

Sentimail

Spécifications techniques

CPE Lyon

Nicola PIEMONTESE, Valentin TOURNIER, Thomas VIOLENT,
Thomas GENIN
25/01/2024

Table des matières

Table des matières	1
Introduction	2
Choix d'architecture.....	2
Architecture en micro-services.....	2
Back-end	2
Micro-services d'analyses	3
Un broker de message pour la communication entre les micro-services	4
Une base de données pour les données des utilisateurs et des données de l'application	4
Choix technologiques.....	6
Langages et Frameworks	6
Infrastructure	8
Choix de déploiement.....	9
Intégration et déploiement continu avec GitHub Actions	10

Introduction

Ce document détaille les spécifications techniques du projet, y compris les choix d'architecture, les choix technologiques, les choix de déploiement et les choix de développement.

Choix d'architecture

Architecture en micro-services

Dans le cadre du projet SentiMail, nous avons décidé de mettre en place une architecture micro-services qui est la plus adaptée aux contraintes du projet.

En effet, l'architecture en micro-services est une architecture logicielle qui consiste à découper une application en un ensemble de services indépendants, qui communiquent entre eux au moyen de messages. Chaque service est déployé indépendamment des autres et communique avec les autres services au moyen de messages. Cette architecture permet de découpler les services et de les déployer indépendamment les uns des autres. Elle permet également de développer les services dans des langages différents.

Des services peuvent être ajoutés ou supprimés sans impacter les autres services. Cette architecture permet également de développer les services dans des langages différents.

Ainsi nous avons mis en place plusieurs micro-services à savoir un backend, 3 services d'analyses, une base de données, un broker de message ainsi qu'un système de stockage objets.

Back-end

Notre application est composée d'un backend permettant de gérer les utilisateurs, les données et l'api.

Tout d'abord le backend permet de générer les pages web grâce au moteur de template de Django.

De plus, lors de l'upload d'un email celui-ci est traité par le backend : l'email est stocké sur Minio, enregistré dans la base de données et un message est transmis sur le broker de message afin que les micro-services puissent réaliser l'analyse. En complément, une API « privé » est en place afin de récupérer les résultats des analyses envoyés par les micro-services d'analyses et les stocker dans la base de données.

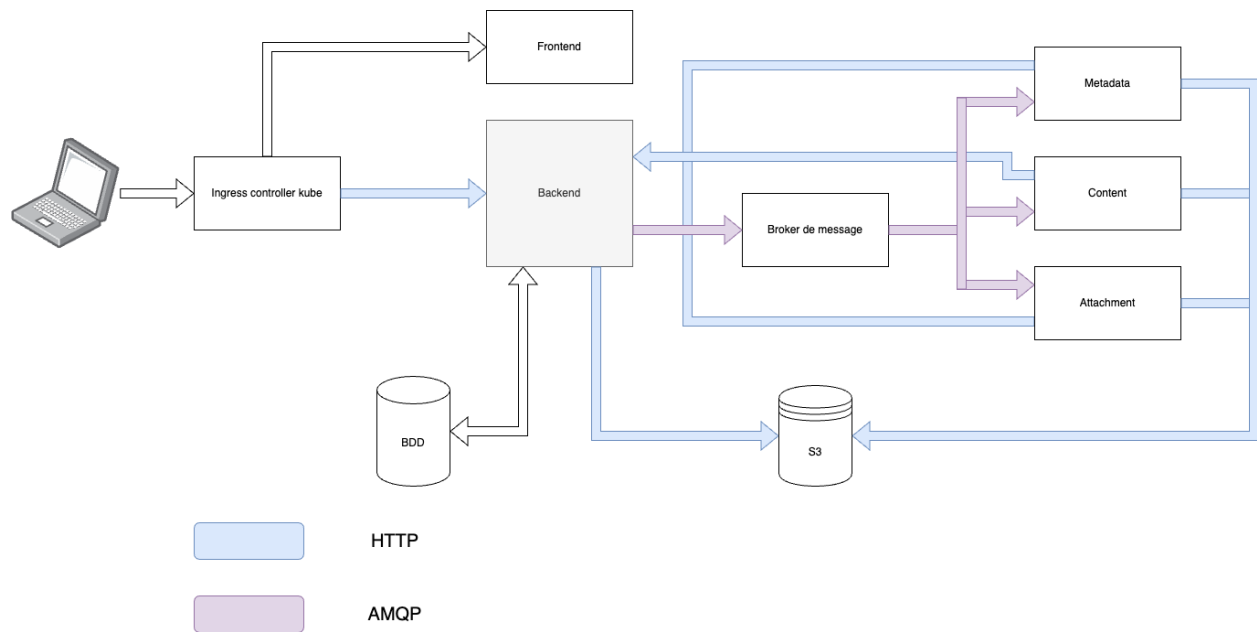


Figure 1 Schéma d'architecture logicielle

Une fois tous les résultats réceptionner un score de risque et un verdict sont calculés.

Une API « publique » est aussi en place. Celle-ci permet l'upload d'un fichier à analyser et la visualisation des résultats. Cela permet ainsi à notre outil de s'interfacer avec d'autres applications.

Le backend est également en charge de l'authentification et de l'autorisation des utilisateurs. Cela permet ainsi aux utilisateurs de s'inscrire sur le site afin d'avoir un historique des analyses et de pouvoir effectuer plus d'analyse qu'un utilisateur non authentifié. Afin d'utiliser l'api, un utilisateur devra générer une clé d'API qui lui permettra de s'authentifier. Enfin un système d'autorisations est aussi en place afin de limiter les accès à certaines ressources aux personnes authentifiées. Par exemple un utilisateur pourra consulter uniquement les résultats de ces analyses.

Micro-services d'analyses

Le premier micro-service **ms-metadata** a la charge de l'analyse des métadonnées des emails. Il récupère les métadonnées de l'email (adresse IP, adresse mail, serveur cote expéditeur...) et effectue des tests visant à :

- Vérifier la réputation de l'adresse IP de l'expéditeur en utilisant l'API Greynoise
- Vérifier la réputation du nom de domaine de l'expéditeur en utilisant l'API Spamchecker
- Vérifier les enregistrements SPF en utilisant les librairies python, et l'utilisation de DKIM par la recherche de balises

Le second micro-service **ms-content** a la charge de l'analyse du contenu des emails. Il effectue des tests visant à :

- Vérifier la présence de liens malveillants en recherchant dans des listes d'urls malveillants gérées par le micro-service et en utilisant l'API GoogleSafeBrowser
- Vérifier la présence de nombreuses fautes d'orthographe en utilisant le module LanguageTool déployé en local
- Vérifier la présence de nombreux mots clés relatifs au spam et au phishing en utilisant des listes de mots clés
- Vérifier la présence de typosquatting dans les domaines des liens et des adresses email en utilisant une liste de domaines
- Vérifier la présence de caractères suspects en utilisant une liste de caractères valides

Le troisième micro-service **ms-attachments** a la charge de l'analyse des pièces jointes des emails. Il effectue des tests visant à :

- Vérifier le type de fichier en utilisant des listes d'extensions
- Vérifier le hash du fichier en utilisant l'API de VirusTotal









Un broker de message pour la communication entre les micro-services

Nous avons opté pour l'intégration du broker de messages [RabbitMQ](#), basé sur le modèle AMPQ, afin d'optimiser la gestion des messages initiés par les utilisateurs. Cette solution nous a permis d'orchestrer efficacement les échanges entre les différents Micro-Services énumérés précédemment. Chaque micro-service est désormais capable de recevoir les messages du broker en se plaçant en attente de demandes. De manière autonome, RabbitMQ gère la file d'attente des demandes et distribue de façon indépendante chaque requête en attente à leur micro-service respectif. Ceci renforce la fluidité de nos processus de communication, contribuant ainsi à une architecture micro-services plutôt réactive.

Une base de données pour les données des utilisateurs et des données de l'application

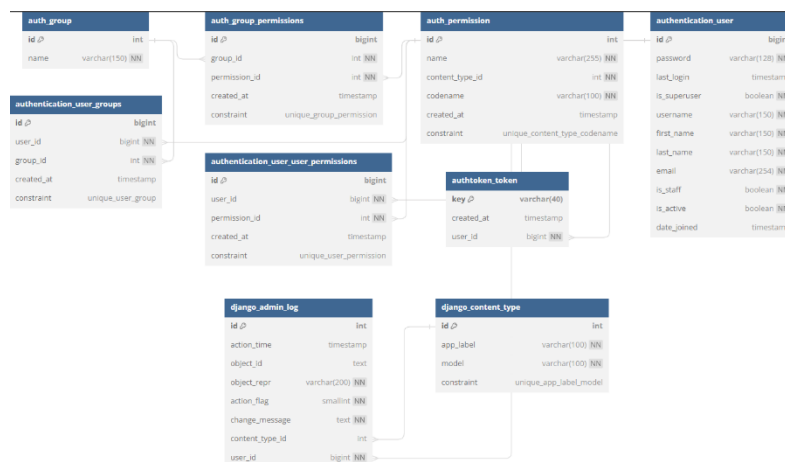
Analyse des fichiers

Concernant les données stockées lors de l'analyse de fichier nous avons décidé de les stocker dans une même table à la fois l'intégralité des résultats remonté par les 3 micro-services mais également des données supplémentaires relatifs aux informations du mail (Date, Sender, Subject...). Il s'agit ici de la table principale pour la gestion du fichier et de ses résultats.

base_email	
uuid 	varchar(100)
created_at	timestamp
user	varchar(100) 
isReady	boolean 
responseMetadataDomain	varchar(100) 
responseMetadataIp	varchar(100) 
responseMetadataSPF	varchar(100) 
responseContentKeywords	varchar(100) 
responseContentLinks	varchar(100) 
responseContentSpelling	varchar(100) 
responseContentTyposquatting	varchar(100) 
responseContentCharacter	varchar(100) 
responseAttachmentFiletype	varchar(100) 
responseAttachmentHash	varchar(100) 
score	int 
recipient	varchar(100) 
sender	varchar(100) 
subject	varchar(100) 
verdict	varchar(100) 
delivery_date	varchar(100) 
responseMetadataDKIM	varchar(100) 

Authentification des utilisateurs (Django) :

Nous avons exploité les fonctionnalités natives de Django pour la gestion des processus d'authentification et la définition des droits des utilisateurs au sein de notre application. Par exemple, nous sommes en mesure de réguler l'accès à des ressources spécifiques, de limiter le nombre d'uploads pour les utilisateurs anonymes, et de définir des règles de sécurité personnalisées. En utilisant ce genre de fonctionnalités de Django, nous renforçons la sécurité de notre application tout en bénéficiant d'une mise en œuvre plus fluide et conforme aux meilleures pratiques de développement web.



Enfin d'autres tables temporaires nous ont été également nécessaire notamment base_uploadfile qui va jouer un rôle uniquement lors de la phase d'upload de fichier.

base_uploadfile			
id	bigint		
upload_on	timestamp		
file	varchar(100)	NN	
uuid	varchar(100)	NN	

django_migrations		django_session	
id	bigint	session_key	varchar(40)
app	varchar(255) NN	session_data	text NN
name	varchar(255) NN	expire_date	timestamp
applied	timestamp		

Choix technologiques

Langages et Frameworks

Nous avons choisi Django pour le back-end car c'est un framework qui est très bien documenté et qui est largement utilisé. Ce framework facilite la mise en place rapide d'un back-end complet avec une base de données et une API. La génération de pages HTML est également simplifiée, tout comme la gestion de l'authentification des utilisateurs. Django offre également la possibilité de créer une API efficacement grâce à Django Rest Framework.

En ce qui concerne les micro-services, notre choix s'est porté sur Python en raison de notre familiarité avec le langage et de sa popularité. Il permet de développer rapidement des micro-services. De plus le back-end est développé en python avec Django, ce qui permet de développer les micro-services plus rapidement bien que les micro-services puissent être développés dans un autre langage.

Nous avons choisi HTML, CSS et Javascript pour le front-end car nous n'avons actuellement pas besoin de fonctionnalités avancées.

Concernant le stockage des fichiers, Minio a été choisi en tant que service de stockage d'objets en raison de sa capacité à stocker des fichiers de manière distribuée tout en offrant un accès via une API. Cette solution s'intègre parfaitement à notre déploiement sur Kubernetes, faisant de Minio un choix idéal pour le stockage de fichiers dans cet environnement.

Nous avons choisi RabbitMQ pour le broker de message car c'est un broker de message très utilisé et qui est très bien documenté. Il permet de communiquer entre les micro-services de manière asynchrone.

Infrastructure

Proxmox a été choisi comme hyperviseur pour faire fonctionner les différentes machines virtuelles nécessaires au fonctionnement et à la sécurisation de l'infrastructure. Il y a deux nodes proxmox fonctionnant dans le cluster : un est dédié au routage avec un CPU modeste et 16Go de RAM l'autre a un CPU plus puissant avec 64Go de RAM. La machine virtuelle hébergeant le kubernetes dispose de 4 vCPU et 8Go de RAM.

Le système d'exploitation choisi pour installer Kubernetes est Rocky Linux version 9. Ici l'objectif est d'avoir un système stable, léger et sécurisé. Rocky linux installe et active par défaut SELinux en mode enforcing. Ceci signifie que SELinux impose ses politiques de sécurité aux applications. Celui-ci est resté tout le long du projet en mode Enforcing ce qui ajoute une couche de sécurité supplémentaire sur la machine virtuelle.

OPNsense a été choisi comme routeur et pare-feu. Celui-ci contient toutes les fonctionnalités dont nous avons besoins pour sécuriser notre infrastructure. Il apporte notamment la possibilité de lire des blocklist accessible sur internet et d'intégrer celles-ci dans un pare-feu et de mettre à jour cette liste à intervalles régulier.

Crowdsec est une application de cybersécurité qui lit les logs des applications, pare-feu et serveur web afin de détecter de potentielles attaques et prendre automatiquement des décisions comme un bannissement temporaire sur une ou plusieurs adresses IP si celle-ci est considéré comme malveillante. Si une instance Crowdsec décide de bannir une adresse IP, ce bannissement est répliqué sur toutes les instances Crowdsec lié au compte utilisateur. Le logiciel inclus aussi ses propres listes d'IP bloqué directement intégré dans le pare-feu de OPNsense.

Kubernetes est le chef d'orchestre de la gestion de notre application et des autres applications utilisés par le projet. Nous utilisons une version minifié nommé K3S, celle-ci est plus légère et surtout plus simple à installer. Dans un déploiement en production de notre application, il faut configurer une storage class ou un volume persistant fourni par un NAS ou autre afin d'avoir une solution de stockage de données plus robuste que celle fourni par défaut qui utilise le driver host-path. À noter que la documentation officielle de Kubernetes¹ ne recommande pas l'utilisation de ce driver en production.

Harbor est utilisé pour stocker les images docker utilisées par Kubernetes. Celui-ci est exposé sur internet avec une limitation de requête pour éviter les tentatives de brute-force.

GitHub action permet de construire les images docker, vérifier si celles-ci contiennent des vulnérabilités et de pousser celles-ci sur notre Harbor. Il se charge aussi du déploiement de manière automatique sur notre environnement de développement et sur la production. Une autre tâche exécutée en parallèle lance un scan statique du code pour Sonarqube.

¹ <https://kubernetes.io/docs/concepts/storage/volumes/#hostpath>

Infrastructure

Le schéma ci-dessous résume l'architecture qui a été réalisé dans le projet :

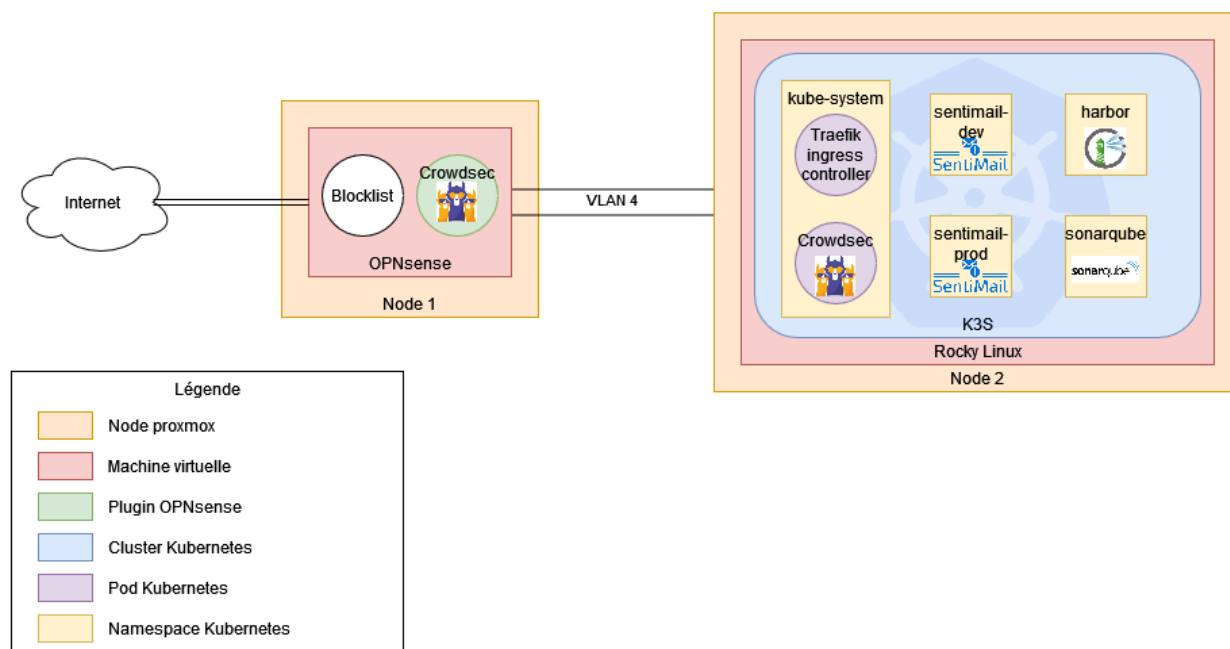


Figure 2 - Schéma d'architecture de l'infrastructure

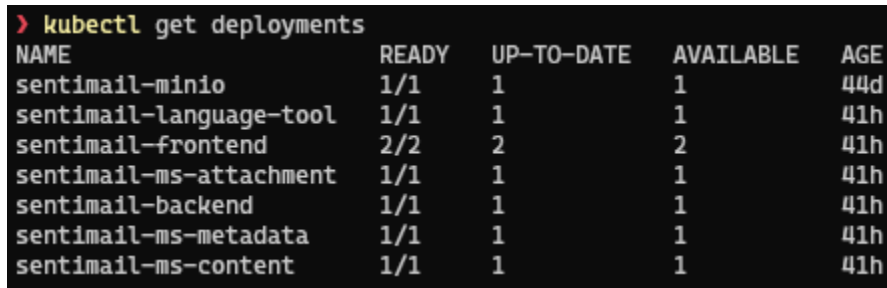
Remarques sur le schéma :

- Le VLAN 4 est un réseau isolé sur mon OPNsense que j'ai nommé « DMZ », OPNsense autorise HTTP, HTTPS, DNS et NTP vers internet, pas à destination d'un autre réseau. Les autres protocoles sont bloqués.
- Le VLAN 4 est utilisé entre la communication à gauche de la VM OPNsense et avec l'hyperviseur à droite. (La machine virtuelle Rocky linux n'a pas la notion de VLAN).
- Il y a deux instances Crowdsec qui n'ont pas les mêmes responsabilités :
 - L'instance sur OPNsense se charge de bloquer les IP qui ont été blacklist par n'importe quelle instance Crowdsec. L'idée ici est de bloquer l'IP dès que possible au niveau réseau. À noter qu'il y a aussi une blacklist mise à jour automatiquement qui est présente avant les filtrages de Crowdsec.
 - L'instance sur Kubernetes surveille les logs de Traefik et va bloquer certaines tentatives d'exploitation de failles et va bloquer toute tentative de brute force.
- Rocky linux a SELinux actif en mode enforcing.

Choix de déploiement

Le déploiement de l'application sur le cluster Kubernetes utilise des fichiers YAML avec Kustomize pour la partie CD assurée par GitHub Actions.

Dans chaque namespace de l'application, c'est-à-dire développement et production, les deployments suivant sont présents :



```
> kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
sentimail-minio	1/1	1	1	44d
sentimail-language-tool	1/1	1	1	41h
sentimail-frontend	2/2	2	2	41h
sentimail-ms-attachment	1/1	1	1	41h
sentimail-backend	1/1	1	1	41h
sentimail-ms-metadata	1/1	1	1	41h
sentimail-ms-content	1/1	1	1	41h

Figure 3 - Liste des deployments présent sur le namespace de développement

On retrouve ici les différents composants de l'application :

Le frontend, backend et les 3 micro-services ms-attachement, ms-content, ms-metadata. On peut aussi voir le serveur « language tool » qui permet à ms-content de vérifier l'orthographe d'un mail.

L'environnement de développement et la production sont dans deux namespaces différents afin d'assurer une isolation complète entre les deux : Les composants nécessaires au fonctionnement de l'application, c'est-à-dire : base de données, minio, rabbitmq sont présents eux aussi en double dans les mêmes namespaces.

Minio, PostgreSQL et RabbitMQ sont tous les trois déployés à l'aide de leur chart HELM, PostgreSQL et RabbitMQ ne sont pas visible sur la capture d'écran ci-dessus car ceux-ci sont déployés comme statefulset.

Intégration et déploiement continu avec GitHub Actions

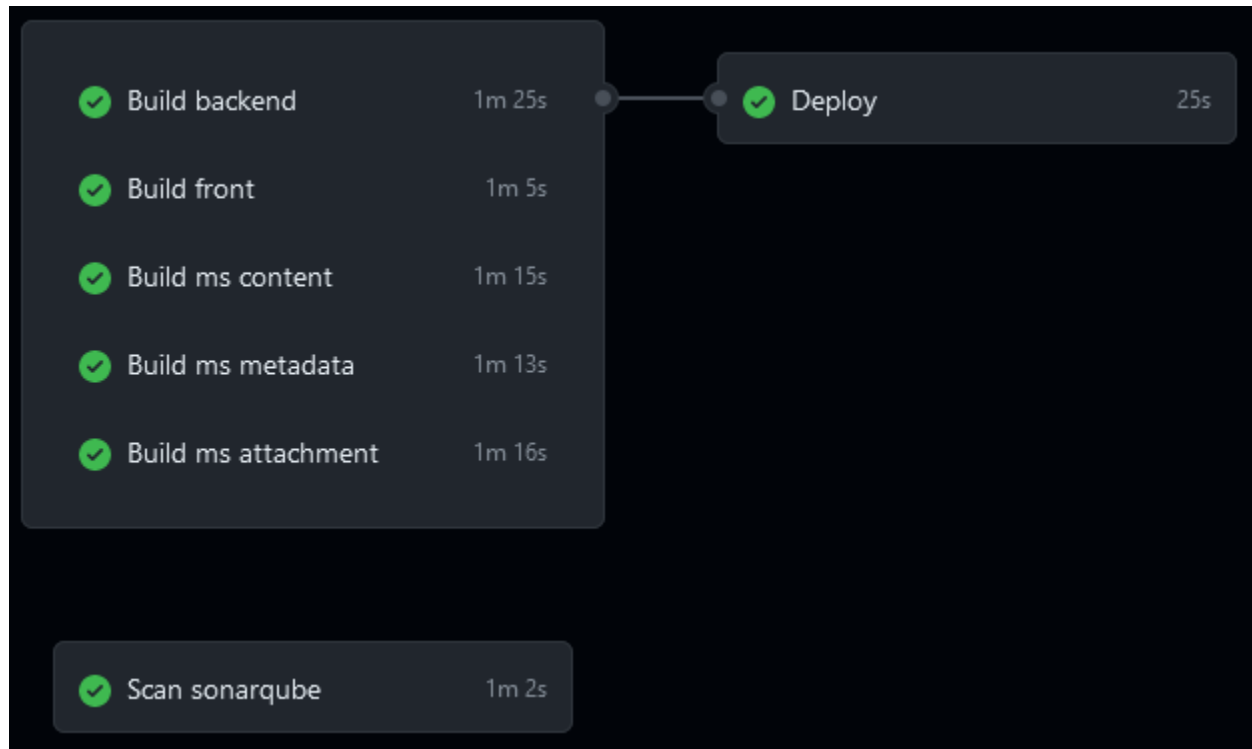


Figure 4 - Tâches présentes sur GitHub Actions

Les 5 jobs de builds font tous les mêmes opérations :

1. Construction d'une image docker
2. Vérification des potentielles vulnérabilités présentes sur l'image construite
3. Si aucune vulnérabilité moyenne ou supérieur est présente, on pousse cette image sur le Harbor.

Le job deploy dépend de si la construction des images s'est produite sans erreurs pour les différents jobs : Celui-ci adapte les manifests Kubernetes : changer les versions des images utilisées, changer le namespace en fonction de si on est en développement ou en production, adapter la valeur des différents identifiants de connexion (notamment base de données, minio...) et de pousser les nouveaux manifests sur Kubernetes avec un service account.

Un job pour effectuer le scan Sonarqube s'exécute uniquement sur la branche de développement.