MEMOIRE D'ALTERNANCE

Industrialisation et Optimisation d'une usine logicielle cloud-native chez GE HealthCare

Réalisé par DASSI MANUEL

Sous la supervision de :

Référent Institut :

M. Marc BENNATAR

Référent Entreprise :

M. Abdelghani ACHIBANE

Année Académique

2024-2025

Table de matières

Remerciements

Je souhaite exprimer ma profonde gratitude à toutes les personnes qui m'ont accompagné et soutenu tout au long de cette année d'alternance, sur les plans professionnel et personnel.

Je remercie tout d'abord M. Abdelghani ACHIBANE, mon référent en entreprise chez GE HealthCare, pour son encadrement rigoureux, sa disponibilité et la confiance qu'il m'a accordée. Son expertise et ses conseils ont été déterminants dans ma progression technique. Les membres de l'équipe chez GE HealthCare la manager Madame Céline, Les staff ingénieurs David, Cedric, Alexis, Mathieu, Doan et tous les autres.

Je remercie également M. Marc Bennatar, mon référent pédagogique à l'institut PMN, pour son accompagnement constant et la qualité de ses conseils tout au long de mon cursus.

J'adresse aussi ma reconnaissance à l'ensemble de l'équipe de GE HealthCare pour leur accueil, leur esprit d'équipe et leur professionnalisme. Leur environnement de travail bienveillant et stimulant m'a permis de développer mes compétences dans les meilleures conditions.

Un remerciement particulier à mes collègues pour leur soutien quotidien, leur patience et leur bonne humeur. Grâce à leurs échanges et à leur esprit collaboratif, cette expérience a été à la fois formatrice et enrichissante.

Enfin, je remercie ma famille et mes proches pour leur soutien indéfectible et leurs encouragements, sans lesquels cette aventure n'aurait pas été possible.

Introduction

Cette alternance m'a offert l'occasion de mettre en pratique les enseignements de mon Master à l'Institut PMN dans un environnement exigeant et concret. Réalisée au sein de GE HealthCare, acteur majeur des technologies médicales, elle a porté sur l'industrialisation et l'optimisation d'une « usine logicielle » orientée cloud. L'objectif est simple à énoncer, mais ambitieux à réaliser : livrer plus rapidement des logiciels de qualité, fiables et sûrs, tout en respectant les exigences élevées du secteur de la santé en matière de sécurité, de conformité et de protection des données.

Le contexte dans lequel s'inscrit ce travail est marqué par des attentes fortes des établissements de santé et des professionnels : disposer d'outils numériques qui améliorent la prise en charge des patients, fluidifient les parcours de soins et soutiennent les décisions cliniques. Parallèlement, le cadre réglementaire se renforce, la sensibilité des données augmente et les organisations doivent faire preuve d'une grande rigueur dans la manière de concevoir, de tester, de déployer et d'exploiter leurs applications. Dans ce paysage, la capacité à industrialiser la production logicielle devient un levier stratégique : elle permet d'accélérer l'innovation sans compromettre la qualité ni la confiance.

Par « usine logicielle », on entend un ensemble cohérent de pratiques, d'outils et de règles de fonctionnement qui structurent tout le cycle de vie des applications : de la conception au déploiement, de l'exploitation au retour d'expérience. Une usine efficace repose sur des méthodes de travail standardisées, une automatisation poussée des étapes répétitives, des contrôles qualité présents à chaque étape, et des mécanismes de retour en arrière rapides en cas d'imprévu. L'approche orientée cloud renforce ces objectifs : elle favorise l'élasticité (adapter les ressources à la demande), la réactivité (déployer des évolutions plus fréquemment) et la résilience (tolérer les pannes et y remédier rapidement), tout en permettant une maîtrise plus fine des coûts.

Au cœur de la démarche se trouve la recherche d'un équilibre entre plusieurs exigences souvent perçues comme contradictoires. Il s'agit de livrer vite, sans sacrifier la qualité ; d'innover, sans prendre de risques disproportionnés; de standardiser, tout en gardant la souplesse nécessaire pour répondre à des besoins spécifiques. Concrètement, cela signifie simplifier et fiabiliser les enchaînements d'étapes qui vont de l'idée jusqu'à la mise à disposition d'une fonctionnalité;

rendre plus visible et plus compréhensible l'état réel des applications à chaque instant; instaurer des points de contrôle réguliers et transparents; et documenter des modes opératoires clairs, partagés par tous.

Mon alternance s'est concentrée sur l'amélioration de ces fondations. J'ai contribué à la clarification des étapes clés et à leur enchaînement, afin de réduire les zones d'ombre et d'éviter les retards. J'ai participé au renforcement des contrôles qualité tout au long du cycle, pour détecter plus tôt les anomalies, limiter les reprises tardives et sécuriser la mise en service. J'ai travaillé à rendre le suivi des applications plus lisible, ce qui permet aux équipes d'identifier rapidement les points d'attention et d'agir avec efficacité. Enfin, j'ai accompagné l'optimisation de l'usage des ressources du cloud, avec une attention particulière portée à la sobriété, à la maîtrise budgétaire et à la capacité à absorber des pics d'activité.

Au-delà des aspects techniques, la réussite d'une usine logicielle est d'abord une question d'organisation et de culture. Les équipes de développement, de qualité, de sécurité, d'exploitation et les métiers doivent partager des objectifs, des standards et un langage commun. Cela suppose des rituels réguliers, des revues croisées, des responsabilités clairement définies et une gouvernance qui tranche rapidement quand c'est nécessaire. L'adhésion collective à des règles simples et compréhensibles est un facteur déterminant : elle garantit la répétabilité des bonnes pratiques et facilite l'intégration de nouveaux projets ou de nouvelles équipes.

La dimension humaine est également centrale. Une usine logicielle performante ne se décrète pas, elle s'apprend et se vit au quotidien. La formation, l'accompagnement au changement et la diffusion des retours d'expérience sont essentiels pour ancrer durablement les nouvelles façons de travailler. Dans le cadre de mon alternance, j'ai contribué à la mise à disposition de guides pratiques, de modèles prêts à l'emploi et de supports de communication interne, afin d'aider les équipes à adopter des gestes communs et à gagner en autonomie.

Un autre enjeu majeur réside dans la capacité à mesurer les progrès. Pour rester à un niveau d'exigence élevé, il faut des indicateurs simples, lisibles et partagés. Le temps nécessaire pour livrer une évolution, le taux d'incidents après une mise en service, la fréquence des retours en

arrière, la satisfaction des utilisateurs internes et externes, autant de repères concrets qui permettent de piloter l'amélioration continue. Ces mesures ne doivent pas être perçues comme des contraintes supplémentaires, mais comme des outils d'alignement et de pilotage, elles aident à décider où concentrer l'effort, à arbitrer entre rapidité et prudence, et à objectiver les résultats.

Dans le secteur de la santé, la question de la confiance est omniprésente. Elle repose sur la qualité des produits, la protection des données et la transparence des processus. Industrialiser et optimiser une usine logicielle, c'est aussi rendre cette confiance visible, décrire comment une fonctionnalité est conçue, testée, et mise en service, expliquer comment on s'assure qu'elle ne dégrade pas l'existant; montrer comment on surveille son comportement dans le temps; et préciser comment on réagit lorsqu'un imprévu survient. Cette transparence, tournée vers l'interne autant que vers l'externe, contribue à créer un cadre de travail serein et responsable.

Le périmètre de ce rapport est volontairement centré sur les aspects structurants et les résultats observés, sans entrer dans des détails trop techniques. Il présentera d'abord le contexte et les missions menées chez GE HealthCare, en mettant en lumière les besoins identifiés et les objectifs fixés. Il décrira ensuite les choix d'organisation et de fonctionnement de l'usine logicielle orientée cloud: principes de standardisation, automatisation des étapes clés, règles de contrôle qualité, modes de déploiement progressifs et dispositifs de suivi. Il abordera également la question de la maîtrise des coûts et de la performance, ainsi que l'attention portée à la sobriété des ressources et à la capacité d'adaptation face aux variations d'activité.

Le rapport s'attachera enfin à rendre compte des effets concrets de cette démarche: réduction des délais de mise à disposition, meilleure stabilité lors des mises en service, diminution des interventions d'urgence, clarté accrue dans le suivi des applications, appropriation des bonnes pratiques par un plus grand nombre d'équipes. Il évoquera les limites rencontrées — qu'il s'agisse de contraintes de calendrier, de dépendances externes ou de la nécessaire montée en compétences — et proposera des pistes d'amélioration pour prolonger l'effort engagé: consolidation des standards, élargissement de la couverture des pratiques communes, approfondissement de la mesure, et poursuite de la sensibilisation des équipes.

En définitive, l'industrialisation et l'optimisation d'une usine logicielle orientée cloud chez GE HealthCare visent à concilier deux ambitions qui se renforcent mutuellement: gagner en vitesse et en fiabilité dans la livraison des logiciels, et élever le niveau de confiance, de sécurité et de

conformité attendu dans le domaine de la santé. Cette alternance m'a permis d'apporter une contribution concrète à cette trajectoire, d'en observer les bénéfices et d'en mesurer les exigences. Les constats et enseignements présentés dans ce rapport ont pour vocation d'éclairer la suite du chemin: consolider ce qui fonctionne, corriger ce qui doit l'être, et poursuivre une amélioration continue au service des patients, des professionnels et des équipes qui conçoivent les solutions de demain.

1. Problématique générale

Comment optimiser les pipelines CI/CD et les protocoles de test dans un contexte de cloud privé hospitalier, soumis à de fortes contraintes réglementaires et techniques, afin d'améliorer la fiabilité et la rapidité des déploiements logiciels ?

2. Contexte de l'alternance

2.1. L'entreprise, le service, le poste

GE HealthCare est une entreprise mondiale de technologie médicale, issue de l'héritage de General Electric et officiellement organisée comme entité dédiée à la santé depuis le début du 19e siècle. D'abord reconnue pour ses équipements d'imagerie médicale, notamment les premiers systèmes de radiographie, elle s'est imposée au fil des décennies comme un pionnier de l'innovation au service des patients et des professionnels de santé.

Aujourd'hui, GE HealthCare est présente dans plus de 160 pays et propose des solutions de pointe en imagerie, diagnostic, outils d'aide à la décision, intelligence artificielle et systèmes connectés pour les établissements de santé. Son ambition est de faciliter l'accès aux soins, d'améliorer la précision des diagnostics et d'optimiser les parcours de santé grâce à la technologie.

2.1.1. Structure d'accueil

GE HealthCare opère à l'échelle internationale, avec une histoire de plus d'un siècle au service des patients et des soignants. L'entreprise compte plus de 51 000 employés à travers le monde et s'affirme comme un acteur majeur du secteur, grâce à un portefeuille couvrant l'imagerie, les solutions numériques et les plateformes cloud dédiées à la santé.
En 2024, GE HealthCare a réalisé un chiffre d'affaires d'environ 18 milliards de dollars, témoignant de sa solidité économique et de sa capacité d'investissement dans des domaines clés tels que l'intelligence artificielle, le cloud médical et l'imagerie avancée.

Figure 1 - Carte des sieges Ge healthCare en France

2.1.2. Quelques chiffres et repères

- Présence dans plus de 160 pays.
- Plus de 51 000 collaborateurs.
- Environ 18 milliards USD de revenus annuels (2024).
- Partenariats technologiques avec de nombreux hôpitaux publics et privés à travers le monde.
- Investissements stratégiques dans l'IA, le cloud et les solutions logicielles pour la santé.

2.1.3. Le service et le poste

- Département dédié au développement d'applications médicales innovantes.
- Une équipe de plus de 200 développeurs impliqués dans la conception, la réalisation et l'évolution des solutions.
- Contribution à la création de logiciels permettant de visualiser et d'analyser des données médicales complexes afin de soutenir la prise de décision clinique.
- Intégration au sein de deux équipes complémentaires, avec des missions couvrant l'amélioration continue des produits, la qualité et la fiabilité des livraisons, ainsi que la collaboration avec les parties prenantes (produit, qualité, sécurité et opérations).

2.2. L’équipe associée – GE HealthCare

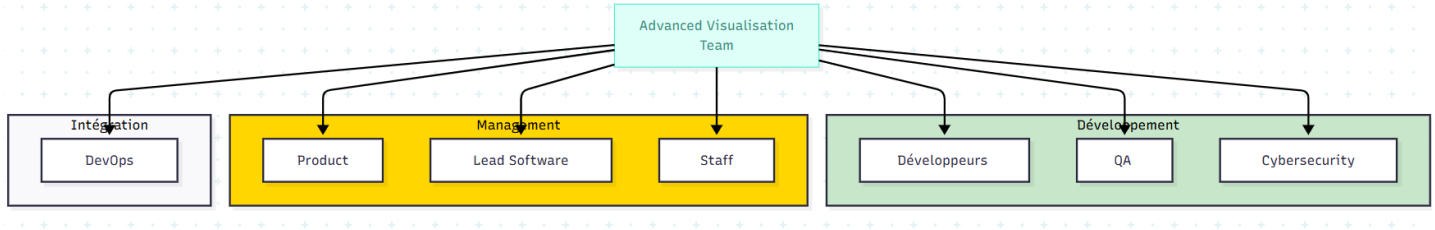


Figure 2 - Organigramme de l'équipe AV

J’ai été intégré à l’Advanced Visualisation (AV) Team, une équipe structurée autour de trois pôles très complémentaires: Integration (axé sur DevOps), Management (Product, Lead Software, Staff) et Development (Développeurs, QA, Cybersecurity). Cette organisation claire m’a immédiatement donné des repères, qui porte la vision, qui arbitre les priorités, qui réalise, qui vérifie, et qui veille à la sécurité. Dès mes premières semaines, j’ai pu circuler entre ces pôles, comprendre leur rôle, et surtout identifier la façon dont ils s’articulent au quotidien pour faire avancer nos produits.
Côté Integration, j’ai trouvé un point d’appui essentiel pour tout ce qui touche à la préparation et à la fluidité des livraisons. L’activité DevOps y joue un rôle de “courroie de transmission” entre l’idée et la mise à disposition ; ils installent les fondations, automatisent ce qui peut l’être, et cherchent à rendre les passages d’une étape à l’autre plus simples et plus fiables. Concrètement, c’est avec eux que j’ai réglé des questions très pratiques : comment organiser les enchaînements de construction et de livraison, comment réduire les points de friction, comment rendre nos enchaînements plus prédictibles. Leur approche m’a appris à regarder le flux de bout en bout, ce n’est pas seulement “faire” une fonctionnalité, c’est penser dès le départ à son trajet complet jusqu’aux utilisateurs, puis à sa vie une fois en service.

Le pôle Management réunit trois fonctions qui ont été mes repères pour cadrer les sujets. Avec le Product, j’ai travaillé sur la clarification des besoins et la priorisation. Ces échanges m’ont aidé à distinguer ce qui était “indispensable maintenant” de ce qui pouvait attendre, et à replacer chaque demande dans une logique de valeur pour les utilisateurs finaux. Le Lead Software m’a apporté un regard de cohérence, s’assurer que les orientations techniques restent alignées avec les choix

structurants de l'équipe, éviter les impasses, garder de la simplicité lorsqu'elle est possible. Enfin, le Staff a été le point d'orgue de l'organisation quotidienne, préparation des jalons, coordination entre équipes, gestion des dépendances. Grâce à ce trio, je savais à qui m'adresser selon la nature du sujet, la finalité avec le Product, la cohérence avec le Lead Software, le rythme et la coordination avec le Staff.

Le pôle Development rassemble les Développeurs, la QA et la Cybersecurity. Avec les Développeurs, les interactions étaient continuées, relectures, découpages de sujets, clarification des hypothèses, gestion des retours. L'ambiance était tournée vers la résolution concrète des problèmes. Chacun cherche à simplifier, à rendre le travail du suivant plus facile, à réduire le nombre d'allers-retours inutiles. La QA a joué un rôle décisif pour garder la qualité au centre, vérifications régulières, retours factuels, recherche d'une couverture de tests suffisante pour éviter les surprises lors des mises à disposition. De son côté, la Cybersecurity intervenait comme un garde-fou permanent: rappel des exigences du secteur santé, sensibilisation aux risques, validation des points sensibles. J'ai appris à intégrer leurs remarques tôt, pour éviter les corrections tardives qui coûtent plus cher et décalent les livraisons.

Au quotidien, cette organisation à trois pôles crée un rythme de travail lisible. Les sujets naissent et se priorisent avec le Product, s'alignent avec le Lead Software, s'organisent avec le Staff; puis ils transitent vers les Développeurs, passent par la QA, et sont accompagnés par l'équipe DevOps pour aller jusqu'en production; tout au long, la Cybersecurity garde un œil sur les zones sensibles. Dans ce circuit, mon rôle a été transversal: je me suis placé aux jonctions. J'ai clarifié des attentes entre Product et Développement, j'ai anticipé des points de coordination avec l'Integration, j'ai consolidé des retours de QA pour qu'ils soient actionnables par les Développeurs, et j'ai pris en compte les recommandations de Cybersecurity dès la conception des sujets, pour éviter les retouches de dernière minute.

Ce fonctionnement m'a aussi appris l'importance des "boucles courtes". Plutôt que d'attendre la fin d'un cycle pour découvrir les problèmes, nous cherchions à obtenir des retours précoces et réguliers : des démonstrations intermédiaires, des points d'avancement factuels, des essais en environnement contrôlé avant d'élargir la diffusion. Cette approche réduit la part d'incertitude et

évite les effets tunnel. Elle oblige également à travailler de manière plus modulaire : découper, livrer par étapes, apprendre de chaque incrément, corriger la trajectoire si nécessaire. Dans un contexte de santé, où la fiabilité et la clarté sont essentielles, cette discipline de petites étapes maîtrisées m'a semblé particulièrement précieuse.

La collaboration avec DevOps a été un fil conducteur de ma mission. Nous avons cherché ensemble à rendre les passages de relais plus fluides : mieux documenter ce qui est attendu à chaque étape, réduire les opérations manuelles, et rendre les enchaînements plus stables. Je retiens notamment l'intérêt de rendre visibles les "engorgements", lorsqu'un point bloque, le mettre au jour rapidement pour que chacun puisse s'ajuster. Cette transparence évite les tensions et permet de traiter les causes plutôt que les symptômes. J'ai aussi vu combien l'anticipation gagnait du temps, préparer tôt les environnements et les paramétrages, valider en avance les chemins de mise à disposition, vérifier les conditions de réussite avant d'engager une étape.
Avec le Product, j'ai appris à formuler des objectifs concrets et mesurables sans tomber dans la sur-spécification. L'idée est de savoir précisément ce que l'on cherche à améliorer (par exemple, la rapidité d'une mise à disposition ou la clarté d'un parcours utilisateur), tout en laissant aux équipes la liberté de trouver la meilleure manière d'y parvenir. Cette posture favorise l'initiative et évite de figer trop tôt des solutions qui pourraient se révéler inadaptées. Les discussions avec le Lead Software m'ont, de leur côté, permis d'adopter un regard plus global: comment une décision locale impacte le reste du système, quels compromis accepter maintenant pour garder des options ouvertes plus tard, où mettre l'effort pour obtenir le meilleur effet de levier.

La relation avec la QA m'a sensibilisé à une vérité simple: la qualité ne se "rajoute" pas à la fin, elle se construit dès le début. En intégrant leurs critères tôt, nous avons réduit le nombre d'allers-retours et gagné en sérénité sur la fin des cycles. Cela passe par des cas de vérification clairs, des critères d'acceptation bien compris, et un langage partagé entre ceux qui demandent, ceux qui réalisent et ceux qui vérifient. C'est aussi un état d'esprit: accepter la contradiction, remercier le signalement d'un problème, et traiter les retours comme une opportunité d'apprendre.

Les échanges avec la Cybersecurity m'ont rappelé l'exigence propre au domaine de la santé. Il ne s'agit pas seulement de faire fonctionner un logiciel; il faut le faire fonctionner de manière

responsable, en protégeant les données, en évitant les failles, et en respectant les cadres en vigueur. Leur présence dès la conception est rassurante: elle évite des oublis et des prises de risque involontaires. J'ai pris l'habitude de les consulter à chaque fois que nous touchions à une zone sensible, ou que nous introduisions un nouveau composant susceptible d'ouvrir une surface d'exposition.

Au fil des semaines, j'ai vu cette organisation produire des effets très concrets. Les sujets avançaient de manière plus régulière, les mises à disposition devenaient plus prévisibles, et les retours étaient mieux canalisés. La coordination entre les pôles nous a permis de réduire les urgences de dernière minute et de gagner en confiance lors des jalons importants. Surtout, chacun savait quel était son rôle et sa responsabilité. Le Product sur le "quoi" et le "pourquoi", le Lead Software sur le "comment" global, le Staff sur le "quand" et "avec qui", les Développeurs sur la réalisation, la QA sur la validation, la Cybersecurity sur la protection, et DevOps sur le passage à l'échelle et la continuité.

Enfin, je retiens la dimension humaine de cette équipe. Au-delà des schémas, ce qui fait la différence, ce sont les attitudes, l'écoute, la capacité à expliquer simplement, le respect des contraintes des autres, et la volonté de faire ensemble. J'ai bénéficié d'un accueil qui m'a permis d'être opérationnel rapidement, tout en ayant la liberté de proposer des améliorations. Cette confiance m'a encouragé à prendre des initiatives, clarifier un flux, simplifier une étape, ou structurer un retour pour le rendre plus utile. En résumé, l'Advanced Visualisation Team m'a offert un cadre solide et bienveillant pour apprendre, contribuer et progresser. Grâce à l'articulation entre Integration, Management et Development, j'ai pu mesurer concrètement ce que signifie "industrialiser" une démarche de livraison logicielle, donner de la cadence sans sacrifier la qualité, sécuriser les passages clés, et maintenir un cap commun au service des utilisateurs finaux. Cette expérience a renforcé ma conviction qu'une organisation claire, des rôles bien tenus et des boucles de feedback régulières sont les meilleurs atouts pour délivrer des logiciels fiables dans un environnement exigeant.

2.3. Vue d'Ensemble de l'Alternance

2.3.1. Contexte d'arrivée

Je viens d'intégrer GE HealthCare et je reçois mes premières missions. Je ne suis pas encore en mesure de tirer des conclusions, je découvre l'organisation, les priorités, les contraintes du secteur médical, et les modes de collaboration en place. Mon enjeu immédiat est d'entrer dans la culture de l'entreprise et d'adopter une posture DevOps utile dès le premier jour, sans brusquer les routines qui fonctionnent déjà.

Posture que j'adopte:

- Humilité active: je commence par écouter et observer. Je cherche à comprendre pourquoi les choses sont faites d'une certaine manière avant de proposer des ajustements.
- Orientation flux: j'essaie de regarder le chemin complet d'une demande, depuis l'expression du besoin jusqu'à l'usage réel, pour repérer où je peux aider à réduire les frictions.
- Petits pas visibles: je privilégie des améliorations modestes, concrètes et réversibles, qui apportent vite de la valeur sans créer d'effets de bord.
- Responsabilité partagée: ma mission n'est pas de "passer" un livrable à une autre équipe, mais de contribuer à ce qu'il aboutisse sereinement, avec les bons interlocuteurs.
- Transparence: je rends mon avancement lisible et je partage tôt mes questions et mes hypothèses, pour éviter les malentendus.

Ce que je cherche d'abord à comprendre

- Le "pourquoi" métier: à qui servent nos solutions, quels risques on veut réduire, quelles attentes non techniques comptent le plus (sécurité, conformité, qualité clinique).
- Les jalons réels: qui valide quoi, à quel moment, et avec quels critères explicites.
- Les points de friction: où ça attend, où ça revient en arrière, où l'on rediscute souvent les mêmes sujets.
- Les responsabilités: qui décide, qui exécute, qui opère, et comment on s'alerte mutuellement.
- Les rythmes: quels sont les rituels d'équipe (revues, planifications, points de suivi) et

comment je m'y insère sans les alourdir.

2.3.2. Mon plan d'installation – 30/60/90 jours

Jours 1–30: comprendre et cartographier

- Observer un cycle complet de "demande → mise à disposition → retour".
- Identifier 2 à 3 irritants concrets vécus par les équipes (ex: manque de clarté sur une étape, validations tardives, rôles flous).
- Tenir une "page personnelle" d'apprentissage : décisions importantes, critères implicites, acronymes, contacts clés.
- Produire une note courte "ce que j'ai compris / ce que je suppose / ce que je propose de tester" et la faire relire par un référent.

Jours 31–60: tester des améliorations légères

- Proposer une simplification de processus "sans regret" (par exemple, clarifier les critères d'entrée/sortie d'un jalon, ou une checklist partagée pour un passage sensible).
- Mettre en place un point régulier de synchronisation avec les interlocuteurs de mes missions pour anticiper blocages et dépendances.
- Aider à rendre plus visibles les informations utiles à tous (état d'avancement, points ouverts, décisions prises), au bon endroit, sans multiplier les supports.

Jours 61–90: ancrer et élargir

- Évaluer, avec les équipes, ce qui a réellement aidé : garder ce qui simplifie, retirer ce qui ne sert pas.
- Étendre prudemment une pratique qui a fait ses preuves (même cadre, autre équipe ou autre périmètre).
- Formaliser des repères de suivi peu coûteux : ce que l'on observe pour savoir si on

progresse (délais, reworks, sérénité des jalons).

2.3.3. Mes engagements personnels

- Clarté : dire ce que je fais, pourquoi je le fais, et quand je livrerai un premier résultat visible.
- Réversibilité : proposer des changements faciles à annuler si l'impact n'est pas celui attendu.
- Respect du cadre: intégrer d'emblée les exigences de sécurité, de confidentialité et de conformité propres au médical.
- Apprentissage continu: après chaque étape importante ou incident, capitaliser ce que j'ai appris dans un format court et réutilisable.

Ce que j'attends des équipes pour réussir

- Un parrain ou point de contact : pour valider mes compréhensions et orienter mes priorités.
- Des retours francs et rapides : pour ajuster le tir sans perdre de temps.
- La possibilité d'expérimenter à petite échelle: un périmètre "bac à sable" où tester une amélioration sans risque pour la production.
- Des critères explicites: savoir ce qui est "assez bon" pour franchir un jalon, afin d'éviter les débats de dernière minute.

Repères simples pour piloter ma progression

- Prévisibilité: moins d'écarts entre ce que l'on pense livrer et ce qui est réellement livré.
- Débit régulier: des petits lots qui avancent sans à-coups plutôt que des pics puis des creux.
- Moins de rework: une meilleure compréhension en amont réduit les retours en arrière.
- Sérénité aux jalons: rôles et critères clairs, moins d'urgence imprévue.
- Partage utile: ce que je produis (notes, checklists, synthèses) est court, trouvé facilement, et réutilisé par d'autres.

Risques identifiés et garde-fous

- Trop vouloir aller vite: je m'impose de valider mes hypothèses avec un référent avant d'impacter un processus partagé.
- Multiplier les documents: je centralise l'information au même endroit et je supprime les doublons.
- Sous-estimer les contraintes du médical: je consulte systématiquement les interlocuteurs qualité et sécurité avant tout changement de routine.
- Changement non accepté: je privilégie la démonstration sur un petit périmètre et je recueille des retours utilisateurs avant d'élargir.

2.3.4. Portée du travail et parties prenantes

Le travail réalisé au cours de cette alternance s'inscrit dans un programme d'innovation chez GE HealthCare, centré sur l'imagerie médicale assistée par l'intelligence artificielle. L'objectif est d'améliorer la qualité et la rapidité des parcours diagnostiques tout en renforçant la fiabilité, la traçabilité et la sécurité des services logiciels en contexte clinique. Ma contribution se situe dans une posture DevOps pragmatique, faire circuler la valeur de façon régulière, visible et sûre, en alignant les équipes et en protégeant les passages sensibles.
Le périmètre couvre les activités qui permettent à des fonctionnalités priorisées d'arriver jusqu'aux utilisateurs finaux sans à-coups: clarification du besoin et des critères d'acceptation; découpage en petits lots; synchronisation des jalons; intégration des points de contrôle qualité "au fil de l'eau"; prise en compte précoce des exigences de sécurité et de confidentialité; préparation des mises à disposition avec critères de décision explicites, plan de repli et traçabilité; recueil structuré des retours terrain pour nourrir l'amélioration continue. Restent hors périmètre direct: la recherche fondamentale en IA, la négociation commerciale avec les établissements et la gestion opérationnelle des infrastructures hospitalières (prises en compte comme contraintes, sans en avoir la responsabilité).

Les parties prenantes sont nombreuses et complémentaires. Côté établissements de santé: radiologues, manipulateurs en électroradiologie et cadres de santé expriment les besoins cliniques, valident l'adéquation au flux de soins et signalent les irritants; les DSI/RSSI veillent à l'intégration, à la sécurité des données et à la continuité de service; les comités qualité/éthique s'assurent de la

pertinence clinique et de la conformité. Côté GE HealthCare: le Product Management porte la vision et arbitre les priorités; le développement met en œuvre les fonctionnalités et corrige les anomalies; la fonction DevOps coordonne le flux de bout en bout, anticipe les passages sensibles et garantit la réversibilité; la cybersécurité prescrit les règles de protection et valide les mesures; la qualité/réglementaire aligne les pratiques sur les référentiels applicables et maintient l'évidence documentaire; le support/service capte la voix du terrain et boucle les retours. La R&D explore de nouvelles approches et transfère les connaissances vers les équipes produit et opérationnelles.
Pour éviter les zones grises, la répartition des responsabilités est clarifiée: le produit répond au "quoi/pourquoi", le développement au "comment fonctionnel", le DevOps au "comment on y va", la qualité au "ce qui est acceptable", la cybersécurité au "ce qui est sûr", le support au "ce qui est vécu", et les utilisateurs référents à "ce qui aide en clinique". La coordination s'appuie sur des rituels courts et utiles: points d'alignement hebdomadaires focalisés sur décisions et blocages; revues de jalons avec critères d'entrée/sortie; retours d'expérience sans blâme après incident avec quelques actions correctives suivies; une source unique et à jour pour décisions, responsabilités et états d'avancement.

L'environnement clinique impose des exigences non négociables: protection des données de santé (confidentialité, minimisation, droits d'accès, journalisation), sécurité des patients et qualité clinique, traçabilité et auditabilité des décisions, continuité de service et éthique (vigilance sur les biais, transparence sur les limites d'usage). Pour accélérer la valeur et réduire les retours en arrière, l'information utile à décider est rendue visible tôt (impacts, risques, dépendances), les critères d'acceptation sont co-construits avec produit/qualité et, lorsque pertinent, des utilisateurs référents; chaque passage sensible comporte une fenêtre de mise à disposition, un plan de repli et une communication claire.

Indicateurs recherchés: prévisibilité (écart réduit entre annoncé et livré), débit régulier (petits lots), baisse du rework (meilleurs critères d'entrée/sortie), sérénité aux jalons (rôles clairs, moins d'urgences) et adoption clinique mesurable via les retours utilisateurs et le support.
Les risques principaux et leurs garde-fous sont identifiés: complexification inutile (standards légers, réversibles, testés à petite échelle avant extension); sous-estimation des contraintes cliniques (implication régulière d'utilisateurs référents et de la qualité/cybersécurité); retour aux

silos (canaux unifiés, responsabilités explicites); documentation “pour l’audit” mais inutilisable (formats courts, à jour, orientés action et réemploi). Les hypothèses de réussite incluent la disponibilité des interlocuteurs clés pour des validations rapides, l’accès à un périmètre de test sans risque pour la production, une stabilité minimale des priorités sur des cycles courts et un engagement partagé à rendre visibles les arbitrages.

Dans ce cadre, mon rôle d’alternant DevOps est d’observer le système de bout en bout, cartographier les jalons et rendre visibles les frictions; proposer des améliorations modestes, concrètes et réversibles; aider à clarifier critères d’entrée/sortie et responsabilités; contribuer à une documentation utile et vivante; tenir un journal d’apprentissage et de décisions pour capitaliser. En synthèse, la portée de l’alternance est d’installer des habitudes qui augmentent clarté, prévisibilité et sécurité, pour transformer l’innovation en bénéfices concrets et sûrs pour les équipes et, in fine, pour les patients.

2.4. Écosystème DevOps

Au cours de mon alternance, j’ai évolué dans un écosystème exigeant où la valeur n’est réellement créée que lorsque les changements traversent, sans à-coups, l’ensemble de la chaîne d’environnements : DEV → INT → MET (Mise en Test/UAT) → PRÉ-PROD → PROD. Mon objectif a donc été de rendre ces “MET” robustes, représentatifs et sûrs, afin que les équipes puissent décider et livrer en confiance. La posture DevOps s’est traduite ici par des pratiques qui privilégient la lisibilité des flux, la traçabilité des décisions et la réversibilité des mises à disposition.

L’environnement technique s’appuie sur des briques ouvertes et industrielles. Pour l’orchestration des livraisons, Jenkins et GitLab CI sont utilisés de façon complémentaire, mais le cœur de mon travail a porté sur la qualité et la gouvernance des environnements non-prod (en particulier les MET), bien plus que sur les spécificités d’une plateforme CI donnée. Les applications sont conteneurisées (Docker) et déployées sur Kubernetes avec Helm; les artefacts (images, charts, paquets) sont gérés via Artifactory pour garantir traçabilité et reproductibilité. La qualité de code et la sécurité applicative sont suivies par Sonar (règles, debt, couverture), tandis que les tests end-to-end (Playwright) et la consolidation des résultats (ReportPortal) assurent une vision claire

de la stabilité fonctionnelle. L'infrastructure de tests et les serveurs de build tournent principalement sur Linux SUSE et sont gérés en IaC avec Ansible pour standardiser les configurations, durcir les accès (SSH) et assurer des déploiements répétables. Le pilotage opérationnel se fait via des tableaux de bord simples (métriques de pipelines, temps de cycle, taux de succès par environnement), adossés à des logs d'audit permettant de remonter la chaîne des responsabilités et des décisions.

Ce dispositif n'a de valeur que si les environnements MET restent fiables, proches de la production et cependant "sûrs" pour expérimenter. Concrètement, cela implique:

- Une stratégie claire de données de test: anonymisation ou jeux synthétiques, référentiels communs, procédures de rafraîchissement contrôlées, et séparation stricte des privilèges.
- Des critères d'entrée/sortie par environnement: ce qui est nécessaire pour promouvoir un change (qualité, sécurité, validation clinique), et ce qui permet de revenir en arrière sans friction.
- Des fenêtres de mise à disposition convenues et des plans de repli testés: les "go/no-go" sont argumentés, documentés et réversibles.
- Des configurations "comme en prod, mais pas en prod": parité des variables, secrets gérés de manière sûre, observabilité alignée (journalisation, traces, métriques) pour éviter les surprises tardives.
- Des tests qui suivent la progression: tests unitaires et de contrat tôt, end-to-end stables et rapides en MET, scénarios de non-régression et smoke tests à chaque promotion.
- Une documentation utile et vivante: topologies d'environnements, modes opératoires, critères de validation et journal des décisions, afin que chacun sache "où on en est" et "ce qui bloque".

Dans ce cadre, mon rôle a été d'industrialiser les passages sensibles, de simplifier ce qui peut l'être et de rendre visible ce qui compte pour décider. Les MET ont servi de point d'appui: c'est là que l'on gagne en confiance (ou que l'on perd du temps). J'ai donc concentré mes efforts sur la robustesse des MET, leur proximité fonctionnelle avec la prod, et la discipline des promotions.

2.5. Organisation et cadence de travail

- Les projets suivent une méthodologie Agile Scrum avec des sprints courts, des revues et

des rétrospectives systématiques. Cette cadence favorise la collaboration inter-équipes, l'adaptabilité aux retours du terrain et une livraison continue de valeur.

- Les environnements sont structurés en chaîne DEV → INT → MET (Mise en Test/UAT) → PRÉ-PROD → PROD, avec des critères d'entrée/sortie explicites, de la parité de configuration et des plans de repli testés. Les MET jouent un rôle clé: représentatifs de la production, ils permettent de valider fonctionnellement et cliniquement les changements avant promotion.

3. Etude et Analyse de l'etat existante

3.1. Etude de l'existant

GE HealthCare opère des plateformes logicielles déployées à l'échelle de plusieurs régions (UE/US/EMEA). Les solutions adressent des usages cliniques critiques — imagerie, workflow diagnostique, supervision et support aux décisions — qui imposent une disponibilité élevée, une conformité stricte et une résilience éprouvée. L'environnement est multi-entités (équipes produit, qualité, sécurité, opérations, cliniques), multi-réglementaire (RGPD/UE, HIPAA/US, exigences pays EMEA) et multi-environnements (DEV, INT, MET/UAT, PRÉ-PROD, PROD).

Dans ce contexte, chaque évolution logicielle doit franchir une chaîne d'intégration et de déploiement sécurisée, traçable et répétable. La promesse vis-à-vis des établissements de santé est double : délivrer de la valeur rapidement et garantir la sûreté de fonctionnement. Cela suppose:

- Des déploiements prévisibles et réversibles ;
- Des environnements de test (MET) proches de la production, avec des données anonymisées/synthétiques et des contrôles d'accès stricts ;
- Des mécanismes d'observabilité qui détectent tôt les dégradations et éclairent les décisions de "go/no-go";
- Une gouvernance qualité/sécurité intégrée à la chaîne de livraison, pour satisfaire audits et exigences cliniques.

3.1.1. Enjeux de disponibilité, conformité et résilience

- Conformité : intégrer automatiquement les contrôles (qualité code, sécurité, conformité IP,

chiffrement en transit/au repos, gestion des secrets) dans les pipelines; garantir la re-jouabilité des preuves pour audits.

- Résilience : déploiements avec des feature flags pour désactiver à chaud une capabilité, plans de repli testés, sauvegardes/restores validés, tests de chaos en environnement non-prod pour valider les hypothèses de récupération.

- Fiabiliser la production
    - Renforcer la parité → PRÉ-PROD → PROD pour réduire les écarts de comportement.
    - Mettre en place des stratégies de mise à disposition avec fenêtres, critères d'entrée/sortie, et plans de repli testés.
    - Diminuer le taux d'échec des changements et réduire l'impact utilisateur lors des mises à jour.
- Standardiser la CI/CD
    - Disposer d'un socle de pipelines réutilisable (templates, conventions, quality gates communs).
    - Harmoniser les nomenclatures de versions, tags, artefacts et environnements entre équipes et régions.
    - Automatiser la génération et la publication des notes de version et des artefacts signés.
- Automatiser (IaC)
    - Décrire serveurs et middlewares en Infrastructure as Code pour des déploiements reproductibles.
    - Durcir les configurations (SSH, packages, politiques de patching) et tracer les changements.
    - Réduire les "neiges techniques" en éliminant les écarts de configuration non documentés.
- Améliorer observabilité et qualité
    - Étendre la couverture des tests (unitaires, contrat, end-to-end) et stabiliser les suites.
    - Standardiser la collecte des métriques, logs et traces; créer des tableaux de bord décisions-prêts.

- Intégrer des seuils d'alerte pertinents, limiter le bruit et privilégier les signaux utiles au diagnostic.

3.1.2. Résultats attendus À l'issue de l'alternance, l'organisation dispose:

- D'un socle CI/CD standard réutilisable entre équipes et régions, avec conventions et quality gates partagés ;
- D'environnements MET/PRÉ-PROD représentatifs, observables et sûrs, accélérant les décisions de mise en production ;
- D'une IaC permettant des déploiements reproductibles, audités et durcis;
- De tableaux de bord utiles pour le reporting des tests;
- D'une trajectoire chiffrée sur la disponibilité, le temps de build et la réduction des vulnérabilités critiques.

4. Méthodologie et cadre théorique

4.1. Principes du DevOps appliqués à un environnement cloud privé

Au cœur de mon alternance, la question de l'adoption des méthodologies DevOps dans un contexte hautement réglementé comme celui de GE Healthcare s'est imposée comme un axe majeur de réflexion. Pour replacer ce travail dans son cadre, il est essentiel d'expliquer ce que recouvrent les principes du DevOps, puis de montrer en quoi leur application dans un environnement cloud privé sécurisé – tel qu'il est déployé dans l'écosystème de GE Healthcare – présente des particularités, des contraintes et aussi des leviers d'optimisation.

Le terme DevOps est la contraction de Development et Operations. Il désigne à la fois une culture, un ensemble de pratiques et un mouvement visant à rapprocher les équipes de développement logiciel et celles en charge de l'exploitation et de l'administration des systèmes. L'objectif est de réduire les silos organisationnels, d'accélérer le cycle de livraison et d'améliorer la qualité des logiciels produits. Concrètement, le DevOps s'appuie sur des notions clés comme l'intégration continue (CI, Continuous Integration), le déploiement continu (CD, Continuous Deployment/Delivery), l'Infrastructure as Code (IaC) ou encore la mise en place de chaînes

d’automatisation des tests et du monitoring.
Dans un contexte classique d’entreprise technologique tournée vers le grand public, ces pratiques reposent sur des infrastructures cloud public telles que Amazon Web Services (AWS), Microsoft Azure ou Google Cloud Platform (GCP). Ces plateformes offrent une grande flexibilité, une scalabilité quasi infinie, et des outils prêts à l’emploi pour supporter des pratiques DevOps avancées. Toutefois, dans le domaine médical et en particulier chez GE Healthcare, l’approche est différente : les logiciels produits concernent des données médicales sensibles (par exemple, les images issues d’examens de radiologie ou les dossiers patients). Leur hébergement et leur utilisation sont strictement encadrés par des normes internationales de protection des données (comme le RGPD – Règlement Général sur la Protection des Données, en Europe, ou la norme HIPAA – Health Insurance Portability and Accountability Act, aux États-Unis).

C’est dans ce contexte que GE Healthcare déploie ses solutions dans des environnements qualifiés de cloud privé. Contrairement à une idée reçue, ce cloud n’est pas totalement isolé d’Internet : il est bien connecté, mais au travers de portes d’accès sécurisées et contrôlées (firewalls, bastions, authentifications fortes, VPN d’entreprise). L’idée n’est pas de couper l’environnement des développeurs et des équipes de test du reste du monde, mais de reproduire les contraintes réelles des hôpitaux, où les machines finales destinées aux praticiens ne sont jamais exposées à Internet. Ainsi, les serveurs de développement, de pré-production et de production sont hébergés sur des infrastructures privées (parfois sur site, parfois dans des datacenters géographiquement éloignés), mais leur configuration interdit toute ouverture non maîtrisée vers l’extérieur. Cela permet de garantir que les applications conçues et testées seront compatibles avec le futur usage en milieu hospitalier, tout en maintenant un haut niveau de sécurité.

L’application des principes DevOps dans un environnement cloud privé repose sur plusieurs piliers.

L’intégration continue (CI)

L’intégration continue consiste à automatiser la fusion régulière du code source produit par différents développeurs dans une branche commune. Chaque modification déclenche l’exécution

de pipelines qui compilent, testent et valident la qualité du code. Dans un environnement cloud privé, cela suppose de disposer d'outils internes comme GitLab CI/CD, Jenkins, ou encore des runners personnalisés, déployés derrière les protections réseau de l'entreprise. L'accès à ces outils est limité aux collaborateurs autorisés, ce qui garantit la maîtrise des flux de données.

Le déploiement continu (CD)

Le déploiement continu vise à automatiser la mise en production des applications validées. Dans un cloud public, ce processus est généralement fluide grâce à des services managés. Dans un cloud privé, il est nécessaire d'intégrer des contraintes supplémentaires : segmentation des environnements (DEV, TEST, PRE-PROD, PROD), validation réglementaire, étapes de revue documentaire (comme la VRR – Verification Readiness Review). Cela allonge parfois le cycle, mais assure une conformité totale aux normes médicales.

L'Infrastructure as Code (IaC)

L'Infrastructure as Code consiste à décrire l'infrastructure (serveurs, réseaux, permissions, etc.) sous forme de fichiers de configuration versionnés, par exemple via Terraform ou Ansible. Cela permet de reproduire de manière identique un environnement sur différents sites, ce qui est essentiel dans un cadre international comme celui de GE Healthcare. L'approche IaC offre aussi une traçabilité précieuse pour les audits.

L'automatisation des tests

Le DevOps repose sur l'idée que les tests doivent être intégrés tout au long du cycle de vie logiciel. Dans mon expérience, cela inclut les tests unitaires, les tests d'intégration, et surtout les tests end-to-end (E2E) qui simulent le comportement réel d'un utilisateur final. En environnement cloud privé, l'automatisation des tests doit composer avec des infrastructures parfois hétérogènes et segmentées, ce qui rend la maintenance des suites de tests particulièrement exigeante.

Le monitoring et le reporting

Enfin, le DevOps implique un suivi continu de la performance et de la disponibilité des systèmes. Dans un cloud privé, le monitoring doit se faire avec des outils internes, compatibles avec les contraintes de confidentialité (ex. Prometheus, Grafana, ou des solutions développées en interne). Le reporting, quant à lui, doit fournir des indicateurs clairs, exploitables à la fois par les équipes techniques et par les auditeurs externes.

Les contraintes propres au secteur médical

La particularité du domaine médical impose d'adapter les pratiques DevOps. Là où une start-up web classique peut se permettre des déploiements plusieurs fois par jour, GE Healthcare doit tenir compte :

- des contraintes réglementaires (auditabilité, certification de chaque version logicielle),
- de la confidentialité absolue des données de santé,
- de la nécessité de reproduire l'environnement hospitalier dans les phases de test et de pré-production.

Ainsi, chaque pipeline CI/CD ne se contente pas d'automatiser une construction logicielle : il doit aussi générer des preuves de conformité (rapports de tests, artefacts signés, journaux d'exécution conservés). Ces éléments sont ensuite intégrés dans les dossiers réglementaires exigés par les organismes de contrôle.

Ce que signifie "adopter la vision DevOps" ici et maintenant

- Penser système : ce n'est pas "mon" bout de chaîne, c'est l'ensemble du parcours jusqu'à l'usage réel.
- Raccourcir les boucles de feedback : voir plus tôt, corriger plus tôt, avec des impacts maîtrisés.
- Rendre le travail visible: l'information circule, on décide sur des faits et l'on sait quoi faire en cas d'imprévu.

- Protéger la qualité: mieux préparer les passages sensibles, prévoir le repli, et garder la traçabilité des décisions.

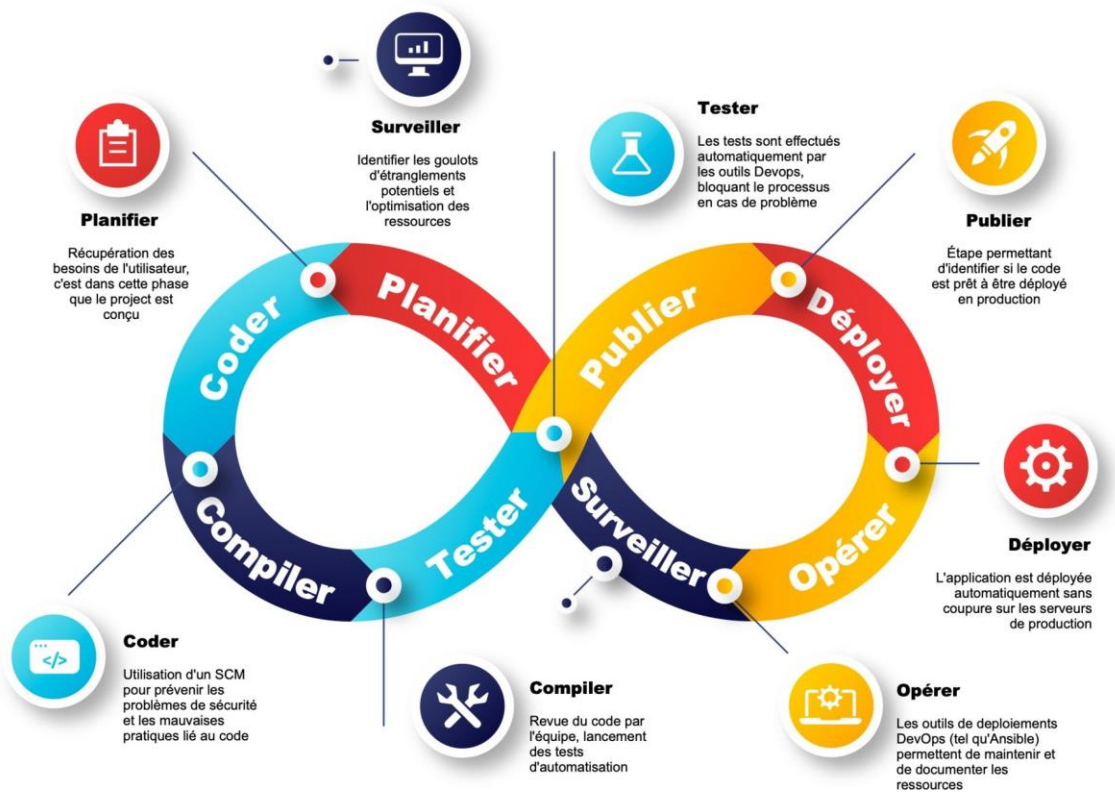


Figure 3 - Cycle de Vie DevOps

4.2. Méthodologies utilisées (NetOps, IaC, CI/CD, SRE)

Au cours de mon alternance, plusieurs méthodologies modernes ont été mobilisées afin de structurer, automatiser et fiabiliser le cycle de vie logiciel dans un environnement cloud privé. Ces approches, bien que distinctes, s'articulent autour d'un objectif commun : réduire la complexité opérationnelle et améliorer la qualité ainsi que la traçabilité des déploiements.

La première approche est le NetOps (Network Operations), qui applique les principes de

l'automatisation et de l'Infrastructure as Code à la gestion des réseaux. Dans un environnement fortement segmenté comme celui de GE Healthcare, marqué par des zones cloisonnées (développement, test, pré-production, production), NetOps permet de standardiser la configuration des réseaux, de garantir leur cohérence et d'assurer un haut niveau de sécurité. L'usage de scripts et de modèles versionnés remplace les interventions manuelles, réduisant les risques d'erreurs humaines et facilitant les audits.

L'Infrastructure as Code (IaC) constitue un second pilier méthodologique. L'IaC consiste à décrire l'infrastructure (machines virtuelles, conteneurs, règles réseau) sous forme de fichiers de configuration versionnés. Dans mon cas, des outils comme Terraform et Ansible ont permis de reproduire à l'identique différents environnements (DEV, TEST, PRE-PROD), tout en conservant une traçabilité complète des changements. L'IaC offre également une rapidité d'exécution : le déploiement d'un cluster Kubernetes ou d'un serveur applicatif peut se faire en quelques minutes, là où une configuration manuelle aurait pris plusieurs heures.

La méthodologie CI/CD (Continuous Integration / Continuous Deployment), cœur de l'approche DevOps, a été essentielle dans l'automatisation des tests et du déploiement. Chaque modification du code déclenche une suite de jobs automatisés : compilation, tests unitaires, tests end-to-end (E2E), génération d'artefacts, puis, le cas échéant, mise à disposition des images Docker sur les serveurs internes. Cette approche réduit les temps de mise en production et assure une détection rapide des anomalies. Dans le cadre médical, cette CI/CD doit cependant être enrichie de mécanismes de traçabilité et de revue documentaire afin de répondre aux exigences de conformité.

Enfin, la philosophie du Site Reliability Engineering (SRE) a également inspiré certaines pratiques. Introduite par Google, l'approche SRE consiste à appliquer une discipline d'ingénierie logicielle à la gestion des systèmes en production, avec une recherche d'équilibre entre fiabilité et rapidité d'innovation. Dans mon alternance, cela s'est traduit par la mise en place d'indicateurs de performance (SLO – Service Level Objectives, SLA – Service Level Agreements) et par un suivi constant via des outils de monitoring comme Grafana et Prometheus. L'objectif est double : garantir la disponibilité des services critiques, tout en permettant aux équipes de continuer à livrer des évolutions régulières.

Ces méthodologies, combinées, forment une base solide qui permet non seulement d'améliorer l'efficacité technique, mais aussi de répondre aux contraintes réglementaires et organisationnelles propres au domaine médical. Elles traduisent la volonté de GE Healthcare de s'inscrire dans une dynamique moderne de production logicielle, sans compromettre la sécurité ni la conformité.

4.3. Bonnes pratiques de QA et observabilité dans les environnements réglementés

Dans un contexte réglementé tel que celui de la santé, la Quality Assurance (QA) ne se limite pas à valider la conformité fonctionnelle d'un logiciel. Elle s'étend à l'ensemble du cycle de développement et de déploiement, avec pour objectif d'apporter des preuves tangibles de qualité et de fiabilité, tout en respectant les standards imposés par les organismes de certification (par exemple, la norme ISO 13485 ou les exigences européennes en matière de dispositifs médicaux).

La première bonne pratique consiste à mettre en place une traçabilité complète. Chaque test exécuté (unitaires, intégration, end-to-end) doit générer des rapports horodatés, archivés et associés à la version du code testée. L'usage de frameworks comme JUnit, Playwright ou pytest, combiné à des plateformes de reporting telles que ReportPortal, garantit une vision claire de la couverture de tests et des éventuelles régressions.

Ensuite, la standardisation des environnements de test est essentielle. Dans les environnements réglementés, il ne suffit pas que le code fonctionne en développement : il doit se comporter de manière identique en test, en pré-production et en production. L'approche Infrastructure as Code (IaC), via des outils comme Ansible ou Terraform, permet de décrire et de répliquer fidèlement chaque environnement, limitant ainsi les écarts de comportement et réduisant les risques de non-conformité.

Une autre dimension incontournable est l'observabilité. Contrairement à la simple supervision, l'observabilité vise à comprendre le « pourquoi » d'un dysfonctionnement et non seulement le « quoi ». Cela repose sur trois piliers :

Logs (suivi détaillé des événements applicatifs et systèmes),

Metrics (indicateurs chiffrés sur la performance, comme le temps de réponse moyen ou l'utilisation CPU/mémoire),

Traces (corrélation des appels distribués dans des architectures complexes).

Des solutions comme Prometheus, Grafana ou Elastic Stack (ELK) permettent de centraliser et visualiser ces données. Dans un cadre réglementé, ces tableaux de bord deviennent également des preuves documentées à présenter lors des audits, attestant de la stabilité et de la disponibilité des systèmes.

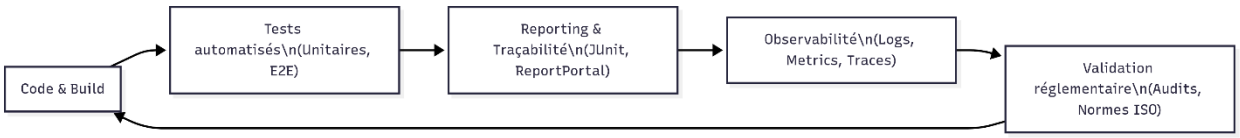


Figure 4 - Cycle QA dans un environnement DevOps réglementé

Enfin, une bonne pratique est l'intégration de la QA et de l'observabilité dans le pipeline CI/CD. Les tests automatisés, les seuils de performance, et les rapports de conformité doivent être générés de manière continue, garantissant que chaque itération logicielle est validée non seulement techniquement, mais aussi réglementairement.

Ces pratiques combinées assurent non seulement la robustesse des applications déployées dans les hôpitaux, mais renforcent également la confiance des équipes médicales et des auditeurs externes dans la fiabilité des logiciels livrés.

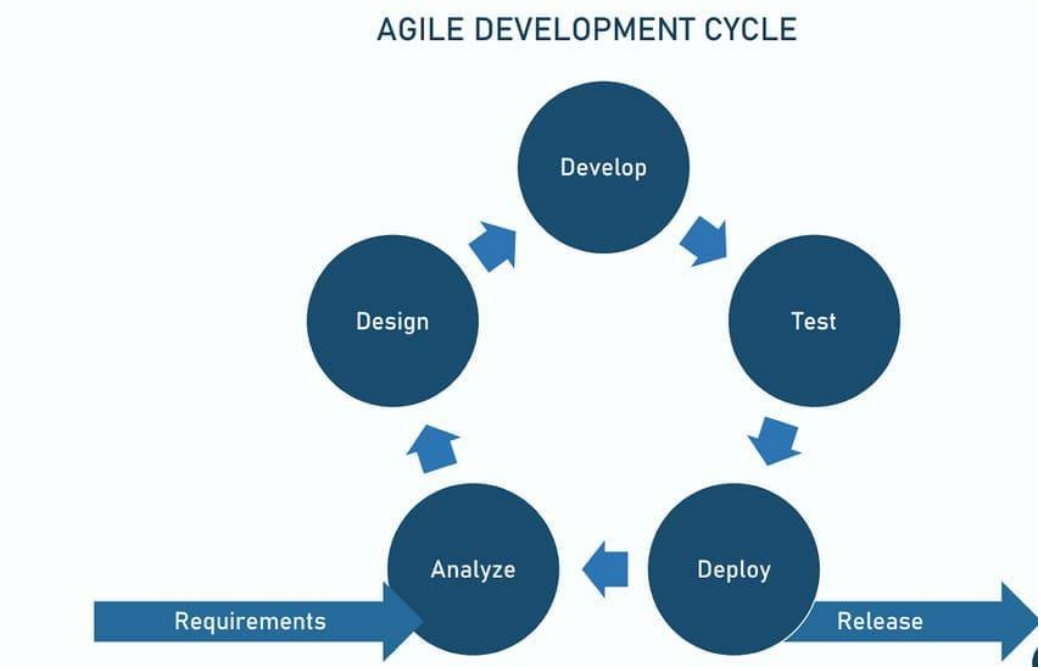


Figure 5 - Quality Assurance (QA), Quality Control and Testing

5. Mise en œuvre des solutions et Réponse à la problématique

Pour répondre à la problématique de ce mémoire, qui consistait à optimiser les protocoles de test et les pipelines CI/CD dans un environnement de cloud privé hospitalier soumis à de fortes contraintes réglementaires, j'ai suivi une démarche progressive et structurée. Mon objectif principal était de rendre les déploiements plus fiables, reproductibles et efficaces, tout en respectant les exigences strictes liées aux données de santé et à la sécurité des systèmes, indispensables dans le secteur médical.

La première étape a été de **prendre du recul sur l'existant**. J'ai analysé les pipelines et les processus en place pour identifier les points de faiblesse, les redondances et les pratiques pouvant ralentir les cycles de développement ou affecter la qualité des livraisons. Cette analyse m'a permis de prioriser les actions à entreprendre et de construire un plan cohérent d'optimisation.

Ensuite, j'ai travaillé sur la **standardisation et l'optimisation des pipelines de livraison** afin de créer une architecture homogène et plus facile à maintenir. Parallèlement, j'ai amélioré la **conteneurisation des applications et les charts Helm**, pour garantir que chaque build puisse être déployé de manière fiable et reproductible sur différents environnements.

L'**industrialisation du cycle de release** a été une étape essentielle. J'ai mis en place l'automatisation de la génération des ISO, la gestion des Release Candidates et la publication des artefacts. L'objectif était de réduire les interventions manuelles, limiter le risque d'erreur et accélérer la mise en production.

La **qualité logicielle et les tests end-to-end** ont également été au centre de mon travail. J'ai refondu les pipelines Jenkins, créé des workflows dédiés et développé/corrigé des tests Playwright pour fiabiliser les campagnes de QA. Ces actions ont permis d'améliorer la couverture des tests, de garantir leur répétabilité et de renforcer la robustesse globale des applications.

Dans le même temps, j'ai appliqué les principes du **Site Reliability Engineering (SRE)** et automatisé certaines tâches systèmes avec Ansible. J'ai notamment travaillé sur le durcissement des systèmes Linux SUSE, la sécurisation SSH et l'optimisation des déploiements, afin d'améliorer la stabilité et la sécurité des environnements.

Enfin, j'ai mis l'accent sur **l'observabilité et le reporting**, en créant des dashboards et indicateurs permettant de suivre précisément l'avancement des tests et la performance des pipelines. J'ai aussi automatisé certaines tâches de maintenance pour augmenter la productivité et rendre les processus plus fluides.

Cette démarche m'a permis de transformer un environnement complexe et hétérogène en un système structuré, fiable et conforme aux exigences réglementaires. Les sections suivantes détaillent chacune de ces actions, en expliquant les choix techniques effectués et les bénéfices observés.

5.1. Standardiser et optimiser les pipelines de livraison. Mise en place d'une architecture de pipelines homogènes pour réduire les duplications, améliorer la maintenabilité et accélérer les déploiements.Présentation des outils et notions clés

Mon alternance au sein de GE HealthCare m'a permis d'évoluer dans un environnement complexe et international, où la qualité logicielle et la rigueur des processus sont essentielles. Avant de présenter le travail réalisé, il est important de détailler les principaux outils et concepts qui ont structuré mes missions.

L'automatisation des processus de développement et de déploiement s'appuie largement sur des pipelines CI/CD, orchestrés principalement à l'aide de Jenkins et GitLab CI. Jenkins, écrit en Java et extensible via de nombreux plugins, est l'outil central qui orchestre les étapes de build, de tests et de déploiement. Il repose sur des scripts en Groovy, permettant une flexibilité et une intégration fine avec les projets. En parallèle, GitLab CI, défini en YAML, est utilisé pour certains projets spécifiques, notamment ceux nécessitant des runners internes pour exécuter les pipelines dans des environnements isolés. Ces runners, généralement déployés dans des conteneurs Docker, garantissent une reproductibilité et une isolation des processus.

Le concept de pipeline CI/CD est au cœur de notre travail : il s'agit d'automatiser la construction, les tests et la mise en production des applications. Chaque étape est scriptée, versionnée et validée via un système de branches Git et de merge requests, garantissant traçabilité et qualité. L'architecture logicielle repose sur un cluster Kubernetes, qui héberge à la fois les environnements de préproduction et certains environnements de test. Kubernetes permet de gérer le déploiement, la scalabilité et la résilience des applications, grâce à des fichiers de configuration YAML définissant les services, pods, quotas et probes.

Enfin, l'approche DevOps dans laquelle j'ai travaillé repose sur l'intégration continue, le déploiement continu et une étroite collaboration entre les équipes de développement et d'exploitation. Dans cet environnement, la notion d'agent Jenkins est essentielle : il s'agit d'une machine, physique ou virtuelle, ou d'un conteneur capable d'exécuter les jobs définis dans les pipelines. Chaque agent est configuré pour supporter des environnements spécifiques (Linux, Windows, Docker) et garantir la compatibilité des builds.

Ces notions constituent le socle des missions qui m'ont été confiées. Elles m'ont permis de participer activement à la standardisation et à l'industrialisation des processus de livraison logicielle dans un cadre exigeant, marqué par la dimension critique des produits de GE HealthCare.

La chaîne de livraison est automatisée via GitLab CI/CD connecté à Jenkins. Chaque commit déclenche la construction, l'exécution des tests et le déploiement vers les environnements d'intégration puis de recette, avec traçabilité des artefacts, gestion des secrets et contrôles qualité (linting, tests automatiques). Cette organisation permet d'itérer rapidement tout en respectant les exigences de fiabilité attendues pour une application de visualisation d'imagerie médicale.

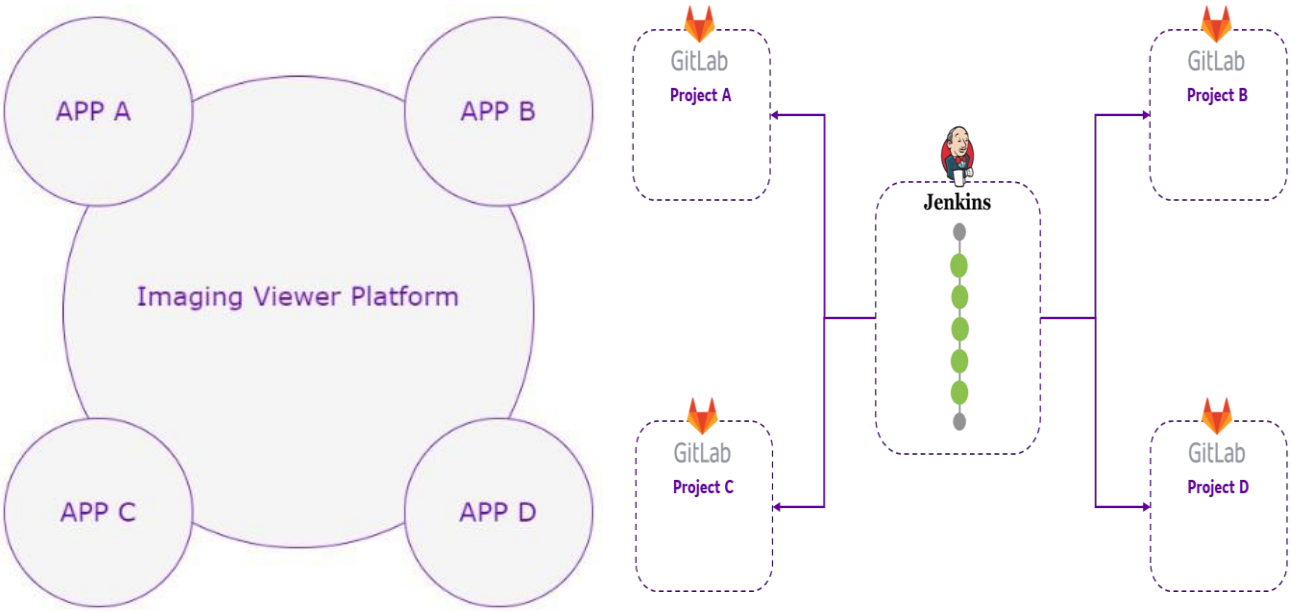


Figure 6 - Application Principale

5.1.1. Contexte et objectifs

Ma mission principale consistait à travailler sur l'architecture et la standardisation des pipelines de livraison logicielle, déployés sur plusieurs régions du globe, notamment l'Europe, les États-Unis et l'EMEA. L'enjeu majeur était d'aligner les environnements de test et de préproduction, afin de garantir une parité stricte et une fiabilité maximale avant toute mise en production. Cette parité concerne non seulement le code applicatif, mais également les configurations, les secrets, les variables d'environnement et les fenêtres de disponibilité des systèmes.
Dans cet environnement international, chaque région possédait des contraintes spécifiques, mais il était indispensable d'instaurer des standards communs. Le but était de réduire les erreurs de déploiement, d'accélérer le processus de validation et de permettre une maintenance simplifiée. Les pipelines devaient répondre à des critères de promotion clairs, validés par des parties prenantes issues des équipes de développement, de qualité et d'exploitation.

5.1.2. Méthodologie de travail et intégration des exigences

Le cœur de mon travail consistait à intégrer de nouvelles exigences fonctionnelles et techniques au sein des pipelines. Cela passait par un processus structuré, allant de l'analyse du besoin à la validation finale par les parties prenantes.

Lorsqu'un projet nécessitait une modification, nous commencions par cloner le dépôt Git contenant le code applicatif et ses pipelines associés. Par exemple, une application web développée en TypeScript et dotée de nombreuses dépendances pouvait présenter un problème nécessitant une refonte partielle du pipeline. Après avoir analysé le projet en local et identifié les points de blocage, nous rédigions une proposition de modification.
Cette modification passait par la création d'une nouvelle branche Git, sur laquelle nous développions les ajustements nécessaires. Pour Jenkins, cela impliquait de modifier ou de créer des jobs en Groovy, tandis que pour GitLab CI, les modifications se faisaient directement dans les fichiers YAML. Une fois les ajustements apportés, un job de test était mis en place afin de vérifier la stabilité de la nouvelle configuration et d'éviter toute régression.
Chaque pipeline modifié était testé dans un environnement isolé, souvent à l'aide d'un runner Docker. Si les résultats étaient positifs, nous soumettions une merge request. Cette dernière était soumise à validation par les parties prenantes, garantissant que la modification respectait les standards de l'organisation et répondait à la demande initiale.

5.1.3. Jenkins

Rôle : Orchestrateur CI/CD polyvalent pour construire, tester et déployer sur plusieurs environnements (DEV → INT → MET → PRÉ-PROD → PROD).

Pourquoi important : Forte extensibilité (plugins), programmabilité (pipelines déclaratifs/scripted), gestion fine des files/agents pour absorber la charge multi-projets.

Usages typiques : Pipelines multi-branches, fan-out des tests, promotion d'artefacts, fenêtres de mise à disposition, déclenchements conditionnels par étiquettes/flags.

Indicateurs : Durée médiane de build, taux de succès par étape, temps d'attente en queue, change failure rate.

Bonnes pratiques : Pipelines "as code", isolation par agents conteneurisés, secrets via credentials store, steps idempotents.

5.1.4. Création et gestion des jobs Jenkins

La création de jobs Jenkins constituait une tâche essentielle dans ce processus. Jenkins est organisé autour d'agents et de contrôleurs : le contrôleur gère la configuration des jobs, tandis que les agents exécutent les étapes définies. Pour créer un nouveau job, il faut tout d'abord définir le type de projet (pipeline, freestyle, multibranch) et configurer le dépôt Git associé. Ensuite, le script Groovy, stocké dans un Jenkinsfile, précise les étapes d'exécution : build, tests, packaging et déploiement.
Dans mon travail, la création de jobs Jenkins impliquait une bonne compréhension des dépendances logicielles et des environnements cibles. Les agents Docker étaient souvent utilisés pour garantir que chaque exécution se faisait dans un environnement propre et reproductible. Ce processus renforçait la fiabilité des pipelines et permettait une intégration fluide de nouveaux

projets.

5.1.5. Intégration dans l'écosystème Kubernetes

L'intégration des pipelines au sein d'un cluster Kubernetes a été un élément clé de l'industrialisation. Les environnements de test et de préproduction étaient orchestrés par Kubernetes, permettant une flexibilité importante dans le déploiement des applications. Les pipelines déclenchaient des déploiements automatisés, utilisant des manifests YAML pour définir les ressources, les quotas et les stratégies de redémarrage des pods.
Cette approche a permis de mettre en place une véritable usine logicielle cloud-native, où chaque étape, de la construction du code au déploiement, était automatisée et tracée. Kubernetes offrait également des outils de monitoring et de debugging (kubectl, htop, logs pods), essentiels pour diagnostiquer rapidement les problèmes.

5.1.6. Bilan et compétences acquises

Ce projet d'industrialisation et de standardisation des pipelines m'a permis d'acquérir une expertise approfondie sur l'automatisation des processus de développement. J'ai appris à travailler dans un contexte international et à collaborer avec des équipes pluridisciplinaires, tout en respectant des standards stricts.
J'ai développé des compétences solides en Groovy et en configuration Jenkins, ainsi qu'une bonne maîtrise de GitLab CI et des runners Docker. La compréhension des environnements Kubernetes et la gestion des déploiements dans un cluster complexe ont été des éléments essentiels de cette expérience.
Ce travail m'a également permis d'améliorer mes compétences en diagnostic, en résolution de problèmes et en gestion de projets techniques. Enfin, cette mission m'a sensibilisé à l'importance de la traçabilité, de la validation rigoureuse des pipelines et de la documentation, des aspects essentiels dans le domaine médical où chaque déploiement peut avoir des impacts critiques.

5.2. Amélioration des charts Helm et conteneurisation d'applications avec publication des images Docker

5.2.1. Présentation des notions et des outils

Dans le cadre d'une industrialisation cloud-native, la gestion des applications conteneurisées et leur déploiement automatisé constituent des piliers essentiels de la chaîne DevOps. Helm est utilisé comme gestionnaire de packages pour Kubernetes, permettant de définir, installer et mettre à jour des applications via des charts standardisés. Ces charts regroupent l'ensemble des fichiers de configuration nécessaires au déploiement d'une application, assurant ainsi une reproductibilité des environnements.

- Kubernetes

Rôle : Orchestrateur de conteneurs pour déployer et scaler les workloads; support des déploiements

progressifs.

Pourquoi important : Haute disponibilité, auto-rétablissement, policy réseau et secrets gérés, déploiements.

Usages typiques : Espaces de noms par environnement (dont MET), stratégies de rollout/rollback, autoscaling, NetworkPolicies pour cloisonner.

Indicateurs : SLO par service, taux d'échec de pods, temps de rollout, saturation CPU/RAM.

Bonnes pratiques : Manifests versionnés, probes de liveness/readiness, ressources/quotas, RBAC minimaliste.

Docker est la technologie de conteneurisation utilisée pour encapsuler les applications et leurs dépendances, garantissant un fonctionnement identique entre environnements de développement, de test (MET), de pré-production et de production. Ces images, versionnées et stockées dans un registre centralisé (Artifactory), offrent une traçabilité complète des artefacts déployés, facilitant les audits et réduisant les risques liés aux différences de configuration entre environnements.
Artifactory joue ici un rôle fondamental : il centralise la gestion des images Docker, assure la traçabilité des versions et permet de contrôler la promotion des artefacts selon des critères précis. Ce processus favorise une gouvernance rigoureuse des déploiements et un suivi optimal des évolutions logicielles.
L'écosystème technique de ce projet repose donc sur Kubernetes pour l'orchestration des conteneurs, Helm pour la gestion des déploiements, Docker pour la conteneurisation, et Artifactory pour la traçabilité et la sécurité des artefacts. L'automatisation de bout en bout est assurée par des pipelines CI/CD (Jenkins ou GitLab CI), qui orchestrent la construction, le test et la mise à disposition des nouvelles versions applicatives.

5.2.2. Contexte du projet

GE HealthCare développe et maintient des solutions logicielles critiques dédiées à l'imagerie médicale. Dans ce contexte, les environnements techniques doivent être stables, reproductibles et sécurisés, afin de garantir la conformité réglementaire et la continuité de service. Les équipes Advanced Visualisation, au sein desquelles j'évoluais, étaient particulièrement confrontées à des problématiques de gestion des versions de logiciels, liées notamment à la dépréciation de certaines bibliothèques ou images Docker, ou à des incompatibilités introduites par de nouvelles fonctionnalités.
La nécessité de maintenir des environnements homogènes (développement, MET, pré-production et production) impliquait une gouvernance stricte des pipelines et des artefacts déployés. Chaque nouvelle version logicielle devait être intégrée dans un processus maîtrisé, avec une traçabilité complète, afin d'éviter toute divergence entre environnements. Ce projet s'inscrivait également dans une volonté de renforcer la standardisation des configurations déployées à travers des charts Helm uniformisés et documentés, et d'optimiser la conteneurisation pour répondre aux exigences de performance et de sécurité de la plateforme.

5.2.3. Réalisations techniques

Mon rôle a consisté à intervenir sur tout le cycle de vie des charts Helm et des images Docker utilisées pour nos solutions logicielles. L'objectif principal était de mettre en place et de maintenir une usine logicielle capable de déployer rapidement et efficacement des applications mises à jour, tout en garantissant la reproductibilité et la traçabilité.
Le travail débutait souvent par l'identification d'une incompatibilité ou d'une dépréciation de composant. Un exemple concret fut l'upgrade d'une version de Nginx utilisée dans l'une des applications déployées sur Kubernetes. Cette opération impliquait une série d'étapes rigoureuses :

- Analyse des dépendances associées à l'image concernée, vérification des changements apportés par la nouvelle version (notamment les breaking changes) et identification des fichiers de configuration impactés.
- Modification des charts Helm afin d'adapter les métadonnées et les valeurs utilisées par les templates YAML. Cette étape nécessitait une parfaite compréhension de la structure des charts et des pratiques recommandées par Helm pour assurer une compatibilité ascendante.
- Mise à jour des configurations Docker associées et reconstruction des images impactées. Ces images étaient ensuite taguées et poussées dans Artifactory, où elles étaient documentées pour garantir leur traçabilité.
- Test des nouvelles configurations dans un environnement de développement dédié. Kubernetes servait de plateforme pour valider le comportement des nouveaux pods, avec des tests fonctionnels automatisés intégrés aux pipelines.
- Une fois validée, la mise à jour faisait l'objet d'une merge request soumise à l'équipe de revue, puis intégrée dans les branches principales pour être déployée sur les environnements MET et pré-production.

Ce cycle de travail impliquait également l'adaptation et le maintien des pipelines CI/CD. Les jobs Jenkins et GitLab CI étaient configurés pour intégrer automatiquement les nouvelles images dans les déploiements Kubernetes, avec des étapes de validation (linting des charts, tests unitaires et fonctionnels) pour minimiser les risques de régression.
Grâce à cette démarche, l'équipe a progressivement amélioré la robustesse et la cohérence des environnements, tout en accélérant le temps de mise en production des nouvelles versions logicielles.

5.2.4. Analyse et enseignements

Ce projet m'a permis de consolider mes compétences en déploiement cloud-native et en gestion des environnements conteneurisés. J'ai acquis une expertise pratique dans la conception et le maintien de charts Helm complexes, ainsi que dans la gestion des images Docker et des pipelines CI/CD multi-environnements.
Le travail sur Artifactory m'a permis de comprendre en profondeur les enjeux de traçabilité et de gouvernance des artefacts logiciels, essentiels dans un contexte industriel où la conformité réglementaire et la reproductibilité sont des exigences incontournables. L'utilisation de Kubernetes comme plateforme de validation a renforcé ma maîtrise des concepts d'orchestration, de déploiement déclaratif et de scalabilité.
Au-delà des aspects techniques, ce projet m'a confronté aux enjeux organisationnels et à la collaboration inter-équipes. Travailler en méthode Agile Scrum au sein d'une équipe

pluridisciplinaire m'a permis de développer une capacité d'analyse rapide et une rigueur méthodologique indispensable pour traiter des incidents complexes, intégrer des changements majeurs et déployer des solutions fiables en production.
Ce retour d'expérience constitue une base solide pour aborder des problématiques de plus grande envergure liées à la standardisation des environnements, à l'automatisation des déploiements et à la mise en place d'usines logicielles résilientes dans le domaine du DevOps.

5.3. Industrialisation du cycle de release : génération d'ISO, gestion des Release Candidates et automatisation des publications

5.3.1. Présentation des notions clés

Dans le cadre de mon alternance en tant qu'ingénieur DevOps chez GE HealthCare, j'ai eu l'opportunité de travailler sur l'industrialisation complète du cycle de release de notre usine logicielle cloud-native. Cette mission s'inscrivait dans une démarche de modernisation et de fiabilisation des processus de livraison logicielle, afin de répondre à des enjeux de qualité, de rapidité et de traçabilité essentiels dans un environnement critique et fortement réglementé comme celui du secteur médical.
L'industrialisation du cycle de release consistait à automatiser et fiabiliser plusieurs étapes clés du développement logiciel :

- La génération d'images ISO et de paquets signés, prêts à être déployés dans des environnements de test et de production.
- La gestion des différentes versions logicielles : versions Alpha et Beta, Release Candidates (RC) et versions stables.
- La mise en place d'un **versioning sémantique** clair, garantissant une compréhension immédiate des changements apportés.
- L'intégration de **feature flags** permettant d'activer ou désactiver des fonctionnalités en production (ou en MET – Milieu d'Exploitation Technique) sans impacter les utilisateurs finaux.
- La génération et publication automatisée des **release notes**, intégrant des marquages explicites (ATD, RC, BETA) pour faciliter leur lecture par les équipes techniques et métiers.

Ces notions ont nécessité une maîtrise poussée des concepts DevOps, des outils d'automatisation tels que Jenkins et GitLab CI/CD, ainsi qu'une compréhension approfondie des API REST pour interagir avec nos outils de gestion de code source, de pipelines et de déploiement.

5.3.2. Contexte du projet

L'environnement technique de GE HealthCare se caractérise par une forte complexité : des applications médicales critiques doivent être déployées sur des infrastructures hybrides, mêlant serveurs bare-metal, machines virtuelles et environnements cloud. La nécessité de livrer des versions fiables, signées et parfaitement traçables est primordiale, car les solutions logicielles déployées ont un impact direct sur la qualité des soins et la sécurité des patients.

Avant mon arrivée, le processus de release était semi-automatisé et reposait largement sur des interventions manuelles. Les pipelines Jenkins existaient déjà, mais nécessitaient des ajustements réguliers pour gérer correctement les versions, intégrer les correctifs de dernière minute et publier les images ISO de manière standardisée. La génération des changelogs et la traçabilité des modifications étaient également fastidieuses, reposant sur des vérifications manuelles. Ce manque d'industrialisation entraînait des délais importants dans le cycle de livraison et augmentait le risque d'erreurs.
Ma mission a donc été d'automatiser et sécuriser l'ensemble de ce processus, en m'appuyant sur :

- **Jenkins** pour l'orchestration des pipelines de build, test et packaging.
- **GitLab et son API REST** pour récupérer les métadonnées associées aux commits, merge requests et releases.
- Des **scripts Shell et Python** pour automatiser les tâches répétitives, exécuter des requêtes API et générer des fichiers de version.
- **Postman** pour concevoir, tester et documenter les scénarios API avant intégration dans les pipelines.
- Une stratégie d'authentification robuste via Bearer Tokens, garantissant la sécurité des échanges avec les différents services.

5.3.3. Réalisations techniques

J'ai structuré mon travail autour de trois axes : la fiabilisation des versions, l'automatisation de la génération des artefacts et la simplification du suivi des releases.

Récupération et analyse des métadonnées via l'API GitLab

Pour chaque release, nous commencions par interroger l'API GitLab afin de récupérer la liste des merge requests associées. Cette étape était cruciale pour automatiser la génération du changelog et documenter précisément les modifications apportées entre deux versions.
Nous avons utilisé différentes méthodes HTTP selon les besoins :

- GET pour obtenir des informations sur les commits, pipelines et artefacts.
- POST pour déclencher la création d'une release.
- PUT pour mettre à jour les informations de version ou ajouter des annotations sur une version existante.

L'authentification était réalisée principalement via **Bearer Token**, un jeton unique intégré dans l'en-tête des requêtes, garantissant une sécurité élevée sans échange de mots de passe. Pour des environnements isolés ou des tests internes, nous avons parfois utilisé l'authentification basique (identifiant/mot de passe). Postman a été un outil précieux pour tester chaque endpoint : j'y configurais des environnements, simulais des scénarios complexes et validais le comportement des API avant de les intégrer dans nos scripts Jenkins.
Les routes GitLab API les plus sollicitées étaient :

- /projects/:id/merge_requests pour récupérer les MR associées à une version.
- /projects/:id/releases pour créer ou mettre à jour des releases.
- /projects/:id/pipelines pour déclencher et surveiller les pipelines.

Intégration avec Jenkins et automatisation des pipelines

Une fois les métadonnées collectées, les pipelines Jenkins prenaient en charge la génération des images ISO et des artefacts signés. Nous avons développé des scripts Shell permettant de :

- Récupérer les commits associés à la release.
- Générer une version unique basée sur le **versioning sémantique** (MAJOR.MINOR.PATCH).
- Signer les artefacts produits pour garantir leur intégrité.
- Déployer automatiquement les fichiers générés dans nos environnements de tests et de validation.

Les API Jenkins nous permettaient de déclencher dynamiquement des jobs via les routes suivantes :

- /job/:name/build pour exécuter un pipeline.
- /job/:name/api/json pour récupérer l'état détaillé d'un job ou ses logs.

Cette intégration fluide entre GitLab et Jenkins a considérablement réduit les délais de release et a supprimé les interventions manuelles à faible valeur ajoutée.

Génération automatisée des Release Notes et gestion des versions

L'un des livrables les plus visibles de ce projet a été la création d'un système automatisé de **release notes**. Grâce aux informations collectées via l'API GitLab, les notes de version étaient générées et publiées automatiquement avec des étiquettes claires :

- **ATD** (Acceptance Test Deployment) : version destinée aux environnements d'intégration pour validation fonctionnelle.
- **RC** (Release Candidate) : version prête à être validée pour la production.
- **BETA** : version intermédiaire, principalement destinée aux tests internes ou aux équipes produit.

Ces notes étaient publiées dans GitLab et intégrées à la documentation technique. Cela a permis aux équipes QA, DevOps et produit de disposer d'une vision claire et partagée de chaque version livrée, réduisant ainsi les incompréhensions et les erreurs lors des déploiements.

5.3.4. Impact et apprentissages

Ce projet m'a permis de développer une expertise avancée dans la gestion des versions logicielles, l'orchestration des pipelines CI/CD et l'intégration des API REST dans des environnements complexes. J'ai acquis une compréhension fine de la manière dont les différentes briques d'une usine logicielle s'articulent : gestion de code source, pipelines, packaging, déploiement et documentation.

Au-delà des compétences techniques, cette expérience m'a permis de prendre du recul sur la conception de processus industriels : comment structurer un pipeline pour qu'il soit robuste, comment sécuriser les interactions entre services, et comment fournir aux équipes une visibilité claire sur l'état des versions.

5.4. Optimisation des temps de préparation des tests en MET : refonte des pipelines Jenkins et introduction de workflows dédiés

5.4.1. Présentation des notions et des outils

Cette partie de mon expérience s'inscrit dans le cadre d'une démarche d'industrialisation des tests automatisés sur une usine logicielle cloud-native à grande échelle. L'environnement technique sur lequel je travaillais reposait principalement sur Jenkins pour l'orchestration des pipelines, GitLab comme plateforme de gestion de code source et d'intégration continue, ainsi que Kubernetes pour le déploiement et l'exécution des environnements de tests. Les pipelines automatisés concernaient un large ensemble de microservices et d'applications front-end, nécessitant une gestion rigoureuse des états des dépôts et des versions déployées.
Les tests end-to-end (E2E), qui constituent une étape critique dans tout processus de validation logicielle, consistaient à simuler des scénarios utilisateurs complets en environnement isolé. Ces tests étaient exécutés de façon **nightly**, c'est-à-dire chaque nuit, afin de détecter au plus tôt tout dysfonctionnement introduit dans la base de code. Dans ce contexte, chaque pipeline devait non seulement déployer les environnements de test, mais aussi s'assurer que les services et configurations étaient cohérents avant l'exécution des tests.
L'outil Jenkins, couplé à des agents configurés sur des nœuds Kubernetes, servait à exécuter les pipelines parallèlement sur plusieurs environnements. Chaque étape (checkout du code, construction des artefacts, déploiement sur Kubernetes, exécution des tests) était décrite sous forme de jobs Jenkins reliés les uns aux autres. Au-delà de Jenkins, nous avons également utilisé npm et des mécanismes de cache pour optimiser la compilation des modules front-end, ainsi que des registres Docker pour la distribution rapide des images nécessaires aux tests.
L'un des points clés de ce projet résidait dans la mise en place de **workflows modulaires**. Nous avons séparé les responsabilités des différents jobs, notamment le job « isv-deployment », qui initialement cumulait la préparation des environnements, le déploiement des solutions ISV (Independent Software Vendor) et l'exécution des tests. La création d'un workflow « Isv Start Workflow » dédié à l'orchestration des tests E2E a permis de clarifier ces responsabilités et d'améliorer la maintenabilité des pipelines.
Enfin, la notion de **nightly builds** occupe une place stratégique dans l'organisation des tests. Ces exécutions automatiques quotidiennes garantissent la fiabilité des livraisons et permettent une détection proactive des problèmes, avant qu'ils ne bloquent les cycles de livraison. Leur optimisation était donc un enjeu majeur, tant en termes de temps de calcul que de ressources consommées.

5.4.2. Contexte

Lorsque j'ai intégré ce projet, les tests end-to-end faisaient déjà partie du processus qualité de l'entreprise, mais leur exécution présentait plusieurs contraintes. Les pipelines Jenkins associés aux nightly builds étaient longs, complexes et peu flexibles. En moyenne, une exécution complète pouvait durer plusieurs heures, mobilisant des ressources considérables sur le cluster Kubernetes et ralentissant le flux de développement. Ces lenteurs constituaient un frein à la productivité des équipes et compliquaient la validation rapide des fonctionnalités critiques.

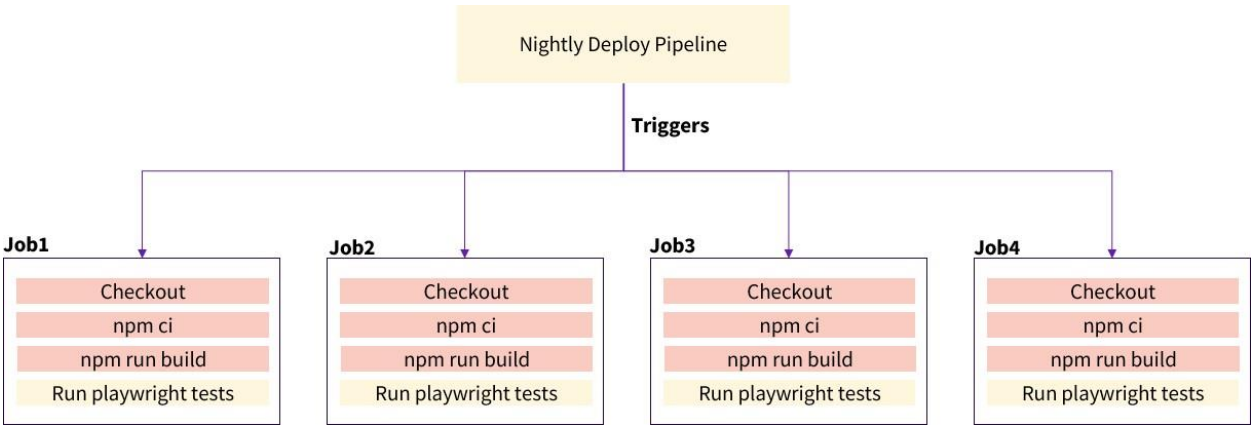


Figure 7 - pipelines Jenkins associés aux nightly builds

De plus, le job central « isv-deployment » avait évolué au fil du temps pour englober de nombreuses responsabilités : préparation des environnements, déploiement de solutions, déclenchement des tests et gestion des rapports de résultats. Cette complexité monolithique augmentait non seulement le temps d'exécution global, mais rendait également difficile toute tentative de relancer uniquement les tests après une modification mineure. Les développeurs perdaient ainsi du temps précieux en attendant des cycles complets de build, même pour des corrections mineures.
Dans ce contexte, ma mission a été de refondre ces pipelines afin de les rendre plus modulaires, plus rapides et plus simples à maintenir. L'objectif était clair : réduire les temps de préparation et d'exécution des tests E2E, mutualiser les étapes communes, et offrir aux équipes une flexibilité accrue dans le déclenchement et la gestion des nightly builds. Cette optimisation s'inscrivait également dans une logique d'industrialisation globale de l'usine logicielle, où chaque gain de temps sur les tests contribuait directement à accélérer la livraison de valeur aux utilisateurs finaux.

5.4.3. Réalisation

Mon travail a débuté par un audit complet des pipelines existants. J'ai analysé le fonctionnement de chaque étape, mesuré les temps d'exécution et identifié les points de contention. L'un des premiers constats a été que certaines étapes, comme le checkout des dépôts et la compilation des modules front-end via npm, étaient répétées inutilement à plusieurs endroits du pipeline. Ces duplications entraînaient une consommation excessive de temps et de ressources, surtout lorsque les builds étaient lancés en parallèle sur plusieurs branches.
J'ai alors entrepris de mutualiser ces étapes en introduisant des mécanismes de cache et de partage des artefacts entre les jobs. Par exemple, les artefacts générés lors d'une compilation étaient stockés dans un registre interne et réutilisés lors des tests, évitant de reconstruire le même code plusieurs fois. Cette mutualisation a permis de réduire significativement le temps de préparation avant le lancement des tests, tout en garantissant que l'état des dépôts restait strictement cohérent entre les différents jobs.

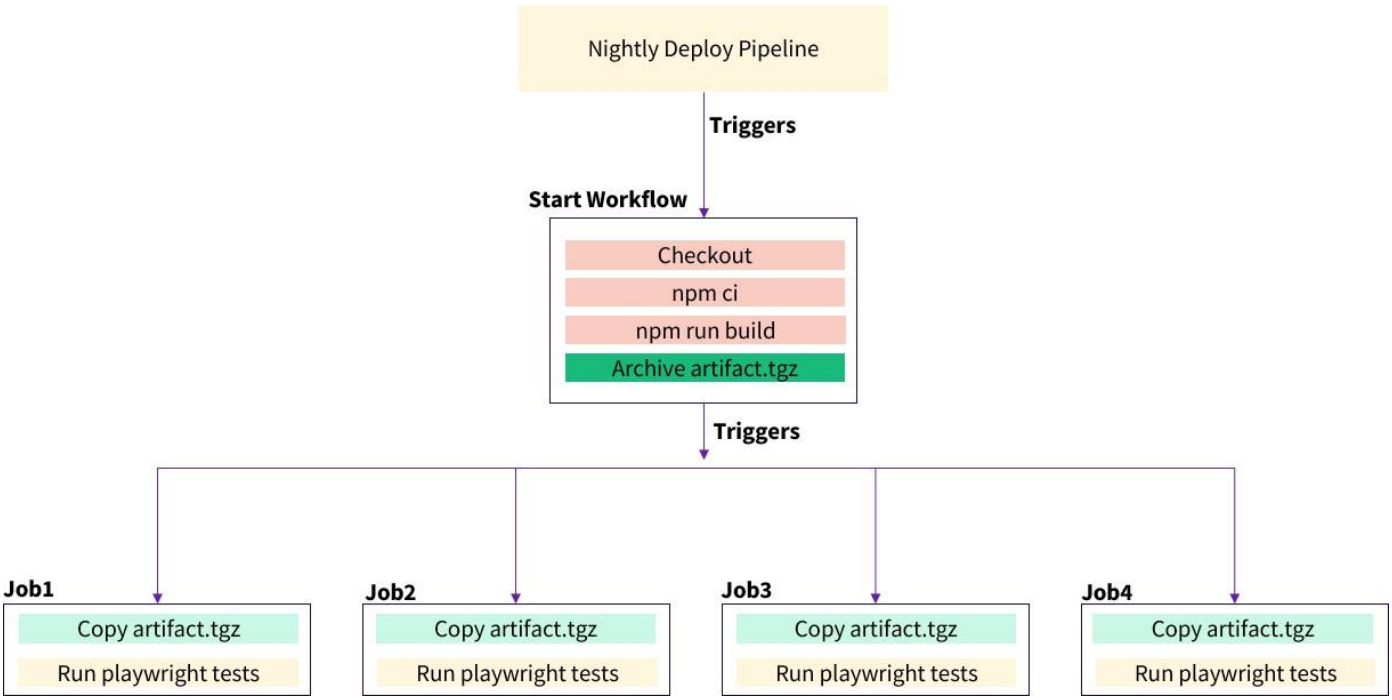


Figure 8 - La mutualisation des étapes

Une autre amélioration majeure a été la séparation des responsabilités du job « isv-deployment ». Plutôt que de tout centraliser dans un seul job complexe, j'ai conçu un workflow dédié, baptisé « Isv Start Workflow ». Ce workflow avait pour unique rôle de déclencher les tests E2E après que le déploiement ait été validé. Cette séparation des tâches a simplifié la logique des pipelines : il est désormais possible de relancer uniquement les tests sans avoir à reconstruire ou redéployer toute la solution, ce qui réduit considérablement les temps de cycle et la consommation de ressources.
Pour orchestrer cette nouvelle architecture, j'ai utilisé les fonctionnalités avancées de Jenkins, notamment les pipelines déclaratifs et les paramètres de build conditionnels. Les nightly builds ont été redéfinis pour tirer parti de cette modularité : chaque nuit, les tests sont exécutés sur des environnements dont la préparation est déjà optimisée, ce qui garantit non seulement des temps d'exécution plus courts, mais aussi une fiabilité accrue grâce à des environnements stables et reproductibles.
En parallèle, j'ai travaillé sur la publication automatisée des résultats des nightly builds. Les rapports de tests sont désormais centralisés et accessibles aux équipes dès le matin, ce qui facilite la détection et la résolution rapide des anomalies. L'ensemble de ces optimisations a permis de réduire de manière tangible le temps d'exécution des tests : les nightly builds, auparavant longs et parfois instables, s'exécutent désormais de manière fluide, offrant aux équipes un retour rapide et fiable sur la qualité du code.

5.4.4. Résultats et enseignements

Cette refonte des pipelines de tests a profondément transformé la manière dont les nightly builds

sont exécutés et exploités. Nous avons constaté une réduction significative du temps global d'exécution, ce qui s'est traduit par une meilleure réactivité des équipes et une accélération du cycle de livraison. La séparation des responsabilités entre le déploiement et l'exécution des tests a apporté une plus grande flexibilité, tandis que la mutualisation des étapes et l'utilisation des caches ont optimisé l'utilisation des ressources.
Au-delà de ces résultats techniques, ce projet m'a permis de développer une expertise avancée dans l'orchestration de pipelines complexes et dans l'industrialisation des tests à grande échelle. J'ai acquis une compréhension approfondie des bonnes pratiques en matière de CI/CD, de gestion d'artefacts et de modularisation des workflows. Cette expérience illustre également l'importance de la rigueur et de l'anticipation dans des environnements distribués, où chaque optimisation peut avoir un impact significatif sur la productivité globale.

5.5. Fiabilisation de la QA end-to-end : développement et correction des tests Playwright

5.5.1. Présentation des notions et outils

La qualité logicielle est un enjeu central dans tout projet de développement, et cela est particulièrement vrai dans le secteur médical, où les erreurs peuvent avoir des conséquences critiques. Dans ce contexte, les tests end-to-end (E2E) constituent un maillon essentiel du processus de validation, car ils simulent le parcours complet d'un utilisateur et permettent de détecter les régressions fonctionnelles et les anomalies de comportement des applications.
Chez GE HealthCare, nous utilisons principalement **Playwright**, une librairie de tests automatisés qui offre des fonctionnalités avancées pour simuler des interactions multi-navigateurs et multi-plateformes. Playwright permet de contrôler avec précision les étapes de navigation, les saisies utilisateur, les validations de contenus et les exécutions conditionnelles.
L'exécution des tests E2E s'inscrit dans des pipelines **Jenkins** et **GitLab CI/CD**, avec des agents dédiés qui isolent les environnements de tests pour garantir la reproductibilité. Les nightly builds, exécutés automatiquement chaque nuit, permettent de vérifier l'intégrité des applications avant la mise à disposition aux environnements pré-prod et production. Ces tests doivent être paramétrables selon l'environnement (MET) et les configurations de déploiement, afin de détecter tôt toute régression et sécuriser les promotions de version.
Pour la gestion des artefacts et la traçabilité, nous avons utilisé des registres Docker et **Artifactory**, afin de conserver les versions exactes des applications et des images utilisées lors des tests. La traçabilité des résultats, combinée à la journalisation des logs et à la capture de vidéos des exécutions, permet de vérifier que chaque test correspond précisément aux exigences fonctionnelles définies dans les cahiers de tests.

5.5.2. Contexte

Lorsque j'ai débuté ce projet, nous faisions face à plusieurs difficultés : certains tests étaient instables, avec des timeouts aléatoires et des échecs non reproductibles, notamment pour des applications nécessitant des environnements précis. Certaines tâches ne pouvaient pas s'exécuter dans des processus Linux standard, car elles requéraient des configurations spécifiques, des accès SSH aux serveurs ou des services dépendants de l'infrastructure.

Les tests existants étaient également dispersés, avec des scripts variés issus de différentes équipes, ce qui compliquait la maintenance et la cohérence des résultats. Les nightly builds prenaient beaucoup de temps à s'exécuter, car chaque test redéployait l'environnement, compilait les modules front-end, et exécutait des séquences de validation parfois redondantes. Ces contraintes rendaient difficile l'identification rapide des régressions et ralentissaient les promotions de version vers les environnements pré-prod et production.
Face à ces problèmes, ma mission consistait à fiabiliser l'exécution des tests Playwright, à stabiliser les timeouts et les dépendances, et à assurer une compatibilité multi-navigateurs. Il fallait également adapter les tests aux contraintes DevOps, en intégrant leur exécution dans les pipelines existants tout en optimisant la consommation de ressources et les temps de cycle.

```
import { ReportPortalReporter } from "@reportportal/agent-js-playwright";

const rpConfig = {
  token: process.env.RP_TOKEN,
  endpoint: process.env.RP_ENDPOINT, // e.g. https://rp.company.tld/api/v1
  project: process.env.RP_PROJECT, // e.g. qa-e2e
  launch: `e2e-${process.env.CI_PIPELINE_ID || Date.now()}`,
  attributes: [
    { key: "branch", value: process.env.CI_COMMIT_REF_NAME },
    { key: "commit", value: process.env.CI_COMMIT_SHORT_SHA },
    { key: "env", value: process.env.TEST_ENV || "nightly" },
  ],
  description: "Playwright E2E run with enriched metadata",
  mode: "DEFAULT", // or 'DEBUG'
};

export default {
  reporter: [
    [ReportPortalReporter, rpConfig],
    ["list"], // reporter console complémentaire
  ],
  use: {
    trace: "on-first-retry",
    screenshot: "only-on-failure",
    video: "retain-on-failure",
    baseURL: process.env.BASE_URL,
  },
  retries: 1, // retry contrôlé pour flakies
};

```

Figure 9 - Fiabilisation de la QA end-to-end

5.5.3. Réalisation

La première étape a été la préparation de l'environnement. Nous avons demandé et configuré un serveur dédié pour les tests, isolé des autres processus Linux, afin de garantir que les scripts Playwright puissent s'exécuter dans un contexte contrôlé. Cela a impliqué la gestion des connexions SSH et la configuration des agents Jenkins sur des nœuds Kubernetes spécifiques, capables de lancer plusieurs navigateurs simultanément et d'isoler les exécutions pour chaque job.
Ensuite, j'ai entrepris la réécriture et la correction des tests existants. Les scripts Playwright ont été refondus pour utiliser notre librairie de test interne, permettant de standardiser la syntaxe, d'améliorer la lisibilité et de garantir la cohérence avec les requirements des tests. Chaque étape du test a été validée individuellement pour s'assurer que le comportement correspondait exactement aux scénarios attendus. Nous avons introduit des mécanismes pour stabiliser les timeouts et gérer les cas de navigation multi-navigateurs, ce qui a permis de réduire les échecs intermittents et d'améliorer la fiabilité globale des tests.
Pour chaque application, nous avons défini des nightly builds paramétrables en MET. Cela signifie que nous pouvions exécuter les tests sur différents environnements de préparation et de validation, tout en réutilisant les mêmes artefacts compilés et les mêmes images Docker. La mutualisation des étapes de build et de déploiement a contribué à réduire le temps total des nightly builds, tout en conservant une traçabilité complète des résultats et des artefacts.
L'intégration dans Jenkins a été réalisée en définissant des pipelines modulaires, où chaque job avait un rôle précis : préparation de l'environnement, exécution des tests, collecte des résultats et publication des logs et vidéos. Cette architecture a permis de relancer facilement les tests de manière isolée, sans redéployer l'ensemble des applications, ce qui est particulièrement utile lors de corrections ponctuelles ou de vérifications rapides.
Enfin, nous avons automatisé la génération et la publication des rapports de tests, afin que les équipes puissent accéder aux résultats dès le matin. Chaque rapport incluait les détails de l'exécution, les captures d'écran, les vidéos et les logs, ce qui facilitait l'identification des anomalies et la communication avec les équipes de développement et de DevOps.

5.5.4. Résultats et enseignements

Les résultats de cette optimisation ont été significatifs. Les nightly builds sont désormais plus rapides, plus fiables et reproduisent systématiquement l'état exact des applications dans les environnements MET. La stabilité des tests Playwright s'est améliorée, avec une réduction notable des échecs intermittents et une meilleure couverture multi-navigateurs.
Cette expérience m'a permis de renforcer mes compétences dans le développement et la maintenance de tests automatisés dans un contexte DevOps, d'approfondir mes connaissances sur Playwright et les pipelines Jenkins, et de comprendre l'importance de la modularité et de la traçabilité dans des projets complexes à grande échelle. Elle illustre également la complémentarité entre les pratiques de QA et l'industrialisation DevOps, où chaque optimisation des tests contribue directement à la fiabilité et à la rapidité des livraisons.

5.6. Application des principes SRE avec Ansible : optimisation des rôles et playbooks pour Linux SUSE, durcissement SSH et déploiements automatisés.

5.6.1. Présentation des notions et outils

Dans un environnement DevOps actuel, je vois le Site Reliability Engineering (SRE) comme une manière très pragmatique d'appliquer les méthodes d'ingénierie logicielle à l'exploitation des systèmes. L'idée est de rendre les déploiements et l'exploitation plus fiables en s'appuyant sur des objets concrets: des objectifs de service qu'on peut mesurer, des procédures reproductibles, et des retours d'expérience qui alimentent des améliorations continues. Je ne cherche pas à ajouter des couches d'outillage, mais au contraire à clarifier le minimum nécessaire qui me permet d'automatiser, d'observer et d'agir sans surprises.

Dans mon contexte, l'Infrastructure as Code (IaC) est la pièce maîtresse. Décrire l'état cible d'un système de manière déclarative et versionnée aide à réduire le nombre de "surprises" entre les environnements. J'ai utilisé Ansible pour piloter des serveurs Linux SUSE, structurer des rôles, écrire des playbooks centrés sur un scénario de déploiement bien défini, et vérifier que le résultat est le même en MET, en Pré-production et en Production. L'avantage d'Ansible, au-delà de l'idempotence, c'est le check mode et la capacité à simuler les changements avant de les appliquer vraiment. J'ai aussi gardé un lien étroit avec les pipelines Jenkins et GitLab CI/CD, mais en décalant une partie des tests en local pour ne pas monopoliser les agents et leurs ressources CPU/GPU. Enfin, je me suis appuyé sur ce qui existe déjà côté système: SSH pour exécuter des commandes à distance, zypper pour la gestion de paquets, systemctl et journalctl pour piloter et diagnostiquer les services, cron et les timers systemd pour la planification, et quelques commandes simples comme uptime, w, who, df -h ou du -sh pour prendre la température d'un hôte.

Je reste volontairement centré sur ces outils, parce que ce sont eux qui sont réellement utilisés et qui s'emboîtent naturellement avec les tâches à réaliser. Chaque outil correspond à une étape précise: Ansible pour décrire et appliquer l'état, SUSE et ses commandes natives pour vérifier le système, SSH pour l'exécution distante, Jenkins/GitLab pour orchestrer les builds et les promotions. Je n'ai pas ajouté d'outils périphériques qui n'auraient pas été indispensables dans ce cadre.

5.6.2. Contexte

Pendant mon alternance, j'ai été confronté à deux difficultés principales. D'abord, la charge importante sur les agents Jenkins et GitLab, surtout lors des nightly builds et des campagnes de test. Quand plusieurs jobs lourds se déclenchaient en même temps, on pouvait observer des ralentissements sensibles, voire des échecs aléatoires compliqués à expliquer, notamment sur des hôtes équipés de GPU partagés. Ensuite, des écarts de configuration existaient parfois entre l'environnement MET et la PROD. Même si ces différences n'étaient pas énormes, elles suffisaient à provoquer des comportements inattendus côté application ou à faire diverger un pipeline qui passait en MET mais échouait en PROD.
Mon objectif a donc été double. D'un côté, j'ai voulu mettre en place un moyen de tester localement les déploiements, de préférence sur un hôte de test SUSE isolé mais représentatif, afin de détecter tôt les problèmes de versions, de dépendances ou de services mal paramétrés. De

l'autre, j'ai renforcé l'observation de l'état des serveurs en m'appuyant sur des commandes simples mais fiables pour suivre la charge, les processus, les sessions actives, l'occupation disque et les tâches planifiées. Ces deux axes se complètent bien: la qualification locale diminue la pression sur les agents partagés, tandis que la visibilité opérationnelle permet d'ajuster la planification des tâches et d'anticiper les pics.

5.6.3. Réalisation

J'ai commencé par poser des bases propres côté IaC. Dans Ansible, j'ai structuré des rôles par domaine fonctionnel: installation de paquets via zypper, configuration de services systemd, déploiement d'artefacts applicatifs, gestion des fichiers de configuration avec des templates. J'ai soigné les inventaires et les variables par environnement pour laisser varier uniquement ce qui devait l'être (par exemple un endpoint différent ou un quota), et garder identiques toutes les options qui ne devaient pas diverger (versions précises, paramètres système, options de démarrage des services). Pour les secrets strictement nécessaires au déploiement, j'ai utilisé Ansible Vault. Cela m'a permis de rester dans le même outil tout en respectant la confidentialité.

Ensuite, j'ai écrit des playbooks de tests de déploiement exécutables en local. L'idée n'était pas de refaire toute la CI en miniature, mais de cibler les scénarios critiques: installer ou mettre à jour une version donnée d'un service interne, déposer et activer une unité systemd, démarrer le service et vérifier son comportement. Chaque playbook suivait une trame lisible: préparation de l'hôte de test, application des changements, validation avec des commandes natives, puis collecte d'un petit rapport d'exécution. Les handlers d'Ansible m'ont servi à ne redémarrer un service que s'il y avait réellement une modification, ce qui évite des interruptions inutiles. J'ai utilisé des assertions pour faire échouer explicitement un run si une condition essentielle n'était pas remplie, par exemple si la version installée n'était pas celle attendue, si le port du service n'était pas ouvert, ou si systemctl renvoyait un état dégradé.

Pour rester concret, je prends l'exemple d'un service interne. Je commence par contrôler l'espace disque disponible sur les points de montage concernés avec df -h et, si besoin, une vérification plus ciblée avec du -sh. Je vérifie ensuite la présence des dépendances via zypper info ou rpm -q selon le packaging. J'installe la version cible, je déploie l'unité systemd associée si elle a évolué, je recharge les daemons, j'active le service au boot et je le démarre. Juste après, j'inspecte l'état avec systemctl status et je consulte les quelques dernières entrées du journal du service avec journalctl -xe en filtrant sur l'unité. Je lance un test de fumée basique pour vérifier que le service répond comme prévu sur son port ou son endpoint local, en laissant un petit délai de stabilisation. À la fin, je capture les sorties importantes et je les archive avec Ansible pour produire un rapport court mais exploitable.

Cette validation locale a deux intérêts. D'abord, je n'encombre pas les agents Jenkins/GitLab quand je suis encore en phase d'itération. Je peux corriger les erreurs de configuration ou de dépendance sans bloquer d'autres équipes. Ensuite, quand je passe dans le pipeline, je sais que la base est saine: les mêmes rôles et playbooks vont s'exécuter, et j'ai déjà confronté l'installation et le démarrage du service à un environnement proche de MET. Dans la pratique, cela réduit les surprises.

Parallèlement, j'ai mis en place une routine d'observation très simple des hôtes, en SSH. Je commence souvent par uptime pour voir la charge moyenne. Si je détecte un pic, j'ouvre htop pour repérer le ou les processus qui consomment le plus. Je regarde w et who pour savoir combien de sessions sont actives et si des connexions inattendues prennent des ressources. Je vérifie aussi l'occupation disque avec df -h et, si un système de fichiers est proche de la saturation, je descends d'un cran avec du -sh sur les répertoires volumineux. Enfin, je dresse la carte des tâches planifiées via crontab -l et systemctl list-timers. L'objectif est de comprendre si des batchs ou des scrubs périodiques se superposent avec des builds ou des tests lourds. Quand j'identifie des chevauchements, je propose une replanification. Cela peut sembler basique, mais c'est souvent là que se nichent des gains immédiats: décaler un timer de quelques minutes ou éviter un créneau déjà chargé suffit parfois à supprimer des ralentissements perçus comme "aléatoires".

Pour réduire les écarts entre MET et PROD, j'ai utilisé deux leviers. Le premier, c'est la clarté des inventaires Ansible: je note explicitement ce qui a le droit de différer et je verrouille le reste. Le second, c'est l'usage systématique du check mode avant d'appliquer un changement. Ansible montre le diff, je vois tout de suite si une modification inattendue s'est glissée, et je corrige avant de toucher à des hôtes sensibles. Quand je dois avancer vers la PROD, je passe par une Pré-production avec un sous-ensemble d'hôtes pour valider le comportement dans des conditions plus proches. Ce passage intermédiaire a limité les retours arrière et m'a donné des arguments concrets pour défendre une promotion par paliers, au lieu d'un déploiement frontal.

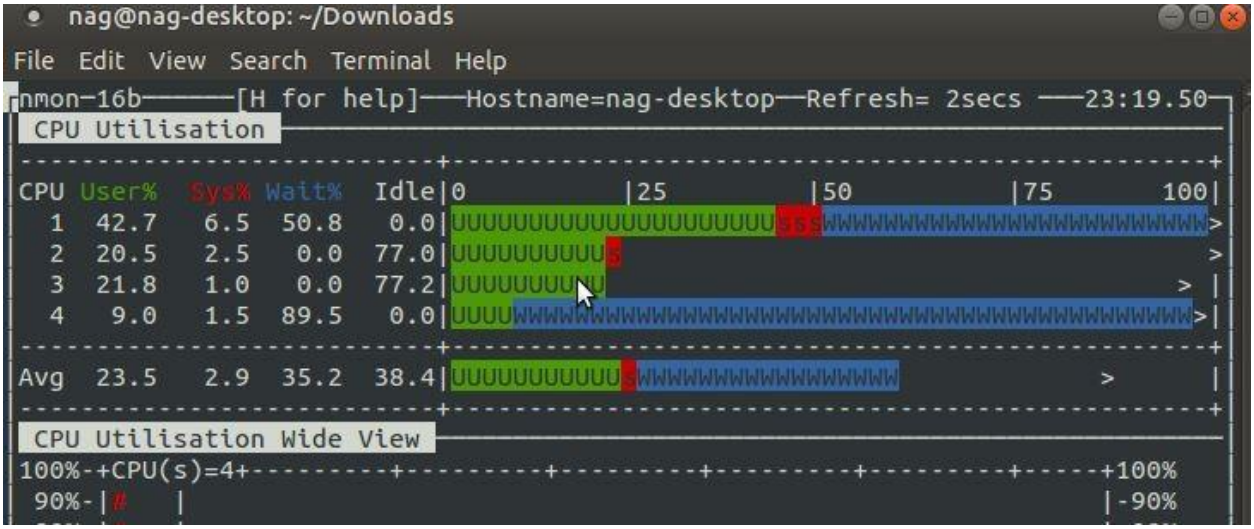


Figure 10 La routine d'observation par SSH

Concernant la charge sur les agents, j'ai fait un tri simple des tests. D'un côté, les tests de cohérence de configuration et de démarrage de service, qui gagnent à être exécutés localement via Ansible. De l'autre, les tests intégrés plus lourds, qui restent sous la responsabilité des pipelines Jenkins/GitLab, parce qu'ils ont besoin d'une plateforme représentative et d'une orchestration que je n'ai pas en local. En procédant ainsi, les jobs des pipelines se concentrent sur ce qui certifie vraiment la version à promouvoir, pendant que les itérations rapides se font sans bruit sur un hôte de test.

Dans l'ensemble, mon flux type ressemble à ceci, en continu sans ajouter de couches inutiles. Je commence par qualifier ma modification en local avec Ansible, d'abord en check mode puis en exécution réelle sur un serveur SUSE dédié. Je lis le rapport, je corrige si besoin, je recommence jusqu'à obtenir un démarrage propre et un test de fumée réussi. Je pousse ensuite mes changements dans le repository, ce qui déclenche ou prépare l'exécution contrôlée dans Jenkins ou GitLab CI/CD. Les runners consomment moins de ressources pour les étapes devenues plus déterministes, et je récupère les artefacts produits par le pipeline pour tracer la promotion. Enfin, je déploie progressivement: MET, Pré-production, puis PROD, avec un suivi rapproché des métriques de base du système et des diffs d'Ansible à chaque étape.

5.6.4. Présentation élargie des notions SRE exploitées

Même si je n'ai pas cherché à formaliser des tableaux de bord complexes, j'ai gardé une logique SRE simple et actionnable. Par exemple, j'ai défini un indicateur de disponibilité après déploiement basé sur la réussite d'un test de fumée dans un délai donné après un systemctl start. Sur une période glissante, je me fixe un objectif réaliste de réussite et j'observe les échecs pour comprendre s'ils sont liés à des dépendances absentes, à des temps de démarrage trop longs, ou à des paramètres système mal calibrés. Cela joue un rôle de "budget d'erreur" informel: si je vois que les échecs augmentent, je réduis le rythme des changements et je renforce les validations locales jusqu'à retrouver un niveau stable.

J'ai aussi produit de petits runbooks opérationnels au même endroit que les rôles Ansible. Ce sont des pas-à-pas courts: quoi vérifier en premier si le service ne démarre pas, quelles commandes lancer, quels journaux consulter, et dans quel ordre. L'avantage est double. D'un côté, je gagne du temps quand un incident survient, parce que je n'ai pas à improviser. De l'autre, ces fiches sont au même format que l'automatisation, donc elles restent synchronisées et faciles à mettre à jour. En pratique, elles s'appuient uniquement sur les outils que j'utilise réellement: systemctl, journalctl, zypper et les modules de vérification d'Ansible. Pas de couches additionnelles, juste une formalisation de ce que je fais déjà, ce qui renforce la reproductibilité.

5.6.5. Retours d'expérience sur l'optimisation des ressources

Le bénéfice le plus visible a été la baisse de la pression sur les agents Jenkins et GitLab pendant les créneaux critiques. Le fait d'avoir déplacé une partie des validations en local a évité des files d'attente interminables et des compétitions de ressources qui dégradaient l'expérience de tout le monde. En parallèle, les étapes restées dans les pipelines ont gagné en clarté. Au lieu d'échouer pour des raisons triviales (une dépendance manquante, un service qui ne démarre pas), elles se concentraient sur des tests qui apportent une vraie valeur de certification.

La routine d'observation par SSH a été très utile pour objectiver les pics de charge. Quand uptime montrait un load moyen inhabituel, je pouvais ouvrir htop, identifier le processus en cause, et vérifier si un cron ou un timer venait de se déclencher. En croisant ces informations avec les horaires des jobs Jenkins/GitLab, j'ai pu proposer des ajustements simples. Par exemple, décaler un nettoyage disque ou une tâche d'indexation pour qu'elle n'arrive pas en même temps qu'un build volumineux. Ces petits réglages, une fois documentés dans le repository Ansible, deviennent

des décisions tracées et faciles à maintenir.

Sur le plan des environnements, le fait d'avoir des inventaires clairs et de recourir au check mode a réduit les différences non désirées entre MET et PROD. Rejouer exactement les mêmes rôles, avec les mêmes templates et les mêmes variables, a rendu les promotions beaucoup plus prévisibles. J'ai aussi pris l'habitude de comparer les "facts" collectés par Ansible entre deux hôtes, pour repérer des divergences matérielles ou de noyau qui peuvent expliquer des variations de comportement. Ce n'est pas spectaculaire, mais cela évite de chercher trop longtemps au mauvais endroit.

5.6.6. Résultats et enseignements

Au final, l'approche SRE que j'ai appliquée est restée sobre, mais efficace. Grâce à Ansible, j'ai industrialisé les déploiements sur SUSE avec des rôles et des playbooks stables, rejouables et traçables. La possibilité de tester en local a permis d'attraper tôt les erreurs de configuration et d'épargner les agents CI/CD. L'observation régulière des hôtes via des commandes natives m'a aidé à anticiper les périodes de surcharge et à ajuster la planification de certaines tâches. La conséquence directe a été la réduction des écarts entre MET et PROD, moins d'échecs inattendus dans les pipelines, et une meilleure fluidité lors des promotions.

Sur le plan personnel, j'ai gagné en méthode. J'accorde plus d'attention à l'idempotence et aux dry-runs avant d'appliquer un changement. Je documente mes interventions sous forme de runbooks courts, au plus près du code d'automatisation, pour que la connaissance reste vivante. Et je m'appuie d'abord sur les outils déjà présents, tant qu'ils répondent au besoin. Cette discipline m'a permis d'améliorer la fiabilité sans grossir l'outillage.

Je retiens aussi que la cohérence entre environnements n'est pas un but abstrait: elle se construit petit à petit, avec des inventaires propres, des variables bien séparées, et l'habitude de simuler les changements. Lorsqu'on ajoute à cela une boucle courte de validation locale et une observation régulière des hôtes, on obtient un cycle de livraison plus constant, où la PROD se comporte comme attendu parce que MET lui ressemble vraiment. Pour moi, c'est exactement l'esprit du SRE: s'appuyer sur l'ingénierie pour produire une fiabilité mesurable, et faire en sorte que cette fiabilité tienne dans le temps grâce à des pratiques reproductibles et raisonnables.

Enfin, ce cadre reste extensible sans se compliquer. Si demain le périmètre s'élargit, je peux reprendre la même démarche: préparer l'état cible avec Ansible, vérifier en local, intégrer dans le pipeline, observer, puis promouvoir par étapes. Les outils restent les mêmes, les gestes aussi. Cette continuité donne confiance, autant aux équipes qui exploitent qu'à celles qui livrent. Et c'est cette confiance, au fond, qui permet d'avancer vite sans sacrifier la stabilité.

5.7. Observabilité et reporting des campagnes de tests

5.7.1. Présentation des notions et outils

Dans un contexte DevOps, la **traçabilité et l'observabilité** des tests et des pipelines de validation sont des éléments essentiels pour garantir la qualité des livraisons et le respect des engagements de cycle. L'observabilité consiste à collecter et organiser des informations sur les processus et les applications afin de comprendre leur comportement et de détecter les anomalies. La traçabilité, quant à elle, permet de relier chaque résultat de test aux artefacts, aux versions de code et aux exigences fonctionnelles correspondantes.
Pour cette mission, l'outil central utilisé a été **ReportPortal**, une plateforme capable de centraliser les résultats de tests provenant de différentes sources, de les analyser et de fournir des métriques opérationnelles. ReportPortal permet d'ingérer des rapports **JUnit**, **XML**, ou **HTML**, puis de les agréger par projet, version, environnement et propriétaire de test. Cette approche facilite la création de dashboards dynamiques et d'indicateurs fiables sur la performance des pipelines et la stabilité des applications.

5.7.2. Contexte

Lorsque j'ai commencé à travailler sur ce projet, l'équipe rencontrait plusieurs difficultés en matière de suivi des tests. Les résultats des campagnes étaient dispersés entre différents environnements et différents pipelines. Il était difficile de savoir rapidement si un échec venait d'un problème dans le code, d'une mauvaise configuration d'environnement MET ou d'un souci lié aux dépendances.
L'absence d'un standard pour agréger les résultats compliquait la communication entre les équipes DevOps, QA et développement. Chaque projet avait ses propres conventions de reporting, ce qui rendait la production de métriques consolidées pour le suivi des SLA et du temps de cycle jusqu'à la pré-production très laborieuse.
L'objectif principal de cette mission était donc de mettre en place une **observabilité complète** et une **traçabilité standardisée** des campagnes, afin de pouvoir suivre de manière fiable l'évolution des tests, identifier rapidement les régressions et produire des indicateurs compréhensibles par toutes les parties prenantes.

5.7.3. Réalisation

La première étape a été l'installation et la configuration de ReportPortal dans un environnement local de test. Nous avons dû adapter les agents pour **npm et Node.js**, afin de permettre la collecte automatique des rapports générés par les tests Playwright et JUnit. L'installation locale a permis de valider la compatibilité avec notre infrastructure et de tester différentes stratégies d'agrégation avant le déploiement en environnement MET.

Figure 11 - chart Visualitsation State E2E

Ensuite, j'ai travaillé sur la **hiérarchisation des résultats de tests**. Nous avons introduit une structure de regroupement par version et par environnement, en utilisant la fonction mergeAllLaunches. Cette approche permet de combiner plusieurs exécutions de tests pour une même version de l'application, et d'obtenir des graphiques consolidés reflétant l'état global de la release et le respect des SLA. Les métriques collectées comprenaient le **taux de réussite quotidien et mensuel**, le nombre de tests échoués ou instables, la durée moyenne des exécutions et les tendances par pipeline ou par propriétaire de test.

Nous avons également configuré des dashboards personnalisés dans ReportPortal. Ces dashboards fournissent des **charts pertinents** pour les parties prenantes : évolution journalière du succès des tests, heatmaps des tests les plus critiques, graphiques de stabilité par environnement et par version. Chaque chart permet de visualiser non seulement l'état actuel des tests, mais aussi les tendances sur plusieurs semaines, ce qui est crucial pour anticiper les problèmes et planifier les actions correctives.

Une attention particulière a été portée à la **traçabilité complète des artefacts**. Chaque test est maintenant lié à la version du code, à la branche GitLab, au job Jenkins associé et aux artefacts générés, comme les packages npm, les images Docker ou les logs d'exécution. Cette traçabilité facilite les audits internes et permet de reconstruire le chemin exact d'un test depuis son déclenchement jusqu'à son résultat final, renforçant ainsi la fiabilité des livraisons.

Enfin, nous avons intégré des fonctionnalités de reporting automatique. Les résultats consolidés sont publiés régulièrement pour les équipes QA et DevOps, avec un suivi des anomalies et des indicateurs de performance des pipelines. Cela a considérablement réduit le temps nécessaire pour identifier les régressions et améliorer la stabilité MET avant toute promotion vers pré-production.

5.7.4. Résultats et enseignements

La mise en place de ReportPortal et la standardisation des campagnes ont eu plusieurs impacts positifs :

- Meilleure visibilité sur la performance des tests et des pipelines.
- Réduction des écarts entre les environnements MET et pré-prod.
- Détection précoce des régressions et anomalies dans les tests.
- Production de métriques fiables et exploitables pour le suivi des SLA et la planification des releases.
- Communication simplifiée entre équipes DevOps, QA et développement grâce à des dashboards unifiés et compréhensibles.

Cette expérience m'a permis de développer une expertise concrète sur l'observabilité et la traçabilité des campagnes de tests dans un environnement complexe, de maîtriser l'outil ReportPortal et de comprendre comment structurer et exploiter des métriques pour optimiser les processus DevOps. Elle illustre également l'importance de la standardisation des pratiques pour industrialiser les pipelines de validation dans des projets critiques à grande échelle.

5.8. Optimisation et productivité : création de jobs de déclenchement pour nettoyage automatisé

5.8.1. Présentation des notions et outils

Dans mon environnement d'alternance chez GE HealthCare, j'ai travaillé au cœur d'une chaîne DevOps qui s'appuie au quotidien sur des serveurs de build, des runners d'intégration continue et des environnements de test basés sur Linux. Les composants avec lesquels j'ai interagi directement sont ceux qui structurent le cycle de vie des artefacts et des exécutions. Docker pour les images et les conteneurs. Kubernetes pour l'orchestration et le cycle de vie des pods. Jenkins et GitLab CI pour l'orchestration des builds, des tests et des déploiements. Bash et la CLI des outils pour automatiser concrètement les opérations récurrentes. Je me suis volontairement limité à ces technologies, car elles sont intrinsèquement liées aux tâches réalisées et déjà en place dans l'usine logicielle.

L'angle d'attaque a été simple et pragmatique. Les jobs que j'ai créés, appelés TRIGGER_TESTS, ont pour rôle de nettoyer ce qui s'accumule dans le temps et qui n'apporte plus de valeur. Images Docker non utilisées. Conteneurs arrêtés et volumes orphelins. Pods Kubernetes terminés ou laissés en attente alors qu'ils ne seront plus relancés. Artefacts de build périmés au regard d'une politique de rétention explicitement fixée. L'objectif n'est pas de "faire de la place pour faire de la place", mais de maintenir un état de travail propre et prévisible afin que les pipelines nightly s'exécutent dans des conditions reproductibles.

Chaque brique outillage répond à une étape concrète. La CLI Docker me sert à inventorier, à mesurer et à supprimer de manière ciblée. La CLI kubectl me permet de lister, qualifier et purger les ressources éphémères qui encombrent un cluster. Jenkins et GitLab CI orchestrent le moment et le contexte d'exécution des scripts de nettoyage, avec un planning régulier et des déclenchements à la demande. Les logs et les journaux produits par ces outils servent de matière première au reporting mensuel que je prépare pour objectiver les gains et vérifier qu'aucune suppression intempestive n'a affecté la stabilité des environnements.

5.8.2. Contexte

Avant la mise en place de ces jobs, j'ai constaté trois symptômes récurrents sur les environnements nightly et de test. D'abord, la saturation progressive de l'espace disque sur certains nœuds de build, due à l'empilement d'images Docker anciennes, de conteneurs stoppés et de volumes jamais réutilisés. Ensuite, l'allongement des temps de démarrage des pipelines nocturnes, les étapes passant davantage de temps à télécharger ou à retagger des images en doublon ou à vérifier des états incohérents. Enfin, une dérive de l'entropie côté Kubernetes, avec des pods terminés qui persistaient, rendant la lecture de l'état du cluster moins lisible et masquant parfois des signaux utiles.
La réponse manuelle à ces symptômes existait, mais elle était coûteuse. Un ingénieur devait se connecter, exécuter les commandes adaptées, trier l'utile de l'obsolète, et répéter l'opération sur plusieurs hôtes et namespaces. Ce type d'intervention dépanne mais ne tient pas dans la durée, surtout lorsque les équipes accélèrent la cadence de livraison. J'ai donc cadré le sujet comme un chantier d'optimisation durable, avec un nettoyage récurrent, instrumenté, traçable et sécurisé par quelques garde-fous simples. Je me suis attaché à rester dans un périmètre strictement lié aux outils déjà utilisés et aux artefacts réellement produits par nos pipelines.

5.8.3. Objectifs

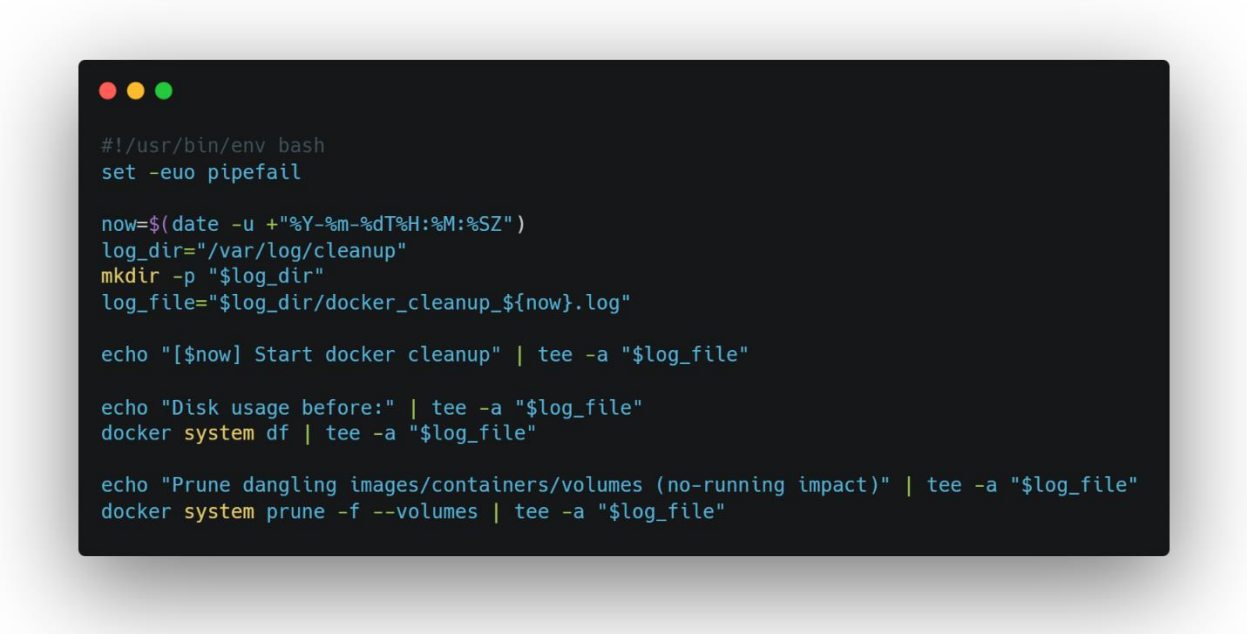
Je me suis fixé des objectifs concrets et observables.
Maintenir un niveau d'espace disque disponible suffisant sur les nœuds de build et de test, sans intervention manuelle récurrente.
Réduire le temps perdu en opérations annexes au démarrage des pipelines nightly, en supprimant les artefacts qui provoquent des vérifications et téléchargements superflus.
Rendre la lecture de l'état des clusters de test plus claire, en éliminant régulièrement les ressources Kubernetes arrivées au bout de leur cycle de vie.
Documenter et tracer chaque action de nettoyage pour pouvoir expliquer, au besoin, ce qui a été supprimé, où et quand, et avec quel critère.
Conserver ce qui est nécessaire à la reproductibilité des builds et des déploiements, en alignant les règles de rétention sur la politique établie par l'équipe Build and Release.

5.8.4. Périmètre

Le périmètre couvre les hôtes Linux qui supportent les jobs Jenkins ou les runners GitLab CI utilisés pour les nightly, ainsi que les namespaces Kubernetes dédiés aux tests. Je n'ai pas étendu ce travail à la production. Je n'ai pas modifié la chaîne de sécurité ni les politiques d'accès. Je me suis limité à des actions de maintenance applicables par les comptes de service déjà en place dans ces environnements. Les ISO et paquets produits par les pipelines font partie du périmètre, mais seulement sous l'angle de la rétention, telle qu'elle a été définie avec l'équipe responsable des releases.

5.8.5. Réalisation détaillée

J’ai commencé par un état des lieux factuel. L’inventaire des artefacts Docker existants se fait avec des commandes de base. docker system df pour mesurer l’empreinte disque des images, des couches et des volumes. docker images -a pour lister l’ensemble des images, y compris celles sans tag, souvent appelées dangling. docker ps -a pour repérer des conteneurs arrêtés qui ne seront pas redémarrés. À ce stade, je ne supprime rien. Je collecte, j’agrège et je mets en forme les informations nécessaires pour décider calmement.
En parallèle, j’ai fait le même travail côté Kubernetes sur les namespaces de test. kubectl get pods --all-namespaces avec des colonnes d’âge et de statut me permet d’identifier ce qui est terminé depuis longtemps ou en erreur sans perspective de redémarrage. Cette observation précède la définition des règles, car c’est la réalité des objets présents qui dicte ce qu’il faut automatiser, pas l’inverse.
Une fois l’inventaire consolidé, j’ai formalisé des critères de nettoyage. Un exemple simple pour Docker. Une image non utilisée par aucun conteneur en cours d’exécution et plus référencée par les derniers builds peut être supprimée. Les images non utilisées depuis plus de trente jours constituent une bonne première cible. Les conteneurs arrêtés depuis plus de sept jours sont également éligibles. Les volumes non attachés à des conteneurs actifs sont candidats à la suppression après vérification. Pour Kubernetes, je vise les pods en statut Succeeded ou Failed au-delà d’un certain âge, ainsi que les jobs de test terminés qui n’apportent plus d’information utile au-delà de la période de rétention convenue.
Avec ces critères approuvés, j’ai rédigé des scripts Bash simples, lisibles et idempotents. L’ordre des opérations est volontaire. Je commence par recalculer l’état du système au moment où le job s’exécute, pour éviter toute décision basée sur un inventaire périmé. Je lance ensuite un mode sec quand c’est pertinent, par exemple docker system prune en dry-run pour évaluer l’impact. Puis j’applique les suppressions ciblées, en enregistrant chaque action dans un log horodaté. En fin d’exécution, je recalcul l’état disque et je résume les gains, ce qui servira au tableau de bord.
Voici un exemple de trame pour la partie Docker. Je l’ai gardée modeste pour qu’elle reste maintenable par les équipes.



```bash
#!/usr/bin/env bash
set -euo pipefail

now=$(date -u +"%Y-%m-%dT%H:%M:%SZ")
log_dir="/var/log/cleanup"
mkdir -p "$log_dir"
log_file="$log_dir/docker_cleanup_${now}.log"

echo "[$now] Start docker cleanup" | tee -a "$log_file"

echo "Disk usage before:" | tee -a "$log_file"
docker system df | tee -a "$log_file"

echo "Prune dangling images/containers/volumes (no-running impact)" | tee -a "$log_file"
docker system prune -f --volumes | tee -a "$log_file"
```

Figure 12 - Images Docker non utilisées depuis plus de 30 jours.

Cette base est volontairement illustrative. Dans la version utilisée au quotidien, je m'appuie sur les tags et les manifest lists des derniers builds pour protéger explicitement les images de référence conservées pour la reproductibilité. J'évite toute logique trop complexe, car l'objectif est d'expliquer facilement ce que fait le script et de pouvoir l'auditer en quelques minutes.
Pour Kubernetes, la logique est similaire. Je collecte la liste des pods par namespace, je filtre ceux en Succeeded et Failed au-delà d'un âge donné, puis je supprime. Je traite ensuite les jobs terminés quand ils existent. Là encore, je logue chaque action, avec le couple namespace plus nom de ressource, ainsi que l'UID quand je dois justifier une suppression précise dans un rapport.

```bash
#!/usr/bin/env bash
set -euo pipefail

now=$(date -u +"%Y-%m-%dT%H:%M:%SZ")
age_days="${AGE_DAYS:-7}"
log_dir="/var/log/cleanup"
mkdir -p "$log_dir"
log_file="$log_dir/k8s_cleanup_${now}.log"

echo "[$now] Start k8s cleanup (pods/jobs older than ${age_days}d in Succeeded/Failed)" | tee -a
"$log_file"

# Pods
kubectl get pods --all-namespaces --no-headers \ -o
```

Figure 13 - Optimisation et monitoring

Une fois les scripts prêts, je les ai intégrés dans des jobs Jenkins et des pipelines GitLab CI existants. Le chaînage est logique. La planification cron de Jenkins exécute le nettoyage sur des créneaux calmes, après les nightly mais avant la fenêtre suivante, pour que le prochain cycle reparte sur un état propre. Les pipelines GitLab disposent d'un job manuel de déclenchement qui appelle les mêmes scripts en cas de besoin ponctuel. J'ai veillé à ce que les environnements et variables nécessaires soient fournis par les credentials déjà gérés par l'équipe, sans introduire de mécanisme nouveau.

Pour éviter tout effet de bord, j'ai ajouté des garde-fous simples. Un seuil minimum d'espace disque à atteindre, paramétrable, en dessous duquel le job tente un second passage ciblé sur les volumes orphelins. Une liste de tags et de digest explicitement protégés, alimentée par les releases récentes, afin que le nettoyage Docker n'approche jamais les images de référence. Un âge minimal configurable pour les pods et jobs Kubernetes, avec une valeur par défaut conservatrice. Des logs verbeux écrits localement et archivés par Jenkins, ce qui rend l'audit très direct. Ces garde-fous ne complexifient pas le pipeline, mais ils rendent le comportement prévisible.

Enfin, j'ai travaillé la restitution. Un dashboard Jenkins synthétise l'état des derniers runs. Succès ou échec. Espace disque récupéré sur la période. Nombre d'objets supprimés par catégorie. À la fin du mois, je consolide ces informations pour fournir un court rapport qui met en regard les résultats du nettoyage et les incidents évités. Cette boucle de reporting m'aide à ajuster les critères. Par exemple, si je constate que les gains s'érodent, soit je dois resserrer l'âge cible, soit une nouvelle catégorie d'artefacts est en train d'apparaître dans les workflows et mérite un traitement dédié.

5.9. Optimiser l'architecture des jobs TRIGGER_TESTS

Sur Jenkins, j'ai créé un job par périmètre technique pour garder la lisibilité. Un job pour Docker sur les nœuds de build ciblés. Un job pour Kubernetes sur le cluster de test. Chaque job est autonome et s'exécute avec ses propres paramètres et garde-fous. Un job parent, TRIGGER_TESTS, orchestre la séquence sur un créneau précis. L'exécution commence par Docker sur les hôtes les plus sollicités. Elle se poursuit par le nettoyage Kubernetes sur le namespace de tests nightly. L'ordre n'est pas arbitraire. Libérer l'espace sur les nœuds de build avant d'attaquer la fenêtre de déploiement suivante est prioritaire. Le nettoyage des pods vient ensuite pour clarifier la lecture du cluster avant les nouveaux déploiements.
Sur GitLab CI, j'ai ajouté une cible pipeline utilitaire qui réutilise exactement les mêmes scripts. Elle est déclenchable manuellement par les responsables des nightly quand un signal local le justifie. Cette duplication contrôlée a un avantage. Si Jenkins est en maintenance, je peux toujours déclencher un nettoyage depuis GitLab CI. Inversement, si GitLab CI est surchargé, le cron Jenkins maintient le cycle minimum de propriété.

5.9.1. Critères de nettoyage et rétention

J'ai formalisé les critères dans le repository au même endroit que les scripts, sous forme de fichiers de configuration simples. Pour Docker, une liste blanche de tags et de digests couvre les trois dernières versions promues, ainsi que les images de base sur lesquelles reposent plusieurs services.

Une fenêtre de trente jours pour les images non référencées par des conteneurs actifs constitue la politique par défaut. Pour les conteneurs arrêtés, la fenêtre est plus courte, autour de sept jours, car leur intérêt décroît rapidement. Pour Kubernetes, la règle de sept jours pour les pods en Succeeded ou Failed et les jobs terminés s'est avérée suffisante pour les environnements de test.
Côté artefacts de type ISO, j'ai aligné la rétention sur ce qui avait été décidé par l'équipe Build and Release. Un nombre fixe de générations récentes conservées, plus une période calendaire minimale. Les suppressions se font toujours après vérification que les dernières versions nécessaires aux reproductions sont intactes. Je ne supprime jamais un artefact si le pipeline de promotion associé n'a pas été marqué comme complet.

5.9.2. Observabilité et reporting

La visibilité sur les effets concrets du nettoyage est essentielle. J'ai donc instrumenté chaque job pour qu'il publie des métriques simples. Espace disque avant et après. Nombre d'images supprimées. Nombre de conteneurs et de volumes supprimés. Nombre de pods et de jobs Kubernetes supprimés. Durée totale du job. Ces métriques sont stockées via les artefacts Jenkins et résumées sur un tableau de bord. Cela rend le gain tangible. Par exemple, après trois semaines, nous avons observé une stabilisation du pourcentage d'espace disque libre sur les nœuds critiques. Les pics de saturation ont disparu, et les nightly ont cessé d'échouer pour cause de manque d'espace au milieu d'un build.
Le reporting mensuel joue un second rôle. Il permet de corréler les périodes de nettoyage avec la fluidité des pipelines. Lorsque les jobs TRIGGER_TESTS passent à l'heure prévue et libèrent ce qu'il faut, les étapes de pull d'images dans les nightly deviennent plus rapides. Les logs le montrent clairement. Les téléchargements de couches déjà présentes en cache deviennent rares. Les erreurs liées à des images en doublon ou mal taguées se raréfient aussi, car la liste blanche nous force à clarifier les conventions de tag.

5.9.3. Résultats

Les effets se sont fait sentir rapidement sur la productivité et la stabilité. Les nightly ont retrouvé une durée plus stable, notamment sur les projets où les images sont volumineuses. Les pipelines ont cessé d'échouer pour des raisons triviales comme une saturation disque. La lecture de l'état du cluster de test est redevenue lisible, ce qui accélère les diagnostics quand un déploiement a réellement un problème. Au-delà des gains directs, le fait d'avoir formalisé les critères de rétention et d'avoir centralisé les scripts a amélioré la collaboration. Quand une équipe souhaite protéger une nouvelle image de base ou prolonger la rétention d'un artefact ISO, elle l'exprime dans la configuration versionnée au lieu de le faire au cas par cas sur une machine.

5.9.4. Limites et points d'attention

Même si le nettoyage apporte des bénéfices évidents, j'ai gardé à l'esprit deux limites. Premièrement, une politique de rétention trop agressive peut supprimer un artefact encore nécessaire à une reproduction particulière. C'est pour cette raison que la liste blanche est explicite et que les fenêtres d'âge sont conservatrices. Deuxièmement, le contexte d'un cluster peut évoluer.

De nouveaux types de ressources peuvent apparaître et réclamer un traitement dédié. J'ai donc prévu un espace dans le repository pour documenter ces extensions, tout en gardant la base simple. Le choix de créneaux horaires adaptés est un autre point d'attention. Nettoyer pendant une phase de build intense peut perturber les caches et provoquer l'effet inverse. J'ai calé la planification Jenkins en dehors des crêtes, en m'appuyant sur l'observation des charges et des historiques de pipelines.

5.9.5. Enseignements personnels

Ce chantier m'a appris à orchestrer des tâches d'entretien avec sobriété, en restant aligné sur les outils réellement utilisés par l'équipe. Je retiens l'importance de l'inventaire initial et des critères écrits noir sur blanc. Sans cela, le nettoyage devient arbitraire. Je retiens aussi l'intérêt des garde-fous simples. Un dry-run quand c'est possible, une liste blanche claire, des logs conservés et consultables, et des paramètres d'âge facilement ajustables.
D'un point de vue méthode, j'ai pris l'habitude de valider chaque évolution par une exécution locale contrôlée, puis de la promouvoir dans Jenkins et GitLab CI, exactement comme pour un déploiement applicatif. Cette discipline évite de transformer un job d'optimisation en source d'instabilité. Sur le plan de la collaboration, le fait de tout versionner au même endroit, scripts et configuration, facilite les contributions et la revue. Lorsque quelqu'un propose d'ajuster une fenêtre d'âge, on le fait via une merge request, ce qui garde la discussion et la décision près du code.

5.9.6. Perspectives

Les perspectives restent dans la continuité de ce que j'ai déjà mis en place. Je peux étendre la logique de nettoyage à d'autres classes d'artefacts produits par les nightly, tant qu'elles sont tracées et qu'une règle de rétention existe. Je peux améliorer la granularité côté Docker en croisant systématiquement les images candidates avec la liste des dernières promotions Jenkins ou GitLab pour renforcer la protection des références. Je peux également affiner la planification en fonction de l'évolution des créneaux de charge, en ajustant légèrement les horaires si de nouvelles fenêtres d'exécution apparaissent.
Je souhaite conserver l'esprit général. Partir de l'observation. Définir des critères simples et expliquables. Automatiser avec des outils déjà présents. Instrumenter et rendre visible ce qui se passe. Ajuster par petites touches à la lumière des résultats. Cette approche m'a permis d'augmenter la productivité sans complexifier la chaîne outillée, ce qui est essentiel dans un contexte où la vitesse de livraison et la fiabilité doivent progresser ensemble.

5.9.7. Conclusion

La création des jobs TRIGGER_TESTS a répondu à un besoin précis et récurrent. En gardant le périmètre centré sur Docker, Kubernetes, Jenkins, GitLab CI et des scripts Bash simples, j'ai réduit l'entropie des environnements nightly et de test, stabilisé la durée des pipelines, et rendu l'état des systèmes plus lisible au quotidien. La clé a été de relier chaque technologie à une tâche claire, avec une étape qui la précède naturellement et une autre qui la suit. Inventorier puis décider. Nettoyer

puis vérifier. Journaliser puis reporter. C'est cette chaîne courte, outillée avec sobriété, qui m'a permis d'obtenir des gains mesurables sans introduire de nouvelles dépendances ni de complexité inutile.
Écrit comme un étudiant de master au sein d'une équipe expérimentée, ce retour d'expérience reflète la manière dont j'ai posé le problème, choisi des solutions cohérentes avec l'existant et livré une amélioration continue qui tient dans le temps. Je reste concentré sur l'essentiel. Des environnements propres, des pipelines qui démarrent vite, des artefacts conservés pour la reproductibilité, et des scripts qui expliquent eux-mêmes ce qu'ils font. C'est une base solide pour poursuivre l'optimisation sans perdre de vue la fiabilité, qui reste le point d'appui de toute démarche DevOps dans un cadre industriel.

6. Difficultés rencontrées et leviers

Au fil de mon alternance, l'une des premières difficultés que j'ai rencontrées a été l'appropriation de l'écosystème très spécifique de GE HealthCare. L'entreprise, en tant qu'acteur majeur du secteur médical, utilise un vocabulaire technique et organisationnel propre, comprenant à la fois des termes liés aux technologies cloud et DevOps, et des notions strictement réglementaires liées à la conformité et à l'assurance qualité. Assimiler l'ensemble de ces concepts a nécessité un temps d'adaptation conséquent, notamment pour comprendre les processus internes encadrant la validation et la mise en production.
Parmi ces processus, la **Verification Readiness Review (VRR)** représentait un enjeu central. Cette étape consiste à fournir des preuves concrètes et traçables attestant de la conformité de chaque développement, tout en respectant les contraintes réglementaires et les exigences de qualité. La VRR impose de documenter rigoureusement chaque modification et de fournir des éléments vérifiables, ce qui exige à la fois rigueur, méthode et organisation. Cette contrainte s'est avérée être un véritable levier pour renforcer ma discipline professionnelle et ma capacité à produire des livrables précis et documentés, mais elle a également constitué une source de pression, surtout lorsqu'il fallait aligner rapidité d'exécution et exhaustivité des preuves.
Sur le plan technique, la gestion de la dette technique autour des tests **end-to-end (E2E)** a constitué un autre défi majeur. Les suites de tests existantes présentaient plusieurs fragilités : des sélecteurs instables qui entraînaient des erreurs aléatoires, des problèmes de synchronisation et des dépendances réseau fluctuantes. Ces instabilités rendaient les exécutions intermittentes et peu fiables, obligeant à relancer fréquemment les pipelines et à investir un temps important dans le débogage.
La diversité des pratiques entre les équipes représentait également un obstacle. Chaque équipe avait ses propres conventions et outils : certains utilisaient des **jobs Jenkins** traditionnels, d'autres des **runners GitLab CI**, ou encore des scripts Shell personnalisés, sans uniformisation sur la gestion des artefacts ou le versioning des builds. Cette hétérogénéité compliquait la collecte et la corrélation des résultats, ralentissant la prise de décision et la validation des pipelines.
Pour surmonter ces difficultés, plusieurs leviers ont été mobilisés : la documentation interne pour comprendre les processus, la collaboration étroite avec les collègues et référents pour accélérer l'appropriation des outils, et la mise en place progressive de standards pour uniformiser les pipelines et les pratiques. Ces mesures ont permis de réduire les incidents, de fiabiliser les tests et de gagner en efficacité opérationnelle.

7. Retours d’expérience et conclusion

Cette alternance chez GE HealthCare a été une expérience extrêmement enrichissante, tant sur le plan technique que professionnel. Elle m’a permis d’acquérir des compétences avancées dans plusieurs domaines clés du DevOps et du cloud, notamment la **gestion et l’optimisation des pipelines CI/CD avec Jenkins et GitLab CI**, la **contenerisation et le déploiement via Docker et Kubernetes**, ainsi que la **gestion d’artefacts et de versions avec Artifactory**.
J’ai également pu approfondir mes connaissances sur les **charts Helm**, l’écriture de **playbooks Ansible** pour des déploiements reproductibles et le monitoring des environnements MET et Pré-Prod. La mise en place de jobs automatisés, le nettoyage des artefacts, l’optimisation des nightly builds et la fiabilisation des tests E2E m’ont permis de comprendre l’importance de l’**industrialisation et de la standardisation dans un environnement cloud-native**, et comment cela influence directement la qualité et la sécurité des livrables.
Sur le plan organisationnel et humain, j’ai développé ma capacité à collaborer dans un environnement complexe, à communiquer efficacement avec des équipes pluridisciplinaires et à m’adapter à des processus très réglementés, tout en maintenant un rythme soutenu. L’expérience de la VRR m’a appris à documenter et justifier mes actions de manière rigoureuse, ce qui constitue un atout majeur pour ma future carrière dans le domaine du DevOps et du cloud computing appliqué au secteur médical.
Enfin, cette alternance m’a permis de constater que la combinaison de l’innovation technologique et de la conformité réglementaire est possible, à condition de mettre en place des pratiques robustes, des pipelines automatisés et une gouvernance claire des artefacts et des déploiements. Elle a également renforcé ma confiance dans mes capacités à mener des projets complexes et à contribuer de manière significative à l’industrialisation des systèmes logiciels à l’échelle internationale.
En conclusion, cette expérience a été un véritable tremplin pour ma carrière, en consolidant mes compétences techniques et organisationnelles, et en me donnant une vision concrète des enjeux de l’industrialisation de l’intelligence artificielle et du DevOps dans un environnement médical exigeant. Elle constitue un socle solide sur lequel je pourrai bâtir mes futures missions, en combinant expertise technique et rigueur professionnelle.

Table des figures

Webographie

- npm (Node Package Manager). Portail et registre des packages pour l'écosystème JavaScript/Node.js. https://www.npmjs.com/.

- Stack Overflow. Communauté de questions/réponses pour développeurs, résolution de problèmes techniques. https://stackoverflow.com.

- Kubernetes. Documentation officielle de l'orchestrateur de conteneurs. http://kubernetes.io.

- Docker. Site officiel et documentation de la plateforme de conteneurisation. http://www.docker.com.

- Helm. Gestionnaire de packages pour Kubernetes (charts) et documentation. http://helm.sh.

- Jenkins. Site officiel et documentation de l'automatisation CI. http://www.jenkins.io.

- GitLab CI/CD – Documentation. Référence officielle des pipelines CI/CD GitLab. http://docs.gitlab.com/ee/ci/.

- GitLab API – Documentation. Référence officielle de l'API GitLab. http://docs.gitlab.com/ee/api/.

- Groovy. Langage et écosystème (site et documentation). http://groovy-lang.org.

- JFrog Artifactory. Site officiel et documentation du gestionnaire d'artefacts. http://jfrog.com/artifactory/.

- Ansible. Site officiel et documentation de l'outil d'automatisation. http://www.ansible.com.

- SUSE. Site officiel de la distribution Linux SUSE et documentation. http://www.suse.com.