T.P Administrateur système DevOps
Dossier de projet - API Utilisateurs

Olivier DOSSMANN
emploi@dossmann.net

mentoré par

Jérémie TAROT

05 mars 2025

Version 1.3

Sommaire

1 Introduction **1**

2 Remerciements **2**
2.1 Via l'organisme de formation DataScientest . . . 2
2.2 Et d'autres encore ! . . . 2

3 Liste de compétences couvertes par le projet **3**
3.1 Introduction . . . 3
3.2 Couverture . . . 3
3.3 Conclusion . . . 4

4 Cahier des charges **4**
4.1 Contexte . . . 4
4.1.1 Dans le cadre d'une formation . . . 4
4.1.2 L'équipe DevU42 . . . 4
4.1.3 Organisation d'équipe . . . 5
4.1.4 Contraintes initiales . . . 5
4.2 Le projet . . . 6
4.2.1 Justification du projet . . . 6
4.2.2 Application existante . . . 6
4.2.3 Ambitions . . . 7
4.2.4 Objectifs . . . 7
4.3 Planification . . . 8
4.3.1 Étapes de travail . . . 8
4.3.2 Livrables . . . 10
4.4 Conclusion . . . 10

5 Spécifications techniques **11**
5.1 Étude de l'application . . . 11
5.1.1 Composants de l'application . . . 11
5.1.2 Outils fournis aux développeurs . . . 12
5.1.3 Points d'attention . . . 13
5.2 Infrastructure . . . 13
5.2.1 Couche basse . . . 13
5.2.2 Environnement Kubernetes . . . 14
5.2.3 Environnements . . . 16
5.2.4 Livraison continue et déploiement . . . 17
5.3 Services et outils divers . . . 18
5.3.1 Données . . . 18
5.3.2 Supervision . . . 21
5.3.3 Sécurité . . . 22
5.4 Conclusion . . . 22

6 Démarche suivie **23**
6.1 Méthodologie . . . 23

6.1.1 Agile . . . 23
6.1.2 Jalons . . . 23
6.1.3 Choix de standards et normes . . . 23
6.2 Outils . . . 25
6.2.1 Nom de domaine . . . 25
6.2.2 Plateforme DevOps . . . 26
6.3 Cas d'une collaboration inter-équipe . . . 29
6.4 Conclusion . . . 30

7 Mes réalisations **31**
7.1 Réalisation 1 : Livraison continue . . . 31
7.1.1 Contexte . . . 31
7.1.2 Analyse . . . 31
7.1.3 Solution . . . 33
7.1.4 Résultats . . . 36
7.1.5 Limites . . . 37
7.1.6 Amélioration(s) possibles(s) . . . 37
7.2 Réalisation 2 : États Terraform . . . 37
7.2.1 Contexte . . . 37
7.2.2 Analyse . . . 38
7.2.3 Solution . . . 38
7.2.4 Résultats . . . 40
7.2.5 Limites . . . 40
7.2.6 Amélioration(s) possibles(s) . . . 40
7.3 Conclusion . . . 41

8 Situation de travail **42**
8.1 Contexte et état des lieux . . . 42
8.2 Démarche . . . 42
8.2.1 Retranscription préalable . . . 42
8.2.2 Méthode de renseignement . . . 43
8.2.3 Point de départ . . . 44
8.2.4 Recherches . . . 44
8.3 Conclusion . . . 46

9 Conclusion **47**

1 Introduction

La **méthodologie Agile** étant de plus en plus adoptée, les **équipes de développement** publient de nouvelles versions de leur application avec une fréquence bien plus élevée. Les **équipes d'opération** ont désormais besoin d'un outillage plus conséquent pour déployer ces versions. C'est pour cette raison que l'**automatisation**, entre autres, est nécessaire pour soulager les équipes d'opération. Ainsi **le DevOps entre en jeu** : une culture et un ensemble de **bonnes pratiques** pour faciliter l'échange entre les équipes de développement et celles d'opération. Mais également de favoriser un **flux continu** entre ces équipes.

Au long de mon parcours, que ce soit en tant qu'**Ingénieur en Conception et Développement d'Environnements Distribués** ou **Responsable d'Applications**, j'ai été sensibilisé aux besoins incessants de l'automatisation et de l'amélioration des processus de création logicielle. J'ai participé à la mise en place progressive d'une **chaîne de publication logicielle**, que ce soit dans le monde professionnel ou dans la sphère privée (pour publier mon blog). C'est donc tout naturellement que je me suis tourné vers ce domaine passionnant qu'est le DevOps. Et ce à travers la **formation DevOps en BootCamp** au sein de l'organisme de formation DataScientest.

Le présent document abordera principalement le projet étudié lors de cette formation. Nous commencerons par faire la corrélation entre les compétences attendues pour le *Titre Professionnel Administrateur Système DevOps* et le(s) projet(s) présenté(s) dans ce dossier. Après quoi nous continuerons avec le cahier des charges et les spécifications techniques de ce(s) dernier(s).
Nous continuerons en expliquant la démarche suivie pour accomplir le projet, quelques réalisations effectuées ainsi qu'une situation de travail qui mérite d'être relevée.

Conventions typographiques

Ce document suit certaines conventions de mise en forme pour faciliter la lecture :

- Les liens hypertextes apparaissent sous la forme suivante : Titre du lien,
- Les références vers d'autres sections sont indiquées comme suit : référence vers un paragraphe spécifique, exemple avec DataScientest,
- Les mots-clés essentiels d'une section sont en **gras**,
- Les termes particuliers sont en *italique*,
- Les noms de famille apparaissent avec de petites majuscules : Prénom NOM,
- Les extraits de code sont affichés dans un format distinct :

```
1 #!/usr/bin/env bash
2
3 echo "Bonjour tout le monde !"
```

2 Remerciements

À **Madame Agnès SCHWEITZER** pour ses **conseils**, son aide **précieuse**, sa **présence d'esprit** et son **exceptionnelle intelligence** : MERCI !

2.1 Via l'organisme de formation DataScientest

Je remercie **l'ensemble de l'équipe DataScientest** pour leur patience et leur savoir-vivre au regard de mes nombreux retours sur les cours, les examens et les machines virtuelles fournies.

Une mention toute particulière aux personnes suivantes pour leur **écoute**, leur **professionnalisme** et leur souci de réponses claires :

- Sarah BOURAS,
- Benjamin FICHE,
- et Jérémie TAROT.

Aux **membres de l'équipe DevU42** qui, malgré les **difficultés**, une **vie sociale et familiale** prenante, des **obligations** et des **responsabilités** ont donné ce qu'ils pouvaient :

- Hrubech HOMBESSA,
- Eliel MONCADA,
- et Julien SABIOLS.

J'imagine que derrière chacune de ces personnes il y a également **un ou une partenaire de vie**. Vous méritez de figurer dans ces lignes et d'avoir un grand MERCI !

Aux **membres de la « cohorte »** que je tiens à remercier. À ceux qui ont participé aux discussions, qui ont voulu échanger, qui ont accepté de répondre à des questions, qui ont partagé des difficultés ou encore des solutions et qui étaient eux-mêmes finalement.

Une mention aux personnes suivantes :

- à Maxime BOULANGHIEN,
- à Michael LACHAND,
- à Philippe RISSER-MAROIX,
- et à Dorian ROLY.

2.2 Et d'autres encore !

D'autres personnes n'ayant aucun lien avec cette formation ont pourtant participé à leur façon. Je tiens à remercier notamment Grégoire STEIN, en sa qualité de **DevSecOps**, pour ses **précieux conseils** tout au long de cette aventure !

À Richard DERN et son œil d'aigle pour les nombreuses relectures et fautes d'orthographes découvertes dans ce document.

3 Liste de compétences couvertes par le projet

3.1 Introduction

Le référentiel de compétences du *Titre Professionnel Administrateur Système DevOps* liste 11 compétences réparties en 3 catégories :

- Automatiser le déploiement d'une infrastructure dans le cloud
  - Compétence **n°1 : Automatiser la création de serveurs à l'aide de scripts**
  - Compétence **n°2 : Automatiser le déploiement d'une infrastructure**
  - Compétence **n°3 : Sécuriser l'infrastructure**
  - Compétence **n°4 : Mettre l'infrastructure en production dans le cloud**
- Déployer en continu une application
  - Compétence **n°5 : Préparer un environnement de test**
  - Compétence **n°6 : Gérer le stockage des données**
  - Compétence **n°7 : Gérer des containers**
  - Compétence **n°8 : Automatiser la mise en production d'une application avec une plateforme**
- Superviser les services déployés
  - Compétence **n°9 : Définir et mettre en place des statistiques de services**
  - Compétence **n°10 : Exploiter une solution de supervision**
  - Compétence **n°11 : Échanger sur des réseaux professionnels éventuellement en anglais**

3.2 Couverture

En utilisant les critères de performance fournis par chaque fiche de compétence, j'ai pu établir le tableau suivant permettant de voir les points à améliorer :

Table 1: Couverture de compétences du projet

Compétence	Nbre critères perf.	Couverte ?	Note
Compétence n°1	3/3	Oui	EC2 / Terraform
Compétence n°2	3/3	Oui	Terraform
Compétence n°3	2/2	Oui	Staging, Vaultwarden, Let's Encrypt
Compétence n°4	**2**/3	Oui	Terraform → « prod »
Compétence n°5	4/4	Oui	Gitlab CI
Compétence n°6	**1**/3	Partiellement	postgresql-ha, Velero
Compétence n°7	4/4	Oui	Docker
Compétence n°8	**3**/4	Oui	Gitlab CI, Terraform, staging/prod
Compétence n°9	**0**/3	Non	Aucun interlocuteur
Compétence n°10	**1**/3	Partiellement	Datadog
Compétence n°11	3/3	Oui	Tout au long du projet

La couverture de la **compétence n°6** (Gérer le stockage de données) manque de tests concernant le système de sauvegarde qui a été développé mais pas déployé, **faute de tests et de temps**.

La **compétence n°9** (Définir et mettre en place des statistiques de services) n'a pu être clairement établie **faute d'interlocuteur autre que le « mentor »**.

La **compétence n°10** (Exploiter une solution de supervision) est **en lien avec la compétence précédente**, ce qui influe sur la pertinence des moniteurs mis en place dans la solution.

3.3 Conclusion

Le projet décrit dans le présent document couvre de manière complète une énorme partie des compétences attendues. Au final 2 type de compétences sont partiellement couvertes : celle concernant la supervision et celle concernant le stockage de données.

4 Cahier des charges

Dans le cadre d'une **formation, dans l'école DataScientest**, il a été demandé à des équipes composées de 3 à 5 personnes de travailler sur un **projet défini, cadré** et suivant un certain nombre de contraintes dont nous allons parler ici.

Le cahier des charges a été la première réflexion sur le projet.

4.1 Contexte

Commençons par le contexte du projet avec l'ensemble des intervenants et des contraintes attenantes.

4.1.1 Dans le cadre d'une formation

DataScientest est une **école de formation** créée en 2017, située sur Paris et proposant **plus de 18 formations différentes**, dont *Administrateur Système DevOps*.

Plusieurs formats de formation existent parmi **Bootcamp**, continue et en alternance. **Ce projet est un exercice de fin de formation** proposé à tous les élèves de la « cohorte » (équivalent d'une classe).

Plusieurs élèves se regroupent autour d'un projet pour lequel ils ont un **intérêt commun**.

4.1.2 L'équipe DevU42

Le **groupe DevU42** s'est formé le **17 octobre 2024**. Il est composé de **quatre étudiants** participant à la formation **Administrateur Système DevOps** de l'organisme de formation DataScientest.

Ses membres sont, par ordre alphabétique :

- **Hrubech HOMBESSA**,
- **Eliel MONCADA**,
- **Julien SABIOLS**,
- et Olivier DOSSMANN.

Ils interviennent sur le projet en qualité d'**Administrateur Système DevOps**.

L'équipe a été supervisée et suivie par **Jérémie TAROT** sous la dénomination de « **Mentor** ». Et ainsi en sera-t-il dans le présent document.

4.1.3 Organisation d'équipe

Pour s'organiser, l'équipe s'est dotée de plusieurs outils et suivi certaines méthodologies particulières.

4.1.3.1 Rapports hiérarchiques

L'équipe s'articule autour de membres ayant le **même degré de responsabilités** et d'objectifs. Leur périmètre et leur champ d'action est similaire qu'il s'agisse de déterminer les tickets à prendre en charge, les axes de travail à rajouter au projet ou bien de nouveaux objectifs.

En cas de désaccord sur un sujet, une décision est prise à la majorité votante, avec pour ultime recours l'avis du Mentor.

4.1.3.2 Moyens de communication

La géolocalisation des membres de l'**équipe étant éparse**, plusieurs moyens de communication **en ligne** ont été nécessaires, parmi lesquels :

- Canaux de **discussions sur Slack** fournis initialement par DataScientest :
  - canal du projet - **sep24_bootcamp_devops** : favorise l'échange entre DevU42 et le Mentor,
  - canal d'équipe - **devu42** : pour les **discussions techniques**, les choix décisifs et de la bonne humeur !
- Un nom de domaine, **devu42.fr**,
- Une boîte courriel, **team@devu42.fr** partagée pour la **gestion des services**,
- **Groupe DevU42 sur Gitlab**, un outil intégré avec gestion d'équipe, gestion de projet, gestion de tickets, gestion de tableau de bord (kanban), gestion de dépôts, gestion de paquets, etc.
- Un **wiki**, DevU42 sur Gitlab (fourni par Gitlab) permettant de rassembler les **connaissances de l'équipe** en un seul endroit,
- **Framapad**, un éditeur de **texte collaboratif** pour les divers travaux de rédaction.

4.1.4 Contraintes initiales

Deux principales contraintes initiales avaient été imposées dans le cadre de la formation :

- le **budget**,
- le **temps**.

4.1.4.1 Budget

Le budget alloué au projet est de **150€ mensuel** pour couvrir l'intégralité des frais issus de l'utilisation de **services Amazon** (AWS).

C'est une limite configurée par DataScientest sur l'espace Amazon qu'ils nous partagent.

4.1.4.2 Temps

Le temps alloué pour réaliser ce projet avant un premier lancement officiel est fixé à **7 semaines**.

Au terme des 7 semaines, un support visuel de présentation du projet pour son lancement est attendu (Cf. Section du présent chapitre sur les Livrables).

Après livraison, l'amélioration du projet est possible, bien qu'un dossier professionnel, un dossier de projet (ci-présent) et un support de présentation est demandé à chacun des membres de l'équipe DevU42.

4.2 Le projet

Le point de départ du projet est un dépôt de version contenant une application à mettre en ligne. Justification du projet, l'application elle-même, ambitions et objectifs sont des points à soulever.

4.2.1 Justification du projet

L'école de formation DataScientest s'évertue à proposer une **situation proche de la réalité** afin que l'équipe nouvellement formée puisse **apprendre par la pratique**.

On imagine facilement le cadre :

> Une entreprise ayant une application de gestion d'utilisateurs souhaite déployer et faire évoluer son service en favorisant les bonnes pratiques suivies dans la culture DevOps.

De la même manière, l'entreprise pourrait rencontrer de nombreux **problèmes** :

- **Reprise d'activité** lente, coûteuse et avec pertes d'informations,
- Service instable, **disponibilité** non garantie, **vulnérabilité** du service,
- Capacité limitée pour encaisser des **augmentations** ponctuelles de la **charge** et du **trafic**,
- Absence de **supervision** et faible capacité à prévoir les problèmes,
- Manque de contrôle sur la **qualité du code** ; procédures de développement pouvant être améliorées par l'**automatisation**,
- Perte de temps et erreurs humaines liées aux actions manuelles,
- Manque de **portabilité de l'application** d'un système à l'autre,
- Lenteurs d'implémentation de **nouvelles fonctionnalités** avec interruptions de service.

4.2.2 Application existante

Pour bien saisir l'état des lieux de ladite application, regardons de plus près.

L'**application API Utilisateurs** (nom de code FastAPI-Træfik) est fournie avec les éléments principaux suivants :

- Backend utilisant le framework **FastAPI** (en Python),
- Frontend statique (en **React**, **Vite.js** et **Chakra UI**) : Interface de gestion et d'enregistrement d'utilisateurs utilisant l'API du backend,
- Base de données **postgreSQL**,
- et **Traefik** (Proxy inverse).

Tout ceci est déposé dans un **dépôt de version** avec de la **documentation** en fichier *Markdown* (*.md) bruts.

De simples fichiers **Docker** et **docker-compose.yml** permettent aux développeurs de **lancer l'application localement**.

Une procédure manuelle existe pour expliquer comment mettre l'application en production sur une machine serveur dédiée.

4.2.3 Ambitions

Le projet **va plus loin** ; en dehors des objectifs à atteindre, il vise à :

- répondre aux problématiques de l'entreprise sur l'application existante par **la méthode DevOps**,
- répondre aux attentes de l'examen en matière de compétences attendues (méthode d'**apprentissage par la pratique**),
- démontrer un **savoir-faire technique** sur les sujets principaux du DevOps, notamment vis à vis du Cloud **en utilisant AWS** (Amazon Web Services),
- appliquer une méthode de travail qui optimise l'efficacité de la production des livrables.

4.2.4 Objectifs

C'est de ces **ambitions** que naîssent plusieurs objectifs spécifiques :

- balayer l'ensemble des **sujets liés à la culture DevOps et ses méthodes**,
- porter, puis **déployer** une application **sur le cloud**,
- préparer une **architecture complète** pour accueillir une à plusieurs applications,
- respecter un **budget** et des **limites de temps**,
- **déployer en continu** un projet en passant par une phase de test, de pré-production puis de production,
- penser à la **gestion des données**,
- **superviser** un système déployé en production avec des tableaux de bord efficaces,
- rendre un système **résilient** en prenant en compte les augmentations ET les diminution de **charges** et par une **haute disponibilité**,
- **automatiser** la plupart des actions afin de **limiter les erreurs humaines**,
- procéder, de manière automatique, à des vérifications sur **la sécurité** des éléments produits et déployés,
- favoriser une **collaboration** plus fluide entre les équipes de développement et les équipes d'opération,
- et obtenir des **alertes** en cas de défaillance d'un ou plusieurs éléments du système.

Ce sont des **objectifs plutôt ambitieux pour un temps aussi court**. Un travail d'équipe et une organisation de cette dernière sont totalement primordiaux.

4.3 Planification

Après quelques échanges avec le mentor nous avons établi le plan d'action pour les 7 semaines de travail à venir ainsi qu'une liste de livrables attendus.

4.3.1 Étapes de travail

Le plan s'articule en étapes bien définies.

4.3.1.1 Semaine 1 : Cadrage

Sous le signe de l'**encadrement**, cette première semaine est dite « douce » :

- **réunion** de prise de contact, présentation et cadrage,
- introduction de chaque membre du projet,
- **découverte** de l'application,
- **analyse** du sujet,
- évaluation du contexte, des objectifs et du **périmètre** du projet,
- et **rédaction** du premier jet du présent **cahier des charges**.

Il faut ce temps pour découvrir de quoi il retourne.

4.3.1.2 Semaine 2 : Planification

Vient ensuite une **phase d'organisation** avec :

- définition du contexte, des objectifs et du périmètre du projet,
- **organisation** des exigences,
- **identification des composants** d'architecture,
- choix de **solutions à implémenter**,
- et **définition de l'architecture** et les **spécifications** du système cible.

Ce qui amène à **compléter le cahier des charges**.

Nous prévoyons aussi :

- une **configuration de Gitlab**, des principaux dépôts, des branches, etc.,
- de s'organiser en équipe : définir des **méthodologies** à suivre.

4.3.1.3 Semaine 3 : Automatisation de la chaine d'intégration et de livraison

Une fois les outils principaux mis en place, il s'agit d'**automatiser** plusieurs parties :

- automatisation des **tests pour le backend et le frontend**,
- création d'**image Docker pour le backend**,
- création d'**image Docker pour le frontend**,
- automatisation des images Docker et de leur **publication sur des registres d'images**,

- création de **chart Helm pour le backend**,
- création de **chart Helm pour le frontend**,
- automatisation de la création des chart Helm et de leur **publication sur des registres d'images**,
- et **automatisation de la création de l'infrastructure** (même si elle n'est pas encore clairement définie).

L'**adoption d'un workflow Git** pour la gestion des branches est la bienvenue.

4.3.1.4 Semaine 4 : Conception de l'infrastructure

Vient ensuite la création de la **couche basse de l'infrastructure** :

- **création d'un schéma** de la couche basse de l'infrastructure,
- description de cette infrastructure via des modules adaptés et en adoptant l'**Infrastructure as Code (IaC)**,
- premiers **choix de solutions** pour le **stockage**, les **sauvegardes**, la **supervision** et l'**environnement d'accueil** de l'application,
- activation des **certificats** Let's Encrypt,
- liaison avec le **nom de domaine**,
- et **mise à l'échelle automatique**.

Une semaine n'est pas suffisante pour traiter tous les sujets. En revanche cela permet de les aborder et d'échanger sur ces derniers.

4.3.1.5 Semaine 5 : Gestion des données

Possiblement en parallèle du sujet de la semaine 6 sur l'Observabilité et des alertes, la gestion des données est importante et passe par :

- créer et configurer des **ressources de stockage de données**,
- mise en place de **politique d'authentification** et d'autorisation,
- **import des données**,
- et installer/configurer une solution de **sauvegarde** et de **restauration de données**.

C'est un sujet bien à part. D'où la possibilité de **faire cette étape en parallèle** d'une autre.

4.3.1.6 Semaine 6 : Observabilité et alertes

Quant à la semaine 6, nous avons **principalement la supervision** à gérer :

- configuration du **système de supervision centralisé**,
- mise en place d'une **collecte** de surveillance **pertinente** pour l'application,
- **choix des métriques** à utiliser pour la création de tableaux de bords,
- création des **tableaux de bords**,
- **définition des niveaux d'alertes** pour les métriques choisies,
- mise en place des **alertes**,
- et envoi des alertes sur **les canaux choisis**.

Ces éléments pourront être complétés et étendus une fois la solution principale de supervision configurée.

4.3.1.7 Semaine 7 : Présentation de la solution finale

Cette dernière semaine ciblera la **présentation et** le **lancement officiel de l'application**. Il faudra donc :

- compléter les **documentations**,
- **mettre à jour les schémas** en fonction d'éventuels changements,
- préparer un **support visuel de présentation du projet**,
- veiller à ce que l'**application tourne sans accrocs** pour la présentation,
- et résoudre les derniers problèmes majeurs connus - s'ils existent.

C'est un **plan ambitieux** quand on considère que **les membres de l'équipe proviennent d'horizons et de formations variés**.

4.3.2 Livrables

Sont attendus les **livrables** suivants :

- un **cahier des charges**,
- une **présentation par diapositives** pour le lancement de l'application,
- une application **déployée en production**, sans bugs,
- un **système de supervision** de la production,
- une **gestion des données** avec un **système de sauvegarde**,
- un **schéma de la couche basse de l'infrastructure** utilisée dans le Cloud pour la production,
- un **accès aux dépôts et scripts** permettant l'automatisation du déploiement de l'application et de la génération des différents éléments (images Docker, Chart Helm, registres, etc.),
- et un **fonctionnement en haute disponibilité** de l'application.

4.4 Conclusion

Le projet **couvre beaucoup de situations** et **domaines** autour de **l'automatisation** et la **mise en production**, dans le Cloud, d'une application avec une **supervision** et une **mise à l'échelle** de cette dernière tout en gérant une **base de données**, le **stockage** et la **sauvegarde**.

En groupe, c'est un projet idéal pour suivre l'« **apprentissage par la pratique** » proposé par DataScientest et couvrir un maximum de compétences demandées pour le *Titre Professionnel d'Administrateur Système* DevOps.

Cela demande de suivre le plan à la lettre dans les **limites de temps et de budget** définies : c'est un projet ambitieux.

5 Spécifications techniques

Après avoir décrit, via un cahier des charges, les différents besoins du projet, les livrables attendus, il est temps de décrire une solution. Nous aborderons avant tout l'étude de l'existant puis enchaînerons sur le détail de l'infrastructure choisie pour terminer sur divers sujets intégrés et quelques outils utilisés.

5.1 Étude de l'application

Le dépôt Git contenant l'application **existait déjà**. Une étude des composants fournis a donc été nécessaire, ce qui a permis d'en mesurer les implications.

5.1.1 Composants de l'application

L'équipe de développement a fait le choix du **monorepo** pour contenir l'ensemble des éléments de l'application. L'application est composée de :

- une **base de données** PostgreSQL,
- un **backend** écrit en Python et construit sur **FastAPI**,
- un **frontend** écrit en TypeScript et construit avec **ReactJS**.

L'ensemble se comprend mieux à l'aide du schéma suivant :

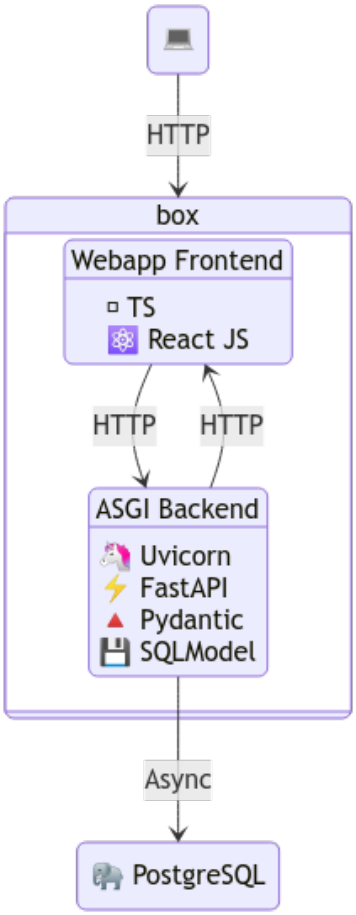

Figure 1: Schéma des composants de l'application

5.1.2 Outils fournis aux développeurs

Via le dépôt, les développeurs ont accès à :

- une documentation principale via un fichier **README.md**,
- une documentation secondaire via :
  - un fichier **backend/README.md** pour les développeurs du Backend,
  - un fichier **frontend/README.md** pour les développeurs du Frontend,
- plusieurs fichiers **Docker Compose** afin de déployer localement :
  - une **base de données (BDD) PostgreSQL** avec un **jeu de données préchargé**,
  - un accès à la BDD via une interface Web, via l'outil nommé **adminer**,
  - le **backend** (FastAPI),
  - le **frontend** (ReactJS),
- une documentation **deployment.md** pour déployer le projet à l'aide de **Docker Compose**,
- et un fichier **docker-compose.traefik.yml** afin de déployer une instance locale du **proxy Traefik** configuré pour utiliser Docker.

Ces éléments sont un **bon point de départ pour l'équipe de DevOps** qui prendra connaissance des particularités de cette application.

5.1.3 Points d'attention

La **partie Frontend** permet de générer des fichiers dits « statiques », du **serverless** est envisageable. L'application, bien que faisant des appels à une API, fonctionne de manière **autonome**. Ce qui, a priori, facilite la préparation pour un déploiement. Cependant les développeurs de l'application utilisent **un nom de domaine inscrit en dur dans le fichier Dockerfile**, ce qui peut compliquer un déploiement sur plusieurs domaines et environnements. En effet, une fois l'image Docker compilée, il sera impossible de changer le nom de domaine. Déployer une telle image limite donc l'usage à un seul domaine. Si on souhaite déployer sur plusieurs domaines (pour un environnement de pré-production et un environnement de production par exemple), il faudra faire une image Docker par domaine !

La **partie backend** s'appuie sur une base de données de type PostgreSQL. Elle est donc **dépendante de la BDD**.

La **partie proxy** utilise actuellement **Traefik comme outil**.

Ces trois parties ont une influence sur le choix de l'infrastructure à mettre en place. Il reste cependant une **marge de manœuvre sur la partie proxy**. En effet il n'est pas obligatoire d'utiliser Træfik en production.

5.2 Infrastructure

Afin d'accueillir l'application dans le Cloud, l'infrastructure de base va poser les fondations de l'environnement complet nécessaire à un usage quotidien.

5.2.1 Couche basse

La couche basse pourrait se définir comme le minimum nécessaire avant d'installer quoique ce soit pour exploiter l'application. En ce sens, on utilise des éléments de base d'Amazon Web Services (AWS) pour fabriquer notre infrastructure :

- un espace privé, Virtual Private Cloud (**VPC**),
- sur plusieurs **régions**,
- avec plusieurs zones de disponibilités (**Availability Zone**),
- contenant chacune, d'un point de vue réseau informatique, des réseaux publics (**public subnet**) et des réseaux privés (**private subnet**),
- en utilisant une passerelle entre le VPC et Internet, appelée **Internet Gateway** (IGW),
- en utilisant des passerelles **NAT** (Network Access Translation) entre les réseaux privés et les réseaux publics pour un accès à l'IGW,
- avec des groupes de sécurité (**Security groups**) pour définir les règles d'entrée/sortie,
- et des équilibreurs de charge de type réseau (**Network Load Balancer**) afin de gérer les accès aux services de manière équilibrée.

Le schéma de la couche basse de l'infrastructure ressemble donc à :

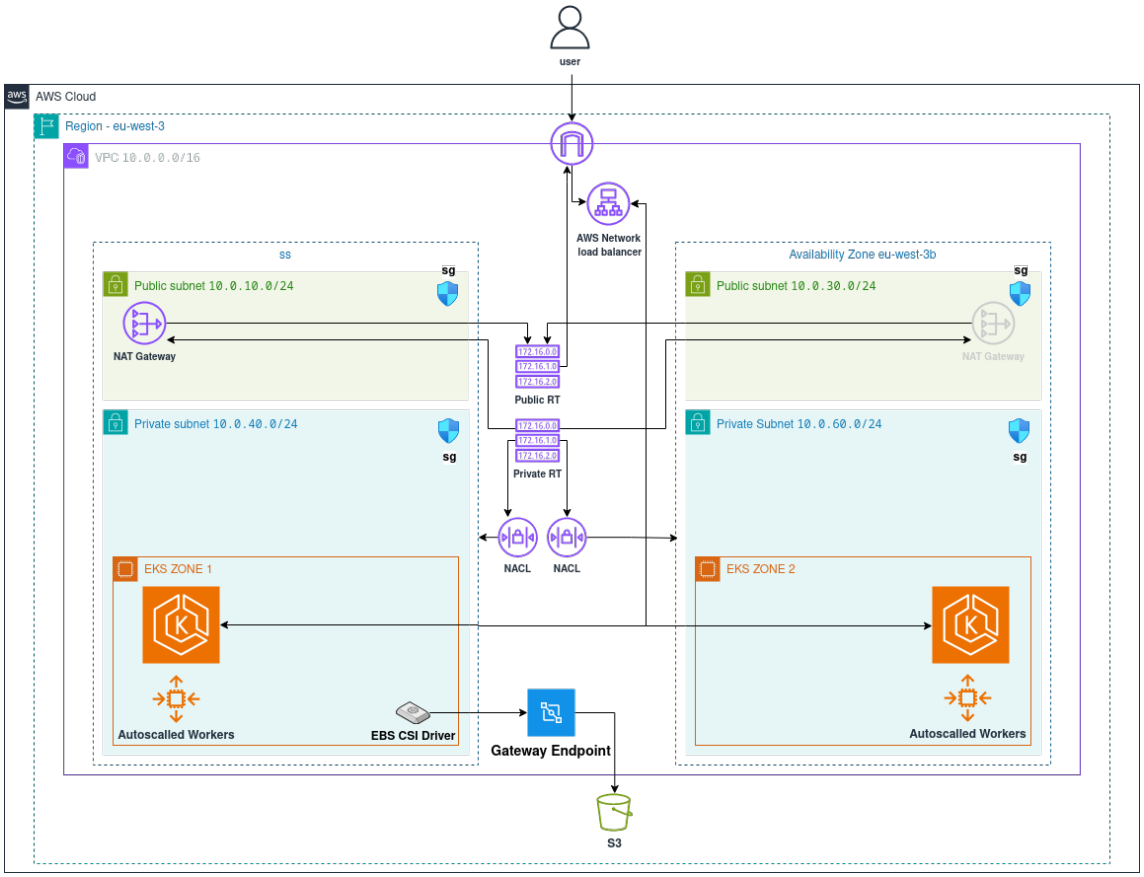

Figure 2: Schéma de la couche basse de l'infrastructure

5.2.2 Environnement Kubernetes

C'est sur la couche basse qu'il est possible d'ajouter un environnement permettant d'intégrer, déployer, automatiser et gérer des applications conteneurisées. Pour cela nous choisissons **Kubernetes**.

Kubernetes est un système capable de faire tourner des **applications conteneurisées**, d'**automatiser le déploiement** et de gérer la mise à l'échelle (**auto-scaling**).

Sur l'infrastructure présentée dans le paragraphe précédent il va falloir ajouter quelques éléments comme :

- un cluster Kubernetes appelé EKS (**Elastic Kubernetes Service**) géré par Amazon,
- avec l'usage d'une mise à l'échelle (**autoscaling**) des **serveurs virtuels EC2**(Elastic Container) d'Amazon,
- un équilibreur de charge de type réseau entre l'extérieur du cluster et le cluster (**AWS Elastic LoadBalancer** ou ELB) ; présent d'ailleurs dans le schéma de la couche basse de l'infrastructure dans le paragraphe précédent,
- un équilibreur de charge de type réseau (couche 4 du modèle OSI - Open Systems

Interconnection) qui fait le lien entre les services et l'extérieur du cluster : **NGINX Ingress Controller** ; il va notamment échanger avec l'AWS Elastic LoadBalancer précédent,
- l'utilisation d'un agent **externaldns** qui va mettre à jour les routes DNS (Domain Name System) de **Route53** pour les sous-domaines utilisés (~16€/mois pour 3 routes),
- et avec l'usage de **cert-manager** pour faire la demande de certificats SSL (Secure Sockets Layer) via le service - gratuit - de **Let's Encrypt** pour nos sous-domaines.

Le contenu du cluster EKS, une fois implémenté, pourrait ressembler à ceci :

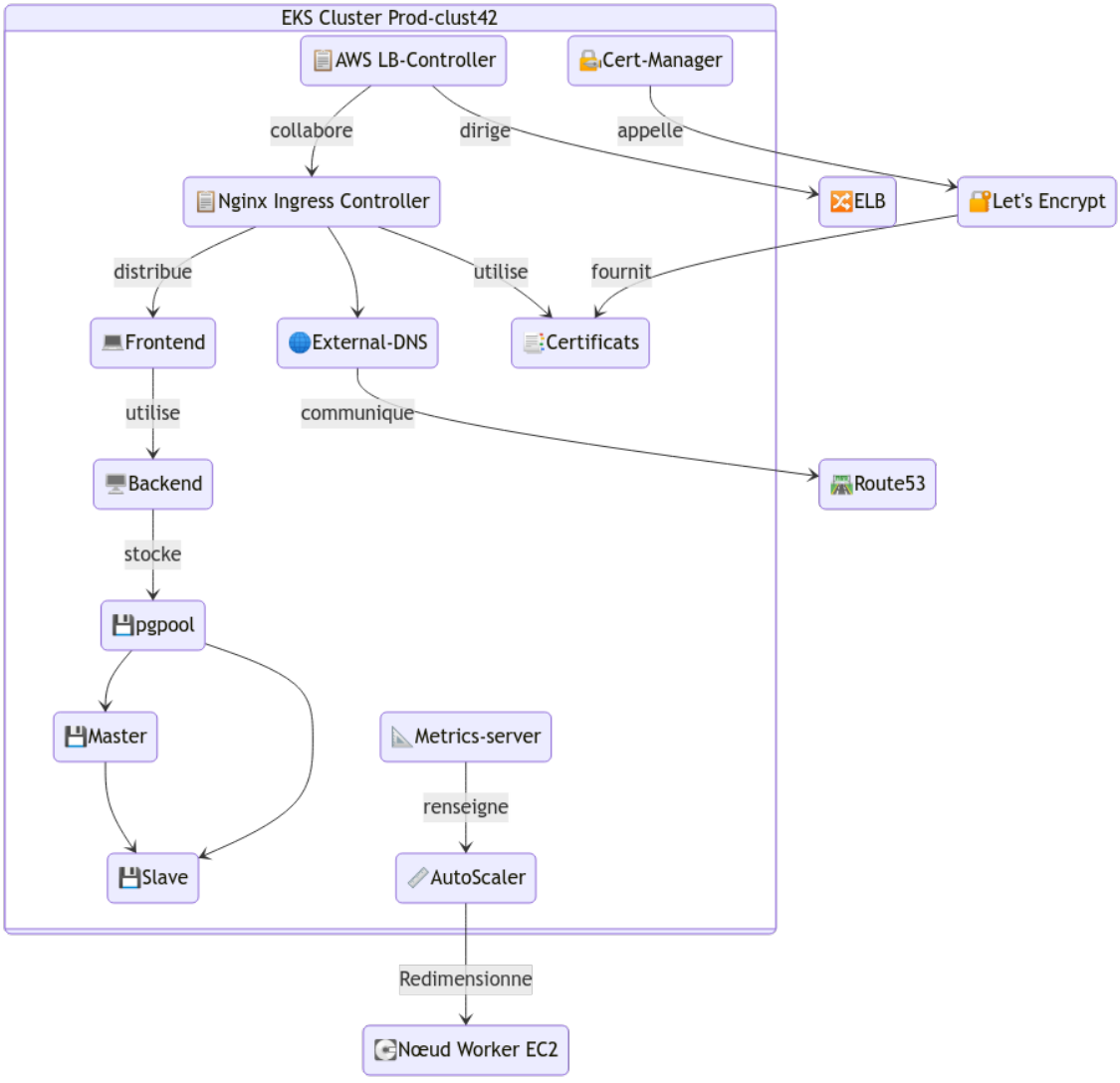

Figure 3: Schéma des applications et services contenu(e)s dans le cluster EKS

Pourra tourner dans le cluster, en plus des agents précédemment cités, **les applications suivantes** :

- le serveur de BDD **postgreSQL**,

- le **backend** de l'application (FastAPI),
- et le **frontend** de l'application (en fichiers statiques).

Afin de permettre une **livraison régulière** de l'application, il va falloir **automatiser au maximum** les étapes entre la publication d'une nouvelle version de l'application et son déploiement en production en passant par la livraison en environnement de pré-production par une intervention humaine.

5.2.3 Environnements

Nous parlons de production depuis un moment. À quoi cela correspond ?

Afin de mener à bien le travail de DevOps, nous avons défini les différents environnements par lesquels nous allons passer pour toute notre chaîne de production logicielle. Ainsi **nous avons 4 environnements** :

- l'environnement de **développement** : c'est l'environnement disponible sur **chaque machine des développeurs** à l'aide des outils fournis (Dockerfile, docker-compose du projet applicatif),
- l'environnement de **test** : il est composé de l'**ensemble des Runner Gitlab** utilisés à l'intérieur de la chaîne d'intégration continue,
- l'environnement de **pré-production** (appelé « **staging** ») : c'est l'infrastructure présentée ci-avant, appliquée dans AWS avec une configuration dite *plus légère* (moins de machines virtuelles EC2 par exemple),
- et l'environnement de **production** : c'est l'**infrastructure finale** ; celle qui est disponible aux utilisateurs finaux (les usagers).

Le schéma des différents environnements est plus compréhensible :

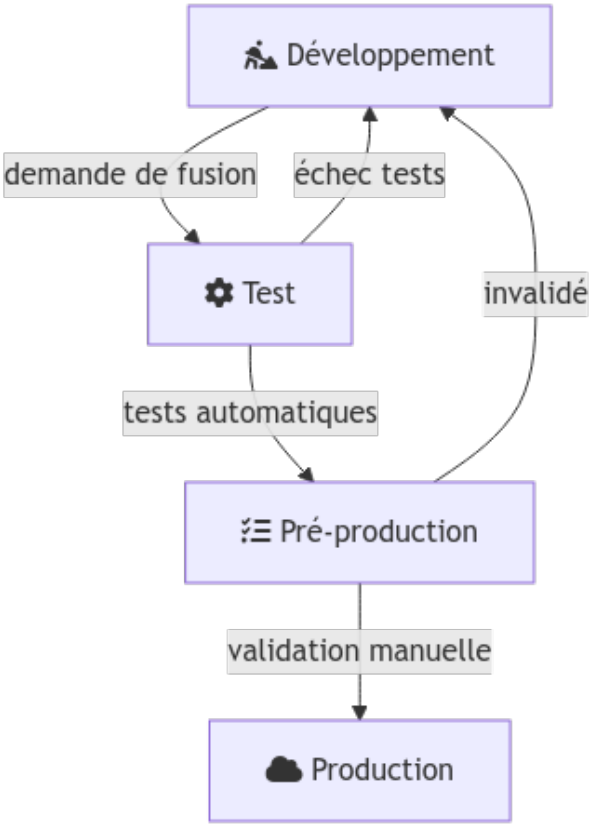

Figure 4: Schéma des environnements utilisés

5.2.4 Livraison continue et déploiement

Il est visé une **CI/cd**, c'est à dire une **Intégration Continue** (Continuous Integration) et une **livraison continue** (continuous delivery). **L'automatisation est donc essentielle**.

Le déploiement en production passera par une validation manuelle.

Doivent être automatisés :

- les **tests du backend et du frontend**,
- l'étude de **sécurité sur les dépendances des composants** de l'application,
- la **génération** d'une **image Docker** pour le **frontend** et le **backend**,
- la **publication** de cette image sur un **dépôt d'images**,
- le **déploiement d'un environnement de pré-production** (appelé *staging*, dont nous parlerons dans le chapitre Démarche suivie),
- le **déploiement**, une fois la validation manuelle sur l'environnement de pré-production faite, **d'un environnement de production**,
- et la **mise en place initiale** de l'infrastructure (appelée **boostrap**).

Nous retrouverons ces éléments dans les chapitres suivants.

5.3 Services et outils divers

Plusieurs sujets tournent autour des services mis en place :

- les **données** : définit les éléments enregistrés à l'usage quotidien de l'application,
- la **supervision** : définit tous les éléments permettant de mesurer les différents services et d'alerter si les métriques dépassent des seuils définis,
- la **sécurité** : définit tout ce qui peut être mis en œuvre pour palier rapidement à des problèmes de sécurité de l'application, de l'infrastructure et des outils utilisés.

5.3.1 Données

Le sujet des données ne couvre pas uniquement les données brutes enregistrées par l'application, mais aussi la manière de les stocker, de les sauver via un plan de sauvegarde et de les restaurer si besoin.

5.3.1.1 Où sont les données ?

Les données de ce projet sont multiples. En voici une liste non-exhaustive :

- les dépôts contenant le **code applicatif** (backend + frontend),
- les dépôts contenant la **description de l'infrastructure**,
- les données brutes de l'application, stockées dans la **base de données postgreSQL**,
- les **données sensibles** (mots de passe, clés d'accès),
- les **copies de sauvegardes** des données elles-mêmes,
- et les **fichiers de journalisation** des services en production.

Regardons pour chacun des points de cette liste où se situe le stockage.

5.3.1.2 Stockage

Les éléments précédents donnent lieu au tableau suivant :

Table 2: Données présentes et leur lieu de stockage

Données	Stockage
Code applicatif	**dépôt Gitlab**
Code de l'infrastructure	**dépôt Gitlab**
Base de données postgreSQL	**stockage AWS EBS** (Elastic Block Store)
Sauvegardes	**stockage AWS S3** (Simple Storage Service)

Cela devrait être suffisant pour stocker les diverses données disponibles.

5.3.1.3 Base de données postgreSQL

Pour le choix d'un service de base de données, nous avons fini par opter pour **postgresql-ha** (pour postgreSQL High Availability) qui s'appuie sur **2 bases de données répliquées** avec un service **pgpool** par lequel on accède pour chaque requête faite à la base de données.

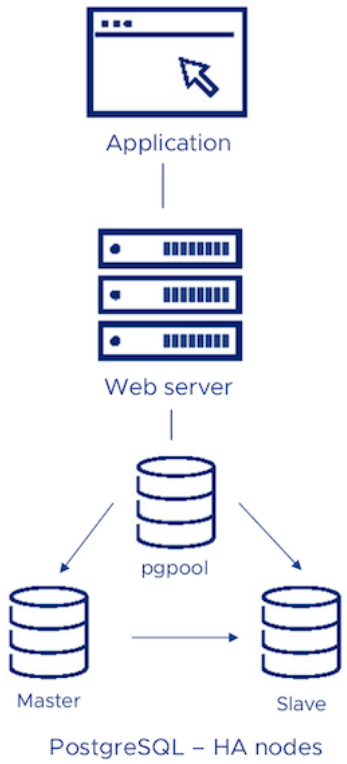

Figure 5: Schéma du fonctionnement de postgreSQL-HA

5.3.1.4 Sauvegardes

Les dépôts de version utilisés étant sur Gitlab, service qui se targue d'être entre 99% et 100% disponible , une **sauvegarde dite « préventive » mensuelle** suffit.

En revanche **la production**, même en **étant un service en haute disponibilité**, doit fournir au moins **une sauvegarde 1 fois par jour**. L'outil de sauvegarde choisi est **Velero**. Compatible avec l'environnement Kubernetes et permettant aussi de **sauver l'état du cluster** à un instant *T*. Son outil en ligne de commande permet de gérer les sauvegardes mais aussi les restaurations de ces dernières.

Comme énoncé précédemment, les sauvegardes du cluster seront mises **sur un stockage AWS S3** (dit « bucket S3 »).

Le fonctionnement de Velero dans ce projet est visible sur le schéma suivant.

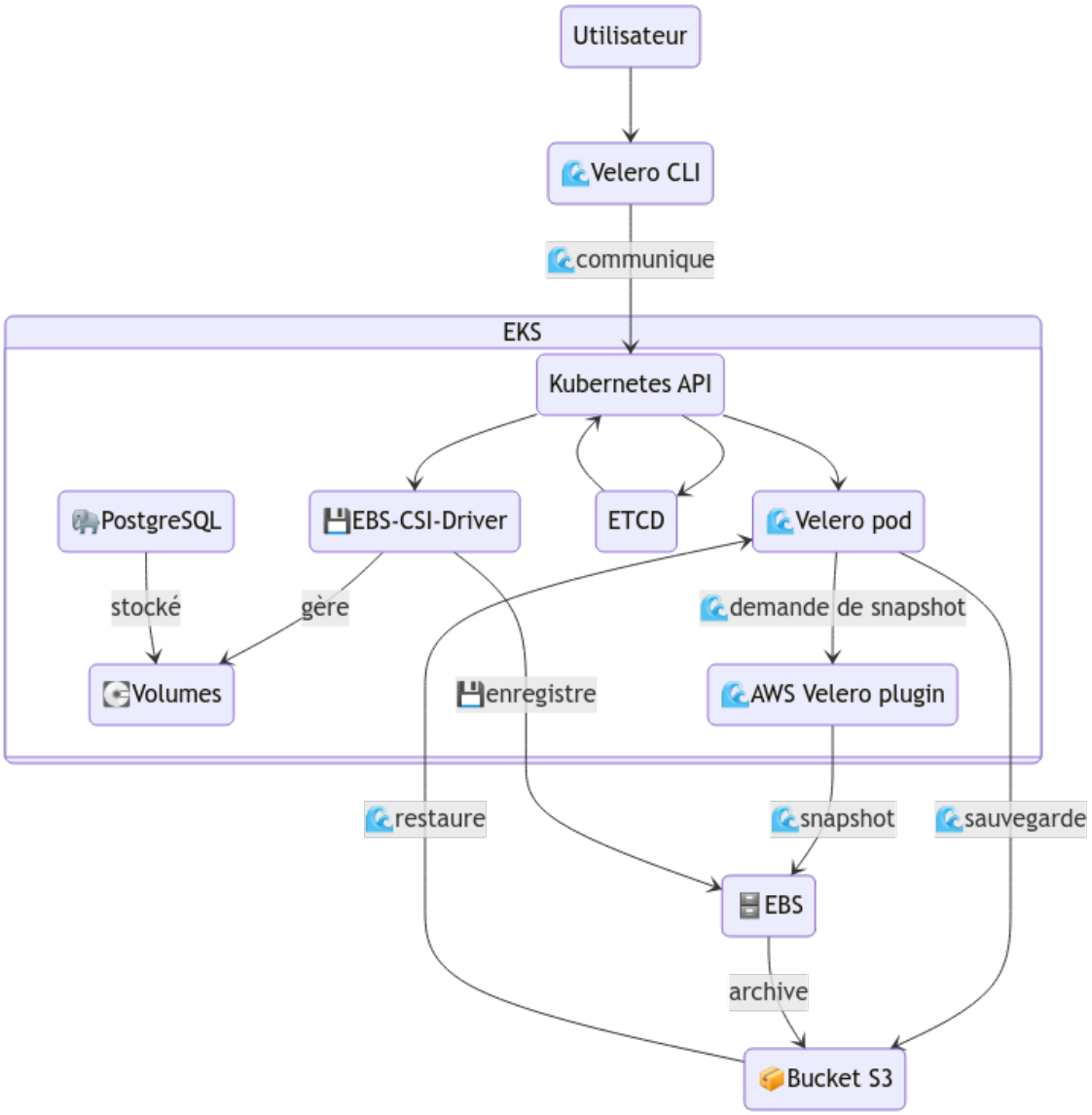

Figure 6: Schéma du système de sauvegarde

5.3.2 Supervision

Le choix de l'outil de supervision se porte sur **Datadog** pour sa **souplesse** et sa capacité à proposer des **services supplémentaires de manière progressive**.

Le fonctionnement semble simple : installation d'agents Datadog dans le système Kubernetes pour receuillir de nombreuses informations. Comme peut le montrer le schéma suivant.

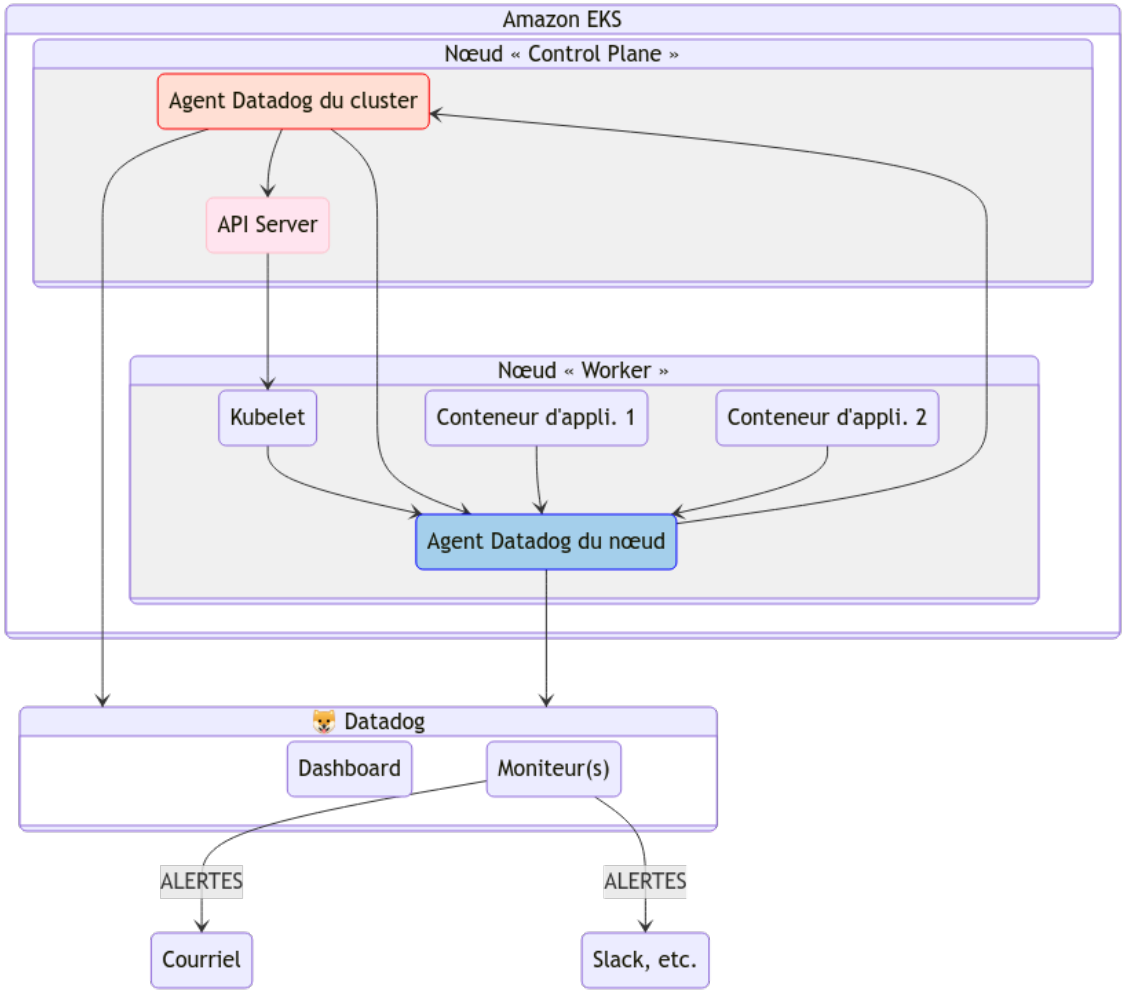

Figure 7: Schéma des agents Datadog dans notre système Kubernetes

Une fois installé, l'agent Datadog recueille et envoie les informations sur les serveurs officiels de l'entreprise Datadog qui en gère le stockage et l'accès.

5.3.2.1 Surveillance (monitoring)

L'utilisation de Datadog donne accès à de nombreux moniteurs pré-configurés.

On veillera à utiliser des **indicateurs simples** comme l'**utilisation CPU** et la **mémoire vive** des instances AWS EC2 (Elastic Compute Cloud) utilisées.

Il y a également :

- le **nombre de pods utilisés** par l'instance EC2,
- et la **disponibilité du service** de l'application *API Utilisateurs*.

Cependant ces moniteurs sont insuffisants. On pourrait ajouter :

- le temps de réponse de l'API,
- le nombre de requêtes par seconde sur l'API,
- même chose pour le nombre de requêtes sur la base de données,
- le nombre d'erreurs 500 retournées par le frontend et le backend,
- etc.

5.3.2.2 Alertes

On va utiliser **un canal Slack** (*#alertes*) **pour avertir les DevOps** d'une quelconque **alerte levée**. Datadog permet cela facilement comme énoncé dans l'introduction de ce chapitre sur la Supervision.

Suite aux alertes, les **DevOps peuvent opérer et notifier** de la prise en charge d'un problème **via le chatbot Gitlab intégré** à ce canal Slack. Nous utilisons donc le **Gitlab ChatOps**.

En suivant toutes ces indications, nous devrions avoir une base stable et saine pour constuire une supervision correcte de l'application *API Utilisateurs*.

5.3.3 Sécurité

Au niveau de la sécurité, **vaste sujet**, nous partirons de ces éléments :

- lancement de **tests** sur le **code applicatif**,
- lancement de **tests SAST** (Static application security testing), sur Terraform notamment,
- et utilisation d'un **coffre fort numérique** pour les mots de passes et clés d'accès en tous genres.

Le coffre-fort choisi est **Vaultwarden**.

Il y a beaucoup à faire pour **améliorer la sécurité**. Ceci est un **processus continu** à faire tout au long du **suivi et la maintenance du projet**.

5.4 Conclusion

Les spécifications techniques décrivent bien l'application, composée de 3 éléments (backend, frontend et base de données) qui s'intégreront bien dans l'environnement Kubernetes installé sur l'infrastructure posée dans le cloud AWS.

On retient ainsi un déploiement automatique dans le Cloud d'une infrastructure prête à accueillir à la fois l'application et d'autres services tierces comme la supervision avec Datadog et les sauvegardes à l'aide de Velero.

La sécurité aura cette spécificité de devoir être régulièrement vérifiée, maintenue et améliorée au fil du temps. C'est donc un processus continu dont la réussite dépend beaucoup de la discipline et la régularité.

6 Démarche suivie

La réalisation de ce projet va plus loin que la simple mise en place d'une application dans le Cloud. C'est une **nouvelle équipe**, dans un **nouvel environnement** avec une application inconnue et **des savoirs à acquérir**.

En pareille situation, c'est tout un **écosystème DevOps** qu'il faut mettre en place. Nous allons donc avant tout parler des **méthodologies utilisées** pour ce faire, puis des **outils utilisés** et enfin aborder le sujet de **collaborations** avec d'autres équipes.

6.1 Méthodologie

6.1.1 Agile

Le choix d'une **méthodologie agile** « basique » s'est offerte à nous, notamment par :

- des **comptes-rendus journaliers rapides de 10mn** en respectant les points suivants :
  - « *ce que j'ai fait hier* »,
  - « *ai-je rencontré un/des problème(s) qui vaut(valent) d'être cité(s)* »,
  - « *ce que je compte faire aujourd'hui* »,
- la **création de jalons** avec un ensemble de tâches sur une période d'une semaine,
- des échanges réguliers en **pair-programming** sur des sujets difficiles **via la plateforme Slack**,
- et l'utilisation d'un **tableau façon Kanban** (méthode de gestion de production en flux tendus).

6.1.2 Jalons

Les **jalons** (milestones en anglais) permettent de savoir si le projet dérive de l'objectif attendu. Voici les jalons qui ont été utilisés :

- Étape 1 : **préparation** - outils (configuration de gitlab, des projets, du wiki, etc.), **cahier des charges**,
- Étape 2 (avec 3) : **CI/CD** sur le projet principal,
- Étape 3 (avec 2) : **Infrastructure** (Terraform),
- Étape 4 (avec 5) : Données (**postgreSQL**),
- Étape 5 (avec 4) : **Supervision** (observabilité, alertes),
- Étape 6 : **extras** (s'il reste du temps).

6.1.3 Choix de standards et normes

Les jalons n'étaient pas suffisants, d'autres éléments ont dû être pris en compte ; des standards/normes ont été choisies pour **rassembler plutôt que diviser le groupe** : c'est

là que SemVer, Git OneFlow et Conventional commit entrent en jeu.

6.1.3.1 SemVer

SemVer ou **Semantic Versioning** est un standard visant à codifier la manière d'écrire le numéro de version d'une application et leur hiérarchisation (c'est à dire la façon dont on incrémente un numéro de version).

Respecter SemVer, c'est permettre d'utiliser des outils conçus pour cela mais aussi et surtout **faciliter la gestion des dépendances**.

Nous ne détaillerons pas l'ensemble du standard, mais voici **quelques exemples de versions** :

```
1.0.0
2.1.4
0.6.3-alpha
0.6.3-alpha.1
1.5.2-rc.2
```

Cela fonctionne donc sous la forme **Majeur**.*Mineur*.`Correctif`.

6.1.3.2 Git OneFlow

Git OneFlow est un workflow Git conçu par Adam Ruka en alternative à GitFlow.

Il se base principalement sur un objectif simple : avoir un arbre Git dont la **branche principale** est **la plus linéaire possible**. C'est à dire avec le moins d'embranchements possibles.

En somme on va favoriser le **rebase** sur les branches dédiées à une nouvelle fonctionnalité **avant de fusionner sur la branche principale**. Cela aura l'avantage :

- de réduire le travail de la personne qui fusionne la branche de fonctionnalités,
- et de favoriser la résolution des conflits de fusion aux développeurs responsables de la branche de fonctionnalités.

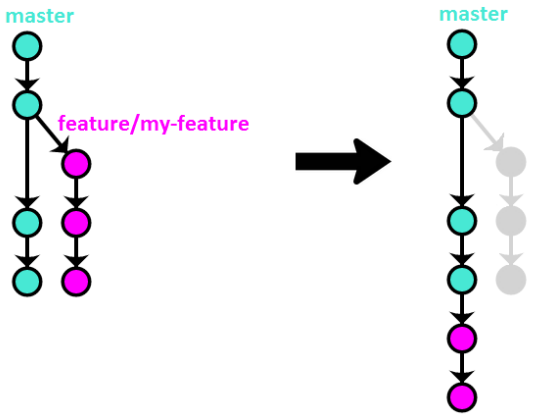

Figure 8: Avant/après la fusion d'une branche en utilisant Git OneFlow

6.1.3.3 Conventional commit

Dans le cadre de la **rédaction de message de commit** pour chaque changement effectué sur un dépôt, nous avons choisi de suivre la **spécification Commits Conventionnels**.

Cela se résume à peu près à cela :

```
<type>[étendue optionnelle]: <description>

[corps optionnel]

[pied optionnel]
```

Qui donne **par exemple** :

```
feat(backend): include postgreSQL BDD + backend

* change EC2 instances to t3.large
* change min_size nodes to 2
* use backend version 0.0.1
* use annotations to have api.r53.devu42.fr as domain for backend
* use annotations to use Let's Encrypt with cert-manager
* update EBS CSI driver to v1.37.0-eksbuild.1
* declare a StorageClass EBS in gp3 for postgreSQL
* use this EBS StorageClass in our backend chart for postgreSQL
* set database size to 1Gi

ref projet#18
```

On notera aussi l'**usage des références Gitlab** en fin de message de commit pour **faire un lien entre le commit et un ticket spécifique**, en l'occurence l'exemple montre une référence vers le ticket #18 du dépôt nommé *projet*.

6.2 Outils

Au delà d'une à plusieurs méthodologies/normes/standards, nous avons utilisé des outils afférents.

6.2.1 Nom de domaine

Afin d'avoir une **boîte courriel commune** (team@devu42.fr) et des **noms de domaine** qui pointent sur le résultat de nos travaux, nous avons opté pour la location d'un nom de domaine : **devu42.fr**.

Ce domaine fait référence à notre cursus de DevOps dans un centre de formation nommé **DevU**niversity. Le **nombre 42** est un chiffre connu lié aux ouvrages de Douglas Adams dans sa trilogie en 5 volumes de « H2G2, Guide du Voyageur Galactique ».

Nous avons ainsi pu avoir des **noms de domaines spécifiques** pour :

- le backend (API) dans l'environnement de pré-production, **api.staging**.devu42.fr,

- le backend (API) dans l'environnement de production, **api.r53**.devu42.fr,
- le frontend dans l'environnement de pré-production, **www.staging**.devu42.fr,
- et le frontend dans l'environnement de production, **www.r53**.devu42.fr.

C'était donc un bon départ.

6.2.2 Plateforme DevOps

Notre outil principal a été la **plateforme Gitlab**. C'est une plateforme complète de DevOps qui couvre bon nombre de besoins DevOps parmi :

- des dépôts de code source (**dépôts Git**),
- un système d'**intégration continue** (**GitlabCI**),
- une gestion des **rôles utilisateurs** pour une granularité de **gestion de permissions** tout en finesse,
- des dépôts/**registres** pour :
  - les **Chart Helm**,
  - les **images Docker**,
  - les **modules Terraform**,
  - les **états Terraform**,
  - les bibliothèques Python, PHP, JS,
  - etc.,
- un **système de gestion de tickets** pour le retour régulier (feedback) des utilisateurs,
- l'usage de **jalons**, **kanban** et **métriques des équipes** pour une gestion efficace d'**une à plusieurs équipes** dans un projet,
- la gestion des versions (**releases**),
- l'**intégration avec Kubernetes** pour un suivi depuis Gitlab de nos clusters Kubernetes,
- etc.

De nombreuses fonctionnalités supplémentaires existent dans Gitlab. Cela en fait donc un outil de choix pour notre projet.

6.2.2.1 Base de connaissances

Une des premières choses qui a été faite sur cette plateforme a été la **mise en place d'une base de connaissances** sous la forme d'un Wiki.

Pour plusieurs raisons :

- **partager nos découvertes** sur les outils (liens web, procédures d'installation, etc.),
- détailler nos diverses **installations**,
- **partager nos travaux**,
- rédaction de contenu concernant les **réunions**, des **outils**, etc.

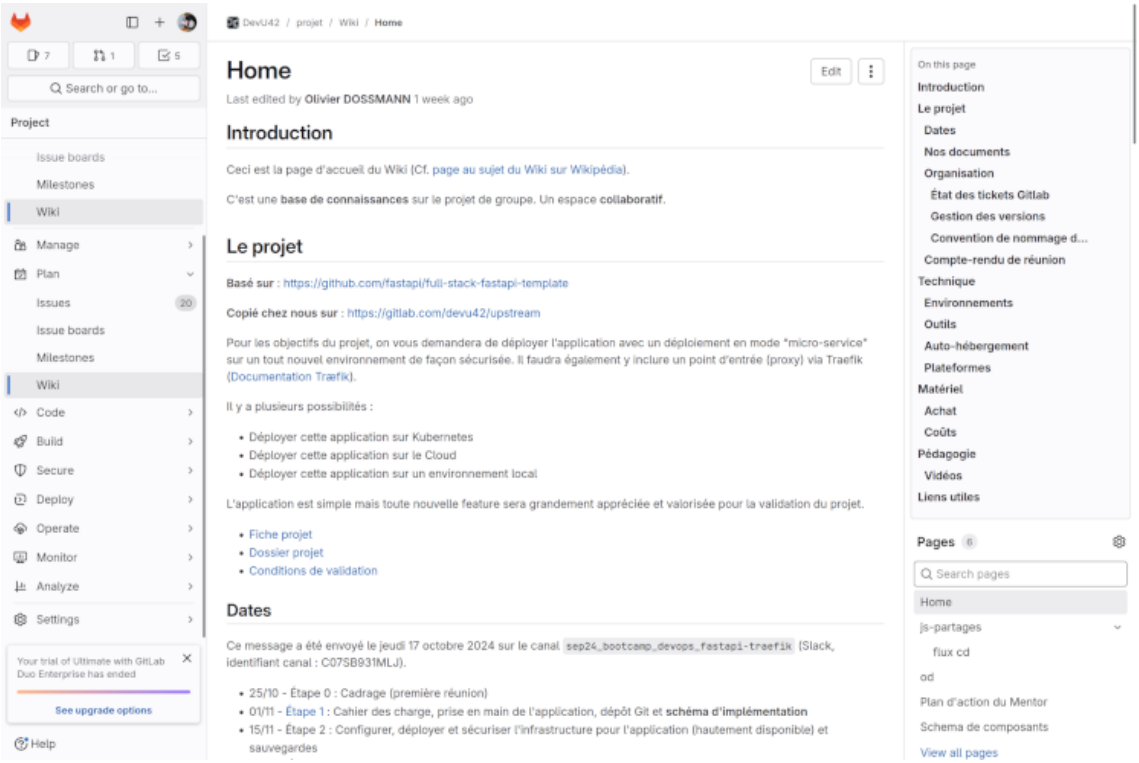

Figure 9: Impression écran de la page d'accueil du Wiki

6.2.2.2 Workflow des tickets

Autre exemple de configuration de la plateforme Gitlab afin de suivre notre organisation : la **création d'un workflow de l'état d'un ticket**, que vous trouverez ci-après.

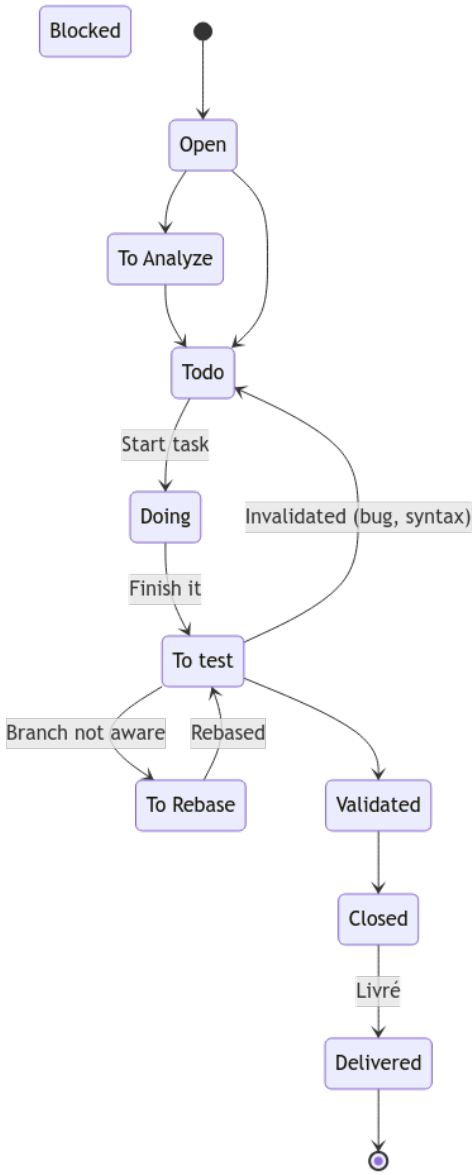

Figure 10: Schéma du workflow de l'état des tickets

Il permet à chaque membre de l'équipe de prendre connaissance des habitudes en matière de gestion de ticket.

Il est utile si jamais nous changions de plateforme DevOps : il suffirait d'appliquer à nouveau le même workflow.

6.3 Cas d'une collaboration inter-équipe

Dans le cadre de nos diverses **réflexions** nous avons eu un contact **avec l'équipe** qui se nomme **« Team Matrix »** dont le sujet était le déploiement de serveurs Matrix en haute disponibilité.

Nous avons échangé sur le **sujet de la promotion des versions entre environnements**. Maxime, de la Team Matrix, et moi avons principalement discuté autour de l'article suivant : How to model your GitOps environments and promote releases between them (https://codefresh.io/blog/how-to-model-your-gitops-environments-and-promote-releases-between-them/).

L'idée de l'article, en quelques mots, est de **favoriser l'utilisation d'un dossier par environnement** plutôt qu'utiliser une branche Git pour chaque environnement. Il conseille également de faire la promotion des versions entre ces différents environnements/dossiers. Avec un protocole à suivre par les « humains » pour limiter de trop nombreuses promotions logicielles en même temps.

Pour reprendre le schéma de l'article :

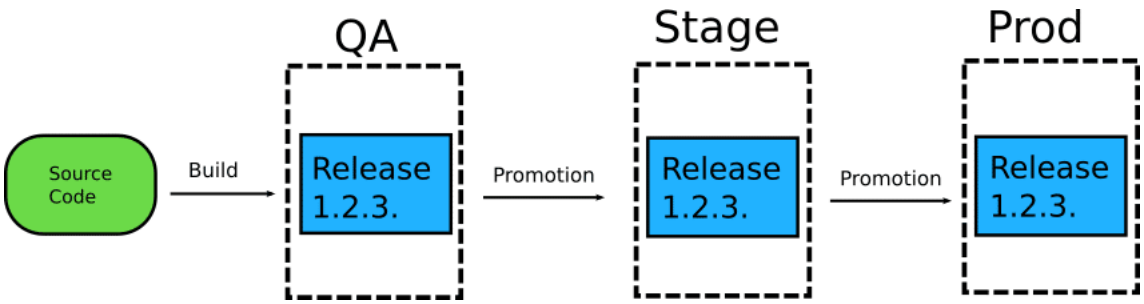

Figure 11: Schéma extrait de l'article sur la promotion des versions

Après moultes discussions, **nous sommes tombés d'accord** sur quelques points, notamment :

- **ne pas utiliser une branche par environnement**,
- essayer au maximum d'**avoir des dépôts indépendants** (favoriser la modularité),
- favoriser la **promotion logicielle par la CI** plutôt que manuellement,
- et **utiliser les particularités des outils que nous avons** (Terragrunt dans le cas de Maxime).

Notre réflexion a **evolué suite à ces échanges**. Et nous avons opté initialement pour utiliser **un dossier par environnement dans un dépôt unique** nommé **infra**. Puis nous avons adapté notre intégration continue pour copier les valeurs communes de l'environnement de pré-production (staging) vers l'environnement de production (prod).

Mon idéal serait :

- avoir des **dépôts indépendants pour chaque application** avec des **tests de sécurité**, des **tests du code**, vérification de la **couverture de code** et **tests sur la compilation** - éventuelle - de l'outil et/ou des **images Docker** puis **publication sur un registre**,

- avoir une gestion de chacun de ces dépôts d'**un système de versionnement** (les fameuses « releases »),
- avoir la **même chose pour chaque module Terraform** que nous avons fabriqué (par exemple notre module « networking »),
- puis utiliser, dans un dépôt **infra** par exemple, seulement les versions de nos modules (situés indépendamment sur des registres),
- de là on peut imaginer :
  - soit de la promotion des versions par une intégration continue entre dépôts (pré-production vers production par exemple) OU entre dossiers représentant un environnement,
  - soit de la promotion des versions par une validation manuelle entre la pré-production et la production,
- ajouter des **tests de sécurité des images** (comme la somme de contrôle) avant utilisation dans d'autres dépôts.

Maxime et moi avons également relevé que lorsqu'on parle de promotions dans l'infrastructure on peut imaginer 2 types de promotions :

- la promotion de versions concernant les logiciels/modules/charts utilisés,
- et la promotion de configurations pour différentes infrastructures.

C'est un autre sujet très intéressant que nous ne détaillerons pas ici.

6.4 Conclusion

La **démarche entreprise** a été structurée autour de **méthodologies** et d'**outils** et nous a permis de faire avancer le projet. L'apport des **discussions autour de la promotion logicielle** avec une autre équipe a eu un impact conséquent sur nos choix concernant l'automatisation et le déploiement continu sur l'environnement de pré-production. Nous en parlerons d'ailleurs dans le prochain chapitre un peu plus en détail.

A contrario, la **gestion d'équipe** lors de ce projet **a fait défaut** et nous aurions pu nous organiser bien mieux qu'il n'a été.

7 Mes réalisations

Ce projet m'a amené à participer sur de nombreux sujets. En voici deux qui, je l'espère, illustreront bien ce que j'ai rencontré.

Je parlerais de **livraison continue** (CI/cd) et d'**états Terraform**.

7.1 Réalisation 1 : Livraison continue

7.1.1 Contexte

Durant le court laps de temps accordé au projet, nous arrivions à la fin, nous étions en manque de ressources humaines :

- le travail de l'équipe n'était **pas toujours qualitatif**,
- parfois cela **ne répondait pas au besoin énoncé**,
- d'autres fois **une ressource prenait plus de temps que prévu** sur sa tâche.

Il en résulta **peu de moyens pour répondre à tous les besoins** du projet. Cela raccourcissait le temps des tâches restantes.

En pareille situation il a fallu **faire avec l'existant** afin de créér une chaîne de publication logicielle. J'ai pris en charge cette tâche.

7.1.2 Analyse

À ce moment du projet nous n'avions que peu d'éléments :

- un dépôt applicatif (nommé **upstream**),
- un dépôt contenant les charts Helm qui utilisent les images publiées du dépôt applicatif (nommé **charts**),
- et un dépôt contenant l'infrastructure de production (nommé **infra**).

Chaque dépôt peut utiliser l'intégration continue de Gitlab, appelée Gitlab CI et utilisant des pipelines.

On souhaite **créer une chaîne d'intégration continue** (CI, Continuous Integration) pour passer d'un dépôt à l'autre suivant **une chaîne**, comme le montre le schéma suivant :

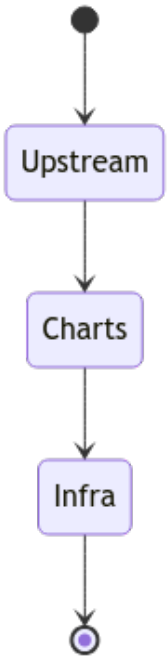

Figure 12: Schéma de la chaîne des dépôts

Cela fait penser au chapitre précédent où nous parlions du sujet de la promotion logicielle. Suivant **quelle(s) règle(s)** pouvons-nous **passer d'un dépôt à l'autre**, d'une version à l'autre ou (in)valider le fonctionnement ?

Après plusieurs heures de réflexions, de dessins en tous genres et du recul, j'ai constaté que :

- quoiqu'il arrive la **pipeline va lancer des tests** sur le code,
- et dans la **situation où l'on pose une étiquette** (appelé **tag** sur un dépôt Git), c'est qu'on souhaite valider - indirectement - le travail effectué.

Ainsi dans la situation où une étiquette apparaît, la pipeline diffère légèrement. Il y a donc globalement **2 règles à suivre**, représentées par le schéma suivant :

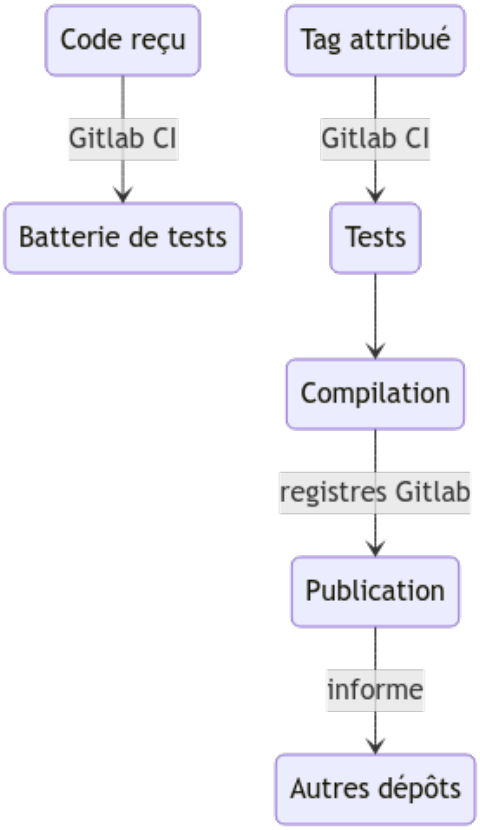

Figure 13: Schéma des 2 règles principales pour la CI d'un dépôt

Comme le résultat positif des tests engendre la compilation puis la publication d'une image sur un registre, une idée apparaît donc : **informer les autres dépôts que l'image est disponible** !

7.1.3 Solution

Suivant l'idée précédente qui consiste à informer les autres dépôts de la publication récente d'une version, il a fallu trouver une solution.

Nous sommes sur Gitlab, avons des dépôts Git, et des pipelines qui se lancent quand un commit est effectué. Dans la situation où une étiquette est posée, pourquoi ne pas **utiliser Git lui-même à l'aide d'un commit sur le dépôt suivant** ? Ainsi, sans outils supplémentaires, nous pouvons lier les dépôts entre eux.

En suivant cette logique, nous obtenons le schéma représentant la pipeline de chaque dépôt :

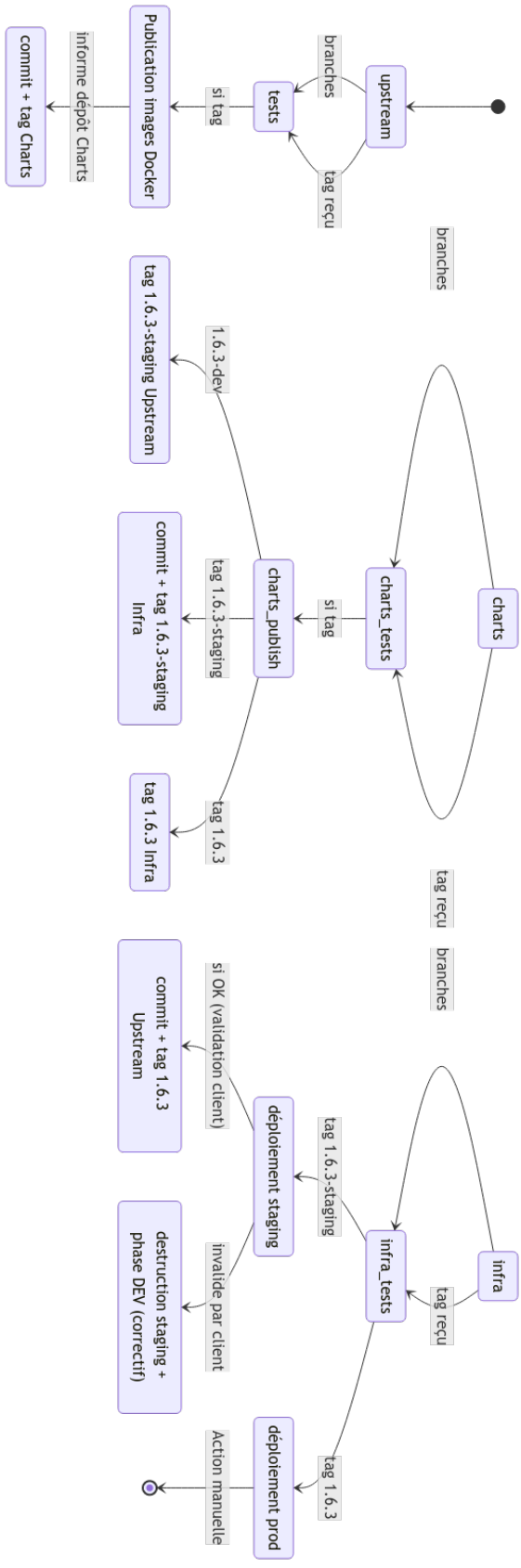

Figure 14: Schéma des 3 pipelines

Ce qui donne le savant mélange suivant :

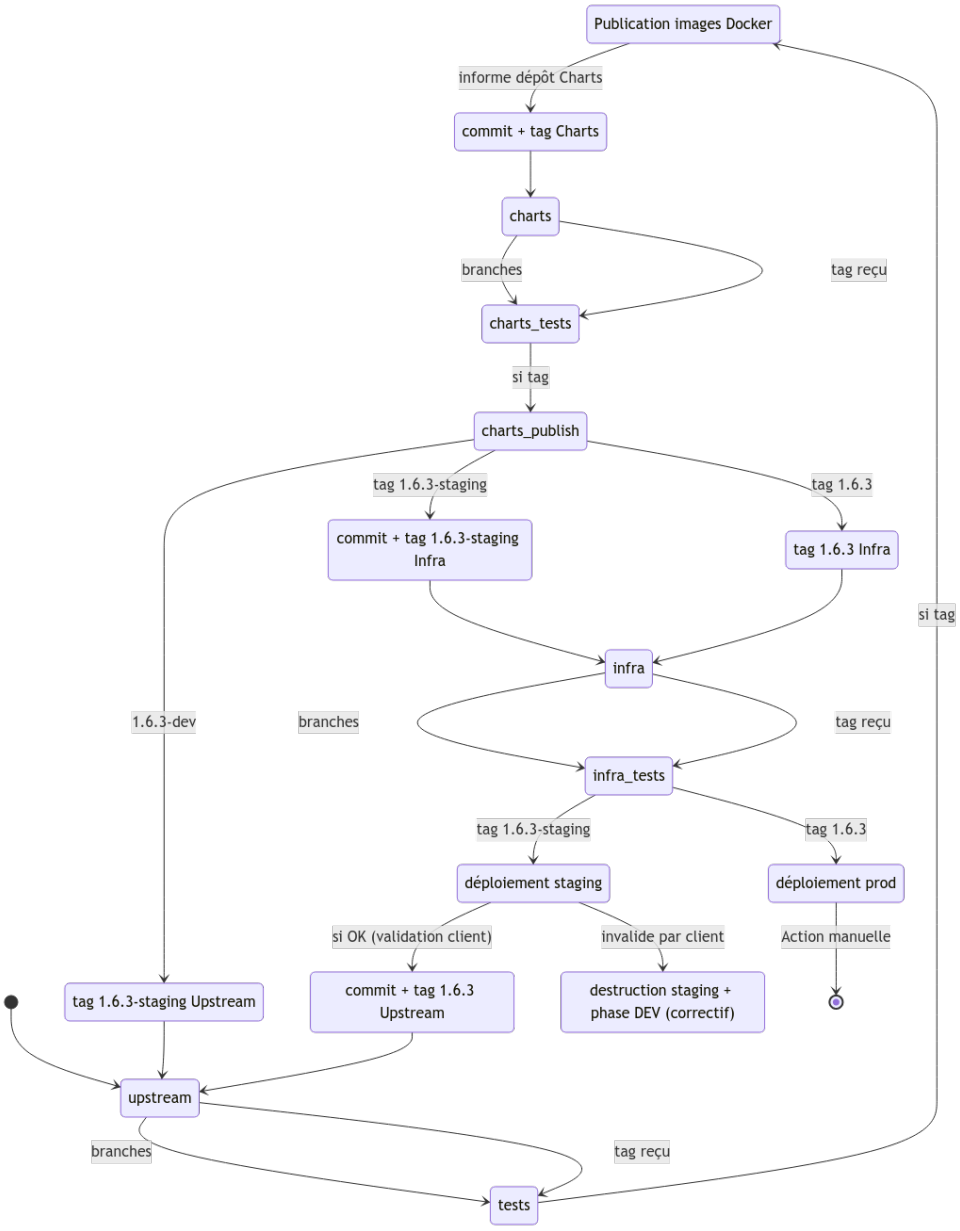

Figure 15: Schéma de toutes les pipelines imbriquées

Ainsi nous avons, par exemple, le code suivant permettant d'informer le dépôt **charts** qu'une nouvelle étiquette a été posée sur le dépôt **upstream** une fois le backend et le frontend publiés sur le registre Docker de Gitlab :

```
1  inform-charts:
2    stage: inform-next
3    variables:
4      BACKEND_CHARTFILE_PATH: charts/backend/Chart.yaml
5      FRONTEND_CHARTFILE_PATH: charts/frontend/Chart.yaml
6    script:
7      - |
8        if ! [ -z "$COMMIT_TAG" ]; then
9          # Configure git user
10         git config --global user.email "git@dossmann.net"
11         git config --global user.name "CI Pipeline"
12         # Get CHARTS repository
13         git clone https://blankoworld:${ACCESS_TOKEN}@${CHARTS_REPOSITORY}
14         cd charts
15         # Update files
16         sed -i "s/^version: \".*\"/version: \"${COMMIT_TAG}\"/" \
17           "${BACKEND_CHARTFILE_PATH}"
18         sed -i "s/^appVersion: \".*\"/appVersion:\"${COMMIT_TAG}-back\"/" \
19           "${BACKEND_CHARTFILE_PATH}"
20         sed -i "s/^version: \".*\"/version: \"${COMMIT_TAG}\"/"
21           "${FRONTEND_CHARTFILE_PATH}"
22         sed -i "s/^appVersion: \".*\"/appVersion:\"${COMMIT_TAG}-front\"/" \
23           "${FRONTEND_CHARTFILE_PATH}"
24         # Add them to git staged area
25         git add "${BACKEND_CHARTFILE_PATH}" "${FRONTEND_CHARTFILE_PATH}"
26         # Commit and push result (to launch CHARTS CI for new code)
27         git commit -m "chore(release): Update back/front image version"
28         git tag $COMMIT_TAG
29         git push origin main --tags
30       fi
31   needs: [push-backend,push-frontend]
```

Le code a dû être modifié (raccourcissement des lignes) pour les fins de rédaction du présent document.

Les **ligne 27 à 29** posent un tag sur le dépôt **charts** en mettant simplement à jour les versions utilisées du backend et du frontend dans les charts Helm sur le registre Gitlab.

7.1.4 Résultats

Après plusieurs points de détails corrigés, nous avons pu **automatiser correctement la chaîne de production logicielle** de la **phase de développement** jusqu'à **la production** en passant **par une validation manuelle** après déploiement en environnement de pré-production **via l'intégration continue de Gitlab** (Gitlab CI).

Ceci nous a permis de facilement **travailler sur 2 environnements** distincts :

- l'environnement de **pré-production**,
- et l'environnement de **production**.

7.1.5 Limites

Cette technique, bien qu'efficace, a ses limites :

- pour fonctionner sans développer plus, il a fallu **utiliser le même numéro de version sur tous les dépôts** : on ne fait que passer le tag courant au dépôt suivant. **Que faire si les numéros de version dérivent ?**,
- plus on ajoute de dépôts dans la chaîne, plus il y aura de code à faire et plus il y aura de **complexité à chaîner les dépôts**.

Trouver une solution temporaire et efficace est - potentiellement - facile. Mais avoir **une solution pérenne demande de l'expérience et plus de temps**.

7.1.6 Amélioration(s) possibles(s)

Cette expérience a été enrichissante. Bien que ce système ait répondu à nos besoins du moment, je pense qu'il serait envisageable de procéder autrement.

Dans un premier temps, **utiliser des dépôts indépendants** avec leur propre pipeline qui **teste** puis **publie** sur des **dépôts utilisables** par d'autres projets. Puis **chaque dépôt pourrait avoir un script de montée de version** pour définir les contraintes spécifiques au projet, au dernier numéro de version, etc.

Ainsi un dépôt d'infrastructure, par exemple, pourrait utiliser des versions spécifiques de chacun de ses dépôts indépendants. Et ce serait à lui d'aller regarder quelle est la dernière version d'un module ou d'une application. Et d'agir en conséquence : lancer une batterie de tests puis intégrer la nouvelle version disponible.

Ceci éviterait d'ajouter une quantité astronomique de code Gitlab dans le dépôt initial (celui de l'application par exemple). Et cela **laisse la responsabilité aux personnes suivantes** (qui utilisent le dépôt initial) plutôt qu'aux personnes qui s'occupent du code dans le dépôt initial.

7.2 Réalisation 2 : États Terraform

7.2.1 Contexte

Pendant la création du dépôt **infra** contenant - entre autre - l'infrastructure de notre environnement de production, **nous avons subis quelques pertes des états Terraform** posés sur le registre Gitlab.

Non seulement **les états Terraform étaient continuellement cassés** par les collaborateurs, mais l'ignorance de ces derniers sur le fonctionnement de Terraform nous a mis **une belle épine dans le pied**.

Il a fallu agir. Sur mon temps libre - au grand dam de ma famille - j'ai décidé d'y regarder de plus près.

7.2.2 Analyse

Comme nous avons plusieurs environnements, l'état Terraform va décrire chaque fois un environnement. Nous aurons besoin au minimum des états Terraform suivants :

- un état **gitlab-ci** pour les pipelines de test,
- un état **staging** pour l'environnement de pré-production,
- un état **prod** pour l'environnement de production,
- et éventuellement **un état Terraform par développeur**.

C'est ce dernier point que je cherchais à résoudre : **faciliter le travail du développeur** sur Terraform **pour éviter toute bévue**.

Nous constatons également :

- le **manque de documentation** par le développeur pour savoir comment procéder pour travailler sur Terraform,
- que **trop d'erreurs manuelles** sont effectués lors de l'utilisation des commandes Terraform.

C'est ce qui **nécessite une simplification** et/ou un changement.

7.2.3 Solution

Dans le Logiciel Libre, dans la plupart des dépôts, nous retrouvons un fichier **Makefile** qui décrit les différentes étapes de compilation d'une application, de son installation ou de la création d'une image.

Pourquoi ne pas partir sur cette solution habituelle et l'adapter à notre cas ?

Ainsi l'idée serait de **reprendre les commandes habituelles de Terraform**, à savoir :

- **init**,
- **plan**,
- **apply**,
- **destroy**,
- **fmt**,
- et **graph**.

J'ai imaginé mettre à disposition les mots-clés suivants avec le fichier **Makefile** :

- **make init** - pour **initialiser l'état Terraform**,
- **make plan** - pour **vérifier les changements attendus** entre l'état Terraform local et l'environnement distant,
- **make apply** - pour **appliquer les changements**,
- **make destroy** - pour **supprimer l'ensemble des ressources distantes**,
- **make format** - pour **formater le contenu des fichiers *.tf** trouvés,
- **make graph** - pour **générer une image vectorielle des dépendances entre ressources** du projet,

- et **make clean** - pour **nettoyer les fichiers Terraform** non nécessaires.

Ces commandes ne résolvent pas le problème en tant que tel. C'est ce qui est derrière qui va résoudre le souci : **des scripts qui vérifient que tout soit OK** avant d'appliquer les commandes.

Ainsi l'**utilisation d'un fichier .env**, non disponible dans le dépôt de code, permet de charger les informations utiles au choix d'un état Terraform et de l'environnement distant à atteindre pour travailler.

La solution réside dans le fait que des scripts soient lancés sous chacune des commandes make et qu'ils vérifient l'existence et le contenu du fichier **.env**.

S'ajoute à cela une **documentation dans le fichier README.md**.

Exemple de script avec **make init** pour bien comprendre de quoi il est question :

```
#!/usr/bin/env bash
#
# init.sh
#
# Initialize for a Developer

ENV_FILE="${PWD}/.env"
BACKEND_HTTP_FILE="${PWD}/backend.hcl"

# Need variables in this file
source "${ENV_FILE}" \
  || echo "Fichier ${ENV_FILE} manquant." \
  || exit 1

echo "[INFO] 'USERNAME': ${TF_HTTP_USERNAME}"

if ! [[ -f "${BACKEND_HTTP_FILE}" ]]; then
  echo "[ERR] ${BACKEND_HTTP_FILE} manquant!"
  exit 1
fi

terraform init \
  -reconfigure \
  -backend-config=${BACKEND_HTTP_FILE} $@
```

Et le contenu du fichier **env.example**, partagé aux développeurs comme d'un template pour fabriquer le fichier **.env** :

```
# TOKEN DOIT avoir permissions read/write
export TF_HTTP_USERNAME="PseudoGitlab"
export TF_STATE_NAME="dev"
# Cf.https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html
export TF_HTTP_PASSWORD="monPersonnalAccessTokenGitlab"
# Backend HTTP
export TF_HTTP_ADDRESS="<myurl>/state/${TF_STATE_NAME}"
export TF_HTTP_LOCK_ADDRESS="<myurl>/state/${TF_STATE_NAME}/lock"
export TF_HTTP_UNLOCK_ADDRESS="<myurl>/state/${TF_STATE_NAME}/lock"
# Accès AWS
export AWS_ACCESS_KEY_ID="AWS-access-key-id"
export AWS_SECRET_ACCESS_KEY="secret-AWS-access-key"
```

Pour faciliter la création de nouveaux dépôts utilisant Terraform, j'ai choisi d'inclure ce travail dans un dépôt dit **template** sur Gitlab.

7.2.4 Résultats

À l'usage le travail sur Terraform s'est amélioré. En procédant ainsi, toute erreur se produisant sur Terraform n'affectait pas les autres collaborateurs. **Chacun avait son état Terraform**.

La **documentation a permis** aux développeurs **de comprendre facilement** quoi changer pour travailler. Et ils se sont sentis moins perdus.

7.2.5 Limites

Toutefois, procéder ainsi a quelques limites :

- **toute modification** sur les éléments du Makefile, les scripts, la documentation, etc. **doit être faite sur CHAQUE dépôt**. Ainsi plus on a de dépôt, plus on consomme de temps à mettre à jour (avec des oublis possibles),
- **si le template évolue**, les dépôts ayant utilisé le template **n'ont plus les mises à jour**. On pourrait imaginer faire un **git rebase, mais cela casserait l'historique des dépôts** ou bien demanderait d'effectuer du travail sur une branche Git à part.

C'est très limitant.

7.2.6 Amélioration(s) possibles(s)

Cette solution a fait ses preuves. Elle **a été utile au moment où nous en avions grandement besoin**. Elle permet de prendre de l'expérience à ce sujet.

Nous pourrions imaginer améliorer cela avec :

- **une forme de dépendance du fichier Makefile** et des scripts avec le dépôt **template** utilisé (par exemple via une commande de mise à jour fournie),

- avoir une possibilité de **choisir entre la commande Terraform et OpenTofu** par l'utilisation d'une variable d'environnement,
- et **permettre l'ajout d'arguments** aux commandes **make apply**, voire ajouter une commande **make apply-autoapprove**.

Il est aussi possible qu'il existe une commande de type **wrapper** (qui utilise Terraform et ses options) pour en faciliter l'usage, tel que **kubectx** pour Kubernetes.

7.3 Conclusion

Deux expériences variées et différentes qui s'imbriquent pourtant dans le processus de production logicielle et de livraison continue. Il y a beaucoup de choses à dire sur chacun des sujets, tellement ils sont passionnants. Ils amènent également à prendre des initiatives et tester localement des outils.

Ces expériences ont aussi permi d'acquérir de nouvelles compétences primordiales dans le domaine du DevOps. La prise de recul sur ces éléments met en évidence des failles dans le schéma de pensée. Il y a donc des points à améliorer et une évolution possible dans les tâches présentées.

Je pense aussi que de nouvelles expériences seraient enrichissantes pour comparaison avec celles présentées ici afin d'estimer les pour et les contre de différentes solutions. Cela permettrait de s'adapter aux situations avec des outils adéquats.

8 Situation de travail

L'exercice demandé dans ce chapitre est particulier pour moi. Il est demandé, je cite, « **une description d'une situation de travail ayant nécessité une recherche effectuée par le candidat durant le projet** ».

Pour saisir **la méthodologie choisie**, je pense qu'il est nécessaire de décrire une situation complète. Nous allons poser le contexte, faire l'état des lieux puis suivre la démarche pour atteindre l'objectif.

8.1 Contexte et état des lieux

Du projet émane un besoin d'**avoir du temps de calcul pour les pipelines**.

Il est nécessaire de réfléchir et trouver une solution viable pour l'environnement de test. L'**objectif** est donc de **préparer un environnement de test** pour l'ensemble des éléments du projet. **Nous utilisons Gitlab**, ce sont donc des Gitlab Runner que nous visons.

Où les installer ? Comment ? **De quelles ressources disposons-nous ?**

Nous utilisons Gitlab qui propose, dans sa version gratuite, une **limite de temps de calcul**. Une fois le seuil atteint : il faut trouver une solution alternative aux **machines de Gitlab, payantes**.

Autre **environnement disponible : AWS**. Mais avec un **budget limité** qui mangerait dans le budget alloué à nos 2 environnements : la production et la pré-production.

Il n'y a pas le choix : **il faut envisager autre chose**. Avec les ressources à disposition du groupe, autrement dit : **avec le matériel de chacun**.

L'idée ? **Utiliser une machine disponible**, y **installer un serveur** et un **environnement Kubernetes** pour **déployer des Gitlab Runner**.

Après en avoir discuté en groupe, je me suis attelé à la tâche. Régulièrement.

8.2 Démarche

8.2.1 Retranscription préalable

Étant donné que j'allais passer du temps à faire des recherches, il fallait **un lieu où synthétiser les résultats**. J'ai opté pour l'**utilisation d'un site** *.gitlab.io* **généré automatiquement par les pipelines de Gitlab** en utilisant comme **moteur statique** un outil nommé **Hugo**. Le résultat est **disponible sur odtre.gitlab.io** afin qu'il puisse **être consulté par mes collaborateurs**.

Ce site va contenir principalement **2 sections** :

- **posts** : les **réflexions sur différents sujets/thèmes** de l'environnement de test,
- **doc** : de la **documentation sur l'installation** effectuée **des outils en place**.

Maintenant qu'un **système de retranscription** est en place, **la recherche commence !**

8.2.2 Méthode de renseignement

Je procède souvent de la même manière :

- contact avec des **personnes de mon entourage déjà dans le métier** et pouvant m'indiquer des pistes à suivre,
- recherche de **livres existants** sur le sujet,
- passage à la **médiathèque**,
- recherches sur **Internet d'articles** sur le sujet (support textuel uniquement).

Et c'est ce que j'ai fait :

- **demande à un ami DevSecOps** si je pouvais l'interpeller de temps en temps sur l'un ou l'autre sujet,
- achat du **livre « Kubernetes - Guide pratique »** aux éditions O'REILLY,
- recherche, infructueuse, **à la médiathèque** : le rayon informatique est maigre,
- recherche sur le domaine du DevOps **sur Internet** avec une **grande lecture initiale** de **Stéphane ROBERT** (Devenez expert DevOps et maîtrisez ses outils) (j'ai repris le titre du site tel quel).

Figure 16: Première de couverture du livre O'Reilly, Kubernetes - Guide pratique

8.2.3 Point de départ

Grâce à la lecture de quantité d'articles sur le blog de Stéphane ROBERT, notamment la série d'article sur « Mon nouveau Homelab DevOps » (Cf. https://blog.stephane-robert.info/docs/homelab/introduction/) et la **Roadmap de son parcours de formation DevSecOps** (Cf. https://blog.stephane-robert.info/docs/#la-roadmap), j'ai pu établir des **objectifs à court terme** et des étapes de sujets à aborder comme :

- le **sujet de la virtualisation**,
- **le GitOps avec** un outil tel que **FluxCD**.

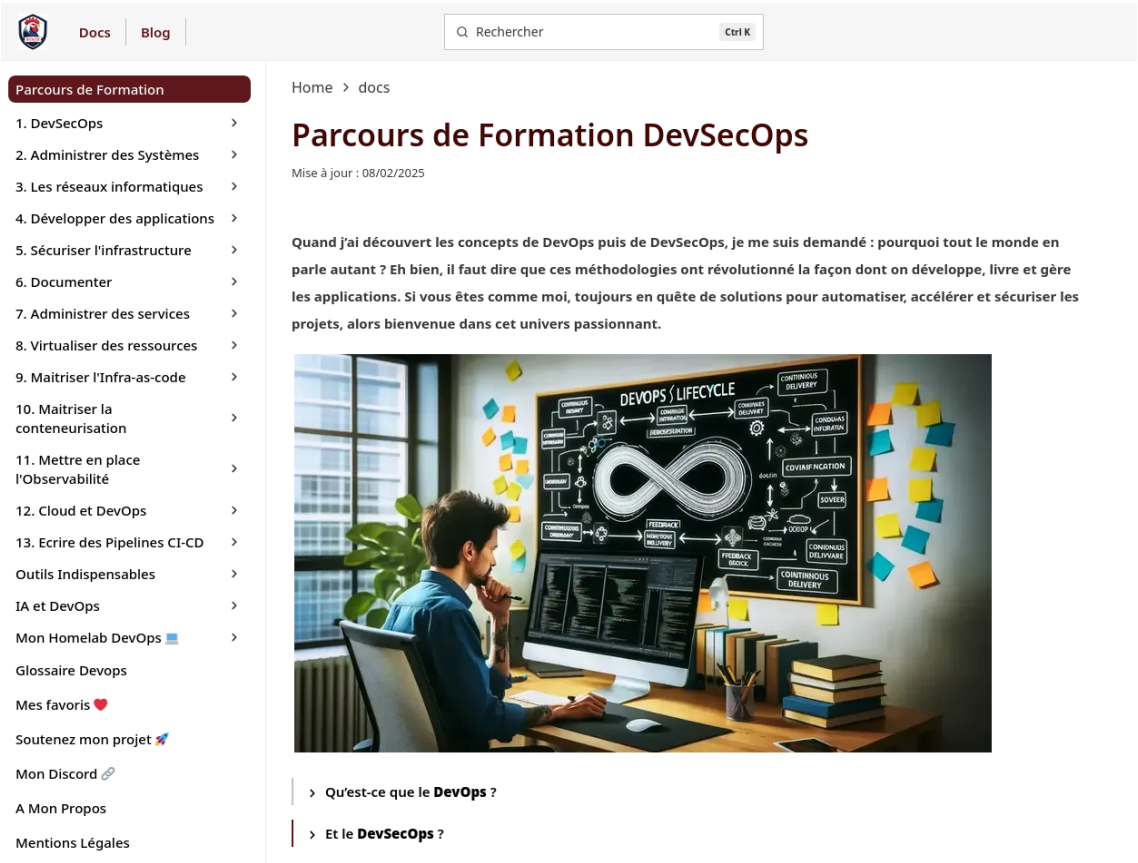

Figure 17: Copie d'écran de la page d'accueil du blog de Stéphane ROBERT

8.2.4 Recherches

8.2.4.1 Sujet de la virtualisation

Je pars souvent de **sites de référence ou de sites que j'ai déjà utilisé** plusieurs fois dans le passé. Le sujet de la virtualisation a donc donné lieu à **quelques recherches sur des sites déjà connus** :

- Le **wiki d'ArchLinux** et sa catégorie « **Virtualization** » : https://wiki.archlinux.org/title/Category:Virtualization,
- Le blog précédemment cité de **Stéphane ROBERT**.

Je me suis concentré sur l'outil conseillé par **mon contact DevSecOps** qui utilise QEMU/KVM/Libvirt pour l'émulation d'une machine complète. J'ai lu et appliqué :

- La **page Wiki d'ArchLinux sur QEMU**, Cf. https://wiki.archlinux.org/title/QEMU,
- La **page Wiki d'ArchLinux sur libvirt**, Cf. https://wiki.archlinux.org/title/Libvirt,
- et un **message de forum** (forums.debian.net) concernant **les bonnes pratiques sur l'usage de virt-manager**, Cf. https://forums.debian.net/viewtopic.php?t=158967.

J'ai ainsi rédigé 2 articles sur le sujet de la virtualisation, disponibles sur odtre.gitlab.io (outil de retranscription) :

- **réflexion sur le sujet de la virtualisation**, Cf. https://odtre.gitlab.io/post/002-virtualisation/,
- et **exemple de virtualisation avec QEMU/KVM/Libvirt sous Linux**, Cf. https://odtre.gitlab.io/doc/virtualisation/.

La virtualisation permet, dans notre situation, de **tester un ou plusieurs systèmes d'exploitation** avant de les adopter et les déployer sur notre infrastructure.

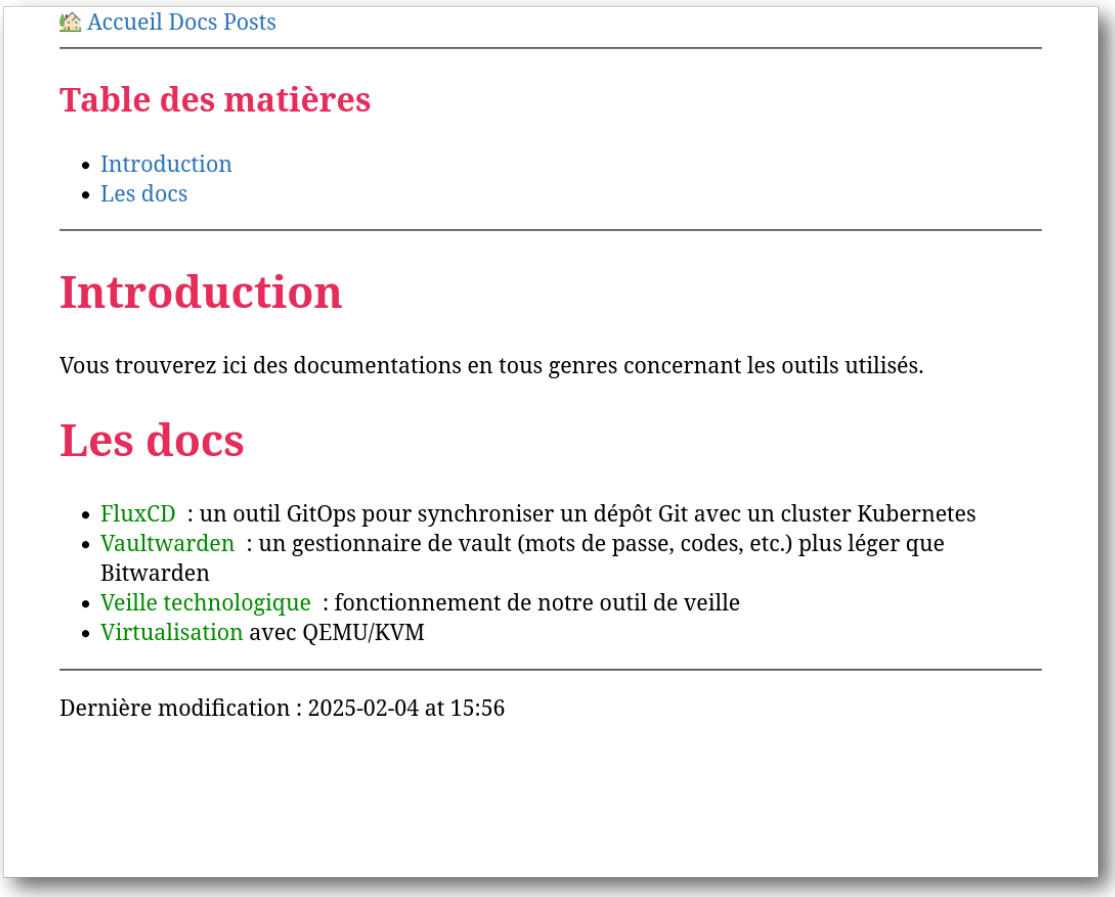

Figure 18: Copie d'écran de la section « Documentation » du site statique odtre.gitlab.io généré sur Gitlab

8.2.4.2 Sujet sur FluxCD

Concernant des outils spécifiques tels que FluxCD, **je favorise avant tout la documentation du site officiel**. Cf. https://fluxcd.io/flux/.

C'est ainsi que plusieurs éléments de **documentation** m'**ont été nécessaires** pour déployer et utiliser **FluxCD** :

- la **page d'installation officielle**, Cf. https://fluxcd.io/flux/installation/,
- les différents « **bootstrap** » possibles en fonction d'un **fournisseur de dépôt Git** choisi, Cf. https://fluxcd.io/flux/installation/bootstrap/,
- et des **explications sur le contrôleur de sources**,
  Cf. https://fluxcd.io/flux/installation/bootstrap/ pour se mettre à jour « tout seul ».

Comme auparavant, ceci a donné lieu à **la rédaction d'un article sur notre outil de retranscription**, Cf. https://odtre.gitlab.io/doc/fluxcd/.

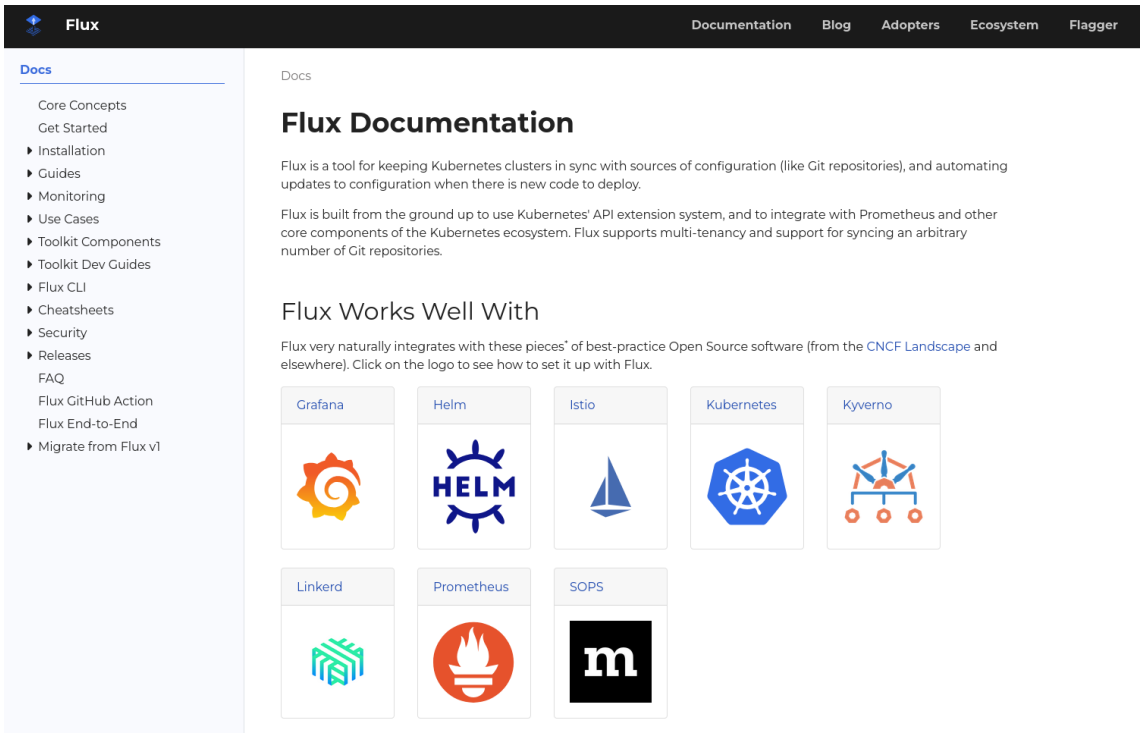

Figure 19: Copie d'écran de la documentation de FluxCD sur le site officiel

8.3 Conclusion

Que les recherches soient fructueuses ou non, le projet a avancé. Les recherches ne donnent pas tous les savoirs, **parfois il faut tester aussi**. Bien que pour la plupart des outils, une fois choisis, la documentation officielle de ces outils semble suffire pour leur installation et leur utilisation. Les **bonnes pratiques** restent **dans les articles des blogueurs** et autres joyeux rédacteurs en tous genres.

Le blog de Stéphane ROBERT est un bon point de départ qui ne dispense absolument pas

de recherches supplémentaires. Des outils tels que QEMU/KVM/libvirt pour la virtualisation sont de bonnes idées pour tester les outils de DevOps dans un environnement isolé. Tout comme FluxCD qui permet d'automatiser, à l'intérieur d'un environnement Kubernetes, la recherche de nouvelles versions de ses applications.

La méthodologie ayant été fructueuse, elle sera dans mes priorités pour les années à venir. Il faudra cependant être attentif au temps passé à la publication de ses recherches qui, bien qu'il soit un investissement sur long terme, consomme un temps certain à la rédaction et la mise en page.

9 Conclusion

Ma **passion anticipée pour l'automatisation** et l'**amélioration des processus** de **création logiciel** m'ont amené à ce moment où j'écris ces lignes pour **embrasser le métier d'Administrateur Système DevOps**. La formation, puis **le projet de fin de formation** ont été **bénéfiques à la prise d'expérience**, mais pas que.

En effet, **il est question de nouvelles compétences**, que le projet, à travers la rédaction d'un cahier des charges, de spécifications fonctionnelles, d'une approche (démarche) suivie, de plusieurs réalisations et de situations diverses, a permis d'**acquérir tout au long de ces derniers mois**.

La découverte de nombreux outils comme Kubernetes, Terraform, FluxCD, Datadog, externaldns, cert-manager, QEMU/KVM, libvirt, etc. a été riche et variée. Il est possible **d'approfondir plus encore** sur ces derniers et de les essayer dans **d'autres environnements de cloud** comme *Microsoft Azure* ou *Google Cloud Platform*.

Ainsi **je ressors riche de nouvelles expériences**, de **nouveaux outils**, de **nouvelles méthodes** et de **nouveaux collaborateurs** dans un monde sans cesse grandissant qu'est **le monde du DevOps**. C'est aussi et surtout une culture qui amène à de **bonnes pratiques**, une **amélioration constante** et la découverte infinie de projets en tous genres.

Ce projet, bien qu'ambitieux au départ, semble désormais une approche basique à reprendre lors de futurs projets. Il faudra alors améliorer la gestion d'équipe, une planification plus réactive et la prise en compte de plus de facteurs comme l'apprentissage de nouvelles technologies, les retards et le temps accordé à chaque étape. D'autres outils, plus adaptés, pourront également mieux se greffer à de nouveaux projets pour en faciliter la finalisation.