# 

# L’IA au service de la qualité des données (POC)

## Contexte & Objectifs

* Les équipes métiers souhaitent disposer d’un moyen simple et rapide afin de détecter les anomalies liées à la qualité de leurs données sans que les règles métiers ne soient codées manuellement.
* L’objectif de ce projet consiste donc à implémenter une solution modulaire pour détecter rapidement les problèmes de qualité de données. Le moteur de cette solution serait un système multi-agents IA (AI Foundry d’Azure).

**Source de données :** Lakehouse (Microsoft Fabric )

**Interface chatbot** : accessible aux métiers, elle permettra entre autres à ces derniers de spécifier les informations relatives à l’asset dont ils souhaiteraient évaluer la qualité des données (détecter les anomalies). Les métiers auront aussi, via le chatbot, la possibilité d’effectuer des feedbacks afin de modifier, valider ou enrichir les règles générées.

**Catalogue de données** (Data Hub) : son rôle sera de stocker les métadonnées (schéma, statistiques, etc..) de la source de données contenant l’asset à évaluer. Il servira également de base de connaissances pour le système multi-agents IA dans le cadre de la génération des règles métiers.

**Système multi – agents IA**(AI Foundry d’Azure) : pour générer les règles métiers relatives à la qualité des données.

**Great Expectations** (Framework Python) : pour appliquer les règles métiers générées sur l’entièreté de l’asset à évaluer afin de remonter les anomalies.

**Dashboard (**Power BI) : pour la restitution des résultats.

## Epics & User Stories

### Epic 1 : Initialisation via le chatbot

#### US1 : Sélection de l’asset

* En tant que métier, je veux pouvoir indiquer au chatbot le nom ou le chemin de l’asset (table, fichier, dataset) que je souhaite évaluer afin de lancer le scan sur la bonne source de données.
* Critères d’acceptation :
  + Le chatbot affiche : “Quel asset voulez-vous scanner ?”
  + L’utilisateur peut saisir librement un chemin (ex. lakehouse.schema.table ou chemin/fichier.csv).
  + Si le champ est vide, le chatbot renvoie un message d’erreur invitant à ressaisir.

#### US2: Saisie des credentials

* En tant que métier, je veux pouvoir fournir au chatbot mes informations de connexion afin de permettre l’accès à la source de données où réside l’asset.
* Critères d’acceptation :
  + Le chatbot demande séquentiellement chaque paramètre (URL, port, utilisateur, mot de passe/token).
  + Les champs sensibles (mot de passe, token) sont masqués à l’écran.
  + Le chatbot vérifie un format basique (non vide, JSON valide si applicable) avant de poursuivre.

#### US3 : Validation des informations saisies

* En tant que métier, je veux pouvoir revoir le résumé : « « asset+credentials » que j’ai fourni (sans afficher les mots de passes ou token en clair) afin de confirmer que j’ai saisi la bonne cible et les bonnes informations d’accès.
* Critères d’acceptation :
  + Le chatbot affiche les éléments renseignés par le métier : asset + credentials
  + L’utilisateur répond “Confirmer” ou “Modifier”.
  + En cas de “Modifier”, le chatbot reprend la saisie correspondante.

#### US4 : Lancement du scan et accusé de réception

* En tant que métier, je veux pouvoir déclencher la détection des anomalies dans mon asset et recevoir immédiatement un identifiant unique de job afin de pouvoir suivre l’avancement de mon scan.
* Critères d’acceptation :
  + Après confirmation par le métier (User Story précédent (US3)), le chatbot répond : Votre scan a démarré. Voici votre identifiant de job : **JOB-xxxxxxxxxx**.
  + L’évènement (asset+credentials+jobId) est publié vers l’orchestrateur.
  + En cas de “Modifier”, le chatbot reprend la saisie correspondante.

### Epic 2 : Orchestration & collecte des métadonnées

#### US5 : Réception du job par l’orchestrateur Airflow

* En tant que composant orchestrateur Airflow, je veux récupérer l’évènement « scan démarré » (asset+credentials +jobId) envoyé par le chatbot afin de lancer le workflow correspondant
* Critères d’acceptation :
  + L’évènement est consommé et un run de pipeline est créé avec le jobId
  + En cas d’échec de consommation, une erreur traçable est remontée au chatbot

#### US6 : Connexion à la source de données

* En tant que composant orchestrateur Airflow, je veux pouvoir utiliser les credentials reçus pour authentifier une connexion API afin de pouvoir récupérer le schéma de la source de données, profiler les données, faire des statistiques sur tous les assets présents dans la source données et récupérer l’asset (les données) à évaluer (on pourra stocker ce dernier dans un espace de stagging par exemple).
* Critères d’acceptation :
  + La connexion est tentée avec un timeout configurable (ex. 30 s).
  + En cas de succès, le pipeline poursuit.
  + En cas d’échec (authentification, timeout), le jobId passe en état **ERREUR\_CONNEXION** et le chatbot est notifié.

#### US7 : Scan de la source de données (schéma et profilage « customizé » des données)

* En tant que composant orchestrateur Airflow, je veux extraire le schéma de la source données, profiler ses objets afin d’alimenter le catalogue de données (Data Hub) qui servira de base de connaissance au système multi-agents.
* Critères d’acceptation :
  + Schéma & Profilage récupérés et formatés.
  + Tout échec partiel (objet inaccessible) est logué sans bloquer le scan global.

#### US8 : Ingestion des métadonnées (schéma & profilage) dans le catalogue de données

* En tant que composant orchestrateur Airflow, je veux pouvoir ingérer via API dans le catalogue de données (DataHub) les métadonnées récupérées.
* Critères d’acceptation :
  + Les métadonnées sont envoyées au Data Hub via API, liées à l’asset et au jobId.
  + En cas d’échec d’écriture, le jobId passe en état **ERREUR\_PUBLICATION** et le chatbot est notifié.

### Epic 3 : Génération & stockage des règles métiers

#### US9 : Récupération métadonnées depuis le Datahub

* En tant que système multi-agents IA, je veux pouvoir interroger le DataHub via protocole MCP pour récupérer les métadonnées correspondant au bon jobId afin de disposer de la connaissance nécessaire (sur la source des données) pour générer les règles de contrôle de la qualité des données.
* Critères d’acceptation :
  + Le système multi-agents IA authentifié récupère avec exactitude les métadonnées liées au jobId.
  + En cas d’erreur d’accès, le système ma\_IA logue l’erreur et envoie une alerte au chatbot.

#### US10 : Récupération de l’asset à évaluer depuis l’espace de stagging

* En tant que système multi-agents IA, je veux pouvoir récupérer, en fonction de la limite de ma fenêtre contextuelle, l’asset à évaluer (qui a été envoyé dans un espace de stagging par l’orchestrateur en US6) afin de disposer d’une source de connaissance supplémentaire (en plus des métadonnées du data hub) pour générer les règles de contrôle de la qualité des données.
* Critères d’acceptation :
  + Le système multi-agents IA (ma\_IA) authentifié récupère avec exactitude dans l’espace de stagging l’asset à évaluer et liée au jobId.
  + En cas d’erreur d’accès, le système ma\_IA logue l’erreur et envoie une alerte au chatbot.

#### US11 : Génération automatique des règles

* En tant que système ma\_IA, je veux pouvoir formuler à partir des métadonnées du Data hub et de l’asset à évaluer un jeu de règles métiers (complétude, validité, dédoublonnage, cohérence, exactitude, actualité) afin de les envoyer au bon format (expectations) au framework great expectations pour leur exécution sur l’entièreté de l’asset.
* Critères d’acceptation :
  + Le résultat est un JSON structuré listant toutes les expectations liées au jobId.
  + Chaque expectation contient son type et ses paramètres (seuils, plages, regex, etc…).

#### US12: Stockage des règles

* En tant que système ma\_IA, je veux persister les expectations générées dans une base dédiée afin de garantir la traçabilité, le versionning et la disponibilité pour le runner Great Expectations
* Critères d’acceptation :
  + Chaque règle est enregistrée avec : jobId, asset, type de règle, paramètres, horodatage, type d’agent, etc ….
  + En cas d’erreur d’écriture, le jobId passe en état **ERREUR\_STOCKAGE** et le chatbot est notifié.

#### US13 : Publication des règles pour feedback utilisateur

* En tant que système ma\_IA, je veux envoyer la liste des règles générées (en langage naturelle et aussi sous forme d’expectations) au chatbot afin de permettre à l’utilisateur de valider, supprimer ou enrichir ou invalider chaque règle
* Critères d’acceptation :
  + Le chatbot reçoit un payload formaté listant chaque expectation avec libellé et paramètres.
  + L’utilisateur peut, pour chaque règle, **Valider**, **Modifier** (nouveau paramètre) ou **Supprimer**.
  + Les modifications se reflètent immédiatement dans la base de règles.

### Epic 4 : Exécution des contrôles & restitution des anomalies

#### US14 : Lancement des expectations par Great Expectations (GE)

* En tant que Runner GE, je veux pouvoir récupérer pour le jobId donné, le jeu d’expectations validées afin de les exécuter sur l’asset à évaluer et identifier les non-conformités.
* Critères d’acceptation :
  + Le runner récupère toutes les expectations liées au jobId.
  + Les tests s’exécutent en parallèle ou séquentiellement (à configurer).
  + En cas d’erreur sur une expectation, elle est marquée **ERREUR\_EXECUTION** et les autres tests continuent.

#### US15 : Transmission des anomalies au système ma\_IA

* En tant que Runner GE, je veux pouvoir ne renvoyer que les enregistrements ayant échoué aux tests afin que le système ma\_IA analyse et explique précisément ces anomalies
* Critères d’acceptation :
  + Seuls les enregistrements non conformes sont extraits, avec règle associée, etc …
  + Le payload inclut jobId, asset et la liste des enregistrements fautives.
  + Si aucune anomalie, le runner renvoie “Aucune anomalie”.

#### US16 : Analyse et explication des anomalies per le système ma\_IA

* En tant que Runner GE, je veux pouvoir recevoir les anomalies et générer pour chacune un libellé d’explication (cause probable) et une suggestion de correction
* Critères d’acceptation :
  + Chaque anomalie reçoit un champ explication et un champ suggestion.
  + Les explications synthétisent les écarts (ex. “Valeur hors intervalle historique”).
  + L’IA agrège les anomalies similaires en groupes si nécessaire.

#### US17 : Stockage des anomalies et de leurs explications

* En tant que système ma\_IA, je veux pouvoir persister chaque anomalie avec expectation et suggestion dans un store dédié afin de garder un historique et pouvoir y revenir s’il le faut.
* Critères d’acceptation :
  + Chaque anomalie est enregistrée avec : jobId, asset, colonne, enregistrement ou valeur fautive, explication, suggestion, horodatage.
  + Le store permet des requêtes par jobId et par type d’anomalie.
  + En cas d’erreur d’écriture, l’anomalie est loguée et le manager central est notifié.

#### US18 : Publication sur le dashboard Power BI

* En tant que composant reporting, je veux pouvoir exposer les anomalies et leurs explications dans un dataset Power BI afin de permettre la visualisation interactive (filtrage, tri …)
* Critères d’acceptation :
  + Les anomalies sont importées dans Power BI avec tous leurs champs.
  + Des vues par colonne, type de règle et gravité sont préconstruites.
  + Le dataset se rafraîchit automatiquement à chaque réception de nouvelles anomalies.

#### US19 : Synthèse et notification via le chatbot

* En tant que chatbot, je veux pouvoir informer l’utilisateur du nombre et des types d’anomalies détectées afin de fournir un retour rapide et un lien vers le dashboard.
* Critères d’acceptation :
  + Le chatbot envoie par exemple :

Votre scan a détecté X anomalies réparties sur Y règles.

* + Il inclut un lien cliquable vers le dashboard Power BI du jobId.
  + L’utilisateur peut demanderpar exemple “Montre-moi les anomalies du job JOB-…” pour un résumé textuel.

### Epic 5 : Industrialisation, maintenance et gouvernance (CI/CD, IaC, tests, monitoring…)

#### US20 – Mise en place de la pipeline CI/CD

* En tant que DevOps, je veux qu’un merge dans la branche principale déclenche : la compilation du code (système ma\_IA, orchestrateur, runner GE, API chatbot), l’exécution des tests unitaires, la génération d’images Docker afin de garantir validation et packaging automatiques.
* Critères d’acceptation:
  + Un push/merge lance la pipeline CI (GitLab CI/GitHub Actions).
  + Build et tests passent ou échouent avec notification.
  + Les images Docker sont publiées dans le registry interne.

#### US21 – Déploiement de l’infrastructure avec IaC

* En tant qu’Ingénieur Cloud je veux pouvoir définir l’infrastructure en Terraform afin de provisionner/mettre à jour dev, prod de façon reproductible
* Critères d’acceptation:
  + Le code IaC décrit tous les services et dépendances.
  + Un terraform apply crée/actualise l’environnement.
  + Les paramètres (tailles, variables) sont gérés par environnement.

#### US25 – Gestion centralisée des secrets et credentials

* En tant que chargé de la sécurité je veux pouvoir stocker/distribuer les secrets (DB credentials, tokens chatbot) afin d’éviter tout stockage en clair et contrôler l’accès.
* Critères d’acceptation:
  + Aucun secret n’apparaît en clair dans le code ou les vars d’env.
  + L’accès aux secrets et credentials est géré par RBAC.

## Stack technique proposée

* **Chatbot API** : Python (FastAPI)
* **Data source**: lakehouse (Ms Fabric)
* **Orchestrateur** : Apache Airflow
* **Data Catalog** : Datahub
* **Système IA** : multi-agents Ai Foundry (Azure)
* **Exécution des règles** : Great Expectations
* **Reporting**: Power BI (dataset Power BI REST API)
* **Infra & CI/CD**: Docker

## Roadmap & Prochaines Étapes

1. **Phase de prototypage** (US1–US4) : implémenter un minimal viable chatbot + orchestration
2. **Phase IA & règles** (US5–US13) : monter le moteur multi-agents et l’intégration Great Expectations
3. **Phase restitution** (US14–US19) : dashboard Power BI + synthèse chatbot
4. **Industrialisation** (CI/CD, monitoring, sécurité)