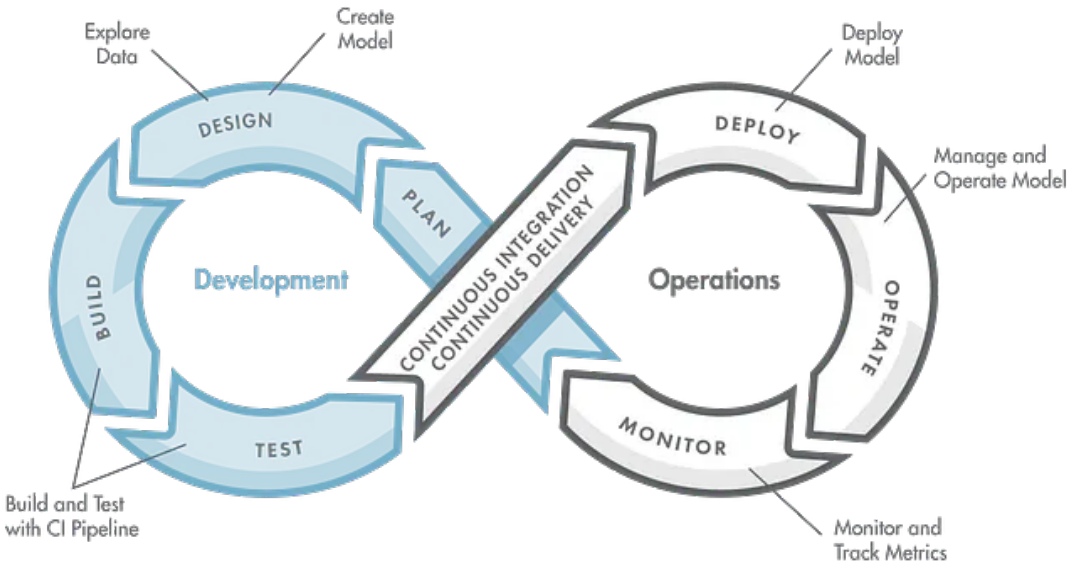
ENSET / Université Hassan II

Classification de texte avec pipeline CI/CD complet

Rapport technique



Réalisé par

Akram Benhammou
Oussama Khouya

Encadré par

Professeur Soufiane HAMIDA

15 décembre 2025

Résumé

Ce rapport présente la conception et la mise en place d'une chaîne MLOps complète pour un problème de classification de texte sur le jeu de données *20 Newsgroups*. Le pipeline couvre le prétraitement linguistique, l'entraînement d'un modèle de classification basé sur TF–IDF et Random Forest, le suivi expérimental avec MLflow, ainsi que la sauvegarde des artefacts. Le modèle est ensuite emballé dans une image Docker et exposé via une API FastAPI, avant d'être intégré dans des workflows CI/CD GitHub Actions pour automatiser les tests, la construction de l'image, la génération de rapports CML et un scénario de déploiement *staging* $\rightarrow$ *production* avec rollback. L'objectif est d'illustrer, de bout en bout, les bonnes pratiques DevOps et MLOps autour d'un cas d'usage NLP réaliste.

Table des matières

1 Introduction **3**
1.1 Contexte et problématique . . . 3
1.2 Objectifs du projet . . . 3
1.3 Périmètre et livrables . . . 3

2 Description fonctionnelle et données **3**
2.1 Cas d'usage métier . . . 3
2.2 Jeu de données et splits . . . 3
2.3 Contraintes et critères de succès . . . 4

3 Architecture globale MLOps **4**
3.1 Vue d'ensemble . . . 4
3.2 Organisation des répertoires . . . 4

4 Pipeline de données et modèle NLP **5**
4.1 Préparation des données . . . 5
4.2 Modélisation et entraînement . . . 6
4.3 Évaluation et validation du modèle . . . 7

5 Suivi expérimental et gestion des modèles **8**
5.1 Tracking des expériences avec MLflow . . . 8
5.2 Gestion des artefacts . . . 8

6 Service d'inférence et packaging **8**
6.1 API FastAPI . . . 8
6.2 Containerisation Docker . . . 9

7 Stratégie de tests et validation **9**
7.1 Tests unitaires et d'intégration . . . 9
7.2 Validation de modèle . . . 10

8 Pipelines CI/CD et déploiement **10**
8.1 Build, tests et image Docker . . . 10
8.2 Rapport automatique avec CML . . . 11
8.3 Déploiement staging → production et rollback . . . 12

9 Monitoring et exploitation **14**

10 Limites et axes d'amélioration **14**
10.1 Limites du modèle et des données . . . 14
10.2 Limites MLOps / CI/CD . . . 14
10.3 Sécurité, robustesse et industrialisation . . . 14

11 Conclusion **15**

1 Introduction

1.1 Contexte et problématique

Le projet s'inscrit dans le cadre du module *DevOps & MLOps*. Un journal en ligne souhaite automatiser la catégorisation de ses articles en fonction de leur contenu, afin de faciliter la navigation, la recommandation et l'analyse éditoriale.

Ce travail met en place une chaîne MLOps complète pour un problème de classification de texte sur le jeu de données *20 Newsgroups*. L'objectif est de couvrir tout le cycle de vie du modèle : prétraitement des données, entraînement et suivi expérimental, packaging dans une image Docker, mise à disposition via une API FastAPI, et automatisation par des pipelines CI/CD GitHub Actions incluant la génération de rapports CML et un flux de déploiement *staging* → *production*.

1.2 Objectifs du projet

Les objectifs principaux sont les suivants :

— **Objectifs ML** : construire un modèle de classification de texte performant sur un corpus multi-classes (20 catégories), en assurant un prétraitement NLP cohérent entre entraînement et inférence et en obtenant des métriques de performance satisfaisantes (accuracy, F1 pondérée, etc.).

— **Objectifs DevOps / MLOps** : mettre en place des pipelines CI/CD pour l'entraînement et le déploiement du modèle, automatiser les tests unitaires et d'intégration, tracer les expériences avec MLflow, générer des rapports CML dans les pull requests et orchestrer un déploiement progressif avec mécanisme de rollback.

1.3 Périmètre et livrables

Le périmètre couvre l'ensemble de la chaîne de traitement, depuis la préparation des données textuelles jusqu'au déploiement d'un service d'inférence containerisé :

— un dépôt Git structuré contenant le code source, les scripts d'entraînement et de prétraitement, les tests et les workflows GitHub Actions

— un pipeline de données reproduisible pour le téléchargement, le nettoyage, le prétraitement et le découpage du jeu de données

— un modèle de classification accompagné de ses artefacts (vectoriseur, métriques, rapports)

— une API FastAPI packagée dans une image Docker

— des pipelines CI/CD intégrant tests, build/push Docker, génération de rapports CML et déploiement *staging* → *production* avec rollback.

2 Description fonctionnelle et données

2.1 Cas d'usage métier

Le contexte est celui d'un journal en ligne qui veut automatiser la catégorisation de ses articles. Chaque texte doit être attribué à une catégorie thématique (informatique, sport, politique, religion, etc.), ce qui permet :

— d'améliorer l'expérience utilisateur (navigation par rubrique, recommandations)

— d'aider la rédaction à analyser la répartition des contenus

2.2 Jeu de données et splits

La solution implémentée s'appuie sur le dataset public *20 Newsgroups* :

— **Entrée** : texte brut (emails / articles en anglais).

— **Labels** : 20 classes correspondant à des groupes de discussion (informatique, sport, politique, religion, etc.).
— **Chargement** : via `sklearn.datasets.fetch_20newsgroups`.

Le pipeline de préparation découpe le dataset en trois ensembles distincts :
— **Train** : 70% des données, utilisé pour l'entraînement du modèle.
— **Validation** : 15% des données, utilisé pour le réglage d'hyperparamètres et la comparaison de variantes.
— **Test** : 15% des données, réservé à l'évaluation finale.

Ces splits sont stratifiés sur la variable `target` afin de conserver la distribution des classes dans chaque sous-ensemble.

2.3 Contraintes et critères de succès

Les principaux critères de succès sont :
— une performance de classification correcte (par exemple accuracy significatif) sur le jeu de test
— un prétraitement NLP cohérent entre la phase d'entraînement et le service d'inférence
— des pipelines CI/CD capables de reconstruire le modèle, d'exécuter les tests et de déployer automatiquement un service à jour
— la possibilité de diagnostiquer les performances via MLflow et CML (rapports automatiques dans les pull requests).

3 Architecture globale MLOps

3.1 Vue d'ensemble

L'architecture globale suit un flux de bout en bout :

1. Préparation des données textuelles (téléchargement, nettoyage, prétraitement, splits).
2. Entraînement du modèle de classification et évaluation sur le jeu de test.
3. Suivi expérimental avec MLflow (métriques, artefacts, hyperparamètres).
4. Export des artefacts (modèle et vectoriseur) dans un répertoire dédié.
5. Packaging du service d'inférence dans une image Docker (API FastAPI).
6. Orchestration de workflows CI/CD GitHub Actions pour lancer les tests, construire/publier l'image, générer un rapport CML et déployer en *staging* puis en *production*, avec rollback en cas d'échec.

3.2 Organisation des répertoires

La structure générale du dépôt est la suivante :
— `src/` : code source Python principal
  — `preprocess.py` : téléchargement et préparation des données.
  — `train.py` : entraînement, évaluation, logging MLflow et sauvegarde des artefacts.
  — `app.py` : API FastAPI exposant les endpoints `/health` et `/predict`.
  — `predict.py` : script de prédiction en ligne de commande.
— `data/processed/` : données prétraitées (`train.csv`, `validation.csv`, `test.csv`).
— `models/` : artefacts de modèle (fichiers `model.joblib` et `tfidf_vectorizer.joblib`).
— `reports/` : `metrics.json`, `classification_report.txt` et la matrice de confusion (`confusion_matrix.p`
— `tests/` : tests unitaires et d'intégration.
— `.github/workflows/` : pipelines CI/CD GitHub Actions (`docker.yaml`, `cml.yaml`, `deploy.yaml`).
— `Dockerfile` : définition de l'image de service pour l'API.

4 Pipeline de données et modèle NLP

4.1 Préparation des données

Le script `src/preprocess.py` implémente un pipeline de préparation standard :

1. Téléchargement du dataset *20 Newsgroups* via `sklearn.datasets.fetch_20newsgroups`, avec suppression des en-têtes, pieds de page et citations pour limiter les biais.
2. Construction d'un `DataFrame` pandas contenant les colonnes `text` (texte brut) et `target` (label).
3. Nettoyage simple via `clean_text` : mise en minuscules, suppression de la ponctuation, des chiffres et des espaces superflus.
4. Traitement linguistique via `process_text` : tokenisation NLTK, suppression des stop-words anglais, lemmatisation `WordNetLemmatizer`.
5. Filtrage des exemples dont le texte prétraité est vide.
6. Découpage en ensembles d'apprentissage, validation et test avec `train_test_split` en respectant une stratification sur `target` (70% train, 15% validation, 15% test).
7. Sauvegarde des trois ensembles au format CSV dans `data/processed/`.

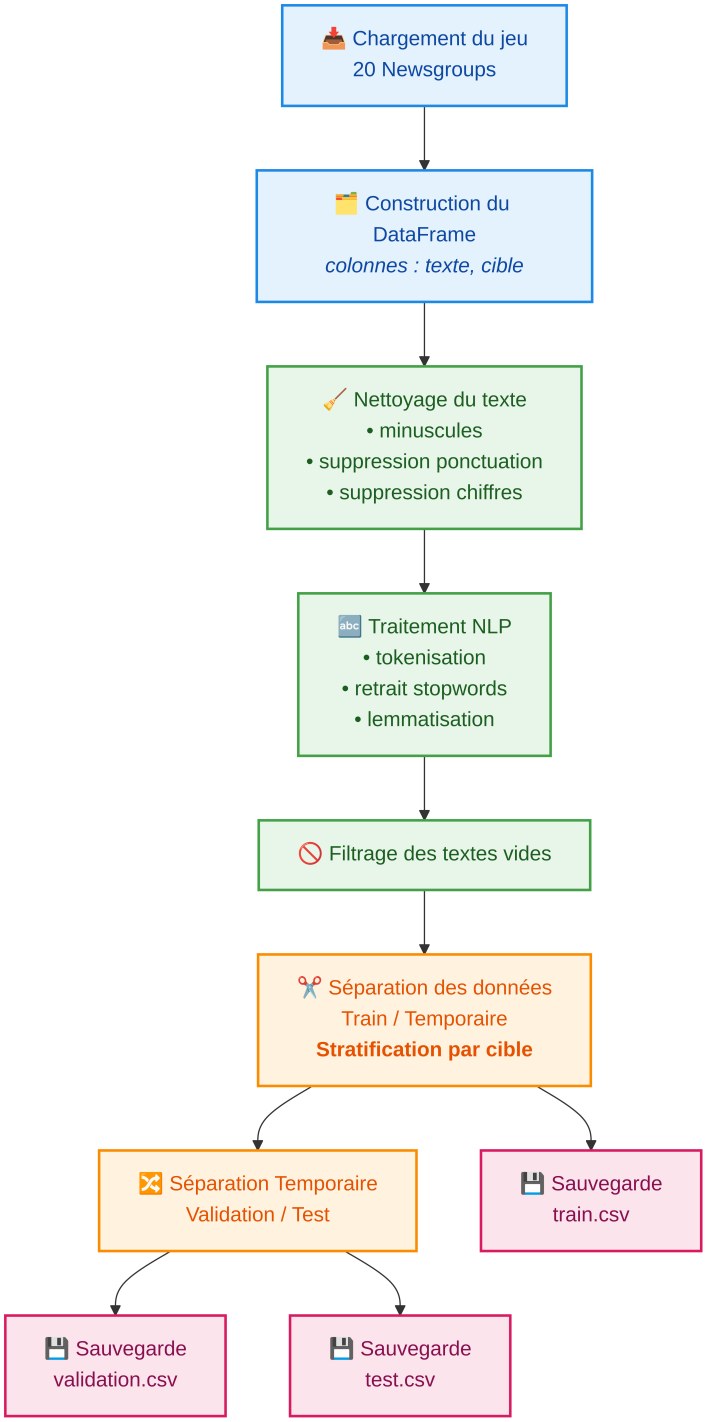


FIGURE 1 – Diagramme du pipeline de données

4.2 Modélisation et entraînement

Le script `src/train.py` orchestre la partie *training* :

— Chargement des fichiers `train.csv` et `test.csv` préalablement générés.

— Nettoyage des valeurs manquantes sur la colonne `processed_text`.

— Vectorisation TF–IDF des textes (`TfidfVectorizer` de `scikit-learn`) avec un nombre maximal de caractéristiques fixé initialement à 5000, puis ajustable (par exemple 10000).

— Entraînement d'une forêt aléatoire (`RandomForestClassifier`) avec 100 arbres, `random_state=42` et `n_jobs=-1`.

— Évaluation sur le jeu de test avec les métriques : accuracy, précision, rappel et F1 pondérés.
— Génération d'un rapport détaillé par classe via `classification_report` et d'une matrice de confusion.
— Sauvegarde locale des métriques (`metrics.json`), du rapport texte (`classification_report.txt`) et de la matrice de confusion (`confusion_matrix.png`).
— Sauvegarde des artefacts de modèle : `model.joblib` et `tfidf_vectorizer.joblib` dans `models/`.

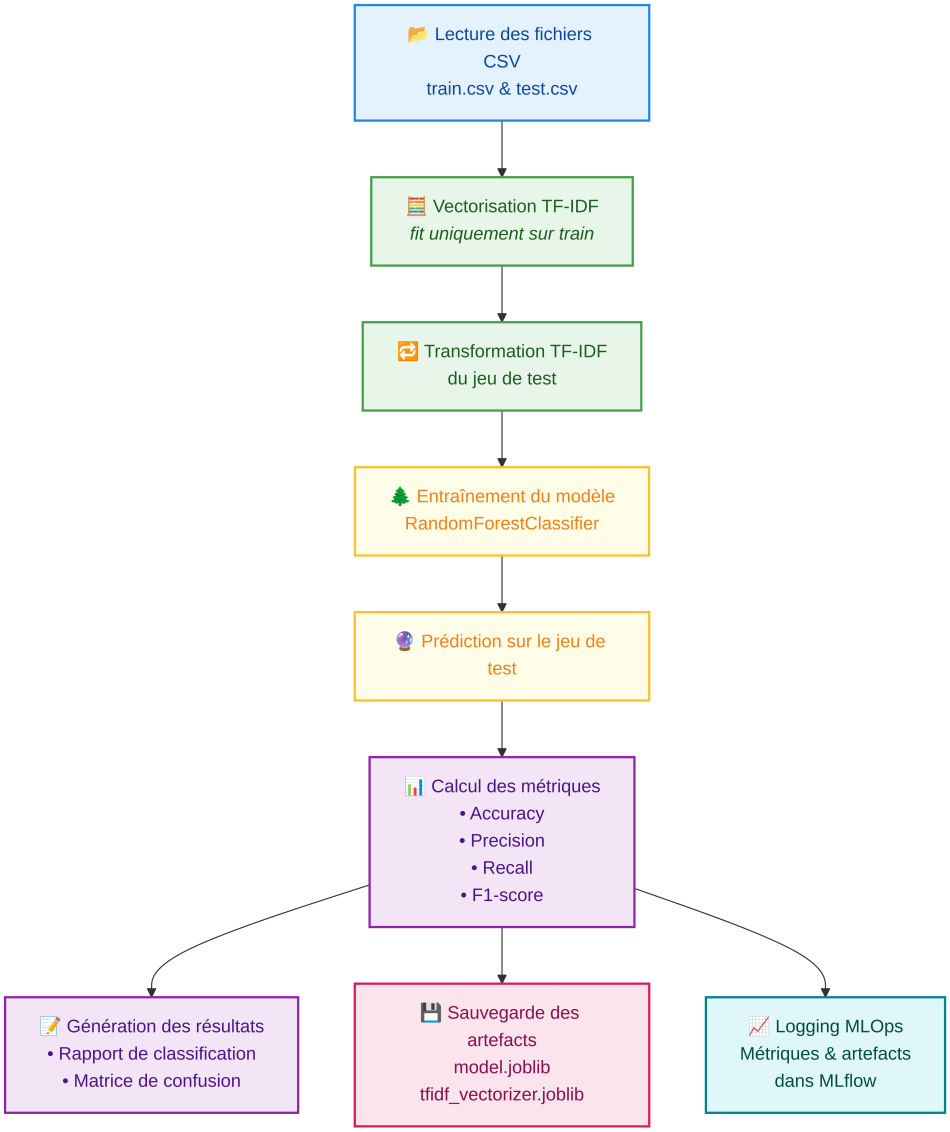


FIGURE 2 – Diagramme du pipeline d'entraînement et d'évaluation du modèle

4.3 Évaluation et validation du modèle

Les performances du modèle sont évaluées principalement à l'aide de l'accuracy et de la F1-score pondérée. La matrice de confusion permet d'analyser les confusions entre classes. Ces métriques servent de base pour :

— comparer différentes configurations d'hyperparamètres (par exemple la valeur de `max_features`)
— définir des seuils minimums de performance à respecter avant tout déploiement en production
— alimenter les rapports CML générés automatiquement dans les pull requests.

5 Suivi expérimental et gestion des modèles

5.1 Tracking des expériences avec MLflow

L'ensemble de l'expérience est instrumenté avec `MLflow` :

— `mlflow.set_tracking_uri("file:./mlruns")` configure un backend local pour le stockage des runs.
— `mlflow.set_experiment("Text_Classification_Projet9")` permet de regrouper tous les essais liés à ce projet.
— Les hyperparamètres (`max_features`, `n_estimators`, etc.), les métriques d'évaluation, la matrice de confusion et les artefacts (`model`, `tfidf_vectorizer.joblib`) sont systématiquement loggés.

L'interface MLflow UI permet de comparer rapidement plusieurs runs, de suivre l'évolution des performances et de sélectionner une configuration de modèle à promouvoir.

5.2 Gestion des artefacts

Les artefacts générés par le pipeline (modèle sérialisé, vectoriseur TF–IDF, métriques, rapports, visualisations) sont :

— sauvegardés dans des répertoires dédiés du dépôt (`models/`, `reports/`)
— loggés dans MLflow pour conserver un historique versionné
— consommés par les workflows CI/CD (par exemple pour générer les rapports CML).

Cette organisation facilite la reprise d'expériences antérieures et simplifie le déploiement du modèle sélectionné.

6 Service d'inférence et packaging

6.1 API FastAPI

Le service de prédiction est implémenté dans `src/app.py` avec FastAPI :

— utilisation d'un gestionnaire de cycle de vie (`lifespan`) pour charger les artefacts au démarrage de l'application : modèle, vectoriseur TF–IDF, stopwords et lemmatiseur NLTK
— endpoint `GET /health` qui retourne un statut `ok` et indique si le modèle est correctement chargé
— endpoint `POST /predict` acceptant un payload JSON contenant un champ `text` ; le texte est prétraité avec les mêmes fonctions que celles utilisées pour l'entraînement, vectorisé, puis passé au modèle
— la réponse contient le texte d'origine, l'identifiant de classe prédit (`prediction_class_id`) et un champ `status`.

Un script `src/predict.py` fournit également une interface en ligne de commande pour charger les artefacts et réaliser une prédiction sur un exemple de texte.

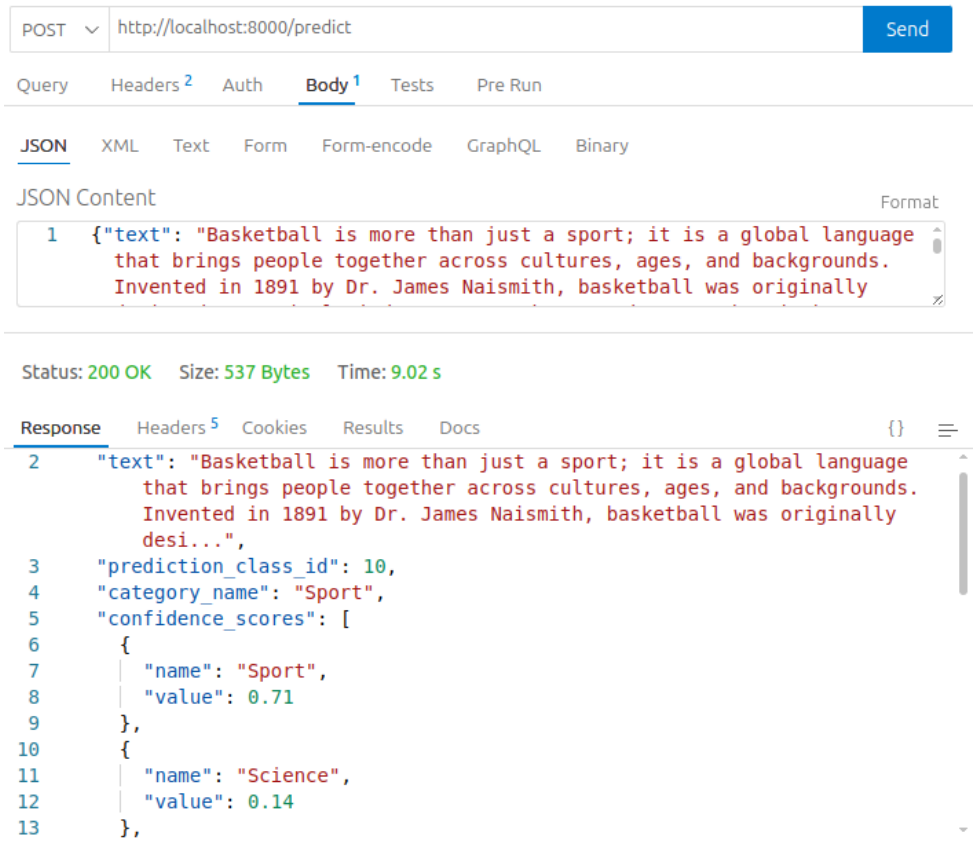


FIGURE 3 – Exemple de requête/réponse de l'API FastAPI

6.2 Containerisation Docker

Le `Dockerfile` définit l'image de service pour l'API :

— utilisation de l'image de base `python:3.9-slim`
— installation des dépendances Python via `requirements.txt`
— téléchargement des ressources NLTK nécessaires en amont pour éviter les téléchargements au runtime
— copie du code source, des artefacts de modèle et des données prétraitées dans l'image
— exposition du port 8000 et lancement de l'application via Uvicorn (ASGI).

Cette containerisation garantit un environnement reproductible pour exécuter le service d'inférence sur différentes machines ou plateformes de déploiement.

7 Stratégie de tests et validation

7.1 Tests unitaires et d'intégration

Le répertoire `tests/` contient des tests couvrant les points suivants :

— fonctions de prétraitement : nettoyage du texte, tokenisation, suppression des stopwords, lemmatisation
— scripts de pipeline : vérification de la génération des fichiers `train.csv`, `validation.csv`, `test.csv`, présence des artefacts de modèle
— service d'API : tests des endpoints `/health` et `/predict` (statuts HTTP, structure des réponses, gestion d'entrées invalides).

Ces tests sont exécutés automatiquement dans les workflows CI avant la construction de l'image Docker.

7.2 Validation de modèle

Au-delà des tests techniques, la validation du modèle s'appuie sur :
— des métriques quantitatives loggées dans MLflow (accuracy, F1 pondérée, etc.)
— des artefacts d'analyse (matrice de confusion, rapport de classification)
— un seuil d'accuracy configuré dans le workflow de déploiement (`MIN_ACCURACY`) qui conditionne la promotion du modèle en production.

Cette validation automatisée permet de réduire le risque de déployer un modèle dégradé.

8 Pipelines CI/CD et déploiement

8.1 Build, tests et image Docker

Le workflow `.github/workflows/docker.yaml` implémente une pipeline de CI centrée sur la construction de l'image de service :
— déclenchement sur `push` et `pull_request` vers la branche `master`
— installation de Python 3.9 et des dépendances à partir de `requirements.txt`
— exécution séquentielle de `src/preprocess.py` puis `src/train.py` afin de régénérer les données prétraitées et les artefacts de modèle
— lancement de la suite de tests `pytest tests/ -v`
— connexion au registre GitHub Container Registry (`ghcr.io`) et construction de l'image Docker à partir du `Dockerfile`, avec publication optionnelle des tags (`latest`, `sha`) sur les branches non-`pull_request`.

Le `Dockerfile` correspondant utilise `python:3.9-slim` comme base, installe les dépendances, télécharge les ressources NLTK nécessaires, copie les sources, les artefacts et les données prétraitées, expose le port 8000 et lance l'application via Uvicorn.

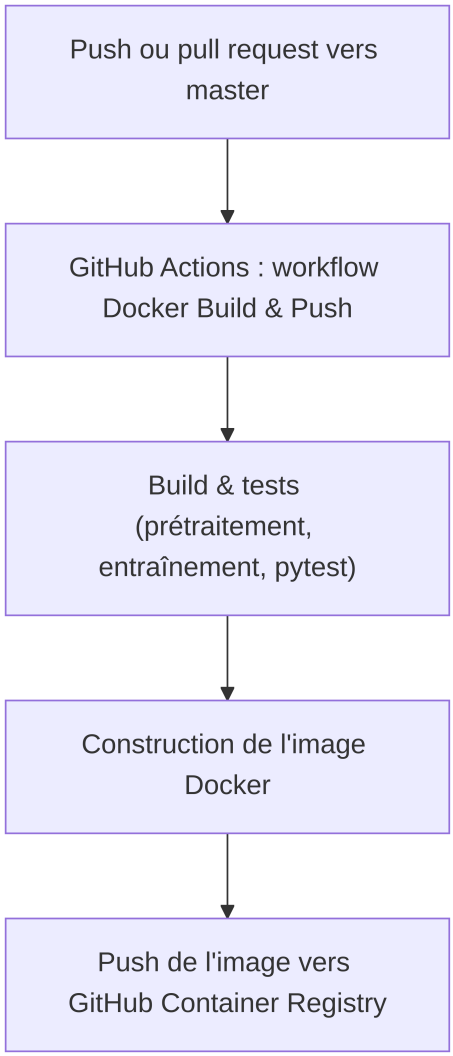


FIGURE 4 – Diagramme du workflow de build, tests et construction/push de l'image Docker

8.2 Rapport automatique avec CML

Le workflow `.github/workflows/cml.yaml` s'appuie sur `iterative/setup-cml` pour générer un rapport expérimental automatique :

- installation des dépendances, exécution du pipeline de prétraitement et d'entraînement
- création d'un fichier `report.md` contenant :
  - un titre
  - les métriques au format JSON (`metrics.json`)
  - le rapport de classification détaillé (`classification_report.txt`)
  - la matrice de confusion publiée comme image via `cml publish`.
- publication automatique du rapport en commentaire sur la pull request via `cml comment create`.

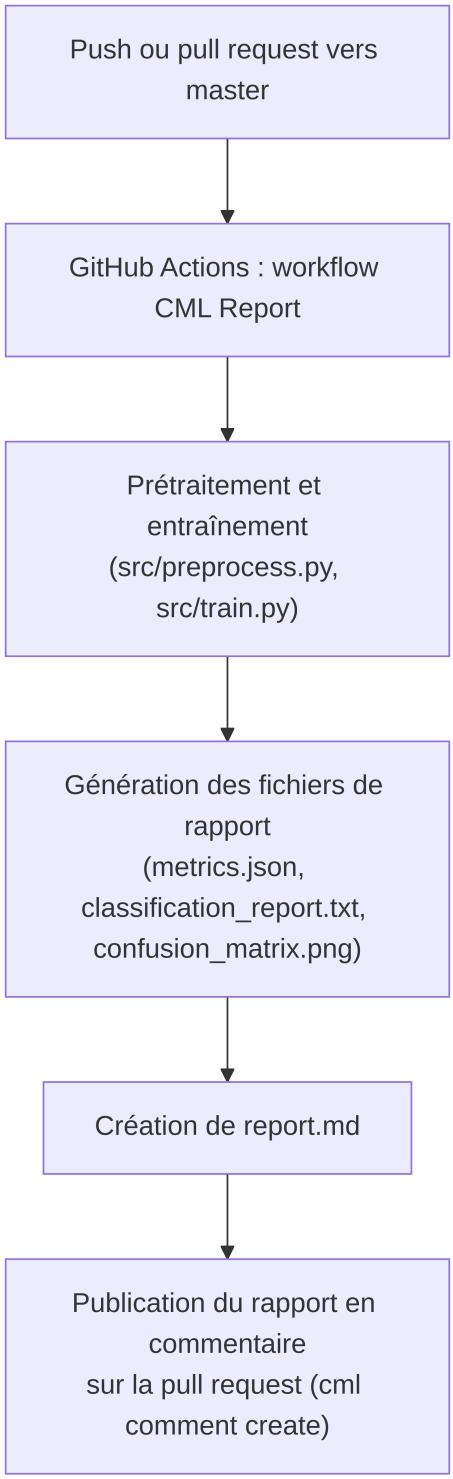


FIGURE 5 – Diagramme du workflow de génération automatique du rapport CML

8.3 Déploiement staging → production et rollback

Le workflow `.github/workflows/deploy.yaml` met en place une logique de déploiement en deux phases :

— il se déclenche à la complétion du workflow « Docker Build & Push » sur la branche `master`

— job **deploy-staging** :
  — récupère la dernière image Docker depuis le registre
  — lance un conteneur en environnement *staging* sur le port 8000
  — effectue des tests d'intégration simples via `curl` sur les endpoints `/health` et `/predict`
  — vérifie que le modèle est bien chargé (`model_loaded=true`) et fixe une accuracy simulée (0,64) pour illustrer la validation.
— job **deploy-production** (conditionné par le succès du *staging*) :
  — vérifie que l'accuracy renvoyée dépasse un seuil minimum configurable (`MIN_ACCURACY`, par défaut 0,5), en utilisant `bc` pour la comparaison flottante
  — « déploie » l'image en production (placeholder avec traces dans les logs et possibilités futures de déploiement Kubernetes ou SSH)
  — marque la version comme release de production (étapes de tagging commentées mais prévues).
— job **rollback** :
  — se déclenche en cas d'échec du *staging* ou de la production
  — journalise les actions de rollback prévues (restauration d'une version précédente, notifications, sauvegarde des logs)
  — crée automatiquement un ticket GitHub décrivant le rollback.

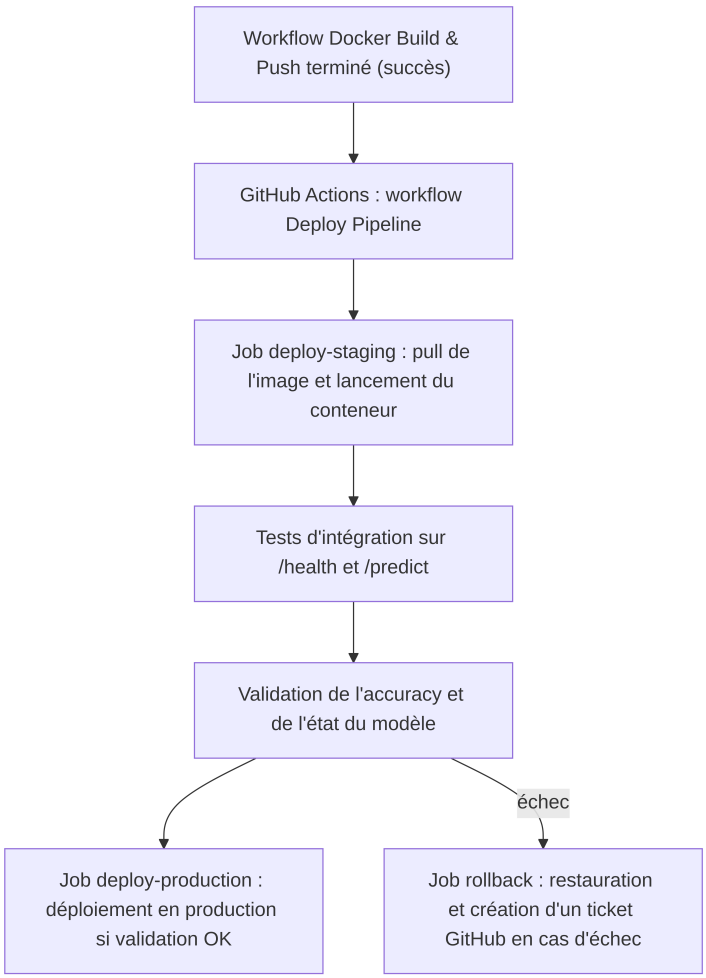


FIGURE 6 – Diagramme du workflow de déploiement staging → production avec rollback

9 Monitoring et exploitation

Le monitoring actuel repose principalement sur :

— les endpoints `/health` et `/predict`, utilisés dans les tests d'intégration et le workflow de déploiement pour vérifier que le service est opérationnel et que le modèle est chargé
— les métriques et artefacts loggés dans MLflow, permettant un suivi des performances au fil des expériences
— les rapports CML générés dans les pull requests, qui résument les performances du dernier entraînement.

En l'état, il n'existe pas encore de monitoring runtime complet (collecte de métriques en production, détection de dérive de données, alertes automatiques). Ces aspects sont identifiés comme des axes d'amélioration majeurs.

10 Limites et axes d'amélioration

10.1 Limites du modèle et des données

— **Modèle classique uniquement** : l'utilisation d'un Random Forest sur des features TF–IDF est simple mais ne profite pas des avancées récentes en NLP (transformers, embeddings contextuels, etc.).
— **Pas de tuning systématique** : les hyperparamètres (`max_features`, `n_estimators`) sont fixés en dur, sans recherche automatisée (Grid Search, Random Search ou optimisation bayésienne).
— **Prétraitement basique** : pas de prise en compte d'éléments comme les bigrammes/trigrammes, la détection de langue, la gestion des caractères spéciaux ou des émojis.
— **Gestion partielle des labels** : la correspondance entre identifiant de classe et nom de catégorie n'est pas exposée dans l'API, ce qui complique l'interprétation métier.

10.2 Limites MLOps / CI/CD

— **Tracking local uniquement** : MLflow est configuré en mode fichier local, ce qui limite l'utilisation en environnement distribué ou multi-utilisateurs.
— **Absence de véritable registry de modèles** : il n'existe pas encore de gestion de versions de modèles avec promotion formelle (staging/production) au niveau du *model registry*.
— **Couplage fort pipeline–build** : le workflow de build exécute systématiquement prétraitement et entraînement avant les tests, ce qui rallonge les temps de CI et peut être coûteux.
— **Seuil de performance codé en dur** : l'accuracy utilisée dans le déploiement est actuellement simulée (valeur fixe 0,64), ce qui réduit le réalisme de la validation automatisée.
— **Monitoring runtime absent** : pas de collecte de logs métier, de métriques en production, ni de détection de dérive de données ou de performance.

10.3 Sécurité, robustesse et industrialisation

— **Gestion des ressources NLTK** : le téléchargement des ressources NLTK au démarrage (dans `app.py`) peut être fragile en contexte de production (dépendance réseau) même si le Dockerfile anticipe déjà certains téléchargements.
— **Tests limités** : les tests couvrent les points clés (API, prétraitement, artefacts) mais ne couvrent pas tous les cas d'erreur (entrées invalides, timeouts, modèles manquants, etc.).
— **Configuration statique** : les chemins, hyperparamètres et URI MLflow sont codés dans le code une externalisation via des variables d'environnement ou des fichiers de configuration renforcerait la flexibilité.

— **Déploiement simplifié** : le job de déploiement est illustratif (logs, placeholders pour Kubernetes ou SSH) sans intégration à une infrastructure réelle.

11 Conclusion

Le projet illustre un pipeline MLOps cohérent de bout en bout pour la classification de texte : préparation des données, entraînement, suivi expérimental, exposition d'une API, containerisation et automatisation CI/CD avec rapport CML et logique de déploiement *staging* → *production*. Les choix techniques privilégient la simplicité et la pédagogie (Random Forest + TF–IDF, MLflow local, GitHub Actions, CML) et offrent une base solide pour explorer les bonnes pratiques MLOps.

Les principales pistes d'amélioration concernent la modernisation du modèle (transformers), la mise en place d'un véritable registry et d'un tracking centralisé, l'enrichissement des tests et du monitoring en production.