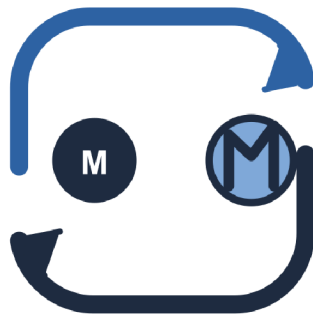


MLOps et Industrialisation des modèles IA

TP 2

Tester et accepter un modèle

Du modèle au système ML traçable, validé et exploitable

**MLOPS**
GL-M326

TP · SOMMAIRE

Table des matières

Mission et question directrice	3
Compétences et ressources	3
Consignes de travail	3
Installer les outils et diagnostiquer l'environnement	3
Préparation	4
Ce qui est nouveau dans le TP2	5
Contrat de progression TP1 → TP2 → TP3	5
Parcours guidé et preuves	6
1. Cartographier les contrôles	6
2. Écrire les tests logiciels	7
3. Renforcer la validation des données	7
4. Établir la baseline et la policy	8
5. Implémenter le quality gate	8
6. Intégrer la décision au pipeline	8
7. Jouer deux scénarios	9
8. Vérifier l'intégration de remise	9
Questions techniques et ouvertes	9
Compte rendu obligatoire et preuves visuelles	10
Minimum requis	10
Création et innovation	11
Livrable et réussite	11

Modalité : travail individuel. Chaque étudiant réalise les manipulations, documente ses propres résultats et remet son propre compte rendu et son archive.

Mission et question directrice

Le pipeline du TP1 est reproductible, mais **reproductible ne signifie pas acceptable**. Un dataset invalide peut entrer dans la chaîne et un modèle peut améliorer l'accuracy tout en dégradant le rappel des pannes.

Comment transformer des tests, des données et des métriques en une décision **ACCEPT ou **REJECT** explicable et traçable ?**

Compétences et ressources

Vous allez reprendre le livrable produit au TP1 pour ajouter les tests logiciels, la validation des données, une baseline, une politique de qualité et la décision dans DVC. Le livrable TP1 est le point d'entrée principal : conservez sa configuration, son modèle, ses métriques, son manifeste et son run MLflow. Le kit [student/](#) est une référence de secours lorsque le livrable TP1 n'est pas disponible ; il reprend la même expérience pour éviter de reconstruire un projet différent.

Le starter implémente le contrat de données avec Pandera sur un DataFrame pandas. Dans votre copie, vous pouvez conserver Pandera ou choisir Great Expectations ou Soda, à condition de vérifier les mêmes règles, de retourner le même contrat (`status`, `rows`, `checks`, `errors`) et de produire `reports/data_validation.json`. Deequ, Spark, Docker, FastAPI et CI/CD sont hors périmètre du TP2.

Consignes de travail

Travaillez dans une copie personnelle `TP2_NOM_PRENOM` et conservez un état propre entre deux scénarios. Pour chaque contrôle, indiquez explicitement l'entrée testée, la commande, le résultat observé, le fichier produit, la règle appliquée et la preuve conservée dans `reports/figures/`.

Un test doit échouer pour une raison maîtrisée et non parce que le workspace est incohérent. Après chaque scénario négatif, restaurez les fichiers et vérifiez que le scénario nominal fonctionne de nouveau.

Installer les outils et diagnostiquer l'environnement

Le TP requiert Python 3.10 ou plus récent, Git, `uv`, DVC, `pytest` et MLflow. Vérifiez :

TERMINAL · BASH

```
python --version
git --version
uv --version
make --version
```

Si `uv` manque sur Linux ou macOS :

TERMINAL · BASH

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Les dépendances sont installées dans l'environnement du kit et verrouillées par `uv.lock`.

Préparation

Téléchargez l'énoncé PDF et l'archive du kit TP2 (.zip) depuis la page du TP. Le ZIP est un **kit de référence exécutable**, utile si le livrable TP1 n'est pas disponible ; il ne remplace pas le travail demandé à partir de votre propre TP1. Conservez le PDF comme référence et extrayez le ZIP dans votre espace de travail personnel, pas dans [Téléchargements](#). L'archive contient directement les fichiers du projet : aucun dossier `student/` n'est à copier.

Linux/macOS, depuis le dossier contenant le ZIP :

TERMINAL · BASH

```
mkdir -p "$HOME/ensias-tps/MLOps/TP2_NOM_PRENOM"
unzip "<nom-exact-du-kit-TP2>.zip" -d "$HOME/ensias-tps/MLOps/TP2_NOM_PRENOM"
cd "$HOME/ensias-tps/MLOps/TP2_NOM_PRENOM"
pwd
ls -la
```

Windows PowerShell :

CODE

```
$work = "$HOME\ensias-tps\MLOps\TP2_NOM_PRENOM"
New-Item -ItemType Directory -Force $work
Expand-Archive -LiteralPath ".\<nom-exact-du-kit-TP2>.zip" -DestinationPath $work
Set-Location $work
Get-ChildItem
```

Vérifiez la présence de `pyproject.toml`, `uv.lock`, `Makefile`, `data/` et `README.md`, puis exécutez les commandes suivantes depuis cette racine. Si vous partez du livrable TP1, vérifiez également que le modèle, les métriques et le manifeste ont été conservés.

TERMINAL · BASH

```
make install
make report
make doctor
make dvc-init
make validate-data
make pipeline
```

`make doctor` vérifie DVC, `pytest` et MLflow après l'installation. Si le réseau est indisponible, essayez `uv sync --locked --offline` si le cache local existe et consignez l'erreur dans le rapport.

Lancez MLflow si votre pipeline le requiert. Avant toute modification, exécutez le starter et notez les sorties produites. Vérifiez que le pipeline du TP1 est rejouable avant d'ajouter les contrôles de qualité.

Ce qui est nouveau dans le TP2

Le TP1 a fourni la configuration, le modèle, les métriques, le manifeste et la traçabilité MLflow. Le TP2 ajoute progressivement :

- un contrat de données, avec Pandera, Great Expectations ou Soda ;
- les tests logiciels avec `pytest` ;
- DVC et les stages `validate`, `prepare`, `train`, `evaluate`, `quality_gate` ;
- la baseline et les seuils paramétrables ;
- le quality gate et les décisions `ACCEPT` / `REJECT` ;
- le scénario valide et le scénario `machines_invalid.csv`.

Contrat de progression TP1 → TP2 → TP3

Le TP2 consomme les sorties du TP1 : configuration utilisée, modèle candidat, métriques, manifeste et run MLflow. Il ajoute la décision d'acceptation ; il ne remplace pas l'analyse de reproductibilité du TP1. Le manifeste TP2 doit conserver le hash des données et le `parent_run_id` du TP1 lorsqu'il est disponible. Dans une copie réelle du TP1, conservez son manifeste dans `models/run_manifest.json` avant de lancer le pipeline TP2 ; c'est ce fichier qui sert de parent, jamais le manifeste TP2 courant.

À la fin du TP2, remettez au minimum `dvc.yaml`, `dvc.lock`, la configuration héritée du TP1, les rapports de validation et d'évaluation, `reports/run_manifest.json`, `reports/quality_gate_decision.json`, ainsi que les scénarios `ACCEPT` et `REJECT`. Le TP3 utilisera le modèle accepté et ce JSON comme précondition avant de l'exposer par API. `make validate-submission` est le contrôle d'acceptation final.

Parcours guidé et preuves

1. Cartographier les contrôles

Le tableau de l'énoncé est une grille de départ : il ne doit pas être rempli dans ce `README.md`. Dans votre copie de travail, exécutez d'abord `make report`, puis complétez la section `## 6. Matrice des contrôles` du fichier `reports/tp2_report.md`. Si vous partez de votre propre TP1, créez ce fichier à partir de `student/reports/tp2_report_template.md`.

Pour chaque ligne, remplacez les cellules vides par des éléments observables de votre projet :

- **Entrée** : chemin relatif du fichier, du dataset ou du manifeste lu ;
- **Règle** : assertion ou seuil réellement vérifié ;
- **Commande ou test** : commande exacte (`make test`, `make pipeline`, etc.) ou nom du test ;
- **Sortie** : fichier produit et résultat observé (`N passed`, `PASS`, `ACCEPT` ou `REJECT`) ;
- **Action en cas d'échec** : arrêt, rejet, correction ou revue humaine.

Vous devez citer au moins un chemin et une preuve d'exécution par ligne. Les chemins ci-dessous sont des points de départ à vérifier dans le starter ; ils ne remplacent pas les résultats obtenus dans votre copie :

Niveau	Question vérifiée	Fichier/test à examiner	Moment	Échec
Code	Le module s'importe-t-il et les comportements sont-ils testés ?	<code>maintenance_ml/</code> , <code>tests/</code> , <code>make test</code>	avant pipeline	arrêt
Données	Le schéma et les contraintes métier sont-ils valides ?	<code>maintenance_ml/data/validate.py</code> , <code>reports/data_validation.json</code>	avant split	rejet
Modèle	Les métriques respectent-elles la baseline et les seuils ?	<code>reports/metrics.json</code> , <code>quality_gates/</code> , <code>quality_gate.py</code>	après évaluation	<code>REJECT</code>
Opérations	Le candidat est-il traçable et remissible ?	<code>dvc.yaml</code> , <code>dvc.lock</code> , <code>reports/run_manifest.json</code> , <code>make validate-submission</code>	avant promotion	revue ou rejet

Expliquez pourquoi un test vert ne suffit pas à accepter un modèle.

Livrable attendu : le tableau rempli se trouve dans `reports/tp2_report.md`, et non dans le code source ni dans le PDF. Ajoutez une capture ou un extrait de chaque preuve dans `reports/figures/`. Si un contrôle n'existe pas encore, écrivez `à implémenter`, indiquez l'emplacement prévu, la commande qui devra l'exécuter et la sortie attendue ; ne remplacez pas cette absence par un simple `OK`.

2. Écrire les tests logiciels

Ajoutez au moins six tests : import/exécution, colonnes attendues, valeurs admissibles, fichier de sortie, métriques présentes et décision structurée.

TERMINAL · BASH

```
make test
```

Provoquez ensuite une erreur contrôlée, conservez la sortie utile et remettez le workspace dans un état propre.

Preuve intermédiaire : rapport de test vert, rapport de test rouge et explication de l'étape bloquante.

Consignes : chaque test doit vérifier un comportement observable, pas une implémentation interne. N'utilisez pas de test qui dépend d'un serveur ou d'un fichier créé par un test précédent sans le préparer explicitement.

3. Renforcer la validation des données

Le validateur doit contrôler :

- colonnes attendues et types ;
- absence de valeurs manquantes ;
- `machine_type ∈ {A,B,C}` ;
- `tool_wear_min ≥ 0` ;
- `machine_failure ∈ {0,1}` ;
- volume minimal et taux de panne documentés.

Le dataset de référence est celui du TP1 : 5 000 lignes, dont 200 pannes. La baseline fournie dans le starter correspond à une exécution de référence de ce dataset ; dans un livrable personnel, remplacez-la par les métriques réellement produites par votre TP1.

Testez `machines_invalid.csv`. Le pipeline doit échouer avant `prepare`, `train` et `evaluate`.

Preuve attendue : la sortie d'erreur identifie la règle violée et montre que les artefacts d'entraînement n'ont pas été régénérés. Comparez au moins un fichier valide et `machines_invalid.csv`.

4. Établir la baseline et la policy

Documentez la baseline dans `quality_gates/baseline_metrics.json`. Définissez dans `params.yaml` des seuils explicites pour F1, recall, précision, écart à la baseline et, si pertinent, par segment.

Exemple de décision :

CODE

```
{
  "decision": "ACCEPT",
  "model_version": "candidate-local",
  "checks": [],
  "reasons": []
}
```

Justifiez chaque seuil par le contexte de maintenance prédictive. Un seuil doit être modifiable sans changer la logique du code.

5. Implémenter le quality gate

Créez ou complétez `maintenance_ml/modeling/quality_gate.py`. Le programme doit :

1. lire les métriques et la baseline ;
2. lire les seuils ;
3. produire toutes les règles évaluées ;
4. expliquer les règles échouées ;
5. retourner un code de sortie cohérent ;
6. écrire `reports/quality_gate_decision.json`.

Ne codez jamais `ACCEPT` ou `REJECT` en dur.

Consignes : évaluez toutes les règles, même après un premier échec, afin que la décision soit diagnostiquable. Le code de sortie doit être non nul pour un rejet et le JSON doit rester lisible lorsqu'une règle échoue.

6. Intégrer la décision au pipeline

Faites dépendre la décision des sorties réellement produites. `make pipeline` commence par exécuter `make test` et ne lance DVC que si les tests sont verts :

TERMINAL · BASH

```
make pipeline
```

Vérifiez que le pipeline s'arrête lorsqu'une règle critique échoue. Comparez le graphe :

CODE

```
tests → validation → prepare → train → evaluate → quality gate
      |
      |— données invalides : arrêt
      |
      |— ACCEPT
      |— REJECT
```

7. Jouer deux scénarios

Produisez un scénario accepté et un scénario rejeté, par exemple en modifiant un seuil, une configuration ou le dataset de test. Dans chaque cas, conservez les métriques, le JSON de décision et la commande exacte.

Pour démontrer le rejet des données avant le split, vous pouvez exécuter :

TERMINAL · BASH

```
PYTHONPATH=. uv run --locked python -m maintenance_ml.data.validate \
--data data/external/machines_invalid.csv
```

Pour démontrer l'arrêt DVC complet, remplacez temporairement `data/raw/machines.csv` par ce dataset invalide, lancez `make pipeline`, conservez le code de sortie non nul, puis restaurez le dataset valide avant de rejouer le scénario `ACCEPT`. Ne mélangez jamais les artefacts des deux états.

Consigne : ne mélangez pas les artefacts des deux scénarios. Utilisez des sous-dossiers ou des noms explicites, par exemple `reports/scenarios/accept/` et `reports/scenarios/reject/`.

8. Vérifier l'intégration de remise

Exécutez `make validate-submission` après `make pipeline`, après avoir rempli `reports/tp2_report.md` et conservé les deux scénarios. Le contrôle vérifie les fichiers produits, le rapport JUnit de `make test`, la structure des décisions, les sections du rapport, les réponses Q1 à Q7, les huit preuves visuelles et les artefacts des scénarios. Créez ensuite la remise avec :

TERMINAL · BASH

```
make package ARCHIVE=MLOps_TP2_Nom_Prenom.zip
```

La commande rejoue le contrôle final avant de produire l'archive. Un test vert isolé ne valide donc pas la chaîne complète et ses artefacts transmissibles au TP3.

Questions techniques et ouvertes

1. Quelle différence observez-vous entre un test logiciel et un quality gate ML ?
2. Pourquoi une donnée invalide doit-elle être rejetée avant le split ?
3. Comment les seuils changeraient-ils si le coût d'un faux négatif augmentait ?

4. Analysez un scénario où l'accuracy augmente mais où le recall devient inacceptable.
5. Retraced un rejet de bout en bout et indiquez la preuve manquante avant correction.
6. Concevez une règle par segment et discutez le risque d'une fragmentation excessive.
7. Qui devrait avoir le dernier mot lorsqu'une règle technique et une contrainte métier se contredisent ?

Compte rendu obligatoire et preuves visuelles

Le compte rendu individuel doit contenir :

- les commandes exécutées et leur contexte ;
- les résultats des tests et des quality gates ;
- les fichiers JSON produits ;
- les réponses à toutes les questions ;
- une interprétation séparant faits, hypothèses et décisions.

Les preuves visuelles doivent respecter les règles suivantes :

- placer les fichiers dans `reports/figures/` ;
- utiliser des noms explicites, par exemple `tp2-03-gate-reject.png` ;
- afficher la commande, la sortie et le contexte utile ;
- ajouter une légende et expliquer ce que la capture démontre ;
- fournir au minimum :
 - installation et `make doctor` ;
 - tests verts ;
 - test volontairement cassé ;
 - dataset invalide ;
 - baseline et seuils ;
 - scénario `ACCEPT` ;
 - scénario `REJECT` ;
 - arrêt du pipeline.

Les captures téléchargées, fabriquées ou dépourvues de contexte ne sont pas recevables.

Minimum requis

Le travail est recevable si les éléments suivants sont présents :

- au moins six tests ;
- validation du dataset invalide ;
- baseline documentée ;
- policy paramétrable ;

- décision **ACCEPT** ;
- décision **REJECT** ;
- intégration dans le pipeline ;
- rapport complet avec preuves.

Checklist avant remise : le scénario accepté est reproductible, le scénario rejeté est explicable, le code de sortie est cohérent avec la décision et aucun seuil métier n'est caché dans le code Python.

Création et innovation

Choisissez une extension :

- quality gate par type de machine ;
- seuils adaptatifs ;
- rapport HTML ;
- test de robustesse aux perturbations ;
- analyse de calibration ;
- politique multi-objectifs.

Présentez le besoin, la règle ou le fichier ajouté, un scénario de test, le gain attendu, le coût et le risque.

Livrable et réussite

Avant la remise, vérifiez que l'archive contient :

- code des tests et de la validation ;
- paramètres et seuils ;
- baseline ;
- scénarios **ACCEPT** et **REJECT** ;
- JSON de décision ;
- rapport JUnit **reports/test_report.xml** ;
- rapport et figures ;
- README de reproduction.

Créez et contrôlez l'archive depuis la racine du starter avec :

TERMINAL · BASH

```
make package ARCHIVE=MLOps_TP2_Nom_Prenom.zip
```

La commande relance la validation finale avant de produire l'archive. Elle inclut les artefacts générés nécessaires à la correction, y compris ceux qui sont ignorés par Git.



Le rapport doit distinguer correction technique, validité des données, qualité prédictive et décision d'acceptation.

Le nom exact de l'archive est `MLOps_TP2_Nom_Prenom.zip`.

