'une visibilité complète sur l'état des systèmes, d'anticiper les incidents et de réagir rapidement en cas d'anomalie. Ces dispositifs contribuent également à renforcer la traçabilité et à répondre aux impératifs de conformité réglementaire.

En parallèle, le renforcement de la cybersécurité constitue un enjeu majeur. La multiplication des points d'entrée et l'interconnexion croissante des services exposent l'infrastructure à de nouvelles menaces qu'il convient de prévenir et de détecter de manière proactive.

Ce mémoire s'inscrit dans cette dynamique, avec pour objectif principal de concevoir et mettre en place une solution automatisée, sécurisée et résiliente permettant de déployer, superviser et maintenir l'infrastructure technique de l'entreprise Oneex. Le projet vise à répondre aux besoins opérationnels croissants, à réduire les erreurs manuelles et à garantir un haut niveau de qualité de service et de transparence, tout en respectant les contraintes strictes de sécurité et de conformité réglementaire.

2.2 Scénarios de référence et hypothèses de travail

Dans le cadre de l'étude préalable à la conception d'une solution d'automatisation et de sécurisation des infrastructures, il est pertinent d'envisager un ensemble de scénarios représentatifs susceptibles de se produire dans des environnements techniques comparables. Ces hypothèses permettent d'illustrer les enjeux et de définir les objectifs fonctionnels et opérationnels du projet.

2.2.1 Divergences entre environnements

Il est possible que la configuration manuelle des serveurs, opérée par plusieurs équipes successives, conduise progressivement à des écarts significatifs entre les environnements de développement, de test et de production. Les différences pourraient concerner notamment :

- Les versions des systèmes d'exploitation, des librairies et des dépendances logicielles.
- Les paramètres réseau tels que l'ouverture de ports ou l'attribution d'adresses IP.
- La définition des règles de sécurité (droits d'accès, politiques de pare-feu).

Dans un tel scénario, ces divergences pourraient générer des dysfonctionnements applicatifs lors des bascules d'environnement et accroître la difficulté de reproduire les incidents constatés.

2.2.2 Gestion hétérogène des secrets

Un autre scénario plausible concerne l'absence de processus unifié de gestion des informations sensibles (identifiants, clés d'API, certificats). Il est envisageable que ces éléments soient stockés et partagés de façon informelle, par exemple :

- Sous forme de fichiers non chiffrés sur les postes individuels.
- Par échange de courriels non sécurisés.
- Par messagerie instantanée, sans traçabilité ni archivage structuré.

Une telle situation serait susceptible d'exposer les infrastructures à des risques accrus de fuite d'informations critiques, ainsi qu'à des difficultés opérationnelles lors des renouvellements ou révocations des secrets.

2.2.3 Déficit d'observabilité

Il est également envisageable qu'une organisation n'ait pas mis en place de dispositif unifié de supervision et de journalisation. Dans ce cas, plusieurs limitations pourraient apparaître :

- L'absence de collecte systématique des métriques de performance.
- La dispersion des journaux applicatifs sur des serveurs multiples, sans agrégation centralisée.
- Le manque de mécanismes de corrélation des événements entre composants.

Un tel déficit d'observabilité réduirait la capacité à détecter précocement les anomalies, à diagnostiquer efficacement les causes racines et à mesurer le respect des engagements de qualité de service.

2.2.4 Processus de déploiement manuel et long

Dans un scénario reposant sur un déploiement entièrement manuel, la création d'une infrastructure nouvelle pourrait nécessiter plusieurs jours d'opérations successives :

1. Préparation et allocation des ressources matérielles ou virtuelles.

2. Installation des systèmes d'exploitation et des dépendances logicielles.
3. Paramétrage des droits d'accès et des configurations de sécurité.
4. Vérification manuelle de la conformité et du bon fonctionnement des services.

Un tel processus induirait des délais importants, une faible reproductibilité et une exposition accrue aux erreurs humaines.

2.2.5 Risques opérationnels associés

Ces hypothèses convergent vers un ensemble de risques potentiels, parmi lesquels :

- L'allongement des délais de livraison et la perte d'agilité opérationnelle.
- L'augmentation de la probabilité d'erreurs de configuration.
- La difficulté de garantir la sécurité des environnements et la confidentialité des données sensibles.
- L'impossibilité de disposer d'une vision globale et en temps réel de l'état de l'infrastructure.

Ces risques soulignent l'intérêt d'intégrer, dès la conception de la solution, des mécanismes de *monitoring*, de *logging* et de *tracing*, afin de renforcer la transparence et la capacité de réaction face aux incidents.

2.2.6 Justification des orientations retenues

L'analyse de ces scénarios de référence et des risques associés conduit à considérer comme prioritaire la mise en place d'une démarche structurée autour des axes suivants :

- L'automatisation des processus de déploiement et de configuration, afin de réduire les délais et d'améliorer la cohérence.
- La centralisation et la sécurisation de la gestion des secrets et des accès, pour prévenir les fuites d'informations sensibles.
- Lorsque cela est possible, l'utilisation de mécanismes de génération et de rotation automatique de secrets temporaires (par exemple des credentials ou des clés TLS via des solutions de type *Vault*), afin de limiter l'exposition prolongée des informations d'authentification.
- La mise en place d'une gestion stricte des droits d'accès, fondée sur le principe du moindre privilège et l'application des bonnes pratiques du *Zero Trust*, pour réduire la surface d'attaque et contrôler finement les autorisations.
- L'intégration d'outils d'observabilité pour assurer un suivi continu et une traçabilité complète des opérations.
- Le renforcement des contrôles de sécurité et la conformité avec les normes en vigueur.

Ces orientations constituent le socle sur lequel s'appuie le projet présenté dans ce mémoire.

2.3 Les besoins fonctionnels

Les besoins fonctionnels décrivent l'ensemble des fonctionnalités attendues de la solution envisagée, ainsi que les objectifs opérationnels qui en découlent. Ils visent à garantir la cohérence, la sécurité, la traçabilité et la résilience de l'infrastructure et des applications. Ces besoins peuvent être regroupés autour de plusieurs axes principaux :

Provisioning et configuration des ressources

- **Automatisation du provisioning des ressources** : permettre la création, la configuration et la suppression des composants d'infrastructure de manière déclarative et reproductible, afin de réduire les délais et d'éviter les interventions manuelles.
- **Gestion centralisée et cohérente des configurations** : mettre en œuvre un mécanisme d'orchestration permettant d'installer les dépendances logicielles, d'appliquer les paramétrages requis et de maintenir l'uniformité entre les différents environnements.

Déploiement et mise à jour des applications

- **Déploiement applicatif automatisé et contrôlé** : intégrer un processus déclenchant les déploiements depuis des référentiels versionnés et assurant la synchronisation permanente entre le code source et les environnements cibles.

Supervision et observabilité

- **Supervision proactive et alertes en temps réel** : disposer d'un système de surveillance permettant de collecter les métriques de performance, de visualiser l'état des services et de générer des alertes en cas d'incident.
- **Détection précoce des anomalies** : mettre en place des mécanismes d'analyse continue et d'identification des écarts de comportement afin d'anticiper les incidents et de réduire leur impact.
- **Journalisation centralisée** : assurer la collecte, le stockage et la consultation unifiée des journaux système et applicatifs.

Sécurité et gestion des accès

- **Gestion sécurisée et dynamique des secrets** : intégrer un système centralisé de stockage, de chiffrement et de distribution des informations sensibles, avec des mécanismes de rotation automatique et de durée de vie limitée des secrets lorsque cela est possible.
- **Séparation stricte des environnements** : organiser l'infrastructure en environnements distincts (développement, test, pré-production, production) afin de garantir leur isolation et de limiter les risques de contamination croisée.

Résilience et continuité de service

- **Correction automatique des incidents et des défaillances** : prévoir des processus d'auto-remédiation capables de restaurer l'état nominal des services, par exemple par le redémarrage ou le reprovisionnement automatisé des ressources en cas de panne.

Interface de pilotage

- **Interface unifiée d'administration** : proposer une interface utilisateur et/ou une API permettant d'interagir avec la plateforme de manière sécurisée et traçable.

Ces besoins fonctionnels constituent la base de la solution à concevoir, en intégrant les outils et les pratiques DevOps adaptés pour répondre aux enjeux opérationnels et réglementaires de l'entreprise.

2.4 Les besoins non fonctionnels

Les besoins non fonctionnels définissent les critères de qualité, de performance, de sécurité et de conformité que la solution doit respecter de manière transversale. Ces exigences sont essentielles pour garantir la fiabilité, la pérennité et la valeur ajoutée de la plateforme. Elles peuvent être regroupées selon plusieurs dimensions complémentaires.

Qualité de service et performance

- **Haute disponibilité** : garantir un taux de disponibilité supérieur à 99,9 % pour les services critiques, en prévoyant des mécanismes de redondance, de bascule automatique et de tolérance aux pannes.
- **Performance** : assurer des temps de réponse optimaux et constants, y compris en période de forte charge, afin de préserver la qualité d'expérience des utilisateurs et le respect des engagements contractuels (SLA).
- **Scalabilité** : permettre une montée en charge fluide et progressive de l'infrastructure, que ce soit en termes de volume de données, de nombre d'utilisateurs ou de capacités de traitement.
- **Réduction du temps de mise en production** : optimiser les processus afin de diminuer significativement les délais nécessaires au déploiement de nouvelles fonctionnalités ou de correctifs.

Sécurité et protection des données

- **Sécurité renforcée** : garantir la protection des données sensibles, la confidentialité des échanges et la résilience face aux attaques, en appliquant les principes de *security by design* et en intégrant les contrôles de sécurité dès la conception.

- **Gestion stricte des droits d'accès** : appliquer le principe du moindre privilège, segmenter les privilèges et mettre en œuvre des mécanismes de contrôle d'accès granulaires et auditables, conformément aux approches de type Zero Trust.
- **Traçabilité et auditabilité** : conserver un historique détaillé, horodaté et inviolable de toutes les opérations critiques, des changements de configuration et des actions administratives.
- **Audits automatiques de conformité et d'intégrité** : générer périodiquement des rapports permettant de vérifier la cohérence des configurations, la robustesse des mécanismes de sécurité et le respect des politiques internes et réglementaires.

Évolutivité et maintenabilité

- **Maintenabilité** : faciliter l'application des mises à jour logicielles, l'évolution des configurations et l'intégration de nouvelles fonctionnalités, tout en minimisant les interruptions de service.
- **Source unique de vérité (Single Source of Truth)** : centraliser et versionner l'ensemble des configurations, des états d'infrastructure et de la documentation technique dans un référentiel unique, fiable et auditable.
- **Respect du principe DRY (Don't Repeat Yourself)** : structurer les configurations et les processus de manière modulaire et réutilisable, afin d'éviter les duplications inutiles, de garantir la cohérence et de réduire le risque d'erreurs lors des modifications ou évolutions.

Conformité réglementaire et normes applicables

- **Conformité réglementaire** : respecter les obligations légales et les standards sectoriels en vigueur (RGPD, ISO 27001, NIS2, etc.), ainsi que les exigences spécifiques à l'activité et aux données traitées.

2.5 Architecture du projet

L'architecture globale du projet repose sur une approche moderne, modulaire et automatisée, intégrant les principes de l'Infrastructure as Code, du GitOps, de la conteneurisation et de l'observabilité. Elle a été conçue pour répondre aux exigences de performance, de sécurité, de scalabilité et de maintenabilité en s'appuyant sur plusieurs principes de l'état de l'art et bonnes pratiques du génie logiciel et de l'ingénierie des plateformes

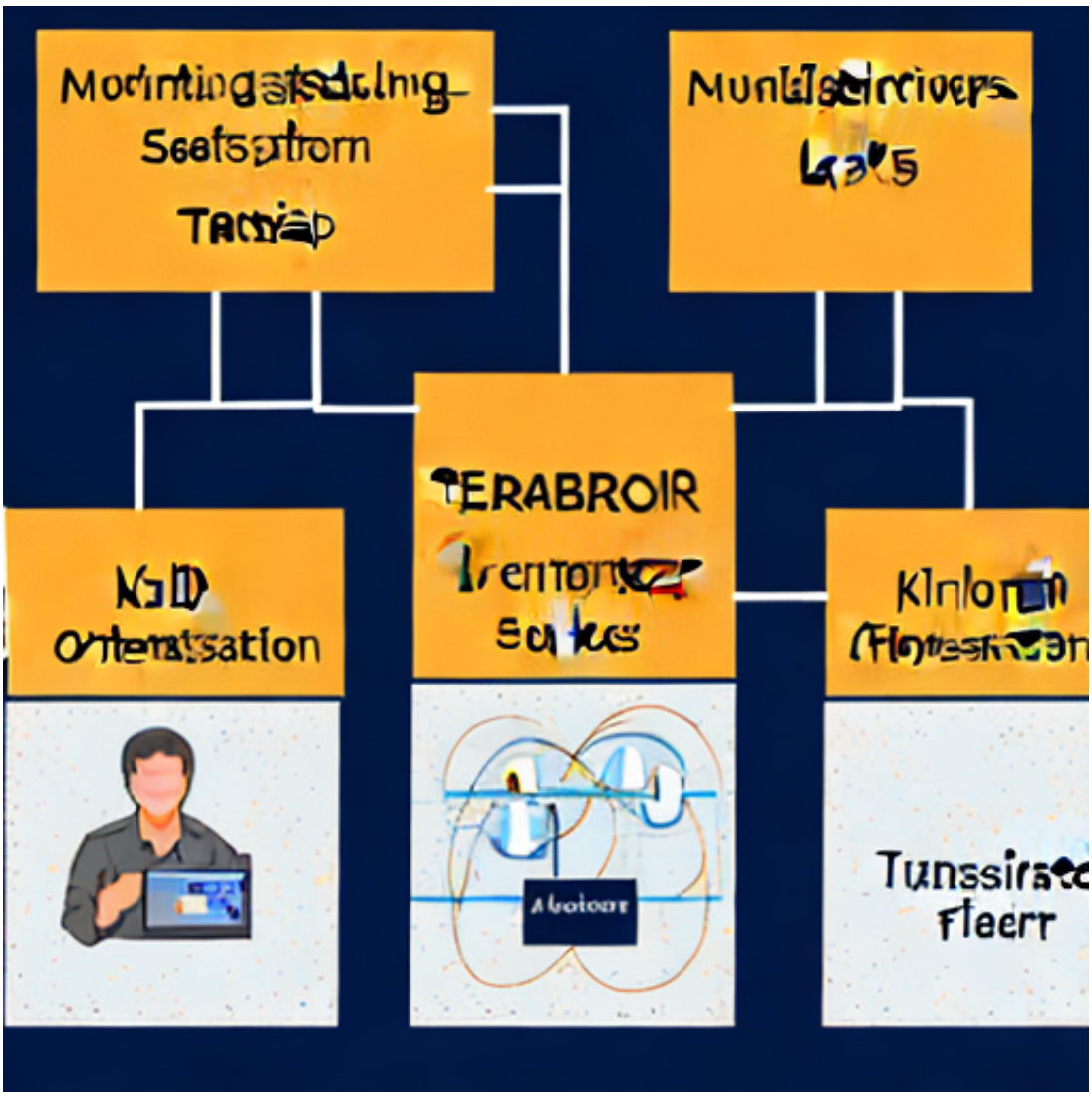

Figure 6. Schéma d'architecture globale

2.5.1 Infrastructure virtualisée et provisioning automatisé

L'infrastructure physique est virtualisée au moyen d'une plateforme Proxmox. La création des ressources a été entièrement automatisée via une démarche Infrastructure as Code.

Infrastructure as Code avec Terraform

Terraform a permis de décrire et de provisionner l'ensemble des ressources suivantes de façon déclarative et reproductible :

- Les machines virtuelles dédiées aux nœuds Kubernetes (masters et workers) et aux services utilitaires.
- Les réseaux virtuels, sous-réseaux et interfaces.
- Les volumes de stockage attachés aux instances.
- Les configurations initiales via cloud-init.

Les modules Terraform ont été organisés par domaines fonctionnels afin de favoriser leur réutilisation et leur évolutivité.

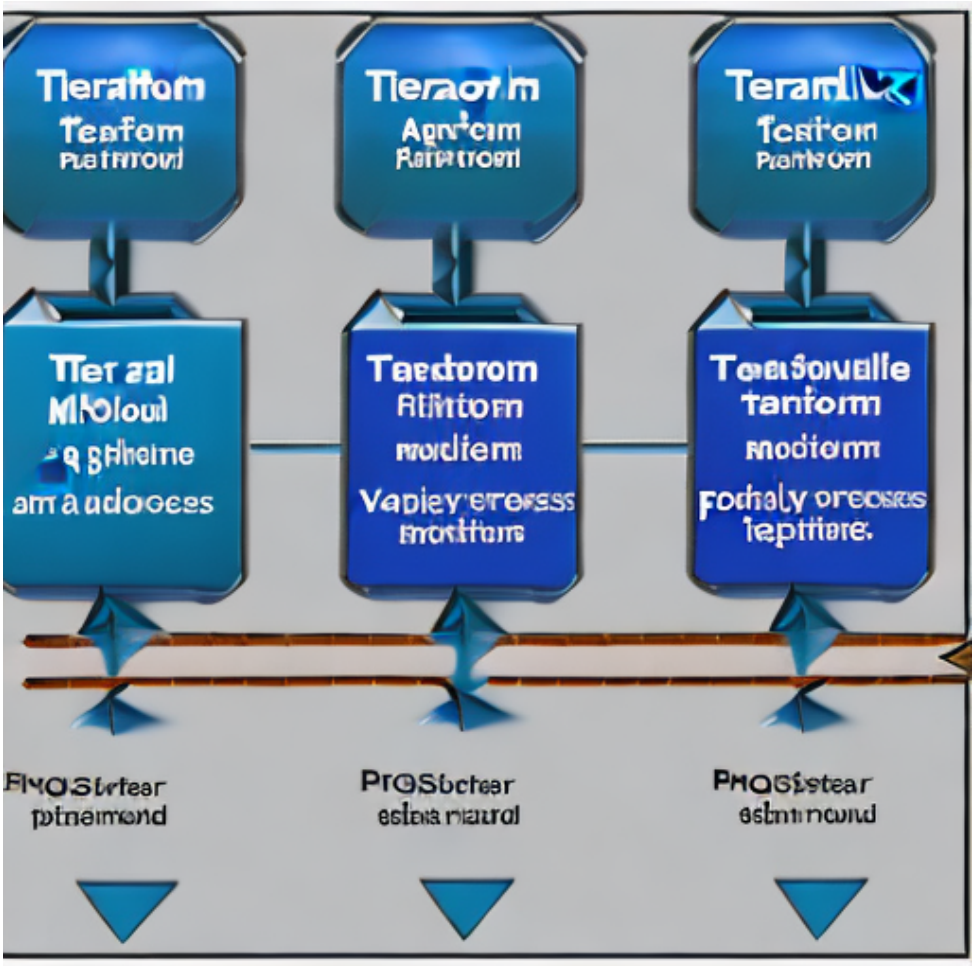

Figure 7. Schéma du processus de provisioning automatisé avec Terraform

Configuration automatisée avec Ansible

Après la création des ressources, Ansible assure la configuration des serveurs :

- Installation et configuration du cluster Kubernetes.
- Déploiement des composants de monitoring et de sécurité.
- Configuration des services réseau et des paramètres système.
- Application des règles de sécurité renforcées.

Cette étape garantit l'homogénéité et la reproductibilité des environnements.

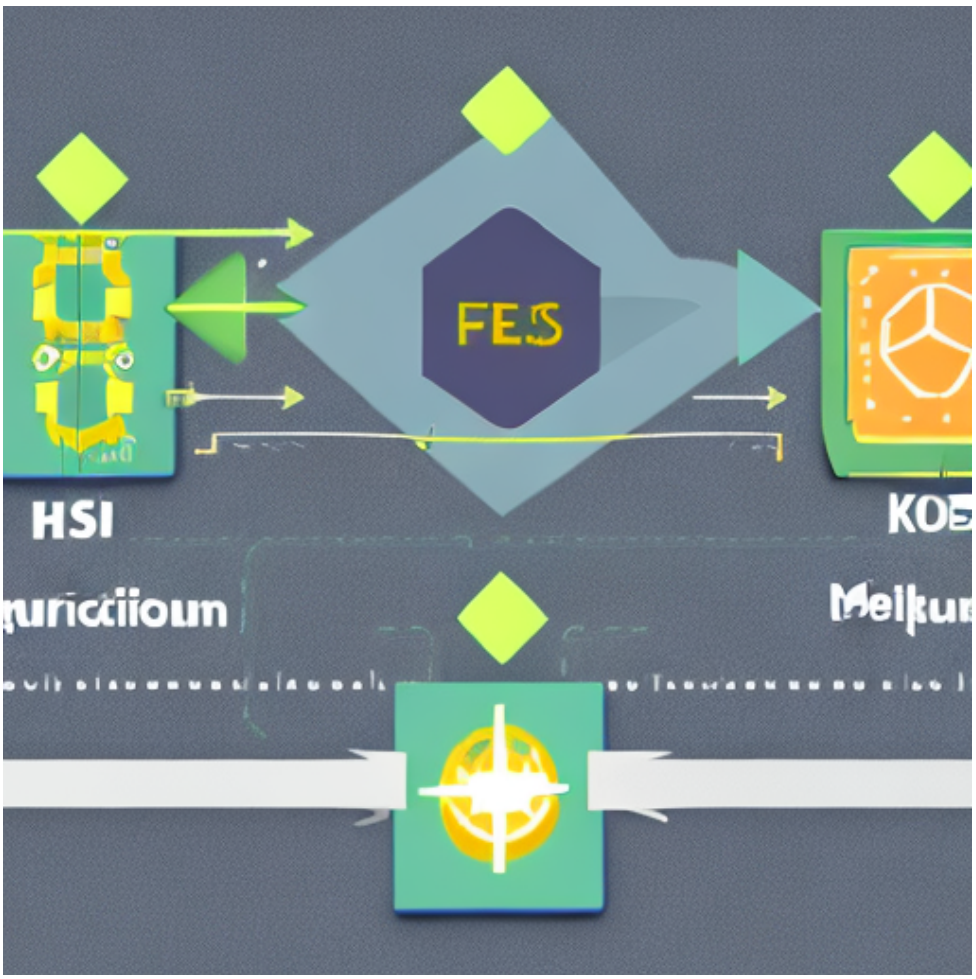

Figure 8. Processus d'automatisation de la configuration avec Ansible

2.5.2 Orchestration et déploiement applicatif

Cluster Kubernetes

Kubernetes est le cœur de l'architecture d'orchestration. Il assure la gestion :

- Du cycle de vie des conteneurs applicatifs.
- De l'équilibrage de charge et du scaling horizontal.
- De l'isolation des workloads.

Déploiement GitOps avec Argo CD

Argo CD implémente une stratégie GitOps permettant :

- Le déclenchement automatique des déploiements depuis un dépôt Git.
- La synchronisation continue des manifestes Kubernetes.
- La traçabilité complète des changements et la simplification des rollbacks.

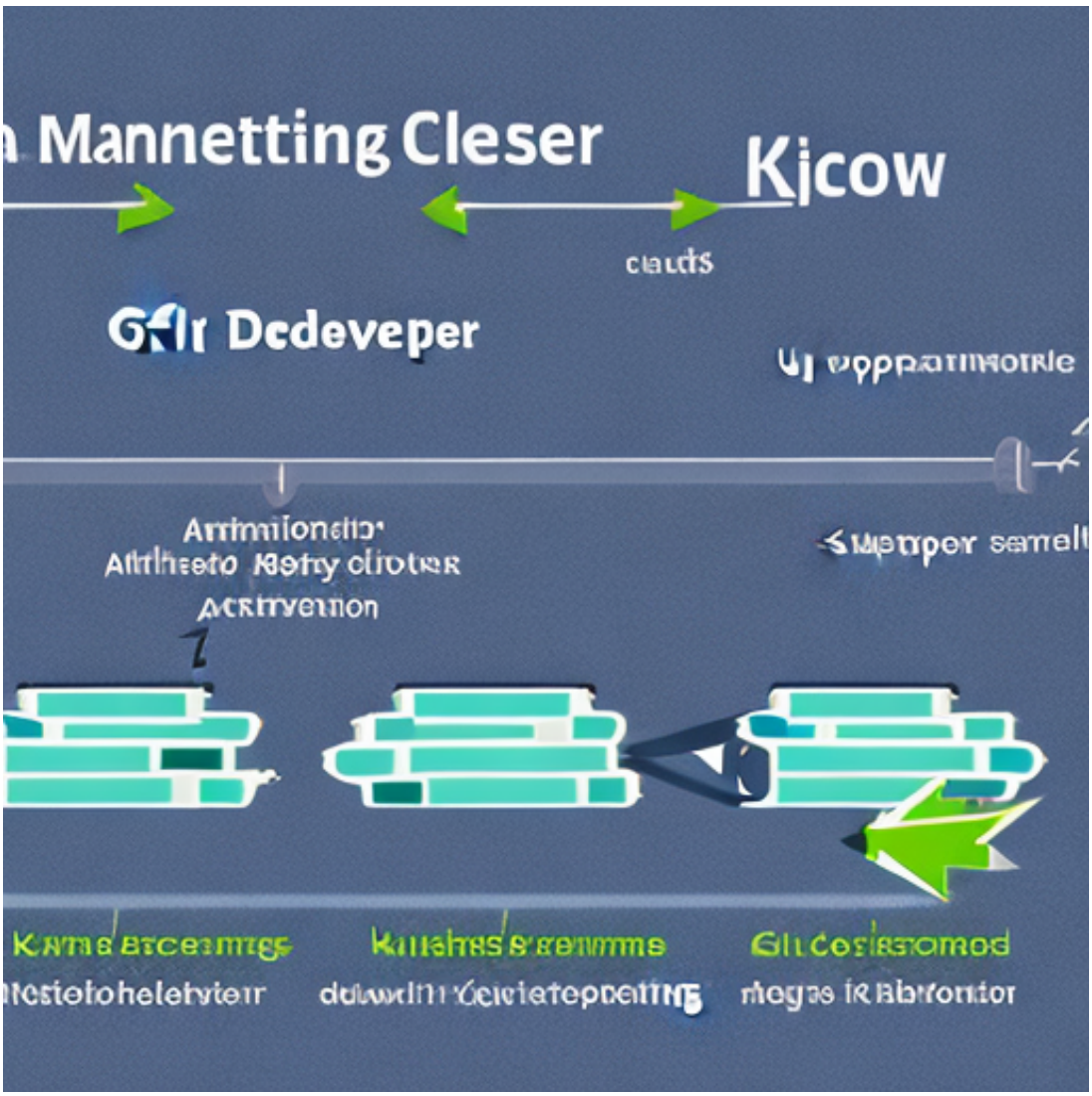

Figure 9. Flux GitOps des déploiements avec Argo CD

2.5.3 Observabilité et monitoring

La supervision s'appuie sur un ensemble d'outils intégrés :

- **Prometheus** pour la collecte des métriques et des alertes.
- **Grafana** pour la visualisation des indicateurs.
- **Loki** pour la centralisation des logs applicatifs et système.
- **Tempo** pour la gestion des traces distribuées.

Cette stack assure une observabilité complète et un diagnostic précis en cas d'incident.

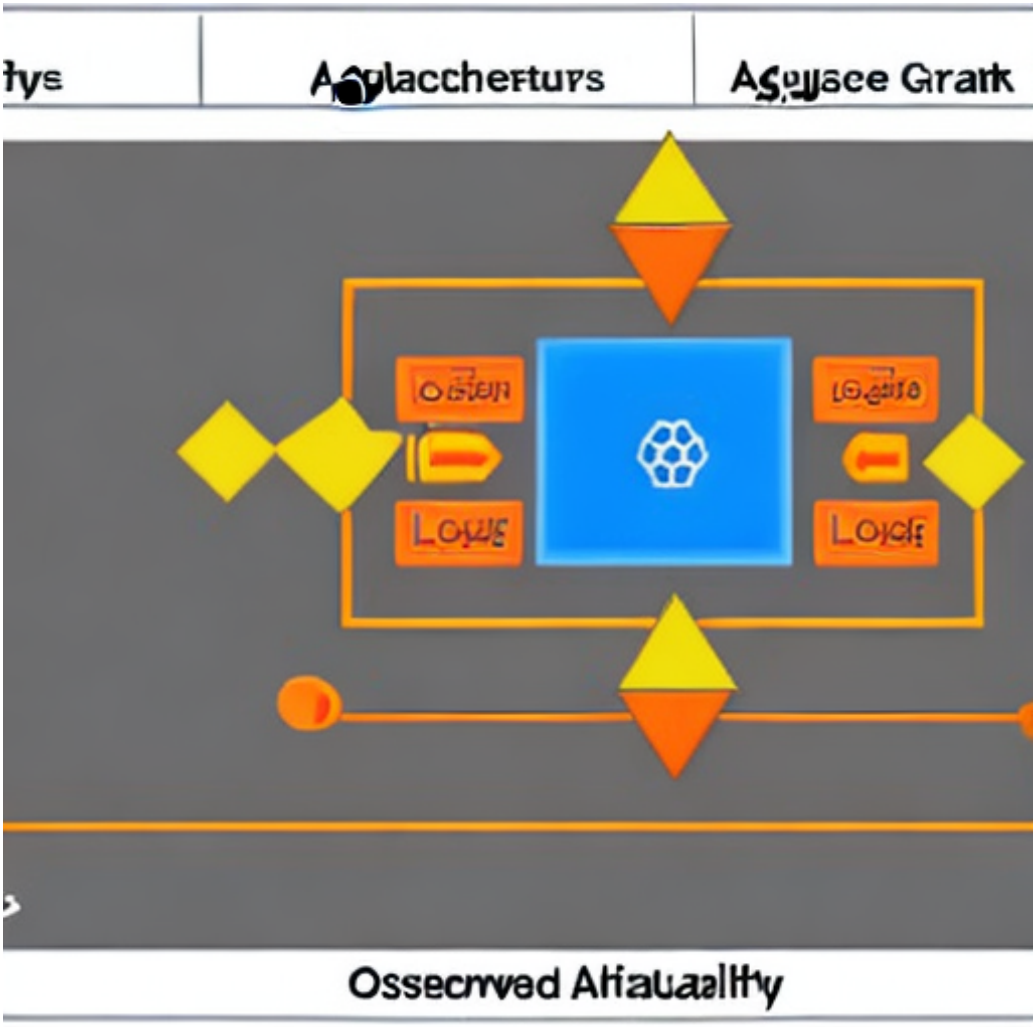

Figure 10. Architecture de la stack d'observabilité

2.5.4 Stockage distribué et persistance des données

Le stockage persistant est assuré par Longhorn, qui fournit :

- Des volumes répliqués tolérants aux pannes.
- Des snapshots automatisés et des fonctionnalités de restauration.
- Une intégration transparente avec Kubernetes.

2.5.5 Gestion sécurisée des secrets

Vault joue le rôle de coffre-fort centralisé :

- Stockage chiffré des mots de passe, certificats et tokens.
- Génération dynamique de secrets temporaires.
- Politiques d'accès granulaires pour limiter les droits.

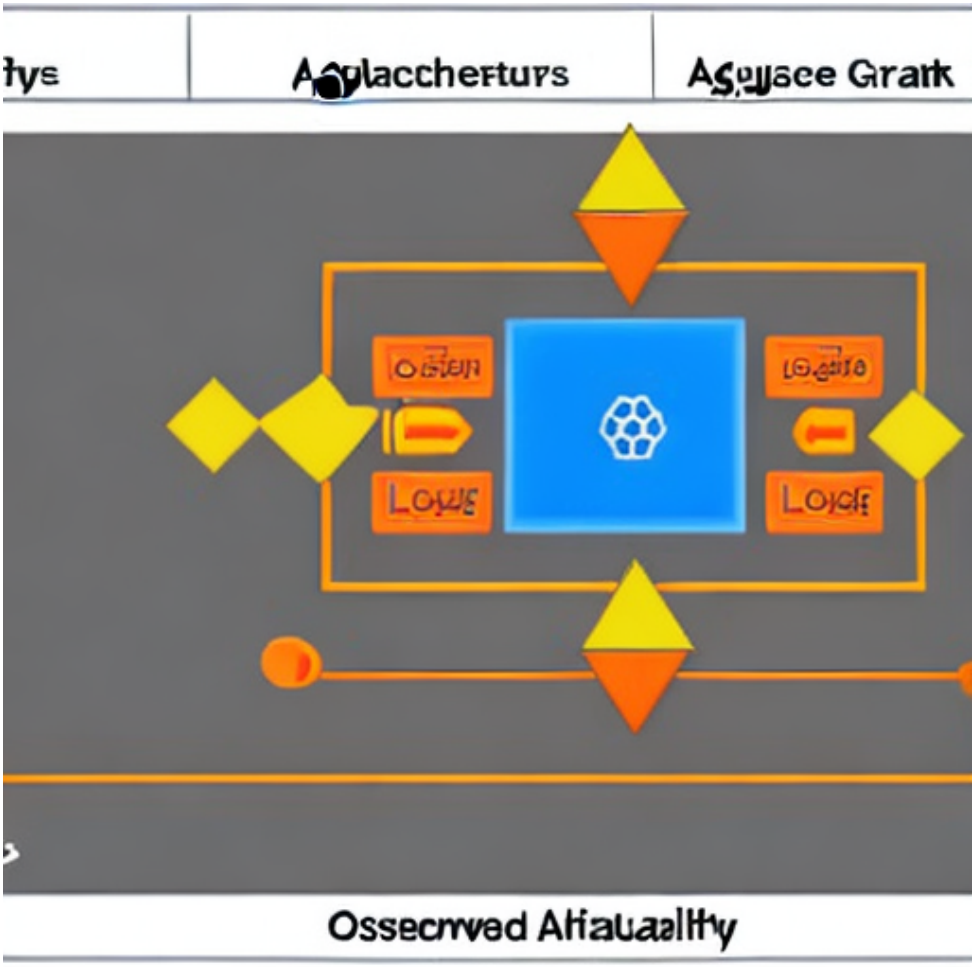

Figure 11. Flux de gestion sécurisée des secrets avec Vault

2.5.6 Sécurité réseau et accès administratifs

La sécurité périmétrique est confiée à un pare-feu pfSense :

- Contrôle des flux entrants et sortants via des règles spécifiques.
- Mise en place d'un VPN pour les accès administratifs.
- Surveillance active des connexions.

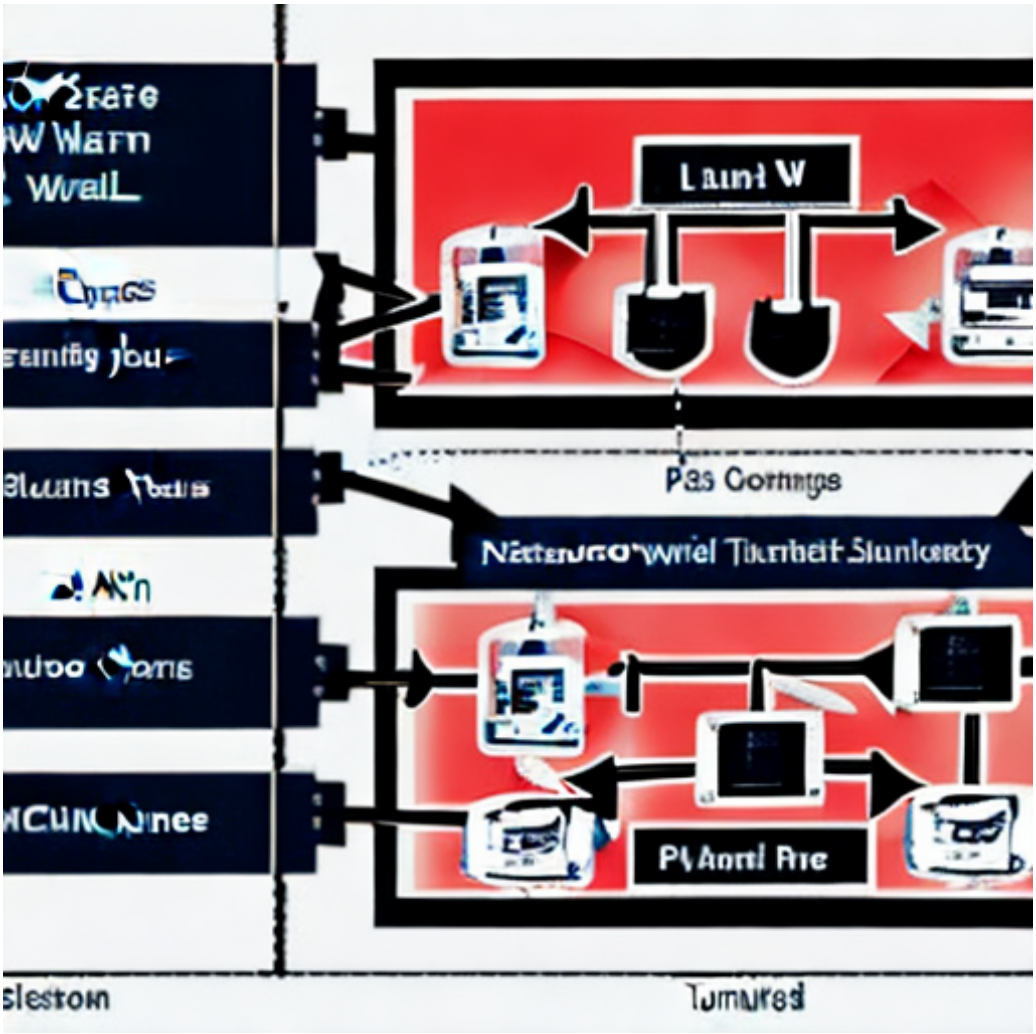

Figure 12. Architecture de sécurité réseau avec pfSense

2.5.7 Services internes pour le cycle de vie applicatif

Pour répondre aux besoins opérationnels des équipes, plusieurs services internes ont été déployés:

- **GitLab** pour la gestion des dépôts, la CI/CD et la revue de code.
- **Harbor** comme registre privé de conteneurs.
- **YouTrack** pour le suivi des incidents et la gestion des tâches.
- **Nextcloud** pour le partage et l'archivage des documents.

Ces outils sont hébergés sur Kubernetes afin d'assurer leur haute disponibilité.

Figure 13. Panorama des services internes

2.5.8 Environnements de test et de production

- Des environnements de **test** et de **staging** reprenant la même architecture que la production ont été mis en place afin de valider les développements et les mises à jour.
- Les environnements de **production** ont été configurés avec des sauvegardes automatiques et des mesures renforcées de sécurité et de supervision.

2.5.9 Intégration et automatisation des déploiements

GitLab CI assure l'automatisation du cycle de vie applicatif :

- Construction des images conteneurs.
- Exécution des tests automatisés.
- Déploiement vers les environnements Kubernetes.

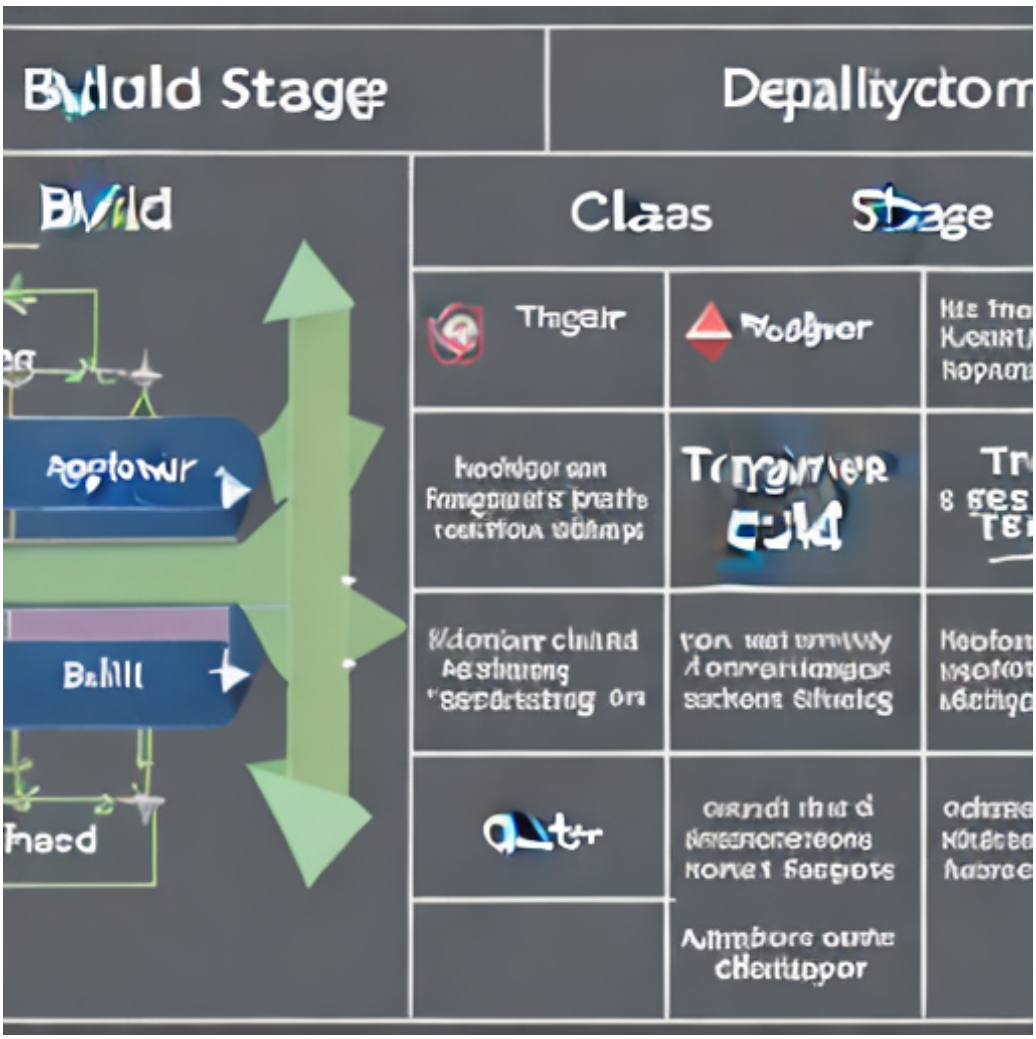

Figure 14. Processus d'intégration et de déploiement continu

Synthèse

L'architecture ainsi décrite allie modularité, automatisation et sécurité. Chaque composant contribue à la robustesse de la solution, tout en facilitant son évolutivité et sa maintenance.

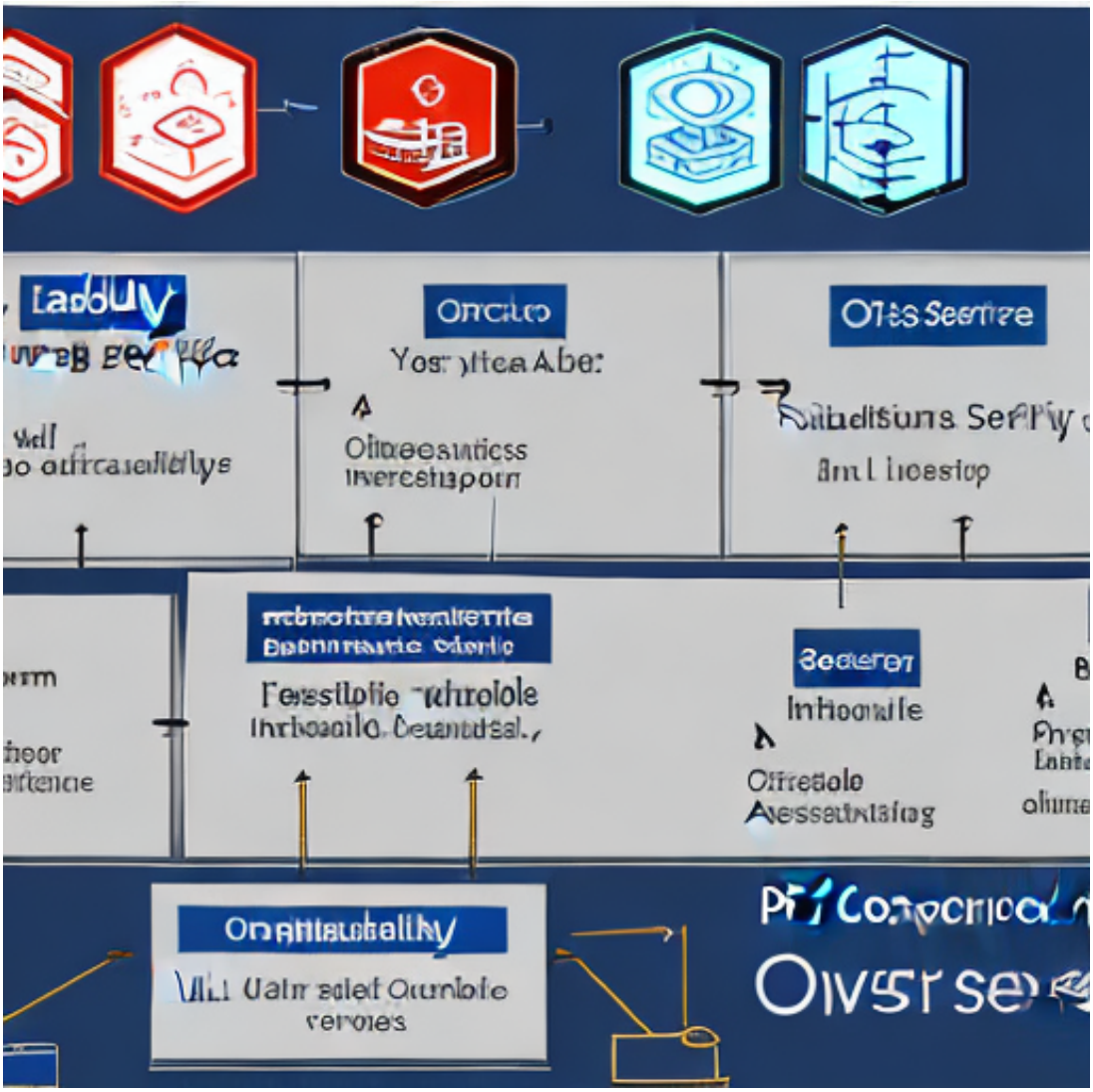

Figure 15. Vue d’ensemble synthétique de l’architecture

3. Étude théorique et analyse bibliographique

3.1 Fondements théoriques de l'automatisation des infrastructures

3.1.1 Évolution historique

La gestion des systèmes d'information a connu une évolution rapide, marquée par plusieurs transformations majeures. Elle est passée de l'administration manuelle des serveurs physiques, où chaque déploiement nécessitait des opérations répétitives et susceptibles d'erreurs, à l'émergence des datacenters virtualisés et du Cloud Computing. Cette progression a été motivée par la recherche d'une meilleure agilité opérationnelle et par la nécessité de réduire les coûts d'exploitation.

La complexification des environnements informatiques a conduit à la formalisation de pratiques visant à automatiser la création, la configuration et la supervision des ressources. C'est dans ce contexte qu'a émergé le paradigme de l'Infrastructure as Code, qui constitue aujourd'hui un socle incontournable des démarches de modernisation.

3.1.2 Approche DevOps

Le développement des infrastructures modernes s'inscrit dans une démarche DevOps, qui associe les équipes de développement et d'exploitation dans une collaboration continue. Cette approche vise à réduire les cycles de livraison, améliorer la qualité logicielle et favoriser l'automatisation des processus. Elle s'appuie sur une culture de responsabilisation partagée, une intégration continue et une surveillance permanente des systèmes. DevOps contribue ainsi à faire converger les objectifs techniques et organisationnels, en alignant la production logicielle et les opérations.

3.1.3 Modèles conceptuels

L'Infrastructure as Code (IaC) désigne l'ensemble des méthodes et outils permettant de décrire l'état souhaité d'une infrastructure sous forme de fichiers de configuration versionnés et exécutables. Deux modèles se distinguent généralement : le modèle impératif, dans lequel l'utilisateur décrit la séquence exacte des opérations, et le modèle déclaratif, qui se concentre sur la définition de l'état final visé en laissant au moteur d'exécution la responsabilité d'y converger.

3.1.4 Approche GitOps

En prolongement de l'IaC, l'approche GitOps propose de faire du système de gestion de version la source unique de vérité pour la configuration et le déploiement applicatif. Elle se caractérise par un processus de déploiement automatisé, piloté par des agents qui observent l'état déclaré dans les dépôts et appliquent les modifications nécessaires aux environnements. Cette méthode garantit une traçabilité complète des évolutions, facilite le retour en arrière et renforce la cohérence entre les différents environnements. GitOps s'intègre naturellement avec les pipelines CI/CD, qui orchestrent la construction, les tests et la mise en production de façon systématique.

3.2 Conteneurisation et orchestration

3.2.1 Principes de la conteneurisation

La conteneurisation a constitué une rupture technologique majeure en introduisant une isolation légère des environnements d'exécution, en comparaison avec les machines virtuelles traditionnelles. Chaque conteneur encapsule l'application et ses dépendances, assurant ainsi une portabilité élevée et une reproductibilité des exécutions sur différents environnements.

Parmi les bénéfices les plus fréquemment identifiés figurent la réduction de la charge système, l'optimisation des coûts d'infrastructure, l'accélération des cycles de déploiement et une meilleure isolation des processus.

3.2.2 Orchestration des conteneurs

Pour coordonner ces conteneurs à grande échelle, des plateformes d'orchestration ont été développées. Kubernetes s'est progressivement imposé comme la solution de référence, grâce à sa capacité à automatiser le placement des workloads, l'ajustement dynamique des ressources, la relance des conteneurs défaillants et la gestion centralisée des configurations ainsi que des secrets. Ces fonctionnalités favorisent l'exploitation efficace d'environnements complexes et distribués.

3.2.3 Patterns d'architecture cloud-native

La conteneurisation et l'orchestration encouragent l'adoption de modèles applicatifs dits *cloud-native*. Ces architectures reposent notamment sur le découpage en microservices, la scalabilité horizontale, la résilience par la redondance et le découplage entre l'infrastructure et les applications. Ces principes sont aujourd'hui largement adoptés par les entreprises souhaitant moderniser leurs systèmes d'information.

3.3 Approches de supervision et d'observabilité

3.3.1 Enjeux de l'observabilité

Dans des environnements distribués et dynamiques, l'observabilité est un facteur déterminant de fiabilité et de performance. Elle va au-delà de la supervision traditionnelle en visant une compréhension globale et en temps réel des comportements des systèmes. Cette démarche repose sur trois piliers essentiels : les métriques, qui mesurent l'état et les performances ; les logs, qui conservent l'historique des événements ; et les traces distribuées, qui permettent de suivre le parcours des requêtes au sein de l'architecture.

3.3.2 Outils et standards de référence

Différentes solutions se sont imposées comme standards de fait dans le domaine. Prometheus est souvent privilégié pour la collecte des métriques et la génération d'alertes, tandis que Grafana assure leur visualisation et leur suivi en temps réel. Elastic Stack ou Loki sont fréquemment utilisés pour l'agrégation et l'analyse des logs, et des outils tels que Jaeger et OpenTelemetry facilitent le traçage distribué. L'adoption de protocoles ouverts et d'API standardisées favorise leur intégration avec les plateformes d'orchestration.

3.4 Sécurité des infrastructures automatisées

3.4.1 Principes de sécurité

La sécurisation des infrastructures automatisées s'appuie sur le principe de *Security by Design*, qui préconise l'intégration des mesures de protection dès les phases initiales de conception. Le modèle *Zero Trust*, largement diffusé, repose notamment sur l'absence de confiance implicite accordée à un composant, l'authentification et l'autorisation systématiques de chaque requête ainsi que la limitation des privilèges au strict nécessaire. Ces approches sont particulièrement adaptées aux environnements hybrides et multi-clouds.

3.4.2 Gestion des secrets et des accès

La gestion centralisée des secrets constitue une pratique essentielle pour sécuriser les identifiants, certificats et autres éléments sensibles. Elle repose sur le stockage chiffré, la rotation périodique des clés et la traçabilité des accès. Des outils spécialisés, tels que Vault, apportent des solutions robustes et éprouvées pour répondre à ces exigences.

3.4.3 Sécurité périmétrique et segmentation

La protection des infrastructures repose également sur des dispositifs périmétriques tels que les pare-feu, les listes de contrôle d'accès et la segmentation réseau. Ces mécanismes permettent de

limiter la surface d'exposition, de cloisonner les environnements et de renforcer la résilience face aux attaques latérales. La mise en œuvre de politiques de filtrage strictes et le principe du moindre privilège complètent ces mesures pour réduire les risques d'intrusion.

3.4.4 Normes et standards applicables

La mise en conformité avec les référentiels internationaux contribue à renforcer la robustesse des systèmes et à répondre aux obligations réglementaires. Parmi les normes les plus mobilisées figurent la norme ISO/IEC 27001 relative à la sécurité de l'information, les recommandations sur le Zero Trust et les benchmarks publiés par le Center for Internet Security.

3.5 Comparaison des approches et des solutions existantes

3.5.1 Provisioning et configuration

Différents outils se distinguent dans le domaine de la gestion automatisée des infrastructures. Terraform et CloudFormation privilégient une approche déclarative du provisioning, facilitant la reproductibilité des déploiements, tandis qu'Ansible et Chef se concentrent sur la configuration et l'orchestration logicielle. La flexibilité de Terraform dans les environnements hybrides et multi-clouds, ainsi que la simplicité d'Ansible pour la configuration idempotente, sont régulièrement mises en avant.

3.5.2 Déploiement et synchronisation

Le déploiement et la synchronisation des configurations s'inscrivent dans les pratiques GitOps, qui renforcent la traçabilité et la cohérence entre les environnements. Argo CD se démarque par une interface graphique aboutie et une intégration poussée avec Kubernetes, tandis que Flux privilégie une approche minimaliste fondée sur des mécanismes de synchronisation automatisés. Ces outils contribuent à standardiser les processus de livraison et à réduire les écarts de configuration.

3.5.3 Limites identifiées

Malgré leurs atouts, ces approches présentent certaines contraintes, parmi lesquelles une complexité opérationnelle accrue, la nécessité d'une montée en compétences significative des équipes et des risques liés à la sécurité si les configurations ne sont pas maîtrisées. Ces limites appellent la définition de processus rigoureux et l'automatisation des contrôles.

4. Conception et automatisation de l'infrastructure

4.1 Introduction

4.1.1 Introduction

La conception et la mise en œuvre d'une infrastructure moderne reposent aujourd'hui sur l'automatisation complète des processus de création, de configuration et de sécurisation des environnements. Cette approche permet de garantir la cohérence, la traçabilité et la reproductibilité des déploiements tout en réduisant la complexité opérationnelle.

Ce chapitre présente la démarche adoptée pour concevoir et automatiser l'infrastructure du projet, en s'appuyant sur les principes de l'Infrastructure as Code et sur des outils spécialisés. Il expose dans un premier temps les technologies sélectionnées, telles que Proxmox, Terraform, Cloud-init, Ansible, Vault et Consul, qui constituent les fondations techniques de l'architecture. La suite du chapitre décrit la mise en œuvre progressive des différents composants : la gestion des secrets, la création automatisée des machines virtuelles, la préparation des inventaires et la configuration des services.

Enfin, une attention particulière est portée aux aspects réseau et à la sécurisation des flux, avec l'intégration de pare-feu, de mécanismes d'équilibrage de charge et de politiques de segmentation. Cette démarche globale vise à démontrer qu'il est possible de déployer une infrastructure complète et sécurisée de manière déclarative, tout en facilitant son évolutivité et sa maintenance.

4.2 Les outils utilisés pour l'infrastructure as code

4.2.1 Proxmox

Proxmox Virtual Environment (Proxmox VE) est une plateforme open source de virtualisation et de gestion d'infrastructure qui combine la virtualisation basée sur des machines virtuelles (KVM) et la conteneurisation légère (LXC) dans une interface unifiée. Elle offre une solution complète pour déployer et administrer des environnements virtualisés, qu'ils soient utilisés en laboratoire, en PME ou dans des centres de données. Proxmox se distingue par sa simplicité de mise en œuvre, sa richesse fonctionnelle et sa capacité à fédérer plusieurs nœuds dans un cluster haute disponibilité.

Proxmox répond à plusieurs enjeux stratégiques : rationalisation des ressources matérielles par la mutualisation, réduction des coûts grâce à une solution libre, amélioration de la flexibilité

opérationnelle et simplification de la gestion des infrastructures. Son interface web ergonomique permet de piloter l'ensemble des ressources, de planifier les sauvegardes et de superviser les performances.

D'un point de vue technique, Proxmox repose sur plusieurs composantes clés :

- **KVM (Kernel-based Virtual Machine)** : moteur de virtualisation complète.
- **LXC (Linux Containers)** : conteneurisation système légère.
- **Ceph Storage** : stockage distribué intégré et hautement disponible.
- **Cluster Management** : fédération et basculement automatique.
- **Interface Web et API REST** : administration centralisée.
- **Sauvegardes et snapshots** : gestion de la résilience.

Cas d'utilisation : déploiement d'un cluster de trois nœuds Proxmox avec stockage Ceph pour héberger des machines virtuelles critiques en haute disponibilité.

4.2.2 Terraform

Terraform est un outil open source d'Infrastructure as Code développé par HashiCorp. Il permet de définir et de provisionner des ressources complètes sous forme de code déclaratif, avec un modèle unifié pour divers fournisseurs (clouds publics, hyperviseurs privés).

Terraform répond à plusieurs enjeux : accélération du provisioning, fiabilisation des configurations et maîtrise des environnements. Il repose sur des concepts clés :

- **Fichiers de configuration (HCL)** : description de l'état souhaité.
- **Providers** : modules d'interface avec les APIs.
- **State file** : enregistrement des ressources créées.
- **Plan d'exécution** et **apply** : gestion des changements.
- **Modules et workspaces** : factorisation et isolation.

Cas d'utilisation : provisionner un cluster Kubernetes sur Proxmox avec un réseau et des volumes configurés.

4.2.3 Cloud-init

Cloud-init est un outil d'initialisation automatique des machines virtuelles lors du premier démarrage. Il est supporté par la plupart des plateformes cloud et hyperviseurs, et permet :

- La configuration réseau.
- La création d'utilisateurs et de clés SSH.
- L'installation de paquets et le lancement de scripts.

Cloud-init contribue à standardiser les environnements et réduire les délais de mise en service.

Cas d'utilisation : automatiser l'installation de Docker et la configuration réseau d'une instance Proxmox.

4.2.4 Ansible

Ansible est un outil open source d'automatisation et de configuration des systèmes, basé sur un langage déclaratif YAML et une architecture agentless (connexion SSH). Il permet de :

- Définir des playbooks réutilisables.
- Orchestrer la configuration de plusieurs hôtes.
- Gérer des inventaires et des variables.
- Assurer la traçabilité des opérations.

Cas d'utilisation : configurer des serveurs applicatifs, installer les dépendances et sécuriser les accès.

4.2.5 Vault

Vault est un gestionnaire de secrets développé par HashiCorp, conçu pour sécuriser, stocker et contrôler l'accès aux informations sensibles telles que les identifiants, les clés d'API ou les certificats. Il centralise le cycle de vie des secrets au sein d'un système unique et auditable, tout en offrant des capacités de distribution dynamique et de rotation automatique.

Principales fonctionnalités :

- Chiffrement systématique des données au repos et en transit.
- Génération dynamique de credentials à durée de vie limitée.
- Contrôle d'accès fin grâce à des politiques (ACL) granulaires.
- Rotation périodique et automatique des secrets.
- Audit complet des opérations et des accès.

Concepts de base Vault repose sur plusieurs concepts fondamentaux :

- **Seal et Unseal** : lors de son démarrage, Vault est scellé (sealed). Dans cet état, aucune donnée ne peut être lue ou écrite. Le processus d'unseal consiste à déverrouiller Vault en fournissant une combinaison de clés de déchiffrement (unseal keys) générées lors de l'initialisation. Ce mécanisme s'appuie sur l'algorithme de *Shamir's Secret Sharing*, qui divise la clé maîtresse en fragments distribués à différents opérateurs. Cela garantit qu'aucun individu ne détient seul la capacité de déverrouiller le système. La nécessité d'unseal manuellement après un redémarrage contribue à limiter l'exposition en cas de compromission physique ou de coupure d'alimentation.
- **Backends de stockage** : Vault peut s'appuyer sur différents backends de stockage (par exemple Consul, fichiers locaux, Amazon S3) pour persister ses données chiffrées. Le choix du backend influence les performances, la disponibilité et la tolérance aux pannes.
- **Secrets Engines** : ce sont les composants responsables de la gestion des secrets. Certains engines fournissent du stockage statique (par exemple les clés stockées manuellement), d'autres génèrent des secrets dynamiques à la demande (par exemple des credentials

temporaires pour une base de données).

- **Auth Methods** : ces méthodes définissent comment les utilisateurs ou applications s'authentifient auprès de Vault. Il peut s'agir de tokens statiques, d'authentification AppRole, de certificats TLS ou d'intégrations avec Kubernetes.
- **Policies** : les politiques d'accès contrôlent avec précision ce que chaque entité est autorisée à faire. Elles définissent les droits de lecture, d'écriture ou de suppression sur des chemins spécifiques.

Design de sécurité Vault adopte une approche de sécurité par défaut :

- Toutes les données sont chiffrées avec une clé maîtresse qui n'est jamais stockée en clair.
- Chaque requête est soumise à une évaluation des politiques d'accès.
- Les secrets peuvent être versionnés et révoqués à tout moment.
- Les logs d'audit permettent de tracer finement chaque opération.

Ce modèle de sécurité réduit le risque d'exposition accidentelle et renforce la conformité aux exigences réglementaires.

Exemple d'utilisation Vault est particulièrement adapté pour :

- Stocker et distribuer les tokens Kubernetes nécessaires à l'accès au cluster.
- Générer des credentials temporaires pour une base de données PostgreSQL.
- Chiffrer les certificats TLS utilisés par des applications.
- Mettre en place un PKI interne pour l'émission automatique de certificats.

Bonnes pratiques de sécurité Pour garantir un niveau de protection élevé, il est recommandé de :

- Distribuer les clés de scellage (unseal keys) à plusieurs opérateurs de confiance afin de prévenir tout point de défaillance individuel.
- Configurer une rotation périodique des clés maîtresses.
- Limiter strictement l'accès aux méthodes d'authentification les plus sensibles.
- Activer et surveiller systématiquement les logs d'audit.
- S'assurer que le backend de stockage est hautement disponible et correctement sécurisé.

Limitations et vigilance Si Vault n'est pas configuré de manière rigoureuse, il peut devenir un point unique de défaillance et un vecteur d'attaque majeur. La compromission d'un nœud ou la perte des clés de déchiffrement peut entraîner une indisponibilité prolongée, voire la perte irrémédiable de certains secrets. Il est donc essentiel d'envisager des scénarios de récupération, tels que la sauvegarde chiffrée des données et la documentation des procédures d'unseal.

4.2.6 Consul

Consul est une solution de service discovery, de configuration distribuée et de service mesh développée par HashiCorp. Elle permet de centraliser la gestion dynamique des services et de renforcer la cohérence des environnements déployés. Consul apporte plusieurs fonctionnalités complémentaires à Vault et s'intègre naturellement dans les architectures automatisées.

Principales fonctionnalités Les principales capacités de Consul sont les suivantes :

- **Découverte automatique des services** : Consul maintient un catalogue des services enregistrés, avec leurs adresses et ports associés, facilitant leur détection et leur utilisation par d'autres composants.
- **Supervision de l'état des services** : le système exécute des contrôles de santé (health checks) qui permettent de vérifier en continu la disponibilité et le bon fonctionnement des instances.
- **Stockage clé-valeur** : Consul fournit un KV store distribué, utilisé pour stocker des paramètres de configuration ou des informations dynamiques accessibles par les applications.
- **Service mesh sécurisé** : grâce à l'intégration avec Envoy, Consul offre un plan de contrôle permettant de chiffrer le trafic inter-services, d'implémenter des politiques d'accès et de réaliser du routage avancé.

Architecture et composants Consul repose sur une architecture distribuée, composée de plusieurs éléments :

- **Agents** : chaque nœud du cluster exécute un agent Consul, qui peut être configuré en mode client ou serveur.
- **Serveurs** : les serveurs Consul assurent la cohérence des données à l'aide d'un consensus basé sur le protocole Raft.
- **Catalogue de services** : une base de données en temps réel contenant les informations sur tous les services enregistrés et leur état de santé.
- **KV Store** : un espace de stockage distribué, utilisé pour partager des données de configuration.
- **Connect** : le composant de service mesh qui gère l'émission de certificats TLS, le chiffrement du trafic et le contrôle des autorisations.

Mécanismes et cas d'usage Consul propose plusieurs mécanismes qui répondent à des besoins spécifiques :

- *Service discovery* : lorsqu'un service s'enregistre auprès de Consul, il devient automatiquement visible pour les clients qui peuvent interroger l'API DNS ou HTTP afin de récupérer ses coordonnées.

- *Health checks* : chaque enregistrement de service peut être associé à un ou plusieurs tests de disponibilité. Si un service échoue aux contrôles, il est retiré du catalogue actif.
- *KV Store* : les applications peuvent récupérer dynamiquement des paramètres ou configurations partagées (par exemple des variables d'environnement ou des endpoints).
- *Consul Template* : cet outil surveille le KV store et les catalogues, puis génère automatiquement des fichiers de configuration lorsque des changements surviennent.
- *Service mesh* : Consul Connect émet des certificats mTLS, configure les proxys Envoy et applique des politiques d'accès qui définissent quels services sont autorisés à communiquer.

Exemple d'utilisation : synchroniser dynamiquement les configurations applicatives avec Consul Template, mettre à jour les fichiers de configuration NGINX lorsque de nouveaux services sont enregistrés, ou établir un canal de communication chiffré entre des microservices.

Bonnes pratiques de sécurité Pour tirer parti des fonctionnalités de Consul tout en maintenant un niveau élevé de sécurité, il est recommandé de :

- Activer le chiffrement TLS pour toutes les communications inter-nœuds.
- Protéger l'interface HTTP API par une ACL stricte.
- Mettre en place des politiques de token management et de rotation régulière des secrets.
- Limiter l'accès aux agents et serveurs Consul au sein d'un réseau privé.
- Activer l'audit des requêtes et opérations critiques.

Limitations et vigilance De meme que vault , malgré sa richesse fonctionnelle, Consul peut devenir un point unique de coordination : une indisponibilité du cluster impacte la découverte des services et la cohérence des configurations. De plus, une configuration incorrecte des ACL ou des intentions (politiques de service mesh) peut entraîner des fuites d'informations ou des interruptions de service. Il est donc essentiel de soigner le dimensionnement du cluster, de planifier des sauvegardes régulières et de tester les scénarios de bascule.

4.3 Mise en place des concepts de l'infrastructure en code

La mise en œuvre d'une infrastructure déclarative repose sur une combinaison d'outils spécialisés qui permettent d'automatiser l'ensemble du cycle de vie des environnements techniques. Cette approche favorise la cohérence, la traçabilité et la reproductibilité des déploiements. Les sections suivantes décrivent les principales étapes mises en place dans le projet.

4.3.1 Préparation des secrets avec Vault

Dans une architecture moderne, la gestion sécurisée des informations sensibles (mots de passe, clés d'API, certificats) est un enjeu majeur. Pour répondre à cette problématique, l'outil HashiCorp Vault a été utilisé comme coffre-fort centralisé.

Vault a été configuré en mode *server* avec un stockage interne et une politique de chiffrement des données au repos. Les secrets sont créés et organisés dans des chemins logiques (`secret/`, `kv/`) permettant de les isoler par projet ou par environnement (développement, production).

L'accès à Vault est contrôlé par des policies granulaires, et l'authentification des outils d'automatisation (Terraform, Ansible) s'effectue via des tokens dynamiques ou l'approche AppRole. Cette centralisation simplifie la rotation des secrets et réduit les risques liés aux configurations manuelles.

4.3.2 Création de templates de machines virtuelles

Afin de garantir l'uniformité des systèmes de base, des templates de machines virtuelles ont été préparés sur Proxmox. Ces templates incluent :

- Le système d'exploitation minimal (par exemple Debian ou Ubuntu LTS).
- Les mises à jour de sécurité appliquées.
- Les dépendances de base (cloud-init, agents QEMU, agents SPICE et agents).
- La configuration des clés SSH nécessaires à l'automatisation.

L'utilisation de ces modèles préconfigurés permet de créer rapidement de nouvelles instances sans avoir à répéter les étapes de préparation initiale. Cette approche contribue à standardiser le socle technique et à réduire le temps de provisionnement.

4.3.3 Création des machines virtuelles à travers Terraform

La création automatisée des machines virtuelles est orchestrée par Terraform. Les ressources sont décrites sous forme de fichiers HCL (*HashiCorp Configuration Language*) qui précisent :

- La taille et le nombre de vCPU et de mémoire.
- Le réseau et les interfaces associées.
- Le disque principal et son format.
- Le template de base à cloner.

L'exécution de `terraform apply` permet de matérialiser l'infrastructure déclarée. Cette étape est également responsable de l'injection initiale des métadonnées (par exemple, le nom de l'instance, l'identifiant d'environnement). Grâce à Terraform, la création des VMs est reproductible et contrôlée dans le temps.

4.3.4 Préparation automatique des inventaires

Après le provisionnement des ressources, la préparation des inventaires est essentielle pour permettre à Ansible de prendre le relais. Pour ce faire, un script d'automatisation collecte dynamiquement les informations des machines créées (adresses IP, identifiants, groupes logiques) et génère un inventaire au format YAML compatible avec Ansible.

Cette génération automatisée évite les erreurs liées aux manipulations manuelles et garantit que

l'inventaire reflète toujours l'état réel de l'infrastructure. L'inventaire est versionné dans un dépôt Git, renforçant la traçabilité des évolutions.

4.3.5 Configuration automatique des machines virtuelles avec Ansible

Une fois les inventaires préparés, Ansible est utilisé pour configurer les machines virtuelles de manière déclarative. Les rôles et playbooks appliquent notamment :

- La création des utilisateurs et des groupes.
- La configuration du pare-feu et des règles de sécurité.
- L'installation des paquets requis.
- Le déploiement des configurations d'applications et des services système.

L'exécution d'Ansible est idempotente, garantissant que les machines convergent toujours vers l'état souhaité, quelle que soit leur configuration initiale. Les variables sensibles sont injectées de manière sécurisée via Vault.

4.4 Outils de réseau, exposition des services et sécurité

Le bon fonctionnement d'une infrastructure passe par une gestion rigoureuse du réseau et une exposition contrôlée des services. Dans le projet, plusieurs composants et bonnes pratiques ont été mis en œuvre :

- **Réseau virtuel et segmentation** : les machines sont placées dans des VLANs distincts afin de séparer les environnements (production, développement) et de limiter les flux inter-zones.
- **Reverse proxy et ingress** : l'exposition des services HTTP(S) s'effectue via des ingress controllers Kubernetes ou des reverse proxies tels que NGINX. Ces composants assurent le routage, la terminaison TLS et l'équilibrage de charge.
- **Certificats TLS** : les certificats sont gérés automatiquement grâce à l'intégration de Let's Encrypt ou Vault PKI, garantissant la sécurité des échanges.
- **Pare-feu et contrôle des accès** : des règles strictes sont appliquées sur les hôtes et les pods, combinant iptables, security groups et Network Policies Kubernetes.
- **Monitoring et logs** : la collecte centralisée des logs réseau et la supervision des connexions permettent d'anticiper les incidents et de renforcer la sécurité.

Cette approche cohérente assure une exposition minimale des services au public et une sécurité renforcée tout en maintenant une haute disponibilité.

4.4.1 pfSense

pfSense est une solution open source de pare-feu et de routage. Elle permet :

- La définition de règles de filtrage réseau.

- La gestion de VPN et la segmentation des VLAN.
- La supervision du trafic.

4.4.2 Réseau de Kubernetes

Le réseau Kubernetes est un élément fondamental qui permet la communication entre les différents composants du cluster et les applications qui y sont déployées. Il repose sur plusieurs concepts clés visant à simplifier la connectivité et à assurer l'isolation logique des workloads.

Modèle de réseau plat Kubernetes adopte un modèle de réseau dit *plat*, dans lequel :

- Chaque **pod** reçoit une adresse IP unique.
- Tous les pods peuvent communiquer entre eux, sans traduction d'adresses (NAT).
- Les pods peuvent accéder aux services exposés par d'autres pods, quel que soit le nœud sur lequel ils s'exécutent.

Ce modèle vise à réduire la complexité des communications et à permettre aux applications de se comporter comme si elles fonctionnaient sur un même réseau local.

CNI (Container Network Interface) Pour mettre en œuvre le modèle réseau, Kubernetes s'appuie sur des plugins CNI (Container Network Interface). Ces plugins sont responsables de :

- L'attribution des adresses IP aux pods.
- La configuration des routes réseau.
- L'application des règles de filtrage ou d'isolation.

Parmi les solutions CNI les plus courantes, on peut citer Calico, Flannel, Cilium et Weave.

Services et ClusterIP Kubernetes introduit l'objet `Service` qui permet d'exposer un groupe de pods sous une adresse IP virtuelle et un nom DNS stable. Le type `ClusterIP` crée un point d'accès interne accessible uniquement depuis le cluster. Le routage vers les pods est assuré par le kube-proxy, qui configure des règles iptables ou IPVS selon le mode choisi.

Services NodePort et LoadBalancer Pour exposer un service à l'extérieur du cluster :

- Le type `NodePort` alloue un port TCP/UDP sur chaque nœud du cluster.
- Le type `LoadBalancer` s'intègre avec un équilibrage de charge externe (cloud provider) afin de disposer d'une IP publique.

Ces mécanismes simplifient l'accès aux applications depuis l'extérieur.

Ingress et contrôleurs Ingress L'`Ingress` est une ressource Kubernetes qui permet de définir des règles de routage HTTP(S) plus avancées, par exemple :

- Routage par nom de domaine.
- Terminaison TLS.
- Redirections et règles de sécurité.

Un contrôleur Ingress (tel que NGINX Ingress Controller ou Traefik) est déployé pour interpréter et appliquer ces règles.

Network Policies Pour renforcer la sécurité, Kubernetes propose les `NetworkPolicies`, qui définissent des règles de filtrage des flux entre pods :

- Sélection des pods sources et destinations.
- Protocoles et ports autorisés.
- Isolation stricte par namespace ou par label.

Les politiques réseau nécessitent un CNI compatible (par exemple Calico).

DNS interne Kubernetes fournit un service DNS interne (CoreDNS) qui résout les noms des services et pods :

- Chaque service est accessible via un nom DNS du type `myservice.mynamespace.svc.cluste`
- Les applications peuvent utiliser la découverte de services sans configuration externe.

En combinant ces composants, le réseau Kubernetes offre un modèle cohérent, flexible et extensible qui facilite le déploiement d'applications distribuées tout en garantissant la sécurité et l'évolutivité.

4.4.3 MetalLB

MetalLB est une solution de load balancing spécialement conçue pour les clusters Kubernetes déployés en environnement on-premise. Contrairement aux plateformes cloud qui fournissent des services d'équilibrage de charge natifs, les installations locales de Kubernetes ne disposent pas par défaut d'un mécanisme équivalent pour l'attribution d'adresses IP publiques et la distribution du trafic.

Principales fonctionnalités MetalLB apporte plusieurs fonctionnalités essentielles :

- **Attribution d'adresses IP virtuelles** : il permet de réserver et d'annoncer des plages d'adresses IP utilisables pour exposer les services en mode *LoadBalancer*.
- **Distribution du trafic réseau** : il redirige les requêtes entrantes vers les pods cibles, en fonction de la configuration et de l'algorithme de balancing choisi.
- **Compatibilité BGP et L2** : MetalLB propose deux modes de fonctionnement : le mode Layer 2, qui utilise ARP ou NDP pour annoncer l'adresse IP sur le réseau local, et le mode BGP, qui permet d'établir des sessions de routage dynamique avec les routeurs de l'infrastructure.

Intérêt dans un contexte on-premise En environnement on-premise, Kubernetes présente une limite importante : le type de service *LoadBalancer* est inutilisable par défaut, car il dépend d'un fournisseur d'infrastructure qui alloue et configure automatiquement l'équilibrage de charge. MetalLB comble cette lacune en reproduisant ce comportement de manière logicielle et en s'intégrant nativement avec l'API Kubernetes. Il permet ainsi d'exposer des services applicatifs sans recourir à des appliances réseau propriétaires ni modifier profondément l'architecture existante.

Cas d'utilisation MetalLB est couramment utilisé pour :

- Exposer en haute disponibilité des applications web, API ou services internes hébergés dans Kubernetes.
- Fournir des adresses IP stables et connues pour les clients et systèmes externes.
- Automatiser la gestion des flux réseau entrants tout en conservant un contrôle fin sur les plages d'adresses attribuées.

4.5 Mise en place des services de réseau

Cette partie décrit la mise en œuvre et l'automatisation des services essentiels au bon fonctionnement du réseau et à la sécurisation des flux. Ces services incluent le pare-feu, le reverse proxy ainsi que la gestion centralisée des secrets.

4.5.1 Configuration automatique de pfSense avec Ansible

Le pare-feu pfSense constitue le point d'entrée et de filtrage du trafic réseau. Afin de garantir la reproductibilité de sa configuration et d'éviter les erreurs manuelles, Ansible a été utilisé pour automatiser son déploiement et sa configuration.

Pour cela, des modules spécifiques à pfSense et des collections Ansible dédiées ont été mis en œuvre, permettant notamment :

- La définition des règles de filtrage (firewall rules) par groupe et par interface.
- La configuration des interfaces réseau et des VLANs.
- L'activation et la configuration du service DHCP.
- La gestion des utilisateurs et des certificats.

Grâce à cette approche, il est possible de versionner les configurations pfSense dans un dépôt Git et de les appliquer de manière cohérente sur plusieurs environnements. En cas de restauration après sinistre, la remise en service peut ainsi être réalisée rapidement et de façon fiable.

4.5.2 Configuration de NGINX avec Ansible

NGINX joue un rôle central comme reverse proxy et point d'entrée HTTP(S) des applications. La configuration de NGINX a été automatisée avec Ansible afin de :

- Créer et gérer les fichiers de configuration des sites virtuels (server blocks).
- Générer automatiquement les certificats TLS via Let's Encrypt.
- Mettre en place les règles de redirection et de réécriture des URLs.
- Définir les paramètres de sécurité (headers HTTP, limitation de débit).

Les rôles Ansible développés permettent de paramétrer NGINX de manière déclarative en fonction des variables d'inventaire et de secrets provenant de Vault. Chaque modification de configuration est ainsi versionnée et peut être appliquée de façon idempotente.

4.5.3 Usage de Vault pour la gestion des secrets

La gestion centralisée et sécurisée des secrets est assurée par HashiCorp Vault. Vault est utilisé comme source unique de vérité pour stocker et distribuer :

- Les mots de passe d'accès aux services.
- Les clés API nécessaires aux applications.
- Les certificats TLS privés.
- Les tokens d'authentification.

L'authentification des systèmes à Vault s'effectue via AppRole et tokens dynamiques, ce qui limite le risque de compromission en cas de fuite d'identifiants. Ansible est configuré pour interroger Vault à l'exécution des playbooks et récupérer les secrets de manière transparente. Cette approche présente plusieurs avantages :

- Les secrets ne sont jamais stockés en clair dans les dépôts Git.
- La rotation régulière est facilitée.
- Les accès sont tracés et audités.

L'intégration de Vault avec Terraform et Ansible permet ainsi de garantir un niveau de sécurité élevé tout au long du cycle de vie des environnements.

4.6 Synthèse

La combinaison d'outils tels que Proxmox, Terraform, Ansible, Vault et pfSense a permis de construire une infrastructure automatisée, sécurisée et reproductible. Cette approche s'inscrit dans la démarche Infrastructure as Code, garantissant un haut niveau de cohérence et facilitant les évolutions futures.

5. Mise en œuvre du modèle GitOps

5.1 Presentation des outils GitOps

5.1.1 Argo CD

Argo CD est un outil open source de déploiement continu (CD) natif Kubernetes, conçu pour mettre en œuvre les pratiques GitOps. Il permet de synchroniser l'état désiré des applications, défini dans un dépôt Git, avec l'état effectif du cluster Kubernetes. En automatisant la gestion et le déploiement des manifestes, Argo CD apporte cohérence, traçabilité et résilience aux environnements cloud-native.

Argo CD répond à plusieurs enjeux stratégiques : fiabiliser les déploiements, réduire le temps de mise en production, renforcer la traçabilité et limiter les erreurs humaines. Il offre un modèle déclaratif et auditable, conforme aux exigences de sécurité et de conformité des organisations modernes. En industrialisant le GitOps, Argo CD contribue à accélérer l'innovation tout en garantissant la stabilité des systèmes.

, Argo CD s'appuie sur plusieurs composants clés :

- **Le dépôt Git** : source unique de vérité contenant les manifestes Kubernetes (YAML) ou les définitions Kustomize/Helm.
- **Le contrôleur Argo CD** : composant qui surveille les différences entre l'état souhaité (Git) et l'état réel du cluster.
- **L'API Server et l'interface Web** : couche d'administration et de visualisation centralisée des applications et des synchronisations.
- **Les applications** : objets Kubernetes représentant l'état désiré d'un ensemble de ressources.
- **Les stratégies de synchronisation** : modes automatique ou manuel permettant de contrôler les mises à jour.

Argo CD offre un modèle de sécurité avancé, intégrant la gestion fine des permissions (RBAC), le support du SSO (OAuth2, OIDC), le chiffrement des secrets et des validations automatiques des changements.

Exemples et cas d'usage :

- Déployer automatiquement une application Helm versionnée depuis un dépôt Git centralisé.
- Gérer des environnements multiples (dev, staging, production) avec des dossiers ou des branches distinctes.

- Appliquer des politiques de synchronisation automatique avec validation de signature Git.
- Visualiser les différences entre l'état courant et l'état cible et lancer un déploiement manuel.
- Auditer l'historique des déploiements et des changements appliqués au cluster.

Avantages principaux :

- Mise en œuvre native du GitOps et centralisation de la configuration déclarative.
- Traçabilité et auditabilité complètes des changements.
- Intégration fluide avec Helm, Kustomize, Jsonnet et plain YAML.
- Réduction du risque d'erreurs grâce au contrôle automatique des dérives d'état.
- Interface Web ergonomique et API REST.
- Sécurité renforcée avec RBAC et chiffrement des secrets.

En synthèse, Argo CD est une solution stratégique pour l'automatisation et la fiabilisation des déploiements Kubernetes. Il contribue à instaurer des workflows GitOps robustes, cohérents et évolutifs, adaptés aux exigences opérationnelles des entreprises modernes.

Références suggérées :

- Argo CD Documentation – `https://argo-cd.readthedocs.io/`
- Argo CD GitHub Repository – `https://github.com/argoproj/argo-cd`
- GitOps Principles – `https://www.gitops.tech/`
- CNCF Argo Project – `https://www.cncf.io/projects/argo/`
- Helm Documentation – `https://helm.sh/docs/`

5.1.2 Helm

Helm est un gestionnaire de packages Kubernetes qui permet de décrire des applications sous forme de charts. Chaque chart contient des modèles de manifestes et des fichiers de valeurs qui définissent les paramètres de l'application. Helm est particulièrement adapté lorsque :

- Les applications déployées nécessitent de nombreuses options configurables.
- Il est souhaité de réutiliser des packages officiels ou communautaires (par exemple nginx, prometheus).
- Les équipes ont besoin de versionner et de publier des applications sous forme de packages standardisés.

Exemples de cas d'usage :

- Déploiement d'un cluster Prometheus avec des paramètres personnalisés.
- Installation d'un Ingress Controller en adaptant les valeurs selon l'environnement.

Exécution : Lorsqu'Argo CD synchronise une application déclarée comme Helm dans le dépôt Git, il exécute le rendu du chart (commande helm template) avant d'appliquer les ressources générées dans le cluster.

5.1.3 Kustomize

Kustomize est un outil natif Kubernetes de composition et de personnalisation des manifestes YAML. Il fonctionne en surchargeant des ressources de base avec des patches et des overlays. Kustomize est adapté lorsque :

- L'application ne nécessite pas un système de packaging complet.
- Les environnements (dev, recette, prod) partagent une base commune mais nécessitent des ajustements ciblés.
- Il est important de conserver des manifestes Kubernetes purs et lisibles.

Exemples de cas d'usage :

- Appliquer des replicas différents selon l'environnement.
- Modifier les variables d'environnement ou les annotations.

Exécution : Argo CD supporte nativement Kustomize. Lors de la synchronisation, le contrôleur applique kustomize build pour générer les manifestes avant de les appliquer au cluster.

Intégration avec Argo CD Argo CD offre un support natif pour Helm et Kustomize. Lors de la définition d'une application, il suffit de préciser le type de source (Helm, Kustomize ou Directory). Le processus de rendu est alors entièrement automatisé :

- Le contrôleur Argo CD surveille le dépôt Git et détecte les modifications.
- Il exécute le rendu des manifestes via Helm ou Kustomize.
- Les ressources générées sont comparées à l'état courant du cluster.
- Les différences sont appliquées automatiquement ou manuellement selon la stratégie choisie.

Bénéfices principaux Le recours à Helm et Kustomize apporte plusieurs avantages :

- Réduction de la duplication des manifestes entre environnements.
- Simplification de la gestion des configurations dynamiques.
- Meilleure lisibilité et maintenabilité des définitions Kubernetes.
- Standardisation des déploiements grâce aux charts officiels ou internes.

5.2 Mise en œuvre du modèle GitOps

Le modèle GitOps vise à centraliser la définition de l'infrastructure et des applications dans des dépôts Git versionnés, en s'appuyant sur un opérateur qui applique automatiquement l'état souhaité dans le cluster Kubernetes. Dans ce projet, l'outil **Argo CD** a été retenu pour assurer ce rôle. La démarche GitOps permet d'améliorer la traçabilité, la cohérence et l'automatisation des déploiements.

5.2.1 Préparation des manifestes des outils internes

Avant l'installation d'Argo CD, les manifestes Kubernetes décrivant les composants internes nécessaires au bon fonctionnement de la plateforme ont été préparés. Ces manifestes incluent :

- Les configurations des namespaces réservés (par exemple `argocd`, `monitoring`, `tools`).
- Les déploiements de services annexes tels que les opérateurs de sauvegarde et les contrôleurs réseau.
- Les configurations des ressources communes (ConfigMaps, Secrets, RBAC).

L'ensemble de ces manifestes est versionné dans un dépôt Git dédié à l'infrastructure, garantissant une source unique de vérité et la possibilité de reconstruire intégralement l'environnement.

5.2.2 Installation d'Argo CD

L'installation d'Argo CD a été réalisée via l'application des manifestes officiels fournis par le projet. Le processus s'effectue en deux étapes principales :

- Création du namespace dédié (`argocd`).
- Application du manifest d'installation complet :

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argopr
```

Après l'installation, les pods principaux (API server, repo server, application controller et dex) sont déployés automatiquement. L'interface web d'Argo CD permet de superviser les applications GitOps et leur état de synchronisation.

5.2.3 Configuration de l'authentification

La sécurisation de l'accès à Argo CD est essentielle. Les mesures suivantes ont été mises en place :

- Activation de l'authentification via Dex avec un connecteur LDAP, permettant une intégration avec l'annuaire interne.
- Création de rôles et de policies RBAC pour définir des droits différenciés selon les équipes (lecture seule, modification, administration).
- Rotation automatique des tokens d'accès.
- Activation de TLS pour sécuriser les communications avec l'interface web.

Ces mécanismes garantissent que seuls les utilisateurs autorisés peuvent interagir avec les ressources et déclencher des déploiements.

5.2.4 Configuration des synchronisations

La synchronisation automatique entre l'état déclaré dans Git et l'état réel du cluster est un principe fondamental de GitOps. Argo CD a été configuré avec les paramètres suivants :

- Mode de synchronisation automatique (`auto-sync`) activé sur les applications critiques.
- Validation stricte des manifests avant application.
- Prise en charge des stratégies de *pruning* pour supprimer les ressources obsolètes.
- Notification par webhook et alerting en cas d'écart détecté entre l'état souhaité et l'état courant.

Cette configuration permet de garantir que le cluster converge toujours vers l'état décrit dans les dépôts Git et de détecter les modifications manuelles non autorisées.

5.2.5 Préparation des manifestes des applications développées par Oneex pour des environnements différents

Les applications développées par Oneex ont été déployées sur plusieurs environnements (développement, recette, production). Pour assurer la cohérence et l'adaptabilité, les manifestes Kubernetes ont été préparés selon les principes suivants :

- Utilisation de `kustomize` pour générer des variantes par environnement (par exemple configuration des replicas, des ressources et des variables d'environnement).
- Définition de ConfigMaps et de Secrets séparés selon les contextes.
- Structuration des dépôts Git avec des arborescences claires par application et par environnement.
- Mise en place de règles de validation continue (linting et contrôle de schéma) avant validation des commits.

Cette approche permet de disposer d'un processus de déploiement uniforme, de simplifier la maintenance et de garantir que chaque environnement est conforme aux spécifications attendues.

6. Intégration et livraison continues

6.1 les outils utilisés

6.1.1 GitLab CI

GitLab CI (Continuous Integration) est un système d'intégration et de livraison continues intégré nativement dans GitLab, une plateforme DevOps complète de gestion du cycle de vie applicatif. Il permet d'automatiser la construction, les tests, la validation, le packaging et le déploiement des applications en s'appuyant sur des pipelines définis de manière déclarative. GitLab CI est aujourd'hui largement utilisé dans les organisations souhaitant industrialiser leurs workflows de développement et renforcer la qualité logicielle.

GitLab CI répond à plusieurs enjeux stratégiques : accélérer le time-to-market, réduire les erreurs humaines, renforcer la traçabilité des changements et améliorer la collaboration entre équipes. Grâce à sa proximité avec le dépôt Git, il apporte une cohérence totale entre le code source, l'historique des commits et les pipelines d'automatisation. Il contribue ainsi à la modernisation et à la professionnalisation des processus de développement.

, GitLab CI repose sur plusieurs concepts essentiels :

- **Le fichier `.gitlab-ci.yml`** : fichier de configuration déclaratif placé à la racine du dépôt, qui décrit les jobs et les étapes du pipeline.
- **Les jobs** : unités atomiques qui exécutent des scripts ou des commandes (build, test, deploy).
- **Les stages** : regroupements logiques des jobs (par exemple, build, test, deploy) exécutés séquentiellement ou en parallèle.
- **Les runners** : exécutants (machines ou conteneurs) qui traitent les jobs. Ils peuvent être partagés, spécifiques ou autoscalés.
- **Les variables** : valeurs dynamiques injectées dans les pipelines (clés, secrets, paramètres d'environnement).
- **Les artefacts** : fichiers générés par les jobs et transmis entre étapes.

GitLab CI prend en charge de nombreuses fonctionnalités avancées : intégration Kubernetes, déclencheurs manuels (manual actions), pipelines multi-projets, stratégies de déploiement progressif et vérification des politiques de sécurité.

Exemples et cas d'usage :

- Compiler automatiquement une application dès la création d'une merge request.
- Exécuter des tests unitaires et fonctionnels dans un pipeline parallèle.

- Construire et publier des images Docker sur GitLab Container Registry.
- Déployer des applications sur Kubernetes via Helm ou kubectl.
- Générer et publier automatiquement la documentation technique.

Avantages principaux :

- Intégration native avec GitLab et l'ensemble du cycle de vie DevOps.
- Modèle déclaratif simple et lisible.
- Traçabilité et auditabilité complète des pipelines.
- Compatibilité avec les conteneurs et les environnements Kubernetes.
- Gestion sécurisée des secrets et des variables sensibles.
- Large écosystème de templates, exemples et intégrations communautaires.

En synthèse, GitLab CI est une solution stratégique pour automatiser l'intégration et la livraison continues. Il permet aux équipes de gagner en efficacité opérationnelle, d'améliorer la qualité logicielle et d'accélérer la mise en production des innovations.

Références suggérées :

- GitLab CI Documentation – `https://docs.gitlab.com/ee/ci/`
- GitLab CI YAML Reference – `https://docs.gitlab.com/ee/ci/yaml/`
- GitLab Runners – `https://docs.gitlab.com/runner/`
- GitLab Kubernetes Integration – `https://docs.gitlab.com/ee/user/project/clusters/`
- GitLab Auto DevOps – `https://docs.gitlab.com/ee/topics/autodevops/`

6.1.2 Commitlint

Commitlint est un outil open source qui permet de vérifier que les messages de commit respectent un format prédéfini. Il est particulièrement utilisé dans les workflows Git modernes pour renforcer la cohérence des messages de commit, faciliter la génération automatique de changelogs et standardiser la documentation des évolutions logicielles. Commitlint est souvent intégré à des processus de validation automatisés grâce aux hooks Git (par exemple avec Husky) ou aux pipelines CI/CD.

Commitlint répond à plusieurs enjeux stratégiques : améliorer la lisibilité de l'historique des changements, garantir une traçabilité complète des évolutions, renforcer la qualité documentaire et faciliter les audits. La standardisation des messages de commit contribue à instaurer une culture de rigueur et de professionnalisation au sein des équipes de développement.

, Commitlint s'appuie sur plusieurs concepts essentiels :

- **Les règles de validation** : définissent le format attendu des commits (par exemple, le standard Conventional Commits).
- **Le parser** : analyse le message de commit et vérifie qu'il correspond au schéma spécifié.
- **La configuration** : fichier `commitlint.config.js` où l'on définit les règles, les

exceptions et les presets.

- **L'intégration avec Husky** : permet de déclencher la vérification lors du hook `commit-msg`.

Le standard le plus répandu est **Conventional Commits**, qui impose un format structuré :

```
<type>(<scope>): <subject>

Exemple :
feat(auth): add JWT authentication
fix(api): handle null pointer exception
docs(readme): update installation instructions
```

Ce format facilite l'automatisation des versions sémantiques (Semantic Versioning) et la génération des changelogs.

Exemples et cas d'usage :

- Empêcher la validation d'un commit si le message ne commence pas par un type valide (ex.: feat, fix, chore).
- Bloquer les commits dont le titre dépasse une longueur maximale.
- Valider automatiquement tous les messages de commit dans un pipeline CI/CD.
- Générer des changelogs structurés à partir des commits normalisés.
- Appliquer un format de commit homogène sur plusieurs équipes et projets.

Avantages principaux :

- Standardisation et lisibilité accrue des messages de commit.
- Réduction des erreurs et des incohérences documentaires.
- Automatisation des processus de release et de génération de changelogs.
- Compatibilité avec les pratiques GitOps et CI/CD.
- Facilité d'intégration avec Husky et d'autres outils de hooks Git.

En synthèse, Commitlint est une brique essentielle pour industrialiser et professionnaliser la gestion des versions et la documentation des projets logiciels. Il contribue à instaurer une culture DevOps rigoureuse et à améliorer la traçabilité du cycle de développement.

Références suggérées :

- Commitlint Documentation – `https://commitlint.js.org/`
- Conventional Commits – `https://www.conventionalcommits.org/`
- Husky Documentation – `https://typicode.github.io/husky/`
- Semantic Versioning – `https://semver.org/`
- GitHub Commitlint Repository – `https://github.com/conventional-changelog/commitlint`

6.1.3 Husky

Husky est un outil open source permettant de gérer et d'exécuter des hooks Git de manière simple et centralisée dans les projets logiciels. Il facilite l'automatisation de tâches de validation et de mise en conformité lors des événements Git, tels que les commits, les pushs ou les merges. Grâce à sa configuration déclarative, Husky contribue à instaurer des pratiques DevOps rigoureuses et à renforcer la qualité du code tout au long du cycle de développement.

Husky répond à plusieurs enjeux stratégiques : réduire les erreurs humaines, homogénéiser les workflows entre équipes, accélérer le feedback lors des validations et améliorer la traçabilité des changements. Il constitue un levier essentiel de professionnalisation, car il garantit que les standards de qualité (tests, linting, conventions de commit) sont systématiquement respectés avant d'intégrer le code au référentiel principal.

, Husky repose sur plusieurs concepts clés :

- **Les hooks Git** : scripts déclenchés automatiquement par Git à différents moments du cycle de vie (par exemple `pre-commit`, `commit-msg`, `pre-push`).
- **La configuration** : Husky utilise des commandes déclarées dans le fichier `package.json` ou dans des fichiers dédiés (`.husky/pre-commit`).
- **Les intégrations** : Husky fonctionne avec de nombreux outils tels que ESLint, Prettier, Commitlint ou les tests unitaires.
- **Le workflow Node.js** : bien qu'installé via npm ou Yarn, Husky est indépendant du langage utilisé dans le projet.

Les hooks les plus fréquemment utilisés sont :

- `pre-commit` : exécute des validations avant l'enregistrement d'un commit (linting, tests).
- `commit-msg` : vérifie que le message de commit respecte une convention donnée.
- `pre-push` : lance des vérifications avant l'envoi du code sur le dépôt distant.

Exemples et cas d'usage :

- Lancer ESLint automatiquement sur les fichiers modifiés avant chaque commit.
- Exécuter Commitlint pour garantir que les messages de commit respectent Conventional Commits.
- Vérifier que les tests unitaires passent avant chaque push.
- Appliquer Prettier pour uniformiser le formatage du code source.
- Bloquer la création de commits vides ou sans description.

Avantages principaux :

- Standardisation des processus de validation dans toute l'équipe.
- Réduction des erreurs humaines et des régressions en amont des CI/CD.
- Facilité de mise en œuvre et de configuration.
- Compatibilité avec de nombreux outils de qualité logicielle.

- Exécution rapide et locale, sans dépendre de l'environnement distant.

En synthèse, Husky est un composant essentiel pour fiabiliser et automatiser les workflows de validation des projets modernes. Il contribue à instaurer une culture DevOps orientée qualité et à renforcer la cohérence entre les contributeurs.

Références suggérées :

- Husky Documentation – `https://typicode.github.io/husky/`
- Git Hooks Documentation – `https://git-scm.com/docs/githooks`
- Commitlint Documentation – `https://commitlint.js.org/`
- ESLint Documentation – `https://eslint.org/docs/latest/`
- Prettier Documentation – `https://prettier.io/docs/en/`

6.1.4 Semantic Release

Semantic Release est un outil open source qui automatise le versionnement et la publication des packages logiciels en s'appuyant sur les messages de commit et le principe du versionnement sémantique (Semantic Versioning). Il supprime le besoin de mise à jour manuelle du numéro de version et de rédaction des changelogs, contribuant ainsi à la fiabilisation et à l'industrialisation des processus de release.

Semantic Release répond à plusieurs enjeux stratégiques : réduire les erreurs humaines dans les versions publiées, accélérer le cycle de livraison, renforcer la traçabilité des évolutions et homogénéiser les workflows de publication entre équipes. En automatisant intégralement la release, il permet aux développeurs de se concentrer sur la qualité fonctionnelle plutôt que sur les tâches administratives.

, Semantic Release repose sur plusieurs concepts clés :

- **Les conventions de commit** : le projet s'appuie sur des formats structurés (par exemple Conventional Commits) pour déduire automatiquement l'impact des changements (correctifs, nouvelles fonctionnalités, breaking changes).
- **Le calcul automatique de la version** : en fonction des types de commits depuis la dernière release, Semantic Release incrémente la version majeure, mineure ou corrective.
- **La génération du changelog** : compilation automatique des changements pertinents dans un format lisible.
- **La publication** : déploiement automatisé vers les registres de packages (npm, Maven, Docker Hub) et création des tags Git correspondants.

Semantic Release s'intègre naturellement dans des pipelines CI/CD (GitHub Actions, GitLab CI, CircleCI), garantissant que chaque merge dans la branche principale déclenche la création d'une nouvelle version stable.

Exemples et cas d'usage :

- Publier automatiquement un package npm lorsque de nouvelles fonctionnalités sont

mergées.
- Générer un changelog détaillé à partir des commits, sans intervention manuelle.
- Tagger les versions dans Git et créer des releases GitHub avec les notes correspondantes.
- Déclencher un pipeline de build Docker et pousser l'image versionnée sur un registre.
- Refuser les releases en cas de non-respect des conventions de commit.

Avantages principaux :

- Automatisation complète et fiabilisée du cycle de versionnement et de publication.
- Réduction drastique des erreurs humaines et des oublis dans la gestion des versions.
- Traçabilité et transparence accrues grâce aux changelogs générés automatiquement.
- Compatibilité avec de nombreux systèmes CI/CD et écosystèmes de packaging.
- Homogénéité des pratiques de release entre les projets et les équipes.

En synthèse, Semantic Release est une solution stratégique pour les organisations souhaitant industrialiser et sécuriser leur processus de publication. Il apporte une cohérence et une rapidité qui renforcent la qualité et la crédibilité des livraisons logicielles.

Références suggérées :

- Semantic Release Documentation – `https://semantic-release.gitbook.io/`
- Conventional Commits – `https://www.conventionalcommits.org/`
- Semantic Versioning – `https://semver.org/`
- GitHub Actions Documentation – `https://docs.github.com/en/actions`
- npm Publishing Guide – `https://docs.npmjs.com/creating-and-publishing-unsc`

6.2 Les conventions de commits

Les conventions de commits désignent l'ensemble des règles et des bonnes pratiques qui encadrent la rédaction des messages de commit dans un système de gestion de versions (comme Git). Elles visent à standardiser la documentation des changements, à faciliter la compréhension de l'historique d'un projet et à automatiser certaines tâches (génération de changelogs, déclenchement de pipelines CI/CD, versionnement sémantique). Leur adoption contribue à renforcer la qualité des projets logiciels et la collaboration entre les équipes.

les conventions de commits permettent de valoriser la traçabilité et la lisibilité du code. Elles facilitent la revue des changements lors des audits, améliorent la communication entre développeurs et garantissent que l'évolution du produit est documentée de façon claire et structurée. Elles sont également un levier de professionnalisation et de crédibilité vis-à-vis des partenaires et des clients, qui attendent des processus de développement rigoureux et transparents.

, plusieurs standards de conventions ont émergé, notamment :

- **Conventional Commits** : une spécification populaire qui définit un format structuré basé sur des préfixes et des catégories. Exemple :

```
feat(auth): add JWT authentication
fix(api): correct error handling in user service
docs(readme): update installation instructions
```

- **Semantic Versioning** : combiné aux conventions de commits, il permet de déclencher automatiquement les incréments de version (MAJOR, MINOR, PATCH) selon la nature des changements.
- **Gitmoji** : l'usage d'emojis standardisés pour symboliser visuellement le type de modification :

```
eat: add search functionality
fix: resolve crash on startup
docs: improve API documentation
```

Les conventions de commits permettent d'automatiser des processus critiques :

- Génération de changelogs clairs à partir des messages structurés.
- Déclenchement de pipelines CI/CD conditionnés à certains types de changements.
- Application automatique de politiques de versionnement.
- Vérification des formats de message via des hooks Git (ex. Commitlint).

Exemples et cas d'usage :

- Utiliser la convention Conventional Commits pour tous les projets d'un département afin de générer automatiquement la documentation des versions.
- Configurer un pipeline CI/CD qui refuse les commits non conformes au format attendu.
- Associer des préfixes (feat, fix, chore) aux incréments automatiques de version selon Semantic Versioning.
- Appliquer des tags de breaking change via l'indication `BREAKING CHANGE` dans le corps du commit.

Avantages principaux :

- Lisibilité et compréhension accrues de l'historique des changements.
- Automatisation de la génération des notes de version et du versionnement.
- Réduction des erreurs humaines grâce aux validations automatiques.
- Amélioration de la collaboration et de la revue de code.
- Renforcement de la transparence et de la traçabilité du cycle de développement.

En synthèse, l'adoption de conventions de commits structurées ne constitue pas uniquement une formalité : elle participe pleinement à l'industrialisation et à la qualité des processus de développement logiciel. Elle s'inscrit dans une démarche globale d'automatisation, de traçabilité et de professionnalisation des projets.

Références suggérées :

- Conventional Commits Specification – https://www.conventionalcommits.org/
- Semantic Versioning – https://semver.org/
- Gitmoji – https://gitmoji.dev/
- Pro Git Book – https://git-scm.com/book/en/v2
- Gousios, G., Spinellis, D. (2012). GIT-EVOLVE: A Software Evolution Tool Based on Git.

6.3 Mise en place des pipelines CI/CD

La mise en place d'une chaîne CI/CD (Continuous Integration / Continuous Deployment) constitue un levier essentiel pour automatiser la construction, le test et le déploiement des applications. Ce processus contribue à réduire les délais de mise en production, à limiter les erreurs humaines et à fiabiliser les évolutions logicielles.

6.3.1 Conteneurisation des applications

La première étape du pipeline CI/CD consiste à conteneuriser les applications développées. Pour ce faire, des fichiers `Dockerfile` ont été créés pour chaque projet, décrivant :

- Le système de base à utiliser (par exemple `python:3.10-slim`, `node:lts`).
- L'installation des dépendances applicatives via `pip`, `npm` ou autres gestionnaires de paquets.
- La copie du code source et des fichiers de configuration.
- La définition du point d'entrée de l'application (`ENTRYPOINT`).

Cette approche garantit que chaque build produit une image identique, facilement transportable et exécutable dans tout environnement Kubernetes.

6.3.2 Tagging des images

Le versionnement des images conteneurisées est essentiel pour assurer la traçabilité des déploiements. Les pipelines CI/CD ont été configurés pour appliquer un schéma de tagging cohérent :

- Utilisation d'un tag unique basé sur le hash du commit Git.
- Création d'un tag lisible incluant le numéro de version et la branche (par exemple `v1.2.3-main`).
- Marquage automatique de l'image la plus récente comme `latest` pour simplifier les tests.

Ces conventions permettent de relier chaque déploiement Kubernetes à l'image exacte qui a été produite par la pipeline.

6.3.3 Configuration de l'authentification

La sécurité des échanges entre les outils CI/CD et le registre d'images est un point critique. Plusieurs mesures ont été prises :

- Création de comptes de service dédiés avec des droits restreints.
- Génération de tokens d'accès utilisés par les runners CI pour authentifier les pushes vers le registre.
- Stockage sécurisé des identifiants et des tokens dans Vault ou les variables protégées du système CI (par exemple GitLab CI/CD variables).

Ces précautions garantissent que seuls les processus autorisés peuvent publier ou récupérer des images.

6.3.4 Préparation des tâches des pipelines

Les tâches élémentaires des pipelines CI/CD ont été définies de manière modulaire. Elles incluent notamment :

- La phase de compilation et de tests unitaires.
- La construction des images Docker.
- Le scan de sécurité (par exemple avec Trivy) pour détecter les vulnérabilités connues.
- La validation syntaxique des manifests Kubernetes (linting).
- La mise en cache des dépendances pour accélérer les builds.

Chaque tâche est décrite dans un fichier YAML de pipeline et peut être exécutée indépendamment, favorisant la réutilisation entre projets.

6.3.5 Préparation des pipelines

Enfin, les pipelines complets ont été structurés et versionnés dans les dépôts Git des applications. Ces pipelines définissent :

- Les déclencheurs automatiques (push sur la branche principale, création d'une release, merge request).
- Les variables d'environnement spécifiques à chaque environnement cible (dev, recette, production).
- Les étapes de build, de test, de publication et de déploiement continu.
- Les conditions de déclenchement manuel ou automatique de certaines étapes sensibles (par exemple le déploiement en production).

L'ensemble du processus CI/CD permet ainsi de passer d'un simple commit de code source jusqu'à la mise à jour des pods Kubernetes de façon totalement automatisée et traçable.

7. Observabilité, supervision et audits

7.1 Introduction

L'observabilité constitue un pilier essentiel dans la gestion moderne des systèmes distribués. Elle regroupe l'ensemble des pratiques, outils et processus permettant de comprendre le comportement d'une infrastructure, de détecter les anomalies et de garantir la conformité aux exigences de sécurité et de qualité de service. Dans le contexte des architectures conteneurisées et des microservices, la complexité opérationnelle s'est fortement accrue. Chaque requête peut transiter par de nombreux composants, rendant le diagnostic des incidents particulièrement difficile sans outils adaptés.

De plus, les exigences réglementaires et les bonnes pratiques imposent de disposer d'audits précis des accès, des changements et des événements critiques. L'observabilité et l'audit ne se limitent donc pas à la surveillance technique, mais participent également à la gouvernance et à la gestion des risques.

7.1.1 Contexte et enjeux de l'observabilité et de l'audit

La mise en place d'un dispositif d'observabilité répond à plusieurs enjeux majeurs :

- **Diagnostic rapide des incidents** : être capable de reconstituer le scénario précis ayant conduit à une panne ou à un comportement inattendu.
- **Optimisation des performances** : mesurer et analyser les temps de réponse, l'utilisation des ressources et la saturation éventuelle des composants.
- **Sécurité et traçabilité** : enregistrer les accès, les tentatives d'intrusion et les changements de configuration, afin de disposer de preuves en cas d'incident de sécurité.
- **Conformité réglementaire** : répondre aux obligations légales ou contractuelles en matière de conservation des journaux et de traçabilité des actions (par exemple, RGPD, ISO 27001).
- **Amélioration continue** : exploiter les données collectées pour identifier les points faibles et guider les évolutions de l'architecture.

Dans le cadre de ce projet, l'objectif est de fournir une visibilité unifiée sur l'état de l'infrastructure Kubernetes, des services applicatifs et des flux réseau, en s'appuyant sur des outils open source et des processus automatisés.

7.2 État de l'art et tendances actuelles

L'état de l'art de l'observabilité se structure autour de trois piliers principaux, souvent désignés sous l'acronyme **MELT** (Metrics, Events, Logs, Traces) :

1. **Les métriques** : valeurs numériques collectées à intervalles réguliers (CPU, mémoire, latence). Des solutions comme Prometheus ou InfluxDB permettent de stocker et d'interroger ces mesures.
2. **Les événements et alertes** : notifications déclenchées par un seuil ou un changement d'état. Ces événements sont souvent gérés par Alertmanager ou des systèmes de gestion d'incidents (PagerDuty, Opsgenie).
3. **Les logs** : traces textuelles produites par les applications et les composants système. Les solutions modernes (Elastic Stack, Loki) facilitent la collecte et la recherche en temps réel.
4. **Les traces distribuées** : enregistrements du parcours d'une requête à travers les microservices. Des outils comme Jaeger et OpenTelemetry sont devenus des standards de facto.

Les tendances actuelles mettent en avant plusieurs évolutions :

- **L'adoption massive de l'open source** : la plupart des organisations privilégient des solutions libres et communautaires pour éviter l'enfermement propriétaire.
- **Le modèle as-a-Service** : de nombreux acteurs (Datadog, New Relic, Grafana Cloud) proposent des plateformes hébergées intégrant l'ensemble des fonctionnalités d'observabilité.
- **La convergence des données** : les métriques, logs et traces ne sont plus traités isolément mais regroupés dans des plateformes unifiées facilitant les corrélations.
- **L'intégration avec GitOps et l'automatisation** : les configurations de monitoring et d'alerting sont versionnées et déployées de la même manière que les ressources Kubernetes.
- **La montée en puissance d'OpenTelemetry** : ce projet CNCF s'impose comme le standard unique de collecte de la télémétrie dans les environnements cloud-native.

Ces évolutions permettent de construire des systèmes plus résilients, plus sécurisés et plus faciles à maintenir, tout en apportant une meilleure compréhension globale des environnements complexes.

7.2.1 Grafana

Grafana est une solution open source de visualisation et d'exploration de données, largement utilisée pour la supervision des infrastructures, le monitoring applicatif et l'analyse d'indicateurs métiers. Elle permet de créer des tableaux de bord interactifs et personnalisables qui agrègent des données provenant de multiples sources (Prometheus, InfluxDB, Elasticsearch, Loki, MySQL, etc.). Grâce à son approche modulaire et à sa richesse fonctionnelle, Grafana s'est imposé comme un standard de facto dans l'écosystème cloud-native et DevOps.

Grafana répond à plusieurs enjeux stratégiques : renforcer la visibilité sur les systèmes critiques,

réduire le temps de résolution des incidents, et améliorer la qualité des services. En centralisant la visualisation et l'analyse des métriques, logs et traces, Grafana facilite la prise de décision et contribue à l'amélioration continue des processus opérationnels. Son interface conviviale et ses capacités de partage simplifient la collaboration entre équipes techniques et parties prenantes.

, Grafana repose sur plusieurs composants essentiels :

- **Les datasources** : connecteurs vers des bases de données et des systèmes de monitoring (Prometheus, Graphite, ElasticSearch, CloudWatch, etc.).
- **Les dashboards** : ensembles de panels configurables qui visualisent les données sous forme de graphiques, jauges, tableaux et alertes.
- **Les panels** : éléments de visualisation individuels, paramétrés avec des requêtes, des transformations et des styles personnalisés.
- **Les alertes** : règles qui surveillent les seuils définis et déclenchent des notifications en cas d'anomalie.
- **Les organisations et utilisateurs** : système de gestion des accès, des permissions et des partages.

Grafana peut être déployé en standalone ou intégré dans des stacks d'observabilité complètes (exemple : Prometheus + Loki + Grafana). Son API REST et son support des plugins en font un outil particulièrement extensible et adaptable à tous les cas d'usage.

Exemples et cas d'usage :

- Visualiser en temps réel les métriques d'un cluster Kubernetes (CPU, mémoire, pods, etc.) en s'appuyant sur Prometheus.
- Corréler les logs applicatifs via Loki avec les indicateurs de performance pour diagnostiquer plus rapidement un incident.
- Configurer des alertes pour notifier les équipes DevOps en cas de dépassement d'un seuil critique (ex. latence élevée).
- Créer des tableaux de bord métiers synthétiques avec des indicateurs clés (KPI) accessibles aux décideurs.
- Intégrer Grafana avec Slack ou PagerDuty pour centraliser les alertes et les escalades.

Avantages principaux :

- Plateforme unifiée de visualisation multi-sources et multi-formats.
- Interface ergonomique et hautement personnalisable.
- Large écosystème de plugins, dashboards communautaires et connecteurs.
- Support des alertes natives et intégration avec les systèmes de notification.
- Extensibilité via API REST, provisioning as code et gestion des permissions fine.
- Solution open source mature, supportée par une large communauté.

En synthèse, Grafana est une brique centrale des stratégies d'observabilité modernes. Il permet aux organisations de transformer leurs données en connaissances actionnables, d'améliorer la performance opérationnelle et de renforcer la confiance dans les systèmes distribués.

Références suggérées :

- Grafana Documentation – `https://grafana.com/docs/`
- Grafana GitHub Repository – `https://github.com/grafana/grafana`
- Prometheus Documentation – `https://prometheus.io/docs/`
- Loki Documentation – `https://grafana.com/docs/loki/latest/`
- Grafana Labs Blog – `https://grafana.com/blog/`

7.2.2 Prometheus

Prometheus est une solution open source de monitoring et d'alerte initialement développée par SoundCloud, puis incubée par la Cloud Native Computing Foundation (CNCF). Il est devenu l'un des piliers des architectures cloud-native grâce à sa capacité à collecter, stocker et interroger des métriques temporelles de manière performante. Prometheus est particulièrement reconnu pour son modèle de données multidimensionnel, son langage de requête puissant (PromQL) et sa facilité d'intégration avec Kubernetes.

Prometheus répond à plusieurs enjeux essentiels : renforcer la visibilité sur les systèmes critiques, anticiper les incidents par une surveillance proactive et réduire le temps moyen de résolution des problèmes (MTTR). Il contribue à l'amélioration continue des performances applicatives et à la qualité de service délivrée aux utilisateurs. Grâce à sa modularité, Prometheus s'adapte à des environnements variés (infrastructures cloud, conteneurs, clusters Kubernetes, applications legacy).

, Prometheus repose sur plusieurs composants clés :

- **Le serveur Prometheus** : responsable de la collecte des métriques via le protocole HTTP/HTTPS (pull model) et du stockage local des séries temporelles.
- **Les exporters** : processus ou agents qui exposent des métriques au format Prometheus (ex.: Node Exporter, Blackbox Exporter, MySQL Exporter).
- **Le langage PromQL** : langage de requête permettant d'agréger, filtrer et analyser les métriques.
- **Les règles d'alerte** : expressions PromQL évaluées en continu pour générer des alertes.
- **Alertmanager** : composant dédié à la gestion et au routage des alertes vers les canaux de notification (email, Slack, PagerDuty).

Prometheus est particulièrement bien intégré dans l'écosystème Kubernetes grâce à la découverte de services automatique, facilitant ainsi la supervision des clusters et des workloads dynamiques.

Exemples et cas d'usage :

- Superviser l'utilisation CPU et mémoire des nœuds Kubernetes via Node Exporter.
- Mesurer la latence et le taux d'erreurs des endpoints HTTP exposés par des microservices.
- Définir une alerte déclenchée lorsque la disponibilité d'un service passe sous un seuil critique.
- Stocker des métriques de performance applicative pour des analyses historiques.

- Visualiser les métriques dans Grafana grâce au connecteur natif Prometheus.

Avantages principaux :

- Modèle de collecte pull simplifiant l'intégration avec les workloads dynamiques.
- Stockage en séries temporelles optimisé pour la performance et la rétention longue durée.
- Langage PromQL expressif et puissant pour l'analyse des données.
- Intégration native avec Kubernetes et les architectures cloud-native.
- Écosystème riche d'exporters et de dashboards communautaires.
- Solution open source mature, soutenue par la CNCF et une large communauté.

En synthèse, Prometheus est un outil incontournable des stratégies de monitoring et d'observabilité modernes. Il apporte robustesse, flexibilité et transparence à la supervision des infrastructures complexes et des applications distribuées.

Références suggérées :

- Prometheus Documentation – `https://prometheus.io/docs/`
- Prometheus GitHub Repository – `https://github.com/prometheus/prometheus`
- CNCF Prometheus Project – `https://www.cncf.io/projects/prometheus/`
- Grafana Documentation – `https://grafana.com/docs/grafana/latest/datasources/prometheus/`
- Monitoring with Prometheus – James Turnbull. O'Reilly Media.

7.2.3 Loki

Loki est une solution open source développée par Grafana Labs pour la centralisation et l'analyse des logs. Conçu pour s'intégrer étroitement avec Prometheus, Grafana et l'écosystème cloud-native, Loki adopte une approche innovante : il indexe uniquement les labels (métadonnées) et non le contenu complet des logs. Cette caractéristique en fait une solution plus légère, plus scalable et plus économique que les systèmes traditionnels d'indexation complète (par exemple Elasticsearch).

Loki répond à plusieurs enjeux stratégiques : renforcer la visibilité sur les applications et les infrastructures, accélérer les diagnostics d'incidents et réduire le coût du stockage et du traitement des logs. Il permet aux équipes SRE, DevOps et de support de disposer d'un outil cohérent avec leur stack de monitoring et d'observabilité, facilitant la corrélation entre métriques, logs et alertes.

, Loki repose sur plusieurs concepts clés :

- **Les labels** : clés et valeurs attachées aux streams de logs (par exemple ‘app="nginx"‘), utilisés comme index.
- **Les chunks** : segments de données compressées regroupant les logs non indexés.
- **Promtail** : agent qui collecte les logs sur les hôtes et les envoie à Loki.
- **Le langage LogQL** : langage de requête inspiré de PromQL, permettant d'interroger et

d'agréger les logs.
- **L'intégration Grafana** : visualisation et exploration des logs dans des dashboards unifiés.

Grâce à sa compatibilité Kubernetes et à sa scalabilité horizontale, Loki est particulièrement adapté aux environnements cloud-native et microservices.

Exemples et cas d'usage :

- Collecter les logs des conteneurs Kubernetes via Promtail et les regrouper par namespace, pod et container.
- Corréler des pics de latence observés dans Prometheus avec les logs applicatifs de la même période.
- Configurer une alerte Grafana qui affiche les logs d'erreurs critiques lors d'un incident.
- Archiver les logs applicatifs de manière compressée et économique sur le long terme.
- Rechercher rapidement les logs d'un service particulier via LogQL.

Avantages principaux :

- Scalabilité horizontale et stockage économique grâce à l'indexation minimale.
- Intégration native avec Grafana et Prometheus.
- Requêtes puissantes et flexibles avec LogQL.
- Support complet de Kubernetes et des environnements multi-cloud.
- Solution open source mature et soutenue par une large communauté.

En synthèse, Loki est un composant central de la stack d'observabilité cloud-native. Il permet aux organisations de simplifier et de rationaliser la gestion des logs, tout en réduisant les coûts et en améliorant la capacité de diagnostic et de supervision.

Références suggérées :

- Loki Documentation – `https://grafana.com/docs/loki/latest/`
- Loki GitHub Repository – `https://github.com/grafana/loki`
- Grafana Documentation – `https://grafana.com/docs/`
- Promtail Documentation – `https://grafana.com/docs/loki/latest/clients/promtail/`
- CNCF Loki Project – `https://www.cncf.io/projects/loki/`

7.2.4 Tempo

Tempo est une solution open source de traçage distribué développée par Grafana Labs. Elle permet de collecter, stocker et interroger des traces issues d'applications distribuées, sans nécessiter d'indexation complexe. Conçu pour compléter Prometheus et Loki, Tempo s'intègre naturellement dans la stack d'observabilité cloud-native. Grâce à son architecture optimisée, il offre un stockage massif et économique des traces, tout en simplifiant la corrélation avec les métriques et les logs.

Tempo répond à plusieurs enjeux stratégiques : comprendre la performance des applications

microservices, diagnostiquer les latences et les erreurs en production, et améliorer l'expérience utilisateur. En facilitant l'analyse des parcours complets des requêtes, Tempo contribue à réduire le temps moyen de résolution des incidents (MTTR) et à optimiser la qualité de service.

, Tempo s'appuie sur plusieurs concepts essentiels :

- **Les traces** : enregistrements d'un ensemble de spans représentant les étapes d'une requête distribuée.
- **Les spans** : unités atomiques contenant les métadonnées sur chaque étape (durée, étiquettes, événements).
- **L'ingester** : composant qui reçoit et stocke les traces dans des blocs compressés.
- **L'indexation minimale** : Tempo utilise un modèle « No Index », reposant uniquement sur l'identifiant de trace, ce qui simplifie le stockage et réduit les coûts.
- **L'intégration avec Grafana** : Tempo permet de visualiser et de rechercher les traces via l'interface Grafana, en corrélation avec les métriques Prometheus et les logs Loki.

Tempo est compatible avec les formats de traçage standardisés comme OpenTelemetry, Jaeger et Zipkin, facilitant l'intégration avec un grand nombre de frameworks et de langages.

Exemples et cas d'usage :

- Collecter les traces d'une application microservices instrumentée avec OpenTelemetry.
- Corréler un pic de latence observé dans Prometheus avec les traces détaillant les appels entre services.
- Rechercher des traces via leur identifiant unique depuis un log collecté par Loki.
- Visualiser les dépendances entre services et la durée de chaque étape d'une requête.
- Conserver l'historique des traces pour analyse et optimisation des performances applicatives.

Avantages principaux :

- Scalabilité horizontale et stockage économique sans indexation complexe.
- Compatibilité native avec OpenTelemetry, Jaeger et Zipkin.
- Corrélation simple avec les métriques et logs via Grafana.
- Facilité de déploiement dans des environnements Kubernetes et cloud.
- Solution open source soutenue par la CNCF et Grafana Labs.

En synthèse, Tempo est un pilier des architectures d'observabilité modernes. Il apporte visibilité, compréhension et capacité de diagnostic sur les systèmes distribués, en complément parfait de Prometheus et Loki.

Références suggérées :

- Tempo Documentation – `https://grafana.com/docs/tempo/latest/`
- Tempo GitHub Repository – `https://github.com/grafana/tempo`
- OpenTelemetry Documentation – `https://opentelemetry.io/docs/`
- Jaeger Documentation – `https://www.jaegertracing.io/docs/`

- Grafana Labs Blog – `https://grafana.com/blog/`

7.2.5 OpenTelemetry

OpenTelemetry est une suite open source de spécifications, d'outils et de SDK destinée à la collecte, au traitement et à l'exportation des signaux d'observabilité : métriques, logs et traces. Né de la fusion des projets OpenTracing et OpenCensus, OpenTelemetry est aujourd'hui un projet de la Cloud Native Computing Foundation (CNCF) et constitue le standard de facto pour instrumenter les applications modernes, qu'elles soient monolithiques ou microservices.

OpenTelemetry répond à des enjeux stratégiques majeurs : renforcer la visibilité sur les systèmes distribués, anticiper les problèmes de performance, améliorer l'expérience utilisateur et faciliter la transformation digitale. En proposant un cadre unifié et standardisé, OpenTelemetry contribue à réduire la complexité opérationnelle et à accélérer la mise en place de stratégies d'observabilité efficaces.

, OpenTelemetry repose sur plusieurs composants essentiels :

- **Les SDK** : bibliothèques spécifiques aux langages (Java, Go, Python, etc.) qui instrumentent automatiquement ou manuellement les applications.
- **Le Collector** : service qui reçoit, traite et exporte les signaux vers des backends comme Prometheus, Jaeger, Tempo, Zipkin ou Grafana.
- **Les exporters** : composants qui envoient les données collectées vers des systèmes tiers.
- **Les ressources** : ensembles de métadonnées qui décrivent les attributs des services (nom, version, environnement).
- **Les protocoles** : principalement OTLP (OpenTelemetry Protocol), conçu pour un transport efficace et interopérable des signaux.

OpenTelemetry est conçu comme un projet modulaire et extensible, permettant aux équipes d'adopter progressivement la collecte des traces, des métriques et des logs, selon leurs priorités.

Exemples et cas d'usage :

- Instrumenter automatiquement une API REST en Java pour collecter les latences et les erreurs.
- Exporter les métriques applicatives vers Prometheus et les traces vers Tempo via le Collector.
- Corréler les logs et les traces grâce à des identifiants de corrélation injectés automatiquement.
- Visualiser le graphe de dépendances entre microservices dans Jaeger ou Grafana.
- Monitorer en temps réel les indicateurs de performance d'un cluster Kubernetes.

Avantages principaux :

- Standardisation des signaux d'observabilité (métriques, traces, logs).
- Large compatibilité multi-langages et multi-plateformes.

- Collecte et exportation flexibles via le Collector.
- Intégration fluide avec les principaux backends d'observabilité.
- Support actif par une large communauté et les principaux éditeurs cloud.
- Réduction de la complexité opérationnelle et meilleure visibilité end-to-end.

En synthèse, OpenTelemetry est une brique fondamentale de l'observabilité moderne. Il offre un cadre unifié, extensible et standardisé, permettant aux organisations de mieux comprendre et améliorer le comportement de leurs applications distribuées.

Références suggérées :

- OpenTelemetry Documentation – `https://opentelemetry.io/docs/`
- OpenTelemetry GitHub – `https://github.com/open-telemetry/opentelemetry-sp`
- CNCF OpenTelemetry Project – `https://www.cncf.io/projects/opentelemetry/`
- Jaeger Documentation – `https://www.jaegertracing.io/docs/`
- Grafana Tempo Documentation – `https://grafana.com/docs/tempo/latest/`

7.3 Mise en place du monitoring continu

La mise en place du monitoring continu est indispensable pour assurer la supervision proactive de l'infrastructure et des applications. Le projet s'appuie principalement sur la solution Prometheus, qui collecte les métriques exposées par les composants Kubernetes et les services applicatifs via des endpoints HTTP (`/metrics`).

Les étapes principales comprennent :

- Le déploiement de l'opérateur Prometheus dans le cluster Kubernetes.
- La définition des `ServiceMonitor` et `PodMonitor` permettant de déclarer les cibles à surveiller.
- La configuration des règles d'alerting basées sur les métriques collectées.
- L'intégration avec Grafana pour la visualisation en temps réel des tableaux de bord.

Grâce à cette approche, l'état des systèmes est suivi en continu et les anomalies peuvent être détectées de manière précoce.

7.4 Gestion des alertes et des incidents

La gestion des alertes repose sur l'utilisation de Prometheus Alertmanager. Ce composant reçoit les alertes générées par Prometheus selon les règles prédéfinies (par exemple seuils de charge CPU, absence de pods, erreurs applicatives).

Les principaux éléments mis en place sont :

- La configuration des routes d'alerte permettant d'acheminer les notifications vers les destinataires appropriés (e-mails, canaux Slack, systèmes d'escalade).
- La définition des silences pour désactiver temporairement des alertes lors de maintenances

planifiées.
- L'organisation des alertes par sévérité et par environnement (développement, recette, production).
- La documentation des procédures de réponse aux incidents.

Ce dispositif contribue à réduire le temps moyen de résolution des incidents et à limiter leur impact sur les utilisateurs.

7.5 Gestion des logs

La collecte centralisée des logs est un pilier de l'observabilité. Dans ce projet, la stack EFK (Elasticsearch, Fluentd, Kibana) ou Loki a été utilisée afin d'agréger les journaux produits par :

- Les pods Kubernetes.
- Les composants système des nœuds.
- Les applications déployées.

Les principales fonctionnalités implémentées sont :

- La normalisation et l'enrichissement des logs avec des métadonnées Kubernetes (namespace, nom du pod, labels).
- L'indexation et la conservation des journaux selon des politiques de rétention définies.
- La recherche en temps réel et la création de tableaux de bord personnalisés dans Kibana ou Grafana.
- La détection automatique d'événements critiques et la génération de notifications.

Cette centralisation des logs permet de gagner en efficacité lors des diagnostics et de garantir la traçabilité complète des événements.

7.6 Gestion des traces distribuées

Les traces distribuées apportent une vision fine du parcours des requêtes à travers les différents microservices. Le projet s'appuie sur la suite OpenTelemetry pour instrumenter les applications et collecter les traces.

Les éléments mis en œuvre sont :

- L'instrumentation automatique et manuelle du code pour générer des spans et propager le contexte de trace.
- Le déploiement du collector OpenTelemetry dans Kubernetes.
- L'export des traces vers un backend comme Jaeger ou Grafana Tempo.
- La visualisation des dépendances et des performances des requêtes via des interfaces web dédiées.

Les traces permettent notamment de :

- Identifier les goulets d'étranglement et les sources de latence.
- Suivre l'impact des erreurs sur l'ensemble du parcours utilisateur.
- Corréler les traces avec les logs et les métriques.

7.7 Automatisation des audits et de la conformité

L'automatisation des audits et de la conformité vise à garantir le respect des exigences réglementaires et des politiques internes.

Pour atteindre cet objectif, plusieurs mesures ont été adoptées :

- L'activation de l'audit logging Kubernetes pour enregistrer toutes les requêtes API et changements d'état.
- La centralisation des journaux d'audit dans Elasticsearch pour une conservation longue durée et une recherche efficace.
- La mise en place de règles de contrôle (OPA/Gatekeeper) validant les configurations déployées (politiques de sécurité, labels obligatoires, restrictions de privilèges).
- La génération automatique de rapports de conformité sur les accès et les déploiements.

Ces mécanismes facilitent les contrôles internes et externes et contribuent à renforcer la confiance dans la plateforme.

8. Conclusion générale

8.1 Conclusion générale

Le projet présenté dans ce mémoire s'inscrit dans un contexte d'évolution rapide des besoins en automatisation, en sécurité et en observabilité des infrastructures informatiques. Partant d'un environnement initial marqué par une forte hétérogénéité et des processus majoritairement manuels, l'objectif principal était de mettre en place une architecture moderne, automatisée et fiable, afin de soutenir la croissance et la performance de l'entreprise Oneex.

Les travaux réalisés ont permis d'atteindre ces objectifs à travers l'intégration d'outils et de pratiques DevOps éprouvés. L'infrastructure virtualisée, combinée à l'orchestration des conteneurs, à la gestion centralisée des secrets et à la mise en place d'un monitoring complet, constitue un socle robuste et évolutif.

Ce projet apporte une valeur ajoutée importante : une réduction significative du temps de déploiement, une amélioration de la sécurité des systèmes et une meilleure visibilité sur l'état des services. Ces bénéfices contribuent directement à renforcer la compétitivité et la capacité d'innovation de l'entreprise.

8.2 Les limites du projet

Malgré les résultats atteints, certaines limites ont été identifiées au cours du projet :

- La complexité de la montée en compétence sur certaines technologies, notamment Vault et Argo CD, a nécessité un temps d'apprentissage important.
- Le projet a été limité par la disponibilité des ressources matérielles et le temps alloué, ce qui a retardé certaines phases de validation.
- La documentation des procédures n'a pas pu être finalisée de manière exhaustive dans les délais impartis.
- Certaines optimisations de performance et de sécurité (par exemple l'automatisation complète des sauvegardes chiffrées) n'ont pas pu être mises en œuvre.

8.3 Les améliorations possibles

Plusieurs pistes d'amélioration pourront être envisagées à l'avenir :

- Compléter et enrichir la documentation technique et utilisateur.
- Renforcer la scalabilité du cluster Kubernetes pour accueillir de nouvelles applications.
- Intégrer des tests automatisés de sécurité (scans de vulnérabilités, compliance).

- Mettre en place des alertes proactives plus fines, basées sur des seuils dynamiques.
- Continuer la formation des équipes sur les outils mis en place afin de favoriser l'adoption.

Ces évolutions contribueront à renforcer encore la robustesse et la maturité de l'infrastructure.

8.4 Les enseignements personnels

Ce projet a été particulièrement riche en apprentissages, tant sur le plan technique que sur le plan humain.

Sur le plan technique, il m'a permis d'acquérir des compétences solides dans la mise en œuvre d'infrastructures automatisées, en explorant des outils variés tels que Terraform, Ansible, Kubernetes, Vault et Prometheus. J'ai également pu mieux comprendre les enjeux liés à la sécurité et à la haute disponibilité des services.

Sur le plan organisationnel, ce projet m'a appris à planifier des tâches complexes, à prioriser les actions et à collaborer efficacement avec les équipes internes. La nécessité de documenter chaque étape et de structurer les livrables a renforcé ma rigueur et ma capacité à travailler de manière autonome.

Enfin, cette expérience a confirmé mon intérêt pour le domaine du DevOps et de l'automatisation, et m'a donné envie de continuer à développer ces compétences dans un cadre professionnel.

8.5 Conclusion

Ce projet de mise en place d'une infrastructure automatisée et sécurisée pour Oneex a permis de répondre à des enjeux à la fois opérationnels et stratégiques. En s'appuyant sur des outils modernes et des pratiques DevOps éprouvées, il a été possible d'améliorer significativement la fiabilité, la sécurité et la rapidité des déploiements.

L'architecture mise en œuvre offre une base solide et évolutive pour le développement futur des services, tout en assurant une gestion centralisée et sécurisée des configurations et des secrets. Les équipes disposent désormais d'un environnement cohérent, scalable et maintenable, capable de s'adapter aux besoins croissants de l'entreprise.

Ce mémoire a présenté de manière détaillée les différentes étapes de conception et de réalisation de cette solution, ainsi que les résultats obtenus. Il met en lumière l'importance de l'automatisation, de la sécurité et de l'observabilité dans la gestion d'infrastructures modernes, et souligne la nécessité d'adopter une approche proactive en matière de gouvernance technique.

Il convient de rappeler la distinction entre deux approches fondamentales en matière de sécurité :

- **La sécurité par design** (*security by design*) consiste à intégrer les mécanismes de protection dès la conception des systèmes, en anticipant les menaces et en appliquant systématiquement des principes comme le moindre privilège, la segmentation et la défense en profondeur.

- **La sécurité par obscurité** (*security by obscurity*) repose uniquement sur la dissimulation des détails techniques (par exemple, masquer les configurations ou ne pas documenter les processus). Bien qu'elle puisse constituer une mesure complémentaire, elle ne peut en aucun cas se substituer à une politique de sécurité robuste et vérifiable.

Enfin, il est essentiel de continuer à investir dans la formation des équipes, l'évolution des outils et la diffusion des bonnes pratiques pour garantir la pérennité et la sécurité de l'infrastructure.

8.5.1 Perspectives d'évolution

L'initiative **Infra_v2** ouvre la voie à plusieurs axes d'amélioration visant à renforcer la performance, la fiabilité et l'efficience des systèmes :

- **Renforcement de l'automatisation** : intégrer de nouveaux processus automatisés, notamment les tests d'intégration, les vérifications de sécurité et les audits de conformité directement dans les pipelines CI/CD.
- **Observabilité avancée** : déployer une solution de télémétrie unifiée (par exemple OpenTelemetry) afin de collecter métriques, logs et traces dans un format standardisé, facilitant l'analyse et le diagnostic en temps réel.
- **Scalabilité horizontale** : permettre l'ajout dynamique de nœuds Kubernetes en fonction de la charge et de l'évolution des besoins applicatifs.
- **Sécurité renforcée** : généraliser le chiffrement des communications internes, la rotation automatique des secrets et l'application stricte du principe du moindre privilège.
- **Standardisation des workflows** : promouvoir l'adoption systématique des principes GitOps et l'harmonisation des pratiques de déploiement au sein de toutes les équipes.
- **Optimisation des coûts** : mettre en place des mécanismes de scaling automatique et d'analyse des consommations pour ajuster les ressources en fonction de l'activité et réduire les coûts d'infrastructure.

Ces perspectives s'inscrivent dans une démarche continue d'amélioration et d'industrialisation des processus techniques.

8.5.2 Bilan technique et organisationnel

L'expérience acquise dans ce projet a permis de mettre en évidence plusieurs points forts et points faibles, tant sur le plan technique qu'organisationnel.

Points forts techniques

- Infrastructure déclarative et versionnée, garantissant la reproductibilité et la traçabilité des configurations.
- Automatisation complète du cycle de vie des environnements (provisionnement, configuration, déploiement).

- Haute disponibilité native grâce à l'orchestration Kubernetes et au découplage des composants.
- Possibilité de rollback rapide et maîtrisé en cas d'incident.
- Intégration transparente d'un système centralisé de gestion des secrets.

Points faibles techniques

- Courbe d'apprentissage élevée pour la maîtrise et l'exploitation de l'ensemble des outils.
- Complexité accrue nécessitant une veille technologique constante et un maintien de compétences soutenu.
- Dépendance forte à l'intégrité des systèmes d'orchestration (GitOps, API Kubernetes), dont une indisponibilité peut impacter la production.

Points forts organisationnels

- Processus standardisés réduisant le risque d'erreurs humaines et améliorant la qualité globale des déploiements.
- Visibilité et transparence des configurations grâce au versionnement et à la centralisation.
- Accélération notable des cycles de livraison et meilleure réactivité des équipes.
- Renforcement de la collaboration inter-équipes via des workflows communs et partagés.

Points faibles organisationnels

- Nécessité d'une conduite du changement approfondie pour faire adopter les nouveaux outils et méthodologies.
- Risque de silos de compétences si la montée en compétence n'est pas homogène.
- Temps initial d'implémentation important pour structurer et standardiser l'ensemble des pipelines et des pratiques.

En synthèse, cette démarche constitue un socle technologique solide et évolutif, posant les bases d'un système plus résilient et sécurisé. Elle ouvre de nombreuses opportunités d'optimisation et d'innovation à moyen terme.

8.6 Synthèse et justification des choix retenus

L'analyse des approches actuelles met en évidence une convergence autour de plusieurs piliers : l'automatisation par l'Infrastructure as Code et le GitOps, la conteneurisation orchestrée, l'observabilité comme fondement de la fiabilité, la sécurité intégrée dès la conception, la protection périmétrique et la standardisation des pipelines CI/CD. L'adoption de ces pratiques répond à la nécessité de garantir la cohérence, la traçabilité et la résilience des environnements tout en favorisant leur évolutivité.

8.7 Conclusion du chapitre

Ce chapitre a permis de dresser un panorama des concepts fondamentaux et des solutions de référence dans le domaine de l'automatisation des infrastructures et de la modernisation des systèmes d'information. Cette étude constitue un socle indispensable pour comprendre les orientations retenues dans la conception de l'architecture détaillée, présentée dans le chapitre suivant.