



GL-M326 · LECTURE ACADÉMIQUE

MLOps et Industrialisation des modèles IA

Du modèle au système ML traçable, validé et exploitable



MLOPS
GL-M326

SÉANCE 3 / 6 **Tester et valider avant la promotion**

ENSEIGNANT

Pr. Youness BOUTYOUR

youness.boutyour@um5.ac.ma

PUBLIC CIBLE

Génie Logiciel

Cycle Ingénieur — 3A

INSTITUTION

Université Mohammed V de Rabat

École Nationale Supérieure d'Informatique et d'Analyse des
Systèmes

Plan de la séance

SÉANCE 3 / 6 – TESTER ET VALIDER AVANT LA PROMOTION

01 Cadrage

02 Contrôler avant d'entraîner

03 Décider avant de promouvoir

04 Relier preuves et outils

05 Éprouver la décision

06 Synthèse

Objectifs pédagogiques

OBJECTIFS PÉDAGOGIQUES

7 OBJECTIFS

- 01 Distinguer software tests, data tests et model tests dans un pipeline ML
- 02 Formaliser un data contract et produire un rapport de validation interprétable
- 03 Définir une baseline, des métriques, des seuils et des contrôles par segment
- 04 Comparer un challenger au champion sans réduire la décision à une métrique unique
- 05 Implémenter un Quality Gate auditable qui produit une décision `ACCEPT` ou `REJECT`
- 06 Évaluer la robustesse, les segments critiques et les contraintes opérationnelles avant promotion
- 07 Justifier une décision `ACCEPT`, `REJECT` ou `REVIEW` à partir de preuves explicites

CHAPITRE

Tester et valider avant la promotion

01 Cadrage

La question directrice



PROBLÈME DE LA SÉANCE

Comment décider automatiquement qu'un modèle peut poursuivre son chemin vers la production ?

À la fin de cette séance, vous saurez :

- distinguer **software tests**, **data tests** et **model tests** ;
- expliquer pourquoi un pipeline reproductible n'est pas nécessairement fiable ;
- écrire des validations de données simples ;
- définir une baseline et des seuils d'acceptation ;
- comprendre la logique **champion / challenger** ;
- implémenter un **quality gate** qui accepte ou rejette un modèle candidat.

Principes MLOps ciblés

Dans cette séance, chaque contrôle doit laisser une preuve lisible et déclencher une action explicite.

PRINCIPE	CE QUE L'ON MET EN PLACE	PREUVE OBTENUE
P1 — Tracer	Relier lot, métriques, paramètres et décision.	Rapports JSON + <code>run_manifest.json</code>
P2 — Versionner	Formaliser le contrat, la baseline et la politique.	Contrat + baseline + paramètres versionnés
P3 — Valider	Contrôler code, données et modèle avant l'étape suivante.	Tests + <code>PASS</code> / <code>FAIL</code>
P7 — Décider	Comparer le candidat aux seuils et à la baseline.	Quality gate + <code>ACCEPT</code> / <code>REJECT</code> + code retour

État du projet au début

Nous partons de l'état final de la séance 2 : un pipeline que l'on peut rejouer.

SÉANCE 2 • ACQUIS

01

Reproductible

Même code, données et paramètres → même expérience rejouable.



SÉANCE 3 • OBJECTIF

02

Fiable

Données contrôlées, métriques comparables et décision justifiable.



SÉANCE 4 • SUITE

03

Déployable

Contrat d'inférence, service et exploitation prêts pour l'usage réel.

Cette séance ajoute la deuxième propriété : décider si le candidat peut avancer, avant de traiter le serving en séance 4.

Les quatre microcycles

Chaque microcycle répond à une question et laisse une preuve réutilisable.

MC1 • 25 MIN

Trois niveaux de qualité

Quels tests sont nécessaires pour un système ML ?

Sortie : taxonomie des contrôles

MC2 • 30 MIN

Validation des données

Comment détecter tôt les données invalides ?

Sortie : contrat + rapport

MC3 • 25 MIN

Validation du modèle

Comment comparer un candidat à une référence ?

Sortie : métriques + baseline

MC4 • 30 MIN

Quality gate

Comment transformer les métriques en décision ?

Sortie : ACCEPT / REJECT

110 min de travail guidé + cadrage et transitions · la décision finale réutilise les preuves des trois étapes précédentes.

Le fil rouge de la démonstration

Nous reprenons le projet churn de la séance 2 et ajoutons des barrières observables.



À chaque flèche : une preuve lisible — fichier, métrique, log ou code de sortie — permettant de comprendre et de rejouer la décision.

Starter -> finish

On part d'un pipeline qui s'exécute ; on le transforme en pipeline qui peut justifier sa décision.

DÉPART

Validation minimale

métriques accuracy / F1



RENFORCEMENT

Règles + seuils

recall, precision, baseline



ARRIVÉE

Décision justifiée

rapport + **ACCEPT** / **REJECT**

Le code de la démonstration se trouve dans [code/seance3/](#).

Feuille de route — du starter au finish

01

Contrat

steps/01-contract/

PASS/FAIL + erreurs

02

Tests

steps/02-tests/

règle métier testée

03

Gate

steps/03-gate/

checks nommés

04

Finish

finish/

baseline, DVC, décision

Projeter les fragments dans cet ordre : demander à la classe de prédire la sortie avant d'afficher la version suivante.

CHAPITRE

Tester et valider avant la promotion

02

Contrôler avant d'entraîner

Pourquoi un pipeline exécutable n'est-il pas encore fiable ?

Un pipeline peut **s'exécuter jusqu'au bout** sans produire un modèle fiable.

CE QUE L'EXÉCUTION PROUVE

```
validate_data → prepare → train → evaluate
```

Les étapes se terminent et les artefacts sont produits :

```
model.joblib · metrics.json · run_manifest.json
```

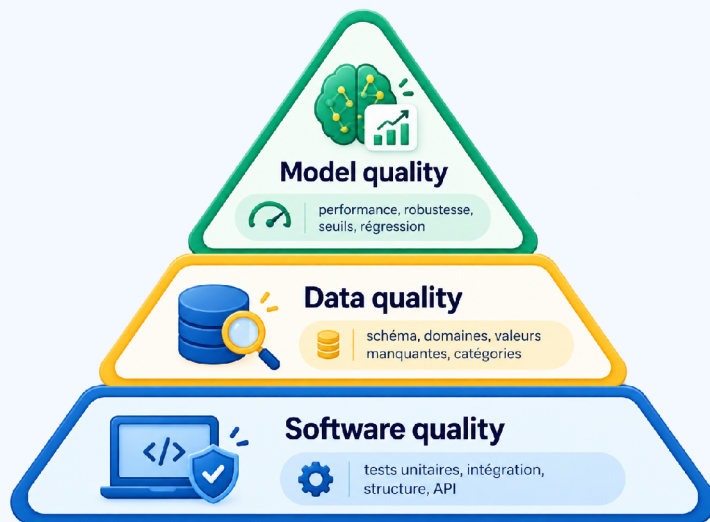
CE QU'ELLE NE PROUVE PAS

- les données respectent le contrat ;
- les métriques atteignent les seuils ;
- le candidat ne régresse pas.

Question : quelles validations faut-il ajouter avant d'autoriser le modèle à poursuivre son chemin ?

Avant de promouvoir : trois preuves complémentaires

Une chaîne ML est acceptable seulement si elle fournit une preuve distincte pour le code, les données et le modèle.



1 · Software quality

Le logiciel respecte ses contrats fonctionnels et ses interfaces.

Preuve : tests unitaires et d'intégration au statut **PASS**.

2 · Data quality

Le lot entrant est conforme au contrat attendu par le pipeline.

Preuve : rapport de validation du schéma, des domaines et de la complétude.

3 · Model quality

Le candidat atteint les seuils et ne régresse pas face à la baseline.

Preuve : métriques comparées, contrôles nommés et décision **ACCEPT** ou **REJECT**.

Règle : une exécution sans exception prouve seulement que le chemin a été parcouru ; la promotion exige les trois preuves.

Software tests

UNE QUESTION : LE CODE RESPECTE-T-IL SON CONTRAT ?



Un test logiciel vérifie le comportement du code, pas la qualité du lot ni du modèle.

DÉFINITION Les **software tests** sont des procédures automatisées qui vérifient, sur des unités isolées ou des interactions contrôlées, que le logiciel satisfait ses contrats fonctionnels.

- Le test répond à une question locale et falsifiable :
 - une fonction retourne le bon type ;
 - le preprocessing conserve le nombre de lignes ;
 - la cible est séparée des features ;
 - les fichiers attendus sont produits.
- **Preuve attendue** : une assertion automatisée et un résultat **PASS** ou **FAIL**.

Data tests

UNE QUESTION : LE LOT EST-IL ACCEPTABLE POUR LE PIPELINE ?



Le data test protège le pipeline contre un lot invalide ; il ne prouve pas sa représentativité.

DÉFINITION Les **data tests** sont des contrôles systématiques qui déterminent si un lot entrant satisfait un contrat de données explicite avant son traitement par le pipeline.

- Le contrat décrit notamment :
 - les colonnes attendues et leurs types ;
 - les domaines de valeurs et les valeurs manquantes ;
 - les catégories connues et la cible binaire.
- **Preuve attendue** : un rapport JSON attaché au lot ; un échec retourne un code non nul et bloque la suite.
- La validation ne prouve pas la représentativité du lot : elle empêche les erreurs évidentes de contaminer le pipeline.

Model tests



Le model test transforme une performance mesurée en décision de promotion.

DÉFINITION Les **model tests** sont des contrôles comparatifs qui établissent si un modèle candidat satisfait des exigences de performance et de non-régression avant toute promotion.

- La comparaison porte sur :
 - une performance minimale ;
 - l'absence de régression par rapport à une baseline ;
 - la stabilité sur sous-groupes ;
 - les contraintes métier et d'inférence.
- **Preuve attendue** : métriques, baseline, checks nommés et décision **ACCEPT** ou **REJECT** .
- Dans le **finish** , les contrôles exécutés portent sur les métriques globales, les seuils et les régressions ; les segments, la latence et les contraintes métier restent des extensions.

Cas pratique — que faire d'un lot invalide ?

Un lot peut être lisible par Python tout en étant **incompatible avec le contrat de données**.

CHAMP	VALEUR INJECTÉE	CONTRAINTE VIOLÉE
age	-10	$18 \leq \text{age} \leq 100$
contract	"weekly"	valeur parmi <code>month-to-month</code> , <code>one-year</code> , <code>two-year</code>
churn	3	valeur binaire : <code>0</code> ou <code>1</code>

QUESTION D'ANALYSE

À quel moment la violation doit-elle être détectée ?

RÉPONSE ATTENDUE

Dans `validate_data`, avant `prepare`, `train` et `evaluate`.

Preuve à collecter : un rapport `FAIL` qui nomme les colonnes en erreur et empêche la production du modèle.

Étape suivante : décrire un lot ne suffit pas ; il faut distinguer le *profilage*, qui observe, de la *validation*, qui décide.

Observer n'est pas encore valider

Le profilage décrit un lot ; la validation le confronte à un **contrat explicite** pour décider de la suite du pipeline.

DÉFINITION Le **profilage des données** est l'opération descriptive qui caractérise un lot par ses types, sa complétude, ses distributions et ses cardinalités, sans conclure à elle seule qu'il est acceptable.

PROFILAGE	VALIDATION
décrit ce qui est observé	vérifie ce qui est acceptable
produit des statistiques du lot	confronte le lot à un contrat versionné
aide à formuler une hypothèse	produit une décision PASS ou FAIL
informe sans autoriser la promotion	bloque le pipeline si une règle est violée

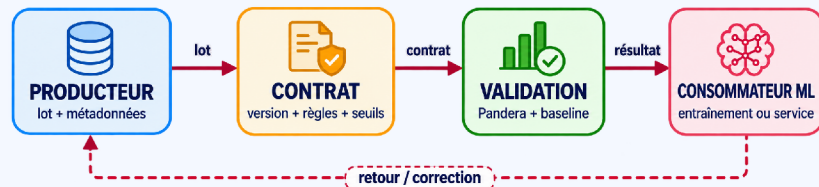
QUESTION FLASH

Une colonne sans valeur manquante est-elle valide si sa distribution s'écarte fortement de la baseline ?

Non. La complétude est une propriété nécessaire, mais elle ne suffit pas à établir la conformité du lot.

Repère : le *contrat* fixe les règles obligatoires ; la *baseline* décrit le comportement statistique habituel auquel on compare le lot.

Du contrat versionné à la règle vérifiable



DÉFINITION Un **contrat de données versionné** est une spécification qui formalise l'accord entre le producteur d'un lot et le pipeline qui le consomme.

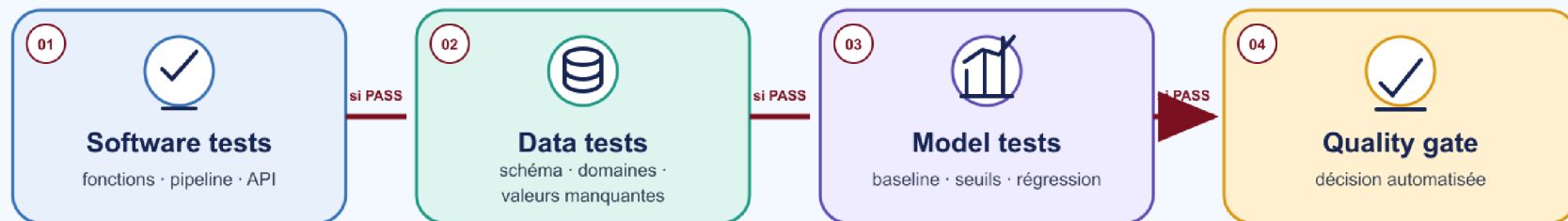
LE CONTRAT PRÉCISE

POUR RENDRE LA RÈGLE OPÉRATOIRE

Attendu	schéma, types, domaines, complétude et seuils
Contrôle	un validateur confronte le lot au contrat <code>v1</code> ou <code>v2</code>
Preuve	rapport, erreurs, empreinte du lot et statut <code>PASS</code> ou <code>FAIL</code>
Réaction	le consommateur ML poursuit ou bloque avant l'entraînement

Pourquoi versionner ? Pour relier chaque décision à la règle en vigueur et rendre les évolutions auditable.

Flux de validation



Règle de propagation : chaque niveau doit fournir une preuve positive avant d'autoriser le suivant ; un résultat **FAIL** arrête le flux.

Exemple — un contrat minimal de données

Cet exemple, spécifique au dataset utilisé durant ce cours, traduit les attentes métier en contraintes que `validate_data.py` peut vérifier automatiquement.

COLONNE	TYPE ATTENDU	RÈGLE VÉRIFIABLE
age	entier	$18 \leq \text{age} \leq 100$
tenure_months	entier	$0 \leq \text{tenure_months} \leq 180$
monthly_spend	réel	$0 \leq \text{monthly_spend} \leq 300$
support_tickets	entier	$\text{support_tickets} \geq 0$
contract	catégorie	month-to-month, one-year ou two-year
autopay	binaire	0 ou 1
churn	binaire	0 ou 1

Règle de validation : une seule contrainte violée suffit à produire `FAIL` et à bloquer le lot avant l'entraînement.

Transformer une règle en preuve

Une validation de données ne se limite pas à accepter ou rejeter un lot : elle doit produire une décision **PASS** / **FAIL** et le contexte qui permet de la comprendre.

ÉTAPE CONCEPTUELLE	QUESTION À RÉSOUDRE	PREUVE ATTENDUE
Contrôler	Le lot respecte-t-il le contrat ?	règles vérifiées et erreurs identifiées
Décider	Le pipeline peut-il continuer ?	statut PASS ou FAIL
Tracer	Pour quel lot et quelle version ?	rapport, schéma et empreinte

Principe : une erreur doit être détectée avant **prepare**, **train** et **evaluate**.

Tester avant d'orchestrer

Avant de brancher une règle au pipeline, on vérifie son comportement sur des cas contrôlés et falsifiables.

CAS DE TEST	QUESTION	RÉSULTAT ATTENDU
Lot invalide	une valeur viole-t-elle le contrat ?	validation FAIL , lot bloqué
Lot valide	toutes les contraintes sont-elles satisfaites ?	validation PASS , suite autorisée
Régression modèle	la baisse dépasse-t-elle la tolérance ?	candidat REJECT

Pourquoi tester d'abord ? Un test isolé révèle une erreur de règle avant qu'elle ne soit masquée par l'orchestration, les fichiers ou la CI.

Question suivante : une fois les contrôles passés, comment décider qu'un modèle candidat peut être promu ?

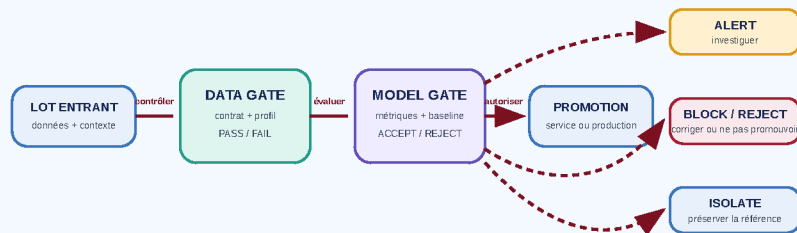
CHAPITRE

Tester et valider avant la promotion

03

Décider avant de promouvoir

Du signal à la réaction



La réaction dépend de la certitude du diagnostic et du risque de propagation.

- Une politique de qualité ne traite pas toutes les anomalies de la même manière : elle associe une réaction proportionnée au niveau de preuve et au risque métier.

SIGNAL OBSERVÉ	RÉACTION	INTERPRÉTATION OPÉRATIONNELLE
Violation déterministe du contrat	BLOQUER	empêcher le lot d'atteindre <code>prepare</code>
Écart statistique sans preuve de défaut	ALERTER	investiguer et collecter les éléments nécessaires
Lot incertain ou à risque	ISOLER	préserver la référence et éviter toute contamination

Dans une architecture complète, ces états constituent une politique de réaction : chaque décision doit être explicable, traçable et associée à un responsable.

États architecturaux et niveau d'implémentation

STATUT / ACTION	DÉCLENCHEUR	MOMENT	CODE SOURCE
PASS → poursuivre	contrat respecté	avant <code>prepare</code>	implémenté
FAIL → BLOCK	violation certaine du contrat	avant <code>prepare</code>	implémenté
ACCEPT → promouvoir	seuils respectés, pas de régression	après <code>evaluate</code>	implémenté
REJECT → ne pas promouvoir	métrique insuffisante ou régression	après <code>evaluate</code>	implémenté
ALERT → investiguer	dérive plausible, preuve incomplète	en production	conceptuel
ISOLATE / <code>COLLECT_LABELS</code>	doute ou labels nécessaires	avant consommation	conceptuel

- Le code d'exemple implémente les chemins de validation et de promotion ; les réactions de monitoring restent des extensions d'architecture. La compatibilité avec le service, le canary et le rollback sera traitée au chapitre 4.

Problème déclencheur : validation du modèle

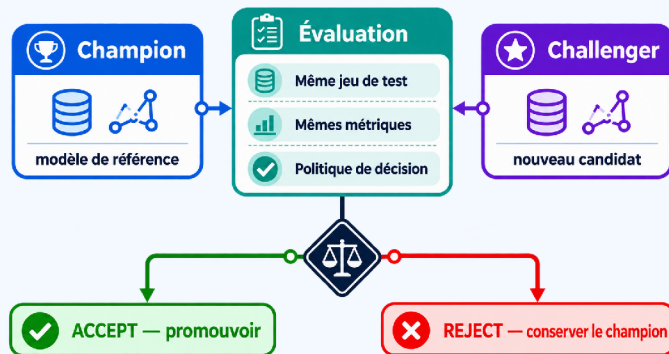
Un nouveau candidat est entraîné sur le même problème que le modèle actuellement utilisé :

RÉFÉRENCE ÉVALUÉE SUR LE MÊME PROTOCOLE	SCORE F1
Champion — modèle actuellement retenu	0.82
Challenger — nouveau candidat	0.80

- **Question :** le candidat doit-il être promu parce qu'il a été entraîné avec succès ? Justifiez votre décision avec les preuves disponibles.
- **Réponse : non, pas en l'état.** Le candidat est inférieur au champion sur cette métrique.

- La décision doit toutefois s'appuyer sur une politique explicite :
 - une **baseline** comparable et un protocole d'évaluation identique ;
 - des **seuils d'acceptation** définis avant l'exécution ;
 - une règle de non-régression conduisant à **ACCEPT** ou **REJECT** .
- La validation du modèle transforme une métrique en décision de promotion traçable ; elle ne se réduit pas à constater que l'entraînement s'est terminé.

Champion / Challenger



Le champion n'est pas remplacé parce qu'un candidat est nouveau, mais parce que les preuves montrent qu'il respecte la politique de qualité.

- La comparaison doit être reproductible et traçable : scores, version des données, paramètres d'évaluation et justification de la décision sont conservés.

- **Champion** : modèle actuellement retenu comme référence ;
- **Challenger** : candidat entraîné ou recalibré que l'on souhaite évaluer ;
- **Comparaison équitable** : même jeu de test, mêmes métriques et même protocole ;
- **Décision** : **ACCEPT** promeut le challenger ; **REJECT** conserve le champion

Baseline et seuils

Une décision de promotion combine deux références différentes :

- 01 la **baseline** décrit une performance observée sur un jeu et un protocole donnés ;
- 02 les **seuils** définissent à l'avance le niveau minimal acceptable et la régression maximale tolérée.

Exemple de baseline mesurée

```
{  
  "accuracy": 0.72,  
  "precision": 0.65,  
  "recall": 0.40,  
  "f1": 0.50,  
  "model_version": "baseline-session-2"  
}
```

Exemple de politique de qualité

```
quality:  
  min_accuracy: 0.70  
  min_recall: 0.40  
  min_f1: 0.50  
  max_f1_regression: 0.03  
  max_recall_regression: 0.05
```

- `min_*` impose un niveau minimal absolu ;
- `max_*_regression` protège la performance du champion ;
- la promotion est autorisée seulement si **toutes** les règles sont satisfaites.
- La baseline fournit le point de comparaison ; elle ne remplace pas la politique de qualité, qui doit être explicite, versionnée et traçable.

Une baseline traçable

- Une baseline n'est exploitable que si sa valeur est accompagnée de la provenance et des conditions de mesure :

```
{  
  "model_version": "baseline-session-2",  
  "dataset_sha256": "...",  
  "split": "test-v1",  
  "seed": 42,  
  "metric_definition": "sklearn.recall_score",  
  "code_version": "git-commit"  
}
```

- `model_version` identifie la référence comparée ;
- `dataset_sha256` et `split` identifient les données et la partition utilisées ;
- `seed` et `metric_definition` rendent le calcul reproductible ;
- `code_version` permet de retrouver l'implémentation de l'évaluation.
- Sans ce contexte, une valeur de baseline ne permet pas de vérifier que le champion et le challenger ont été évalués dans les mêmes conditions. La baseline doit donc être conservée comme une preuve versionnée, pas comme un simple nombre recopié dans un tableau.

Produire les métriques enrichies

`evaluate.py` évalue le candidat sur un jeu de test inchangé et produit plusieurs angles de lecture :

- **accuracy** — proportion de prédictions correctes parmi toutes les observations ;
- **precision** — proportion de prédictions positives qui sont correctes ;
- **recall** — proportion de cas positifs effectivement détectés ;
- **f1** — moyenne harmonique de **precision** et **recall**.

DU SCORE À LA DÉCISION

- Le **quality gate** vérifie ici **accuracy**, **recall** et **f1** par rapport aux seuils définis ;
- **precision** reste disponible pour analyser le coût des faux positifs ;
- une décision **ACCEPT** exige que les critères retenus soient satisfaits simultanément.

Question flash — pourquoi une **accuracy** élevée ne suffit-elle pas nécessairement à déclarer le modèle fiable ?

Réponse : si les classes sont déséquilibrées, un modèle peut obtenir une bonne **accuracy** en détectant mal la classe à risque. Il faut donc examiner au moins le **recall**, la **precision** et leur compromis **f1**.

Tests de modèle

Un test de modèle vérifie une propriété mesurable du candidat avant que celui-ci ne soit comparé pour promotion.

```
def test_f1_above_policy_threshold():  
    metrics = json.loads(Path("reports/metrics.json").read_text())  
    quality = load_params()["quality"]  
    assert metrics["f1"] >= quality["min_f1"]
```

- Le test vérifie un **critère isolé** : ici, le F1 minimal ;
- le seuil vient de la politique de qualité, plutôt que d'être dispersé dans le code ;
- un échec fournit une preuve localisée et empêche de considérer le critère comme satisfait.

Un test unitaire répond à « cette règle est-elle vraie ? » ; le quality gate répond à « le candidat peut-il être promu ? ».

Protocole d'évaluation contrôlé

Avant de comparer deux modèles, verrouiller les conditions qui pourraient fausser la conclusion :

- **même cible et même jeu de test ;**
- **même preprocessing et même convention de seuil ;**
- jeu de test séparé du réglage du modèle et des seuils ;
- `split` , `seed` et versions des données conservés ;
- variation ou incertitude documentée lorsque la décision est proche du seuil.

POURQUOI CE CONTRÔLE ?

- Un changement de jeu de test peut créer un écart artificiel ;
- un preprocessing différent ne mesure pas le même système ;
- un seuil ajusté sur le test transforme l'évaluation en apprentissage indirect.
- Le résultat du gate porte donc sur une **comparaison contrôlée** : un écart de score n'est interprétable que si les conditions de mesure sont comparables.

Validation par segments

Une métrique globale peut masquer une dégradation sur une partie des usagers. On examine donc la performance sur des sous-populations définies à l'avance. **Exemple illustratif** :

SEGMENT	RECALL CANDIDAT	MINIMUM	DÉCISION
contrat mensuel	0.58	0.55	PASS
contrat annuel	0.43	0.55	FAIL
autopay activé	0.61	0.55	PASS

- Cette vérification complète la métrique globale ; elle ne la remplace pas. Elle est présentée comme extension méthodologique de ce qui est implémenté par le code du cours.
- La validation du modèle transforme une performance observée en **comparaison contrôlée** avec une référence et des seuils.

- un segment sous le seuil signale une **faiblesse localisée**, même si la moyenne globale est acceptable ;
- le seuil, la taille minimale du segment et la règle d'écart doivent être documentés ;
- la décision globale doit préciser si un seul segment en échec suffit à rejeter le candidat.

Une métrique ne raconte pas toute l'histoire

Pour une classe rare, une `accuracy` élevée peut coexister avec un `recall` trop faible. La bonne question n'est donc pas seulement « quel modèle a le meilleur score ? », mais « quelles erreurs le système accepte-t-il dans ce contexte ? ».

LIRE LES ERREURS AVANT LE SCORE

Un score global agrège des erreurs qui n'ont pas nécessairement la même importance. La métrique prioritaire doit donc être reliée à l'usage du modèle.

SIGNAL	QUESTION À POSER
<code>accuracy</code>	quelle proportion totale est correcte ?
<code>precision</code>	parmi les alertes, combien sont justes ?
<code>recall</code>	combien de cas à risque sont détectés ?
<code>f1</code>	quel compromis entre <code>precision</code> et <code>recall</code> est obtenu ?

Relier les métriques au coût métier

On appelle **classe positive** l'événement que le modèle cherche à détecter : une panne à venir ou un churn probable.

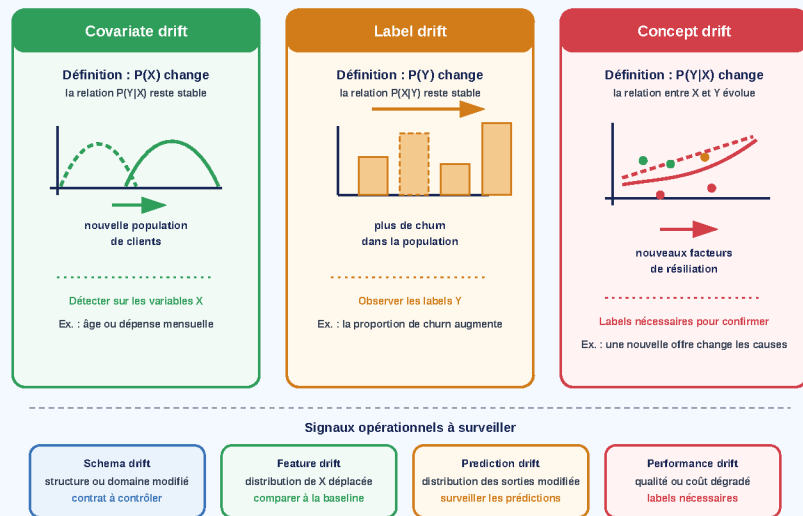
	ÉVÉNEMENT RÉEL	PAS D'ÉVÉNEMENT
Alerte	événement détecté (TP)	fausse alerte (FP)
Pas d'alerte	événement manqué (FN)	cas correctement ignoré (TN)

- La matrice de confusion relie **chaque prédiction à une conséquence** : intervenir sur l'équipement, contacter le client, ou analyser le cas a posteriori.

DEUX POLITIQUES MÉTIER POSSIBLES

- Maintenance prédictive** — positif = panne prochaine ; un **FN** peut entraîner un arrêt de production. On privilégie souvent le **recall**, en maîtrisant le coût des interventions déclenchées par les **FP**.
- Churn client** — positif = résiliation probable ; un **FN** signifie perdre une occasion de rétention. On arbitre le **recall** avec le coût des offres ou contacts inutiles (**FP**).
- Le même modèle de classification peut donc conduire à des **seuils différents** : la métrique prioritaire dépend de la gravité, du coût et du délai de réaction propres au métier.
- La matrice de confusion rend la **décision discutable** : elle relie chaque score à un type d'erreur, à son coût et à une action de contrôle.

Drift : signal de diagnostic



QUESTION CENTRALE Quelle distribution a changé ?

- Cette distinction aide à choisir la mesure et la preuve appropriées.
 - $P(X)$ change : la population ou les entrées évoluent ;
 - $P(Y)$ change : la fréquence des classes évolue ;
 - $P(Y|X)$ change : la relation entre les entrées et la cible évolue.
- Ces trois catégories décrivent **ce qui change** dans la distribution. Les signaux observables et les réactions opérationnelles seront distingués dans les deux slides suivants.

Observer le drift dans le système

SIGNAL OPÉRATIONNEL	DÉFINITION	EXEMPLE
Schema drift	structure ou domaine d'entrée modifié	catégorie <code>contract</code> inconnue
Feature drift	distribution d'une variable X déplacée	dépense mensuelle plus élevée
Prediction drift	distribution des sorties $\hat{Y}$ modifiée	davantage de <code>churn = 1</code> prédit
Performance drift	qualité de $\hat{Y}$ dégradée	<code>recall</code> inférieur sur labels récents

- **Schema drift** : vérifiable immédiatement par le contrat de données ;
- **Feature drift** : détectable en comparant le lot courant à la baseline ;
- **Prediction drift** : signal de surveillance, sans preuve automatique d'erreur ;
- **Performance drift** : nécessite des labels pour comparer les métriques.

REPÈRE Un signal opérationnel est une observation. Il déclenche une investigation, mais ne prouve pas encore quel mécanisme statistique est à l'œuvre ni que la performance du modèle s'est dégradée.

Interpréter le drift dans son contexte

MAINTENANCE PRÉDICTIVE

- **Feature drift** : `support_tickets` ou la température moyenne se déplace ;
- **Prediction drift** : les alertes de panne deviennent plus fréquentes ;
- **Performance drift** : le taux de pannes détectées diminue lorsque les labels arrivent ;
- **Réaction** : surveiller, qualifier le signal, puis ajuster le seuil ou le modèle si nécessaire.

CHURN CLIENT

- **Feature drift** : la dépense mensuelle ou la durée d'abonnement change ;
- **Prediction drift** : la proportion de `churn = 1` prédite augmente ;
- **Concept drift** : une nouvelle offre modifie les facteurs associés à la résiliation ;
- **Réaction** : vérifier la distribution, collecter les labels, puis recalculer le `recall` .



Même signal, interprétation différente : un déplacement de `X` est détectable sans labels ; une baisse de performance ou un concept drift exige de connaître la cible réelle.

Étape suivante : une fois le signal qualifié, comment décider formellement si le modèle peut avancer vers la promotion ?

Transformer les preuves en décision

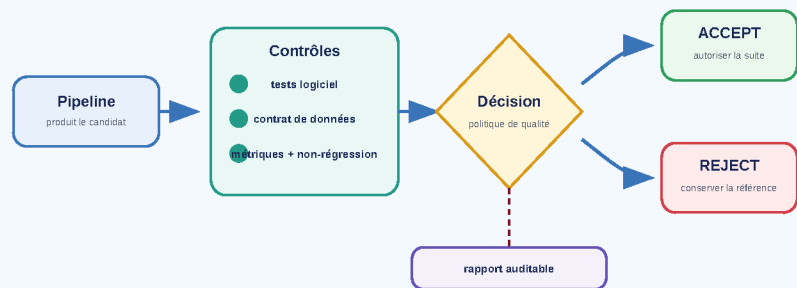
Le pipeline a produit un modèle candidat et plusieurs artefacts de validation. Leur présence ne suffit pas : il faut une règle qui autorise ou refuse l'étape suivante.

preuves → politique → décision

- **Entrées** : rapport de données, métriques du candidat, baseline et seuils ;
- **Règle** : les critères sont-ils tous satisfaits ? une régression est-elle tolérée ? ;
- **Sortie** : **ACCEPT** pour poursuivre ou **REJECT** pour conserver le champion ;
- **Preuve** : checks, raisons et contexte de comparaison enregistrés dans un rapport.

Question directrice — comment rendre cette décision automatique, explicable et auditable ?

Quality gate



- **Entrées** : validation des données, métriques du candidat, baseline et seuils ;
- **ACCEPT** : autorise l'étape suivante selon la politique définie ;
- **REJECT** : bloque la promotion et conserve la référence ;
- **Preuve** : contrôles, raisons, versions et contexte enregistrés.
- **ACCEPT** signifie « le candidat satisfait les critères de ce gate » ; ce n'est pas, à lui seul, une autorisation de mise en production.

DÉFINITION Un **quality gate** est un point de contrôle qui applique une politique explicite aux preuves produites par le pipeline et retourne une décision traçable.

Étape 3 — Implémenter la décision du quality gate

Objectif : centraliser la politique de promotion dans `evaluate_quality(...)`. La fonction reçoit des dictionnaires, ne lit ni fichier ni serveur et peut donc être testée isolément.

évaluer les contrôles



`reports/quality_gate_decision.json`

- chaque règle produit un résultat nommé ;
- la décision agrège ces résultats ;
- le rapport conserve la décision et les éléments permettant de la recalculer.

ENTRÉES	RÔLE
<code>reports/metrics.json</code>	métriques du candidat
<code>quality_gates/baseline_metrics.json</code>	référence de comparaison
<code>configs/params.yaml</code>	seuils et tolérances

Logique de décision

```
checks = {
    "accuracy_above_threshold": metrics["accuracy"] >= min_acc,
    "recall_above_threshold": metrics["recall"] >= min_recall,
    "f1_above_threshold": metrics["f1"] >= min_f1,
    "f1_regression_acceptable": metrics["f1"] >= baseline["f1"] - max_delta_f1,
    "recall_regression_acceptable": (
        metrics["recall"] >= baseline["recall"] - max_delta_recall
    ),
}
accepted = all(checks.values())
```

UNE DÉCISION CONJONCTIVE

ACCEPT est produit si et seulement si toutes les règles sont vraies :

$\text{ACCEPT} \Leftrightarrow \forall \text{contrôle, résultat} = \text{True}$

- une seule règle critique échouée suffit à produire **REJECT** ;
- les seuils absolus protègent un niveau minimal ;
- les tolérances de régression protègent le champion ;
- les noms des contrôles restent identiques dans le code et le rapport.

Le contrat de sortie du gate

Le rapport doit être suffisamment riche pour répondre à trois questions : **quelle décision ? sur quelles valeurs ? avec quelles preuves ?**

```
{
  "decision": "REJECT",
  "checks": {
    "recall_regression_acceptable": false
  },
  "metrics": {"recall": 0.51},
  "baseline": {"recall": 0.62},
  "thresholds": {"max_recall_regression": 0.05},
  "reasons": ["recall régresse au-delà de la tolérance"],
  "provenance": {
    "code_version": "git-commit",
    "test_sha256": "..."
  }
}
```

- `decision` donne le verdict global ;
- `checks` indique quelle règle a échoué ;
- `metrics`, `baseline` et `thresholds` permettent le recalcul ;
- `reasons` rend le rejet compréhensible ;
- `provenance` rattache la décision aux données et au code exécutés.

Le texte affiché, le code retour et le rapport JSON doivent raconter la même décision. Un booléen sans contexte n'est pas une preuve d'audit.

Exercice — appliquer le quality gate

Cas hypothétique. Un challenger est comparé au champion sur le même jeu de test.

RÉFÉRENCE	ACCURACY	RECALL	F1
Champion / baseline	0.90	0.62	0.66
Challenger	0.93	0.51	0.62

Seuils : $\text{min_f1} = 0.60$ · $\text{max_recall_regression} = 0.05$ ·
 $\text{max_f1_regression} = 0.03$

Question — quelle décision le quality gate doit-il produire ?
Examinez chaque règle, pas seulement le score qui s'améliore.

CORRECTION

CONTRÔLE	CALCUL	RÉSULTAT
$\text{f1} \geq 0.60$	$0.62 \geq 0.60$	✅ réussi
Régression recall	$0.62 - 0.51 = 0.11 > 0.05$	❌ échoué
Régression f1	$0.66 - 0.62 = 0.04 > 0.03$	❌ échoué

Décision : `REJECT`. Une seule règle critique en échec suffit à conserver le champion. L'accuracy supérieure du challenger ne compense pas les régressions sur le recall et le f1.

Étape 4 — Observer deux décisions du gate

Objectif : obtenir un `ACCEPT`, puis provoquer volontairement un `REJECT` sans modifier la configuration versionnée.

```
make init-dvc
make pipeline
# QUALITY GATE: ACCEPT

python src/quality_gate.py --min-f1 0.95
# QUALITY GATE: REJECT, code de sortie non nul
```

La surcharge `--min-f1` provoque un rejet sans modifier la configuration versionnée. Relancez `make gate` pour rétablir la décision `ACCEPT`.

- **Décision nominale :** `QUALITY GATE: ACCEPT` et code de sortie `0` ;
 - **Décision forcée :** `QUALITY GATE: REJECT` et code de sortie non nul ;
 - **Retour à l'état nominal :** relancer `make gate` avec la politique inchangée ;
 - **Preuve attendue :** message, rapport JSON et code de sortie concordants.
- Le test vérifie le comportement du gate, pas seulement son implémentation : un rejet doit empêcher l'étape suivante et expliquer sa cause.

Jouer la matrice des scénarios

Le runner construit des variantes en mémoire : le dataset, la baseline et la politique versionnée restent inchangés.

```
make scenarios
# data-pass -> PASS
# data-unknown-category -> FAIL
# quality-accept -> ACCEPT
# quality-recall-regression-reject -> REJECT
# quality-missing-metric -> ERROR
```

```
python src/run_scenarios.py --scenario quality-recall-re
```

- `data-*` vérifie le comportement du contrat de données ;
- `quality-*` vérifie les seuils, les régressions et les métriques manquantes ;
- chaque scénario compare le résultat observé au résultat attendu ;
- un résultat inattendu provoque un code de sortie non nul. Ces scénarios rendent les cas limites rejouables et documentent le comportement attendu du gate.

Ajouter le gate au pipeline DVC

```
quality_gate:
  cmd: python src/quality_gate.py
  deps:
    - src/quality_gate.py
    - reports/metrics.json
    - quality_gates/baseline_metrics.json
    - reports/run_manifest.json
  params:
    - configs/params.yaml:
      - quality
  outs:
    - reports/quality_gate_decision.json:
      cache: false
```

- `deps` garantit que le gate dépend des preuves qu'il lit ;
- `params` lie le stage à la politique `quality` versionnée ;
- `outs` conserve la décision sans la mettre en cache ;
- un code de sortie non nul empêche la chaîne DVC/CI de poursuivre.

```
make repro
```

Le graphe DVC rend explicite la dépendance entre évaluation, quality gate et rapport de décision.

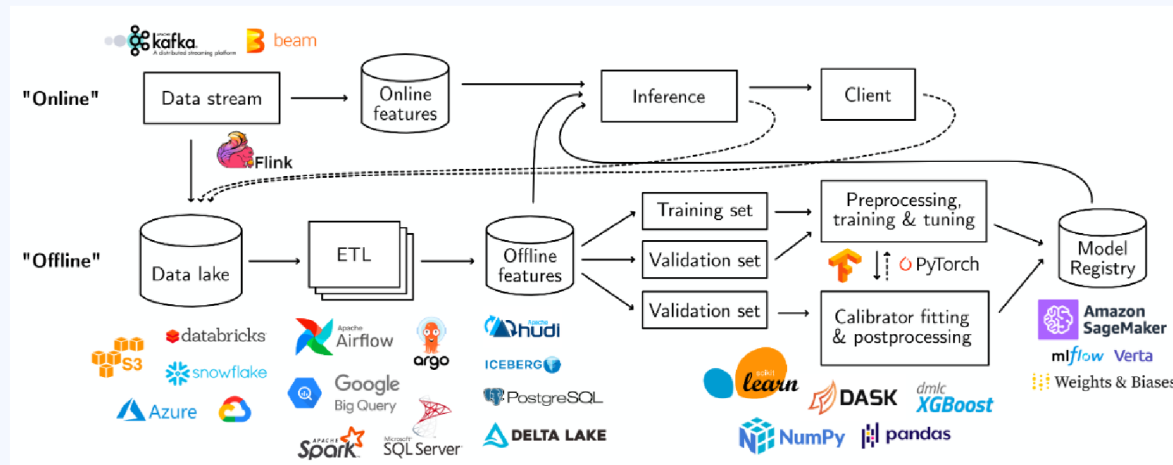
CHAPITRE

Tester et valider avant la promotion

04

Relier preuves et outils

Un pipeline ML, plusieurs responsabilités



- Un framework ne couvre généralement qu'une partie du cycle. La fiabilité vient de l'articulation entre contrôles, traçabilité, décision et observation.

À chaque risque, sa famille d'outils

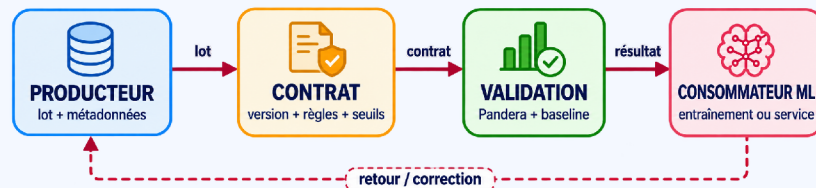
TÂCHE	EXEMPLES DE FRAMEWORKS
Tester le code	<code>pytest</code> , <code>unittest</code> , JUnit, SonarQube
Valider les données	Pandera, Great Expectations, TFDV, Soda
Versionner les données	Git, DVC, LakeFS, Pachyderm
Suivre les expériences	MLflow, Weights & Biases, Neptune, Comet
Évaluer le modèle	Deepchecks, Giskard, WhyLabs, tests métier
Gérer les modèles	MLflow Registry, SageMaker Registry, Vertex AI Registry, Azure ML Registry
Observer le drift	Evidently, WhyLabs, Arize, Fiddler
Orchestrer	Airflow, Dagster, Prefect, Kubeflow

Question de méthode

La question n'est pas « quel outil est le meilleur ? », mais « quelle preuve doit être produite, à quel moment et par quelle équipe ? ».

Formaliser et vérifier le contrat de données

DÉFINITION Un **contrat de données** spécifie, de manière versionnée et vérifiable, la structure, les domaines de valeurs, la complétude et les contraintes métier attendus pour un lot.

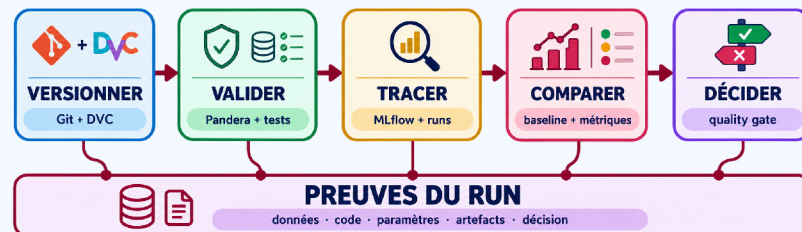


- **Formaliser** : exprimer l'attendu sous forme de règles explicites;
- **Vérifier** : comparer le lot reçu à ces règles avant `prepare` ;
- **Prouver** : conserver le statut, les violations et l'identifiant du lot;
- **Agir** : bloquer, corriger ou investiguer selon la gravité.

Frameworks : Pandera. Great Expectations. TFDV et Amazon

Tracer les expériences et comparer les modèles

DÉFINITION La **traçabilité expérimentale** associe à chaque run les données, le code, les paramètres, les métriques et les artefacts nécessaires pour reproduire et discuter une comparaison.



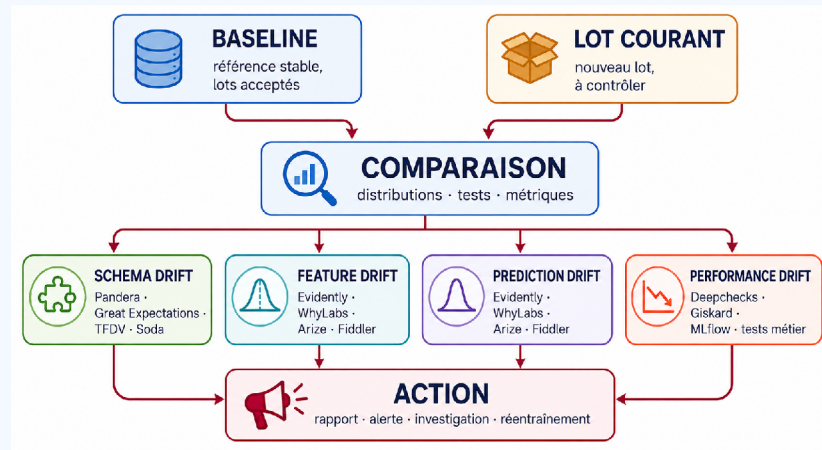
- **Versionner le contexte** : Git, DVC, LakeFS ou Pachyderm ;
 - **Enregistrer le run** : MLflow, Weights & Biases, Neptune ou Comet ;
 - **Comparer équitablement** : même protocole, même jeu de test et mêmes métriques ;
 - **Décider** : confronter le candidat à la baseline avant toute promotion.
- Un run traçable relie les contrôles et les preuves à la décision.
 - Le tracking décrit ce qui a été exécuté ; la politique de qualité définit ce qui est acceptable.

Observer les dérives après la promotion

Après la validation initiale, les frameworks de monitoring automatisent la comparaison entre une référence et les observations de production.

- **Evidently** : rapports, tests et indicateurs de dérive ;
- **WhyLabs** : observabilité des données et alertes de qualité ;
- **Arize** : suivi des modèles et des prédictions en production ;
- **Fiddler** : monitoring, analyse des erreurs et explicabilité.

Preuves attendues : rapport horodaté, seuil déclenché, lot ou période concernés et lien vers le run de référence.



Les frameworks produisent des signaux ; l'équipe définit la réaction.

Open source ou service managé ?

CRITÈRE	OPEN SOURCE	SERVICE MANAGÉ
Exemples	MLflow, DVC, Airflow, Evidently	SageMaker, Vertex AI, Azure ML, Databricks
Portabilité	généralement élevée	dépendance au fournisseur
Exploitation	installation, sécurité et maintenance par l'équipe	infrastructure partiellement prise en charge
Personnalisation	forte, avec davantage de compétences	plus rapide, mais bornée par la plateforme
Coût	infrastructure + compétences	abonnement + consommation + intégration

- **Règle de choix :** comparer le coût total, la portabilité, les exigences d'audit, les compétences disponibles et le niveau d'exploitation accepté.
- Un outil observe ou automatise une étape ; il ne remplace ni le contrat, ni la politique de qualité, ni le responsable de l'action.

Transition : Quels scénarios permettent d'éprouver cette chaîne de preuves et ses décisions ?

CHAPITRE

Tester et valider avant la promotion

05

Éprouver la décision —

Relier dataset, lot et run

OBJECTIF DE L'ÉPREUVE Une décision n'est éprouvée que si l'on peut relier le lot contrôlé, la validation, les métriques, la baseline et le run qui les a produits.

- identifier le lot par son empreinte ;
- conserver le rapport de validation ;
- associer métriques, baseline et décision ;
- retrouver le code, les paramètres et le run.



Chaque étape ajoute une preuve jusqu'à la décision et au manifeste du run.

Exécuter le protocole de vérification

Depuis `code/seance3/finish/`, exécuter la chaîne nominale puis observer le résultat du gate :

```
make repro-offline
cat reports/quality_gate_decision.json
dvc dag
```

- **Résultat attendu :** le pipeline se termine, le rapport indique `ACCEPT` et le code de sortie vaut `0`.
- Le graphe montre les dépendances ; le rapport et le code de sortie rendent la décision observable et automatisable.

Jouer trois incidents contrôlés

Les scénarios isolent une cause et imposent une preuve observable :

SCÉNARIO	DÉCISION ATTENDUE	PREUVE À VÉRIFIER
colonne <code>autopay</code> absente	FAIL	règle violée + code non nul
<code>recall</code> sous la tolérance	REJECT	métrique, baseline et raison
distribution d'une feature déplacée	signal à investiguer	période, seuil et responsable

```
make scenarios
```

Les deux premiers scénarios sont exécutables ; le troisième sert à formuler le diagnostic et la réaction. `ALERT` et `ISOLATE` restent des extensions d'architecture.

Constituer le dossier de preuve

Une personne absente de l'exécution doit pouvoir répondre à trois questions : **quel lot ? quelles règles ? quelle décision ?**

- `data/raw/customers.csv` : lot contrôlé et empreinte ;
- `reports/data_validation_report.json` : règles et violations ;
- `reports/metrics.json` : métriques du candidat ;
- `quality_gates/baseline_metrics.json` : référence de comparaison ;
- `reports/quality_gate_decision.json` : décision et raisons ;
- `reports/run_manifest.json` : code, données, paramètres et `run_id`.

CRITÈRE DE RÉUSSITE

- Les artefacts, le message affiché et le code de sortie racontent la même histoire.
- Une décision fiable est une décision testée, expliquée et retrouvable.

CHAPITRE

Tester et valider avant la promotion

06 Synthèse —

Une décision de promotion se construit par preuves

La séance a transformé un pipeline qui produit des artefacts en un pipeline qui justifie une décision.



- le **data gate** vérifie le lot avant `prepare` et `train` ;
- le **model gate** vérifie les métriques après `evaluate` ;
- la baseline rend la comparaison interprétable ;
- `ACCEPT` autorise la suite, mais ne signifie pas encore « production ».

Du pipeline reproductible au pipeline contrôlé

DÉBUT

Pipeline reproductible

validate → prepare → train → evaluate

FIN

Pipeline contrôlé

tests + rapport + baseline + quality gate

Le pipeline ne se contente plus de produire un modèle : il produit une **décision contrôlée, expliquée et retrouvable**.

- **Avant l'entraînement** : le lot doit respecter le contrat de données ;
- **Après l'évaluation** : le candidat doit respecter les seuils et les tolérances ;
- **À chaque étape** : le rapport doit conserver la preuve et le code de sortie.

La règle opérationnelle à retenir

OBSERVATION	CONTRÔLE	DÉCISION / SUITE
schéma ou valeur impossible	contrat de données	BLOCK avant l'entraînement
métrique sous le seuil	model gate	REJECT après l'évaluation
régression par rapport à la baseline	comparaison contrôlée	conserver le champion
dérive plausible en production	monitoring	ALERT puis investigation

Dans la démonstration, le blocage des données et le rejet du modèle sont exécutables ; l'alerte de production reste une extension d'architecture.

FORMULE DE SYNTHÈSE **Pas de preuve, pas de promotion.**

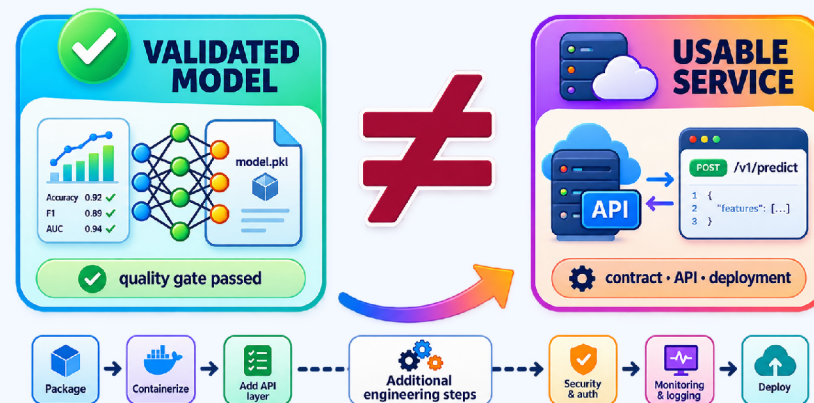
Après le quality gate : rendre le modèle utilisable

Le quality gate répond à la question : **le candidat respecte-t-il la politique de qualité ?**

`validated model` $\neq$ `usable service`

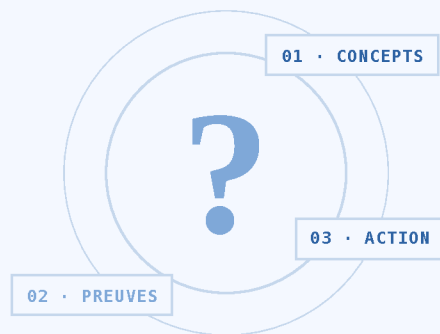
- **Séance 3** : artefact évalué, décision justifiée et preuve conservée ;
- **Séance 4** : contrat d'inférence, API et packaging pour rendre le modèle consommable.

QUESTION SUIVANTE **Comment exposer ce modèle dans un service fiable ?**



La validation précède l'exposition du modèle.

MLOPS ET INDUSTRIALISATION DES MODÈLES IA



Questions et discussion

Qu'est-ce qui mérite d'être vérifié, discuté
ou mis en pratique ?

POUR PROLONGER

Pr. Youness BOUTYOUR youness.boutyour@um5.ac.ma