CryptoStream - Traitement Fonctionnel de Données Crypto

Source de Données

API Binance

Nous avons choisi l'API publique de Binance pour récupérer les données de marché crypto :

- URL : https://api.binance.com/api/v3/klines
- Données : Prix OHLC (Open, High, Low, Close), volume, nombre de trades
- Fréquences : 1s, 1m, 15m, 1h, 1d
- Cryptos surveillées : BTC, ETH, XRP, SOL (contre USDC)

Avantages de cette source

- Données en temps réel : Flux continu et fiable
- Richesse des données : Métriques complètes pour l'analyse technique
- Haute fréquence : Permet de tester les performances du pipeline
- Gratuité : Pas de limitation pour ce volume de données

Architecture Fonctionnelle

Vue d'ensemble

L'architecture suit un modèle de pipeline fonctionnel basé sur la séparation des responsabilités :
API Crypto → Producers → Kafka → Consumers → InfluxDB → Grafana

Composants principaux

Producers (Producteurs)

- Récupèrent les données de prix depuis une API crypto
- Appliquent des transformations pures sur les données
- Publient les messages dans Kafka

Consumers (Consommateurs)

- Lisent les flux Kafka
- Enrichissent et transforment les données via des fonctions pures
- Persistent le résultat dans InfluxDB (données brutes et données enrichies)

Kafka

- Centralise la distribution des flux de données
- Découple les producteurs des consommateurs
- Garantit la résilience et la scalabilité

InfluxDB

- Base de données orientée séries temporelles
- Optimisée pour les métriques financières et streaming
- Stockage des données enrichies

Choix Techniques

Composant	**Choix**	**Justification**
Kafka	Confluent Platform	Écosystème complet avec interface de monitoring
InfluxDB v2	Time-Series DB	Adaptée aux données OHLC et métriques financières
Python	Langage principal	Riche écosystème pour le traitement de données
Docker	Conteneurisation	Isolation des services et reproductibilité

Paradigme Fonctionnel Appliqué

1. **Immutabilité des Données**
    - Les messages Kafka ne sont jamais modifiés après publication
    - Les transformations créent de nouvelles structures de données
    - Pipeline : RAW → ENRICHED → PERSISTED
2. **Fonctions Pures**
    - Toutes les transformations de données sont effectuées via des fonctions pures
3. **Composition de Fonctions**

    - Le traitement suit une logique de pipeline fonctionnel : récupération → transformation → enrichissement → persistance

4. **Absence d'Effets de Bord**
    - Séparation claire entre fonctions pures et fonctions d'effet (I/O)
    - Logique métier isolée des opérations d'entrée/sortie
    - Testabilité accrue
    - Prédictibilité des résultats

Traitements Effectués

1. Nettoyage et validation des données
    - Validation des formats reçus
    - Conversion sécurisée des types
    - Gestion des valeurs aberrantes ou nulles
2. Enrichissement des données
    - Calcul d'indicateurs techniques (ex. moyenne mobile simple)
    - Ajout de métadonnées temporelles
    - Enrichissement via calculs purs
3. Agrégation multi-intervalles
    - Traitement parallèle sur différentes fréquences
    - Synchronisation des données entre intervalles
    - Calculs statistiques sur fenêtres glissantes
4. Persistance structurée
    - Séparation des données brutes et enrichies
    - Stockage optimisé pour séries temporelles
    - Indexation par crypto, intervalle et timestamp
5. Monitoring et logs
    - Logging structuré au format JSON Lines
    - Mesure des performances : latence, débit

- Alertes en cas d'erreurs

Fonctionnement des Applications via Docker Compose

Pour orchestrer l'ensemble des services de notre pipeline de traitement de données crypto, nous utilisons **Docker Compose**. Docker est une plateforme qui permet d'exécuter des applications dans des **containers** isolés, garantissant ainsi qu'elles fonctionnent de manière cohérente, indépendamment de l'environnement sous-jacent. Cela évite notamment les conflits de versions de bibliothèques ou de dépendances entre différents services.

Docker Compose est un outil complémentaire qui permet de **décrire et configurer plusieurs containers dans un seul fichier YAML**. Ce fichier définit les services, les réseaux, les volumes, les variables d'environnement, et facilite le lancement simultané de tous les composants de l'application.

Structure du Docker Compose dans CryptoStream

Notre fichier Docker Compose est organisé de manière claire et modulaire :

- **Templates de services** : Nous définissons en haut du fichier deux templates (modèles) réutilisables, un pour les producteurs (producers) et un autre pour les consommateurs (consumers).
  Ces templates contiennent toute la configuration commune — build, volumes, dépendances, ressources, réseau, commande de lancement — et sont utilisés comme base pour créer différentes instances de producteurs ou consommateurs, avec des paramètres spécifiques (variables d'environnement).
- **Services d'infrastructure** :
  Ici se trouvent les composants essentiels au fonctionnement du pipeline :
  - **Zookeeper** et **Kafka** pour la gestion et le streaming des messages
  - **InfluxDB** comme base de données spécialisée pour les séries temporelles
  - **Control Center Kafka** et **Grafana** pour le monitoring et la visualisation
- **Services applicatifs** :
  Ces services correspondent aux producteurs et consommateurs, chacun dédié à un intervalle de temps (1s, 1m, 15m, 1h, 1d). Chaque service est une instance distincte utilisant le template approprié, avec des variables d'environnement spécifiques comme l'intervalle de traitement.

Lisibilité, Réutilisabilité et Robustesse du Code

- **Un seul fichier de code pour tous les producteurs, un autre pour tous les consommateurs** :
  Toute la logique métier des producteurs est centralisée dans un unique fichier producer.py ; de même pour les consommateurs avec consumer.py.
  Cette approche garantit une grande **lisibilité** et une **maintenabilité simplifiée** : il n'y a pas de duplication du code. Toute modification ou amélioration dans la logique métier est faite en un seul endroit.

- **Réutilisation via Docker Compose** :
  Pour créer une nouvelle instance d'un producteur ou consommateur (par exemple pour un nouvel intervalle de temps), il suffit de **copier-coller un bloc de service dans le fichier Docker Compose**, puis de modifier les variables d'environnement correspondantes (comme INTERVAL ou CONSUMER_ID).
  Cela rend la solution **extrêmement modulaire et extensible**.

- **Robustesse et Scalabilité** :
  Grâce à Docker Compose et aux containers isolés, chaque instance s'exécute indépendamment, permettant de répartir la charge sur plusieurs ressources machine ou serveurs.
  Le système peut donc être **scalé horizontalement** en lançant plusieurs producteurs ou consommateurs selon les besoins, sans impacter la stabilité des autres services.

- **Gestion des ressources** :
  Le fichier Docker Compose spécifie des limites de mémoire et CPU pour chaque service, garantissant que les containers n'utilisent pas plus de ressources que nécessaire, contribuant à la stabilité globale du système.

En résumé

Grâce à Docker Compose :

- Nous garantissons une **installation simple et reproductible** de l'ensemble des composants
- Nous maintenons un **code propre, modulaire et réutilisable**
- Nous pouvons facilement **scaler** le pipeline en multipliant les instances pour couvrir différents intervalles ou cryptomonnaies
- L'isolation des containers assure la **robustesse** et la prévention des conflits

Cette architecture permet de déployer rapidement, d'adapter la solution à de nouvelles exigences, et de monitorer efficacement la santé du système.

Installation et Démarrage

Prérequis

- Création du réseau Docker
- Préparation des secrets InfluxDB (utilisateur, mot de passe, token)

Lancement

- Démarrage des services via Docker Compose

Accès aux interfaces

- Confluent Control Center : http://localhost:9021
- InfluxDB UI : http://localhost:8086

Structure du Code

Producer (producer.py)

- Fonctions pures pour la récupération et transformation des données
- Fonctions d'effet pour la connexion à Kafka et l'envoi des messages
- Pipeline fonctionnel orchestrant les opérations

Consumer (consumer.py)

- Fonctions pures pour le parsing et enrichissement des messages
- Fonctions d'effet pour la connexion Kafka et écriture dans InfluxDB
- Pipeline fonctionnel pour le traitement des messages

Architecture Docker

- Services isolés en containers
- Gestion des ressources (CPU, mémoire)
- Sécurisation via Docker secrets
- Réseaux privés pour communication interne

Sécurité

- Utilisation des Docker secrets pour les credentials
- Pas de mots de passe codés en dur
- Kafka accessible uniquement en interne

Démonstration des Concepts Fonctionnels

- Utilisation des fonctions Map, Filter, Reduce pour la gestion des flux
- Fonctions de haut niveau retournant des closures pour le traitement contextuel
- Currying et application partielle pour la configuration dynamique
- Gestion fonctionnelle des erreurs avec des monads et chaînage sécurisé

Bonnes Pratiques

Programmation Fonctionnelle

- Séparation claire des responsabilités
- Usage exclusif de fonctions pures pour la logique métier
- Immutabilité des données
- Composition et réutilisabilité des fonctions

Architecture

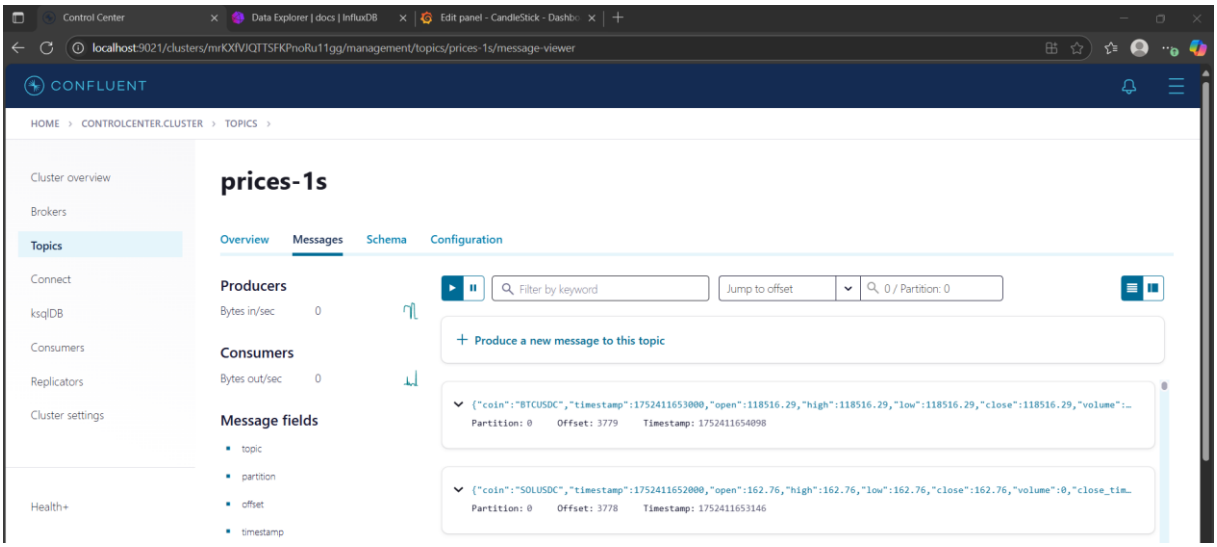
- Basée sur les événements (event-driven)
- Modularité et isolation des composants
- Observabilité via monitoring intégré
- Résilience et gestion des pannes

Cas d'Usage

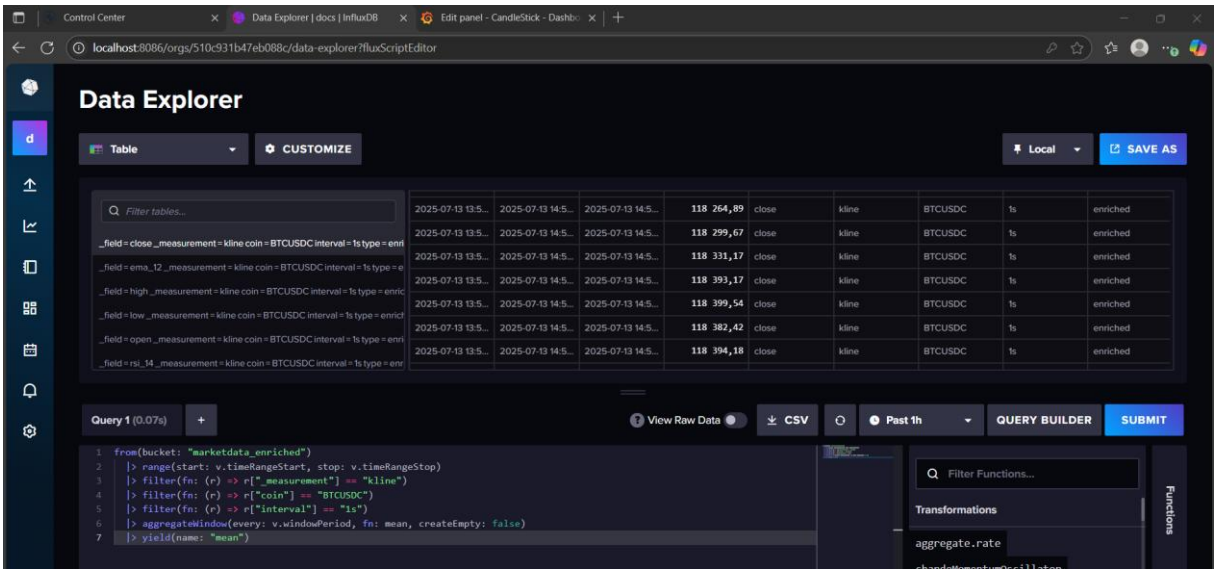
- Analyse technique en temps réel
- Backtesting de stratégies sur données historiques
- Surveillance des marchés crypto
- Alertes sur seuils de prix ou volume
- Recherche sur les patterns de marché

Captures d'Écran et Visualisations

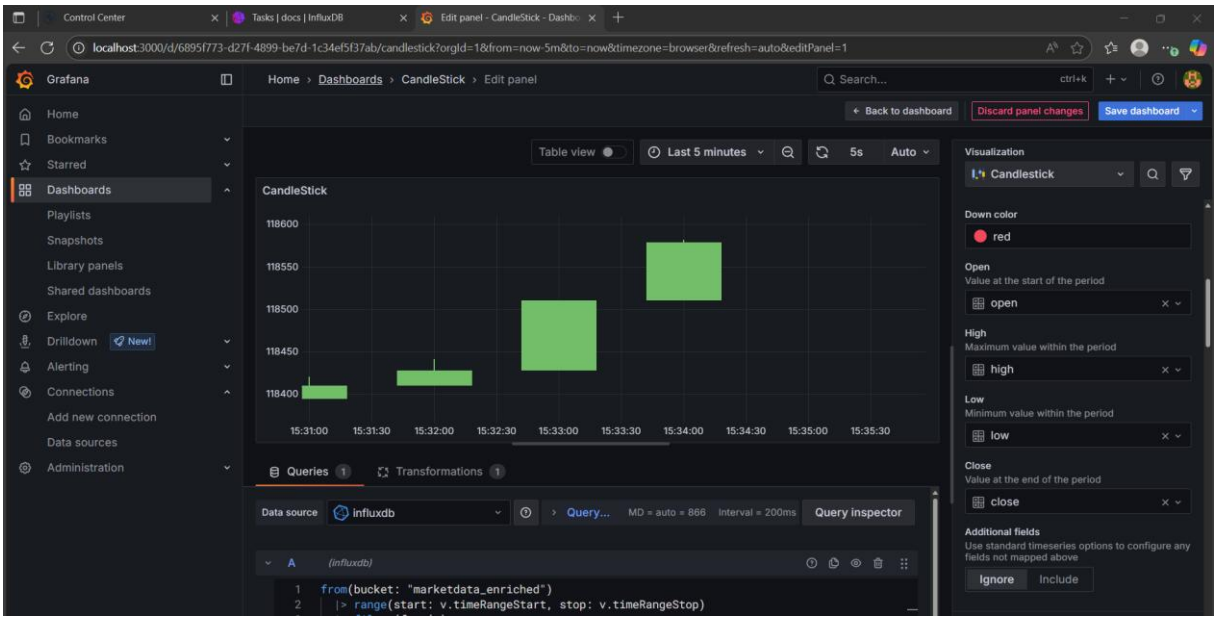
Confluent Control Center (messages kafka)



InfluxDB Dashboard (stockage)



Grafana (DataViz)



Architecture Visuelle

API Binance → Producers → Kafka Topics → Consumers → InfluxDB → Grafana
(Pure Funcs, Streaming Events, Time-Series Storage)

Conclusion

CryptoStream illustre l'application pratique de la programmation fonctionnelle dans un contexte de traitement de données en temps réel. Ce projet combine principes fonctionnels, performances élevées et architecture robuste pour le traitement de flux financiers.