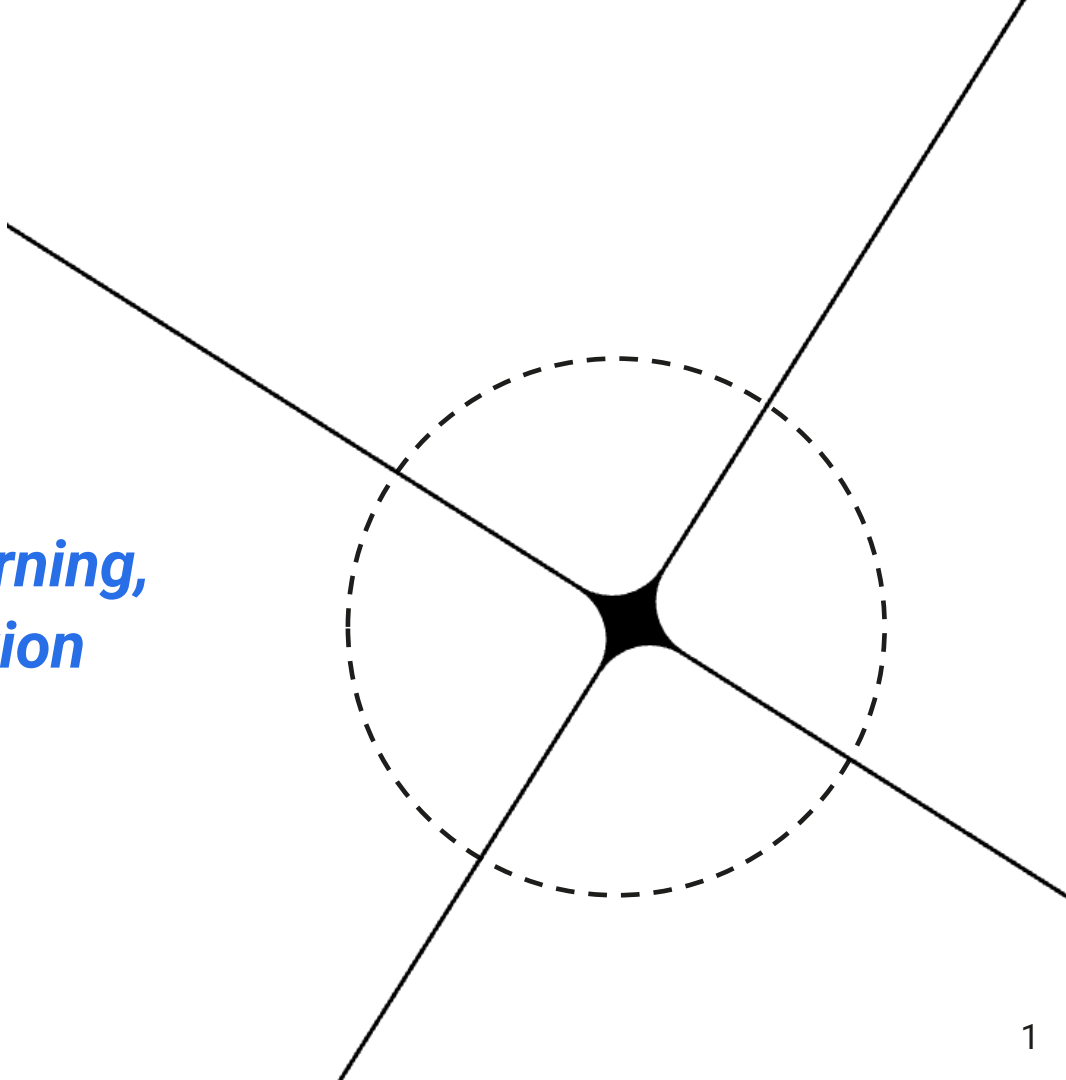
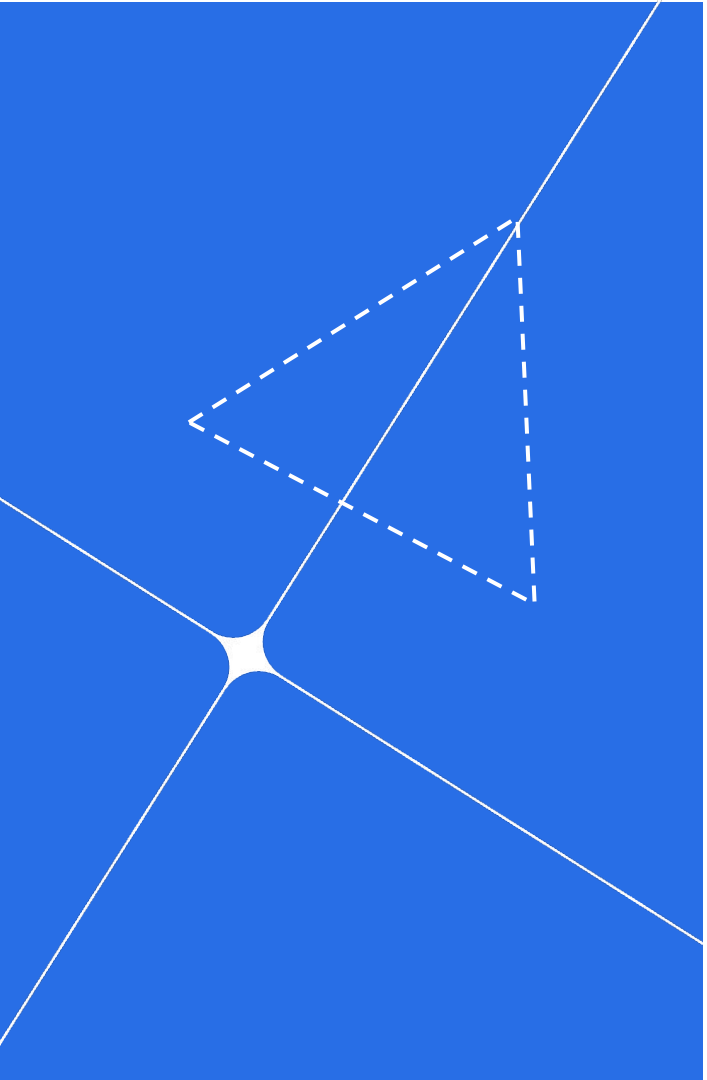
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

CI/CD pour le machine learning, réentraînement et Intégration

Julien Romero





Cadrage et motivation

TP6 : CI/CD ML, retraining automatisé, intégration complète

- Objectif : boucler la boucle (dev -> prod -> monitoring -> retraining -> promotion)
- Résultat : système plus fiable, plus reproductible, moins “fragile”
- Livrables techniques (vue globale)
  - CI GitHub Actions (tests + intégration)
  - Promotion/rollback de modèles (registry)
  - Stratégies de déploiement (concepts + où ça vit)

Motivation : pourquoi les systèmes ML cassent en production

- Sources de régression
  - **Code** : refactor, dépendances, config
  - **Données** : schéma, distributions, valeurs rares, qualité
  - **Features** : calcul, disponibilité, cohérence offline/online
  - **Modèle** : drift, surapprentissage, changement de target
- Risques typiques
  - Erreurs silencieuses (mauvais scores sans crash)
  - Dégradations progressives (drift lent)
  - “Ça marche en local” mais pas en stack complète
- Réponse MLOps
  - Tests + gates (qualité)
  - Automatisation (répétable)
  - Rollback (recovery rapide)

Rappel : pipeline MLOps complet (vue système)

- Chaîne principale (offline -> serving)

- Boucle de feedback (prod -> amélioration)

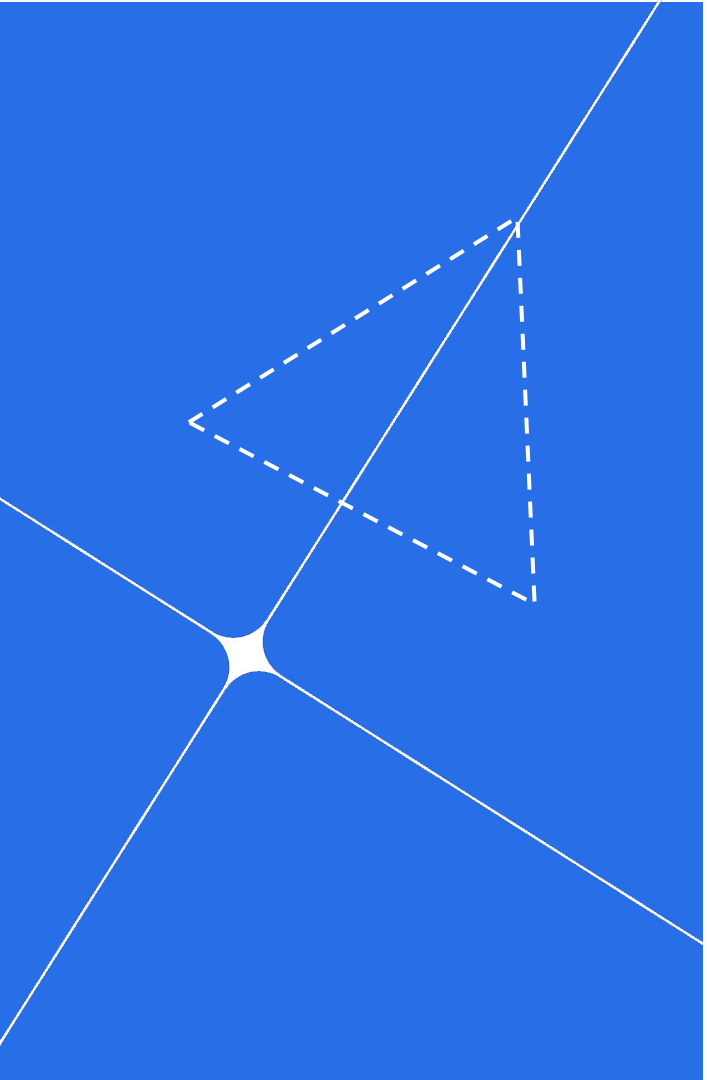
- Point clé : “Déployer” en ML = code + artefacts modèle + décisions de promotion

Où on en est (après TP1–TP5)

- Stack locale docker-compose
  - API (serving), MLflow (tracking/registry), Prefect (orchestration), monitoring/drift
- Capacités déjà acquises
  - Pipelines orchestrés (Prefect)
  - Validation données (Great Expectations)
  - Observabilité / drift (dashboards + rapport)
  - Entraînement + logging MLflow, modèle servi via API
- Limite actuelle
  - Qualité non vérifiée : pas de vérification automatiques sur chaque changement
  - Boucle retraining/promotion pas encore “production-grade”

Ce que TP6 ajoute (résultats attendus)

- **CI (GitHub Actions)** : qualité code + intégration système
  - unit tests (pytest)
  - tests d'intégration (compose + appel FastAPI)
  - data checks "light" (invariants)
- **CD côté modèle** : train → evaluate → compare → promote (+ rollback)
  - stratégies : manuel / semi-auto / auto (aperçu théorique)
  - règle de marge (stabilité)
- **Déploiement modèles (concepts)** : shadow / A/B / canary / interleaving / bandits
  - clarification : promotion registry ≠ routage de trafic



Définitions CI/CD et spécificités ML

Définitions : CI, CD, CT

- **CI (Continuous Integration)**
  - intégrer fréquemment (à chaque push/Pull Request)
  - exécuter automatiquement : build + tests + checks
  - objectif : détecter régressions tôt
- **CD (Continuous Delivery / Deployment)**
  - delivery : artefacts prêts à être déployés (release candidate)
  - deployment : déploiement automatique en prod (selon politique)
- **CT (Continuous Training)**
  - ré-entraîner régulièrement / sur événement (drift, nouvelle période)
  - produire un candidat (pas forcément prod)
- Exemples typiques
  - Pull Request -> CI -> OK
  - Nouveau mois data -> CT -> modèle candidat

CI/CD logiciel vs CI/CD ML : différences structurelles

- **Logiciel “classique”**
  - entrée : code
  - tests : déterministes
  - livrable : binaire / image / service
- **ML system**
  - entrée : code + données + features + config
  - non-déterminisme (seed, sampling, infra, versions libs)
  - “qualité” = métriques + contraintes (pas juste “ça compile”)
- **Conséquence**
  - CI doit tester le système, pas uniquement des fonctions
  - CD côté ML implique gates (évaluation, compatibilité, risques)

Production = prêt à servir un client

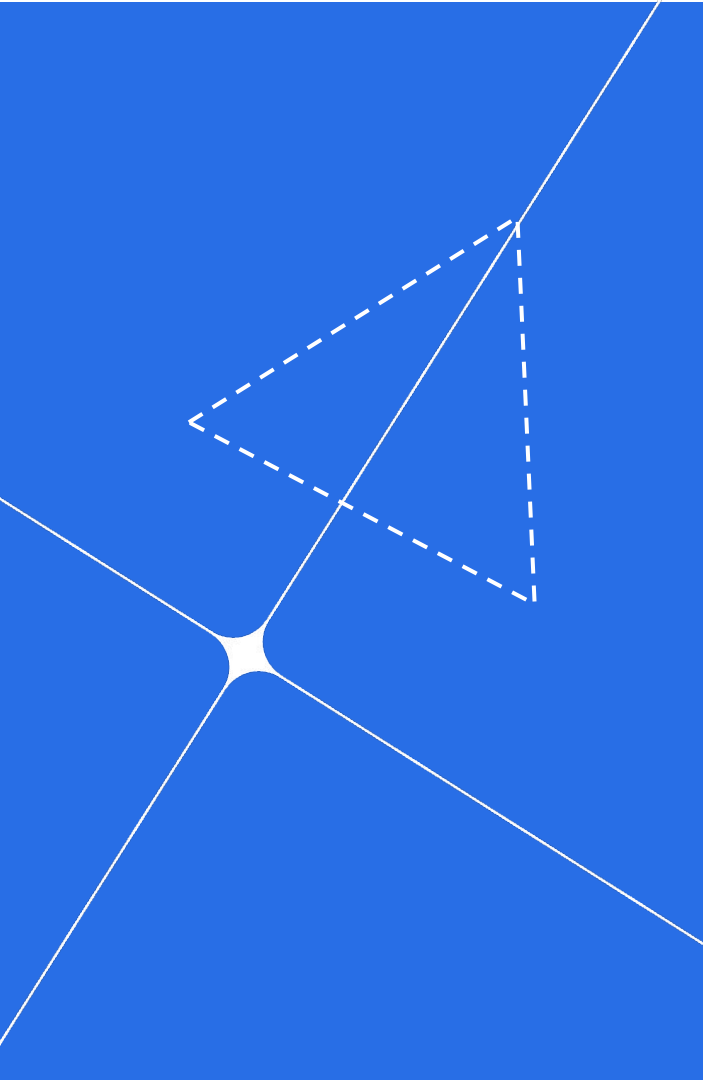
- “Production model” = modèle servable dans un contexte réel
  - latence acceptable, robustesse, compat API
  - observabilité : logs/metrics
  - traçabilité : version, données, features, config
- Attention : “meilleur offline” ≠ “meilleur en prod”
  - distribution change, coût erreurs, contraintes infra
- Implication : promotion = décision technique + produit + risque

Environnements : dev / staging / prod

- Pourquoi des environnements
  - isoler les changements
  - valider intégration avant exposition client
  - réduire incidents
- Caractéristiques typiques
  - **dev** : rapide, itératif, logs verbeux
  - **staging** : proche prod, tests réalistes, données contrôlées
  - **prod** : stable, observé, changements maîtrisés
- Dans le cours (local/compose)
  - environnements ≈ config + profils compose + variables
  - Même stack, paramètres différents (ports, volumes, niveaux logs)

Deux livraisons : code et modèle

- Livraison code (CI/CD)
  - GitHub Actions : tests → build image → intégration
  - objectif : garantir “le système démarre et répond”
- Livraison modèle (CT + promotion)
  - orchestration : train/eval/compare
  - artefact : version modèle (registry)
  - Décision : promouvoir / conserver / rollback
- Message clé
  - on peut changer le modèle sans changer le code
  - et inversement : changer le code sans changer le modèle
  - ne pas confondre : promotion registry vs déploiement trafic (à venir)



Stratégie de tests pour systèmes ML

Pourquoi tester un système ML est multi-couches

- Un modèle en prod = un système distribué (data + services + artefacts)
- Les régressions possibles ne sont pas seulement “bugs code”
  - données invalides mais “parsables”
  - features incohérentes (offline/online)
  - modèle incompatible avec l’API (signature, types)
- Objectif des tests
  - détecter tôt
  - isoler la cause (code vs data vs modèle vs intégration)

Taxonomie des tests ML

- **Unit tests (pytest)**
  - parsing, transformations, validation paramètres
  - fonctions pures, rapides
- **Contract tests (API)**
  - schéma request/response, types, champs obligatoires
- **Data tests**
  - schéma, null-rate, ranges, catégories autorisées
  - invariants (ex : âge ≥ 0, dates cohérentes)
- **Model tests**
  - performance minimale (floor)
  - non-régression vs baseline / production
  - compatibilité signature + features attendues
- **Integration tests**
  - plusieurs services (compose), appel HTTP réel

Politique de test du cours

- Contraintes
  - CI doit être rapide et stable
  - pas d'entraînement lourd à chaque PR
- Donc, en CI (GitHub Actions)
  - unit + contract : systématique
  - integration smoke (compose + FastAPI) : systématique
  - data checks légers : systématique
- Hors CI (orchestrateur / exécution contrôlée)
  - entraînement complet + évaluation + comparaison
  - promotion/rollback

Unit tests (pytest) : style attendu

- Tests sur fonctions pures
  - load_config(): valeurs par défaut, erreurs explicites
  - make_features(df): colonnes attendues, types, pas de NaN introduits
  - split_train_valid(): tailles, reproductibilité (seed)
- Bonnes pratiques
  - tests courts, déterministes
  - fixtures pour données jouets
  - messages d’erreur utiles (assert explicites)

Pytest : l'essentiel pour démarrer

- Convention
  - fichiers : test_*.py (ou *_test.py)
  - fonctions : def test_xxx(): ...
- Exécution
  - pytest (tous les tests)
  - pytest -q (compact), pytest -k "pattern" (filtre)
- Assertions
  - assert expr + messages explicites
- Fixtures (réutilisation)
  - @pytest.fixture pour données jouets / clients API
- Structure conseillée
  - tests/unit/, tests/contract/, tests/data/, tests/model/, tests/integration/

```python
def test_sanity():
    assert 1 + 1 == 2
```

Unit test : logique pure

- Cible : fonctions de transformation / parsing / validation
- Données : petites, en mémoire, reproductibles

```python
# src/features.py
def normalize_age(age: int) -> float:
    if age < 0:
        raise ValueError("age must be >= 0")
    return min(age, 100) / 100.0


# tests/unit/test_features.py
import pytest
from src.features import normalize_age

def test_normalize_age_ok():
    assert normalize_age(50) == 0.5
    assert normalize_age(200) == 1.0  # cap à 100

def test_normalize_age_negative_raises():
    with pytest.raises(ValueError):
        normalize_age(-1)
```

Pytest fixtures : partager des “objets de test” propres et réutilisables

- Problème
  - duplication de setup (données jouets, config, client API)
  - tests difficiles à lire / maintenir
- Solution : fixture
  - fonction décorée @pytest.fixture
  - injectée automatiquement par nom de paramètre
  - scope possible : function (défaut), module, session
- Bon usage
  - données petites, déterministes
  - helpers communs (client FastAPI, sample DataFrame)

Pytest fixtures : partager des “objets de test” propres et réutilisables

```python
import pytest
import pandas as pd
from fastapi.testclient import TestClient
from src.api import app

@pytest.fixture
def sample_df():
    return pd.DataFrame({
        "user_id": ["u1", "u2", "u3"],
        "age": [25, 40, 31],
        "country": ["FR", "FR", "ES"],
    })

@pytest.fixture
def api_client():
    return TestClient(app)

def test_make_features_no_nan(sample_df):
    out = make_features(sample_df)
    assert out.isna().sum().sum() == 0

def test_predict_contract(api_client):
    r = api_client.post("/predict", json={"user_id": "u1", "as_of": "2025-01-01"})
    assert r.status_code == 200
```

Contract test : API stable (schéma request/response)

- Cible : contrat (inputs/outputs) indépendamment du “meilleur modèle”
- Outil : fastapi.testclient (pas besoin de réseau)

```python
# tests/contract/test_api_contract.py
from fastapi.testclient import TestClient
from src.api import app  # FastAPI app

client = TestClient(app)

def test_predict_contract():
    payload = {"user_id": "u_123", "as_of": "2025-01-01"}
    r = client.post("/predict", json=payload)

    assert r.status_code == 200
    body = r.json()

    # champs attendus
    assert "user_id" in body
    assert "proba" in body

    # types attendus
    assert isinstance(body["proba"], float)
    assert 0.0 <= body["proba"] <= 1.0
```

Data tests : philosophie + exemples concrets

- Pourquoi : les données cassent souvent sans “crash”
- Exemples de checks
  - **schema** : colonnes, types
  - **completeness** : taux de valeurs manquantes ≤ seuil
  - **ranges** : bornes numériques, dates valides
  - **categorical** : valeurs autorisées, cardinalité max
  - **volume** : nombre de lignes dans une plage attendue
- Où les placer
  - CI : checks légers (détecter breaking changes)
  - pipeline data : checks complets (qualité et drift)

Data test : invariants sur données/features

- Cible : schéma + règles simples (light checks)
- But : détecter breaking changes (colonne manquante, null-rate explosif…)

```python
# tests/data/test_data_invariants.py
import pandas as pd

REQUIRED_COLS = {"user_id", "age", "country", "label"}

def test_schema_and_null_rate():
    df = pd.read_parquet("data/features_sample.parquet")  # sample contrôlé

    assert REQUIRED_COLS.issubset(df.columns)

    null_rate_age = df["age"].isna().mean()
    assert null_rate_age <= 0.01, f"null_rate_age={null_rate_age:.3f}"

def test_ranges():
    df = pd.read_parquet("data/features_sample.parquet")
    assert (df["age"].dropna() >= 0).all()
    assert (df["age"].dropna() <= 120).all()
```

Model tests offline : ce qu'on garantit

- Quality gate
  - métrique ≥ minimum (floor)
- Non-régression
  - nouveau modèle ≥ production + marge (stabilité)
- Compatibility gate
  - même signature / mêmes features requises
  - output stable (proba/logit, nom de champs)
- Rappel
  - ces tests sont des garde-fous, pas une preuve d'optimalité

Model test : qualité minimale + non-régression + compatibilité

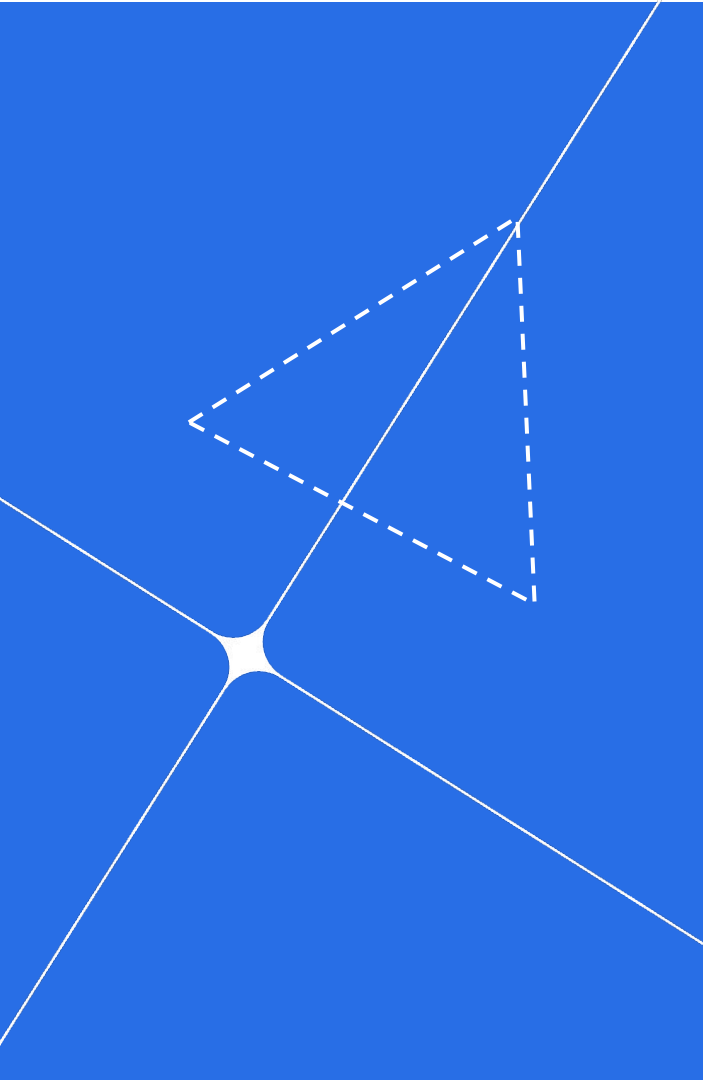
- Cible : empêcher un modèle “pire” ou incompatible de passer en prod
- Sans détour statistique : seuils + marge

```python
# tests/model/test_model_gates.py
def test_quality_gate(metrics_new):
    # fixture: dict {"auc": ..., "logloss": ...}
    assert metrics_new["auc"] >= 0.70

def test_non_regression(metrics_new, metrics_prod):
    margin = 0.01
    assert metrics_new["auc"] >= metrics_prod["auc"] + margin

def test_signature_compatibility(signature_new, signature_prod):
    # signature = {"features": [...], "output": "..."}
    assert signature_new["features"] == signature_prod["features"]
    assert signature_new["output"] == signature_prod["output"]
```

Integration test : système complet

- Cible : vérifier que la stack démarre et répond (réseau, config, dépendances)
- Principe : ping /health puis /predict

```python
# tests/integration/test_stack_smoke.py
import time
import requests

BASE_URL = "http://localhost:8000"

def wait_until_ready(timeout_s=60):
    t0 = time.time()
    while time.time() - t0 < timeout_s:
        try:
            r = requests.get(f"{BASE_URL}/health", timeout=1)
            if r.status_code == 200:
                return True
        except requests.RequestException:
            pass
        time.sleep(1)
    return False

def test_stack_predict_smoke():
    assert wait_until_ready(), "API not ready"

    payload = {"user_id": "u_123", "as_of": "2025-01-01"}
    r = requests.post(f"{BASE_URL}/predict", json=payload, timeout=3)

    assert r.status_code == 200
    assert "proba" in r.json()
```



GitHub Actions : pipeline CI exécutable

GitHub Actions : concepts minimum

- Déclencheurs (events)
  - push, pull_request
- **Workflow** = fichier YAML dans .github/workflows/
- **Job** = suite d’étapes sur un runner (VM)
- **Step** = commande (checkout, install, tests…)
- Résultat
  - statut PASS/FAIL
  - logs consultables
  - artefacts possibles (logs, rapports)

Lire un run CI : quoi regarder, dans quel ordre

1. Statut global : quel job échoue ?
2. Logs du job
   - étape qui échoue (souvent visible immédiatement)
   - message d'erreur + stack trace
3. Indices fréquents
   - dépendances manquantes
   - variables d'environnement
   - service non prêt (healthcheck)
   - timeout réseau
4. Artefacts (si upload en échec)
   - docker compose logs
   - rapports de tests (pytest)

Architecture CI du projet

- Runner GitHub exécute :
  - docker build (images)
  - docker compose up (stack)
  - pytest (tests)
- Tout est local au runner
  - pas de credentials externes
  - reproductible : même workflow pour tous
- Pré-requis pour stabilité
  - healthchecks (services prêts)
  - timeouts maîtrisés
  - logs exploitables

Workflow CI : structure recommandée

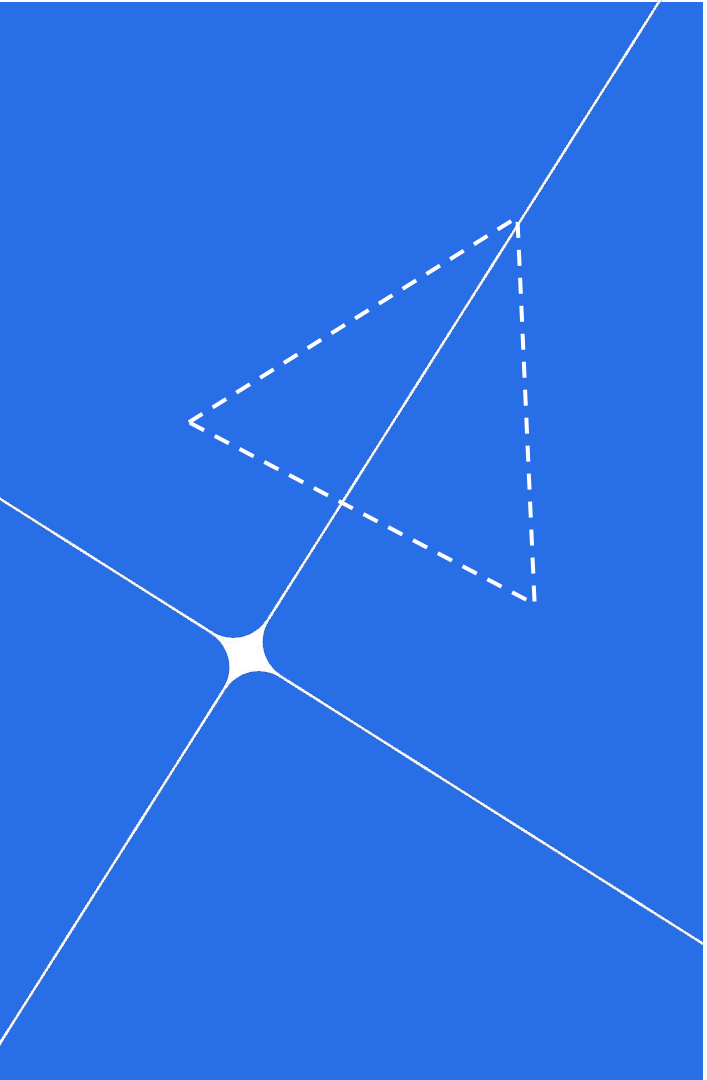
- Étapes typiques
  - checkout repo
  - setup Python
  - install deps
  - (option) lint
  - unit + contract tests
  - démarrage compose + integration tests
  - data checks “light”
  - upload logs en cas d’échec
- Philosophie
  - **fail fast**
  - tests rapides d’abord
  - intégration ensuite (plus coûteux)

Tests d'intégration en CI : "FastAPI smoke test"

- Objectif
  - vérifier que la stack démarre et que l'API répond
- Pattern
  - compose up -d
  - attendre /health (ou healthcheck)
  - appeler /predict
  - vérifier contrat (HTTP 200, champs attendus)
- Bonnes pratiques CI
  - pas de "sleep(30)" fixe si possible
  - timeouts explicites
  - logs compose en artefacts si fail

Limites assumées de la CI (et pourquoi c'est OK)

- CI ≠ pipeline de training complet
  - entraînement lourd : lent, coûteux
  - CI doit rester rapide et fiable
- Ce que CI garantit
  - qualité du code + intégration système
  - contrat API stable
  - invariants data "anti-breaking change"
- Ce qui tourne ailleurs
  - entraînement + évaluation + comparaison + promotion (orchestrateur)
- Message clé
  - CI = garde-fou à chaque changement
  - CT/promotion = amélioration continue pilotée par données/monitoring



Livraison continue des modèles : promotion, gates, rollback

Model Registry : versionner un modèle “servable” (Rappel)

- Rôle du registry
  - centraliser artefacts (modèle, preprocess, signature)
  - versioning + traçabilité
  - point d’intégration avec le serving (quel modèle servir ?)
- Notions clés
  - Run (expérience) : produit une version de modèle
  - métadonnées : params, métriques, dataset tag, code version (commit)
- Production-ready (rappel)
  - modèle = artefact + contrat + observabilité

Stratégies de promotion

- Manuelle
  - un humain choisit la version à promouvoir
  - contrôle fort, mais lent et peu scalable
- Semi-automatique
  - training + évaluation auto
  - promotion avec approval (ex : reviewer)
  - bon compromis pour systèmes à risque
- Automatique
  - règles : métrique + marge + checks compatibilité
  - rapide et itératif, mais nécessite de bons garde-fous
- Risk-based
  - politique dépend de la criticité (finance/santé vs recommandation)
  - plus de gates + validation humaine pour cas sensibles

Gating : exemples de critères “offline”

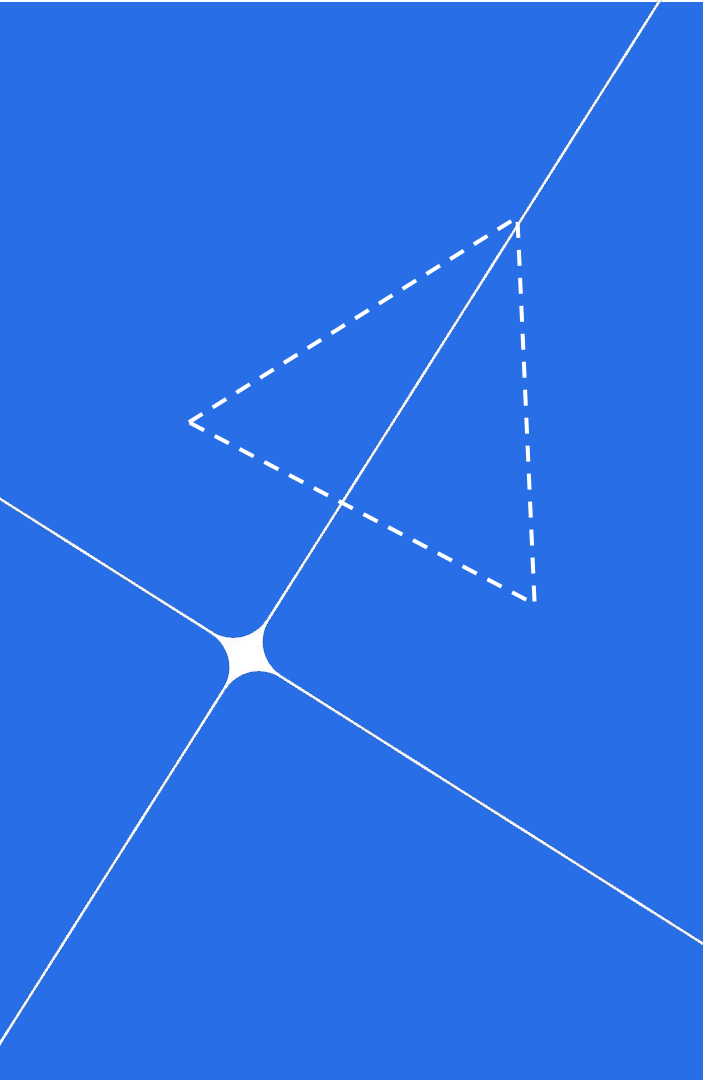
- Quality gate
  - métrique ≥ seuil minimal (floor)
  - non-régression vs baseline/Production (+ marge)
- Compatibility gate
  - signature identique (features attendues, types, output)
  - invariants API (contrat stable)
- Safety/robustness gate (light)
  - sanity checks : distribution outputs, taux de positifs, NaN
  - latence inference sur batch de test (grossier)
- Important
  - gates = “stop the bleeding” (réduire risques), pas “optimiser”

Rollback : stratégie essentielle

- Pourquoi rollback
  - métriques offline trompeuses
  - incidents prod : latence, bugs, dérive inattendue
- Principe
  - conserver l'historique des versions servies
  - capacité à revenir à N-1 en minutes
- Déclencheurs typiques
  - SLO (objectif de niveau de service) violé (latence, erreurs)
  - alerting drift/anomalies
  - incidents business (si métrique online existe)
- Traceability
  - "qui a promu quoi, quand, pourquoi"

Promotion registry ≠ Déploiement trafic

- Promotion (registry)
  - décision : “ce modèle est candidat à être servi”
  - change un label / stage (Production)
- Déploiement trafic
  - décision : “combien de requêtes vont vers quel modèle”
  - nécessite routing (gateway/app/serving layer), monitoring fin
- Pourquoi séparer
  - sécurité : on peut promouvoir sans exposer 100% trafic
  - permet canary / A/B / shadow (section suivante)



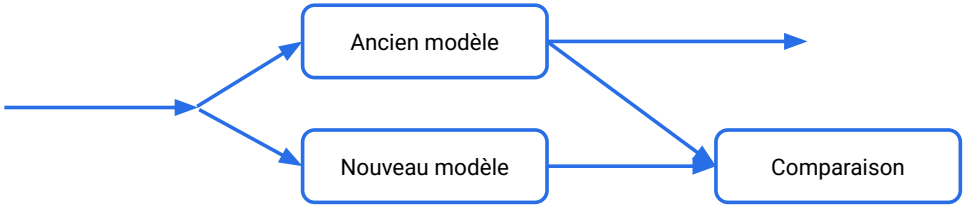
Stratégies de déploiement modèles

Panorama : stratégies de déploiement

- Objectifs
  - réduire risque d'incident
  - mesurer impact (qualité, latence, stabilité)
  - contrôler l'exposition (progressif)
- 2 axes de choix
  - besoin de feedback online (ici : faible / absent)
  - niveau de risque acceptable
- Vocabulaire
  - shadow (miroir), A/B (split), canary (progressif)
  - interleaving (ranking), bandits (adaptatif)

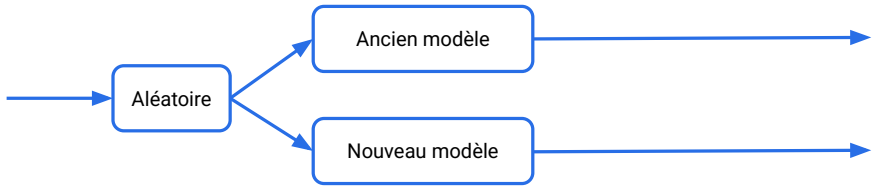
Shadow deployment (mirroring)

- Principe
  - même requête envoyée à modèle prod + modèle candidat
  - seule réponse prod est utilisée (candidat “observe”)
- Ce qu’on mesure
  - latence, erreurs, timeouts
  - divergence outputs (statistiques)
- Avantages / limites
  - très sûr (pas d’impact client direct)
  - ne mesure pas un KPI métier (si pas de feedback)
- Où ça vit techniquement
  - gateway / service mesh / application layer
  - nécessite duplication contrôlée des requêtes



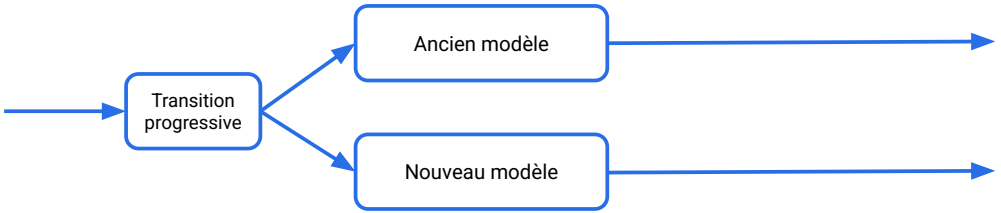
A/B testing (trafic split)

- Principe
  - une fraction du trafic va vers A, l'autre vers B
  - comparaison sur métriques observées (perf, erreurs, éventuellement KPI)
- Pré-requis (important)
  - instrumentation solide, assignation stable
  - définition claire des métriques
- Dans ce cours (sans feedback online)
  - intérêt surtout conceptuel
  - on peut néanmoins mesurer : latence, erreurs, stabilité output
- Où ça vit
  - gateway/router ou application



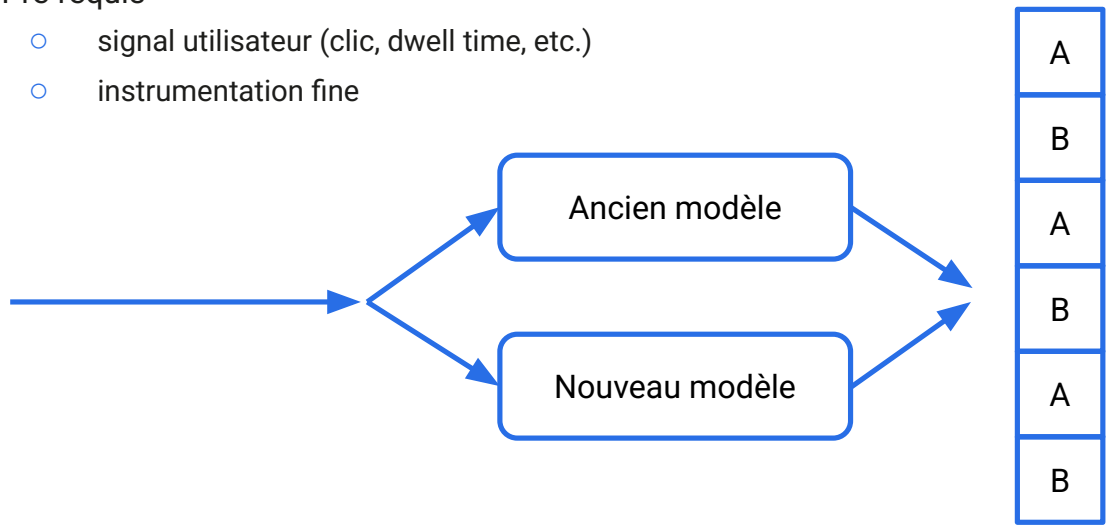
Canary release (progressive rollout)

- Principe
  - exposition progressive : 1% → 5% → 25% → 100%
  - conditions d'arrêt : seuils (erreurs, latence, anomalies)
- Avantages
  - limite blast radius
  - rollback rapide si dérive / bug
- Pré-requis
  - monitoring (latence, taux d'erreur, saturation)
  - alerting + runbook rollback
- Où ça vit
  - gateway/service mesh (idéal)
  - ou application layer (routing interne)



Interleaving (pour ranking/reco) : concept

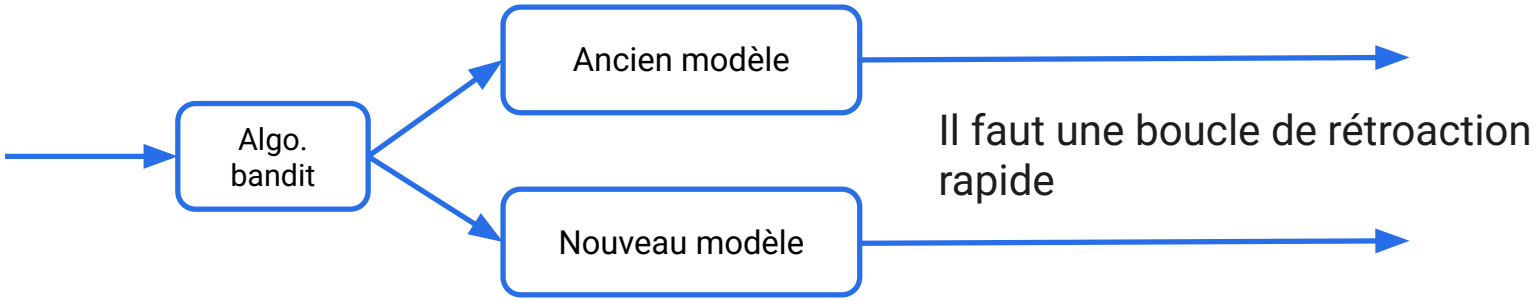
- Quand
  - systèmes de ranking (recherche, recos)
- Principe
  - mélanger résultats de deux modèles dans une même liste
  - feedback utilisateur plus “efficace” qu’un A/B classique
- Pré-requis
  - signal utilisateur (clic, dwell time, etc.)
  - instrumentation fine

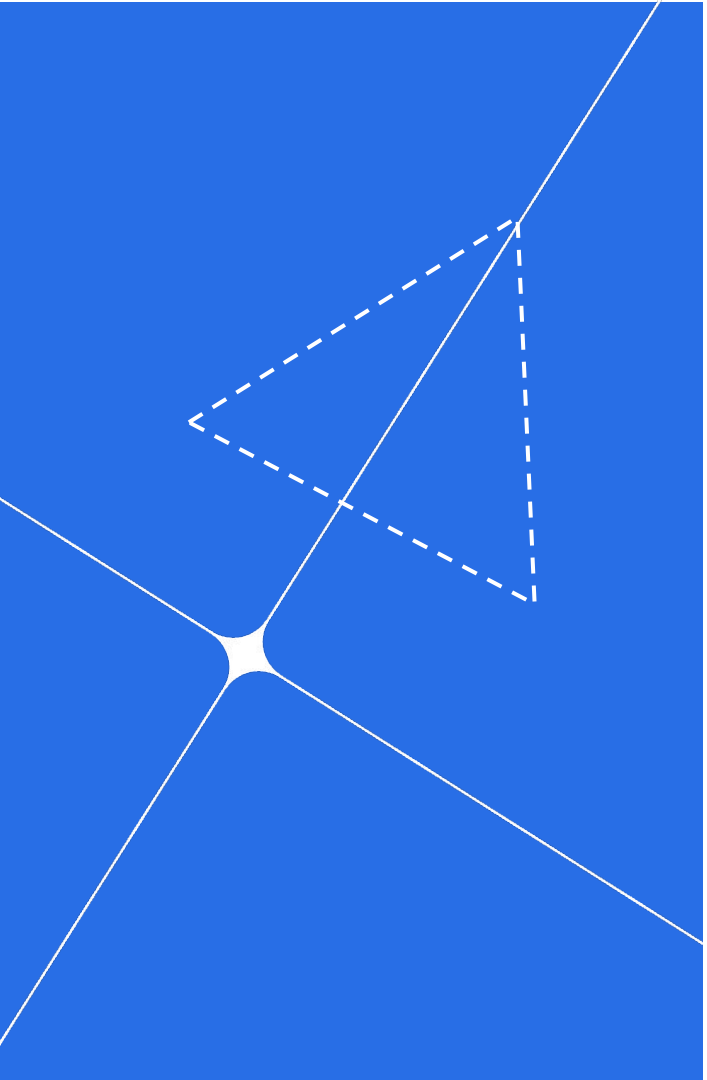


Attention de choisir aléatoirement le premier à afficher

Bandits (multi-armed) : concept et prérequis

- Principe
  - choisir dynamiquement le modèle à servir (exploration/exploitation)
  - maximise reward à court terme
- Pré-requis
  - reward online rapide et fiable
  - gestion de biais, monitoring, garde-fous
- Risques
  - instabilité, sur-optimisation court terme
  - complexité opérationnelle





Continuous learning et pourquoi c'est un problème d'infra

Niveaux de “continuous learning”

- **Niveau 0 : Manual**
  - entraînement ponctuel, déploiement manuel
- **Niveau 1 : Automatisé (schedule)**
  - retrain périodique (ex : chaque mois)
- **Niveau 2 : Automatisé + stateful**
  - conservation d’état : features historisées, lineage, backfills
- **Niveau 3 : Event-triggered**
  - retrain déclenché par événements (drift, data arrival, alert)
- Positionnement du cours
  - objectif : event-triggered (drift => retrain) + promotion contrôlée

Pourquoi “continuous learning” = problème d’infrastructure

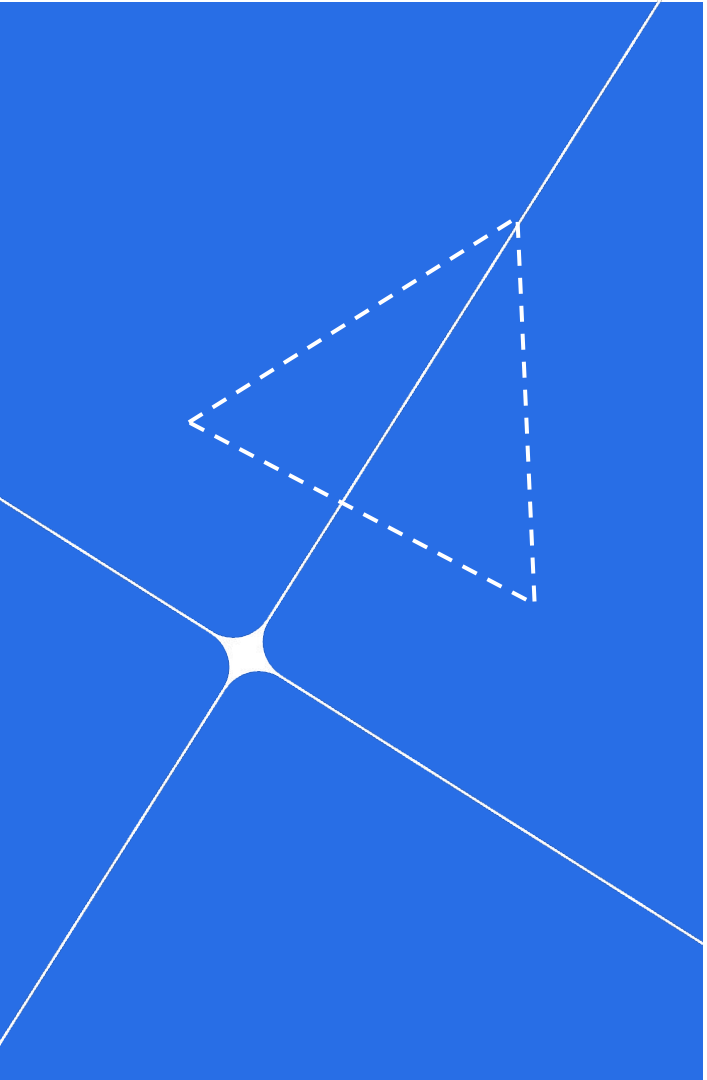
- Problèmes “non-ML” mais critiques
  - **data freshness** (arrivée des données, SLA)
  - **compute budget** (coût, durée, contention)
  - **reproductibilité** (versions libs, config, seeds, datasets)
- Problèmes “système”
  - **orchestration fiable** (retries, idempotence)
  - **observabilité** (logs, métriques, traces)
  - **gouvernance** (qui a promu quoi, audit)
- Message clé
  - automatiser sans garde-fous = usine à incidents
  - MLOps = engineering de fiabilité + contrôle du risque

Qui fait quoi : CI vs Orchestrateur vs Monitoring

- CI (GitHub Actions)
  - qualité du code + intégration (compose + API smoke)
  - bloque les régressions “software/system”
- Orchestrateur (Prefect)
  - pipelines data/model : train → eval → compare → promote/rollback
  - exécutions traçables, relançables
- Monitoring / Drift
  - signaux de prod (latence, erreurs, drift)
  - déclencheurs (event) + alerting
- Séparation des responsabilités
  - CI ≠ retraining
  - Monitoring ≠ décision de merge
  - Registry ≠ routage trafic

Diagramme final : système complet intégré (boucle fermée)

- Flux “code”
  - commit/PR → CI tests → build → stack déployable
- Flux “data/model”
  - nouvelles données → validation → features → train → eval
  - compare vs prod → promote (ou non) → rollback possible
- Flux “prod”
  - API sert un modèle “Production-ready”
  - monitoring + drift → événement → retrain
- Ce que TP6 apport
  - boucle automatisée, testée, observable, récupérable (rollback)



Aperçu du TP

TP6 : ce que vous allez construire

- Objectif : rendre le système production-like
- Étapes
  1. Ajouter logique train → eval → compare
  2. Implémenter promotion (selon policy) + rollback possible
  3. Connecter drift → retraining (event-triggered)
  4. Mettre en place CI GitHub Actions
     - unit/contract/data light/integration smoke
  5. Valider le cycle complet
     - “nouvelle donnée” → drift → retrain → décision → modèle servi
- Ce que vous devez observer
  - run CI (green/red) + logs
  - exécutions orchestrateur (logs + décisions)
  - registry (versions + stage)
  - API (contrat + latence de base)

À retenir

- CI/CD en ML = qualité système + qualité modèle
  - CI : empêche régressions de code/intégration
  - CT/promotion : empêche régressions modèle
- “Production” = servable pour un client
  - contrat API, latence, observabilité, traçabilité
- Déploiement modèle = stratégie de risque
  - shadow / A/B / canary / rollback
  - promotion registry ≠ routage trafic
- Continuous learning = engineering d’infrastructure
  - data freshness, budget compute, reproductibilité, gouvernance