M2 BIDABI, BIG DATA

Système de Surveillance des Données de Pression Artérielle patient avec Kafka, Elasticsearch et Kibana

DOCUMENTATION DU SYSTÈME ET SON DÉPLOIEMENT

GUENDOUZ MAHI ELAMINE
BAO PHILIPPE

CONTEXTE :

La surveillance des patients à partir de leurs mesures de pression artérielle (blood pressure) est cruciale pour détecter les cas nécessitant une attention médicale rapide. En s'appuyant sur le standard FHIR (Fast Healthcare Interoperability Resources), ce projet vise à développer une solution pour analyser ces données, identifier les anomalies, et gérer efficacement les résultats en vue d'un suivi renforcé.

OBJECTIF GÉNÉRAL :

Créer un système qui génère des messages FHIR contenant des données de pression artérielle, les transmet via **Kafka**, détecte les anomalies nécessitant un suivi médical renforcé, et traite les données comme suit :

- Les données anormales (pression artérielle trop élevée ou trop basse) sont indexées dans **Elasticsearch** et visualisées dans **Kibana** pour un suivi détaillé.
- Les données normales sont archivées localement.

OBJECTIFS SPÉCIFIQUES :

1. **Génération des Messages FHIR :**
    - Implémenter un module Python pour générer des messages FHIR au format JSON, contenant des observations de pression artérielle (systolique et diastolique) pour différents patients.
2. **Transmission avec Kafka :**
    - Configurer Kafka pour gérer les échanges de données :

- **Producer Python** : Publier les messages FHIR sur un topic Kafka.
- **Consumer Python** : Consommer des messages depuis un topic kafka.

3. Analyse des Données de Pression Artérielle :

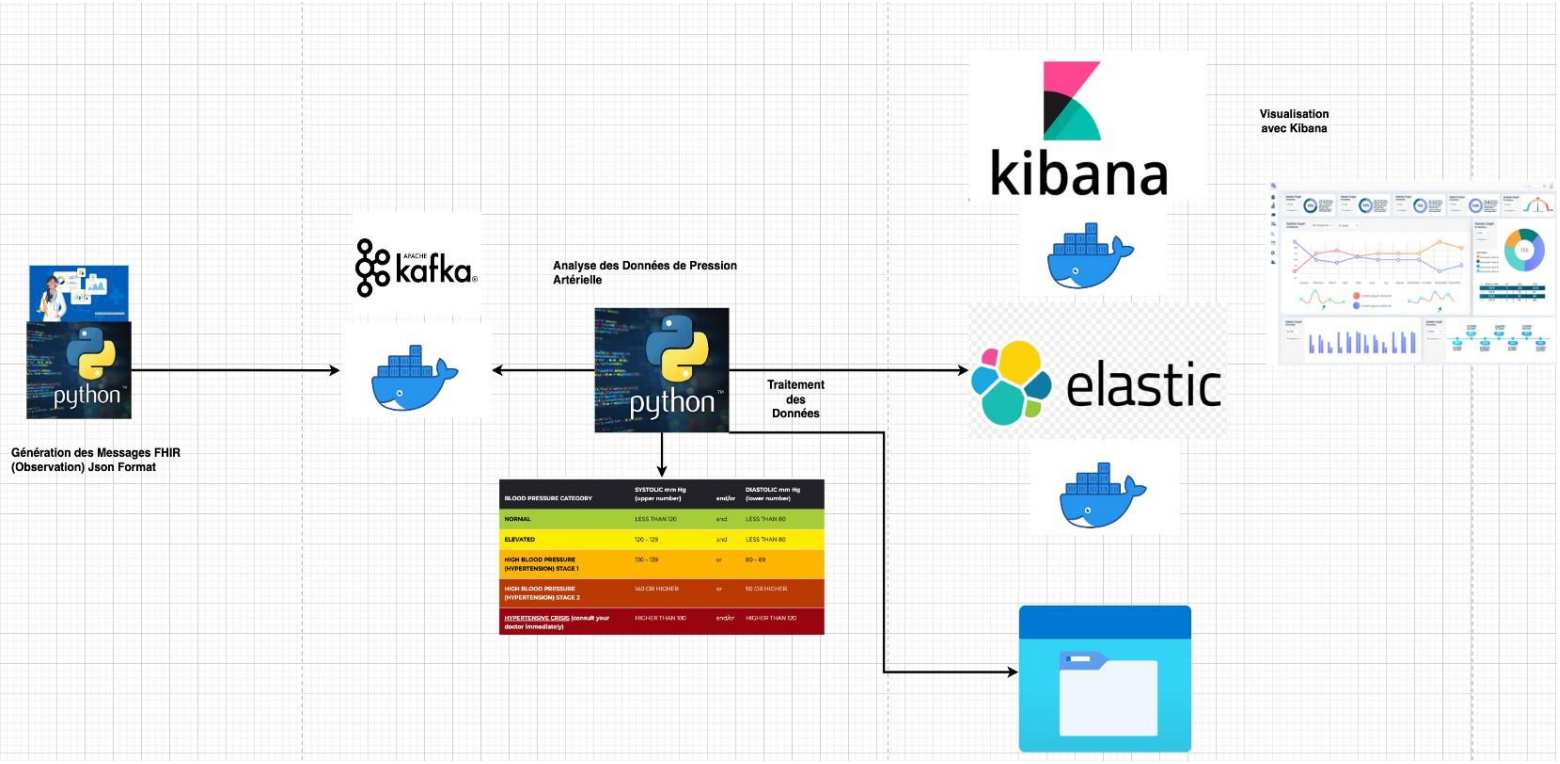
- Développer des règles d'analyse pour détecter les anomalies :
  - Pression artérielle systolique supérieure à 140 mmHg ou inférieure à 90 mmHg.
  - Pression artérielle diastolique supérieure à 90 mmHg ou inférieure à 60 mmHg.
- Identifier les patients dont les valeurs sortent de ces plages comme nécessitant un suivi médical renforcé.

4. Traitement des Données :

- Patients avec pression artérielle anormale :
  - Indexer leurs données dans **Elasticsearch**, avec des métadonnées précisant les anomalies détectées.
- Patients avec pression artérielle normale :
  - Sauvegarder leurs données dans des fichiers JSON locaux pour archivage.

5. Visualisation avec Kibana :

- Configurer Kibana pour afficher les anomalies, y compris :
  - Distribution des patients avec des pressions artérielles anormales.
  - Analyse des tendances (ex. : pics de pression artérielle par période).
  - Alertes sur les cas critiques.



SCRIPTS PYTHON :

Pour mener à bien les objectifs, nous avons décidé de segmenter le travail en trois scripts Python distincts.

PRODUCER.PY :

(Voir annexe 1)

Code de Simulation et Publication d'Observations FHIR avec Kafka

Script Python qui génère et publie des observations de pression artérielle dans un cluster Kafka. Ce pipeline est conçu pour simuler des données médicales conformes au standard FHIR (Fast Healthcare Interoperability Resources) et les transmettre efficacement via Kafka.

Description Générale

Le code est structuré pour :

1. Générer des observations de pression artérielle conformes au standard FHIR pour 200 patients.

2. Simuler une période d'observation de 30 jours avec quatre mesures par jour pour chaque patient.

3. Publier les observations générées dans un topic Kafka nommé `fhir_observations`.

Le pipeline comprend plusieurs étapes :

- Configuration et gestion des logs.
- Création des observations conformes à FHIR.
- Génération de données aléatoires.
- Publication dans Kafka.

Analyse Détaillée

Importation des Bibliothèques

Le code utilise plusieurs bibliothèques clés :

- **`confluent_kafka`** : Pour la publication de messages Kafka.
- **`json`** : Manipulation des données JSON.
- **`datetime`, `timedelta`, et `pytz`** : Gestion des dates et fuseaux horaires.
- **`random`** : Génération aléatoire des valeurs de pression artérielle.

- **`fhir.resources.observation`** : Création des ressources conformes à FHIR.
- **`logging`** : Enregistrement des événements et erreurs.

Fonction create_blood_pressure_observation

Cette fonction crée une observation médicale conforme au standard FHIR. Elle prend en entrée :

- **`patient_id`** : Identifiant unique du patient.
- **`systolic_pressure`** et **`diastolic_pressure`** : Pression systolique et diastolique.
- **`date`** : Date et heure de l'observation.

La fonction renvoie un objet JSON formaté avec les attributs requis par FHIR, notamment :

- Type de ressource (`Observation`).
- Catégorie (« Signes vitaux »).
- Pressions systolique et diastolique avec unités (« mmHg »).

Configuration du Producer Kafka

Un producteur Kafka est configuré pour se connecter à un cluster via :

- **`bootstrap.servers`** : Adresse du serveur Kafka.
- **`client.id`** : Identifiant unique du producteur.

Fonction publish_message

Cette fonction publie un message dans un topic Kafka spécifique. Un callback (`delivery_report`) est utilisé pour gérer les erreurs ou confirmer la livraison.

Génération des Données Simulées

Pour chaque patient, le code génère des observations sur une période de 30 jours :

- **Patients** : 200 patients avec des identifiants uniques.
- **Observations** : 4 par jour (toutes les 6 heures).
- **Valeurs aléatoires** : Pression systolique (70-190 mmHg) et diastolique (60-130 mmHg).
- **Formatage FHIR** : Les observations sont formatées conformément au standard.

Publication dans Kafka

Les observations sont publiées dans le topic `fhir_observations`. Le processus inclut :

- Envoi de chaque observation JSON.
- Enregistrement des événements et gestion des erreurs.

Logging

Le code utilise le module `logging` pour suivre l'exécution. Les logs incluent :

- Nombre de patients traités.
- Dates de début et de fin d'observation.
- Nombre total d'observations générées.
- Confirmation de la publication des messages.

Résultats

- Nombre total de patients : 200.

- **Nombre total d'observations** : 200 patients * 30 jours * 4 observations = 24 000 observations.

- **Publication Kafka** : Les observations ont été publiées avec succès dans le topic `fhir_observations`.

CONSUMER.PY :

(Voir annexe 2)

Code de Consommation Kafka pour Observations FHIR

Script Python permettant de consommer des messages Kafka contenant des observations FHIR (Fast Healthcare Interoperability Resources). Le script s'abonne à un topic Kafka, traite les messages reçus, et affiche les observations sous forme JSON.

Description Générale

Le code implémente un consommateur Kafka qui :

1. Se connecte à un cluster Kafka.
2. S'abonne à un topic spécifique (« `fhir_observations` »).
3. Traite les messages reçus en les décodant en JSON.
4. Gère les erreurs potentielles de Kafka ou de décodage JSON.

Analyse Détaillée

Importation des Bibliothèques

Le script utilise les bibliothèques suivantes :

- **`confluent_kafka`** : Pour consommer les messages depuis un cluster Kafka.
- **`json`** : Pour décoder et manipuler les messages au format JSON.

- **`logging`** : Pour enregistrer des informations, erreurs ou avertissements.

Configuration du Consommateur Kafka

Le consommateur Kafka est configuré via un dictionnaire `conf` :

- **`bootstrap.servers`** : Adresse du serveur Kafka.
- **`group.id`** : Identifiant du groupe de consommateurs.
- **`auto.offset.reset`** : Définit la stratégie pour consommer les messages (ici, depuis le début si aucun offset précédent n'existe).

Un objet consommateur est créé à l'aide de la classe `Consumer`.

Abonnement au Topic Kafka

Le consommateur s'abonne au topic nommé `fhir_observations` via la méthode `subscribe`. Cela lui permet de recevoir tous les messages publiés dans ce topic.

Traitement des Messages

Les messages reçus sont traités dans une boucle infinie :

1. **Polling** : La méthode `poll` est utilisée pour récupérer les messages avec un timeout d’1 seconde.
2. Vérification des erreurs :
    - Si aucun message n’est reçu (équivalent à `None`), la boucle continue.
    - Si un message contient une erreur, elle est vérifiée et gérée (ég., fin de partition ou erreur fatale).
3. Traitement des messages valides :

- Les messages valides sont décodés en JSON via la fonction `process_message`.
- Le contenu est affiché sous une forme indentée pour une meilleure lisibilité.

Fonction process_message

Cette fonction :

- Décode le message reçu en JSON.
- Affiche le contenu dans le log si le décodage réussit.
- Capture et logue toute erreur de décodage JSON.

Gestion des Exceptions

Le code gère les exceptions potentielles :

- **Erreurs Kafka** : Échec de connexion ou erreur fatale du consommateur.
- **Interruption utilisateur** : Le consommateur Kafka est arrêté proprement si le programme est interrompu par l'utilisateur (via `KeyboardInterrupt`).
- **Fermeture gracieuse** : La méthode `close` est appelée pour libérer les ressources.

Logging

Le module `logging` est configuré pour afficher les messages de niveau `INFO` et `ERROR`. Les logs incluent :

- Démarrage et arrêt du consommateur.
- Contenu des messages reçus.
- Erreurs de décodage JSON ou de Kafka.

Résultats

Le script consomme les messages publiés dans le topic Kafka `fhir_observations` et affiche leur contenu JSON. Il gère correctement les erreurs et garantit une exécution stable.

ANOMALIES_TRAITEMENT.PY :

(Voir annexe 3)

Code de Détection d'Anomalies avec Kafka et Elasticsearch

Script Python qui consomme des messages Kafka contenant des observations FHIR, identifie des anomalies dans les données de pression artérielle, et stocke ces anomalies dans Elasticsearch. Les observations normales sont sauvegardées localement pour une analyse ultérieure.

Description Générale

Le script est conçu pour :

1. Consommer des messages Kafka depuis le topic `fhir_observations`.
2. **Analyser les données FHIR** pour identifier les anomalies de pression artérielle.
3. Enregistrer les anomalies dans Elasticsearch.
4. Sauvegarder localement les données normales.

Analyse Détailée

Importation des Bibliothèques

Le script utilise les bibliothèques suivantes :

- **`confluent_kafka`** : Pour consommer les messages Kafka.

- **`elasticsearch`** : Pour interagir avec un cluster Elasticsearch.
- **`json`** : Pour manipuler les données au format JSON.
- **`os`** : Pour gérer les fichiers et répertoires.
- **`logging`** : Pour enregistrer les événements, erreurs, et informations.

Configuration

Kafka

Le consommateur Kafka est configuré avec :

- **`bootstrap.servers`** : Adresse du serveur Kafka.
- **`group.id`** : Identifiant du groupe de consommateurs.
- **`auto.offset.reset`** : Stratégie de consommation (depuis le début si aucun offset précédent n'existe).

Elasticsearch

- Une connexion à Elasticsearch est établie via l'URL `http://elasticsearch:9200`.
- L'index utilisé pour stocker les anomalies est nommé `fhir_observations_anomalies`.
- Une vérification initiale de la connexion à Elasticsearch est effectuée. En cas d'échec, le script s'arrête.

Sauvegarde des Données Normales

Un répertoire local (`normal_data`) est créé pour stocker les observations jugées normales.

Fonctions Principales

categorize_blood_pressure

Cette fonction catégorise les pressions artérielles systoliques et diastoliques selon les critères médicaux :

- **Normal** : Systolique < 120 et Diastolique < 80.
- **Elevated** : Systolique entre 120-129 et Diastolique < 80.
- **Hypertension Stage 1** : Systolique entre 130-139 ou Diastolique entre 80-89.
- **Hypertension Stage 2** : Systolique ≥ 140 ou Diastolique ≥ 90.
- **Hypertensive Crisis** : Systolique > 180 ou Diastolique > 120.

process_observation

Cette fonction traite chaque message JSON :

1. Décode le message en un dictionnaire Python.
2. Extrait les pressions systolique et diastolique des composants de l'observation.
3. Catégorise la pression artérielle à l'aide de `categorize_blood_pressure`.
4. Si l'observation est normale, elle est sauvegardée localement dans un fichier JSON (un fichier par patient).
5. Si une anomalie est détectée, elle retourne un dictionnaire contenant les détails de l'anomalie.

send_to_elasticsearch

Cette fonction envoie les anomalies détectées à Elasticsearch. En cas d'échec, une erreur est enregistrée dans les logs.

Boucle Principale

Le script consomme les messages Kafka en boucle :

1. **Polling** : Les messages sont récupérés avec un délai d'attente de 1 seconde.
2. Gestion des erreurs :
    - Les erreurs mineures (comme la fin d'une partition) sont loguées.
    - Les erreurs fatales entraînent l'arrêt du script.
3. Traitement des messages :
    - Chaque message valide est analysé par `process_observation`.
    - Les anomalies détectées sont envoyées à Elasticsearch via `send_to_elasticsearch`.

Gestion des Exceptions

Le script gère les interruptions (par exemple, `Ctrl+C`) et les exceptions liées à Kafka, Elasticsearch, ou au traitement JSON.

Logging

Le module `logging` enregistre des informations clés :

- Démarrage et arrêt du consommateur.
- Connexion à Elasticsearch.
- Anomalies détectées.
- Erreurs lors de la consommation ou du traitement des messages.

Résultats

- Les messages Kafka contenant des observations FHIR sont analysés en temps réel.
- Les anomalies sont correctement détectées, catégorisées et indexées dans Elasticsearch.
- Les observations normales sont sauvegardées localement pour une analyse ultérieure.

DOCKER-COMPOSE.YAML :

(Voir annexe 4)

Docker Compose pour l'Infrastructure Kafka et Elasticsearch

Le fichier Docker Compose permet de déployer une infrastructure comprenant des services pour la production et la consommation de messages Kafka, ainsi qu'un service de détection d'anomalies intégrant Elasticsearch. Cette configuration facilite la communication entre les services grâce à des réseaux Docker partagés.

Description Générale

Le fichier Docker Compose définit trois services principaux :

- **Producer** : Produit des messages et les envoie à un cluster Kafka.
- **Consumer** : Consomme les messages depuis Kafka.
- **Anomaly Detector** : Analyse les messages consommés pour détecter des anomalies et les enregistre dans Elasticsearch.

Deux réseaux Docker externes sont utilisés pour faciliter la communication :

- **`kafka_net_kafka_nifi`** : Connecte les services à Kafka.
- **`elasticsearch_net_es`** : Connecte le service de détection d'anomalies à Elasticsearch.

Analyse Détailée

Services Définis

Producer

- Build :
  - Construit l'image Docker à partir d'un `Dockerfile` situé dans le même répertoire.
- **Command** : Exécute le script `producer.py`.
- Environnement :
  - `KAFKA_BOOTSTRAP_SERVERS` : Définit l'adresse du serveur Kafka pour la connexion.
- Network :
  - Se connecte au réseau `kafka_net_kafka_nifi` pour interagir avec Kafka.

Consumer

- Build :
  - Construit l'image Docker à partir du même `Dockerfile`.
- **Command** : Exécute le script `consumer.py`.
- Environnement :
  - `KAFKA_BOOTSTRAP_SERVERS` : Définit l'adresse du serveur Kafka.
- Network :
  - Se connecte au réseau `kafka_net_kafka_nifi` pour interagir avec Kafka.

Anomaly Detector

- Build :
    - Construit l'image Docker en utilisant le même `Dockerfile` que les autres services.
- **Command** : Exécute le script anomalies_traitement.py.
- Environnement :
    - `KAFKA_BOOTSTRAP_SERVERS` : Adresse du serveur Kafka.
    - `ELASTICSEARCH_HOST` : Adresse d'Elasticsearch.
- Volumes :
    - Monte un répertoire local (`./normal_data`) dans le conteneur pour stocker les observations normales.
- Networks :
    - `kafka_net_kafka_nifi` : Pour interagir avec Kafka.
    - `elasticsearch_net_es` : Pour se connecter à Elasticsearch.

Réseaux

kafka_net_kafka_nifi

- Réseau externe permettant aux services de communiquer avec Kafka.

elasticsearch_net_es

- Réseau externe permettant au service de détection d'anomalies de communiquer avec Elasticsearch.

Fonctionnalités

1. **Isolation des services** : Chaque service utilise son propre conteneur, ce qui garantit une isolation des environnements.

2. Communication interservices :

    - Kafka facilite la transmission des messages entre les producteurs et les consommateurs.

    - Elasticsearch permet de stocker les données analysées par le service de détection d'anomalies.

3. **Persistance locale** : Les données normales sont sauvegardées localement dans un volume partagé avec le conteneur du détecteur d'anomalies.

4. **Flexibilité** : Les réseaux externes permettent d'utiliser un cluster Kafka ou Elasticsearch déjà configuré.

Résultats

- Le fichier Docker Compose est bien conçu pour déployer une infrastructure distribuée qui inclut Kafka et Elasticsearch.

- Les services sont correctement configurés pour communiquer entre eux via des réseaux partagés.

- Les volumes locaux permettent une gestion efficace des données.

DOCKERFILE :

(Voir annexe 5)

Le fichier Dockerfile est utilisé pour créer une image Docker qui exécute des scripts Python dans un environnement conteneurisé. Voici un résumé de ce qu'il fait :

Image de base

Utilise une image officielle de Python en version 3.10-slim (une version légère de Python).

Répertoire de travail

Définit le répertoire de travail dans le conteneur comme `/app`. Toutes les commandes suivantes seront exécutées dans ce répertoire.

Copie des fichiers

Copie les fichiers nécessaires dans le conteneur :

- `producer.py` : Script pour produire des données.

- `consumer.py` : Script pour consommer des données.

- `anomalies_traitement.py` : Script pour détecter des anomalies.

- `requirements.txt` : Fichier listant les dépendances Python à installer.

Installation des dépendances

Installe les dépendances Python listées dans `requirements.txt` en utilisant `pip`.

L'option `--no-cache-dir` permet de réduire la taille de l'image en évitant de stocker le cache des packages.

Résumé

Ce Dockerfile crée une image Docker qui :

1. Utilise Python 3.10 comme base.

2. Installe les dépendances nécessaires pour exécuter les scripts Python.

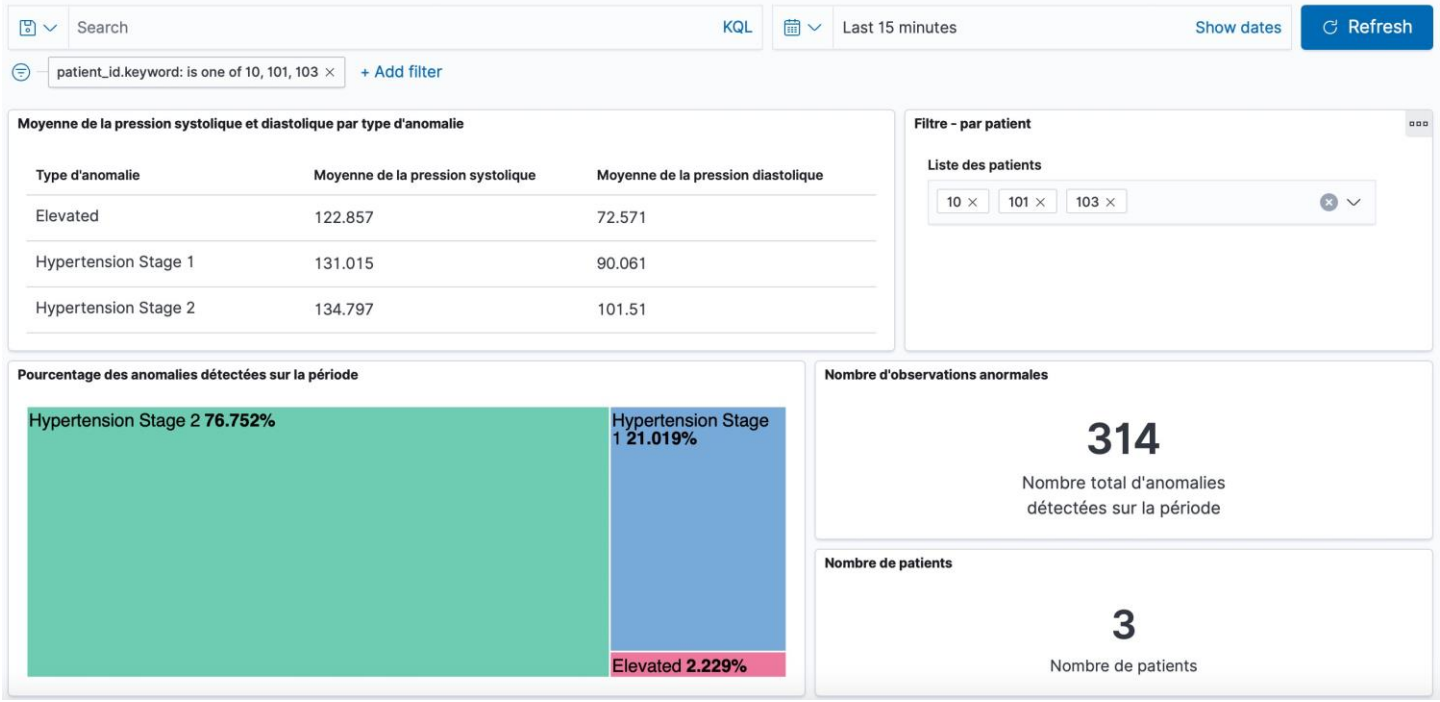
3. Copie les scripts et le fichier de dépendances dans le conteneur.

En résumé, il permet de ***conteneuriser*** les scripts Python pour les exécuter dans un environnement isolé et reproductible.

PROBLÈMES RENCONTRÉS !!!

Lors du déploiement de l'application, plusieurs problèmes ont été rencontrés. Tout d'abord, **la compatibilité avec Python 3.10 et les versions antérieures** a nécessité une attention particulière pour garantir le bon fonctionnement du code. Ensuite, **ElasticSearch en version 7.9.1** a présenté des défis liés à la gestion des dépendances, comme indiqué dans le fichier `requirements.txt`, ce qui a nécessité des ajustements spécifiques. Enfin, un problème récurrent a été l'absence ou la mauvaise configuration préalable des services **Kafka** et **ElasticSearch** sur la machine, rendant nécessaire leur déploiement manuel avant de pouvoir exécuter correctement l'application.

DASHBOARD KIBANA



Le dashboard Kibana présenté offre une visualisation des données collectées et analysées sur les observations de pression artérielle détectées comme anormales pour les patients 10, 101, et 103 sur les 15 dernières minutes. Voici les informations principales :

1. **Moyenne des pressions systolique et diastolique par type d'anomalie** : La table en haut à gauche présente les moyennes de la pression systolique et diastolique selon les types d'anomalies détectées (Hypertension Stage 1, Hypertension Stage 2). Par exemple, les patients classés en Hypertension Stage 2 ont une moyenne systolique de 134.797 mmHg et une diastolique de 101.51 mmHg.

2. **Répartition des anomalies détectées** : Le graphique en bas à gauche montre la distribution des types d'anomalies sous forme de pourcentages. La majorité (76.752 %) correspond à l'Hypertension Stage 2, suivie de l'Hypertension Stage 1 (21.019 %) et des cas Elevated (2.229 %).

3. **Nombre total d'observations anormales** : La boîte en bas à droite indique qu'un total de 314 anomalies a été détecté au cours de la période considérée.

4. **Nombre de patients analysés** : L'analyse porte sur 3 patients, filtrés par leurs identifiants (10, 101, 103), comme indiqué dans le panneau de filtrage en haut à droite.

Processus :

Les données ont été collectées depuis Kafka, où les observations de pression artérielle ont été publiées. Le service de détection d'anomalies a traité ces observations pour catégoriser les pressions artérielles selon des critères médicaux (Elevated, Hypertension Stage 1, etc.). Les anomalies détectées ont ensuite été envoyées à Elasticsearch pour stockage et visualisation via Kibana. Ce dashboard permet d'avoir une vue d'ensemble rapide et claire des anomalies par type et par patient.

ANNEXE 1:

```python
from confluent_kafka import Producer
import json
from datetime import datetime, timedelta
import random
import pytz
from fhir.resources.observation import Observation
import logging

# Configuration du logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Fonction pour créer une observation de pression artérielle
def create_blood_pressure_observation(patient_id, systolic_pressure, diastolic_pressure, date):
    if date.tzinfo is None:
        date = date.astimezone(pytz.UTC)

    observation_data = {
        "resourceType": "Observation",
        "id": "blood-pressure",
        "meta": {"profile": ["http://hl7.org/fhir/StructureDefinition/vitalsigns"]},
        "status": "final",
        "category": [
            {
                "coding": [
                    {
                        "system": "http://terminology.hl7.org/CodeSystem/observation-category",
                        "code": "vital-signs",
                        "display": "Signes vitaux"
                    }
                ],
                "text": "Signes vitaux"
            }
        ],
        "code": {
            "coding": [
                {
                    "system": "http://loinc.org",
                    "code": "85354-9",
                    "display": "Blood pressure"
                }
            ],
            "text": "Blood pressure"
        },
        "subject": {"reference": f"Patient/{patient_id}"},
        "effectiveDateTime": date.isoformat(),
        "component": [
            {
                "code": {
                      "coding": [
                          {
                              "system": "http://loinc.org",
                              "code": "8480-6",
                              "display": "Systolic blood pressure"
                          }
                      ],
                      "text": "Systolic blood pressure"
                  },
                  "valueQuantity": {
                      "value": systolic_pressure,
                      "unit": "mmHg",
                      "system": "http://unitsofmeasure.org",
                      "code": "mm[Hg]"
                  }
              },
              {
                  "code": {
                      "coding": [
                          {
                              "system": "http://loinc.org",
                              "code": "8462-4",
                              "display": "Diastolic blood pressure"
                          }
                      ],
                      "text": "Diastolic blood pressure"
                  },
                  "valueQuantity": {
                      "value": diastolic_pressure,
                      "unit": "mmHg",
                      "system": "http://unitsofmeasure.org",
                      "code": "mm[Hg]"
                  }
              }
          ]
    }

    observation = Observation(**observation_data)
    return observation.json(indent=4)

# Configuration du producteur Kafka
conf = {
    'bootstrap.servers': 'kafka1:29092',  # Utiliser le nom du service Docker
    'client.id': 'python-producer'
}

# Callback pour la gestion des erreurs
def delivery_report(err, msg):
    if err is not None:
        logging.error(f"Message delivery failed: {err}")
    else:
        logging.info(f"Message delivered to {msg.topic()} [{msg.partition()}]")

# Créer un producteur Kafka
producer = Producer(conf)

# Fonction pour publier un message dans Kafka
def publish_message(topic, message):
    try:
        producer.produce(topic, key="fhir-observation", value=message, callback=delivery_report)
        producer.flush()
    except Exception as e:
        logging.error(f"Erreur lors de la publication du message : {e}")

# Générer des observations pour 200 patients
patients = [{"id": str(i), "name": f"Patient {i}"} for i in range(1, 201)]  # 200 patients
all_observations = []

# Log pour vérifier le nombre de patients
logging.info(f"Nombre de patients : {len(patients)}")

for patient in patients:
    patient_id = patient["id"]
    start_date = datetime.now(pytz.UTC) - timedelta(days=30)  # Période de 30 jours
    end_date = datetime.now(pytz.UTC)

    # Log pour vérifier les dates de début et de fin
    logging.info(f"Traitement du patient {patient_id} - Date de début : {start_date}, Date de fin : {end_date}")

    current_date = start_date
    for day in range(30):  # 30 jours
        for observation_count in range(4):  # 4 observations par jour (toutes les 6 heures)
            systolic_pressure = random.randint(70, 190)
            diastolic_pressure = random.randint(60, 130)
            observation_json = create_blood_pressure_observation(patient_id, systolic_pressure, diastolic_pressure, current_date)
            all_observations.append(observation_json)
            current_date += timedelta(hours=6)  # Intervalle de 6 heures

# Log pour vérifier le nombre total d'observations générées
logging.info(f"Nombre total d'observations générées : {len(all_observations)}")

# Publier les observations dans Kafka![<# alt text #>](../../../../../var/folders/l1/48nh_bys3nb0frvc2q508lr00000gn/T/TemporaryIte
topic = "fhir_observations"
for observation_json in all_observations:
    publish_message(topic, observation_json)

logging.info(f"Tous les messages ont été envoyés au topic '{topic}'.")
```

ANNEXE 2:

```python
import json
import logging

# Configuration du logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Configuration du consommateur Kafka
conf = {
    'bootstrap.servers': 'kafka1:29092',  # Utiliser le nom du service Docker
    'group.id': 'python-consumer',
    'auto.offset.reset': 'earliest'
}

# Créer un consommateur Kafka
consumer = Consumer(conf)

# S'abonner au topic
topic = "fhir_observations"
consumer.subscribe([topic])

# Fonction pour traiter les messages reçus
def process_message(message):
    try:
        data = json.loads(message)
        logging.info(f"Message reçu : {json.dumps(data, indent=4)}")
    except json.JSONDecodeError as e:
        logging.error(f"Erreur de décodage JSON : {e}")

# Consommer des messages en boucle
try:
    logging.info("Démarrage du consommateur Kafka.")
    while True:
        msg = consumer.poll(timeout=1.0)
        if msg is None:
            continue
        if msg.error():
            if msg.error().code() == KafkaError._PARTITION_EOF:
                logging.info(f"Fin de la partition : {msg.partition()}")
            else:
                raise KafkaException(msg.error())
        else:
            process_message(msg.value().decode('utf-8'))

except KeyboardInterrupt:
    logging.info("Arrêt du consommateur Kafka.")
finally:
    consumer.close()
    logging.info("Consommateur Kafka arrêté.")
```

ANNEXE 3:

```python
from confluent_kafka import Consumer, KafkaException, KafkaError
from elasticsearch import Elasticsearch, exceptions as es_exceptions
import json
import os
import logging

# Configuration du logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Configuration du consommateur Kafka
kafka_conf = {
    'bootstrap.servers': 'kafka1:29092',  # Utiliser le nom du service Docker
    'group.id': 'anomaly-detection-group',
    'auto.offset.reset': 'earliest'
}

# Configuration d'Elasticsearch
es = Elasticsearch(["http://elasticsearch:9200"])
es_index = "fhir_observations_anomalies"

# Vérifier la connexion à Elasticsearch
try:
    es.info()
    logging.info("Connexion à Elasticsearch réussie.")
except es_exceptions.ConnectionError as e:
    logging.error(f"Erreur de connexion à Elasticsearch : {e}")
    exit(1)

# Dossier pour sauvegarder les données normales
normal_data_dir = "./normal_data"
os.makedirs(normal_data_dir, exist_ok=True)

# Fonction pour catégoriser la pression artérielle
def categorize_blood_pressure(systolic, diastolic):
    if systolic < 120 and diastolic < 80:
        return "Normal"
    elif 120 <= systolic <= 129 and diastolic < 80:
        return "Elevated"
    elif 130 <= systolic <= 139 or 80 <= diastolic <= 89:
        return "Hypertension Stage 1"
    elif systolic >= 140 or diastolic >= 90:
        return "Hypertension Stage 2"
    elif systolic > 180 or diastolic > 120:
        return "Hypertensive Crisis"
    elif systolic > 180 and diastolic > 120:
        return "Hypertensive Crisis"
    return "Uncategorized"

# Fonction pour traiter une observation
def process_observation(observation_json):
    try:
        observation = json.loads(observation_json)
        if observation.get("resourceType") == "Observation" and "component" in observation:
            patient_id = observation["subject"]["reference"].split("/")[-1]
            systolic = None
            diastolic = None

            for component in observation["component"]:
                code = component["code"]["coding"][0]["code"]
                value = component["valueQuantity"]["value"]
                if code == "8480-6":  # Systolic blood pressure
                    systolic = value
                elif code == "8462-4":  # Diastolic blood pressure
                    diastolic = value

            if systolic is not None and diastolic is not None:
                category = categorize_blood_pressure(systolic, diastolic)
                if category == "Normal":
                    file_path = os.path.join(normal_data_dir, f"{patient_id}.json")
                    with open(file_path, "a") as f:
                        f.write(json.dumps(observation) + "\n")
                    return None
                return {
                    "patient_id": patient_id,
                    "systolic_pressure": systolic,
                    "diastolic_pressure": diastolic,
                    "anomaly_type": category,
                    "observation_id": observation["id"]
                }
    except Exception as e:
        logging.error(f"Erreur de traitement : {e}")
    return None

# Fonction pour envoyer les anomalies à Elasticsearch
def send_to_elasticsearch(data):
    try:
        es.index(index=es_index, body=data)
        logging.info(f"Anomalie envoyée à Elasticsearch : {data}")
    except Exception as e:
        logging.error(f"Erreur d'envoi à Elasticsearch : {e}")

# Consommateur Kafka
consumer = Consumer(kafka_conf)
consumer.subscribe(["fhir_observations"])
```

```python
try:
    logging.info("Démarrage du détecteur d'anomalies.")
    while True:
        msg = consumer.poll(1.0)
        if msg is None:
            continue
        if msg.error():
            if msg.error().code() == KafkaError._PARTITION_EOF:
                logging.info(f"Fin de la partition : {msg.partition()}")
            else:
                raise KafkaException(msg.error())
        else:
            observation_json = msg.value().decode('utf-8')
            anomaly = process_observation(observation_json)
            if anomaly:
                logging.info(f"Anomalie détectée : {anomaly}")
                send_to_elasticsearch(anomaly)

except KeyboardInterrupt:
    logging.info("Arrêt du détecteur d'anomalies.")
finally:
    consumer.close()
    logging.info("Détecteur d'anomalies arrêté.")
```

ANNEXE 4:

```yaml
services:
  producer:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: producer
    command: ["python", "producer.py"]
    environment:
      KAFKA_BOOTSTRAP_SERVERS: kafka1:29092  # Connexion à Kafka
    networks:
      - kafka_net_kafka_nifi  # Réseau de Kafka

  consumer:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: consumer
    command: ["python", "consumer.py"]
    environment:
      KAFKA_BOOTSTRAP_SERVERS: kafka1:29092  # Connexion à Kafka
    networks:
      - kafka_net_kafka_nifi  # Réseau de Kafka

  anomaly-detector:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: anomaly-detector
    command: ["python", "anomalies_traitement.py"]
    environment:
      KAFKA_BOOTSTRAP_SERVERS: kafka1:29092  # Connexion à Kafka
      ELASTICSEARCH_HOST: elasticsearch:9200  # Connexion à Elasticsearch
    volumes:
      - ./normal_data:/app/normal_data
    networks:
      - kafka_net_kafka_nifi  # Réseau de Kafka
      - elasticsearch_net_es  # Réseau d'Elasticsearch

networks:
  kafka_net_kafka_nifi:
    external: true  # Utilise le réseau Docker existant pour Kafka
  elasticsearch_net_es:
    external: true  # Utilise le réseau Docker existant pour Elasticsearch
```

ANNEXE 5:

```
# Utiliser une image de base Python officielle
FROM python:3.10-slim
# Définir le répertoire de travail dans le conteneur
WORKDIR /app

# Copier les fichiers nécessaires dans le conteneur
COPY producer.py consumer.py anomalies_traitement.py requirements.txt ./

# Installer les dépendances Python
RUN pip install --no-cache-dir -r requirements.txt
```