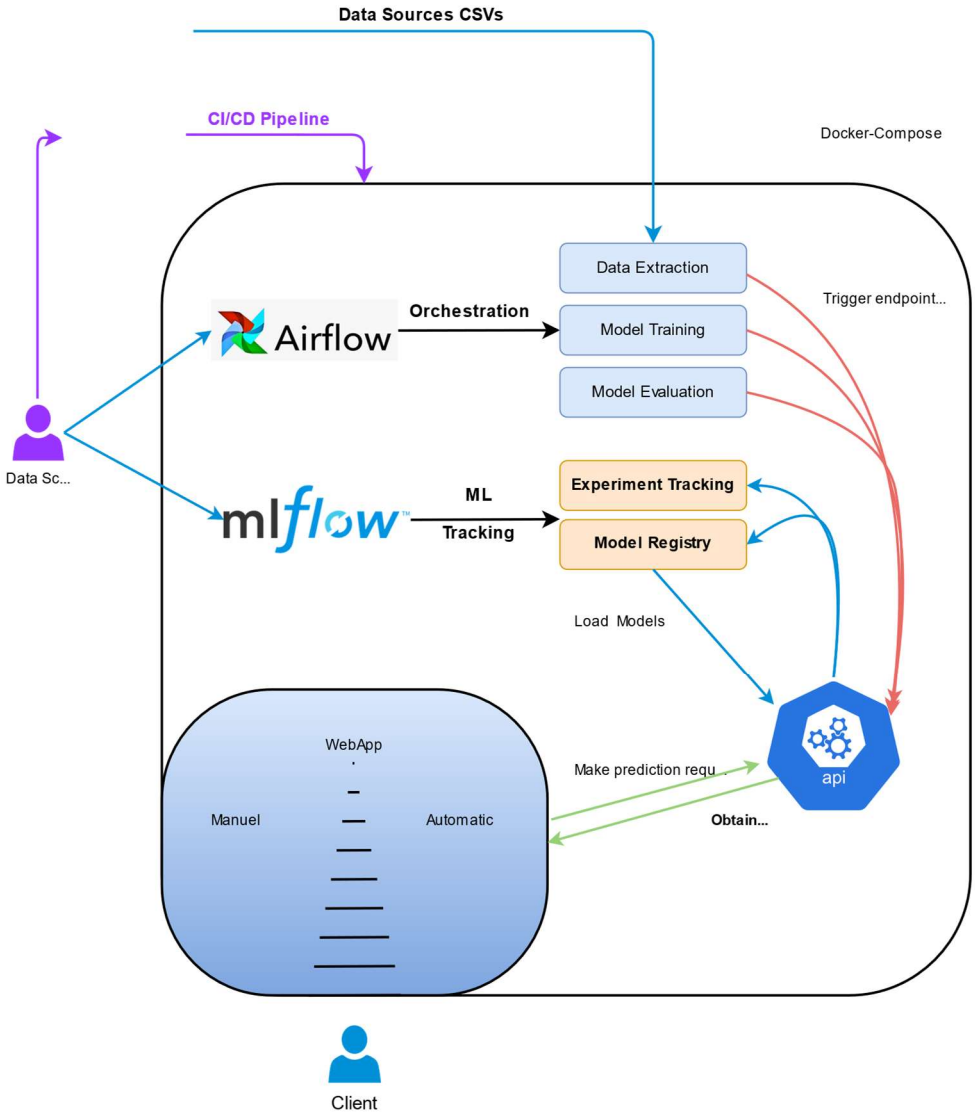
Machine Learning Canvas



Text is not SVG - cannot display

Table des matières

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 obtenir des 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 estimé : 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

Nous avons opté pour une décomposition du code du modèle de machine learning en fonction de chaque étape de développement, de l'extraction des données brut jusqu'à l'évalution du modèle.

En effet, au delà de la nécessité d'avoir une application structurée et organisée, chaque endpoint doit être suffisamment explicite et robuste pour pouvoir être exploitable par notre gestionaire de modèles MLFlow et notre orchestrateur Airflow (il est primordial que chaque tâche (DAG) se réfère à un endpoint spécifique au suivi attendu). Cela se traduit par la présence d'un script pour un endpoint avec un total de 4 scripts (extract_api.py, training_api.py, predict_api.py et evaluate_api.py).

APIRouter:

nous utilisons l'instance APIRouter de fastapi pour répondre à ce besoin:

```
from fastapi import APIRouter

router = APIRouter()

@router.get("/extract") ---> endpoint extract du script extract_api.py

@router.get("/predict") ---> endpoint predict du script predict_api.py
```

Enfin, chaque route est centralisée dans un script workflow_api.py pour obtenir la synthèse a complète du code de notre modèle:

```
@router.api_route("/workflow/start", methods=["GET", "POST"])
```

L'avantage d'une telle approche est la modularité (code organisé en modules facilitant la maintenance et l'évolutivité de l'infrastructure (intégration de MLFlow par exemple), la réutilisabilité (les routes sont facilement partageables entre différentes application FastAPI,) et la clarté (chaque fonctionnalité de l'application est définie séparemment dans un script propre à chacune)

Intégration de MLFlow:

L'intégration de MLFlow dans l'API demande une adaptation du code pour chaque endpoint. En effet chaque workflow créé par MLFow doit être importé et explicité dans notre code (utilisation du *run_id).*Vous retrouverez la déscription complète dans la partie dédiés à MLFow dans la suite du rapport.

Ainsi que de la présence de endpoints spécifiques à MLFlow (script *workflow_api.py)* et correspondant à chaque étape “workflows” de MLFlow:

```
@router.api_route("/workflow/start", methods=["GET", "POST"])

async def start_workflow():

@router.post("/workflow/complete")

async def complete_workflow(run_id: str, status: str = "COMPLETED",
        error_message: str = None):

@router.post("/workflow/run-full")

async def run_full_workflow():
```

Prédictions automatiques et manuelles:

Nous avons fait le choix d'utiliser deux options pour prédire le temps, une prédiction automatique sur la dernière ligne de notre dataset ou une prédiction manuelle où des données sont saisis manuellement par l'utilisateur.

Au niveau de la fonction de prédiction la présence ou non de l'argument user_input définira son orientation:

```
def predict_weather(user_input = None):

"Début du code"

if user_input is not None:

                logger.info("Preparing user input data")
                # User input prediction
                mlflow.set_tag("prediction_type", "user_input")
                input_df, _, _ = extract_and_prepare_df(
        PREDICTION_RAW_DATA_PATH,
        csv_file_daily_prediction,
        user_input=user_input,
        log_to_mlflow=True
        )
    input_df = input_df.reindex(columns=feature_order)
    logger.info(f"Prepared input DataFrame: {input_df}")

  else:

    # Input file for prediction
    mlflow.set_tag("prediction_type", "file_based")
    input_file = PREDICTION_RAW_DATA_PATH / csv_file_daily_prediction
    mlflow.set_tag("prediction_input_file", str(input_file))
    logger.info(f"prediction input file: {input_file}")

    # Prepare the data (with minimal logging)
    input_df, _, _ = extract_and_prepare_df(
       PREDICTION_RAW_DATA_PATH,
       csv_file_daily_prediction,
       log_to_mlflow=False  # Disable detailed logging for prediction
preparation
    )

    « Suite du code »
```

Pour rendre l'option manuelle effective, nous devons introduire une class composée des arguments utilisés pour la prédiction et disponibles après le nettoyage du dataset d'origine. Pour cela nous utilisons la BaseModel de la bibliothèque Pydantic.

Cette classe définit la structure des données d'entrées et les validera lorsque qu'elles seront envoyées par l'utilisateur à notre API. Vous trouverez ci-dessous l'exemple d'un argument de la classe:

```
from pydantic import BaseModel

class UserInputPrediction(BaseModel):
```

Location: int = Field(

...,

gt=0,

description="Location code (positive integer)"

)

De plus, des tests de validation sont également définits pour assurer la cohérance de certains arguments les uns par rapport aux autres (par exemple la température maximale ne peut pas dépasser la température minimale):

@validator('MaxTemp')

```
def check_max_temp_greater_than_min(cls, v, values):
"""Ensure MaxTemp is not less than MinTemp"""
if 'MinTemp' in values and v < values['MinTemp']:
raise ValueError("MaxTemp must be greater than or equal to MinTemp")
return v
```

Enfin, une classe Config est paramétré pour empêcher la présence d'arguments supplémentaires :

```
class Config:
  extra = 'forbid'
```

Pour conclure, le routage du endpoint reprend la classe telle que définit, comme argument de la fonction.

@router.post("/predict_user")

async def predict_user_input(input_data: UserInputPrediction):

Synthèse des Endpoints :

- **/extract (GET)** : extrait et nettoie les données d'entraînement brutes (.CSV)
- **/training (GET)** : Entraîne le modèle (fonction train_model()), l'enregistre (joblib) ainsi que ses metrics (json)
- **/evaluate (GET)** : Évalue le modèle (fonction evaluate_model())
- **/predict (GET)** : Effectue une prédiction automatique basée sur les données actuelles.
- **/predict_user (POST)** : Effectue une prédiction manuelle basée sur les données saisies par l'utilisateur dans le formulaire streamlit

Lancement de l'API:

L'instance API web "Weather Prediction API" est créé lors du lancement du script main.py présent dans le dossier principal de l'api */api.*

app = FastAPI(title="Weather Prediction API")

Chaque routeur est importé, intégré et étiqueté pour couvrir l'ensemble du cycle de vie du modèle de machine learning :

```
app.include_router(extract_router, tags=["Data Extraction"])
app.include_router(training_router, tags=["Model Training"])
app.include_router(evaluate_router, tags=["Model Evaluation"])
app.include_router(predict_router, tags=["Inference"])
app.include_router(workflow_router, tags=["Workflows"])
```

Un endpoint général */ (GET)* fournit les informations sur l'API et ses fonctionnalités :

```
@app.get("/", tags=["Root"])

async def root():
    return {
        "message": "Welcome to the Weather Prediction API",
        "endpoints": {
            "extract": "/extract",
            "training": "/training",
            "evaluate": "/evaluate",
                "predict": {
            "automatic": "GET /predict",
            "manual": "POST /predict_user"
            },
            "workflow": {
            "start": "/workflow/start",
            "complete": "/workflow/complete",
            "run-full": "/workflow/run-full"
            }
    }
```

et un serveur (port 8000) est configuré pour son exécution.

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

User_api :

Il était initialement prévu de déployer une option d'enregistrement et d'identification des utilisateurs selon leur statut (administrateur ou utilisateur).

Le code et les endpoints sont présents dans le script *endpoint/user_api.py* cependant cette option n'est pas disponible dans notre application faute de temps.

Elle est néenmoins constitué comme suit :

- **Base de données PosgreSQL :** le script *init_db.py* est utilisé pour créer la base de données utilisateurs lors du montage du conteneur tel que configuré dans le *docker-compose.yml* :

```
db:
    image: postgres:13
```

```
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${DB_USER} -d ${DB_NAME}"]
      interval: 5s
      timeout: 5s
      retries: 3
```

Les variables d'environnement sont enregistrées dans un fichier .env caché.

- **/users (POST) :**

```
@router.post("/users", response_model=User)
    async def create_user(user: UserCreate, conn = Depends(get_db_user))
```

Endpoint utilisé pour la création de l'utilisateur avec chiffrement du mot de passe (bcrypt.hashpw()) et insertion du nouvel utilisateur dans la base de données PosgreSQL

- **/token (POST) :**

```
@router.post("/token", response_model=Token)
    async def login(form_data: OAuth2PasswordRequestForm = Depends(), conn =
        Depends(get_db_user)):
```

Permet la consultation et d'extraction les données utilisateurs de la base de données. Une fois extraites ces données sont vérifiées (vérification du nom de l'utilisateur et vérification de son mot de passe par rapport à celui encrypté dans la base de données (*bcript.checkpw()*). Une fois les vérification effectuées le endpoint renvoit un « token » qui permettra à l'utilisateur de se connecter.

- **/users (GET) :**

```
@router.get("/users", response_model=List[User])
    async def get_active_user(active_user: User = Depends(get_user),conn =
        Depends(get_db_user))
```

Endpoint spécifique pour consulter la base de données avec un profil un administrateur.

Tests Unitaires

Fonctionnement général :

Les tests unitaires sont effectués sur chaque endpoint avant le lancement de l'application pour permettre de vérifier leur fonctionnalité. Cette approche est fondamentale pour détecter les problèmes suffisamment tôt et garantir l'utilisation de l'interface.

Les tests relatifs aux endpoints d'extraction, d'entraînement et de prédiction automatique utilisent un run_id fictif et s'assure que l'appel est bien réalisé :

```
def test_extract_endpoint():

"""Test de l'endpoint d'extraction"""
response = client.get("/extract?run_id=test_run_id")
```

En ce qui concerne la prédiction manuel nous effectuons un test sur un jeu de données valides (chaque attribut est couvert par sa plage de valeurs limites) ainsi qu'un jeu de données invalides (la condition d'un argument ne respecte pas les spécificités définits dans la classe *UserInputPrediction*.

```
def test_invalid_predict_user_input():

    """Test avec des données invalides"""
    invalid_data = {
        "Location": 1,
        "MinTemp": 100.0,  # Température trop élevée
        "MaxTemp": 25.0    # MaxTemp < MinTemp,
    }

    response = client.post("/predict_user", json=invalid_data)
    assert response.status_code == 422
```

Intégration des tests avec MLFlow

La logique des tests est d'être effectués avant la mise en service de l'api et donc du lancement du serveur MLFlow. Or une configuration de tests « classiques » échouerait dans notre cas, le serveur MLFlow étant indisponible au lancement de tests (pour rappel nos endpoints sont définis dans l'infrastructure MLFlow). C'est pour cette raison que nous devons isoler nos tests pour qu'ils ne dépendent pas du serveur. Cela implique le développement d'une simulation d'objets qui imitent le comportement de MLFlow sans communiquer avec le serveur.

Les tests unitaires résident dans la conception de trois scripts, chacun répondant à un champs spécifique (configuration de l'environnement de test, remplacement (mock) de l'environnement MLFlow par un environnement simulée, tests).

Configuration de l'environnement de test (conftest.py):

Nous utilisons la bibliothèque Pytest. Elle a les avantages d'être simple et intuitive à utiliser. Avec le déploiement de fixtures, nous garantissons une configuration sur l'ensemble des fichiers (scripts) présents dans le répertoire de test sans nécessité d'être importé.

Différentes fixtures sont présentes :

- la fixture check_model_files :

```
@pytest.fixture(autouse=True)

def check_model_files():
```

Elle va vérifier l'existence du répertoire des modèles et la présence en son sein du modèle de machine learning (rfc.joblib et scaler.joblib)

- la fixture mock_application_functions

```
@pytest.fixture(autouse=True)
```

def mock_application_functions():

Simule un serveur MLFlow

os.environ["MLFLOW_TRACKING_URI"] = "http://fake-mlflow:5000"

Elle va remplacer les fonctions utilisées dans notre application pour effectuer les tests. De même elle simule les chemins où les données extraites, le modèle entraîné ou les métriques seraient normalement sauvegardés (comme explicité dans la partie du code de la fixture ci-dessous)

```
with patch('utils.functions.extract_and_prepare_df', return_value=(
        test_df, # DataFrame créé pour la simulation
            test_encoders, # str encodés (location, raintoday, WinDir3pm)
        "/app/api/data/prepared_data/test_output.csv"
)), \
patch('utils.functions.train_model', return_value={
        "model_path": "/app/api/data/models/test_model.pkl",
        "metrics_path": "/app/api/data/metrics/test_metrics.json"
}), \
patch('utils.functions.predict_weather', return_value=(0, 0.85)):
yield
```

Remplacement des modules de MLFlow (mock_mlflow.py) :

Nous utilisons l'instance MagikMock de unittest.mock pour créer un environnement simulé complet pour MLflow, permettant de tester notre application sans dépendre d'un serveur MLflow réel ou d'une connexion réseau.Tous les appels des tests seront redirigés vers l'environnement mocké. Le script peut se décomposer de la manière suivante :

- Préparation et configuration :

```
MagicMock.__reduce__ = lambda self: (MagicMock, ())
class MockBase:
def __repr__(self):
return f"MockMLflow({self.__class__.__name__})"
```

Configure le mock pour fonctionner avec pickle (sérialisation) et création d'une base commune pour les mocks.

- Reproduction de la structure de MLFlow :

```
# Mock pour le module mlflow.tracking
class MockTracking(MockBase):
    class MlflowClient(MockBase):
```

Cette section implémente la classe MockTracking qui simule le module mlflow.tracking
composé del a classe MlflowClient qui gère les interactions avec le serveur MLFlow. C'est sous cette classe que sont simulées toutes les opérations de tracking (expériences, runs, métriques).

```
# Mock pour le module mlflow.models
class MockModels(MockBase):
# Mock pour le module mlflow.sklearn
class MockSklearn(MockBase):
# Mock pour le module mlflow.pyfunc
class MockPyfunc(MockBase):
```

Cette section concerne la simulation des modules de gestion des modèles de machine learning, des modèle scikit-learn et des fonctions de prédicions.

```
# Mock pour un run MLflow actif
class MockRun(MockBase):
```

Ici est implémenté la classe qui simule le module de gestion des runs avec la configuration de la structure de données pour enregistrer les informations du run (tags, params, metrics).

```
# Mock pour le module MLflow principal
class MockMLflow(MockBase):
```

Enfin cette section implémente la classe principale MockMLflow qui simule le module mlflow lui-même, soit toutes les méthodes principales accessibles directement via la bibliothèque mlflow (création des runs, lancement des runs, méthodes de connexions, méthodes de recherche et de gestion des runs.

- Initialisation et insertion des mocks

```
mock_mlflow = MockMLflow()
mock_mlflow_tracking = mock_mlflow.tracking
mock_mlflow_models = mock_mlflow.models
mock_mlflow_sklearn = mock_mlflow.sklearn
mock_mlflow_pyfunc = mock_mlflow.pyfunc
```

Creation des instances des objets mocks et mise en relation des différents composants.

```
sys.modules['mlflow'] = mock_mlflow
sys.modules['mlflow.tracking'] = mock_mlflow_tracking
sys.modules['mlflow.models'] = mock_mlflow_models
sys.modules['mlflow.models.signature'] =
mock_mlflow_models.signature
sys.modules['mlflow.sklearn'] = mock_mlflow_sklearn
sys.modules['mlflow.pyfunc'] = mock_mlflow_pyfunc
```

Remplace les modules MLFlow réels sans sys.modules. Tout import et utilisation ultérieur d'instances de mlflow utilisera ces mocks à la place.

- **Script de test (test_endpoint.py) :**

  Il contient une série de fonctions spécifiques pour tester chaque endpoint.

```
def test_extract_endpoint():
"""Test de l'endpoint d'extraction"""
# Appel de l'endpoint avec un run_id (paramètre obligatoire)
response = client.get("/extract?run_id=test_run_id")

# Vérifications
assert response.status_code == 200
assert "status" in response.json()
```

Les endpoints d'extraction, l'entrainement et la prédiction automatique se résument à un appel avec un run_id fictif.

Pour les endpoints de prédiction manuelle nous utilisons des données valides et des données invalides :

```
def test_invalid_predict_user_input():
"""Test avec des données invalides"""
invalid_data = {
"Location": 1,
"MinTemp": 100.0,  # Température trop élevée
"MaxTemp": 25.0    # MaxTemp < MinTemp, devrait échouer
}

response = client.post("/predict_user", json=invalid_data)
assert response.status_code == 422
```

- **Fonctionnement général du script :**
  - importation des mock_mlflow
  - exécution du mock_mlfow.py
  - Tous les modules MLFlow réels sont remplacés par leur version mockée
  - Les tests appellent les endpoints
  - les endpoints utilisent des fonctions qui font appels à MLFlow
  - Les appels sont de fait redirigés vers les mocks (exécution du script de mock et donc remplacement des modules de MLFlow par leur version mockée : sys.modules)

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.

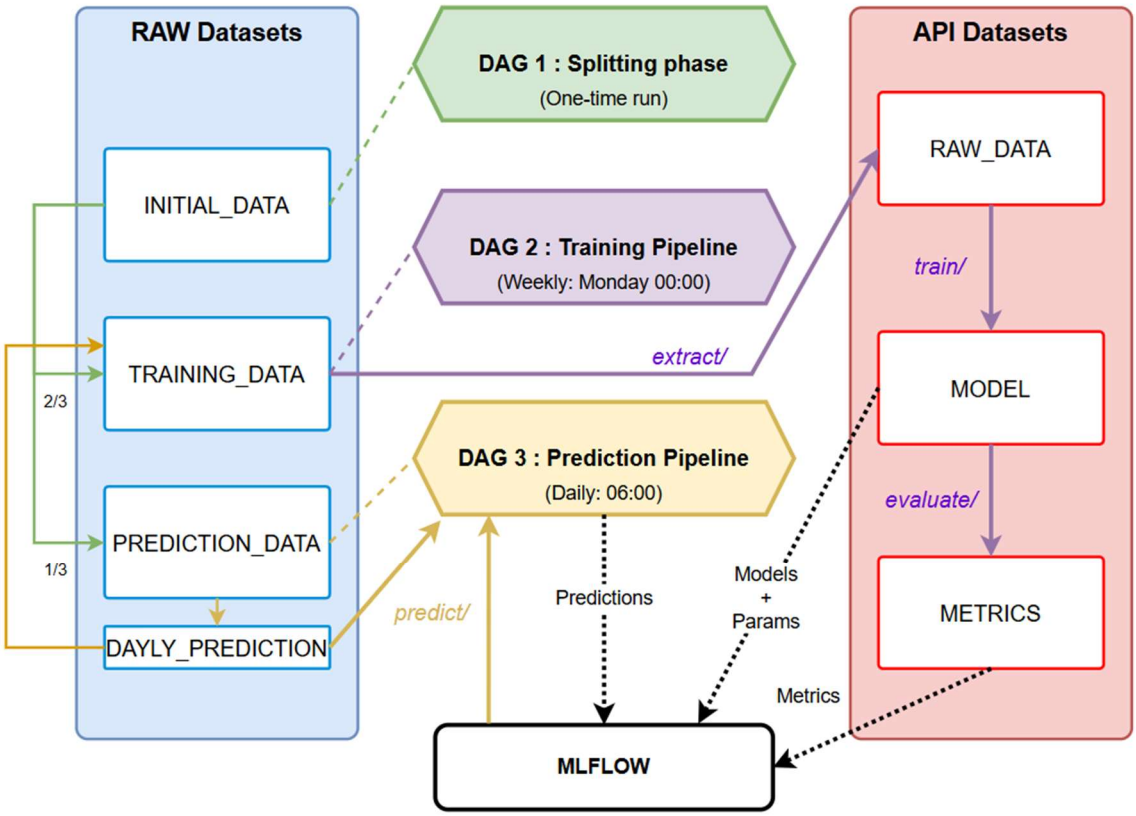


Diagramme : Chainage des DAGs

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'),
       ...
   )
   ```

2. **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',
       ...
   )
   ```

3. **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

```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

```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.

INTEGRATION 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.

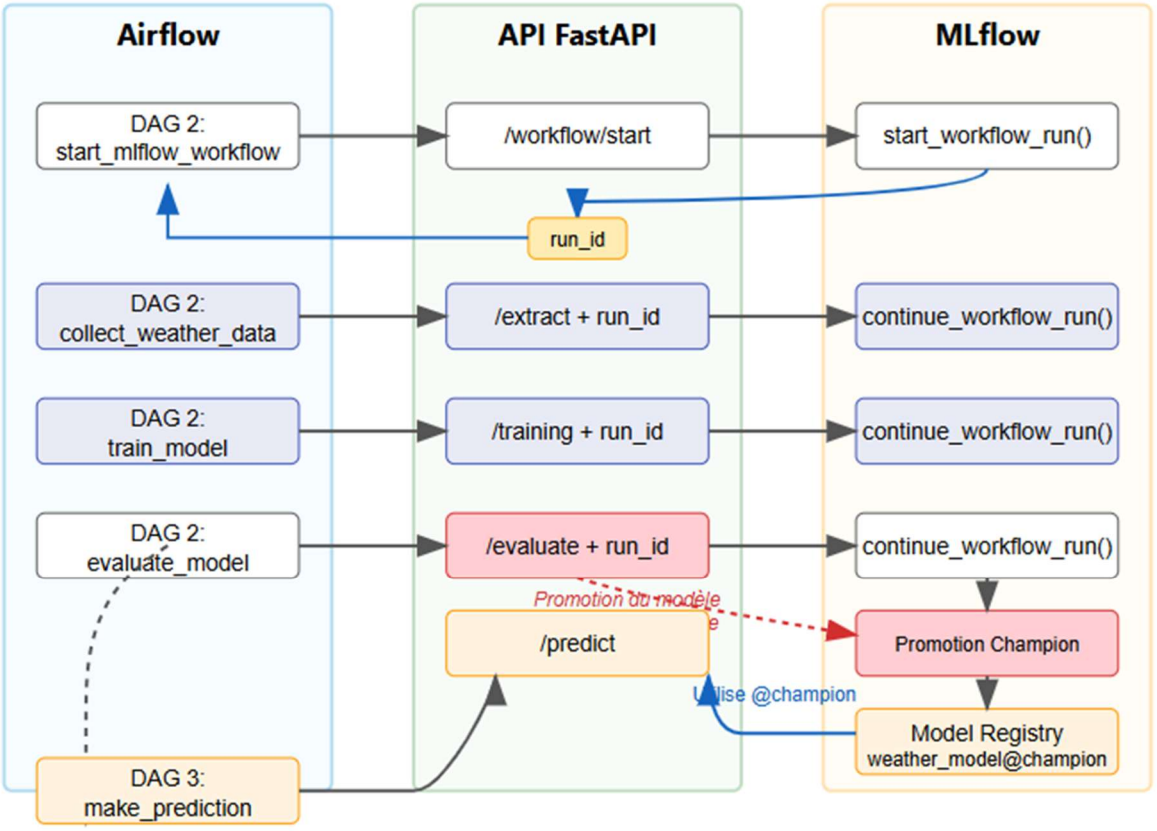


Diagramme : Cycle de gestion MLflow

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

```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

```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)
```

2. 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)
```

3. 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
```

4. 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})")
```

5. 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...
```

Runs de prédiction imbriqués avec le modèle champion

Notre approche MLflow intègre un mécanisme sophistiqué de runs imbriqués pour les prédictions, offrant une traçabilité complète tout en maintenant l'efficacité opérationnelle. Ce système s'articule autour du concept de "runs de déploiement" associés au modèle champion.

Lors d'une requête de prédiction, voici comment s'organise ce processus:

```
python
# Obtenir le run de déploiement pour le modèle champion actuel
deployment_run_id, model_version = get_deployment_run()

# Créer un run imbriqué spécifique à cette prédiction
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
prediction_run_name = f"automatic_prediction_V{model_version}_{timestamp}"

# Utiliser le run de déploiement comme parent
with mlflow.start_run(run_id=deployment_run_id):
    with mlflow.start_run(run_name=prediction_run_name, nested=True) as nested_run:
        # Ajouter des tags pour identifier ce run
        mlflow.set_tag("pipeline_type", "prediction")
        mlflow.set_tag("prediction_date", timestamp)
        mlflow.set_tag("model_version", str(model_version))

        # Exécuter la prédiction avec le modèle champion
        prediction, probability = predict_weather(user_input=row_data)
```

La fonction `get_deployment_run()` joue un rôle central en identifiant ou créant un run de déploiement lié au modèle champion actuel. Son fonctionnement repose sur un système ingénieux de tags et de recherche:

```
python
# Attribution de tags spécifiques lors de la création d'un run de déploiement
with mlflow.start_run(run_name=f"model_deployment_v{model_version}_{timestamp}") as
run:
    # Tags qui serviront ultérieurement de critères de recherche
    mlflow.set_tag("model_deployment", "True")  # Identifie le type de run
    mlflow.set_tag("model_version", str(model_version))  # Associe au modèle
spécifique
    mlflow.set_tag("pipeline_type", "deployment")  # Catégorisation du pipeline
```

Ces tags sont ensuite utilisés comme critères de recherche pour retrouver le run de déploiement correspondant à une version spécifique du modèle:

```
python
# Recherche ciblée d'un run existant grâce aux tags précédemment attribués
runs = client.search_runs(
    experiment_ids=[experiment.experiment_id],
    filter_string=f"tags.model_deployment = 'True' AND tags.model_version =
'{model_version}'",
    max_results=1
)
```

Cette requête agit comme une véritable clause WHERE dans une base de données, filtrant les runs pour ne retenir que ceux qui correspondent exactement aux critères spécifiés. Le système peut ainsi déterminer automatiquement s'il doit créer un nouveau run de déploiement ou réutiliser un run existant pour la version courante du modèle champion.

Chaque prédiction utilise ensuite ce modèle champion pour générer un résultat:

```
python
# Chargement du modèle via l'alias champion
model_uri = f"models:/{model_name}@champion"
model = mlflow.sklearn.load_model(model_uri)

# Standardiser les données et faire la prédiction
input_scaled = scaler.transform(input_df)
prediction = model.predict(input_scaled)[0]
probability = model.predict_proba(input_scaled)[0][1]

# Loguer les résultats dans le run imbriqué
mlflow.log_metric("prediction_result", int(prediction))
mlflow.log_metric("prediction_probability", float(probability))
```

Cette architecture présente plusieurs avantages majeurs:

Traçabilité à deux niveaux

Chaque prédiction individuelle est enregistrée comme un run dédié, tout en maintenant la relation avec le run de déploiement parent qui représente une "session" d'utilisation du modèle.

Isolation des logs et métriques

Les détails de chaque prédiction sont isolés dans leur propre run, facilitant l'analyse des performances individuelles tout en évitant de surcharger le run principal.

Structure hiérarchique métier

La relation parent-enfant entre le déploiement et les prédictions reflète fidèlement la logique métier du système.

Accès transparent au modèle courant

Grâce à l'alias "champion", le système charge automatiquement la version validée la plus récente sans modification du code.

Cette approche par runs imbriqués complète notre stratégie d'alias pour créer un écosystème MLflow où chaque prédiction est non seulement traçable mais également reliée de façon explicite au modèle qui l'a générée, formant ainsi un historique complet et auditable de notre activité de prédiction en production.

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.

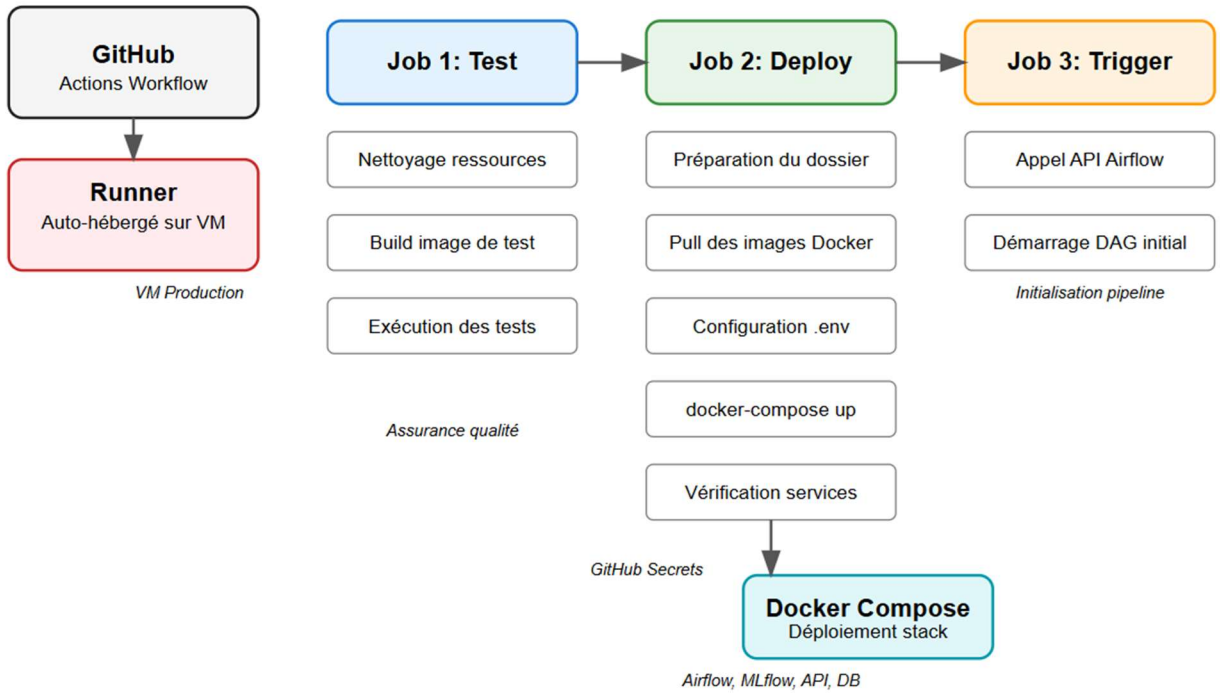


Diagramme : Pipeline CI/CD

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:
```

```yaml
    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
```

```yaml
- 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

a) **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

b) **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

c) **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

d) **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

e) **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

a) 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

b) 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

c) 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

d) 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

e) 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

Réentraînement automatique hebdomadaire via le DAG 2

Monitoring & Déploiement

Suivi & gestion post-déploiement ---- Cf script prediction_dag.py) :

- **Suivi automatisé** (via Airflow)
- **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) :

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'ensemble plus 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
- ✓ Une API REST modulaire permettant un déploiement standardisée des modèles via des endpoints aux tâches spécifiques, facilitant l'intégration et l'automatisation des prédictions dans divers environnements applicatifs.