

# Bitcoin DATA IA

OUALID CHEIKH  
THOMAS DOAN  
ABDERAHIM CHEMMOU

---



# Sommaire



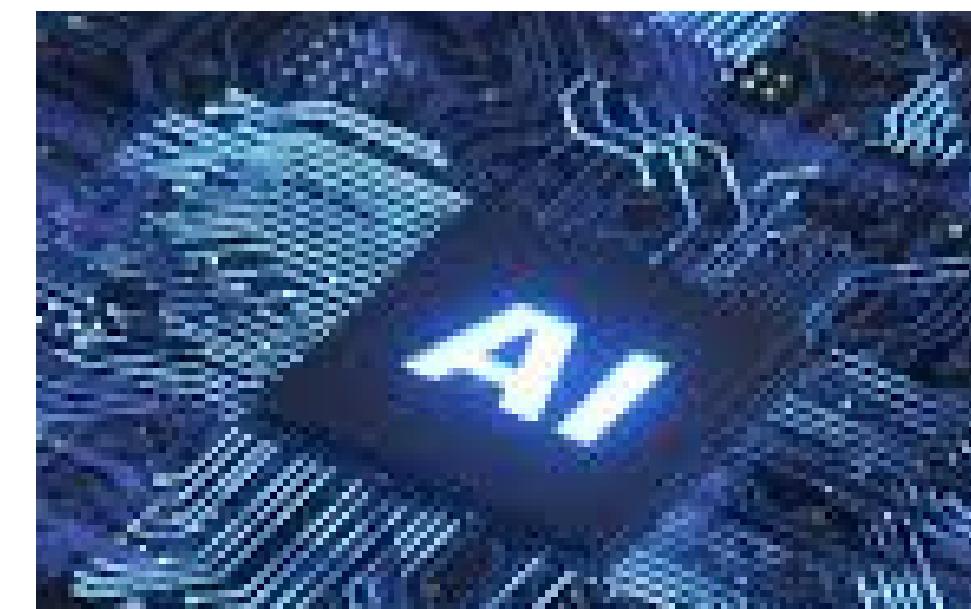
## Étape 1

Fetch la data à partir d'API



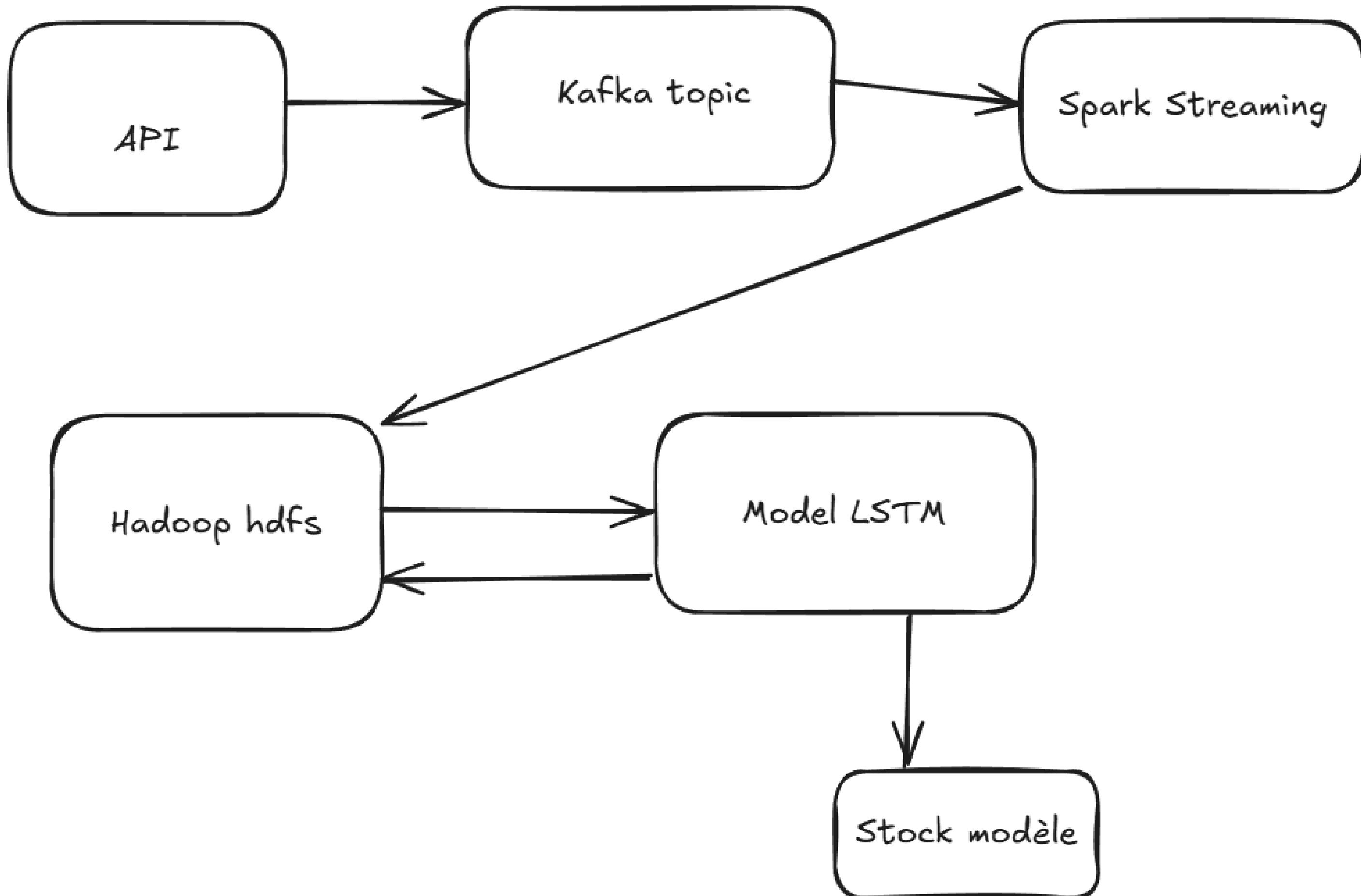
## Étape 2

Transformer et nettoyer la data



## Étape 3

Entraîner une IA à partir des données nettoyées en temps réel





# Etape 1

---

Fetch la data à partir d'API

**Cryptocurrency Prices by Market Cap**

The global cryptocurrency market cap today is \$3.34 Trillion, a ▼ 0.1% change in the last 24 hours. [Read more](#)

Rank	Coin	Price	1h	24h	7d	24h Volume
1	Bitcoin BTC	\$96,591.32	▼ 0.1%	▲ 0.6%	▼ 1.4%	\$29,108,206,016
2	Ethereum ETH	\$2,689.16	▲ 0.3%	▲ 1.0%	▼ 2.6%	\$16,809,957,672
3	XRP XRP	\$2.70	▲ 0.2%	▲ 10.6%	▲ 11.7%	\$6,956,338,870

**Aperçu du marché** Données de trading Occasion Déblocage de tokens

Nom	Prix	24h	Variation	Volume 24 h	Capitalisation
BTC Bitcoin	€92,253.41	+0.66%	+0.66%	€28.93B	€1,915.74B
ETH Ethereum	€2,569.50	+0.98%	+0.98%	€16.77B	€323.62B
XRP XRP	€2.58	+10.68%	+10.68%	€7.70B	€156.13B

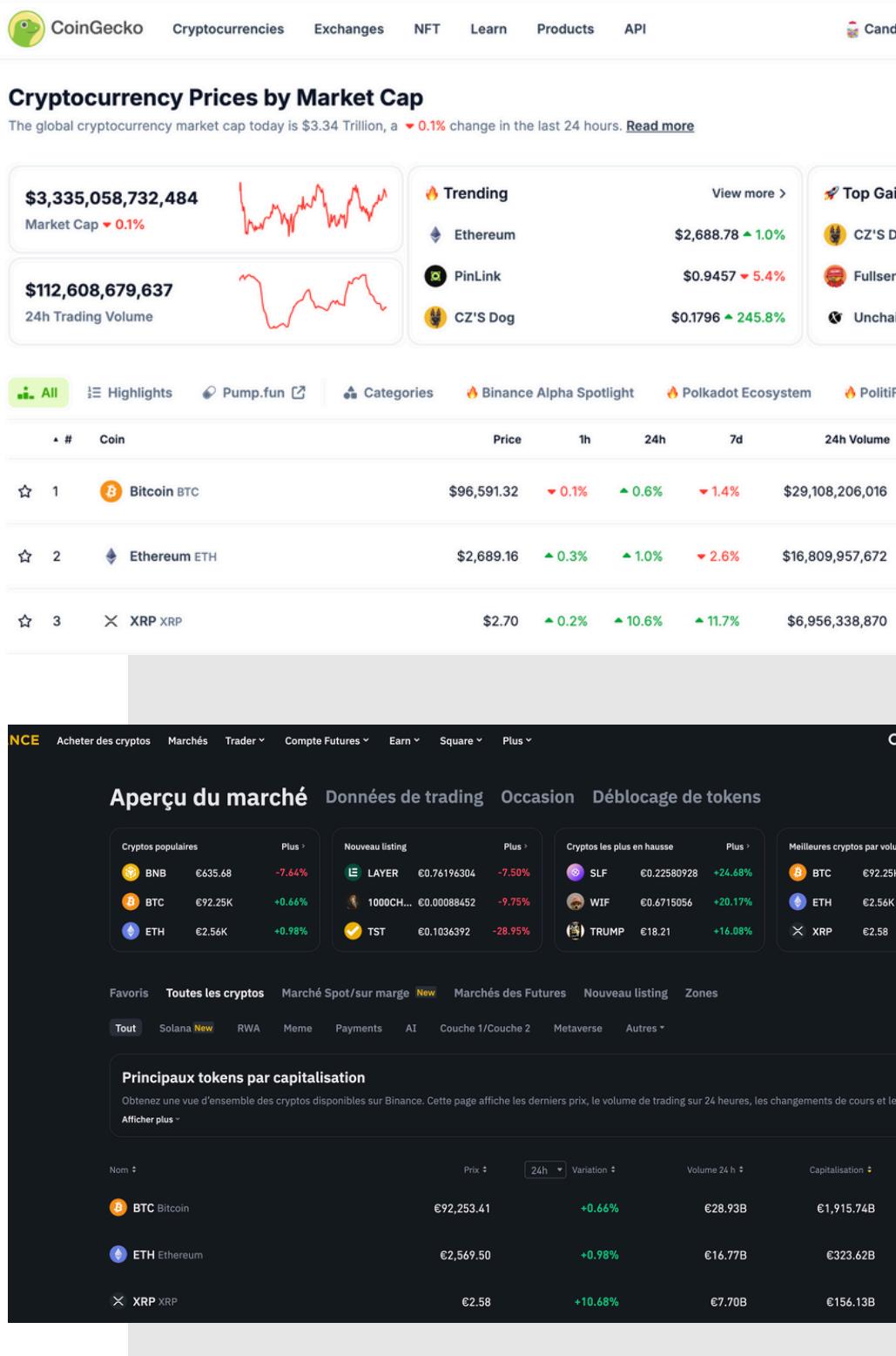
# Processus fonctionnel

Percevoir un flux de donnée en temps réel

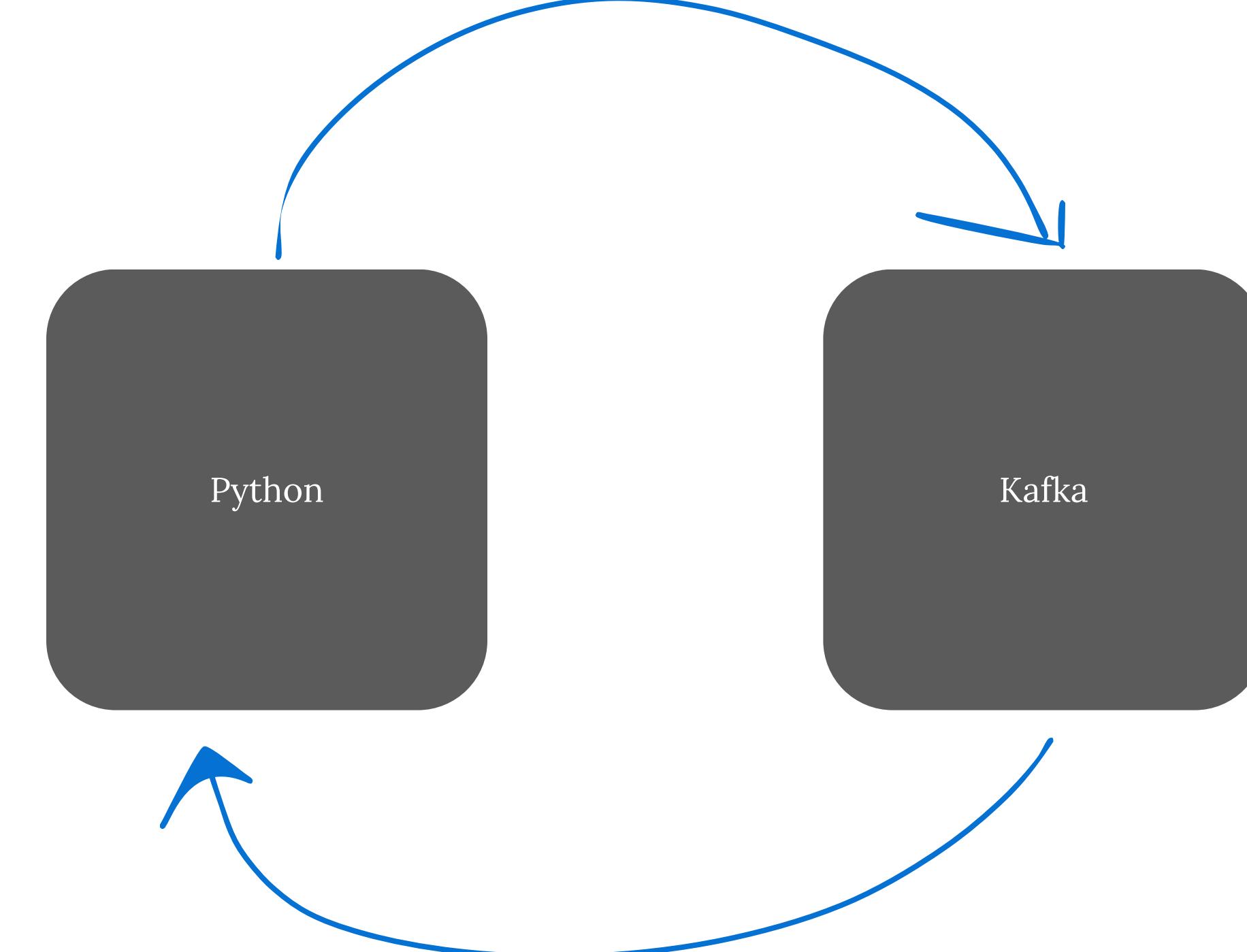
Récupérer la donnée de plusieurs API :  
CoinGecko & Binance

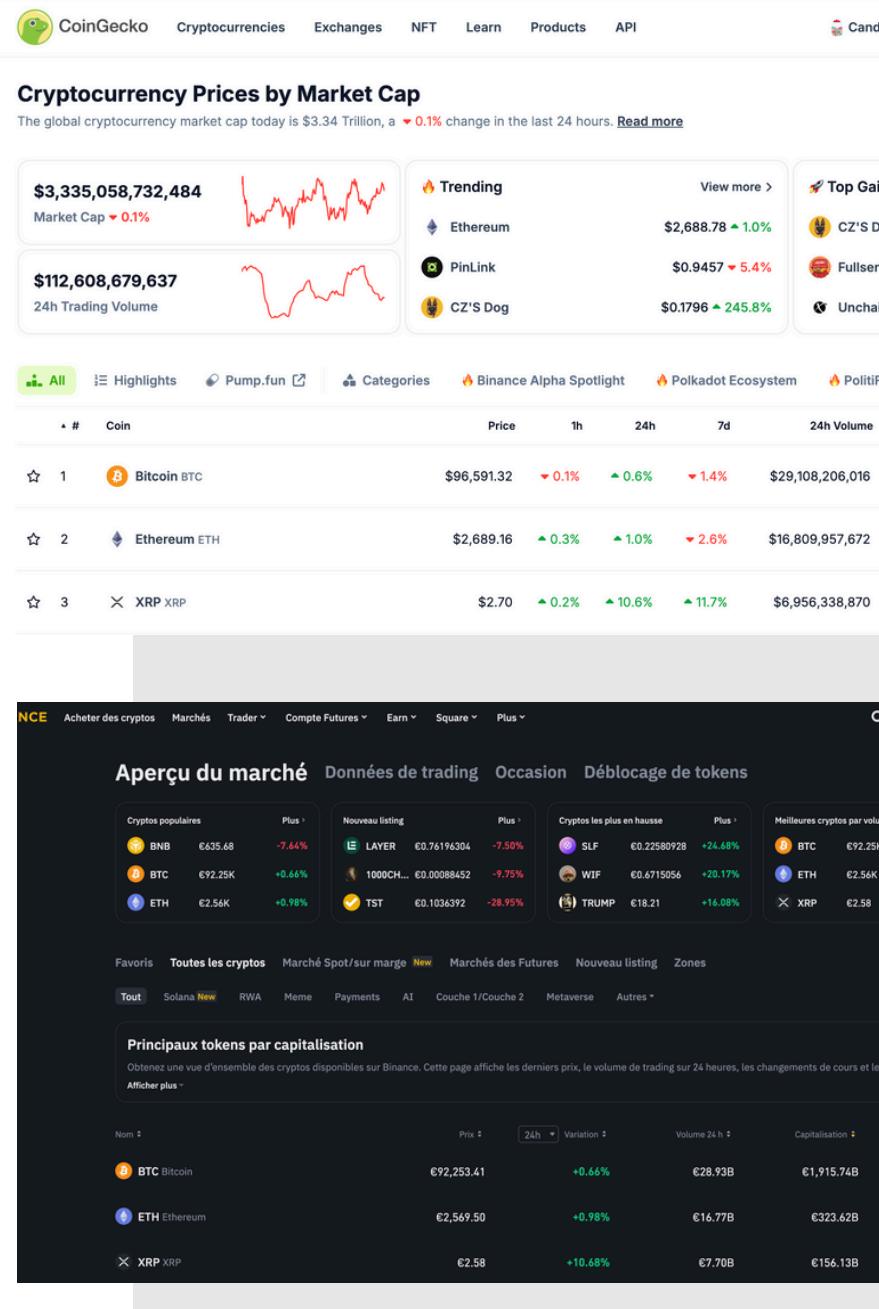
Aggrégation

Envoyer cette donnée semi structuré sur un topic en utilisant Kafka



# Technologie





# Récupérer la data des API

Collecte toute les 60 secondes

CoinGecko API :  
<https://api.coingecko.com/api/v3coins/markets>

retour :

```
'vs_currency': 'usd',
'ids': 'bitcoin',
'order': 'market_cap_desc',
'per_page': 1,
'page': 1,
'sparkline': False
```

Binance API :  
<https://api.binance.com/api/v3/ticker/bookTicker>

retour :

```
'volume_24h',
'market_cap',
'price_change_24h',
'price_change_percentage_24h',
'high_24h',
'low_24h',
```

## Aggrégation de donnée

2025-02-14 14:41:16 INFO:\_\_main\_\_:Données récupérées avec succès:  
{'timestamp': '2025-02-14T13:41:16.832521', 'price\_usd': 96838, 'volume\_24h': 25661166064, 'market\_cap': 1921603007346, 'price\_change\_24h': 781.11, 'price\_change\_percentage\_24h': 0.81317, 'high\_24h': 97231, 'low\_24h': 95410, 'trade\_timestamp': 1739540476}

Topic



## Etape 2

---

Transformer et nettoyer la data

# Spark & Prétraitement des Données

**Spark est un framework de traitement distribué permettant de manipuler de grandes quantités de données en mémoire.**

**Dans notre projet, Spark est utilisé pour :**

**✓ Charger les données Bitcoin stockées dans HDFS.**

**✓ Les nettoyer et les formater avant de les utiliser pour le modèle LSTM.**

**✓ Préparer les données pour l'apprentissage machine.**

**✓ Consomme les données depuis Kafka**

**✓ Effectue des transformations en temps réel :**

**✓ Calcul du spread (différence prix achat/vente)**

**✓ Prix moyen du marché**

**✓ Liquidité totale**

**✓ Volatilité des prix**

**✓ Stocke les données dans HDFS :**

**✓ Données brutes enrichies**

**✓ Métriques agrégées par heure**

✓ Initialise Spark et le configure pour lire depuis Kafka (via le package Kafka-Spark).

✓ C'est le point d'entrée pour toutes les opérations de traitement des données.

```
def create_spark_session():
    return SparkSession.builder \
        .appName("BitcoinDataProcessing") \
        .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0") \
        .getOrCreate()
```

- ✓ Vérifie si HDFS est disponible avant d'écrire des données.
- ✓ Crée les répertoires nécessaires pour stocker les données Bitcoin.

```
def wait_for_namenode(spark):  
    fs=spark._jvm.org.apache.hadoop.fs.FileSystem.get(spark._jsc.hadoopConfiguration())  
    paths = ['/bitcoin/raw_data', '/bitcoin/hourly_metrics', '/bitcoin/checkpoints']  
    for path in paths:  
        hdfs_path = spark._jvm.Path(path)  
        if not fs.exists(hdfs_path):  
            fs.mkdirs(hdfs_path)
```

- ✓ Lit les messages en temps réel depuis Kafka.
- ✓ Se connecte au topic cryptoTopic pour récupérer les données Bitcoin.

```
df_stream = spark \  
    .readStream \  
    .format("kafka") \  
    .option("kafka.bootstrap.servers", "kafka:9092") \  
    .option("subscribe", "cryptoTopic") \  
    .option("startingOffsets", "latest") \  
    .load()
```

- ✓ Calcule des indicateurs financiers importants (spread, mid-price, volatilité, liquidité).
- ✓ Transforme les timestamps en formats lisibles (date et heure).

```
transformed_df = parsed_df \
    .withColumn("timestamp", to_timestamp("timestamp")) \
    .withColumn("date", to_date("timestamp")) \
    .withColumn("hour", hour("timestamp")) \
    .withColumn("spread", col("ask_price") - col("bid_price")) \
    .withColumn("mid_price", (col("ask_price") + col("bid_price")) / 2) \
    .withColumn("spread_percentage", (col("spread") / col("mid_price")) * 100) \
    .withColumn("total_liquidity", col("bid_qty") + col("ask_qty")) \
    .withColumn("price_volatility", (col("high_24h") - col("low_24h")) / col("mid_price") *100)
```

- ✓ Stocke les données Bitcoin en continu dans HDFS sous format Parquet.
- ✓ Utilise un checkpoint pour éviter la duplication en cas de redémarrage.

```
query_raw = transformed_df \  
.writeStream \  
.outputMode("append") \  
.format("parquet") \  
.option("path", "hdfs://namenode:9000/bitcoin/raw_data") \  
.option("checkpointLocation", "hdfs://namenode:9000/bitcoin/checkpoints/raw") \  
.start()
```

- ✓ Regroupe les données par heure pour calculer des statistiques utiles.
- ✓ Sauvegarde ces indicateurs pour analyse ultérieure avec Spark ou un modèle

```
def process_batch(df, epoch_id):  
    hourly_metrics = df \  
        .groupBy("date", "hour") \  
        .agg(  
            avg("mid_price").alias("avg_price"),  
            avg("spread").alias("avg_spread"),  
            avg("spread_percentage").alias("avg_spread_percentage"),  
            sum("total_liquidity").alias("total_liquidity"),  
            avg("price_volatility").alias("avg_volatility"),  
            max("mid_price").alias("high_price"),  
            min("mid_price").alias("low_price"),  
            avg("volume_24h").alias("avg_volume"),  
            last("market_cap").alias("last_market_cap")  
        )  
    hourly_metrics.write \  
        .mode("append") \  
        .parquet("hdfs://namenode:9000/bitcoin/hourly_metrics")
```

- ✓ Regroupe les données par heure pour calculer des statistiques utiles.
- ✓ Sauvegarde ces indicateurs pour analyse ultérieure avec Spark ou un modèle

```
query_metrics = transformed_df \
    .writeStream \
    .trigger(processingTime='1 minute') \
    .foreachBatch(process_batch) \
    .start()
```



## Etape 3

---

Entrainer une IA à partir des données nettoyées en temps réel

# LSTM (Prédiction des prix Bitcoin)

Dans notre projet, LSTM est utilisé pour :

- ✓ Apprendre à prédire le prix futur du Bitcoin en utilisant l'historique des prix.
- ✓ Capturer les tendances et fluctuations du marché en analysant les 50 dernières valeurs.
- ✓ Produire des prédictions en temps réel pour anticiper les évolutions du marché.

## Points clés :

- Chargement des données depuis HDFS en Parquet.
- Préparation des données :
- Sélection des features pertinents.
- Normalisation des données.
- Création des séquences temporelles pour l'entraînement du modèle LSTM.
- Entraînement du modèle LSTM avec des séquences de données.
- Environnement Docker pour gérer l'exécution du code en toute autonomie, avec les bonnes configurations Spark et Python.

## *Construction du modèle LSTM*

- Préparation des données : Transformer les données en séquences avec la méthode prepare\_data et les normaliser.
- Construction du modèle LSTM : Utilisation de Sequential avec des couches LSTM, et ajout de régularisation avec Dropout.
- Entraînement du modèle : Entraînement du modèle avec les données d'entraînement sur plusieurs epochs.
- Prédictions : Effectuer des prédictions sur de nouvelles données avec le modèle entraîné.
- Évaluation des performances : Comparer les résultats réels et prédits avec une visualisation graphique.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler
import numpy as np

class BitcoinPredictionModel:
    def __init__(self, sequence_length=60):
        self.sequence_length = sequence_length
        self.scaler = MinMaxScaler(feature_range=(0, 1))

    def build_model(self, input_shape):
        # Construction du modèle LSTM
        model = Sequential()
        model.add(LSTM(units=50, return_sequences=True, input_shape=input_shape))
        model.add(Dropout(0.2))
        model.add(LSTM(units=50, return_sequences=False))
        model.add(Dropout(0.2))
        model.add(Dense(units=1)) # Prédiction d'un seul prix à chaque fois
        model.compile(optimizer='adam', loss='mean_squared_error')

        return model

    def train_model(self, X_train, y_train, X_test, y_test, batch_size=32, epochs=50):
        # Entraînement du modèle LSTM
        model = self.build_model((X_train.shape[1], X_train.shape[2]))

        # Entraînement du modèle
        model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_test, y_test))

        return model
```

## *Préparation des données pour l'entraînement LSTM*

- Normalisation des données : Utilisation d'un scaler (`self.scaler.fit_transform`) pour mettre à l'échelle les features avant de les envoyer dans le modèle LSTM.
- Création de séquences pour LSTM : Génération de séquences de données pour prédire `avg_price` (prix moyen) sur une fenêtre de taille `sequence_length`.
- Séparation des données en ensembles d'entraînement et de test : 80% pour l'entraînement et 20% pour les tests.

```
def prepare_data(self, df):  
    """Prépare les données pour l'entraînement"""  
    try:  
        features = ['avg_price', 'avg_spread', 'total_liquidity', 'avg_volatility', 'avg_volume']  
  
        # Vérification des valeurs manquantes  
        null_counts = df[features].isnull().sum() if null_counts.any(): df = df.dropna(subset=features)  
  
        # Mise à l'échelle des données  
        scaled_data = self.scaler.fit_transform(df[features])  
  
        # Création des séquences pour l'entraînement LSTM  
        X, y, timestamps = [], [], []  
        for i in range(len(scaled_data) - self.sequence_length):  
            X.append(scaled_data[i:(i + self.sequence_length)])  
            y.append(scaled_data[i + self.sequence_length, 0]) # Cible : 'avg_price'  
            timestamps.append(df['datetime'].iloc[i + self.sequence_length])  
        X = np.array(X) y = np.array(y)  
        # Division en ensembles d'entraînement et de test  
        train_size = int(len(X) * 0.8)  
        X_train, X_test = X[:train_size], X[train_size:]  
        y_train, y_test = y[:train_size], y[train_size:]  
        timestamps_test = timestamps[train_size:]  
        return (X_train, y_train), (X_test, y_test, timestamps_test), features  
    except Exception as e: logger.error(f"Erreur lors de la préparation des données: {str(e)}")  
    raise
```

## Définition de l'environnement Docker pour l'entraînement

- Dockerfile : Construction d'une image Docker pour exécuter le code Python.
- Environnement Python : Utilisation de Python3 avec PySpark pour gérer les traitements de données.
- Dépendance avec le NameNode : Assure que le conteneur dépend du NameNode pour pouvoir accéder aux données HDFS.
- Options de Spark : Paramétrage des options mémoire pour les tâches Spark.
- Exécution de prediction.py : Exécution du script de prédiction avec les modèles LSTM.

```
query_raw = transformed_df \
.writeStream \
.outputMode("append") \
.format("parquet") \
.option("path", "hdfs://namenode:9000/bitcoin/raw_data") \
.option("checkpointLocation", "hdfs://namenode:9000/bitcoin/checkpoints/raw") \
.start()
```



Merci

---