epsi®
l'école d'ingénierie informatique

Rapport du projet MSPR TPRE501

Bloc 1 - Développement et déploiement d'une application dans le respect du cahier des charges Client, création d'un BackEnd métier permettant le nettoyage et la visualisation des données

Certification professionnelle :

Développeur en Intelligence Artificielle et Data Science RNCP 36581

Réalisé par :

DA SILVA Aimé José
GAUNET Aline
NDONGMO DJOUMESSI Wilfrid
WAMBA Forestin

2024 – 2025

Documentation Technique - Pipeline ETL COVID-19

Table des matières

1. Introduction

Ce document présente la documentation technique complète du pipeline ETL développé pour traiter les données mondiales de la COVID-19. Cette solution back-end métier permet l'extraction, le nettoyage, la transformation et le chargement des données dans une base de données relationnelle PostgreSQL pour des fins d'analyse et de visualisation.

Le projet s'inscrit dans le cadre du **MSPR N°1** "Création d'un backend métier permettant le nettoyage et la visualisation des données". Il répond au besoin de traiter de grands volumes de données épidémiologiques de manière efficace et fiable.

Les objectifs principaux de cette solution sont :

- Extraire les données brutes de sources CSV
- Nettoyer et normaliser les données selon des règles métier spécifiques
- Transformer les données pour les adapter au modèle de données cible
- Charger les données dans une base de données structurée
- Fournir une base solide pour la visualisation ultérieure des données

2. Architecture de la Solution

2.1 Vue d'ensemble

Notre pipeline ETL adopte une architecture modulaire en trois couches principales, conforme aux principes de séparation des responsabilités :

i. **Couche d'extraction** : Responsable de l'importation des données brutes depuis les fichiers CSV source
ii. **Couche de transformation** : Gère le nettoyage, la normalisation et l'agrégation des données
iii. **Couche de chargement** : Assure le stockage efficace des données dans la base PostgreSQL

Cette architecture permet une maintenance facilitée et une évolution indépendante de chaque composant.

2.2 Flux de données

Le flux de données dans notre pipeline suit un parcours structuré :

- Les fichiers CSV contenant les données COVID-19 sont placés dans le répertoire **data/raw/**
- Le module d'extraction lit et analyse ces fichiers
- Les données passent par plusieurs étapes de transformation :
    i. Nettoyage (suppression des valeurs nulles, correction des formats)
    ii. Normalisation (standardisation des noms de pays, conversion des types)
    iii. Agrégation (regroupement par pays, date et autres dimensions)
- Les données transformées sont chargées dans la base de données PostgreSQL
- Des logs détaillés sont générés à chaque étape pour assurer la traçabilité

2.3 Schéma d'architecture

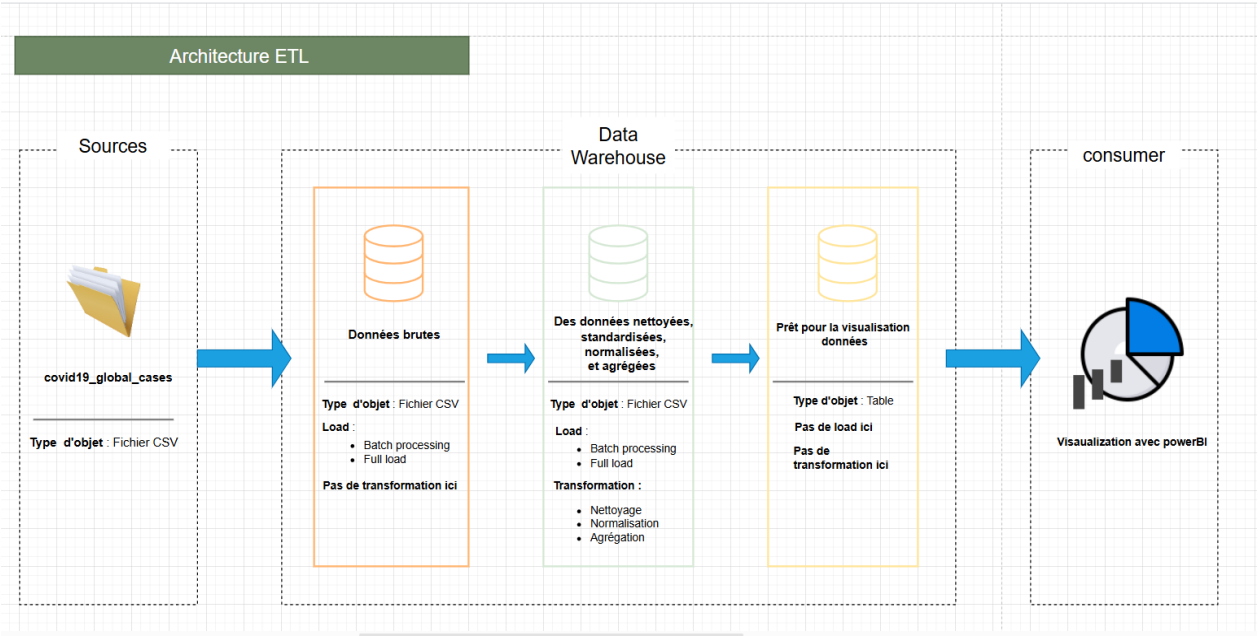

Figure 1: Diagramme d'architecture ETL

L'architecture ETL présentée dans le diagramme ci-dessus montre les interactions entre les différentes composantes du système et le flux de données, de l'extraction à la visualisation.

3. Technologies et Frameworks

3.1 Stack technique

Notre solution s'appuie sur les technologies suivantes :

- **Langage principal** : Python 3.12
- **Base de données** : PostgreSQL
- **Gestion de version** : Git/GitHub
- **CI/CD** : GitHub Actions
- **Visualisation** : Préparation des données pour Power BI

3.2 Dépendances principales

Le projet utilise plusieurs bibliothèques Python spécialisées :

Base de données

- sqlalchemy : ORM pour l'interaction avec la base de données
- psycopg2-binary : Connecteur PostgreSQL
- alembic : Gestion des migrations de schéma

Traitement des données

- pandas : Manipulation et analyse des données
- numpy : Opérations mathématiques et vectorielles

Configuration et environnement

- python-dotenv : Gestion des variables d'environnement
- pyyaml : Lecture des fichiers de configuration YAML

Logging et validation

- python-json-logger : Journalisation au format JSON
- pydantic : Validation des données et des modèles

Tests et qualité de code

- pytest : Framework de test unitaire
- pytest-cov : Mesure de la couverture de code
- black : Formatage automatique du code
- pylint : Analyse statique du code

3.3 Justification des choix technologiques

Notre stack technologique a été sélectionnée selon des critères précis :

Python a été choisi pour :

- Sa richesse en bibliothèques spécialisées dans le traitement des données
- Sa simplicité d'apprentissage et d'utilisation
- Son écosystème mature pour les projets data
- Son excellente intégration avec PostgreSQL

PostgreSQL s'est imposé pour :

- Ses performances avec les grands volumes de données
- Son support natif pour les types de données avancés
- Sa scalabilité horizontale et verticale
- Sa conformité ACID (Atomicité, Cohérence, Isolation, Durabilité) pour l'intégrité des données

SQLAlchemy permet :

- Une abstraction de la couche d'accès aux données
- Une flexibilité dans les requêtes
- Une sécurité accrue contre les injections SQL
- Une portabilité vers d'autres SGBD si nécessaire à l'avenir

4. Fonctionnalités Principales

4.1 Extraction des données

Le module d'extraction implémente les fonctionnalités suivantes :

- Lecture de fichiers CSV avec détection automatique des délimiteurs
- Validation des colonnes requises

- Profiling automatique des données (statistiques, distributions, anomalies)
- Journalisation détaillée du processus d'extraction
- Gestion des erreurs robuste

Exemple de code d'extraction :

classe CSVExtractor — Python

```python
class CSVExtractor:
    def __init__(self, filepath):
        self.filepath = filepath

    def extract(self):
        try:
            logger.info(f"Début extraction CSV: {self.filepath}")
            df = pd.read_csv(self.filepath)
            logger.info(f"Extraction réussie: {len(df)} lignes")

            # Profil des données
            profile = {
                "nombre_lignes": len(df),
                "nombre_colonnes": len(df.columns),
                "valeurs_manquantes": df.isnull().sum().to_dict(),
                "doublons": df.duplicated().sum(),
                "types_donnees": df.dtypes.to_dict(),
                "colonnes": list(df.columns),
                "memoire_utilisee": f"{df.memory_usage(deep=True).sum() / 1024**2:.2f} MB",
                "apercu_donnees": df.head().to_dict(),
            }

            logger.info(f"Profil des données:\n{profile}")
            return df, profile

        except Exception as e:
            logger.error(f"Erreur extraction CSV: {str(e)}")
            raise
```

Figure 2: Extrait du code d'extraction de donnée brut

4.2 Transformation des données

La couche de transformation est conçue selon le principe de responsabilité unique et comprend:

- Une classe abstraite BaseTransformer définissant l'interface commune
- Des implémentations spécifiques pour chaque type de transformation :
  i. DataCleaner : Suppression des valeurs aberrantes, correction des formats des dates.
  ii. DataNormalizer : Standardisation des valeurs (noms de pays, dates).
  iii. DataAggregator : Regroupement et calculs statistiques.

La chaîne de transformation suit un pattern de pipeline où chaque étape produit un DataFrame Pandas qui sert d'entrée à l'étape suivante.

Extrait du code de normalisation :

classe DataNormalizer — Python

```python
class DataNormalizer(BaseTransformer):
    def transform(self, df: pd.DataFrame) -> pd.DataFrame:
        try:
            logger.info("Début normalisation")

            # Renommage des colonnes
            column_mapping = {
                "Country/Region": "nom_pays",
                "Confirmed": "cas_confirmes",
                "Deaths": "deces",
                "Recovered": "guerisons",
                "Active": "cas_actifs",
                "New cases": "nouveaux_cas",
                "New deaths": "nouveaux_deces",
                "New recovered": "nouvelles_guerisons",
                "Date": "date_observation",
                "WHO Region": "region_oms",
            }
            df = df.rename(columns=column_mapping)

            # Standardisation des noms de pays
            df["nom_pays"] = df["nom_pays"].str.strip()

            # Mapping spécifique des noms de pays
            pays_mapping = {
                "Us": "US",
                "Cote D'Ivoire": "Cote d'Ivoire",
                "West Bank And Gaza": "West Bank and Gaza",
                # ... autres mappings
            }
            df["nom_pays"] = df["nom_pays"].replace(pays_mapping)

            # Conversion de la date
            df["date_observation"] = pd.to_datetime(df["date_observation"])

            # Nettoyage des valeurs numériques
            numeric_columns = [
                "cas_confirmes",
                "deces",
                "guerisons",
                "cas_actifs",
                "nouveaux_cas",
                "nouveaux_deces",
                "nouvelles_guerisons",
            ]

            for col in numeric_columns:
                if col in df.columns:
                    df[col] = (
                        pd.to_numeric(df[col], errors="coerce").fillna(0).astype(int)
                    )
                else:
                    logger.warning(
                        f"Colonne {col} manquante, création avec valeurs à 0"
                    )
                    df[col] = 0

            return df

        except Exception as e:
            logger.error(f"Erreur normalisation: {str(e)}")
            raise
```

Figure 3: Extrait du code de normalisation

4.3 Chargement des données

Le module de chargement assure l'insertion efficace des données dans PostgreSQL :

- Utilisation de SQLAlchemy pour les opérations ORM
- Support pour les insertions en masse (bulk insert).
- Gestion des transactions et rollbacks en cas d'erreur.
- Validation des données avant insertion.
- Optimisation des performances via chunking et paramètres de connexion.

Exemple de code de chargement :

classe PostgresLoader — Python

```python
class PostgresLoader:
    def __init__(self):
        self.db = db_manager

    def bulk_insert(self, table_name: str, df: pd.DataFrame, if_exists="append"):
        try:
            with self.db.get_session() as session:
                df.to_sql(
                    name=table_name,
                    con=session.connection(),
                    if_exists=if_exists,
                    index=False,
                    chunksize=1000,
                )
                logger.info(f"Chargement réussi dans {table_name}: {len(df)} lignes")
        except SQLAlchemyError as e:
            logger.error(f"Erreur chargement {table_name}: {str(e)}")
            raise
```

Figure 4: Extrait du code de chargement des données

4.4 Journalisation et suivi

Un système de logging complet est implémenté pour assurer la traçabilité des opérations :

- Logs détaillés pour chaque étape du processus ETL.
- Rotation automatique des fichiers de log.
- Format de log standardisé avec horodatage et niveau de gravité.
- Capture et documentation des erreurs.
- Métriques d'exécution (temps, mémoire utilisée, nombre d'enregistrements).

5. Structure du Projet

5.1 Organisation des répertoires

Le projet suit une organisation modulaire et claire :

```
covid_19_etl/
├── src/                    # Code source principal
|   ├── config/             # Configuration de l'application
|   ├── models/             # Modèles de données ORM
|   ├── extractors/         # Composants d'extraction
|   ├── transformers/       # Composants de transformation
|   ├── loaders/            # Composants de chargement
|   └── utils/              # Utilitaires et fonctions communes
├── data/                   # Données
|   ├── raw/                # Données brutes d'entrée
|   └── processed/          # Données transformées
├── logs/                   # Fichiers de logs
├── sql/                    # Scripts SQL
|   ├── create_tables.sql   # Création du schéma de la BDD
|   └── indexes.sql         # Création des index d'optimisation
├── tests/                  # Tests unitaires et d'intégration
├── .github/workflows/      # Configuration CI/CD
├── README.md               # Documentation principale
├── requirements.txt        # Dépendances Python
└── .env                    # Variables d'environnement (non versionné)
```

5.2 Composants clés

Les composants principaux du système sont :

Modèles de données (src/models/)

- Représentation ORM des tables de la base de données.
- Définition des relations et contraintes.
- Classes Pays, Maladie, et SituationPandemique.

Extracteurs (src/extractors/)

- Classes pour l'extraction de données depuis différentes sources.
- Actuellement implémenté : CSVExtractor pour les fichiers CSV.

Transformateurs (src/transformers/)

- Composants de transformation des données.
- Implémentation du pattern Template Method avec BaseTransformer.
- Classes dérivées : DataCleaner, DataNormalizer, DataAggregator.

Chargeurs (src/loaders/)

- Composants pour le chargement des données dans le stockage cible
- Implémentation principale : PostgresLoader

Utilitaires (src/utils/)

- Outils communs utilisés dans tout le projet
- Logger configuré avec rotation
- Validateurs de données et conversions

5.3 Patterns de conception utilisés

Plusieurs patterns de conception ont été implémentés pour assurer la qualité et la maintenabilité du code :

Context Manager

- Utilisé pour la gestion des sessions de base de données.
- Garantit la libération des ressources en cas d'erreur.

6. Guide d'Installation et Configuration

6.1 Prérequis système

Pour installer et exécuter le pipeline ETL, les prérequis suivants sont nécessaires :

- Python 3.12 ou supérieur
- PostgreSQL 14 ou supérieur
- Git
- Pip (gestionnaire de paquets Python)

6.2 Étapes d'installation

6.3 Configuration de l'environnement

Guide d'Installation et Configuration — Python

```
# 1. Cloner le dépôt
git clone https://github.com/Wambaforestin/covid_19_etl.git
cd covid_19_etl

# 2. Créer un environnement virtuel
python -m venv etlenv
source etlenv/bin/activate  # Linux/Mac
# ou
etlenv\Scripts\activate     # Windows

# 3. Installer les dépendances
pip install -r requirements.txt

# 4. Créer la base de données PostgreSQL
psql -U postgres -c "CREATE DATABASE covid_19_etl;"

# 5. Exécuter les scripts de création de tables
psql -U postgres -d covid_19_etl -f sql/script.sql
```

Figure 5: Installation de l'etl depuis le repos GitHub

Créez un fichier **.env** à la racine du projet avec le contenu suivant :

Configuration de l'environnement — Python

```
# Configuration de la base de données
DB_HOST=localhost
DB_PORT=5432
DB_NAME=covid_19_etl
DB_USER=postgres
DB_PASSWORD=votre_mot_de_passe
```

Figure 6: Configuration de l'environment

6.4 Démarrage de l'application

Pour exécuter le pipeline ETL complet :

Exécuter le pipeline ETL complet — Python

```
# Assurez-vous que votre environnement virtuel est activé
source etlenv/bin/activate  # Linux/Mac
# ou
etlenv\Scripts\activate     # Windows

# Placez les fichiers CSV (brut) dans le répertoire data/raw/

# Exécutez le pipeline ETL
python run_etl.py
```

Figure 7: Exécution le pipeline ETL

7. CI/CD et Tests Automatisés

7.1 Stratégie de tests

Notre approche de test :

Tests unitaires

- Chaque classe et fonction importante possède des tests unitaires.

7.2 Pipeline CI/CD avec GitHub Actions

Notre pipeline CI/CD est configuré via GitHub Actions pour automatiser les tests et le

Pipeline CI/CD avec GitHub Actions — YAML

```yaml
name: CI/CD Pipeline for ETL (Test uniquement)

on:
  push:
    branches:
      - master # Déclenche le workflow à chaque push sur la branche master

jobs:
  test-and-notify:
    runs-on: ubuntu-latest # Utilise une machine virtuelle Ubuntu

    steps:
      # Étape 1 : Checkout du code
      - name: Checkout code
        uses: actions/checkout@v2 # Récupère le code du dépôt

      # Étape 2 : Configuration de Python
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: "3.12.0"

      # Étape 3 : Installation des dépendances
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip  # Met à jour pip
          pip install -r requirements.txt  # Installe les dépendances du projet
          pip install pytest  # Installe pytest pour les tests

      # Étape 4 : Exécution des tests
      - name: Run tests
        id: tests # Donne un ID à cette étape pour récupérer le résultat plus tard
        run: |
          export PYTHONPATH=$PYTHONPATH:$(pwd)/src
          pytest tests/test.py

      # Étape 5 : Envoi d'un e-mail avec les résultats des tests
      - name: Send email notification
        if: always() # Exécute cette étape même si les tests échouent
        uses: dawidd6/action-send-mail@v3
        with:
          server_address: smtp.gmail.com # Serveur SMTP (pour Gmail)
          server_port: 465
          username: ${{ secrets.EMAIL_USERNAME }} # Ton adresse e-mail (configurée dans les
secrets GitHub)
          password: ${{ secrets.EMAIL_PASSWORD }} # Mot de passe ou token d'application (dans
les secrets GitHub)
          subject: "Résultats des tests du pipeline ETL"
          body: |
            Les tests du pipeline ETL ont été exécutés.
            Résultat : ${{ steps.tests.outcome }}
            Voir les détails ici : ${{ github.server_url }}/${{ github.repository
}}/actions/runs/${{ github.run_id }}
          to: ${{ secrets.TEAM_EMAILS }} # Liste des adresses e-mail de l'équipe (configurée
dans les secrets GitHub)
          from: ${{ secrets.EMAIL_USERNAME }} # Expéditeur
```

déploiement :

Figure 8: Pipeline CI/CD avec GitHub Actions

7.3 Notifications par email

Le système de notification par email est configuré pour alerter l'équipe en cas de :

- Échec des tests automatisés
- Problèmes de build
- Déploiements réussis
- Dépassement des seuils de performance

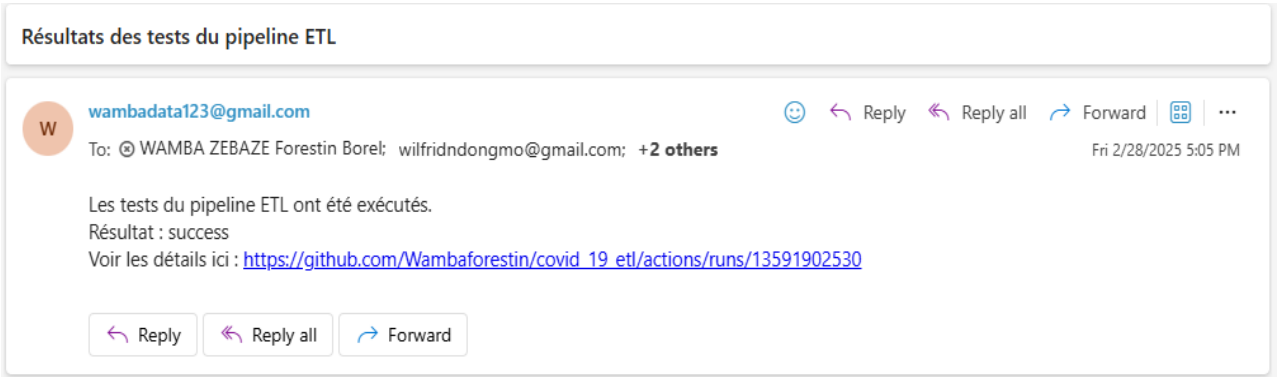

Figure 9: Notifications par email après l'éxécution des tests

8. Maintenance et Dépannage

8.1 Logs et monitoring

Notre système de logging est structuré pour faciliter le diagnostic et le monitoring :

- Les logs sont stockés dans le répertoire logs/
- Format standardisé : [TIMESTAMP] - [MODULE] - [LEVEL] - [MESSAGE]
- Niveaux de log configurables via variable d'environnement

8.2 Problèmes courants et solutions

Voici quelques problèmes fréquemment rencontrés et leurs solutions :

Problème	Cause probable	Solution
psycopg2.OperationalError: connection refused	Base de données PostgreSQL non démarrée	Vérifier que le service PostgreSQL est actif
FileNotFoundError: data/raw/covid19_global_cases.csv	Fichier source manquant	Placer le fichier CSV dans le répertoire spécifié
sqlalchemy.exc.IntegrityError	Violation de contrainte lors de l'insertion	Vérifier la validation des données dans le transformer
Erreurs de mapping des pays	Différences dans les	Ajouter le mapping dans le

Problème	Cause probable	Solution
	noms de pays	normalizer.py
Performance lente sur gros fichiers	Consommation excessive de mémoire	Ajuster le paramètre chunksize dans le loader

9. Bonnes Pratiques et Recommandations

9.1 Standards de code

Pour maintenir une qualité de code élevée, nous suivons ces standards :

- **Black** pour le formatage automatique
- **Docstrings** pour toutes les classes et fonctions publiques
- **Type hints** pour améliorer la lisibilité et le support IDE
- **Tests unitaires** pour les nouvelles fonctionnalités
- **Revue de code** obligatoire avant merge sur la branche principale

9.2 Sécurité des données

Pour garantir la sécurité des données sensibles :

- Ne jamais stocker les identifiants de connexion dans le code
- Utiliser des variables d'environnement ou **.env** pour les secrets
- Restreindre les droits d'accès à la base de données
- Valider toutes les entrées pour éviter les injections SQL

9.3 Performance et optimisation

Recommandations pour optimiser les performances :

- Traiter les gros fichiers par chunks pour limiter l'utilisation mémoire
- Utiliser des index appropriés sur les colonnes fréquemment consultées
- Préférer les opérations vectorisées Pandas aux boucles Python
- Monitorer les temps d'exécution des différentes étapes

9.4 Limitations actuelles et évolutions futures

Il est important de noter que, notre système présente actuellement certaines limitations :

Architecture pour nouvelles sources de données

Notre système est conçu pour intégrer de nouvelles sources de données avec un minimum de modifications.

Exemple concret : Le fichier de configuration permet d'ajouter de nouvelles sources sans modifier le code :

Configuration extensible dans config.py — Python

```python
# Configuration extensible dans config.py
@property
def DATA_SOURCES(self):
    return {
        'covid19': os.path.join(self.DATA_PATHS['raw'], 'covid19_global_cases.csv'),
        'mpox': os.path.join(self.DATA_PATHS['raw'], 'mpox_global_cases.csv'),
        'influenza': os.path.join(self.DATA_PATHS['raw'], 'influenza_cases.json'),
        # Facile d'ajouter une nouvelle source ici
    }

# Factory pattern pour créer l'extracteur approprié
def create_extractor(source_name):
    source_path = config.DATA_SOURCES.get(source_name)
    if not source_path:
        raise ValueError(f"Source inconnue: {source_name}")

    if source_path.endswith('.csv'):
        return CSVExtractor(source_path)
    elif source_path.endswith('.json'):
        return JSONExtractor(source_path)
    elif source_path.startswith('http'):
        return APIExtractor(source_path)
    else:
        raise ValueError(f"Format non pris en charge: {source_path}")
```

Figure 10: Configuration extensible dans config.py

Grâce à cette approche, ajouter une nouvelle source de données ne nécessite que l'ajout d'une entrée dans la configuration et, si nécessaire, l'implémentation d'un nouvel extracteur.

Conteneurisation pour déploiement flexible

La conteneurisation facilite le déploiement dans différents environnements et améliore la scalabilité. Cette approche facilite également l'intégration avec des plates-formes d'orchestration comme Kubernetes pour une scalabilité horizontale.

Déclenchement manuel

Le pipeline ETL nécessite un lancement manuel pour être exécuté et ne s'exécute pas automatiquement selon un calendrier ou en réponse à des événements.

Mise à jour manuelle des données sources

Les fichiers sources doivent être placés manuellement dans le répertoire **data/raw/** avant l'exécution du pipeline.

Pour surmonter ces limitations et améliorer l'automatisation du système, nous prévoyons d'intégrer les outils suivants dans les versions futures :

i. **Apache Airflow** : Pour orchestrer et planifier l'exécution automatique du pipeline selon un calendrier défini, avec gestion des dépendances entre tâches et interface de monitoring.

ii. **Apache Kafka** : Pour implémenter une architecture orientée événements où de nouvelles données déclenchent automatiquement le pipeline, permettant un traitement en temps quasi-réel.

iii. **Airbyte** : Pour faciliter l'extraction automatique depuis diverses sources de données, avec des connecteurs préconçus et une interface de configuration simplifiée.

En adoptant ces principes d'évolutivité et en planifiant ces améliorations futures, notre solution ETL reste maintenable et extensible, capable de s'adapter à de nouveaux besoins sans nécessiter une refonte complète. Cette flexibilité est particulièrement importante dans le contexte épidémiologique où les besoins d'analyse peuvent évoluer rapidement.

10. Annexes

10.1 Glossaire

Terme	Définition
ETL	Extract, Transform, Load - Processus d'extraction, transformation et chargement de données
CSV	Comma-Separated Values - Format de fichier texte où les valeurs sont séparées par des virgules
PostgreSQL	Système de gestion de base de données relationnelle open-source
ORM	Object-Relational Mapping - Technique de programmation pour convertir des données entre systèmes incompatibles
Pandas	Bibliothèque Python pour la manipulation et l'analyse de données
SQLAlchemy	Bibliothèque Python SQL toolkit et ORM
Airflow	Plateforme d'orchestration de workflows de traitement de données
Kafka	Plateforme de streaming distribuée pour le traitement des flux de données en temps réel

10.2 Références

- Documentation Python : https://docs.python.org/3/
- Documentation PostgreSQL : https://www.postgresql.org/docs/
- Tutoriel ETL avec Python : *https://www.youtube.com/watch?v=dfouoh9QdUw&t=519s*
- Documentation SQLAlchemy : https://docs.sqlalchemy.org/

- Documentation Pandas : https://pandas.pydata.org/docs/
- GitHub Actions : https://docs.github.com/en/actions