

Cahier des charges projet MLOps

**Détection d’anomalies pulmonaires**

**(COVID et Pneumonies Virales)**

## Contexte et Objectifs

Début 2020, la propagation rapide du coronavirus (COVID-19) a entravé la capacité des systèmes de santé à réaliser les diagnostics et tests requis dans les délais imposés par la pandémie. Ainsi, une recherche active de solutions alternatives pour le dépistage a été initiée.

En raison des effets significatifs du COVID-19 sur les tissus pulmonaires, l'usage de l'imagerie par radiographie thoracique s'est avéré incontournable pour le dépistage et le suivi de la maladie lors de la crise COVID

**Problématique actuelle**

Aujourd’hui, l’isolation des patients COVID reste d’actualité, mais il est aussi question de traiter de manière efficace les patients atteints d’une anomalie pulmonaire.

**Un Objectif double**

* **Isolation des patients COVID :** En minimisant les faux négatifs, une détection maximale des maladies comme le COVID permettra d’atténuer la propagation du virus.
* **Traiter les patients atteints d’anomalie pulmonaire :** La détection de pneumonie Virale permet de traiter les patients

Afin d’améliorer la précision de l’identification et afin que notre outil soit pertinent pour le métier et rendent de réels services au corps médical :

* Une qualification précise de l’anomalie pulmonaire est nécessaire. Pour cette raison, une opacité pulmonaire est écartée des classifications, alors que la détection d’une Pneumonie Virale peut indiquer un état de santé fragile du patient qui permet au radiologue d’orienter le patient vers des analyses complémentaires.
* Il se doit d’atteindre un seuil de résultats élevé. Selon Yang et al. (2020), les modèles prouvant une précision de 89% ou plus sont tout à fait appropriés à l’usage clinique.
* Une qualification rapide qui permet de réaliser un premier diagnostic par les internes avant de solliciter des radiologues pour un 2e avis si besoin.

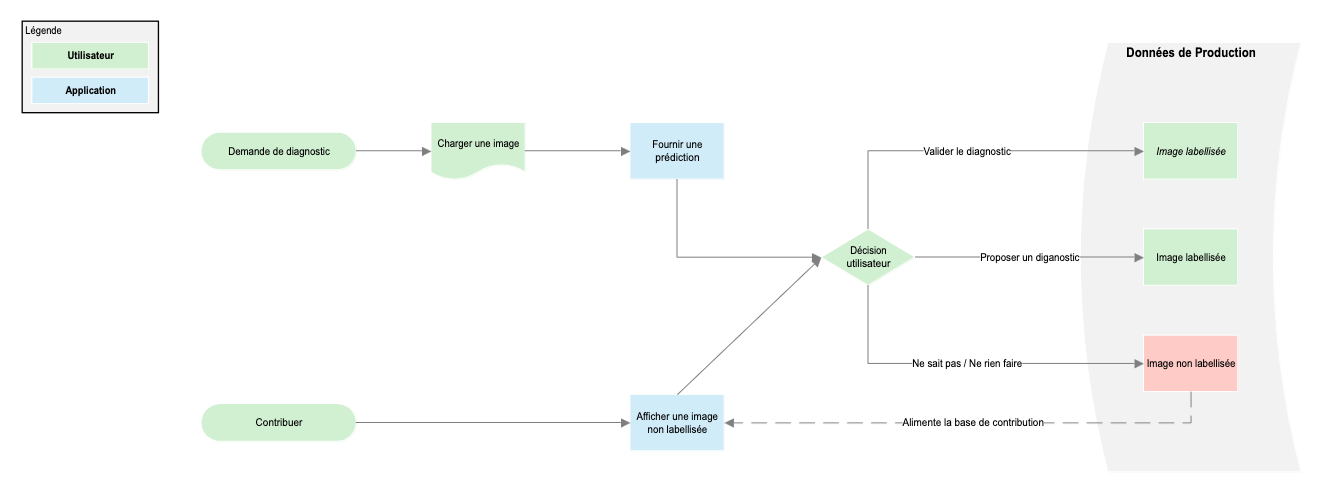
**Commanditaire**

Unité de radiologie d’un hôpital public

**Utilisateurs**

**Internes** : Effectuer une première demande de classification

**Radiologues** : Améliorer l’outil en évaluant les prédictions



**Administrateur**

Data Scientists : Choix des données, de l’architecture du modèle ainsi que son entrainement et évaluation

MLOps engineer : Chargé du déploiement de nouveaux modèles, du Monitoring du service l’application et de la performance du modèle ainsi que du réentrainement du modèle si nécessaire

**Contexte d’intégration**

L'application sera déployée sur un serveur cloud.

**Support d’utilisation**

Interface graphique accessible de manière sécurisée via un navigateur web et implémentée via Streamlit

## Modèle

Il s’agit d’une classification basée sur des **images** de radiographie pulmonaire, le modèle final sera basé sur un modèle pré-entrainée de **Deep Learning** et un **Réseau de neurones convolutif** (CNN)

**Type de modèle**

Le type de modèle pré-entrainé qui a été sélectionné est le Réseau de neurones convolutif pré-entrainé **EfficientNetB0**. En effet, ce type d’architecture a présenté des avantages en termes de temps d’exécution et de précision lors d’un premier Benchmark avec d’autres modèles (ResNet50, VGG16 et VGG19)

**Données en entrée :** Images de dimensions 224x224 et en mode couleur (RGB)

Alors que le modèle pré-entrainé sur la base de données ImageNet contient 1000 classes en sortie, le nombre sera limité à notre besoin

**3 Classes en sortie**

* Normal
* COVID
* Pneumonie Virale

**Stratégie de Finetuning**

* Utilisationdu Keras Tuner **RandomSearch** pour trouver les meilleurs hyperparamètres :
  + Taux d'apprentissage
  + Nombre de neurone par couche dense (units)
  + Nombre de couches de Dropout et taux de Dropout/désactivation
  + Fonction de régularisation l2 et le taux d’apprentissage associé
* Dégel des couches du modèle EfficientNetB0 de type Conv2D pour réentrainement
* Ajout de la couche de sortie de 3 neurones qui correspond au nombre de classes en sortie
* Optimizer Adam

**Métrique d’évaluation**

* **Temps d’entrainement :** Métrique d’évaluation Data Scientist
* **Temps de prédiction :** Métrique d’évaluation métier
* **Sensibilité/Rappel :** Pour la classe COVID en particulier, l’objectif étant de maximiser le nombre de vrais positifs VP et de minimiser le nombre de faux négatifs FN (Rappel = (nombre de VP)/(nombre de VP + nombre de FN)
* **F1 score :** Pour un bon compromis en le Rappel et la précision, en particulier pour les classifications de Pneumonie Virale

## Base de données

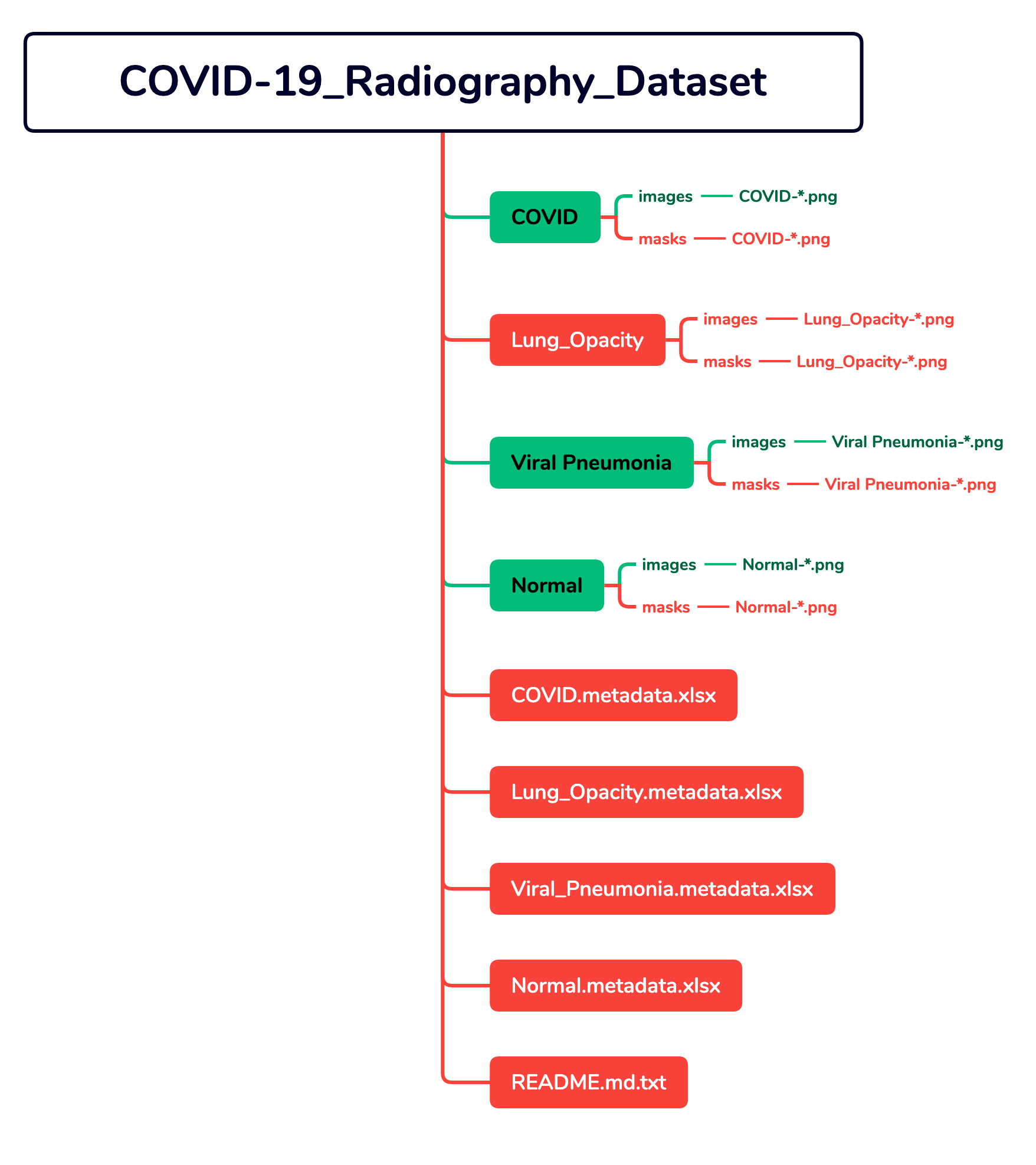
En début du projet, une base de données KAGGLE sera utilisée pour construire notre **base de données de référence**.

Cependant, au fur et à mesure de l’utilisation du modèle et des retours utilisateurs, des nouvelles **données de production** vont enrichir ces données de références pour contribuer à améliorer notre modèle.

**La base de données KAGGLE**

URL : <https://www.kaggle.com/datasets/tawsifurrahman/covid19-radiography-database>

* La fréquence de rafraîchissement prévue est mensuelle.
* Arborescence de la base de données :



*Les Données non utilisées (en rouge) :*

* Les masques
* La classe Lung Opacity
* Les fichiers metadata.xlsx

*Le traitement qui sera réalisé*

*Actions réalisées après récupération des données :*

* Une incrémentation de version des données est réalisée après chaque récupération de nouvelles données de KAGGLE
* Un fichier **metadata.csv** est créé à chaque récupération de nouvelle version avec des informations telle que le md5, date d’ajout, classe et un flag permettant d’ignorer un fichier lors de l’intégration aux données de référence, ainsi que le statut pour chaque image par rapport à la récupération précédente (UNCHANGED si déjà présente, ADDED s’il s’agit d’un nouvel ajout)

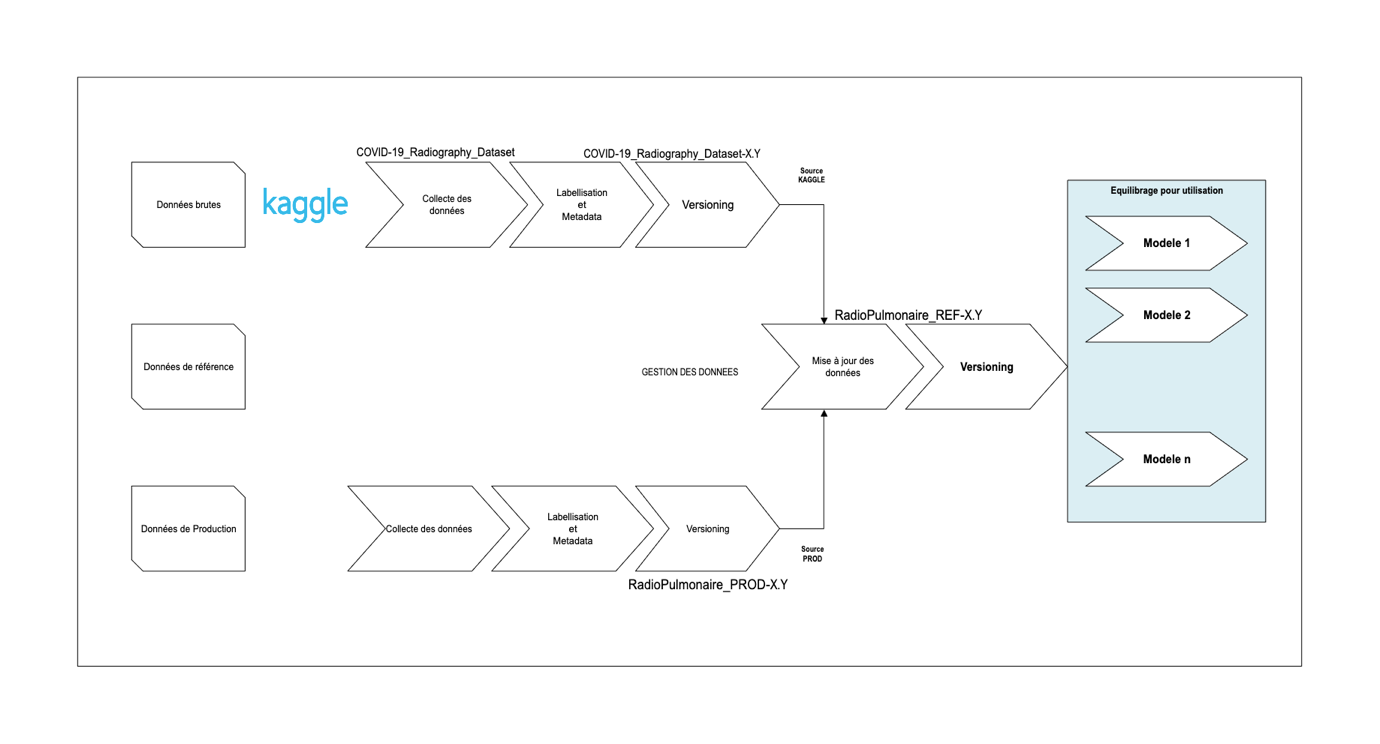
**Gestion des nouvelles données dans la base de données de référence :**

* Données de Production : Pour lesquelles l’utilisateur « Radiologue » aura fourni une validation ou une correction de la classification
* Techniquement, que ce soit de source KAGGLE ou de PRODUCTION, seules les images dont le hash MD5 n’existe pas dans la base de référence sera ajouté
* Une incrémentation de version des données de référence est réalisée à chaque intégration de nouvelles données.

**Le traitement des données pour les modèles :**

* L’équilibrage des données se fait par la sélection aléatoire de nombre d’image correspondant à la classe minimale (ici Viral Pneumonia : 1346)
* Les données sont également divisées en trois sous-ensembles de données pour la validation, l'entraînement et le test du modèle (respectivement 60%, 20% et 20%)

**Aperçu du Cycle de vie des données**



## APIs

3 APIs seront réalisées dans le cadre du projet afin de répondre aux besoin des utilisateurs et des administrateurs :

* API User : Dédiée à la prédiction
* API Administrateur : Dédiée aux Data Scientist et aux administrateurs, pour réaliser les actions courantes de gestion des données et des modèles
* API Monitoring : Dédiée à la surveillance des performances du modèle et de l’état de service
* L’authentification se fait par profil

**Endpoints**

## API User

GET

@/

Cette requête permet de vérifier si l'API est en fonctionnement correctement. Elle renvoie un statut code "200" si tout fonctionne comme prévu.

POST

@/predict

Cette requête permet de faire une prédiction sur une image envoyée à l'API. L'image doit être au format .jpg, .jpeg ou .png.

La réponse est un objet JSON contenant le nom du modèle utilisé pour la prédiction, la prédiction elle-même, le niveau de confiance, le temps de prédiction et le chemin de l'image téléchargée.

POST

@/login

Cette requête permet à un utilisateur de se connecter à l'API en fournissant un nom d'utilisateur et un mot de passe.

La réponse est un objet JSON contenant le statut de la connexion, le jeton d'accès, le nom d'utilisateur, le nom complet de l'utilisateur, le rôle de l'utilisateur et le type de jeton.

POST

@/add\_image

Cette requête permet d'ajouter une image et son label au dataset. L'image doit être fournie sous forme de chemin d'accès et le label doit être une chaîne de caractères.

La réponse est un objet JSON contenant le statut de l'opération.

GET

@/get\_classes

Cette requête permet de récupérer la liste des classes disponibles pour la prédiction. La réponse est un objet JSON contenant la liste des classes.

POST

@/update\_log\_prediction

Cette requête permet de mettre à jour le journal de prédiction avec le retour utilisateur. L'identifiant de prédiction et le label doivent être fournis.

La réponse est un objet JSON contenant le statut de l'opération.

POST

@/token

Cette requête permet d'obtenir un jeton d'accès en fournissant un nom d'utilisateur et un mot de passe.

La réponse est un objet JSON contenant le jeton d'accès et le type de jeton.

GET

@/metrics

Cette requête permet de récupérer les métriques de l'API au format Prometheus.

La réponse est une chaîne de caractères contenant les métriques au format texte.

## API Admin

GET @/

Cette requête permet de vérifier si l'API est en fonctionnement correctement. Elle renvoie un statut code "200" si tout fonctionne comme prévu.

GET @/metrics

Cette requête permet de récupérer les métriques de l'API au format Prometheus.

La réponse est un objet JSON contenant contenant les données des métriques.

GET @/download\_dataset

Cette requête permet de lancer le téléchargement du dataset Kaggle.

La réponse est un objet JSON contenant un statut code "200" et un message indiquant que le téléchargement est terminé.

POST @/update\_dataset

Cette requête permet de mettre à jour le dataset de référence avec des données de production ou de kaggle. Elle prend en paramètres optionnels le chemin du dataset, le type de source (par défaut "KAGGLE") et l'ID du dataset de base.

La réponse est un objet JSON contenant un statut code "200" et un message indiquant que la mise à jour est terminée.

POST @/train\_model

Cette requête permet d'entraîner un modèle. Elle prend en paramètres optionnels si l'on souhaite réentraîner un modèle existant, le nom et la version du modèle, si l'on souhaite inclure les données de production, si l'on souhaite équilibrer les données, la version du dataset (par défaut la dernière version), le nombre d'epochs (par défaut défini dans le fichier de configuration) et le nombre de trials (par défaut défini dans le fichier de configuration).

La réponse est un objet JSON contenant un statut code "200", un message indiquant que l'entraînement est terminé, ainsi que l'ID du run, le nom et la version du modèle.

GET @/clean\_dataset

Cette requête permet de nettoyer et de restructurer un dataset si nécessaire.

La réponse est un objet JSON contenant un statut code "200", un message indiquant que le nettoyage est terminé, ainsi que la version actuelle et la nouvelle version du dataset.

POST @/retrain\_model

Cette requête permet de réentraîner un modèle. Elle prend en paramètres le nombre d'epochs, le nombre de trials et l'option d'entraînement (par défaut "ZERO" pour entraîner un modèle à partir de zéro avec le dataset actuel).

La réponse est un objet JSON contenant un statut code "200", un message indiquant que le réentraînement est terminé, ainsi que l'ID du run, le nom et la version du modèle.

POST @/make\_model\_prod\_ready

Cette requête permet de mettre à jour le tag d'un modèle pour le rendre prêt à être déployé en production. Elle prend en paramètre le numéro de version du modèle.

La réponse est un objet JSON contenant un statut code "200" et un message indiquant que la mise à jour est terminée.

POST @/deploy\_ready\_model

Cette requête permet de déployer un modèle prêt pour la production.

La réponse est un objet JSON contenant un statut code "200" et un message indiquant que le déploiement en production est terminé.

POST @/force\_model\_serving

Cette requête permet de forcer le déploiement d'un modèle en production. Elle prend en paramètre le numéro de version du modèle.

La réponse est un objet JSON contenant un statut code "200" et un message indiquant que le déploiement est terminé.

GET @/get\_models\_list

Cette requête permet de lister tous les modèles disponibles.

La réponse est un objet JSON contenant un statut code "200", un message indiquant que la requête est un succès, ainsi qu'une liste des modèles disponibles et des informations sur les runs MLFlow.

POST @/get\_runs\_info

Cette requête permet de récupérer les informations sur un ou plusieurs runs MLFlow. Elle prend en paramètre une liste d'IDs de runs.

La réponse est un objet JSON contenant un statut code "200", un message indiquant que la requête est un succès, ainsi que les informations sur les runs demandés.

POST @/get\_datasets\_list

Cette requête permet de lister tous les datasets disponibles. Elle prend en paramètre optionnel le type de dataset (par défaut "REF" pour les datasets de référence).

La réponse est un objet JSON contenant un statut code "200", un message indiquant que la requête est un succès, ainsi qu'une liste des datasets disponibles.

POST @/add\_images

Cette requête permet d'ajouter un batch d'images et leurs labels. Elle prend en paramètre une liste de dictionnaires contenant le chemin de chaque image et son label.

La réponse est un objet JSON contenant un statut code "200", un message indiquant que la requête est un succès, ainsi que la version actuelle et la nouvelle version du dataset.

## API Monitoring

GET @/

Cette requête permet de vérifier si l'API est en fonctionnement correctement.

La réponse est un objet JSON contenant un statut code "200" si tout fonctionne comme prévu.

GET @/metrics

Cette requête permet de récupérer les métriques de l'API au format Prometheus. Les métriques incluent le temps de traitement des requêtes, le nombre total de requêtes, la durée des requêtes par méthode ou point de terminaison, la différence entre la nouvelle moyenne et la moyenne originale, et la différence entre le nouvel écart-type et l'écart-type original.

GET @/drift\_metrics

Cette requête permet de lancer le calcul des métriques pour la détection de drift. Le calcul inclut le nom du modèle, la nouvelle moyenne, la moyenne originale, le nouvel écart-type, l'écart-type original, la différence entre la nouvelle moyenne et la moyenne originale, la différence entre le nouvel écart-type et l'écart-type original, et un indicateur de drift.

La réponse est un objet JSON contenant ces informations.

## Testing & Monitoring

## Tests unitaires

Les tests unitaires doivent assurer le fonctionnement :

* Du téléchargement et mise à jour des données KAGGLE
* De la mise à jour des données de références (données KAGGLE ou PROD)
* De la journalisation des prédictions pour les performances du modèle
* De l’Entraînement ou réentrainement des modèles
* Du calcul de Model Drift

## Monitoring

Le monitoring du service et des performances du modèle :

* Etat de service des APIs et des applications streamlit et MLFlow
* Calcul périodique du drift sur le modèle de production afin de détecter un possible drift.

**Evaluation des performances et seuils :**

* Temps moyen de prédiction : Maximum 1 seconde
* Evolution des prédictions validées et invalidées par les utilisateurs : Seuil Moins 89% des prédictions validées
* Calcul Détection de Drift du modèle : Différence sur la moyenne et écart type des prédictions du modèle sur les données de production et les données de référence :
  + Sur la moyenne : 0.01
  + Sur l’écart type : 0.05

**Réentrainement du modèle**

L’option par défaut (automatique) : Réentrainement du **modèle de production** sur les **données de production**

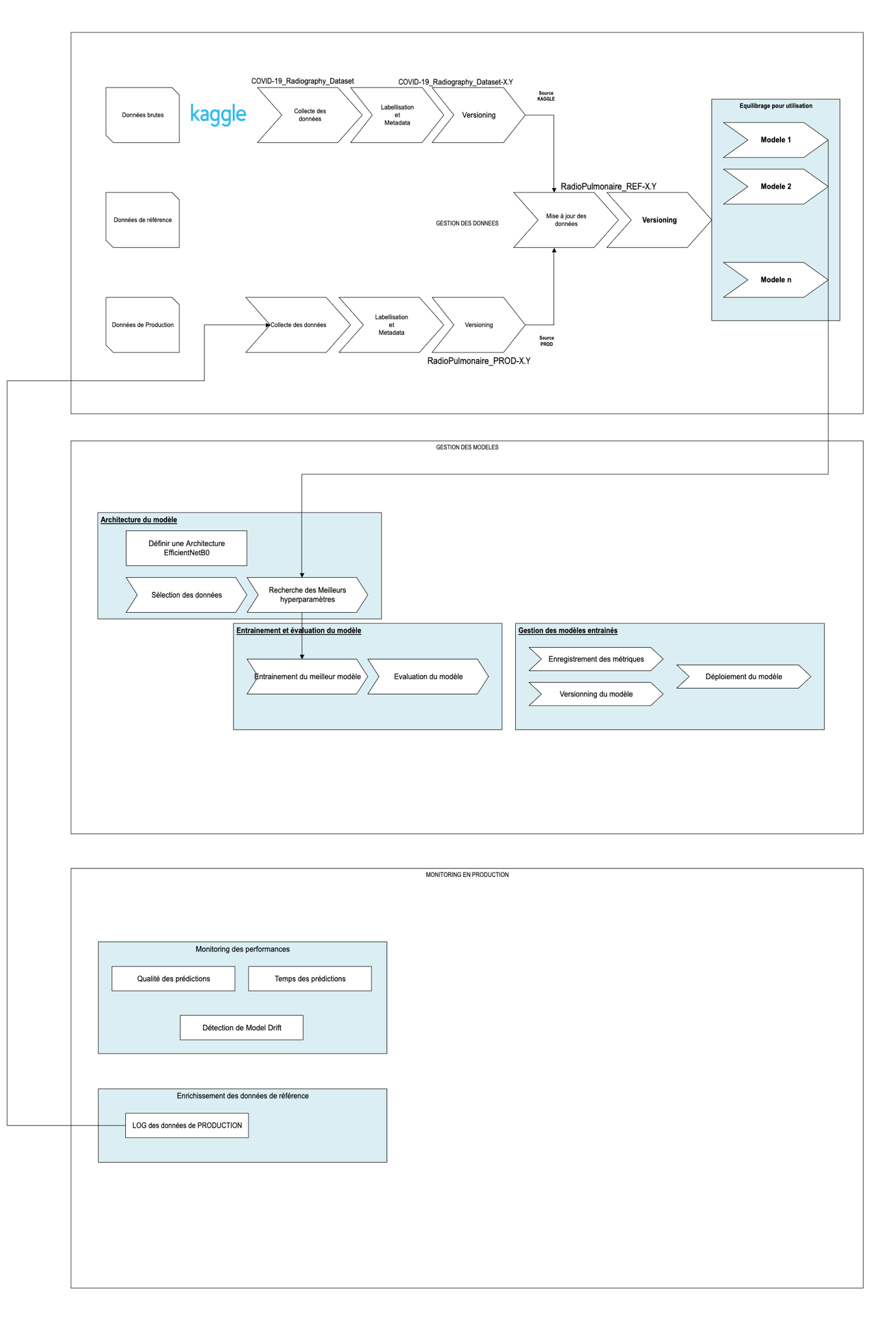
**Action de ré-entrainement :** Un batch sera exécuté périodiquement pour calculer le drift, et exécuter un ré-entrainement (option par défaut) si un drift du modèle est détecté, et envoyer également une notification à l’administrateur

L’Administrateur aura également la possibilité d’entrainer manuellement un nouveau modèle sur la base des données de références **et** de production

## Roadmap projet

## Schéma d’implémentation

**Gestion des données, modèles et monitoring**



Dans cette partie, vous devrez créer un schéma récapitulatif du projet, qui intègre les différentes composantes du projet et leurs interactions. Ce dernier n’a pas besoin d’être normalisé, mais devra respecter un code couleur compréhensible et se doit d’être le plus exhaustif possible. Vous pourrez pour ce faire vous aider des outils <https://app.diagrams.net/> ou <https://docs.google.com/drawings>

Voici un exemple de schéma d’implémentation :

