RAPPORT

SYSTÈMES DISTRIBUÉS POUR LE TRAITEMENT DES DONNÉES

Tao BEAUFILS

Mouahé KOUAKOU

Arthur LENNE

Alex SOUBEYRAND

Samuel VANIE

Caroline WANG

Le Projet

Rappel du Projet

L'objectif principal de notre projet est de mettre en œuvre un système distribué de traitement de données pour démontrer sa capacité à analyser et traiter de grandes quantités d'informations.

Dans notre cas d'usage, nous nous concentrons sur l'analyse des données financières, en particulier les valeurs des actions. Voici les principales étapes du projet :

1. **Récupération des données brutes** :
   Nous collectons les valeurs d'un grand nombre d'actions (symbole) provenant de sources variées.
2. **Traitement des données** :
   Ces données sont analysées pour effectuer des calculs complexes, tels que le RSI (*Relative Strength Index*), et d'autres indicateurs techniques utiles pour l'analyse financière.
3. **Stockage des résultats** :
   Les résultats traités sont ensuite centralisés et stockés dans une base de données, permettant une exploitation future et une visualisation structurée.

Ce projet met en évidence les capacités d'un système distribué à traiter des volumes importants de données de manière performante et fiable, en tirant parti des technologies modernes telles que Kafka, Kubernetes, et des bases de données distribuées.

Conception et implémentation de l'architecture logicielle

Rôle de Développeur

Dans le cadre du déploiement de l'application de traitement à grande échelle, le rôle du développeur est essentiel pour concevoir une architecture logicielle robuste, évolutive et adaptée au scaling des applications. Ce rôle implique notamment la conception des pipelines de traitement de données, l'optimisation des modules logiciels pour garantir des performances à grande échelle, et l'intégration des solutions nécessaires pour assurer une mise à l'échelle fluide et automatique. Le développeur est également responsable de l'utilisation de stratégies de conteneurisation et de l'orchestration des microservices pour maximiser la fiabilité et la maintenabilité de l'ensemble du système.

Description de l'architecture

L'architecture de l'application est organisée autour de plusieurs composants interconnectés, chacun jouant un rôle spécifique dans le pipeline de traitement des données financières.

Le **Manager**, qui agit comme un producteur, génère des tâches sous forme d'actions contenant le symbole d'une action et une période à traiter. Ces tâches sont ensuite envoyées dans un topic Kafka nommé **`Stock_topic`**, où elles sont réparties entre différentes partitions pour permettre un traitement parallèle et une meilleure scalabilité. Pour garantir la continuité en cas de panne, le Manager utilise une table PostgreSQL **`symbol_state`** pour se souvenir de la dernière date traitée pour chaque symbole. Cette table est mise à jour après chaque traitement pour éviter les doublons et reprendre facilement en cas d'interruption.

Les **Agents**, qui agissent à la fois comme consommateurs et producteurs, récupèrent ces actions depuis le topic **`Stock_topic`**. Pour chaque action, ils utilisent l'API YFinance pour collecter les données journalières correspondantes. Une fois ces données collectées, elles sont envoyées dans un autre topic Kafka nommé **`Stock_data_topic`**, où elles sont organisées pour permettre leur traitement parallèle.

Les **Traiteurs**, qui agissent uniquement comme consommateurs, récupèrent les données brutes depuis le topic **`Stock_data_topic`**. Ils appliquent ensuite des traitements spécifiques, notamment le calcul des indicateurs techniques comme le RSI et la MACD. Les résultats traités sont ensuite stockés dans une base de données **PostgreSQL**, où ils peuvent être exploités ultérieurement à l'aide de requêtes SQL.

La **base de données PostgreSQL** joue un rôle central en tant que stockage pérenne des données financières traitées. Elle permet de conserver les résultats de manière organisée et structurée, tout en offrant des possibilités d'interrogation et d'analyse approfondies.

Choix technologiques

Pour le développement, nous avons sélectionné plusieurs technologies et outils adaptés aux besoins du projet.

En premier lieu, **Python 3** a été choisi comme langage principal pour le développement des microservices. Ce choix s'explique par la richesse de ses bibliothèques, qui sont particulièrement adaptées aux besoins d'analyse et de manipulation de données. Parmi ces bibliothèques, **Pandas** joue un rôle clé pour manipuler et traiter efficacement les données financières, notamment grâce à ses structures de données optimisées comme les DataFrames. Par ailleurs, nous avons utilisé **Pandas-TA**, une bibliothèque spécialisée dans le calcul des indicateurs techniques tels que le RSI (Relative Strength Index), la MACD (Moving Average Convergence Divergence) et d'autres métriques financières.

Pour l'accès aux données boursières, nous avons utilisé **YFinance**, une API fiable qui permet de récupérer facilement l'historique des valeurs boursières. Cet outil offre un accès simplifié à des données structurées et régulièrement mises à jour, ce qui le rend idéal pour répondre aux exigences du projet.

La communication entre les différents microservices, tels que le Manager, les Agents et les Traiteurs, repose sur **Confluent-Kafka**. Kafka a été sélectionné en raison de ses fonctionnalités avancées, notamment sa capacité à gérer efficacement les messages dans un environnement distribué. Il facilite également le découplage des services, garantit une tolérance aux pannes et offre un traitement parallèle des tâches grâce à ses partitions.

Les résultats obtenus, qu'ils s'agissent des données financières brutes ou des indicateurs techniques calculés, sont stockés de manière pérenne dans une base de données relationnelle **PostgreSQL**. Ce choix a été motivé par les capacités robustes de PostgreSQL à organiser et manipuler des données structurées tout en garantissant une gestion fiable des transactions.

Pour garantir une exécution uniforme et simplifier le déploiement des différents services, tous les microservices sont conteneurisés à l'aide de **Docker**. Cette approche permet de s'assurer que les services fonctionnent de manière identique sur tous les

environnements, ce qui améliore considérablement leur portabilité et leur maintenance.

Enfin, l'optimisation des calculs et le traitement des données sont réalisés en respectant une modularité stricte. Les services sont divisés en microservices indépendants, chacun ayant un rôle bien défini dans le pipeline de traitement. Par exemple, le Manager génère les tâches, les Agents collectent les données, et les Traiteurs appliquent les traitements nécessaires avant de stocker les résultats. Cette architecture modulaire, orientée flux (Streaming), permet de traiter les données en temps quasi-réel tout en minimisant la latence entre la collecte, le traitement et le stockage des résultats.

Choix techniques justifiés

Les choix technologiques effectués pour ce projet s'appuient sur plusieurs critères. **Kafka**, par exemple, est utilisé pour assurer une communication fluide et fiable entre les différents services. Grâce à ses partitions, Kafka garantit une répartition efficace des tâches, permettant ainsi une scalabilité horizontale. De plus, ses mécanismes de relecture des messages et de tolérance aux pannes renforcent la fiabilité globale du système.

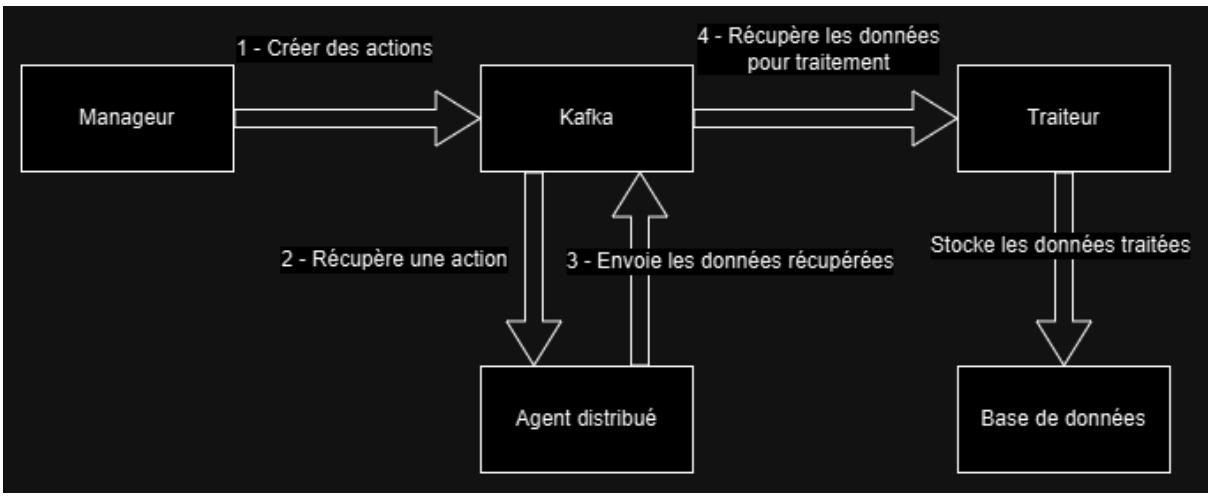
Les Consumer Groups pour les Agents et les Traiteurs permettent de distribuer les tâches entre plusieurs instances, assurant ainsi une gestion dynamique de la charge.

L'utilisation de **YFinance** comme source de données financières offre un accès structuré et fiable à l'historique des actions boursières, simplifiant le processus de collecte des données.

Le choix de **PostgreSQL** comme solution de stockage final repose sur sa capacité à gérer des données structurées tout en offrant des performances optimales pour les requêtes complexes.

Enfin, l'architecture orientée flux (Streaming), combinée à l'utilisation de Kafka et des microservices, garantit un traitement des données en temps quasi-réel avec une latence minimale entre la collecte, le traitement et le stockage des résultats. Cette approche modulaire et scalable garantit également la robustesse et la flexibilité nécessaires pour gérer efficacement des volumes importants de données financières.

Cette architecture garantit la modularité, la scalabilité et la fiabilité nécessaires pour gérer efficacement le traitement des données financières à grande échelle.



Environnement de l'application

Objectifs

- Construire un cluster Kubernetes à l'aide de machines virtuelles GCP.

- Automatiser le déploiement avec Terraform et Ansible.

- Installer des composants critiques pour le projet (Kafka, PostgreSQL, KEDA)

- Garantir une configuration facilement reproductible

Cluster Kubernetes à l'aide de Terraform sur GCP

Pour mettre en place notre cluster kubernetes à l'aide de Terraform sur des machines virtuelles GCP, plusieurs étapes ont été réalisées. Tout d'abord, nous configurons le compte GCP sur la machine locale afin d'interagir avec les ressources cloud. Ensuite, la configuration par défaut des VMs se présente comme suit : type **e2-medium**, avec un **master unique** et **deux workers** (configurable dans le fichier variable.tf des ressources terraform). On génère, par la suite, une clé SSH et la copions sur chaque instance pour

assurer une communication sécurisée entre les machines. Nous avons créé un script d'installation pour déployer les dépendances essentielles et préparer la construction du cluster sur le master et sur les workers. Les dépendances installées incluent **kubeadm** qui génère un token unique permettant aux workers de rejoindre facilement le cluster. Après la configuration complète du master, le script permettant la configuration des workers est exécuté, ce qui permet à ceux-ci d'intégrer le cluster.

Concernant la configuration réseau, toutes les VMs du même réseau local ont un accès illimité de ports entre elles. Pour permettre la communication directe avec ces dernières, votre IP locale doit être rajoutée à la configuration du cluster. Nous effectuons cette dernière opération automatiquement dans notre script de déploiement dont nous parlons un peu plus bas. Enfin, après la création du cluster, nous copions la configuration **.kube/config** depuis le master vers notre machine locale, en remplaçant l'IP privée du master par son adresse publique afin de faciliter les interactions et les installations ultérieures.

Installation de services sur le cluster avec Ansible

Afin de faire tourner notre application sur le cluster kubernetes créé, nous avons utilisé Ansible. Les services installés en plus de ceux nécessaires au fonctionnement de l'application tels que Postgresql, kafka sont :

- **keda :** Pour le scaling horizontal des pods en fonction du nombre de messages dans notre file kafka (configuré à 10 dans notre cas)
- **autoscaling :** Service utilisé pour le scaling horizontal en fonction de la charge de CPU des machines sur lesquels l'application tourne
- **kubeprometheus :** Pour le monitoring global de notre architecture à travers une interface graphique

Automatisation du déploiement

Dans le but de faciliter le déploiement de notre environnement et de notre application, nous avons créé un projet ruby permettant de réaliser toutes les différentes étapes de manière atomique. Il nous a permis de gagner du temps sur le débogage et sur les différentes actions répétitives que nous avons eu à réaliser pendant le projet.

Fonctionnalités du script deploy_architecture.rb

- Installation des dépendances (gcloud, Terraform, Ansible).
- Création du projet GCP et configuration IAM.
- Déploiement Terraform + configuration Kubernetes.
- Nettoyage des ressources.

Workflow typique

./deploy_architecture.rb --install-dependencies # Installer les outils

./deploy_architecture.rb –setup # Configurer GCP sur votre machine

./deploy_architecture.rb –terraform # Déployer l'infrastructure

./deploy_architecture.rb –install_ansible_apps # Installer les services avec Ansible

Environnement du script

Lancer le script sur une machine demande la présence de plusieurs outils qui sont :

- **which :** permet de faire la recherche de binaire dans le path de la machine
- **jq :** Outil permet de manipuler les fichiers json
- **grep :** rechercher une chaîne de caractères dans un ou plusieurs fichiers.
- **ruby :** interpréteur du langage de programmation Ruby

Pour faciliter l'installation de ces outils nous avons créé un fichier flake.nix qui pourra être utilisé avec l'outil de construction d'environnement reproductible nix.

Ainsi, la commande nix develop dans le dossier racine de notre script permet d'installer tous les outils nécessaires.

Supervision, Fiabilité et Performance de l'Application

Rôle du SRE

Dans le cadre du déploiement de l'application de traitement à grande échelle, le rôle du System Reliability Engineer est nécessaire pour garantir la supervision, la fiabilité et la performance continue de l'application. Cette partie aborde les outils mis en œuvre pour collecter les métriques, permettre leurs affichages et garantir une tolérance aux fautes.

Choix technologiques

Une supervision efficace repose sur la récupération, l'analyse en temps réel de mesures clés et l'affichage de ceux-ci. Pour cela, deux technologies principaux ont été choisis :

- **Prometheus** : collecte et stockage en temps réel des valeurs telles que l'utilisation CPU, les latences voire défaillances de l'application.
- **Grafana** : représentation graphique de données statistiques sous forme de tableaux de bord interactifs pour visualiser les données et identifier rapidement les anomalies liées à l'exploitation.

Pour cela, un dépôt préexistant a été utilisé. Il intègre ces deux outils tout en assurant l'interconnexion avec les données au sein du cluster Kubernetes.

Kube-Prometheus est une pile préconfigurée qui combine les capacités de Kubernetes et Prometheus pour la gestion des métriques et de la surveillance. Elle regroupe les outils nécessaires pour surveiller un cluster Kubernetes tout en offrant des dashboards prêts à l'emploi pour une prise en main rapide.

Ce choix a été motivé par les points suivants :

- **Configurations pré-faites** : Kube-Prometheus simplifie l'intégration entre Kubernetes et Prometheus en fournissant des configurations par défaut adaptées à la majorité des besoins.
- **Lien natif entre Kubernetes et Prometheus** : La pile inclut des intégrations étroites permettant de récupérer facilement les métriques des composants Kubernetes et des applications déployées.

L'avantage le plus important inhérent à l'utilisation d'une telle technologie est la disponibilité d'un grand nombre de configurations par défaut. Ces configurations ont permis une mise en place rapide sans nécessiter une connaissance approfondie de Prometheus ou Kubernetes. Cependant, il est utile d'observer que la personnalisation d'un tel outil reste limitée : adapter la pile pour répondre à des besoins spécifiques peut s'avérer complexe.

Cependant, la mise en place de ces choix n'a pas été exempt de difficultés :

- **Configurations** : La modification des configurations par défaut pour répondre aux besoins spécifiques a nécessité une analyse approfondie des fichiers YAML et des dépendances entre les composants.
- **Installations** : L'installation initiale de Kube-Prometheus présente des défis liés à la compatibilité avec l'environnement déployé.

Configurations des dashboards

Dans un premier temps, une personnalisation des dashboards fournis a été réalisée afin qu'ils répondent aux besoins spécifiques de notre projet. Cela inclut l'ajout de métriques critiques et l'optimisation de la lisibilité des données affichées.

De plus, ces dashboards ont été adaptés afin de répondre aux besoins des autres membres de l'équipe (développeurs et développeurs opérateurs). Cela leur a permis un suivi efficace pendant les phases de déploiement.

Efficacité de la surveillance

Au cours de ce projet, différents problèmes ont été rencontrés et résolus grâce à un suivi et une visualisation efficace de l'ensemble des métriques collectées :

- Crashs des actions : Identification et résolution des crashs récurrents lors de l'exécution de certaines actions.
- Goulet d'étranglement : Détection et optimisation des goulots d'étranglement dans le traitement des données. Par exemple : une attente prolongée

inhabituelle de certains nœuds a pu être observée. Cela a permis la correction d'erreurs dans le logiciel.