

### **Consignes :**

- Former des groupes de 5 étudiants maximum et désigner un chef d'équipe chargé d'organiser les réunions et de présenter l'avancement aux professeurs.
- Créer un tableau Trello avec les listes : Stories, À faire, En cours, Terminé, Testé, Validé. Et partager le tableau aussi avec les professeurs.
- Choisir un projet pour chaque groupe dans le fichier Excel :  
[https://docs.google.com/spreadsheets/d/16qjnTsemjOuMnZH3kpAyAfzuHTvb47YzCC  
Kz-TCXFxO/edit?usp=sharing](https://docs.google.com/spreadsheets/d/16qjnTsemjOuMnZH3kpAyAfzuHTvb47YzCCKz-TCXFxO/edit?usp=sharing)
- Ne pas répéter un même sujet dans la même classe.

## 1) AgroTrace-MS — Surveillance des cultures et irrigation

### Contexte :

Les agriculteurs font face à des pertes importantes dues à des maladies végétales et à une mauvaise gestion de l'eau. Les données provenant des capteurs IoT, des stations météo et des drones agricoles restent souvent dispersées et peu exploitées.

### Objectif :

Créer une plateforme intelligente capable de détecter automatiquement les maladies des plantes à partir d'images UAV (drones) et de recommander un plan d'irrigation optimal selon les conditions environnementales, tout en visualisant les résultats sur une carte interactive.

### Architecture (microservices)

#### 1. IngestionCapteurs

- Rôle : Collecter les données issues des capteurs IoT (humidité du sol, température, luminosité) et des stations météorologiques.
- Entrées/Sorties : Reçoit des flux JSON et les stocke dans des séries temporelles.
- API/Technologies : API REST + Kafka pour la diffusion en temps réel.
- Base de données : TimescaleDB, adaptée aux données chronologiques.
- Description : Ce microservice harmonise les flux capteurs/météo et vérifie leur cohérence avant de les envoyer au module de prétraitement.

#### 2. Prétraitement

- Rôle : Nettoyer les données brutes (valeurs manquantes, anomalies) et segmenter les images UAV en tuiles exploitables.
- Entrées/Sorties : Reçoit les données capteurs/images → renvoie des séries normalisées et tuiles géoréférencées.
- Technologies : Python (Pandas, GDAL, Rasterio).
- Base de données : MinIO (stockage d'images) et TimescaleDB (séries temporelles).
- Description : Ce service prépare les données pour les modèles d'IA en appliquant des filtres, des corrections radiométriques et des conversions de formats.

#### 3. VisionPlante

- Rôle : Déetecter les maladies foliaires à partir d'images aériennes.

- Entrées/Sorties : Images UAV → masques de segmentation + scores de probabilité.
- Technologies : PyTorch (CNN, ViT, U-Net).
- Base de données : MinIO pour les modèles et les images traitées.
- Description : Ce module applique des réseaux de neurones convolutifs pour identifier des signes de stress ou de maladie et produit des cartes thématiques de santé des cultures.

#### 4. PrévisionEau

- Rôle : Prédire le besoin en eau à court terme pour chaque parcelle.
- Entrées/Sorties : Données capteurs/météo → prévisions du stress hydrique (1–7 jours).
- Technologies : Prophet et LSTM (PyTorch).
- Base de données : TimescaleDB.
- Description : Ce service apprend à partir de l'historique hydrique et climatique afin d'estimer les besoins d'irrigation selon les conditions à venir.

#### 5. RèglesAgro

- Rôle : Appliquer les règles agronomiques (type de sol, stade de croissance, culture).
- Entrées/Sorties : Données prédictives → recommandations textuelles.
- Technologies : Drools (Java Rule Engine).
- Base de données : PostgreSQL (référentiels agricoles).
- Description : Les règles traduisent l'expertise humaine (par exemple : "Si humidité < 20% et température > 30°C, recommander irrigation urgente").

#### 6. Recolrrigation

- Rôle : Fournir des recommandations concrètes : dose, fréquence et calendrier d'irrigation.
- Entrées/Sorties : Prédictions + règles → plan d'action (JSON).
- Technologies : FastAPI (Python), OpenAPI pour documentation.
- Base de données : PostgreSQL (historique des conseils).
- Description : Génère des recommandations personnalisées par zone et les envoie au tableau de bord via API REST.

## 7. DashboardSIG

- Rôle : Visualiser l'état sanitaire et hydrique des parcelles sur une carte interactive.
- Entrées/Sorties : Données consolidées → affichage Web SIG.
- Technologies : React.js, Leaflet/MapLibre pour la carte, API REST.
- Base de données : PostGIS (stockage spatial).
- Description : Le tableau de bord permet à l'agriculteur de visualiser les zones stressées, les prévisions et les recommandations, avec possibilité d'export en PDF ou CSV.

### Résultat attendu :

Un système modulaire déployé sous Docker, capable d'intégrer des données hétérogènes et de générer des recommandations d'irrigation fiables et reproductibles, documentées dans un pipeline ouvert conforme aux exigences SoftwareX.

## 2) RoadSense-MS — Analyse automatique des routes

### Contexte :

Les infrastructures routières se détériorent rapidement sous l'effet du trafic et des conditions climatiques. L'inspection manuelle des routes est coûteuse et peu fréquente, ce qui entraîne des retards dans les réparations et une allocation inefficace des ressources publiques. Une solution automatisée permet de détecter les dégradations à partir de vidéos et d'images, d'évaluer leur gravité et de planifier la maintenance de manière proactive.

### Objectif :

Développer un système intelligent capable d'analyser automatiquement les routes à partir de vidéos embarquées (dashcam ou drone) pour identifier les fissures, nids-de-poule et autres dégradations, de les géolocaliser avec précision et de générer une carte de priorisation des interventions.

### Architecture (microservices)

#### 1. IngestionVideo

- **Rôle :** Extraire les images, les métadonnées GPS et l'orientation à partir des vidéos brutes captées par des véhicules ou drones.
- **Entrées/Sorties :** Vidéos (MP4, AVI) → frames JPEG + données GPS/IMU.
- **API / Communication :** POST /video/upload pour dépôt et extraction automatique.
- **Technologies :** Go (pour performance), GStreamer, FFmpeg pour la découpe.
- **Base de données :** MinIO pour stocker les images extraites.
- **Description :** Ce service décompose les vidéos en images exploitables, les horodate et les synchronise avec les coordonnées GPS afin d'assurer la traçabilité des défauts détectés.

#### 2. DetectionFissures

- **Rôle :** Identifier automatiquement les fissures, trous ou dégradations à partir des images extraites.
- **Entrées/Sorties :** Images JPEG → boîtes de détection (bounding boxes) + masques de segmentation.
- **API :** POST /detect pour lancer la détection.
- **Technologies :** PyTorch (modèles YOLOv8, Mask R-CNN), CUDA pour accélération GPU.
- **Base de données :** MinIO pour stocker les images annotées et les résultats.

- **Description** : Ce service applique des modèles de vision par ordinateur pour repérer les anomalies visibles sur la chaussée et attribue un score de confiance à chaque détection.

### 3. GeoRef

- **Rôle** : Associer chaque défaut détecté à une position géographique précise sur le réseau routier.
- **Entrées/Sorties** : Fissures détectées + coordonnées GPS → entités géospatiales (tronçons).
- **API** : POST /georef pour géoréférencer les anomalies.
- **Technologies** : Python (OSRM, Shapely, GeoPandas).
- **Base de données** : PostGIS (extension spatiale de PostgreSQL).
- **Description** : Ce microservice effectue le “map-matching” des détections avec le graphe routier et permet de visualiser chaque défaut sur une carte.

### 4. ScoreGravite

- **Rôle** : Calculer la gravité de chaque défaut selon sa taille, sa profondeur et la densité locale des anomalies.
- **Entrées/Sorties** : Données de détection → scores de gravité.
- **API** : POST /severity/compute pour calculer le score.
- **Technologies** : Python (XGBoost, Scikit-learn).
- **Base de données** : PostgreSQL pour stocker les scores et rapports.
- **Description** : Ce module traduit les observations visuelles en un score de risque utilisable pour la priorisation des réparations.

### 5. Priorisation

- **Rôle** : Classer les tronçons de route selon leur niveau de dégradation et l'importance du trafic.
- **Entrées/Sorties** : Scores de gravité + trafic → liste priorisée.
- **API** : GET /priority/list pour accéder au classement.
- **Technologies** : Node.js/NestJS pour logique métier et orchestration.

- **Base de données :** PostgreSQL.
- **Description :** Ce microservice combine plusieurs critères (coût, gravité, accessibilité, importance du tronçon) pour générer un plan d'intervention optimal.

## 6. ExportSIG

- **Rôle :** Exporter les résultats (défauts et priorités) sous formats compatibles avec les systèmes d'information géographique (SIG).
- **Entrées/Sorties :** Données PostGIS → GeoJSON, WMS, WFS, MBTiles.
- **API :** GET /export/map pour générer les couches SIG.
- **Technologies :** GeoServer, GDAL.
- **Base de données :** PostGIS.
- **Description :** Ce service rend les données exploitables dans des logiciels SIG externes (QGIS, ArcGIS) pour faciliter les rapports techniques et la planification.

## 7. Dashboard

- **Rôle :** Interface web pour visualiser l'état des routes, les détections, les scores et les priorités.
- **Entrées/Sorties :** API agrégées → tableaux et cartes interactives.
- **API :** GET /dashboard (données agrégées).
- **Technologies :** React.js, Leaflet/MapLibre pour les cartes, Chart.js pour les statistiques.
- **Base de données :** PostgreSQL (requêtes agrégées).
- **Description :** Le tableau de bord fournit une vision d'ensemble : état du réseau, zones critiques, statistiques par région, et rapports exportables.

### Résultat attendu :

Un système complet, déployable via Docker, capable d'analyser automatiquement les routes à partir de vidéos, de géolocaliser les défauts, d'évaluer leur gravité et de produire une cartographie dynamique de la maintenance.

### 3) SafeOps-LogMiner — Analyse de logs CI/CD et détection de failles DevSecOps

#### Contexte

Les chaînes DevOps modernes s'appuient fortement sur des pipelines d'intégration et de déploiement continus (CI/CD). Cependant, ces chaînes sont souvent vulnérables à des erreurs de configuration, des fuites de secrets, ou des mauvaises pratiques (actions non "pinnées", permissions excessives, etc.). Les logs de ces pipelines sont rarement analysés automatiquement, ce qui laisse passer de nombreuses failles de sécurité.

#### Objectif

Fournir une plateforme modulaire de microservices permettant d'analyser les logs d'exécution des pipelines CI/CD (GitHub Actions, GitLab CI, Jenkins), d'identifier automatiquement les faiblesses de sécurité, de suggérer des correctifs, et de générer des rapports de conformité alignés avec les standards OWASP, SLSA et CIS.

#### Architecture (microservices)

##### 1. LogCollector

- **Rôle :** Collecter les logs CI/CD (GitHub, GitLab, Jenkins) via API ou webhook.
- **API :** POST /logs/upload ou GET /logs/github
- **Technologies :** Node.js + Express
- **Base de données :** MongoDB (NoSQL pour stockage rapide de logs bruts)
- **Description :** Ce microservice s'interface avec les plateformes CI pour extraire les logs des workflows, soit en temps réel (webhooks), soit à la demande (pull API), et les stocke avec leurs métadonnées (auteur, pipeline, date, repo...).

##### 2. LogParser

- **Rôle :** Analyser les logs et extraire des événements sémantiques (jobs, erreurs, secrets, URLs).
- **API :** POST /logs/parse
- **Technologies :** Python (regex, parsing YAML, jsonpath)
- **Base de données :** MongoDB

- **Description :** Ce microservice transforme les logs bruts en structures exploitables. Il détecte les étapes du pipeline, identifie les artefacts produits et repère les anomalies visibles (secrets, erreurs, tentatives de bypass...).

### 3. VulnDetector

- **Rôle :** Identifier les vulnérabilités DevOps courantes selon une base de règles.
- **API :** POST /scan
- **Technologies :** Python + moteur de règles basé sur YAML
- **Base de données :** PostgreSQL (pour règles mappées OWASP, SLSA)
- **Description :** Ce service applique des règles de sécurité déclaratives sur les logs parsés pour repérer : actions non "pinnées", secrets exposés, permissions trop larges, jobs non isolés, etc. Il génère un rapport de vulnérabilités annoté.

### 4. FixSuggester

- **Rôle :** Générer des patchs YAML pour corriger les problèmes détectés (auto-fix).
- **API :** POST /fix
- **Technologies :** Python (Jinja2 templates, diff-match-patch)
- **Base de données :** PostgreSQL
- **Description :** Ce microservice propose des corrections automatiques sûres : verrouillage des versions d'actions, restriction de permissions, ajout de secrets chiffrés, amélioration du hardening, etc. Chaque correctif est fourni sous forme de diff YAML commenté.

### 5. AnomalyDetector

- **Rôle :** Appliquer des modèles ML/IA pour repérer des comportements anormaux ou suspects.
- **API :** POST /anomaly
- **Technologies :** Scikit-learn + IsolationForest, Autoencoder
- **Base de données :** TimescaleDB
- **Description :** Ce service complète la détection statique par une analyse comportementale : détection d'exécutions inhabituelles, jobs qui changent de

comportement, délais anormaux, etc. Il apprend de l'historique pour identifier les écarts.

## 6. ReportGenerator

- **Rôle :** Générer un rapport global de sécurité DevOps (PDF, HTML, SARIF).
- **API :** GET /report/:id
- **Technologies :** Node.js (PDFKit, handlebars), SARIF format
- **Base de données :** PostgreSQL
- **Description :** Fournit un rapport lisible intégrant la gravité des failles, des métriques (score sécurité), les suggestions de correction, et l'historique d'évolution des pipelines. Intégrable dans GitHub ou GitLab comme artefact.

## 7. Dashboard

- **Rôle :** Vue frontale de la santé CI/CD (failles, alertes, tendances).
- **API :** GET /dashboard
- **Technologies :** React.js + TailwindCSS + Chart.js
- **Base de données :** PostgreSQL
- **Description :** Interface web avec filtres, scores de sécurité par pipeline, timeline des alertes, courbes de progression, et historique des correctifs appliqués.

### Résultat attendu

Une plateforme sécurisée en microservices pour surveiller, auditer et corriger automatiquement les failles de sécurité dans les chaînes CI/CD. Utile pour les DevSecOps, les RSSI, et les projets open source souhaitant garantir une livraison logicielle fiable et conforme aux bonnes pratiques.

## 4) EcoLabel-MS — Score environnemental des produits

### Contexte

Les consommateurs souhaitent connaître l'impact écologique réel des produits qu'ils achètent : origine, composition, transport, emballage, etc. Or ces informations sont souvent dispersées dans des fiches techniques ou difficilement comparables. Une solution automatisée, transparente et reproductible est nécessaire pour estimer un **éco-score** fiable à partir de sources multiples.

### Objectif

Créer une plateforme en microservices capable de calculer un score environnemental dynamique pour chaque produit, en combinant des données issues des étiquettes, de la composition, de la logistique et des facteurs d'analyse du cycle de vie (ACV). L'objectif est d'obtenir un score clair (A–E) accompagné d'explications et d'une traçabilité complète des données.

### Architecture (microservices)

#### 1. ParserProduit

- **Rôle :** Extraire automatiquement les informations textuelles des fiches produits, des codes-barres (GTIN) ou d'images.
- **Entrées/Sorties :** Fichiers (PDF, HTML, image) → JSON normalisé.
- **API :** POST /product/parse (pour importer un lot de produits).
- **Technologies :** Python, Tesseract (OCR), BeautifulSoup.
- **Base de données :** PostgreSQL (métadonnées produits).
- **Description :** Ce service nettoie et uniformise les données textuelles et visuelles ; il gère la reconnaissance de texte et la correspondance entre les fiches produits et leurs codes GTIN.

## 2. NLPIngrédients

- **Rôle :** Identifier et normaliser les ingrédients, matériaux d'emballage et lieux de provenance.
- **Entrées/Sorties :** Texte brut → liste d'entités normalisées.
- **API :** POST /nlp/extract.
- **Technologies :** spaCy + Transformers (BERT multilingue).
- **Base de données :** PostgreSQL (taxonomies ingrédients, référentiels EcoInvent).
- **Description :** Détecte automatiquement les ingrédients, les labels (bio, recyclé...), et associe chaque élément à une base de données environnementale pour évaluer son impact individuel.

## 3. LCALite

- **Rôle :** Calculer une analyse du cycle de vie simplifiée (ACV) pour chaque produit.
- **Entrées/Sorties :** Ingrédients + transport → indicateurs CO<sub>2</sub>, eau, énergie.
- **API :** POST /lca/calc.
- **Technologies :** Python + pandas ; formules d'ACV issues de bases FAO et Ademe.
- **Base de données :** PostgreSQL (poids des facteurs) + MinIO (stockage des fichiers ACV).
- **Description :** Ce microservice calcule les impacts globaux et pondère les résultats selon le type de produit et son emballage (plastique, verre, papier).

## 4. Scoring

- **Rôle :** Générer le score final (A–E) avec incertitude et explication.
- **Entrées/Sorties :** Indicateurs ACV → éco-score numérique + lettre.
- **API :** POST /score/compute.
- **Technologies :** FastAPI (Python), scikit-learn (pondération et normalisation).
- **Base de données :** PostgreSQL (scores et historiques).
- **Description :**  
Agrège les différents indicateurs et produit un score compréhensible par le consommateur avec explication des poids et niveau de confiance.

## 5. WidgetAPI

- **Rôle :** Fournir les scores via API publique et interface web.
- **Entrées/Sorties :** Scores calculés → affichage web ou mobile.
- **API :** GET /public/product/:id.
- **Technologies :** React + FastAPI ; format GraphQL disponible.
- **Base de données :** PostgreSQL (catalogue produits publiques).
- **Description :**  
Ce microservice rend le score disponible pour les sites e-commerce ou applications mobiles et permet aux consommateurs de consulter le détail de l'impact par catégorie.

## 6. Provenance

- **Rôle :** Assurer la traçabilité des données et versions de calcul.
- **Entrées/Sorties :** Facteurs ACV → métadonnées de version.
- **API :** GET /provenance/:score\_id.
- **Technologies :** DVC (Data Version Control) + MLflow.
- **Base de données :** MinIO (pour les artefacts et datasets).
- **Description :**  
Ce service permet de retracer les données utilisées pour chaque score, assurant la réproductibilité et la fiabilité scientifique des résultats.

## Résultat attendu

Un système complet de notation écologique, interopérable et reproductible :

- calcul d'un éco-score dynamique par produit,
- transparence et traçabilité des sources,
- intégration facile dans les sites marchands ou applications mobiles,
- et publication de benchmarks ou datasets ouverts pour la recherche et l'évaluation de l'impact environnemental.

## 5) EduPath-MS — Learning Analytics & Recommandations

### Contexte

Dans l'enseignement supérieur, de nombreuses plateformes d'apprentissage accumulent des données (notes, interactions, rythme), mais elles sont rarement exploitées pour accompagner les étudiants de manière personnalisée. Face au décrochage scolaire et aux difficultés d'adaptation, un système intelligent de suivi, d'analyse des parcours et de recommandations pédagogiques peut fortement améliorer l'engagement et la réussite.

### Objectif

Développer une plateforme en microservices pour **analyser les trajectoires d'apprentissage** à partir de données issues des LMS (Learning Management Systems) et proposer automatiquement des **recommandations pédagogiques personnalisées**. Elle aide les enseignants à identifier les profils à risque et suggère des ressources ou activités adaptées à chaque étudiant.

### Architecture (microservices détaillés)

#### 1. LMSConnector

- **Rôle** : Synchroniser les données depuis Moodle, Canvas, etc. (notes, connexions, participations).
- **Technologies** : Node.js + REST API + OAuth2.
- **Base de données** : PostgreSQL (logs + identifiants étudiants).
- **Description** : Ce microservice extrait les traces d'apprentissage brutes des plateformes éducatives et les normalise dans un format commun exploitable.

#### 2. PrepaData

- **Rôle** : Nettoyage, normalisation, et agrégation des données temporelles (scores, délais, complétions).

- **Technologies** : Python + pandas + Airflow.
- **Base de données** : PostgreSQL (vue analytique).
- **Description** : Regroupe les informations sur les sessions, calcule des indicateurs (taux d'engagement, de réussite, fréquence d'accès aux contenus).

### 3. StudentProfiler

- **Rôle** : Regroupe les données par étudiant et détecte les typologies (ex : procrastinateur, assidu, en difficulté).
- **Technologies** : scikit-learn + KMeans + PCA.
- **Base de données** : PostgreSQL.
- **Description** : Utilise l'apprentissage non supervisé pour classer les étudiants et construire des profils types sur leur comportement d'apprentissage.

### 4. PathPredictor

- **Rôle** : Prédit les probabilités de réussite/échec d'un étudiant sur un module à venir.
- **Technologies** : XGBoost + MLflow (tracking).
- **Base de données** : PostgreSQL (historique modèles).
- **Description** : Utilise les trajectoires passées pour anticiper les risques et générer des alertes préventives (ex : "attention, retard chronique dans 3 matières").

### 5. RecoBuilder

- **Rôle** : Génère des recommandations ciblées (ressources, vidéos, exercices, tutorat).
- **Technologies** : Transformers (BERT) + Faiss (moteur de similarité).
- **Base de données** : PostgreSQL (ressources) + MinIO (contenus multimédias).
- **Description** : Ce moteur propose automatiquement des contenus pertinents selon les difficultés détectées et les objectifs pédagogiques à atteindre.

### 6. TeacherConsole

- **Rôle** : Interface pour les enseignants (tableau de bord de la classe, alertes, suivi individuel).
- **Technologies** : React + Chart.js.

- **Base de données** : PostgreSQL (restitutions anonymisées ou individuelles).
- **Description** : Vue agrégée des performances, clustering des élèves par profil, suggestions de remédiation collective ou individualisée.

## 7. StudentCoach

- **Rôle** : Interface mobile étudiante (feedback + coaching + suggestions).
- **Technologies** : Flutter + FastAPI.
- **Base de données** : PostgreSQL.
- **Description** : Permet aux apprenants de consulter leur progression, recevoir des messages motivants, des conseils personnalisés, et accéder aux ressources conseillées par RecoBuilder.

## Résultat attendu

Cette plateforme de **Learning Analytics intelligente** permet de :

- détecter les étudiants à risque et les accompagner,
- visualiser les parcours d'apprentissage et les patterns d'échec,
- automatiser les recommandations pédagogiques,
- améliorer l'engagement et la réussite étudiante,
- générer des **benchmarks publics anonymisés** pour publication dans *SoftwareX* et recherche reproductible en éducation numérique.

## 6) MicroLearn — Orchestrateur AutoML par microservices

### Contexte

L'exploration de modèles d'apprentissage automatique (ML) nécessite souvent des compétences avancées et un processus long : nettoyage des données, sélection de modèles, entraînement, évaluation, déploiement. Les solutions AutoML (Automated Machine Learning) restent souvent monolithiques, difficiles à personnaliser et peu ouvertes à l'expérimentation distribuée. Dans un contexte académique ou industriel, il devient crucial de **composer dynamiquement des pipelines ML** pour tester plusieurs approches, itérer rapidement, et répliquer les expériences.

### Objectif

Développer une plateforme modulaire d'**AutoML distribuée** basée sur des microservices, chacun réalisant une étape du pipeline de data mining : préparation, sélection de modèles, entraînement, évaluation, déploiement. L'utilisateur (développeur ou data scientist) pourra composer un pipeline ML sur mesure via API, lancer des entraînements parallèles sur GPU, et comparer les performances. Le tout est conçu pour être **scalable, personnalisable, et réplicable**, conformément aux standards *SoftwareX*.

### Architecture (microservices détaillés)

#### 1. DataPreparer

- **Rôle** : Nettoyage des données, encodage, gestion des valeurs manquantes, normalisation.
- **Technologies** : Python + Pandas + FastAPI.
- **I/O** : CSV/JSON brut → DataFrame nettoyée.
- **Base de données** : PostgreSQL (dataset préparé) + MinIO (dump brut).
- **API** : /prepare?pipeline=[...].

- **Description** : Supporte les pipelines de prétraitement définis via YAML ou JSON (scaling, one-hot, imputation...).

## 2. ModelSelector

- **Rôle** : Sélection automatique de modèles à tester (XGBoost, SVM, RandomForest, CNN).
- **Technologies** : Python + scikit-learn + PyCaret.
- **I/O** : dataset → liste de candidats.
- **Base de données** : PostgreSQL (catalogue modèles).
- **API** : /select?metric=accuracy.
- **Description** : Évalue la compatibilité modèle/données (type, taille, objectif) et propose les modèles à instancier.

## 3. Trainer

- **Rôle** : Entraînement parallèle des modèles sélectionnés.
- **Technologies** : PyTorch Lightning + Ray + Docker GPU.
- **I/O** : modèle + data → checkpoints.
- **Base de données** : MinIO (modèles entraînés).
- **API** : /train?model=XGBoost&data\_id=123.
- **Description** : Gère la répartition GPU, suit la progression, journalise les hyperparamètres (via MLflow).

## 4. Evaluator

- **Rôle** : Évalue les performances (AUC, F1, RMSE) + génération de rapport comparatif.
- **Technologies** : Python + scikit-learn + Plotly.
- **I/O** : checkpoints → métriques + courbes ROC.
- **Base de données** : PostgreSQL (logs de résultats).
- **API** : /evaluate?model\_id=xyz.
- **Description** : Fournit un tableau récapitulatif multi-métriques, trace les erreurs, sauvegarde les performances.

## 5. HyperOpt

- **Rôle** : Optimisation bayésienne ou random search sur hyperparamètres.
- **Technologies** : Optuna + Redis + FastAPI.
- **I/O** : espace de recherche → meilleurs paramètres.
- **Base de données** : PostgreSQL (essais + historique).
- **API** : /optimize?model=RF&target=auc.
- **Description** : Peut fonctionner indépendamment ou orchestré par Trainer, logue les runs, gère les arrêts précoces.

## 6. Deployer

- **Rôle** : Déploiement des modèles en REST, batch ou edge.
- **Technologies** : TorchServe + Flask + Docker.
- **I/O** : modèle → endpoint REST ou conteneur exporté.
- **Base de données** : PostgreSQL (metadata de déploiement).
- **API** : /deploy?model\_id=xyz&type=rest.
- **Description** : Automatise l'emballage du modèle en service, avec interface OpenAPI + auto-logs.

## 7. Orchestrator

- **Rôle** : Coordination du pipeline (prétraitement → déploiement) à partir de YAML ou requêtes API.
- **Technologies** : Node.js + NATS (événements) + API Gateway.
- **Base de données** : Redis (états des tâches).
- **API** : /pipeline/execute, /status/:id.
- **Description** : Orchestration asynchrone, suivi des états (running, failed, success), notifications Webhooks.

## 8. Dashboard

- **Rôle** : Interface de suivi des expériences, modèles, métriques, erreurs.

- **Technologies** : React + D3.js + Chart.js.
- **Base de données** : PostgreSQL (rapports) + MinIO (artefacts).
- **Description** : Vue centralisée sur tous les entraînements, trie par métrique, télécharge les meilleurs modèles, compare les runs.

## Résultat attendu

- Pipelines AutoML **personnalisables et composable par API**,
- Optimisation **parallèle et distribuée** sur GPU,
- Interface web pour explorer les expériences,
- Moteur reproductible et ouvert aux chercheurs (*SoftwareX-ready*),
- Support du **FAIR ML** (versionnage, logs, comparaisons).

## 7) HealthFlow-MS — Analyse FHIR et prédition de réadmission

### Contexte

Dans les systèmes de santé modernes, l'un des défis majeurs est **la réadmission évitable des patients** dans un court délai après leur sortie. Cela engendre des coûts supplémentaires, une surcharge des hôpitaux, et soulève des questions sur la qualité des soins initiaux.

Parallèlement, les normes **FHIR (Fast Healthcare Interoperability Resources)** deviennent la référence pour l'échange structuré de données médicales. De nombreux hôpitaux cherchent à tirer parti des données FHIR pour faire de l'analytique prédictive, tout en respectant la confidentialité et les réglementations (ex : RGPD, HIPAA).

### Objectif

Déployer une architecture modulaire basée sur des microservices pour :

- **extraire, nettoyer et transformer des données FHIR** en variables exploitables,
- **prédir le risque de réadmission** d'un patient à court terme,
- **expliquer et auditer les décisions du modèle**, en assurant l'équité et la conformité.

Cette solution vise à **améliorer la planification des soins**, identifier les patients à risque élevé, et soutenir les décisions cliniques basées sur les données.

### Architecture (microservices détaillés)

#### 1. ProxyFHIR

- **Rôle** : Connecte l'hôpital à la plateforme FHIR (serveur HAPI), récupère les ressources médicales standardisées.
- **Technologies** : Java + HAPI FHIR + PostgreSQL.
- **Description** : Sert de couche d'interopérabilité. Synchronise les données de patients, observations, diagnostics, médicaments, procédures, etc., via REST. Enregistre les FHIR Bundles dans la base PostgreSQL pour traitement interne.

#### 2. DelD

- **Rôle** : Anonymise les données personnelles et sensibles en amont de tout traitement.
- **Technologies** : Python + Faker + PostgreSQL.
- **Description** : Supprime ou remplace les identifiants directs (nom, adresse, ID) par des pseudonymes. Gère l'encodage temporel et la suppression des données sensibles non nécessaires à la prédiction.

### 3. Featurizer

- **Rôle** : Extrait des variables pertinentes à partir des textes médicaux (notes cliniques) et des mesures structurées (signes vitaux, lab tests).
- **Technologies** : Python + BioBERT + spaCy + PostgreSQL.
- **Description** : Combine NLP (reconnaissance d'entités médicales) et extraction statistique. Résume les antécédents, traitements, comorbidités, valeurs aberrantes. Génère une table patient+features exploitable par le modèle.

### 4. ModelRisque

- **Rôle** : Calcule un **score de risque de réadmission** (0-1) à l'aide d'un modèle supervisé.
- **Technologies** : Python + XGBoost + SHAP + PostgreSQL.
- **Description** : Le modèle est entraîné sur des données historiques FHIR. Génère une probabilité de réadmission à 30 jours. Fournit aussi une explication locale (SHAP) pour chaque prédiction.

### 5. ScoreAPI

- **Rôle** : Expose les scores de risque, les explications, et les variables via une API REST sécurisée.
- **Technologies** : FastAPI + JWT Auth + PostgreSQL.
- **Description** : Permet d'interroger les scores par patient ID, date de sortie ou service. Intégration possible dans un EHR, une application mobile clinique, ou une alerte interne.

### 6. AuditFairness

- **Rôle** : Vérifie que les prédictions du modèle sont **justes et non discriminatoires**.
- **Technologies** : EvidentlyAI + Python + Dash + PostgreSQL.

- **Description** : Analyse les biais potentiels par âge, sexe, origine ethnique, comorbidité. Génère des rapports de fairness, d'équité intergroupes, et de dérive temporelle du modèle.

## Résultats attendus

- Réduction des réadmissions évitables grâce à des alertes précoces
- Intégration fluide avec les systèmes FHIR des hôpitaux
- Auditabilité et transparence du modèle prédictif
- Conformité aux exigences éthiques et réglementaires
- Plateforme ouverte, modulable, et prête à être documentée pour **SoftwareX**.

## 8) DocQA-MS — Assistant médical sur documents cliniques (LLM + microservices)

### Contexte

Les établissements de santé possèdent de vastes corpus de documents cliniques non structurés : comptes-rendus, ordonnances, lettres de liaison, résultats de laboratoire, etc. Ces documents sont difficiles à interroger rapidement, ce qui ralentit la prise de décision médicale, la recherche clinique ou encore la réponse à des audits. Les récents modèles de langage (LLMs) permettent de transformer ces données textuelles en réponses précises et contextualisées. Pour autant, leur intégration dans des systèmes hospitaliers requiert modularité, confidentialité et supervision.

### Objectif

Construire un assistant intelligent basé sur des microservices couplés à un LLM (modèle de langage) pour permettre :

- l'interrogation en langage naturel de documents cliniques internes,
- l'extraction ciblée d'informations médicales (pathologies, traitements, antécédents),
- des synthèses contextualisées ou comparatives entre patients,
- tout en garantissant la sécurité des données et la traçabilité des requêtes.

### Architecture (microservices détaillés)

#### 1. DocIngestor

- **Rôle** : Ingestion sécurisée de documents (PDF, DOCX, TXT, HL7, FHIR notes).
- **Technologies** : Python + Tika + OCR + RabbitMQ + PostgreSQL.
- **Description** : Extrait le texte brut, les métadonnées, et indexe les documents dans la base. Prend en charge des dépôts manuels ou automatisés depuis un système hospitalier (EHR).

## 2. DelD

- **Rôle** : Anonymisation automatique des documents avant traitement LLM.
- **Technologies** : spaCy + Presidio + PostgreSQL.
- **Description** : Repère et supprime les identifiants personnels (nom, IPP, numéro sécu, etc.). Peut générer des données synthétiques pour le test en mode bac à sable.

## 3. IndexeurSémantique

- **Rôle** : Embedding des documents dans une base vectorielle pour recherche sémantique.
- **Technologies** : SentenceTransformers + FAISS + PostgreSQL.
- **Description** : Crée des représentations vectorielles sémantiques pour chaque paragraphe ou section. Utilisé en combinaison avec le LLM pour retrouver le contexte pertinent.

## 4. LLMQAModule

- **Rôle** : Répond à une question posée en langage naturel en s'appuyant sur les documents indexés.
- **Technologies** : LangChain + LlamaIndex + GPT-4 (ou modèle open source) + GPU.
- **Description** : Reçoit une question, interroge l'index vectoriel, reformule la réponse avec citations, mise en contexte, et source des données cliniques.

## 5. SyntheseComparative

- **Rôle** : Génère un résumé structuré ou une comparaison inter-patients.
- **Technologies** : Transformers + Template Prompting + Python.
- **Description** : Produit une synthèse : ex. « évolution du traitement anticoagulant chez patient X », ou « différence entre cas X et cas Y sur 6 mois ».

## 6. AuditLogger

- **Rôle** : Trace toutes les interactions utilisateur/LLM.
- **Technologies** : FastAPI + PostgreSQL.

- **Description** : Enregistre la question, les documents consultés, le temps de réponse, l'identifiant de l'opérateur, pour audit et supervision médicale.

## 7. InterfaceClinique

- **Rôle** : Application web pour interrogation et affichage des réponses.
- **Technologies** : React + Tailwind + Auth0 + Chart.js.
- **Description** : Permet aux médecins, chercheurs, ou responsables qualité d'interroger les documents, filtrer par patient/date/type, et afficher réponses + explications.

### Résultats attendus

- Gain de temps dans la recherche d'informations médicales
- Utilisation du langage naturel pour des requêtes complexes
- Respect de la confidentialité grâce à la désidentification et l'audit
- Solution modulaire, interopérable, adaptée à une publication SoftwareX

## 9) AquaWatch-MS — Qualité de l'eau en temps réel

### Contexte

Dans de nombreuses régions, la qualité de l'eau (potable, fluviale, côtière) est affectée par des sources de pollution industrielle, agricole ou domestique. Or, la détection de cette pollution est souvent lente, manuelle et coûteuse. Les collectivités territoriales et agences environnementales manquent d'outils numériques intégrés pour surveiller et anticiper les pics de pollution. Une solution automatisée, basée sur des données hétérogènes (stations au sol, satellites, normes OMS), permettrait une alerte rapide et une prise de décision éclairée.

### Objectif

Développer une plateforme modulaire fondée sur des microservices pour :

- surveiller en temps réel la qualité de l'eau via des capteurs IoT et données satellites,
- effectuer des prévisions à court terme basées sur des modèles spatio-temporels,
- alerter automatiquement les autorités et les citoyens en cas de dépassement de seuils,
- fournir des interfaces cartographiques interactives pour visualiser l'état des masses d'eau.

### Architecture détaillée des microservices

#### 1. Capteurs

- **Rôle** : Collecte des données brutes en temps réel depuis les stations de mesure (pH, turbidité, température, conductivité...).
- **Technologies** : MQTT + Node.js pour la passerelle, base de données TimescaleDB pour les séries temporelles.
- **Fonction** : Normalise et stocke les données en continu, avec horodatage et géolocalisation.

## 2. Satellite

- **Rôle** : Intègre et traite les données de télédétection issues de Sentinel-2 ou Copernicus.
- **Technologies** : GDAL, rasterio, SentinelHub API, stockage dans MinIO.
- **Fonction** : Extraction de variables dérivées (chlorophylle, turbidité, NDWI) sur zones définies, complétant les capteurs au sol.

## 3. STModel

- **Rôle** : Prédiction de la qualité de l'eau dans le temps et l'espace.
- **Technologies** : Modèle spatio-temporel ConvLSTM implémenté en PyTorch, entraîné sur séries multisources.
- **Fonction** : Anticipe les variations à 24h ou 72h (ex. augmentation de turbidité post-pluie).

## 4. Alertes

- **Rôle** : Génère des notifications dès qu'un seuil critique (selon les normes OMS) est franchi.
- **Technologies** : Node.js, PostgreSQL pour journalisation, notification via email/SMS/API webhook.
- **Fonction** : Système d'alerte paramétrable par type de polluant, localité ou population exposée.

## 5. API-SIG

- **Rôle** : Fournit les couches de données environnementales aux utilisateurs externes via une interface cartographique.
- **Technologies** : GeoServer, PostGIS, endpoints REST/GeoJSON.
- **Fonction** : Sert les cartes interactives, historiques de qualité, zones rouges/vertes pour les communes.

## Résultats attendus

- Surveillance continue, automatisée et territorialisée de la qualité de l'eau.
- Détection proactive des risques de pollution.

- Outil modulaire interopérable avec d'autres plateformes environnementales.
- Application concrète pour les collectivités, les agences de bassin et les citoyens.
- Solution librement déployable avec publication dans *SoftwareX*.

## 10) MobileSec-MS — Analyse de sécurité des applications mobiles

### Contexte

Les applications mobiles sont souvent publiées avec des vulnérabilités graves : clés API exposées, mauvaise gestion du stockage local, absence de chiffrement, ou encore manque de validation côté client. Les tests manuels de sécurité sont longs et coûteux, et les outils existants ne sont pas toujours adaptés à des pipelines DevSecOps continus. Une plateforme modulaire permettrait de détecter automatiquement ces failles lors de l'analyse statique ou dynamique.

### Objectif

Développer une plateforme modulaire composée de microservices spécialisés qui permettent :

- l'analyse automatisée de fichiers APK/iOS (APK/IPA),
- la détection de vulnérabilités OWASP MAS (Mobile Application Security),
- l'extraction de bonnes pratiques manquantes (sécurité SSL, permissions, export, stockage),
- la suggestion de correctifs,
- l'intégration continue dans les workflows DevSecOps.

### Architecture détaillée des microservices

#### 1. APKScanner

- **Rôle** : Désassemble et analyse les APK pour extraire le manifest, les permissions et les endpoints.
- **Technologies** : Python (Androguard, Apktool), base SQLite pour métadonnées.

- **Fonction** : Identifie composants exportés, permissions dangereuses, debuggable, allowBackup, cleartextTrafficPermitted, etc.

## 2. SecretHunter

- **Rôle** : Recherche les secrets exposés dans le code ou les ressources.
- **Technologies** : GitLeaks, Regex personnalisées, YARA Rules.
- **Fonction** : Déetecte clés API, tokens OAuth, mots de passe codés en dur, etc.

## 3. CryptoCheck

- **Rôle** : Vérifie l'utilisation correcte des API cryptographiques.
- **Technologies** : SAST Java/Kotlin, règles codifiées (CWE).
- **Fonction** : Déetecte mauvais usages de AES/ECB, absence de padding, faiblesse dans le random, MD5/SHA1.

## 4. NetworkInspector

- **Rôle** : Intercepte les communications réseau via un proxy mitm.
- **Technologies** : mitmproxy, Docker sandbox Android (AVD).
- **Fonction** : Analyse HTTPS/TLS, en-têtes HTTP, gestion de certificats, fuites de données.

## 5. ReportGen

- **Rôle** : Agrège les résultats et génère des rapports PDF, JSON et SARIF.
- **Technologies** : Node.js, Puppeteer.
- **Fonction** : Génère une fiche détaillée des failles détectées et des recommandations correctives.

## 6. FixSuggest

- **Rôle** : Fournit des suggestions de correctifs ou configurations à appliquer.
- **Technologies** : Modèle de règles MASVS → YAML → patchs.
- **Fonction** : Propose des actions concrètes : android:exported="false", activer Proguard, isoler signingConfig.

## 7. CIConnector

- **Rôle** : Intégration avec GitHub Actions / GitLab CI.
- **Technologies** : Docker CLI, plugins YAML.
- **Fonction** : Permet d'activer les scans automatiquement à chaque build.

### Résultats attendus

- Détection rapide et automatisée des failles de sécurité mobiles.
- Aide à la conformité avec les normes OWASP MASVS, CWE, et CIS Mobile.
- Intégration fluide dans le cycle DevSecOps des applications Android/iOS.
- Plateforme extensible en open source, avec jeux d'exemples et rapports reproductibles pour publication dans *SoftwareX*.

## Projet 11) Maintenance prédictive temps-réel pour usines intelligentes

### Contexte :

Les usines subissent des arrêts non planifiés coûteux (~50 Md USD/an dans le manufacturing ; coût médian > 125 000 USD/heure). Les données issues des PLC/SCADA, capteurs (vibration, température, courant, acoustique) et systèmes OT restent souvent silotées et peu exploitées. L'objectif industriel est de passer de la maintenance corrective/préventive à une maintenance prédictive basée sur l'analyse de flux IIoT en continu.

### Objectif :

Concevoir une plateforme temps-réel capable de détecter précocement les anomalies, estimer la Remaining Useful Life (RUL) des équipements, planifier des interventions optimales (main-d'œuvre, fenêtres d'arrêt, pièces), et s'intégrer nativement aux systèmes OT/IT (SCADA/MES/CMMS/ERP) avec tableaux de bord opérationnels.

### Jeux de données :

NASA C-MAPSS (CSV, 21 capteurs, 3 réglages moteur, 4 scénarios) comme dataset de référence pour l'entraînement/évaluation des modèles RUL et le transfert d'apprentissage vers les actifs de l'usine.

### Architecture (microservices)

#### 1. IngestionIIoT

- **Rôle** : Collecter les flux capteurs depuis PLC/SCADA/edge (vibro, température, courant, acoustique) et normaliser les métadonnées (actif, ligne, cadence).
- **Entrées/Sorties** : OPC UA/Modbus/MTConnect/MQTT → topics temps-réel (Kafka).
- **API/Technologies** : Connecteurs OPC UA (Eclipse Milo), Telegraf, REST/gRPC pour contrôle, Kafka pour diffusion.
- **Base de données** : TimescaleDB/InfluxDB (séries temporelles), MinIO (bruts/Parquet).
- **Description** : Horodatage unifié, conversion d'unités, gestion qualité (QoS), tampon edge en cas de perte réseau.

#### 2. Prétraitement

- **Rôle** : Nettoyer et aligner (rééchantillonnage, débruitage, imputation), fenêtrage glissant.
- **Entrées/Sorties** : Flux bruts → séries normalisées et fenêtres prêtes apprentissage.

- **Technologies** : Python (Pandas, SciPy), Kafka Streams/Faust, STFT/FFT, filtres passe-bande.
- **Base de données** : TimescaleDB (fenêtres), MinIO (traces).
- **Description** : Détection/traitement des valeurs aberrantes, synchronisation multi-capteurs.

### 3. ExtractionFeatures

- **Rôle** : Calculer des descripteurs temps/fréquence (RMS, kurtosis, crest factor, énergie de bande, centroides spectraux, order tracking, enveloppe, ondelettes).
- **Entrées/Sorties** : Fenêtres → vecteurs de caractéristiques (JSON/Parquet).
- **Technologies** : tsfresh/tsflex, PyWavelets, SciPy.
- **Base de données** : Feast (feature store) pour réutilisation online/offline.
- **Description** : Standardisation par famille d'actifs (pompes, moteurs, convoyeurs, CNC).

### 4. DétectionAnomalies

- **Rôle** : Déetecter des déviations par rapport au fonctionnement nominal (alertes précoces).
- **Entrées/Sorties** : Features → scores/événements d'anomalies.
- **Technologies** : PyOD (IsolationForest, One-Class SVM), autoencodeurs (PyTorch).
- **Base de données** : PostgreSQL/TimescaleDB (journal d'événements).
- **Description** : Seuils adaptatifs par criticité ligne/actif, agrégation multi-capteurs.

### 5. PrédictionRUL

- **Rôle** : Estimer la RUL (distribution + intervalle de confiance) et la probabilité de défaillance à horizon H.
- **Entrées/Sorties** : Séquences/Features → RUL + incertitudes.
- **Technologies** : PyTorch (LSTM/GRU/TCN), XGBoost, calibration, MLflow (experiments + model registry).
- **Base de données** : Feast (features), MLflow (modèles), MinIO (artefacts).
- **Description** : Pré-entraînement sur NASA C-MAPSS, affinement par transfert sur données usine.

### 6. OrchestrateurMaintenance

- **Rôle** : Appliquer politiques et contraintes (sécurité, SLA, fenêtres d'arrêt, inventaire) et générer des actions concrètes.

- **Entrées/Sorties** : Anomalies + RUL + contraintes → recommandations/ordres de travail (JSON).
- **Technologies** : Drools (règles), OR-Tools (optimisation planning), FastAPI (exposition).
- **Base de données** : PostgreSQL (référentiel actifs/BOM/criticité).
- **Description** : Règles exemple : « Si RUL < 120 h et stock pièce = 0, créer commande + inspection sous 24 h ».

## 7. DashboardUsine

- **Rôle** : Visualiser état des lignes, alertes, RUL, backlog interventions et ROI évité.
- **Entrées/Sorties** : Données consolidées → UI web temps-réel + exports PDF/CSV.
- **Technologies** : React.js, API REST/gRPC, websockets, Grafana/Plotly.
- **Base de données** : PostgreSQL (métadonnées), TimescaleDB (séries), option PostGIS pour cartographie d'atelier.
- **Description** : Per-asset drill-down (spectres, tendances), heatmaps de criticité, KPI (MTBF, MTTR, disponibilité OEE).

**Résultat attendu :**

- Un système modulaire déployé sous Docker/Kubernetes, intégrant des flux OT (OPC UA/SCADA) et IT (MES/CMMS/ERP), capable de détecter les anomalies, prédire la RUL et planifier des interventions optimisées avec traçabilité complète (datasets, features, modèles via MLflow/Feast). La solution est reproductible (pipelines entraînement/inférence versionnés), observée (OpenTelemetry) et documentée selon les attentes SoftwareX (code, jeux d'essai, scripts, notices d'installation).

**Compétences mobilisées** : ingestion IIoT, traitement de séries temporelles, modélisation séquentielle RUL, MLOps (MLflow/Feast), interprétabilité (SHAP sur features), intégration OT/IT et règles métier.

## Projet 12) Recommandation Automatisée des Classes Logicielles à Tester (ML pour améliorer la couverture unitaire)

### Contexte :

Dans les grands dépôts, la masse de classes et l'évolution rapide du code rendent la couverture unitaire incomplète et hétérogène. Les priorisations manuelles/heuristiques laissent des zones non testées et dégradent la qualité. Un système data-driven permet de cibler en priorité les classes à plus haut risque de défaut, en s'appuyant sur métriques de code, historique de commits, couverture et bugs.

### Objectif :

Concevoir une plateforme qui recommande automatiquement les classes à tester en priorité, augmente la couverture unitaire de façon ciblée, réduit l'effort de sélection manuelle et démontre des gains mesurables face aux méthodes traditionnelles (baseline heuristiques).

### Jeux de données :

PROMISE (source principale) pour l'entraînement/évaluation en prédition de défauts à partir de métriques logicielles. Élargissement optionnel : historiques internes (commits, issues, couverture JaCoCo), jeux publics proches (NASA MDP/AEEEM/SEACRAFT, Defects4J pour validation externe).

### Architecture (microservices)

#### 1. CollecteDepots

- Rôle : Ingestion des dépôts (Git/GitHub/GitLab), issues/bugs (Jira/GitHub Issues), artefacts CI (rapports tests/couverture).
- Entrées/Sorties : Webhooks/cron → topics d'ingestion (Kafka) ; snapshots métriques/commits → stockage brut.
- API/Technos : GitHub/GitLab/Jira API, JGit, REST/gRPC, Kafka.
- Base : PostgreSQL (métadonnées), MinIO (artefacts/rapports), TimescaleDB (séries de métriques par classe/commit).
- Description : Aline commits, issues et builds CI ; versionne les jeux internes (DVC).

#### 2. AnalyseStatique

- Rôle : Extraction de métriques de code : LOC, complexité cyclomatique (McCabe), CK (WMC, DIT, NOC, CBO, RFC, LCOM), dépendances (in/out degree), smells.

- Entrées/Sorties : Sources → vecteurs de métriques par classe/commit.
- Technos : JavaParser/CK/PMD/SonarQube API (Java), radon (Python), JGraphT/networkx (graphes de dépendances).
- Base : Feast (feature store) pour réutilisation online/offline.
- Description : Normalise par module/langage ; gère multi-projets.

### **3. HistoriqueTests**

- Rôle : Agréger couverture et résultats : line/branch coverage, tests KO/OK, flakiness, mutation score.
- Entrées/Sorties : Rapports CI (JaCoCo/XML, Surefire, PIT) → métriques tests/commit.
- Technos : Parsers JaCoCo/Surefire/PIT, REST.
- Base : TimescaleDB (évolution), PostgreSQL (indices par classe).
- Description : Relie classes <-> tests couvrants ; calcule dette de test.

### **4. PrétraitementFeatures**

- Rôle : Nettoyage, imputation, encodage ; construction de features dérivées (churn, nb auteurs, fréquence modifs, proximité avec bug-fix commits).
- Entrées/Sorties : Flux bruts → jeux train/val/test time-aware (évite fuite temporelle).
- Technos : Python (Pandas, scikit-learn), DVC pour data lineage.
- Base : Feast (features versionnées).
- Description : Balancement de classes (SMOTE/cost-sensitive), normalisation par repo.

### **5. MLService**

- Rôle : Entraîner/servir modèles de risque de défaut par classe et priorisation effort-aware.
- Entrées/Sorties : Features → score [0–1] + incertitude + top-K classes.
- Technos : XGBoost/LightGBM/LogReg/RandomForest ; unsupervisé en appui (IsolationForest/LOF) ; MLflow (experiments, model registry).
- Base : MLflow (artefacts), MinIO (modèles), Feast (online features).
- Description : Validation time-aware (train sur anciens commits, test sur récents) ; calibration des probabilités ; SHAP pour l'explicabilité locale/globale.

### **6. MoteurPriorisation**

- Rôle : Transformer scores en liste ordonnée en intégrant effort (LOC), criticité module, dépendances et objectifs de sprint.

- Entrées/Sorties : Scores + contraintes → plan de tests priorisé (JSON).
- Technos : OR-Tools (optimisation), heuristiques effort-aware (Popt@20).
- Base : PostgreSQL (politiques/poids).
- Description : Stratégies : top-K couvertures manquantes, maximisation Popt@20, budget de tests.

## 7. TestScaffolder (optionnel)

- Rôle : Générer des squelettes JUnit pour classes prioritaires, suggestions de cas (équivalence, limites, mocks).
- Entrées/Sorties : Liste priorisée → templates de tests + checklist mutation.
- Technos : Analyse AST (Spoon/JavaParser), templates Mustache.
- Base : Repo "tests-suggestions".
- Description : Accélère l'écriture sans imposer de génération automatique complète.

## 8. DashboardQualité

- Rôle : Visualiser recommandations, couverture, risques, tendances et impact (défauts évités).
- Entrées/Sorties : Données consolidées → UI temps réel + exports PDF/CSV.
- Technos : React.js, FastAPI, websockets, Grafana/Plotly.
- Base : PostgreSQL/TimescaleDB.
- Description : Vue par repo/module/classe, drill-down SHAP, comparaison avant/après.

## 9. Intégrations & Ops

- Rôle : Intégration CI/CD (GitHub Checks/GitLab MR), commentaires automatiques sur PR, triggers d'entraînement, auth/SSO.
- Technos : GitHub Actions/GitLab CI, Docker/Kubernetes, OpenTelemetry (observabilité), Keycloak (IAM).
- Description : Politique "gate" optionnelle (alerte si classe risquée modifiée sans tests).

### Résultats attendus :

- Modèle performant fournissant un classement des classes à tester, explicable (SHAP).
- Pipeline complet microservices (collecte → prétraitement → apprentissage → priorisation → visualisation) déployé sous Docker/K8s.

- Tableau de bord interactif avec métriques : F1/PR-AUC/ROC-AUC, effort-aware (Popt@20), Recall@Top20% lignes/classes, gain de couverture, défauts "échappés" réduits, temps de sélection économisé.
- Évaluation comparative vs heuristiques (complexité seule, couverture seule, récenteté).
- Documentation SoftwareX-ready : code, scripts d'installation, jeux d'essai (PROMISE + échantillons internes anonymisés), fiches d'expériences MLflow, schémas de données, guide d'extension.

**Compétences mobilisées :** analyse statique, métriques logicielles, MLOps (Feast/MLflow/DVC), classification/novelty detection, optimisation effort-aware, intégration CI/CD, UX de qualité logicielle.