**Machine learning canvas**

**Table des matières**

[INTRODUCTION 4](#_Toc197448109)

[Défis Méthodologiques 4](#_Toc197448110)

[Approche Algorithmique 4](#_Toc197448111)

[PREDICTION TASK 4](#_Toc197448112)

[Objectif 4](#_Toc197448113)

[Entité cible 4](#_Toc197448114)

[Horizon temporel 4](#_Toc197448115)

[DECISIONS 5](#_Toc197448116)

[Préparation des données 5](#_Toc197448117)

[Modélisation 6](#_Toc197448118)

[Prédictions 7](#_Toc197448119)

[Transformation en valeur finale pour l'utilisation 7](#_Toc197448120)

[VALUE PROPOSITION 7](#_Toc197448121)

[Qui est l’utilisateur final ? 7](#_Toc197448122)

[Objectif des utilisateurs 7](#_Toc197448123)

[Bénéfices des utilisateurs vis-à-vis du système 7](#_Toc197448124)

[Flux de travail / Interface 8](#_Toc197448125)

[DATA COLLECTION 9](#_Toc197448126)

[Stratégie de ‘’Train’’ initial 9](#_Toc197448127)

[Mise à jour continue 9](#_Toc197448128)

[Coûts et contraintes pour observer les résultats 10](#_Toc197448129)

[DATA SOURCES 10](#_Toc197448130)

[APIs pour les données météorologiques 10](#_Toc197448131)

[Sites web pour l'exploration des données météo 11](#_Toc197448132)

[Bases de données météorologiques (tables SQL ou autres formats) 11](#_Toc197448133)

[Méthodes pour intégrer les données dans un système de machine learning 11](#_Toc197448134)

[IMPACT SIMULATION 11](#_Toc197448135)

[Déploiement des modèles 11](#_Toc197448136)

[Données d’essaies pour évaluer les performances 11](#_Toc197448137)

[Valeurs de coût/gain pour les décisions correctes ou incorrectes 12](#_Toc197448138)

[Contrainte d’équité 12](#_Toc197448139)

[MAKING PREDICTIONS 13](#_Toc197448140)

[Quand faisons nous des prédictions en temps réel ou pas batch ? 13](#_Toc197448141)

[Prédictions en temps réel 13](#_Toc197448142)

[Prédictions par batches (lots) 13](#_Toc197448143)

[Temps disponible pour le feature engineering, la prédiction et le post-traitement 14](#_Toc197448144)

[Prédictions en temps réel 14](#_Toc197448145)

[Prédictions par batches 14](#_Toc197448146)

[Objectif de calcul 14](#_Toc197448147)

[Prédictions en temps réel 14](#_Toc197448148)

[Prédictions par batches 15](#_Toc197448149)

[API DETAILS & MLFLOW INTEGRATION 16](#_Toc197448150)

[Descriptif 16](#_Toc197448151)

[Framework API - FastAPI 16](#_Toc197448152)

[PIPELINE DE PREDICTIONS 17](#_Toc197448153)

[Extraction des données : 17](#_Toc197448154)

[Préparation des données 17](#_Toc197448155)

[Chargement du modèle 17](#_Toc197448156)

[Génération des prédictions 17](#_Toc197448157)

[Stockage des résultats 17](#_Toc197448158)

[Dépendances entre tâches 18](#_Toc197448159)

[Orchestration avec Airflow 18](#_Toc197448160)

[Vue d'ensemble de l'architecture d'orchestration 18](#_Toc197448161)

[Fonctionnement des DAGs 19](#_Toc197448162)

[Avantages d'Airflow pour ce projet 20](#_Toc197448163)

[Transmission du run\_id MLflow aux endpoints API 20](#_Toc197448164)

[Vérification de l'intégrité des données avec hachage 21](#_Toc197448165)

[Conclusion 22](#_Toc197448166)

[Intégration de MLflow 22](#_Toc197448167)

[Vue d'ensemble de l'architecture MLflow 22](#_Toc197448168)

[Points clés de l'intégration MLflow 23](#_Toc197448169)

[Serveur de Tracking MLflow 24](#_Toc197448170)

[Configuration centralisée 25](#_Toc197448171)

[Registre de Modèles MLflow 26](#_Toc197448172)

[Mécanisme de Promotion du Modèle Champion 26](#_Toc197448173)

[Avantages de l'Approche par Alias 28](#_Toc197448174)

[Runs de déploiement et traçabilité 28](#_Toc197448175)

[Bénéfices et leçons apprises 28](#_Toc197448176)

[Conclusion 29](#_Toc197448177)

[Pipeline CI/CD 29](#_Toc197448178)

[Architecture globale du pipeline CI/CD 30](#_Toc197448179)

[Job de test : Assurance qualité du code 31](#_Toc197448180)

[Job de déploiement : La pierre angulaire du pipeline 32](#_Toc197448181)

[Configuration du Job de Déploiement 32](#_Toc197448182)

[Préparation du déploiement 32](#_Toc197448183)

[Récupération des Images Docker 33](#_Toc197448184)

[Configuration de l'Environnement d'Exécution 34](#_Toc197448185)

[Démarrage des Services 34](#_Toc197448186)

[Initialisation et Vérification 35](#_Toc197448187)

[Déclenchement du pipeline initial 36](#_Toc197448188)

[Pourquoi Docker Compose pour le déploiement ? 36](#_Toc197448189)

[Avantages du déploiement avec Docker Compose 36](#_Toc197448190)

[Runner auto-hébergé sur la VM 37](#_Toc197448191)

[Avantages du runner auto-hébergé 37](#_Toc197448192)

[Utilisation de Localhost pour les vérifications 37](#_Toc197448193)

[Gestion des secrets avec GitHub 38](#_Toc197448194)

[Conclusion 38](#_Toc197448195)

[MONITORING 39](#_Toc197448196)

[Mesures pour quantifier la création de valeur pour les utilisateurs finaux 39](#_Toc197448197)

[Mesures pour quantifier la création de valeur pour les entreprises 40](#_Toc197448198)

[Mesures pour la performance du modèle ML 40](#_Toc197448199)

[DEPLOIEMENT & STRATEGIES DE MISE EN PRODUCTION 41](#_Toc197448200)

[Conteneurisation 41](#_Toc197448201)

[Monitoring Proactif 41](#_Toc197448202)

[Monitoring & Déploiement 42](#_Toc197448203)

[Axes d'Amélioration 43](#_Toc197448204)

[Considérations Éthiques et Environnementales 43](#_Toc197448205)

[Conclusion Technique 43](#_Toc197448206)

# **INTRODUCTION**

La prédiction météorologique représente un **défi complexe de classification binaire** où **l'objectif est de prédire la probabilité de précipitations pour le lendemain avec une précision maximale.**

## Défis Méthodologiques

* Haute variabilité des données météorologiques
* Interactions non-linéaires entre variables
* Nécessité de modèles robustes et adaptatifs

## Approche Algorithmique

Pour ce projet de prédiction météo du lendemain, plusieurs algorithmes et outils de Machine Learning pourraient être efficaces, aussi certains d’entres eux ont été envisagés :

1. **Random Forest** : basé sur des arbres de décision, cet algorithme fonctionne bien pour des données avec des variables multiples (température, humidité, précipitations, vent, etc.).
2. **Logistic regression** : plus adapté aux données dont la variable à expliquer est binaire (1 : il va pleuvoir ; 2 : il ne va pas pleuvoir).
3. **Gradient Boosting Machines (GBM)** : inclut des modèles comme XGBoost, LightGBM,….. Ils sont performants sur des données tabulaires.
4. **Réseaux de neurones artificiels (ANN)** : utiles pour détecter des modèles complexes dans des données météorologiques historiques
5. **LSTM (Long Short-Term Memory)** : Conçu pour prédire des séquences temporelles, ce qui peut être un bon choix pour des données météorologiques.

**Dans l'implémentation actuelle**, seul le modèle **Random Forest** a été développé pour notre projet.

Les rubriques suivantes ont pour **objectif de** **décrire l'architecture et le fonctionnement de notre système de prédiction météorologique.**

# **PREDICTION TASK**

Objectif**:** Notre projet vise à prédire la présence de pluie pour le lendemain (RainTomorrow) à partir de données météorologiques actuelles. **Il s'agit d'une classification binaire avec deux résultats possibles** : "Yes" (pluie) ou "No" (pas de pluie).

Entité cible**:** Les prédictions seront effectuées pour des zones géographiques spécifiques identifiées par la variable "Location".

Horizon temporel**:** Nos prédictions portent sur un horizon de 24 heures, ce qui signifie que nous devrons attendre un jour pour évaluer la précision de nos prévisions.

1. **Type de tâche** : classification binaire ; on pourra prédire si ‘’RainTomorrow’’ sera "Yes" ou "No" (pluie ou pas).
2. **Entité sur laquelle les prédictions seront faites** : prédiction est faite sur la météo (spécifiquement la pluie) pour le lendemain dans une certaine "location" (région ou ville).
3. **Résultats possibles** : les résultats sont binaires "Yes (pluie)" ou "No (pas de pluie)" pour la variable ‘’RainTomorrow’’.
4. **Temps d'attente avant l'observation** : 24 heures

# **DECISIONS**

Préparation des données **:**

**► Nettoyage des données**

* Gestion des valeurs manquantes, conversion des variables catégorielles en variables numériques par encodage, et gestion des valeurs aberrantes.

**Gestion des valeurs manquantes**

* + Techniques utilisées :
    - Imputation par mode pour les variables catégorielles
    - Concernant les variables numériques, les lignes contenant des valeurs manquantes dans les colonnes critiques (notamment ‘’RainToday’’ et ‘’RainTomorrow’’) sont supprimées.
  + Objectif : Minimiser la perte d'information

**Encodage des Variables**

* + Label Encoding pour les variables catégorielles
  + Transformation binaire : 'Yes/No' → 1/0

df['RainTomorrow'] = df['RainTomorrow'].map({'Yes': 1, 'No': 0})

df['RainToday'] = df['RainToday'].map({'Yes': 1, 'No': 0})

lencoders = {}

for col in df.select\_dtypes(include=['object']).columns:

lencoders[col] = LabelEncoder()

df[col] = lencoders[col].fit\_transform(df[col])

**Standardisation des données**

* Les données numériques sont standardisées à l'aide de StandardScaler pour améliorer les performances du modèle

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

**Gestion des Valeurs Aberrantes**

* Méthode IQR (Interquartile Range)
* Calcul des seuils : Q1 - 1.5*IQR et Q3 + 1.5*IQR
* Remplacement des outliers par les valeurs seuils

def replace\_with\_thresholds(dataframe, column):

low\_limit, up\_limit = outlier\_thresholds(dataframe, column)

dataframe.loc[(dataframe[column] < low\_limit), column] = low\_limit

dataframe.loc[(dataframe[column] > up\_limit), column] = up\_limit

**► Sélection des features**

* Utilisation de variables pertinentes pour la prédiction, comme la température, l'humidité, la pression, etc. du jour actuel pour prédire ‘’RainTomorrow’’.
* On peut aussi inclure les observations météorologiques des jours précédents, car elles peuvent aider à améliorer les prévisions.

**Critères de Sélection**

* Corrélation avec la variable cible
* Importance prédictive
* Réduction de la dimensionnalité

**Features Supprimées**

* 'Date'
* 'Temp3pm'
* 'Pressure9am'
* 'Temp9am'
* 'Rainfall'

Modélisation

**► Choix du modèle :**

* Le modèle que nous avons retenu est un ‘’Random Forest Classifier’’ pour les avantages suivants :
* Gestion efficace des données multidimensionnelles (température, humidité, vent, etc.).
* Robustesse face aux valeurs aberrantes.
* Capacités de prédiction probabiliste.

**► Paramétrage du Modèle ‘’Random Forest Classifier’’**

* Hyperparamètres optimisés, permettant d’obtenir un niveau de performance acceptable, dans le cadre de notre projet MLOps :

params =

{

"n\_estimators": 10, # Nombre d'arbres

"max\_depth": 10, # Profondeur maximale des arbres

"random\_state": 42 # Reproductibilité

}

**► Stratégie de Validation**

* Split stratifié : 80% entraînement, 20% test
* Préservation de la distribution des classes

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=data['RainTomorrow'])

**► Entraînement du modèle**

Le modèle est entraîné sur les données historiques disponibles. Il apprend les relations entre les différentes caractéristiques météorologiques (température, humidité, vitesse du vent, etc.) et la cible « RainTomorrow ».

**► Stockage du modèle**

Le modèle entraîné est sauvegardé au format pickle pour être utilisé en production :

with open("../model/rfc.pkl", "wb") as file:

pickle.dump(rfc, file)

Prédictions

**► Prédiction binaire**

* Le modèle retourne une valeur de probabilité (par exemple, 0.8) qui indique la probabilité que la pluie survienne le lendemain. Cela peut être converti en une prédiction binaire.
* Si la probabilité est supérieure à un certain seuil (souvent 0.5), la prévision sera Yes (pluie).
* Si elle est inférieure, la prévision sera No (pas de pluie).

**► Seuil ajustable**

* Le seuil pour transformer la probabilité en prédiction binaire peut être ajusté selon la sensibilité désirée du modèle (plus ou moins conservateur pour prédire la pluie).

Transformation en valeur finale pour l'utilisation

**► Affichage des prévisions :** L'application renvoie à l'utilisateur une information claire et interprétable

**► Paramètres de l'application**

* Interface utilisateur (UI) : **avec streamlit** ou une application mobile
* Actualisation régulière
* Modèle dynamique : réentraînement périodique avec de nouvelles données
* Personnalisation des notifications : notification sur une app ou e-mail

**► Paramètres spécifiques du processus :**

* Modèle de machine Learning : choix d'un modèle et de ses hyperparamètres
* Seuil de décision : probabilité à partir de laquelle on détermine s'il va pleuvoir
* Métrique d'évaluation : accuracy, précision, rappel et AOC-ROC
* Fréquence de mise à jour des données : chaque jour ou plusieurs fois par jour en fonction des besoins de précision

# **VALUE PROPOSITION**

## Qui est l’utilisateur final ?

* Hôpital ou service d’un hôpital :
  + Gestionnaire d’évènement en plein air : besoin de prévoir les conditions météorologiques pour ajuster leurs événements.
  + Grand public : pour sortir avec un parapluie, planifier des voyages.

## Objectif des utilisateurs

* Prédire les conditions météorologiques du lendemain (prédiction de pluie).
* Prendre des décisions basées sur ces prévisions (transport, prévoir des vêtements appropriés, prévoir des solution de repli, …..).

## Bénéfices des utilisateurs vis-à-vis du système

* Précision accrue des prévisions : fournir des précisions plus précises que les approches traditionnelles.
* Alerte en temps réel :faciliter la prise de décision grâce à des alertes sur des pluies prévues ou pas, sur une interface adaptée.
* Personnalisation des prévisions : obtenir des prévisions specifique a une région (en sélectionnant ‘’location’’ dans les données.

## Flux de travail / Interface

**a) Flux de travail**

**► Chargement des données météo :** données météorologiques fournies dans le fichier pour diverses localisations en Australie.

**► Prétraitement des données :** nettoyage et transformation des données brutes, encodage et normalisation.

**► Sélection des caractéristiques pertinentes :** des variables comme la température, l'humidité, la pression atmosphérique, et la pluie du jour actuel sont extraites pour prédire si la pluie va survenir le lendemain

**► Prédiction via un modèle de machine learning** : le modèle de classification utilise les données prétraitées pour prédire la probabilité de pluie le lendemain ‘’**RainTomorrow**’’.

**► Résultats et interprétation** : Le système renvoie une prédiction binaire (pluie ou pas de pluie) accompagnée d'une probabilité (80% de chance qu'il pleuve)

**► Retour à l'utilisateur** : Les prévisions sont ensuite transmises à l'utilisateur via une interface (application web ou mobile)

**► Mise à jour continue des prévisions** : L'application actualise les prévisions chaque jour avec de nouvelles données météorologiques et les utilisateurs peuvent consulter les prévisions à tout moment.

**b) Interface proposée**

**► Interface utilisateur**

* Application web : **Utilisation de Streamlit.**
* Entrée : Région(location), date (automatique pour le lendemain).
* Sortie : Prévision de pluie (oui/non), probabilité de pluie (%), prévision détaillée pour d'autres paramètres météorologiques (température, humidité, etc.).

**► Notification et alertes personnalisées** : notification par e-mail ou SMS en cas de prévision de pluie (ex : il y a 70% de chance qu'il pleuve demain, veuillez ....).

**► Tableau de bord analytique**:

* Tableau de bord : pour une analyse approfondie des données passées et de prévisions futures
* Graphiques : Afficher les tendances de la météo pour plusieurs jours, avec des courbes montrant l'évolution des probabilités de pluie.
* Filtres : L'utilisateur peut filtrer par localisation ou ajuster la plage temporelle (par exemple, prévisions sur 3 jours).

**c) Exemple de flux utilisateur**

* Sélection de la région : l'utilisateur sélectionne une ville comme *Sydney* dans le menu déroulant.
* Consultation des prévisions : le système affiche *=>* "Pluie prévue demain avec une probabilité de 75 %".
* Prise de décision : L'utilisateur peut alors décider de reporter ou de maintenir une activité prévue pour le lendemain, en fonction de la prévision.
* Notification (optionnelle) : l'utilisateur peut configurer une alerte pour recevoir une notification automatique si la probabilité dépasse un certain seuil.

# **DATA COLLECTION**

## Stratégie de ‘’Train’’ initial

**a) Préparation des données**

* Nettoyage des données historiques : traiter les valeurs manquantes de certaines colonnes par correction ou imputation pour éviter les biais dans l'entraînement.
* Sélection des caractéristiques (features) : conservation des variables ayant un impact direct sur les précipitations (température, humidité, direction du vent, pression, etc.)

**b) Division du jeu de données : division effectuée de façon chronologique**

* 80 % des données pour l'entraînement initial du modèle (données anciennes)
* 20 % des données pour tester les performances du modèle (données récentes)

**c) Sélection et entraînement du modèle**

* Choix de l'algorithme : Actuellement, seul le modèle Random Forest est implémenté pour prédire la variable cible ‘’RainTomorrow’’. Les autres algorithmes (Logistic Regression, GBM, ANN, LSTM) pourraient être testés ultérieurement.
* Hyperparamétrage : effectuer une recherche d'hyperparamètres (par exemple avec une grille de recherche ou Random Search) pour trouver la configuration optimale. Ensuite, mesurer les performances avec des métriques telles que l'accuracy, le rappel, la précision, F1-Score, ROC AUC et PR AUC, pour évaluer la capacité du modèle à prédire correctement la pluie ou l'absence de pluie.

**d) Validation croisée**

Une validation croisée avec la méthode de K-fold cross-validation sur le jeu de données d'entraînement permettra d'évaluer la robustesse du modèle.

Mise à jour continue

**a) Taux de collecte des données**

* Source de données: Source de données météorologiques en temps réel : via des API comme OpenWeatherMap, BOM - Bureau of Meteorology en Australie. Les données peuvent être collectées quotidiennement ou plusieurs fois par jour (horaire).
* Fréquence de mise à jour : mise à jour quotidienne. Chaque jour, les nouvelles données collectées pour la journée seront ajoutées à l'ensemble de données.

**b) Rétention sur les entités de production :**

* Suivi des entités (villes ou localisations): la variable ‘’location’’ qui représente les différentes localisations en Australie, lesquelles localisations doivent être surveillées dans la phase de production
* Rétention des entités : conserver les données pour chaque localisation dans le système afin de suivre les tendances historiques propres à chaque entité géographique.
* Enrichissement des données: Si de nouvelles localisations apparaissent ou deviennent importantes pour les prévisions, elles devront être ajoutées dans le processus de mise à jour pour maintenir la précision du modèle sur une échelle géographique plus large.

**c) Mise à jour du modèle**

Le modèle est mis à jour à intervalles réguliers selon le schéma de pipeline suivant :

* Training Pipeline : exécution hebdomadaire (chaque lundi à 00:00)
* Prediction Pipeline : exécution quotidienne (chaque jour à 06:00)

Cette approche assure que le modèle reste performant à long terme en intégrant régulièrement les nouvelles données météorologiques.

Coûts et contraintes pour observer les résultats

**a) Coût de collecte des données**

* API météo : possibilité d'avoir des coûts associés à l'accès aux données via certaines APIs.
* Infrastructure : Stocker et traiter les données météorologiques en continu nécessite une infrastructure de serveur ou de cloud (AWS, Google Cloud, etc.), ce qui peut entraîner des coûts de stockage et de traitement.

**b) Coûts liés à la mise à jour du modèle**

* Temps de calcul : Réentraîner un modèle régulièrement peut exiger beaucoup de ressources de calcul, surtout avec de grandes quantités de données. Il est important de prévoir des coûts liés à l'infrastructure nécessaire pour maintenir et exécuter ces modèles.
* Déploiement en production : Si le modèle est déployé dans une application utilisée par de nombreux utilisateurs, il faudra également prévoir des coûts liés à l'hébergement de l'application, ainsi qu'à la maintenance et à la mise à jour du modèle en production.

**c) Contraintes d'observation des résultats**

* Temps d'observation : Pour évaluer la performance des prédictions, il faut attendre au moins 24 heures (ou plus) après l'émission des prévisions pour comparer les résultats (prédictions de pluie vs observations réelles). Cela impose un délai avant de pouvoir ajuster ou améliorer les performances du modèle.
* Contraintes de validation: Mettre en place un système de suivi des performances nécessite la collecte des résultats réels pour valider la précision du modèle.

# **DATA SOURCES**

## APIs pour les données météorologiques

Pour obtenirdes données en temps réel et historiques sur les différentes entités (villes, régions, etc.), ainsi que les résultats observés (précipitations, température, vent, etc.).

* OpenWeatherMap API : [OpenWeatherMap API](https://openweathermap.org/api)
* WeatherAPI (anciennement Weatherstack) [WeatherAPI](https://www.weatherapi.com/docs/)
* Bureau of Meteorology (BOM) – Australie [Bureau of Meteorology API](http://www.bom.gov.au/data/)
* NOAA (National Oceanic and Atmospheric Administration) [NOAA API](<https://www.ncdc.noaa.gov/cdo-web/webservices/v2>)

### Sites web pour l'exploration des données météo

* World Meteorological Organization (WMO) [WMO](<https://public.wmo.int/en>)
* Meteostat [Meteostat](<https://meteostat.net/en/>)

### Bases de données météorologiques (tables SQL ou autres formats)

* Global Historical Climatology Network (GHCN): [GHCN](<https://www.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network>)
* NOAA Climate Data Online (CDO): [NOAA CDO](<https://www.ncdc.noaa.gov/cdo-web/>)

### Méthodes pour intégrer les données dans un système de machine learning

* **Bases de données SQL** : pour structurer les données météorologiques en utilisant des bases de données relationnelles, comme **MySQL**, **PostgreSQL**, ou **SQLite.**
* **Accès via API :** en intégrant directement les **APIs météorologiques** mentionnées plus haut dans un pipeline de données, nous pouvons automatiser la collecte de données météo en temps réel. Également, des **scripts Python** peuvent être utilisés pour appeler les APIs quotidiennement et stocker les données dans une base de données ou système de fichier (par exemple, CSV, Parquet)
* **Pipelines de données** : Outils comme **Apache Airflow, ETL pipelines**

# **IMPACT SIMULATION**

## Déploiement des modèles

**Modèle pré-entraîné**

* Utiliser les données historique du fichier de base téléchargé pour entrainer initialement le modèle.
* Déployer le modèle dans une plateforme de production, où il recevra de nouvelles données météorologiques (via des APIs comme OpenWeatherMap ou BOM) pour générer des prédictions.

**Infrastructure de déploiement**

* + Plateformes cloud comme AWS SageMaker, Google Cloud AI, ou Azure ML permettent de déployer des modèles en production.
  + Via API REST personnalisée qui reçoit les données météo actuelles et renvoie une prédiction sur la base du modèle déployé.

**Interface utilisateur**

* + Intégrer le modèle dans une application web ou mobile où l'utilisateur final peut obtenir des prévisions météorologiques.

## Données d’essaies pour évaluer les performances

**Données de validation**

* Evaluation du modèle sur des données de test

**Métriques de performance**

* Précision (accuracy) : proportion de prédictions correctes.
* Rappel : prédictions vraies positives sur toutes les occurrences de pluie.
* Précision : % de prédictions correctes parmi celles prédites comme étant de la pluie.
* F1-score : Une mesure qui combine précision et rappel pour évaluer globalement les performances du modèle.
* Courbe ROC et AUC : Pour évaluer la capacité de distinction entre pluie et non-pluie
* Matrice de confusion : Pour visualiser les vrais positifs, faux positifs, vrais négatifs et faux négatifs

**Exemple de Métriques**

metrics\_rfc =

{"Accuracy": 0.85, # Précision globale

"Precision": 0.82, # Précision des prédictions positives

"Recall": 0.78, # Capture des événements réels

"F1-Score": 0.80, # Équilibre précision-recall

"ROC AUC": 0.88, # Capacité de discrimination

"PR AUC": 0.85 # Performance sur données déséquilibrées

}

## Valeurs de coût/gain pour les décisions correctes ou incorrectes

**Prédictions correctes**

* Gain lié à une bonne prédiction de pluie (vrai positif)

🡺Valeur estimée : Gain modéré à élevé selon le contexte

* Gain lié à une bonne prédiction de l’absence de pluie (vrai négatif)

🡺Valeur estimée : Gain modéré.

**Prédictions incorrectes**

* Coût lié à une fausse alerte de pluie (faux positif) : annulation d’évènement

🡺Valeur estimée: Coût faible à modéré selon le contexte

* Coût lié à une absence de prévision de pluie (faux négatif) : dommages matériels

🡺Valeur estimée : Coût potentiellement élevé

## Contrainte d’équité

**Equité géographique**

* Problème :localisations sous représentées dans les données historiques, ce qui pourrait entraîner des prévisions moins précises pour ces régions.
* Solution : S’assurer que les données couvrent de manière équitable différentes régions géographiques (urbaines, rurales, côtières, etc.).

**Equité sociale**

* Problème : Les prévisions météorologiques peuvent avoir des impacts différents selon les groupes sociaux.
* Solution : Concevoir le modèle et les stratégies d’intervention en tenant compte de ces disparités pour s'assurer que certaines communautés ne soient pas plus affectées par les erreurs de prédiction.

**Equité temporelle**

* Problème : Les prévisions pourraient être plus fiables à certaines périodes de l'année que d'autres (par exemple, saison sèche vs saison des pluies), introduisant un biais temporel
* Solution : Analyser la performance du modèle à différentes périodes de l'année et ajuster les modèles pour éviter de privilégier certaines périodes.

# **MAKING PREDICTIONS**

## Quand faisons nous des prédictions en temps réel ou pas batch ?

### Prédictions en temps réel

**Cas** **d’usage** :

* Un système automatisé d’alerte météorologiques reçues
* Un service qui alerte immédiatement sur des conditions critiques (orage, pluie imminente)

**Données** :

* Mise à jour des données en continu à partir de sources telles que des APIs météorologiques

**Fréquence** :

* Les prédictions peuvent être générées chaque minute ou chaque heure, selon la disponibilité des données et les besoins des utilisateurs

**Exemple** :

* Un modèle de prédiction qui traite les données d'une API météo chaque fois qu'une nouvelle observation est reçue et donne immédiatement la prédiction de pluie pour le lendemain.

**Contraintes** :

* Les calculs doivent être rapides, ce qui nécessite une infrastructure capable de traiter les données en quelques secondes ou minutes

### Prédictions par batches (lots)

**Cas d’usage :**

* Les applications où les prédictions n'ont pas besoin d'être mises à jour en temps réel, mais plutôt à des intervalles réguliers ( prévisions journalières, hebdomadaires, ou mensuelles)
* Utilisé lorsque des ensembles de données plus volumineux sont traités en une seule fois, permettant une analyse approfondie des tendances météorologiques

**Données :**

* Les données météorologiques historiques ou agrégées sont utilisées pour prédire les conditions futures.

**Fréquence :**

* Les prédictions sont faites à des intervalles prédéfinis, par exemple, une fois par jour, une fois par semaine, ou selon un programme planifié

**Exemple :**

* Un système de prévision météo pour les agriculteurs qui génère des prédictions hebdomadaires à partir de données météorologiques collectées au cours des sept derniers jours.

**Contraintes :**

* Moins de contraintes sur la rapidité de traitement par rapport aux systèmes en temps réel, mais il faut tout de même maintenir un bon équilibre entre les performances du modèle et le temps de calcul disponible

## Temps disponible pour le feature engineering, la prédiction et le post-traitement

### Prédictions en temps réel

**Feature Engineering** :

* La transformation des données en caractéristiques utiles (features) doit être rapide. Les données reçues via des API météo doivent être prétraitées instantanément.

**🡺Temps estimé** : Quelques millisecondes à secondes

**Prédiction**:

* L'algorithme de machine learning doit être capable de générer une prédiction en quelques millisecondes à secondes.

**🡺temps estmé**: Millisecondes à secondes

**Post-traitement** :

* Cela inclut l'interprétation des résultats, la génération de visualisation ou l'envoi d'alertes et se faire rapidement

**🡺Temps estimé**: Millisecondes à secondes

### Prédictions par batches

**Feature Engineering :**

* Les données historiques peuvent être agrégées et traitées en amont, donc le processus peut être plus lent et plus complexe

**🡺Temps estimé :** Quelques minutes à heures selon la taille du batch et la complexité du pipeline de données

**Prédiction**:

* le modèle est appliqué sur un ensemble de données plus important et peut être exécuté en parallèle ou distribué sur plusieurs machines

**🡺Temps estimé :** Minutes à heures selon la taille des données et le modèle

**Post-traitement :**

* Les résultats doivent être agrégés, analysés et interprétés pour des rapports plus approfondis. Des visualisations ou des rapports peuvent être générés automatiquement.

**🡺Temps estimé :** Minutes à heures (génération de rapports, stockage des résultats)

## Objectif de calcul

Il dépend des contraintes de l’application et des ressources disponibles.

### Prédictions en temps réel

**Objectif principale** :

* Faible latence et haute disponibilité
* Le système doit être capable de traiter les requêtes en quelques secondes pour permettre à l'utilisateur final de recevoir des prédictions immédiatement

**Infrastructures** :

* Le système doit être capable de traiter les requêtes en quelques secondes pour permettre à l'utilisateur final (ou à un système automatisé) de recevoir des prédictions immédiatement
* Optimisation du code pour réduire la charge de calcul

### Prédictions par batches

**Objectif principal** :

* Traitement de grandes quantités de données pour obtenir des résultats fiables et précis sur une période donnée
* Pas de contraintes strictes sur la latence, mais l'efficacité du calcul est importante pour réduire les coûts et éviter les goulots d'étranglement dans le traitement des données

**Infrastructures** :

* Utiliser des solutions comme Apache Spark pour le traitement distribué des données, ou des instances cloud avec des capacités de calcul parallélisées
* Les batchs peuvent être traités la nuit ou pendant les périodes de faible activité

**Prédictions générées via une API RESTful construite avec FastAPI.**

Détails sur le fonctionnement de l'API et l'orchestration des prédictions.

**Types de prédictions :**

* + Prédiction automatique : basée sur les données du jour via une requête GET à **l'endpoint /predict.**
  + Prédiction manuelle : via une requête POST avec un fichier CSV contenant les données. Cette fonctionnalité n'est pas visible dans le code fourni et doit être vérifiée.

**Orchestration :** Utilisation de MLflow pour le suivi des runs de prédiction, imbriqués dans un run de déploiement de modèle parent.

**Logique de l'API :**

* + Récupération du run de déploiement actuel (et configuration de MLflow).
  + Démarrage d'un run MLflow imbriqué pour la prédiction spécifique.
  + Exécution de la fonction predict\_weather() utilisant le modèle MLflow.
  + Retour des résultats (prédiction, probabilité, identifiants de run MLflow).

**Exemple de réponse (GET /predict):**

{

"status": "success",

"message": "Daily prediction successfully completed",

"run\_id": "...",

"deployment\_run\_id": "...",

"model\_version": "...",

"prediction": "No",

"probability": 0.15

}

**Orchestration des prédictions via Airflow *(Cf script prediction\_dag.py)* :**

* **Orchestration des prédictions** : Le fichier prediction\_dag.py met en place un DAG (Directed Acyclic Graph) Airflow pour automatiser le processus de prédiction.
* **Étapes principales du DAG** :
  + Extraction des données brutes depuis une source définie.
  + Préparation des données pour le modèle (nettoyage, transformations).
  + Chargement du modèle entraîné depuis un chemin spécifié.
  + Génération des prédictions et stockage des résultats.
* **Exemple de définition d'un DAG dans le fichier :**

with DAG(

dag\_id="prediction\_pipeline",

schedule\_interval="@daily",

start\_date=datetime(2023, 1, 1),

catchup=False,

) as dag:

* **Planification** : Le DAG est configuré pour s'exécuter quotidiennement (@daily), garantissant une mise à jour régulière des prédictions.

# **API DETAILS & MLFLOW INTEGRATION**

Descriptif

* Utilisation de MLflow pour le monitoring et le suivi des performances (via predict\_api.py)
* Logging (via predict\_api.py)
* Variables d’environnement (via predict\_api.py)
* Exposition d’une API

## Framework API - FastAPI

**Endpoints :**

* + **/predict (GET)** : Effectue une prédiction automatique basée sur les données actuelles.

**MLflow :**

* **Suivi des runs** : Chaque requête de prédiction crée un run MLflow imbriqué dans le run de déploiement du modèle.
* **Tags MLflow** : Utilisation de tags pour identifier le type de pipeline, la date de prédiction, l'endpoint utilisé et la version du modèle.
* **Gestion des runs** : Utilisation de mlflow.start\_run() et mlflow.end\_run() pour gérer les runs MLflow et assurer un suivi correct.
* **Fonction get\_deployment\_run()** issue de utils.mlflow\_run\_manager : Récupère les informations du run de déploiement du modèle courant depuis MLflow.

**Gestion des erreurs :**

* L'API capture les exceptions et retourne des erreurs HTTP 500 avec des détails.

# **PIPELINE DE PREDICTIONS**

Cette section détaille précisément comment les prédictions sont orchestrées ainsi que la structure et les composants du pipeline **(Cf script prediction\_dag.py).**

Le pipeline de prédiction est orchestré via **Airflow** et suit les étapes suivantes :

## Extraction des données :

* + Source : Les données brutes sont extraites depuis un chemin ou une base de données spécifiée.
  + Exemple dans le script :

extract\_data = PythonOperator(

task\_id="extract\_data",

python\_callable=extract\_data\_function,

)

Préparation des données :

* + Nettoyage et transformation des données pour les rendre exploitables par le modèle.
  + Exemple dans le script :

preprocess\_data = PythonOperator(

task\_id="preprocess\_data", python\_callable=preprocess\_function,

)

Chargement du modèle :

* + Le modèle est chargé depuis un chemin défini (par exemple, un répertoire local ou un stockage distant).
  + Exemple dans le script :

load\_model = PythonOperator(

task\_id="load\_model",

python\_callable=load\_model\_function,

)

Génération des prédictions :

* + Les prédictions sont effectuées sur les données préparées.
  + Exemple dans le script :

predict = PythonOperator(

task\_id="predict",

python\_callable=predict\_function,

)

Stockage des résultats :

* Les résultats sont sauvegardés dans un emplacement spécifié (base de données ou fichier).
* Exemple dans le script : save\_predictions = PythonOperator(

task\_id="save\_predictions",

python\_callable=save\_predictions\_function,

)

Dépendances entre tâches :

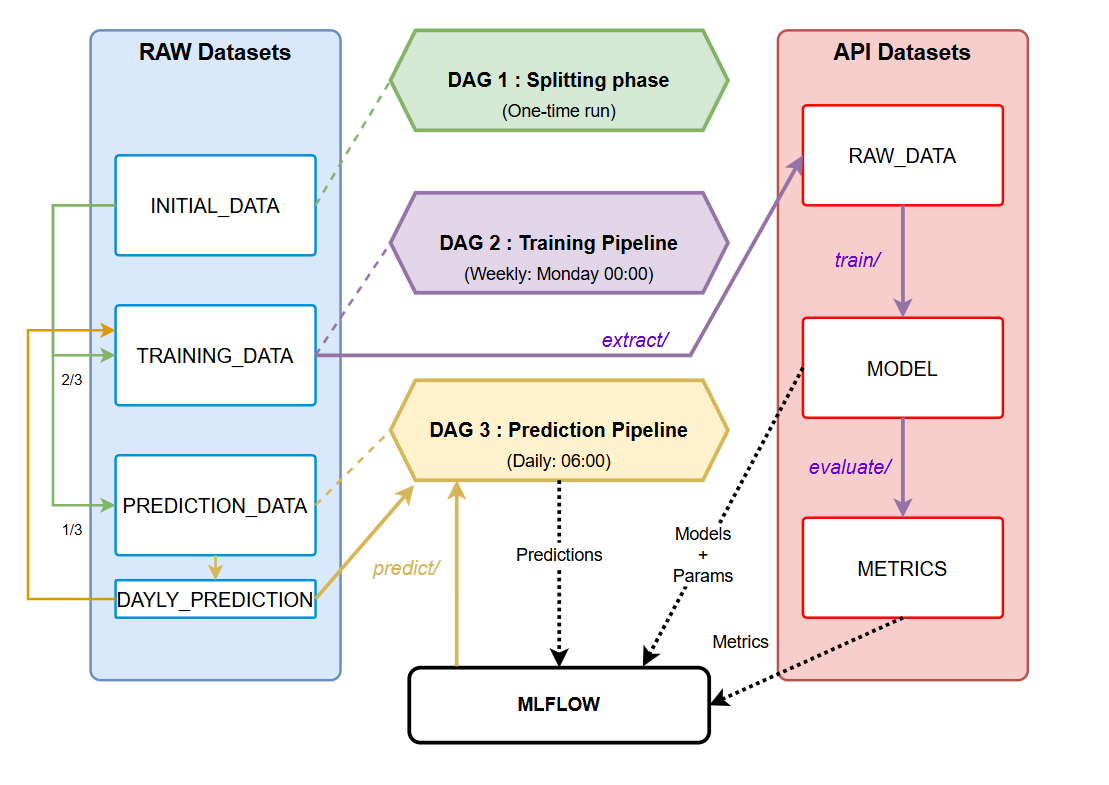
* + Les tâches sont exécutées séquentiellement selon leur dépendance logique.
  + Exemple dans le script :

extract\_data >> preprocess\_data >> load\_model >> predict >> save\_predictions

## Orchestration avec Airflow

### Vue d'ensemble de l'architecture d'orchestration

Apache Airflow joue un rôle central en orchestrant trois workflows principaux (DAGs) qui nous permettent de simuler un scénario réel de prédiction quotidienne de pluie. Cette architecture répond au défi de travailler avec un jeu de données statique tout en simulant des prédictions journalières.

****

**Diagramme : Chainage des DAGs**

### Fonctionnement des DAGs

#### DAG 1 : Préparation des données

Ce DAG initial divise notre unique dataset en deux parties distinctes :

* Un jeu de données d'entraînement (2/3 du dataset)
* Un réservoir de lignes pour les prédictions quotidiennes (1/3 restant)

Cette séparation est essentielle pour simuler l'arrivée quotidienne de nouvelles données météorologiques, comme dans un scénario réel où nous recevrions des données journalières d'une API météo.

#### DAG 2 : Entraînement hebdomadaire du modèle

Ce DAG s'exécute chaque semaine pour entraîner notre modèle de prédiction de pluie :

with DAG(

'2\_weather\_training\_dag',

description='Weekly weather model training pipeline',

schedule\_interval='0 0 \* \* MON', # Exécution tous les lundis

...

) as dag:

Un aspect technique crucial de ce DAG est la persistance du run MLflow à travers différents appels API. Pour résoudre ce problème :

* Le DAG démarre un workflow MLflow via l'endpoint /workflow/start
* L'ID du run est stocké dans XCom, puis transmis à chaque tâche subséquente
* Les fonctions utilitaires comme start\_mlflow\_workflow gèrent cette continuité

#### DAG 3 : Prédictions quotidiennes

Ce DAG s'exécute quotidiennement pour simuler les prédictions météorologiques du lendemain :

with DAG(

'3\_weather\_prediction\_dag',

description='Daily weather prediction pipeline',

schedule\_interval='0 6 \* \* \*', # Exécution quotidienne à 6h du matin

...

) as dag:

Son fonctionnement est particulièrement ingénieux :

* Il extrait une ligne du dataset de prédiction via process\_daily\_prediction\_row
* Il effectue une prédiction sur cette ligne grâce au modèle entraîné
* Il ajoute cette ligne (avec son label réel) au dataset d'entraînement

Ainsi, chaque semaine, le modèle est réentraîné avec 7 nouvelles observations, simulant parfaitement un scénario réel d'enrichissement continu des données.

### Avantages d'Airflow pour ce projet

1. **Gestion des dépendances** : Les FileSensors garantissent que les fichiers nécessaires existent avant l'exécution des tâches :

python

check\_raw\_file = FileSensor(

task\_id='check\_raw\_data\_file',

filepath=str(TRAINING\_RAW\_DATA\_PATH / 'weatherAUS\_training.csv'),

...

)

1. **Communication entre DAGs** : Le TriggerDagRunOperator permet au DAG d'entraînement de déclencher le DAG de prédiction :

python

trigger\_prediction = TriggerDagRunOperator(

task\_id='trigger\_prediction\_dag',

trigger\_dag\_id='3\_weather\_prediction\_dag',

...

)

1. **Persistance des données entre tâches** : XCom facilite le partage d'informations comme le run\_id MLflow :

python

workflow\_response = ti.xcom\_pull(task\_ids='start\_mlflow\_workflow')

### Transmission du run\_id MLflow aux endpoints API

Une caractéristique essentielle de notre solution est la fonction make\_api\_request qui assure la continuité du suivi MLflow à travers les différentes étapes du pipeline :

python

def make\_api\_request(endpoint, context, timeout=300):

"""Generic function to make API requests"""

try:

*# Get run\_id from XCom*

ti = context.get('ti')

run\_id = None

params = {}

if endpoint != 'predict':

if ti:

workflow\_response = ti.xcom\_pull(task\_ids='start\_mlflow\_workflow')

*# Extract run\_id from the response if it's a dictionary*

if isinstance(workflow\_response, dict):

run\_id = workflow\_response.get('run\_id')

else:

*# Fall back to using the value directly if it's not a dict*

run\_id = workflow\_response

*# Add run\_id to parameters only if it exists*

if run\_id:

params["run\_id"] = run\_id

*# Make the request*

response = requests.get(f"{API\_URL}/{endpoint}", params=params, timeout=timeout)

Cette fonction permet de :

* Récupérer le run\_id depuis XCom où il a été précédemment stocké
* L'ajouter comme paramètre de requête pour les endpoints API (sauf pour 'predict')
* Maintenir ainsi la cohérence du suivi d'expérience à travers tout le workflow

### Vérification de l'intégrité des données avec hachage

Le DAG de prédiction intègre également un mécanisme sophistiqué pour vérifier que les données journalières soit bien actualisées avant de procéder a la prédiction :

python

*# Calculate hash of current daily prediction file (if exists)*

calculate\_hash\_before = PythonOperator(

task\_id='calculate\_current\_hash',

python\_callable=calculate\_current\_prediction\_hash,

provide\_context=True

)

*# Verify the daily prediction data has changed*

verify\_data\_changed = PythonOperator(

task\_id='verify\_prediction\_data\_changed',

python\_callable=verify\_prediction\_data\_changed,

provide\_context=True

)

Ce mécanisme fonctionne en deux phases :

1. **Avant traitement** : calculate\_current\_prediction\_hash calcule une empreinte MD5 du fichier de prédiction journalier existant
2. **Après traitement** : verify\_prediction\_data\_changed vérifie que l'empreinte a changé, garantissant que de nouvelles données sont utilisées

Cette approche ingénieuse empêche l'utilisation répétée des mêmes données en cas d'échec ou de redémarrage du DAG, ce qui pourrait perturber la séquence temporelle des prédictions.

### Conclusion

Airflow s'avère un outil exceptionnellement adapté à ce projet de prédiction météorologique. Sa nature programmable en Python, sa capacité à gérer des DAGs complexes et son système de communication XCom ont permis d'implémenter une simulation réaliste de prédictions météorologiques quotidiennes.

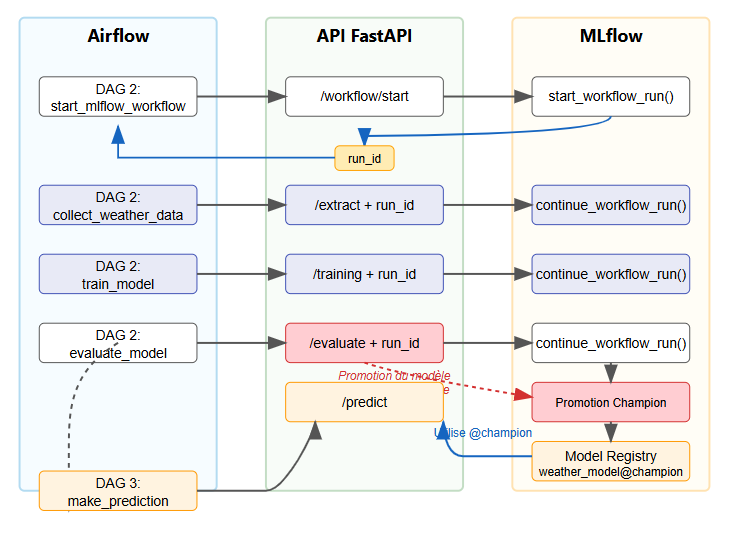
La solution est particulièrement élégante car elle transforme un dataset statique en un flux simulé de données journalières, permettant à la fois des prédictions quotidiennes et un réentraînement hebdomadaire du modèle avec des données enrichies.

De plus, l'intégration avec MLflow et les mécanismes avancés comme la transmission du run\_id et la vérification par hachage démontrent la flexibilité d'Airflow à mettre en œuvre des workflows robustes et sécurisés. Sa compatibilité native avec Docker et différents opérateurs de bases de données en fait une solution complète pour l'orchestration de workflows de machine learning.

## Intégration de MLflow

### Vue d'ensemble de l'architecture MLflow

Notre projet utilise MLflow comme composant central pour le suivi d'expériences, la gestion des versions de modèles et le déploiement. Voici comment nous avons conçu cette intégration pour assurer une traçabilité complète et une gestion efficace du cycle de vie des modèles.

**Diagramme : Cycle de gestion MLflow**

### Points clés de l'intégration MLflow

#### 1. Gestion des workflows MLflow à travers une API FastAPI

Un aspect innovant de notre architecture est la gestion des runs MLflow via des endpoints API dédiés :

python

@router.api\_route("/workflow/start", methods=["GET", "POST"])

async def start\_workflow():

"""Starts a new MLflow workflow and returns the run\_id."""

run\_id = start\_workflow\_run("weather\_prediction\_workflow")

return {

"status": "success",

"run\_id": run\_id,

"next\_steps": [

{"endpoint": "/extract", "method": "GET", "params": {"run\_id": run\_id}},

*# Autres étapes...*

]

}

Cette approche offre plusieurs avantages :

* Découplage entre l'orchestration Airflow et le tracking MLflow
* Possibilité d'appeler les étapes indépendamment tout en maintenant le tracking
* Centralisation de la logique MLflow dans une API réutilisable

#### 2. Persistance des runs MLflow à travers les étapes du pipeline

Le défi majeur était de maintenir la continuité des runs MLflow à travers différentes étapes du pipeline exécutées séparément. Notre solution utilise un mécanisme de continuation de run :

python

def continue\_workflow\_run(run\_id, step\_name):

"""Continues an existing workflow run for a specific step."""

*# Configuration préalable...*

*# Terminer tout run actif pour éviter les runs imbriqués involontaires*

active\_run = mlflow.active\_run()

if active\_run and active\_run.info.run\_id != run\_id:

mlflow.end\_run()

*# Reprendre le run existant*

run = mlflow.start\_run(run\_id=run\_id, nested=False)

*# Marquer l'étape courante avec des tags*

mlflow.set\_tag(f"step\_{step\_name}\_start\_time", datetime.now().strftime("%Y%m%d\_%H%M%S"))

mlflow.set\_tag("current\_step", step\_name)

return run

### Serveur de Tracking MLflow

Le serveur de tracking constitue la mémoire de nos expériences ML. Il est responsable de :

* L'enregistrement des paramètres d'entraînement (mlflow.log\_params)
* Le suivi des métriques de performance (mlflow.log\_metric)
* La conservation des métadonnées via des tags (mlflow.set\_tag)
* L'archivage des artefacts comme les fichiers de métriques (mlflow.log\_artifact)

Dans notre architecture, chaque run MLflow représente une séquence complète d'opérations, depuis la préparation des données jusqu'à l'évaluation, avec des tags comme current\_step qui indiquent l'étape en cours:

python

*# Dans la fonction d'entraînement*

mlflow.set\_tag("current\_step", "model\_training")

*# Plus tard, dans la fonction d'évaluation*

mlflow.set\_tag("current\_step", "evaluation")

Nous utilisons également le tracking pour documenter la transformation des données et les performances détaillées:

python

*# Logging des métriques essentielles d'entraînement*

mlflow.log\_metric("train\_accuracy", train\_accuracy)

mlflow.log\_metric("train\_precision", train\_precision)

mlflow.log\_metric("train\_f1", train\_f1)

### Configuration centralisée

La cohérence de la configuration MLflow est essentielle pour assurer l'intégrité des données de suivi. Notre fonction setup\_mlflow() centralise cette configuration et est appelée au début de chaque opération MLflow :

python

def setup\_mlflow():

"""Configure MLflow tracking settings"""

*# Configuration de l'URI du tracking server*

mlflow.set\_tracking\_uri(MLFLOW\_TRACKING\_URI)

*# Création de l'expérience si elle n'existe pas*

experiment = mlflow.get\_experiment\_by\_name(DEFAULT\_EXPERIMENT\_NAME)

if experiment is None:

mlflow.create\_experiment(DEFAULT\_EXPERIMENT\_NAME)

*# Définition de l'expérience active*

mlflow.set\_experiment(DEFAULT\_EXPERIMENT\_NAME)

Cette centralisation permet de :

* Garantir que tous les composants utilisent les mêmes paramètres MLflow
* Faciliter les changements de configuration (par exemple, passer d'un serveur MLflow local à un serveur distant)
* Assurer que les expériences et runs sont correctement organisés

### Registre de Modèles MLflow

Alors que le tracking se concentre sur l'historique des expériences, le registre de modèles gère le cycle de vie des modèles destinés à la production. Dans notre projet, nous utilisons le registre pour :

* Stocker les versions successives de nos modèles
* Gérer la promotion des modèles en production via l'alias "champion"
* Faciliter le déploiement des modèles pour l'inférence

Pendant l'enregistrement du modèle, nous utilisons log\_model avec l'option registered\_model\_name :

model\_info = mlflow.sklearn.log\_model(

sk\_model=rfc,

artifact\_path="model",

signature=signature,

input\_example=X\_train\_scaled[:5],

registered\_model\_name=MODEL\_NAME,

metadata={"performance\_summary": performance\_summary}

)

### Mécanisme de Promotion du Modèle Champion

Un aspect prépondérant de notre architecture est le mécanisme automatique de promotion du "champion", implémenté dans la fonction evaluate\_model(). Voici son fonctionnement détaillé :

1. **Évaluation du modèle récemment entraîné** :

python

*# Predictions*

y\_test\_pred = model.predict(X\_test\_scaled)

*# Calculate and log accuracy score for evaluation*

test\_accuracy = metrics.accuracy\_score(y\_test, y\_test\_pred)

mlflow.log\_metric("eval\_accuracy", test\_accuracy)

1. **Tentative d'identification du champion actuel** :

python

client = mlflow.tracking.MlflowClient()

*# Get the current model version*

all\_versions = client.search\_model\_versions(f"name='{MODEL\_NAME}'")

latest\_version = sorted(all\_versions, key=lambda x: int(x.version), reverse=True)[0]

current\_version = latest\_version.version

current\_performance = test\_accuracy *# Current model's accuracy score*

*# Try to find if there's a model with the "champion" alias*

try:

champion\_version = client.get\_model\_version\_by\_alias(MODEL\_NAME, "champion")

champion\_run = client.get\_run(champion\_version.run\_id)

1. **Comparaison des performances** :

python

*# Get champion model accuracy score (check both possible metric names)*

if "eval\_accuracy" in champion\_run.data.metrics:

champion\_performance = champion\_run.data.metrics["eval\_accuracy"]

elif "test\_accuracy" in champion\_run.data.metrics:

champion\_performance = champion\_run.data.metrics["test\_accuracy"]

else:

logger.warning("Champion model has no accuracy score metric")

champion\_performance = 0

1. **Décision de promotion basée sur l'accuracy** :

python

*# Promote if accuracy is greater than or equal to current champion model*

if current\_performance >= champion\_performance:

*# Set the current model version as the new "champion"*

client.set\_registered\_model\_alias(MODEL\_NAME, "champion", current\_version)

mlflow.set\_tag("model\_promotion", f"New model promoted to champion (Accuracy: {current\_performance:.4f} vs {champion\_performance:.4f})")

logger.info(f"Model version {current\_version} promoted to champion with improved accuracy")

else:

*# No promotion needed*

mlflow.set\_tag("model\_promotion", "Not promoted (no accuracy improvement)")

logger.info(f"Model version {current\_version} not promoted (Accuracy: {current\_performance:.4f} vs {champion\_performance:.4f})")

1. **Gestion du cas particulier du premier modèle** :

python

except Exception as e:

*# No champion model exists yet - always promote the first model*

client.set\_registered\_model\_alias(MODEL\_NAME, "champion", current\_version)

mlflow.set\_tag("model\_promotion", "First model assigned as champion")

logger.info(f"Model version {current\_version} assigned as first champion")

Ce mécanisme garantit que seuls les modèles présentant des performances égales ou supérieures deviennent le nouveau "champion", assurant ainsi que la qualité de nos prédictions s'améliore progressivement ou reste stable au fil du temps.

### Avantages de l'Approche par Alias

Notre choix d'utiliser un alias "champion" plutôt que les stages MLflow (désormais dépréciés) offre plusieurs avantages :

* **Flexibilité accrue** : Possibilité de créer des alias personnalisés adaptés à notre workflow ("champion", "challenger", etc.)
* **Déploiement transparent** : Le code d'inférence peut toujours référencer le même URI models:/weather\_prediction\_model@champion indépendamment de la version spécifique
* **Rollback facilité** : En cas de problème, il suffit de réassigner l'alias à une version précédente

Cette approche est utilisée dans notre fonction de prédiction :

python

model\_uri = f"models:/{model\_name}@champion"

model = mlflow.sklearn.load\_model(model\_uri)

### Runs de déploiement et traçabilité

Pour maintenir une traçabilité complète, nous créons des "runs de déploiement" spécifiques chaque fois qu'un modèle est déployé pour l'inférence. Cette approche nous permet de documenter précisément :

* Quelle version du modèle est utilisée à quel moment
* Les performances observées en production pour chaque version
* L'historique complet des déploiements

python

*# Création d'un run de déploiement*

with mlflow.start\_run(run\_name=f"model\_deployment\_v{model\_version}\_{timestamp}") as run:

mlflow.set\_tag("model\_deployment", "True")

mlflow.set\_tag("model\_version", str(model\_version))

mlflow.set\_tag("deployment\_start\_date", timestamp)

*# Autres informations de contexte...*

### Bénéfices et leçons apprises

Notre architecture MLflow nous a apporté de nombreux avantages :

**Traçabilité sans faille** :

Chaque étape du processus est documentée dans MLflow, facilitant l'audit et le débogage. Lorsqu'un problème survient, nous pouvons facilement remonter la chaîne de causalité jusqu'à sa source.

**Flexibilité opérationnelle** :

Le découplage entre Airflow et MLflow nous permet de faire évoluer indépendamment ces deux composants critiques.

**Déploiement simplifié** :

L'utilisation d'alias comme "champion" a considérablement simplifié nos procédures de déploiement et de rollback.

**Documentation vivante** :

MLflow devient une "documentation vivante" de notre activité ML, réduisant le besoin de documentation externe et assurant que les informations restent synchronisées avec la réalité.

**Facilité de comparaison** :

La structure cohérente des runs MLflow facilite la comparaison entre différentes versions des modèles et l'analyse de leurs performances relatives.

### Conclusion

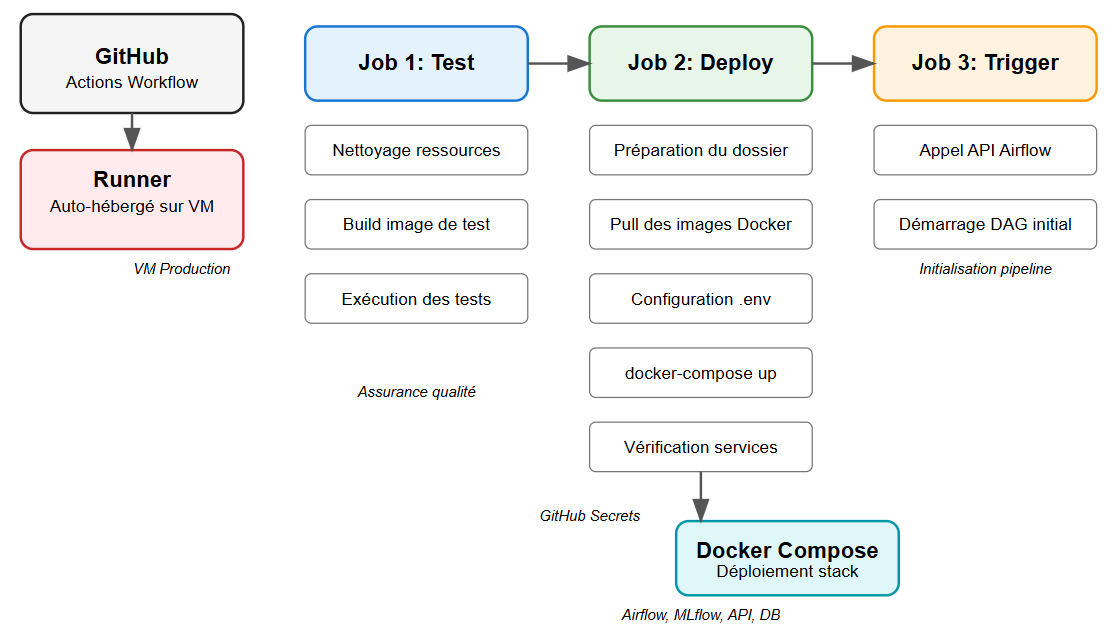
L'intégration de MLflow dans notre projet de prédiction météorologique illustre parfaitement comment un outil de MLOps bien intégré peut transformer un simple pipeline de machine learning en un système robuste et évolutif.

En exploitant intelligemment les fonctionnalités de MLflow comme les alias, les tags et la structuration des runs, nous avons pu créer une solution qui non seulement fonctionne aujourd'hui, mais qui est également préparée pour évoluer avec nos besoins futurs.

La leçon principale que nous avons tirée de cette implémentation est l'importance d'une conception réfléchie de l'architecture de tracking dès le début du projet. L'investissement initial dans cette architecture a été largement rentabilisé par la facilité de gestion et d'évolution du système.

## Pipeline CI/CD

Notre projet de prédiction météorologique utilise un pipeline CI/CD robuste implémenté via GitHub Actions pour automatiser le déploiement et assurer la qualité du code. Cette section détaille l'architecture et les choix techniques de notre pipeline.



**Diagramme : Pipeline CI/CD**

### Architecture globale du pipeline CI/CD

Le workflow GitHub Actions est structuré en trois jobs principaux qui s'exécutent séquentiellement :

**Test** : Exécution des tests unitaires

**Deploy** : Déploiement de l'application

**Trigger-initial-pipeline** : Initialisation des données et démarrage du pipeline de prédiction

Le workflow se déclenche automatiquement lors des push sur les branches main et master, des pull requests, ou peut être lancé manuellement avec des paramètres spécifiques via workflow\_dispatch.

yaml

on:

push:

branches: [ main, master ]

pull\_request:

branches: [ main, master ]

workflow\_dispatch:

inputs:

environment:

description: 'Environment to deploy to'

required: true

default: 'production'

type: choice

options:

- development

- staging

- production

### Job de test : Assurance qualité du code

Le premier job exécute les tests unitaires dans un conteneur Docker dédié. Cette approche garantit que les tests s'exécutent dans un environnement isolé et reproductible.

yaml

test:

name: Run Unit Tests

runs-on: [self-hosted, weather-app]

steps:

- name: Clean up Docker resources

run: |

# Display initial disk space

echo "Initial disk space:"

df -h

*# Stop and remove all containers*

echo "Stopping and removing containers..."

docker stop $(docker ps -a -q) || true

docker rm $(docker ps -a -q) || true

*# Remove dangling images (untagged images)*

echo "Removing dangling images..."

docker image prune -f

*# Full system prune for everything else*

echo "Performing system prune..."

docker system prune -af --volumes

- name: Build test Docker image

run: |

docker build -t weather-app-tests -f app/api/Dockerfile.test .

- name: Run unit tests

run: |

echo "Running unit tests..."

docker run --rm weather-app-tests

TEST\_EXIT\_CODE=$?

if [ $TEST\_EXIT\_CODE -ne 0 ]; then

echo "Tests failed. Exiting with error code $TEST\_EXIT\_CODE"

exit $TEST\_EXIT\_CODE

else

echo "All tests passed successfully"

fi

L'utilisation d'un Dockerfile spécialisé pour les tests (Dockerfile.test) nous permet de :

* Isoler l'environnement de test
* Définir précisément les dépendances nécessaires
* S'assurer que les tests s'exécutent de manière cohérente sur n'importe quel environnement

Le nettoyage initial des ressources Docker est essentiel pour éviter les problèmes d'espace disque et d'interférence entre les exécutions de workflow, particulièrement important sur notre runner auto-hébergé à ressources limitées.

### Job de déploiement : La pierre angulaire du pipeline

Le job de déploiement constitue l'élément central de notre pipeline CI/CD, orchestrant le déploiement complet de notre stack applicative.

### Configuration du Job de Déploiement

yaml

deploy:

name: Deploy Weather App

needs: test

runs-on: [self-hosted, weather-app]

environment: ${{ github.event.inputs.environment || 'production' }}

Cette section initiale définit les paramètres fondamentaux du job :

* name: Deploy Weather App : Identifie clairement la finalité du job
* needs: test : Établit une dépendance cruciale — le déploiement ne s'exécutera que si les tests ont réussi
* runs-on: [self-hosted, weather-app] : Spécifie que le job s'exécute sur notre runner auto-hébergé
* environment : Détermine dynamiquement l'environnement de déploiement, avec "production" comme valeur par défaut

### Préparation du déploiement

yaml

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.9'

- name: Create deployment directory

run: |

# Create new empty directory with proper permissions

mkdir -p ~/weather-app-deployment

sudo chown ubuntu:ubuntu ~/weather-app-deployment

sudo chmod 775 ~/weather-app-deployment

- name: Copy deployment files

run: |

# Copy only the app directory with sudo to avoid permission issues

sudo cp -r ./app/\* ~/weather-app-deployment/

*# Set proper ownership for the entire deployment directory*

sudo chown -R ubuntu:ubuntu ~/weather-app-deployment

*# Ensure directories have execute permissions*

sudo find ~/weather-app-deployment -type d -exec chmod 755 {} \;

*# Ensure files have read/write permissions*

sudo find ~/weather-app-deployment -type f -exec chmod 644 {} \;

Cette étape établit le dossier de déploiement avec les permissions appropriées, une précaution essentielle pour éviter les problèmes d'accès aux fichiers lorsque les conteneurs s'exécutent avec des utilisateurs spécifiques.

### Récupération des Images Docker

yaml

- name: Pull Docker images

run: |

# Pull all required Docker images

docker pull postgres:13

docker pull redis:latest

docker pull ghcr.io/mlflow/mlflow:v2.21.0rc0

docker pull ${AIRFLOW\_IMAGE\_NAME}

Cette étape effectue un pré-téléchargement stratégique de toutes les images Docker nécessaires :

* **Base de données PostgreSQL** : Stockage persistant pour les données
* **Redis** : Utilisé par Airflow comme broker de messages
* **MLflow** : Notre plateforme de suivi d'expériences ML
* **Airflow** : Orchestrateur de nos workflows de données

Le pré-téléchargement réduit considérablement le temps de démarrage lors de l'exécution du docker-compose up et garantit que les versions spécifiques sont disponibles localement.

### Configuration de l'Environnement d'Exécution

yaml

- name: Configure environment

run: |

cd ~/weather-app-deployment

*# Create .env file from secrets*

cat > .env << EOF

AIRFLOW\_UID=${AIRFLOW\_UID}

AIRFLOW\_GID=${AIRFLOW\_GID}

\_AIRFLOW\_WWW\_USER\_USERNAME=${\_AIRFLOW\_WWW\_USER\_USERNAME}

\_AIRFLOW\_WWW\_USER\_PASSWORD=${\_AIRFLOW\_WWW\_USER\_PASSWORD}

DB\_NAME=${DB\_NAME}

DB\_USER=${DB\_USER}

DB\_PASSWORD=${DB\_PASSWORD}

DB\_SECRET\_KEY=${DB\_SECRET\_KEY}

EOF

Cette étape critique génère dynamiquement le fichier .env qui configure tous nos services. Les valeurs proviennent des secrets GitHub, garantissant que les informations sensibles ne sont jamais exposées dans le code ou les logs.

L'utilisation d'un fichier .env présente plusieurs avantages :

* **Sécurité renforcée** : Les secrets sont injectés directement dans le fichier de configuration
* **Modularité** : Permet de maintenir des configurations différentes pour chaque environnement
* **Compatibilité avec Docker Compose** : Le fichier est automatiquement chargé par Docker Compose

### Démarrage des Services

yaml

- name: Start services

run: |

cd ~/weather-app-deployment

docker-compose down

docker-compose up --build -d

Cette étape finale déclenche le déploiement effectif de l'application :

* docker-compose down : Arrête proprement tous les services existants
* docker-compose up --build -d :
  + --build : Force la reconstruction des images locales
  + -d : Exécute les conteneurs en mode détaché

### Initialisation et Vérification

yaml

- name: Initialize data directories

run: |

cd ~/weather-app-deployment

*# Make sure directories have correct permissions*

sudo chown -R ${AIRFLOW\_UID}:${AIRFLOW\_GID} airflow

sudo chown -R ${AIRFLOW\_UID}:${AIRFLOW\_GID} raw\_data

sudo chown -R ${AIRFLOW\_UID}:${AIRFLOW\_GID} api

- name: Verify deployment

run: |

cd ~/weather-app-deployment

*# Define function to check service readiness*

check\_service\_ready() {

local service\_name=$1

local url=$2

local max\_attempts=$3

local wait\_seconds=$4

echo "Checking if $service\_name is ready..."

for ((i=1; i<=$max\_attempts; i++)); do

if curl -s "$url" > /dev/null; then

echo "$service\_name is ready!"

return 0

fi

echo "Attempt $i/$max\_attempts: $service\_name not ready yet, waiting ${wait\_seconds}s..."

sleep $wait\_seconds

done

echo "$service\_name failed to start after $(($max\_attempts \* $wait\_seconds)) seconds"

return 1

}

*# Maximum 30 attempts with 10 seconds wait = up to 5 minutes per service*

check\_service\_ready "Airflow" "http://localhost:8080/health" 30 10 || exit 1

check\_service\_ready "MLflow" "http://localhost:5000/api/2.0/mlflow/experiments/list" 30 10 || exit 1

check\_service\_ready "Weather API" "http://localhost:8000/docs" 30 10 || exit 1

check\_service\_ready "Streamlit" "http://localhost:8501" 30 10 || exit 1

Cette section établit correctement les permissions des répertoires et vérifie que tous les services sont opérationnels avant de poursuivre, assurant un déploiement fiable.

### Déclenchement du pipeline initial

Une fois le déploiement terminé, le dernier job déclenche le pipeline de préparation des données :

yaml

trigger-initial-pipeline:

needs: deploy

runs-on: [self-hosted, weather-app]

steps:

- name: Trigger initial data split and training

run: |

# Trigger the data loading DAG if it exists

curl -X POST http://localhost:8080/api/v1/dags/1\_weather\_initial\_split\_dag/dagRuns \

-H "Content-Type: application/json" \

-u "${\_AIRFLOW\_WWW\_USER\_USERNAME}:${\_AIRFLOW\_WWW\_USER\_PASSWORD}" \

-d '{"conf": {}}'

echo "Initial pipeline triggered"

Cette étape utilise l'API REST d'Airflow pour déclencher programmatiquement le DAG qui préparera notre dataset initial pour l'entraînement et la prédiction.

## Pourquoi Docker Compose pour le déploiement ?

Nous avons choisi d'utiliser Docker Compose pour le déploiement de notre application pour plusieurs raisons stratégiques :

### Avantages du déploiement avec Docker Compose

**Persistance des volumes préexistants** : Tous nos volumes étaient déjà configurés et Docker Compose permet de maintenir cette structure sans reconfiguration complexe.

**Simplicité de configuration** : La définition complète de l'infrastructure dans un seul fichier docker-compose.yml facilite la maintenance et la compréhension.

**Orchestration multi-conteneurs** : Notre application implique plusieurs services (Airflow, MLflow, API, base de données) qui doivent fonctionner ensemble.

**Cohérence entre environnements** : Garantit que l'application fonctionne de manière identique en développement et en production.

**Gestion simplifiée des dépendances** : Les dépendances entre services sont explicitement déclarées et gérées.

**Facilité de mise à l'échelle** : Possibilité d'augmenter le nombre de répliques de certains services si nécessaire.

**Coût réduit** : Solution légère ne nécessitant pas d'infrastructure Kubernetes ou d'autres technologies complexes.

### Runner auto-hébergé sur la VM

Notre choix d'utiliser un runner GitHub auto-hébergé directement sur la VM de production présente plusieurs avantages significatifs :

### Avantages du runner auto-hébergé

**Indépendance vis-à-vis de l'adresse IP** : Les machines virtuelles en cloud peuvent changer d'adresse IP lors des redémarrages. Avec un runner auto-hébergé, pas besoin de reconfigurer les accès à chaque changement d'IP.

**Pas besoin d'adresse publique** : La VM n'a pas besoin d'être accessible depuis l'extérieur, car c'est le runner qui établit la connexion sortante vers GitHub.

**Sécurité renforcée** : Les informations d'authentification restent sur la VM et ne transitent pas à travers des machines hébergées par des tiers.

**Performance améliorée** : Les déploiements sont plus rapides car il n'y a pas de transfert de fichiers entre un runner externe et la VM.

**Accès direct aux services** : Le runner peut accéder directement aux services via localhost, simplifiant les vérifications post-déploiement.

La configuration du runner dans notre workflow est visible ici :

yaml

runs-on: [self-hosted, weather-app]

### Utilisation de Localhost pour les vérifications

Vous aurez remarqué que nous utilisons localhost pour vérifier les services et déclencher le DAG initial :

yaml

check\_service\_ready "Airflow" "http://localhost:8080/health" 30 10 || exit 1

Cette approche est possible uniquement parce que notre runner s'exécute directement sur la VM de déploiement. Elle présente plusieurs avantages :

* **Vérification réelle** : Nous vérifions que les services sont réellement accessibles sur la machine cible.
* **Sécurité** : Les services peuvent rester en accès privé, sans nécessité d'exposition externe.
* **Performance** : Les requêtes localhost sont beaucoup plus rapides et fiables.
* **Simplicité** : Pas besoin de gérer des URLs dynamiques ou des configurations DNS complexes.

### Gestion des secrets avec GitHub

Notre workflow utilise plusieurs variables d'environnement sensibles, qui sont stockées dans les GitHub Secrets :

yaml

env:

AIRFLOW\_IMAGE\_NAME: apache/airflow:2.8.1

AIRFLOW\_UID: 50000

AIRFLOW\_GID: 50000

\_AIRFLOW\_WWW\_USER\_USERNAME: airflow

\_AIRFLOW\_WWW\_USER\_PASSWORD: ${{ secrets.AIRFLOW\_PASSWORD }}

DB\_NAME: ${{ secrets.DB\_NAME }}

DB\_USER: ${{ secrets.DB\_USER }}

DB\_PASSWORD: ${{ secrets.DB\_PASSWORD }}

DB\_SECRET\_KEY: ${{ secrets.DB\_SECRET\_KEY }}

SMTP\_PASSWORD: ${{ secrets.SMTP\_PASSWORD }}

Les avantages de cette approche sont :

* **Sécurité** : Les informations sensibles ne sont jamais exposées dans le code source
* **Gestion centralisée** : Tous les secrets sont gérés au même endroit
* **Restriction d'accès** : Seuls les administrateurs du dépôt peuvent accéder aux secrets
* **Environnements multiples** : Possibilité de définir des secrets spécifiques à chaque environnement

### Conclusion

Notre pipeline CI/CD offre un équilibre entre simplicité et robustesse, en tirant parti des outils modernes comme GitHub Actions, Docker Compose et les runners auto-hébergés. Cette approche nous permet de :

* Garantir la qualité du code via des tests automatisés
* Simplifier le déploiement grâce à Docker Compose
* Sécuriser nos informations sensibles avec les GitHub Secrets
* Vérifier efficacement que tous les services sont fonctionnels
* Initialiser automatiquement le pipeline de prédiction

Cette infrastructure DevOps solide nous permet de nous concentrer sur le développement des fonctionnalités de notre application de prédiction météorologique, en réduisant le temps consacré à la gestion des déploiements et à la résolution des problèmes d'infrastructure.

# **MONITORING**

Pour quoi faire ?

* Suivi des performances du modèle via Mlflow
* Détection de dérive de données
* Réentraînement automatique hebdomadaire via DAG 2
* Suivi automatisé via Airflow
* Logs centralisés via Airflow
* Alertes via Airflow

## Mesures pour quantifier la création de valeur pour les utilisateurs finaux

1. **Précision des prévisions météorologiques**

* **Mesure :** Exactitude (%)
* **Méthode / Cf** **train\_model.py** : le modèle est évalué avec plusieurs métriques, notamment l'accuracy, la precision, le recall et le F1-score
* **Objectif :** Atteindre une exactitude élevée (idéalement > 85 %), en particulier pour des paramètres critiques comme les précipitations

1. **Impact sur la prise de décision des utilisateurs finaux**

* **Mesure :** Réduire des erreurs dans la planification
* **Exemple :** Agriculteurs pouvant planifier l'irrigation en fonction des prévisions de pluie, ce qui optimise l'utilisation de l'eau.
* **Méthode :** Réaliser des enquêtes avant/après l'utilisation du système ML pour quantifier la réduction des erreurs de planification

1. **Satisfaction des utilisateurs**

* **Mesure :** Score de satisfaction des utilisateurs (sur 10 ou en pourcentage)
* **Exemple** : Un agriculteur qui évite des pertes de récolte en anticipant des précipitations excessives
* **Objectif** : Améliorer ce score de satisfaction au fil du temps

1. **Réduction des coûts liés aux mauvaises décisions météorologiques**

* **Mesure** : Coût évité
* **Exemple** : Un transporteur économisant sur les coûts de carburant en ajustant ses itinéraires selon les prévisions de vent ou de tempête
* **Objectif**: Quantifier les économies réalisées et montrer la valeur ajoutée du système

1. **Temps de réaction amélioré**

* **Mesure** : Gain de temps de reaction (en % ou en heure)
* **Objectif** : Minimiser le délai entre la prévision et l’action des utilisateurs finaux

## Mesures pour quantifier la création de valeur pour les entreprises

L'objectif est de démontrer l'impact en termes de réduction des coûts, augmentation de la productivité, et optimisation des opérations

1. **Réduction des coûts opérationnels**

* Mesure :coût évité
* Mesurer les \*\*économies\*\* réalisées par les entreprises grâce aux prédictions précises
* Optimisation des itinéraires logistiques pour éviter les conditions météorologiques défavorables

1. **Augmentation de la productivité**

* Mesure : Amélioration de la productivité (%)
* Les prédictions météorologiques permettent aux entreprises d'améliorer la gestion des ressources (personnel, matériel) en fonction des conditions attendues

1. **Temps de calcul et de prédiction optimisé**

* Mesure : Temps de traitement et de prédiction (en secondes ou minutes)
* Mesurer la rapidité avec laquelle le système peut fournir une prévision après réception des données d'entrée (surtout pour les modèles en temps réel)

🡺**Objectif** : Réduire ce délai pour permettre aux utilisateurs de réagir rapidement aux conditions météorologiques changeantes

1. **Stabilité et performance du système**

* Mesure : Temps d’arrêt du système (% de disponibilité)
* Mesurer la disponibilité du système ML en production et sa robustesse face aux charges de travail croissantes.

🡺**Objectif** : Garantir une disponibilité proche de 100 %, surtout lors des périodes météorologiques critiques

1. **Retour sur investissement (ROI)**

* Mesure : ROI (%)
* Le **retour sur investissement** (ROI) est une mesure essentielle pour déterminer si la mise en place du système de ML est financièrement rentable pour l'entreprise

## Mesures pour la performance du modèle ML

**a)** **Précision de la prédiction**

* **Mesure :** Cf train\_model.py, le modèle est évalué avec plusieurs métriques:
* Accuracy: Pourcentage global de prédictions correctes
* Precision: Proportion de vrais positifs parmi les cas prédits positifs
* Recall: Proportion de vrais positifs détectés parmi tous les cas réellement positifs
* F1-score: Moyenne harmonique entre précision et recall
* ROC AUC: Mesure la capacité du modèle à distinguer les classes
* PR AUC: Performance sur données déséquilibrées
* **Objectif :** Réduire au maximum ces erreurs pour que les prédictions soient aussi fiables que possible

**b)** **Robustesse du modèle**

* **Mesure :** Robustesse des prévisions face aux outliers ou aux conditions exceptionnelles
* Le script prepare\_data.py inclut un traitement des valeurs aberrantes, ce qui améliore la robustesse du modèle
* **Objectif :** Vérifier si le modèle reste performant même lors d'événements météorologiques inhabituels ou extrêmes

**c)** **Temps de réentraînement du modèle**

* **Mesure :** Temps de réentraînement (en heures ou minutes)
* **Méthode :** Mesurer le temps d'exécution du DAG 2 (Training Pipeline) qui s'exécute hebdomadairement
* **Objectif** : Minimiser le temps de réentraînement pour s'assurer que le modèle reste à jour en permanence

**d) Impact de la performance sur l'utilisateur final**

* **Mesure :** Impact de la précision des prédictions sur l'utilisateur final (en % ou en satisfaction)
* **Méthode :** Relier la précision des prédictions à la satisfaction des utilisateurs finaux ou aux économies réalisées
* **Exemple :** Un modèle avec une précision accrue de 10% pourrait réduire les coûts de transport d'une entreprise de 5%

**Les mesures pour quantifier la création de valeur** et **mesurer l'impact d'un système de machine learning dans la production** doivent inclure à la fois **des métriques techniques** (précision, erreurs, disponibilité) et **des métriques métier** (réduction des coûts, gain de productivité, satisfaction des utilisateurs).

Ces mesures permettent de **démontrer l'efficacité du système ML** à la fois pour les utilisateurs finaux et pour les entreprises, **tout en garantissant un retour sur investissement positif.**

# **DEPLOIEMENT & STRATEGIES DE MISE EN PRODUCTION**

## Conteneurisation

* **Docker pour environnement reproductible**

Nous utilisons Docker pour assurer la reproductibilité de notre environnement de

déploiement. Docker permet de packager l’application avec toutes ses dépendances,

garantissant ainsi que l’exécution est cohérente quel que soit l’environnement

(développement, test, production).

* **Kubernetes pour orchestration des DAGs (future implémentation)**

Nous n’avons pas eu le temps d’implémenter Kebernetes pour orchestrer l’exécution des DAGs (Directed Acyclic Graphs) de notre pipeline de Machine Learning.

Toutefois, dans l’optique d’une future implémentation, nous utiliserons Kubernetes pour gérer le déploiement, la mise à l’échelle et l’ordonnancement des tâches définies dans nos DAGs.

## Monitoring Proactif

* **Suivi des performances du modèle via MLflow**

MLflow est intégré pour suivre les métriques de performance de notre modèle au fil du temps. Ceci nous permet de suivre les métriques spécifiques suivantes : précision, rappel et F1-score. Mlflow nous aide à visualiser et à comparer les performances des différentes versions du modèle.

* **Détection de dérive de données entre les distributions des données d'entraînement et de prédiction**

Décrire notre approche pour détecter la dérive des données :

Préciser les outils que nous utilisons pour comparer les distributions des données d’entraînement et des données de prédiction.

Expliquez comment nous définissons des seuils pour déclencher des alertes en cas de dérive significative.

* **Réentraînement automatique hebdomadaire via le DAG 2**

Préciser comment le DAG 2 est configuré pour réentraîner automatiquement le modèle chaque semaine.

Détailler les étapes du DAG 2 (collecte des nouvelles données, prétraitement, entraînement du modèle, évaluation, déploiement) et comment il s’intègre avec MLflow pour le suivi des expériences.

## Monitoring & Déploiement

**Suivi & gestion post-déploiement** ---- **Cf script prediction\_dag.py) :**

* **Suivi automatisé** (via Airflow)

Expliquer comment Airflow permet de surveiller l’état d’exécution de chaque tâche dans nos DAGs.

Détailler comment nous utilisons l’interface d’Airflow pour suivre les succès, les échecs et la durée des tâches.

* **Logs centralisés** (via Airflow) : Les journaux d'exécution sont centralisés et accessibles via l’interface Airflow pour faciliter le diagnostic des problèmes.
* **Alertes** (via Airflow) :

Décrire comment nous avons intégré des notifications (e-mail ou Slack) en cas d’échec d’une tâche critique. Préciser les types d’événements qui déclenchent ces alertes.

**Suivi & gestion post-déploiement** ---- **Cf script predict\_api.py) :**

Informations sur l'utilisation de MLflow pour le monitoring et le suivi des performances des prédictions.

* **MLflow :** Le suivi des runs MLflow permet de monitorer les performances des prédictions au fil du temps.
* **Logging :** Utilisation de ce module pour enregistrer les erreurs et les informations importantes.
* **Variables d'environnement :** L'API dépend de variables d'environnement pour la configuration de MLflow et d'autres paramètres.
* **Infrastructure:** Dépendances sur un registre de modèles MLflow et potentiellement d'autres services (base de données, etc.).

## Axes d'Amélioration

* Intégration de modèles d'ensembleplus complexes (augmenter n\_estimators au-delà de 10)
* Features engineering avancé incluant des lag features et des agrégations temporelles
* Incorporation de données satellitaires comme sources complémentaires
* Modèles probabilistes plus sophistiqués pour mieux quantifier l'incertitude des prédictions

## Considérations Éthiques et Environnementales

* Équité géographique des prédictions grâce à une représentation équilibrée des différentes régions
* Impact carbone du machine learning (optimisation des ressources de calcul)
* Transparence algorithmique et interprétabilité des prédictions (avantage du RandomForest)

## Conclusion Technique

Un système de prédiction météorologique efficace nécessite une approche multidimensionnelle combinant :

* Des prétraitement rigoureux (comme implémenté dans prepare\_data.py)
* Des algorithmes adaptatifs (RandomForest avec potentiel d'optimisation)
* Un monitoring continu via MLflow et une orchestration des workflows par DAGs
* Une amélioration itérative basée sur les métriques de performance collectées