Dossier de Conception d'Architecture (EC01) - EnerVision

Table des Matières

1. Analyse des besoins et structure du projet
    1.1 Contexte métier et réglementaire
    1.2 Exigences fonctionnelles (EF)
    1.3 Exigences non-fonctionnelles (ENF)
2. Étude de faisabilité et infrastructure
    2.1 Comparatif de l'hébergement
    2.2 Gestion l’infrastructure et FinOps
3. Architecture Applicative (UML / C4)
    3.1 Couches logiques
    3.2 Justification du choix Backend : Directus vs PostgREST
4. Traitement et flux de données (ETL)
    4.1 Récupération des données
    4.2 Gestion des pannes et valeurs manquantes
5. Modèle de prédiction et gestion de l'IA (MLflow)
    5.1 Modèle de Prédiction
    5.2 Intégration MLflow
6. Déploiement automatisé (CI/CD) et sécurité
    6.1 Structure de la pipeline CI/CD
    6.2 Sécurité du réseau et du système
7. Tests et qualité du code
    7.1 Types de tests et outils
8. Observabilité, alertes et documentation continue
    8.1 Surveillance et alertes
    8.2 Documentation centralisée (MkDocs)
9. Synthèse finale de l'architecture

1. Analyse des besoins et structure du projet

1.1 Contexte métier et réglementaire

Les entreprises industrielles européennes évoluent dans un contexte énergétique instable, principalement à cause de la fluctuation des coûts et du renforcement des contraintes légales, telles que la directive CSRD et la taxe verte. Smart Energy Optimizer développée par EnerVision a pour objectif d'aider à résoudre ces contraintes. Le but est de convaincre un industriel ainsi qu'un fonds d'investissement green-tech en démontrant une capacité opérationnelle en deux semaines.

1.2 Exigences fonctionnelles (EF)

- **EF1 - Collecte des données :** Récupération des données des capteurs (consommation, tension, température, etc.) par une API Mock IoT(`/api/v1/sites/{site_id}/current` et `/readings`).
- **EF2 - Prédiction de la consommation :** Anticipation des pics de consommation à l'aide d'un modèle de Machine Learning entraîné sur des données passées.
- **EF3 - Recommandations :** Suggestion d'actions en cas d'anomalie ou de pic de consommation détecté.
- **EF4 - Tableau de bord :** Interface utilisateur affichant les données temps réel (`/api/v1/stats/summary`), l'état des capteurs et les alertes (`/api/v1/alerts`).
- **EF5 - API Sécurisée :** Mise à disposition des données et des prédictions via le backend.

1.3 Exigences non-fonctionnelles (ENF)

- **ENF1 - Délais :** Contrainte de 2 semaines de développement imposant d'automatiser le déploiement et la configuration de l'infrastructure.
- **ENF2 - Sécurité :** Pipeline DevSecOps avec des analyses du code et des conteneurs (Trivy, OWASP ZAP), pare-feu, blocage des attaques (CrowdSec) et contrôle du serveur (Lynis) .
- **ENF3 - Gestion des défaillances :** Gestion des pertes réseau et pannes de capteurs par nettoyage et remplacement des valeurs manquantes.
- **ENF4 - Observabilité :** Surveillance des ressources via Prometheus, gestion des alertes et suivi des configurations via AWX.
- **ENF5 - Interdiction :** Interdiction des solutions SaaS clés en main ; chaque brique logicielle reste hébergée et gérée en interne.

2. Étude de faisabilité et infrastructure

2.1 Comparatif de l'hébergement

Deux options étaient envisageables : un déploiement cloud sur AWS ou l'utilisation d'un serveur local (*on-premise*).

Critère d'Analyse	**AWS Cloud**	**Serveur local (*on-premise*)**
Souveraineté (RGPD)	Données soumises au Cloud américain, nécessitant un chiffrement préalable.	Conservation locale, garantissant leur souveraineté.
Gestion des Coûts (FinOps)	Coût variable, risque de dépassement de budget.	Coût fixe et prévisible, évitant toute surprise.
Automatisation	Configuration via les services managés AWS (RDS, ECS).	Déploiement direct via Docker Compose et automatisation avec Ansible
Pertinence pour l'évaluation	Capacité d'intégration de services tiers.	Maîtrise complète du système : réseau, système d'exploitation et sécurité

Décision : Déploiement sur serveur local. Cette option répond aux exigences des sites industriels sur la sécurité de leurs données et permet de maîtriser l'ensemble de la chaîne technique sur une machine dédiée.

2.2 Gestion de l’infrastructure et FinOps

L'infrastructure de conteneurs est définie avec Terraform via le connecteur Docker pour automatiser le déploiement sur la machine physique. Atlantis sera déployé afin d’optimiser le travail des développeurs et de sécuriser la production.

- **Atlantis :** Automatise l'exécution de Terraform sur le dépôt Git. Chaque Pull Request génère un aperçu des modifications (terraform plan) directement en commentaire.
- **Prometheus & AlertManager :** Surveillent l'utilisation du processeur, de la mémoire et du stockage pour prévenir la saturation du serveur physique.
- **Ansible (piloté par AWX ou Ascender) :** Exécute les scripts de configuration du serveur hôte (installation des paquets, mises à jour et durcissement de sécurité)

Justification matérielle et FinOps :
Le serveur dédié (Dell PowerEdge R650xs) dispose de 128 Go de mémoire vive, d'un processeur 12 cœurs et d'une carte graphique Nvidia Tesla T4 (16 Go). L'entraînement et le calcul des prédictions IA avec MLflow nécessitent cette puissance de calcul. Utiliser ce matériel existant évite de louer des machines équivalentes sur le cloud (type instances AWS EC2 G4dn facturées à l'heure), ce qui supprime les coûts d'infrastructure du projet.

3. Architecture Applicative (UML / C4)

L'architecture repose sur des conteneurs isolés communiquant via un réseau Docker interne privé, exposant uniquement les ports nécessaires au reverse proxy (Nginx).

3.1 Couches logiques

- **Conteneur 1 (Front-End) :** Tableau de bord pour visualiser les mesures en direct et les alertes développé en React ou Vue.
- **Conteneur 2 (Backend & API) :** Directus, déployé en local sur Docker. Il se connecte sur PostgreSQL pour générer automatiquement les API et fournir l'interface d'administration.
- **Conteneur 3 (Base de données) :** Instance PostgreSQL pour le stockage transactionnel et Data Warehouse.
- **Conteneur 4 (ETL Worker) :** Script Python avec Cron ou Celery interrogeant l'API Mock externe.
- **Conteneur 5 (Modèle IA) :** Serveur MLflow servant le modèle prédictif via une API REST.

3.2 Justification du choix Backend : Directus vs PostgREST

Fonctionnalité	PostgREST	Directus
Génération d'API	Directement sur la structure PostgreSQL en REST.	REST et GraphQL basés sur le schéma PostgreSQL.
Authentification	Nécessite des fonctions SQL et des tokens manuels.	Prête à l'emploi avec gestion native des rôles et des accès
Interface d'Administration	Aucune interface graphique	Interface web intégrée pour visualiser et modifier les données.

Même si les solutions SaaS externes sont interdites dans le projet, Directus reste un outil open-source auto-hébergé sur notre propre serveur. Il lit directement la base PostgreSQL. Ce choix permet de tenir le délai de deux semaines en évitant d'écrire un backend complet à la main, tout en gardant la maîtrise de nos données en local.

4. Traitement et flux de données (ETL)

Le traitement des données est assuré par un script Python (ETL) qui récupère, nettoie et charge les mesures

4.1 Récupération des données

Le script interroge l'API Mock selon deux moments différents :

- **Par lot (nuit) :** requête sur `/api/v1/readings` pour récupérer l'historique complet, qui sera utilisé pour l'entraînement du modèle IA.
- **En direct :** Requête toutes les 30 ou 60 secondes sur `/api/v1/sites/{site_id}/current` pour mettre à jour le tableau de bord.

4.2 Gestion des pannes et valeurs manquantes

L'API simule des anomalies indiquées par un statut **data_quality** (good, partial, degraded, critical). La base PostgreSQL sépare les données en deux tables : une table brute (*staging*) pour garder l'historique intact, et une table propre (*production*) pour le tableau de bord et l'IA.

```
data = extract_from_api(site_id)

db.insert_raw(data)

if data["data_quality"] != "good":
    null_reasons = data.get("null_reasons", [])
    if "weather_sensor_failure" in null_reasons:
        data["temperature_celsius"] = interpolation_lineaire(last_3_readings)
        data["humidity_percent"] = interpolation_lineaire(last_3_readings)
    elif "solar_sensor_failure" in null_reasons:
        data["solar_irradiance_wm2"] = interpolation_lineaire(last_3_readings)
    elif "sensor_outage" in null_reasons:
        data["consumption_kw"] = forward_fill(last_known_value)
    elif "network_loss" in null_reasons:
        data["consumption_kw"] = forward_fill(last_known_value)

db.insert_clean(data)
```

Ce découpage garantit la traçabilité des pannes sans bloquer le calcul des prédictions ni l'affichage du tableau de bord.

5. Modèle de prédiction et gestion de l'IA (MLflow)

5.1 Modèle de Prédiction

Le modèle choisi sera un algorithme de régression (ex: Random Forest ou Gradient Boosting via Scikit-Learn). Ce choix est adapté aux mesures relevées au fil du temps combinant plusieurs variables (consommation, tension, température, météo). Le modèle calcule la consommation attendue afin de détecter les écarts anormaux et d'anticiper les pics.

5.2 Intégration MLflow

MLflow gère l'ensemble du cycle de vie du modèle en local :

- **Suivi des entraînements :** Enregistre les paramètres, les métriques d'erreur (RMSE, MAE) et le fichier du modèle généré.
- **Gestion des versions :** Stocke les modèles entraînés et marque la version validée pour la production.
- **Déploiement :** MLflow expose une API REST. Le backend (Directus/ETL) appellera cet endpoint local (par exemple: http://mlflow:5000/invocations) en lui passant les mesures nettoyées et récupère les prédictions en retour.

6. Déploiement automatisé (CI/CD) et sécurité

6.1 Structure de la pipeline CI/CD

Le code est géré sur un dépôt GitLab. Chaque validation sur la branche principale déclenche automatiquement les étapes suivantes :

1. **Vérification du code :** Analyse de la syntaxe et des erreurs potentielles (ESLint pour le web, Flake8 pour Python).
2. **Tests unitaires :** Exécution des tests avec Pytest (Backend) et Jest (Frontend).
3. **Build sécurisé :** Construction des images Docker. Scan des images via Trivy pour bloquer le déploiement en cas de vulnérabilité critique .
4. **Déploiement de test (le staging) :** Démarrage d'un environnement temporaire via Terraform et son connecteur Docker.
5. **Tests fonctionnels et simulation d'attaques :** Exécution de tests Playwright (E2E) sur le staging, suivi d'une analyse OWASP ZAP simulant des attaques.
6. **Déploiement production :** Déploiement final sur le serveur local via Docker Compose et Ansible.

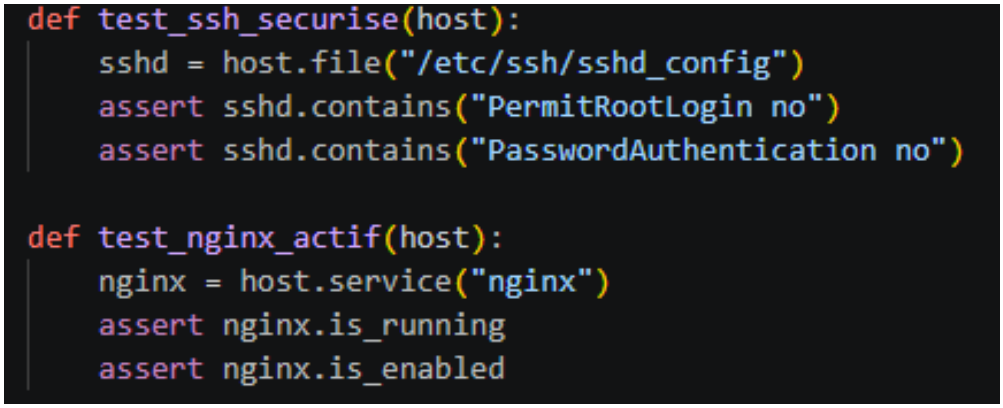
6.2 Sécurité du réseau et du système

- **Pare-feu (UFW) :** Seuls les ports web (80 et 443 pour Nginx) et l'accès d'administration SSH (port 22 filtré) sont ouverts vers l'extérieur. Tout le reste est bloqué.
- **Blocage des attaques (CrowdSec) :** Analyse les journaux de connexion pour bannir automatiquement les adresses IP suspectes (tentatives d'intrusion par force brute ou scans de ports)
- **Contrôle régulier (Lynis) :** Audit hebdomadaire du serveur pour vérifier les droits des fichiers, les mises à jour système et l'application des bonnes pratiques de sécurité.

7. Tests et qualité du code

7.1 Types de tests et outils

- **Tests fonctionnels (Front) :** Playwright ou Cypress. Un script simulera un utilisateur se connectant, regardant une alerte, et demandant une prédiction.
- **Tests d'intégration (API) :** Contrôle du comportement du backend en cas de panne de l'API simulée (dépassement de délai ou données dégradées) .
- **Tests d'infrastructure :** L'outil Testinfra vérifiera que Terraform a bien agi.



```python
def test_ssh_securise(host):
    sshd = host.file("/etc/ssh/sshd_config")
    assert sshd.contains("PermitRootLogin no")
    assert sshd.contains("PasswordAuthentication no")

def test_nginx_actif(host):
    nginx = host.service("nginx")
    assert nginx.is_running
    assert nginx.is_enabled
```

8. Observabilité, alertes et documentation continue

8.1 Surveillance et alertes

La surveillance du système utilisera Prometheus et Grafana.

- Prometheus : Collecte les mesures d'activité du serveur hôte via NodeExporter (processeur, mémoire vive, disque) et l'état des conteneurs via cAdvisor.
- Grafana affichera deux tableaux de bord : un écran technique (état de santé des services, logs Nginx) et un écran métier (latence des modèles ML).
- La gestion des alertes (CPU, RAM, état des conteneurs) sera traitée par Alertmanager. Les alertes métiers (pics énergétiques, anomalies) proviennent quant à elles directement de l'API (/api/v1/alerts) et sont transmises aux canaux de communication (Discord/Mail).

8.2 Documentation centralisée (MkDocs)

Pour éviter la dispersion des informations, l'architecture, les guides d'exploitation et la documentation de l'API interne sont rédigés en fichiers Markdown et mis en ligne avec MkDocs. Le site documentaire est régénéré automatiquement par la pipeline d'intégration continue à chaque mise à jour du dépôt.

9. Synthèse finale de l'architecture

En conclusion, l'architecture choisie pour le projet Smart Energy Optimizer remplit le cahier des charges et répond aux critères de l'évaluation EADL. En partant pour un serveur local on-premise, nous garantissons la sécurité des données et contrôlons précisément les dépenses. Le déploiement s'appuie sur des outils comme Terraform pour préparer le terrain, et Ansible pour tout configurer automatiquement. Le code suit un parcours précis avec des étapes de validation (GitLab CI, Atlantis, Trivy) avant d'être mis en production et testé par divers outils (Testinfra, Playwright, OWASP ZAP). De plus, les données énergétiques sont stockées en sécurité, puis analysées par des modèles de prédiction gérés par MLflow. Tout cela montre que la solution est prête à être déployée chez un client dans le milieu industriel.

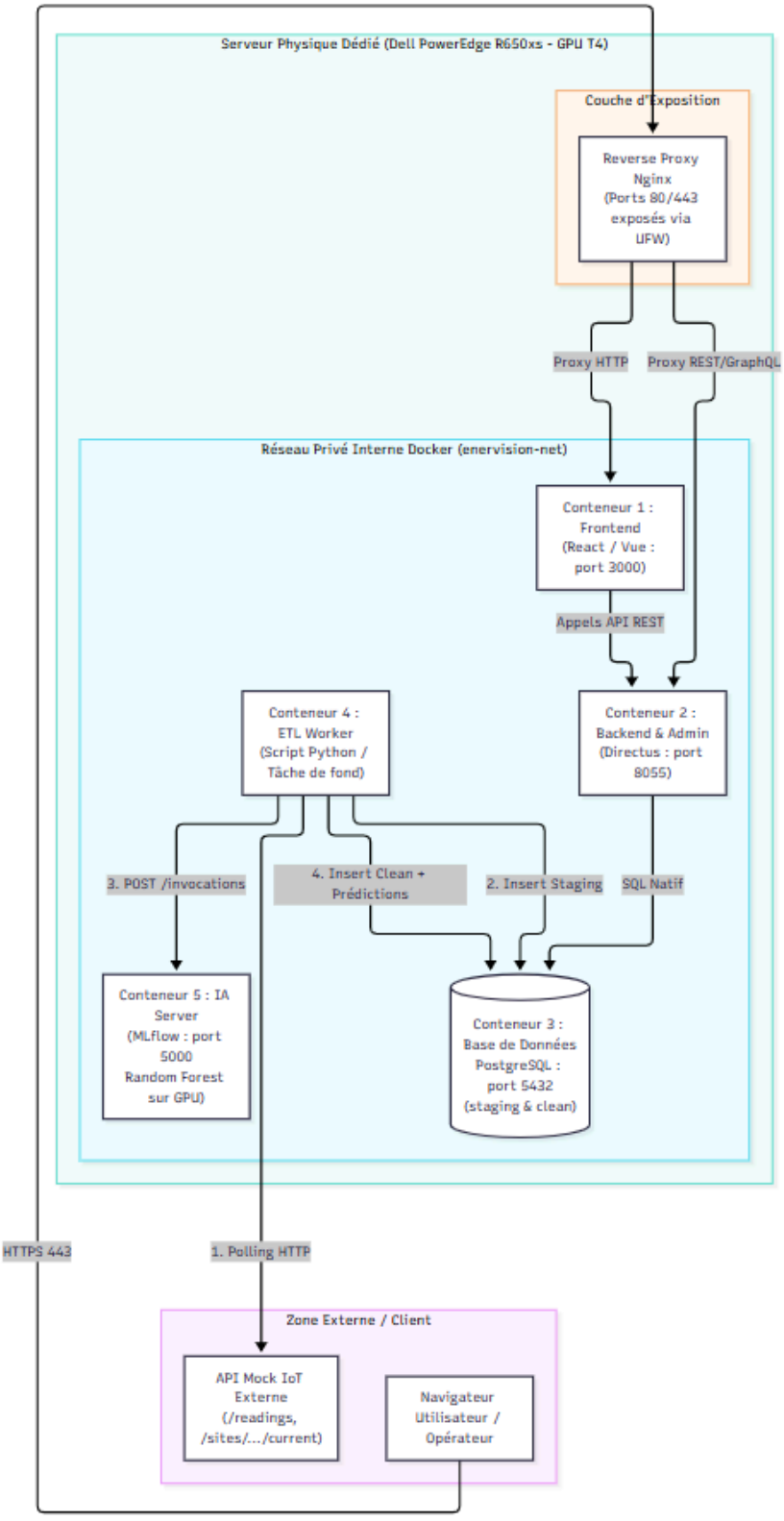


Schéma 1 : Diagramme de déploiement des conteneurs Docker

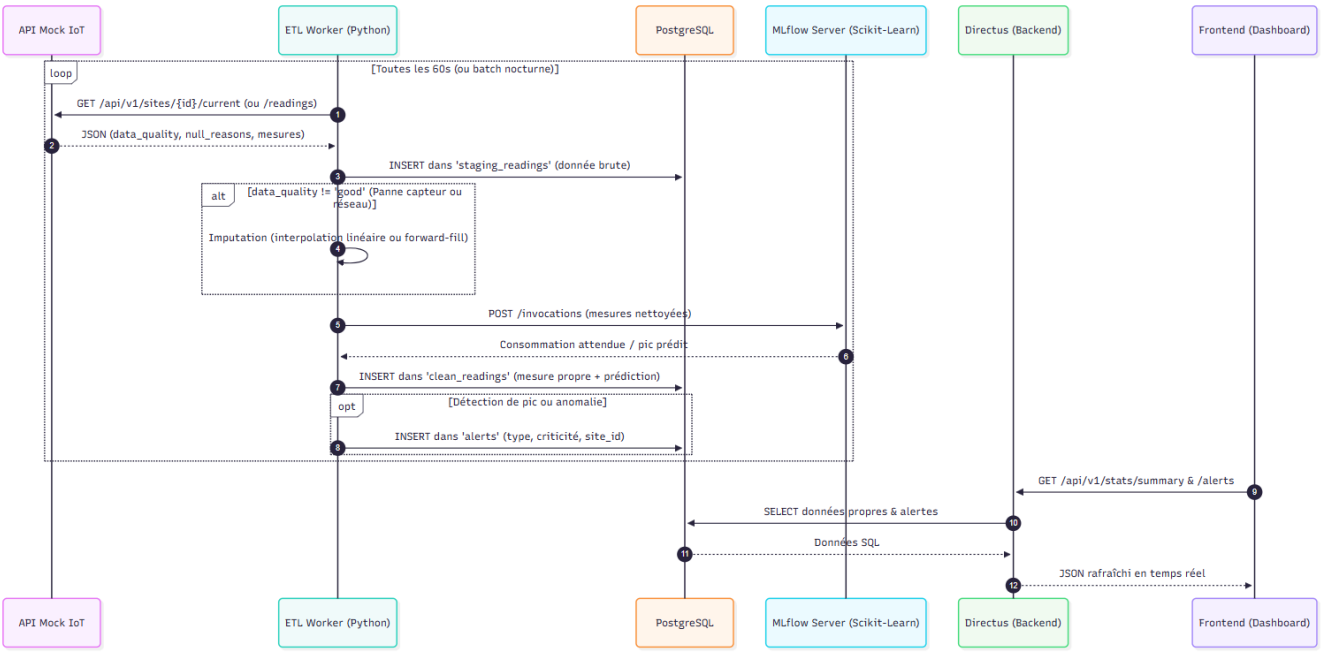


Schéma 2 : Diagramme de séquence — Pipeline de traitement de données ETL

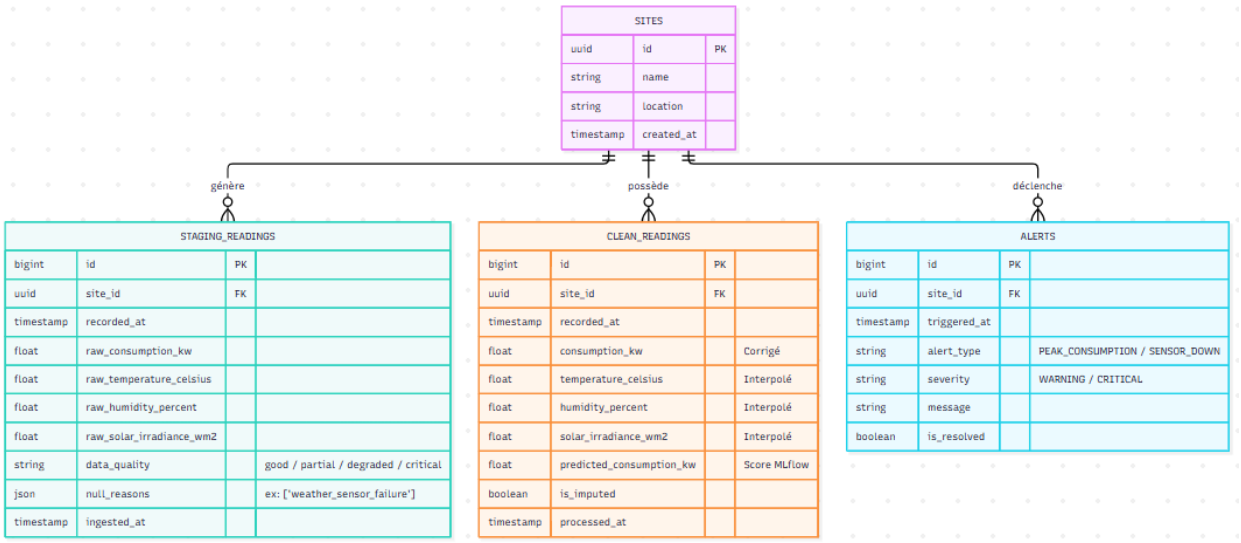


Schéma 3 : Modèle Physique de Données (MPD) PostgreSQL

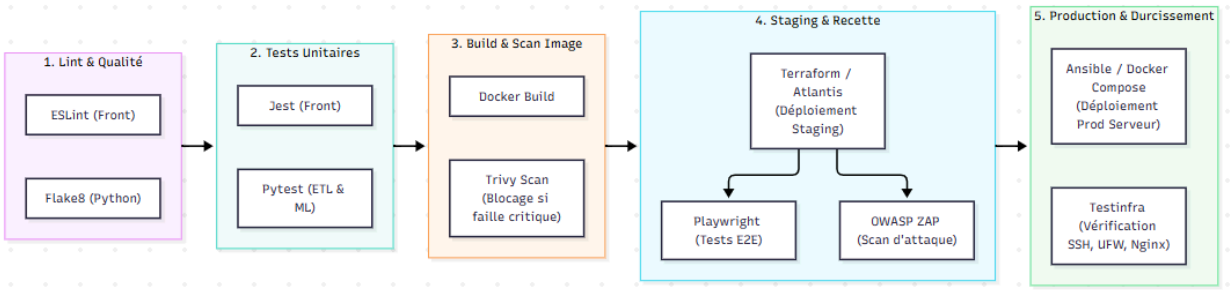


Schéma 4 : Diagramme de flux de la chaîne CI/CD (Pipeline DevSecOps)