



RAPPORT DE PROJET

MONITORING D'UNE APPLICATION WEB AVEC PROMETHEUS & GRAFANA

Prepared By :

**Dabo Ali
Coulibaly Admas**



Prepared For:

DR DIOMANDE

06/18/2025

Contents

RÉSUMÉ EXÉCUTIF DU PROJET	3
OBJECTIFS DU PROJET	3
Les objectifs principaux de ce projet sont les suivants :.....	3
CHOIX TECHNIQUES.....	4
ARCHITECTURE DE LA SOLUTION	5
Schéma Général	5
Description Détaillée	7
Configuration des fichiers clés	11
ÉTAPES DE RÉALISATION.....	13
RÉSULTATS OBTENUS	15
DIFFICULTÉS RENCONTRÉES ET SOLUTIONS	16
CONCLUSION ET PERSPECTIVES	17
Perspectives.....	18
Annexes :	18

RÉSUMÉ EXÉCUTIF DU PROJET

Ce projet vise à mettre en place une solution professionnelle de supervision d'une application web à l'aide de Prometheus et Grafana. L'objectif est de permettre la collecte, la visualisation et l'alerte en temps réel sur les métriques applicatives et système, tout en facilitant l'exploitation et la maintenance grâce à des outils modernes et open source. La solution est entièrement dockerisée pour assurer une reproductibilité et une facilité de déploiement maximales.

OBJECTIFS DU PROJET

Les objectifs principaux de ce projet sont les suivants :

- ✚ Mettre en place un monitoring complet d'une application web (Flask) en collectant des métriques applicatives, système et de conteneurs.
- ✚ Visualiser les données en temps réel via des dashboards Grafana dynamiques et personnalisés.
- ✚ Détecter et alerter sur les anomalies de performance ou d'erreur via Alertmanager.
- ✚ Fournir une architecture reproductible et facilement déployable grâce à Docker Compose.
- ✚ Simuler des scénarios de charge et d'erreur pour valider l'efficacité du système de monitoring.

CHOIX TECHNIQUES

Le succès de ce projet repose sur la sélection et l'intégration judicieuse des technologies suivantes :

Langage principal : Python (avec le framework Flask) pour l'application web à monitorer.

Monitoring:

Prometheus: Utilisé pour le scraping, le stockage des séries temporelles et l'évaluation des règles d'alerte.

Node Exporter : Collecte des métriques système (CPU, mémoire, disque, réseau) de la machine hôte.

cAdvisor: Collecte des métriques d'utilisation des ressources des conteneurs Docker.

Visualisation : Grafana (dashboards dynamiques, explorations, alerting).

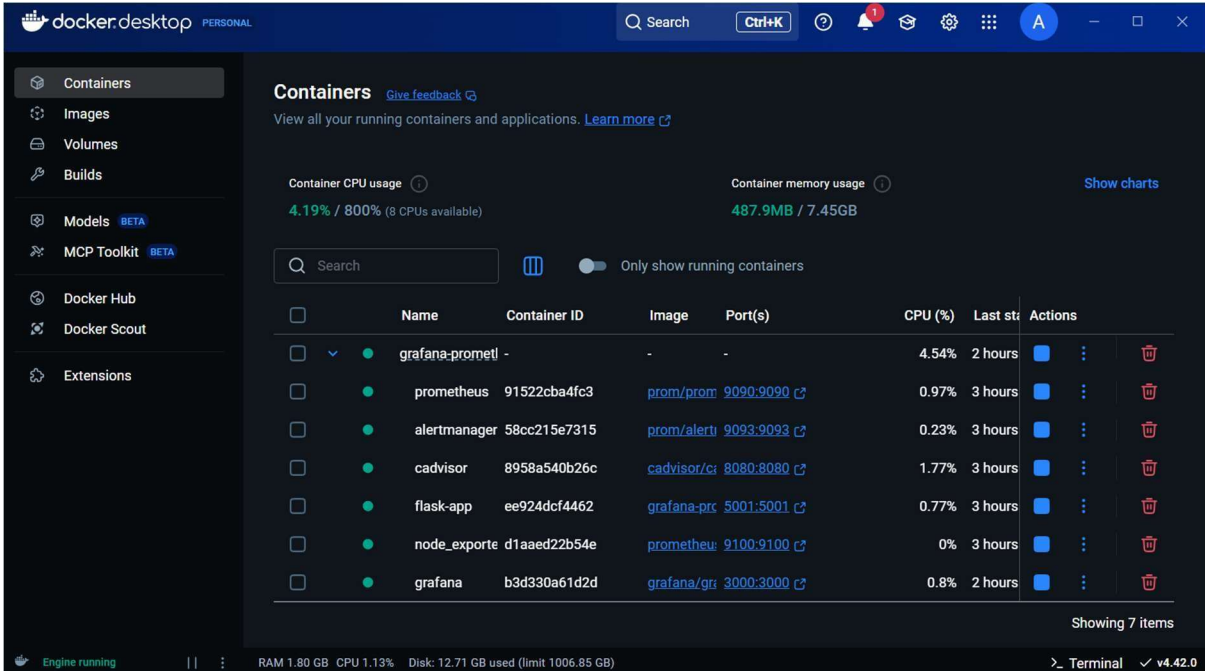
Orchestration: Docker Compose pour définir et exécuter l'application et tous les services de monitoring dans des conteneurs isolés et connectés.

Alerting: Alertmanager (gestion des alertes, déduplication, regroupement et routage vers des récepteurs comme Slack ou des webhooks).

Gestion des secrets: Utilisation d'un fichier .env pour la gestion des variables d'environnement sensibles (ex: `PROMETHEUS_HEX`).

Système d'exploitation cible: Windows 10/11, compatible Linux (via les scripts de démarrage).

Autres outils: gunicorn pour un serveur WSGI de production, psutil pour des métriques système fines dans l'application Flask, requests pour les tests de charge.



The screenshot shows the Docker Desktop interface. The left sidebar contains navigation options: Containers, Images, Volumes, Builds, Models (BETA), MCP Toolkit (BETA), Docker Hub, Docker Scout, and Extensions. The main panel displays the 'Containers' section with a search bar and a toggle for 'Only show running containers'. Below this is a table of running containers.

	Name	Container ID	Image	Port(s)	CPU (%)	Last state	Actions
<input type="checkbox"/>	grafana-prometl	-	-	-	4.54%	2 hours	
<input type="checkbox"/>	prometheus	91522cba4fc3	prom/prom	9090:9090	0.97%	3 hours	
<input type="checkbox"/>	alertmanager	58cc215e7315	prom/alert	9093:9093	0.23%	3 hours	
<input type="checkbox"/>	cadvisor	8958a540b26c	cadvisor/c	8080:8080	1.77%	3 hours	
<input type="checkbox"/>	flask-app	ee924dcf4462	grafana-pr	5001:5001	0.77%	3 hours	
<input type="checkbox"/>	node_exporte	d1aaed22b54e	prometheu	9100:9100	0%	3 hours	
<input type="checkbox"/>	grafana	b3d330a61d2d	grafana/gr	3000:3000	0.8%	2 hours	

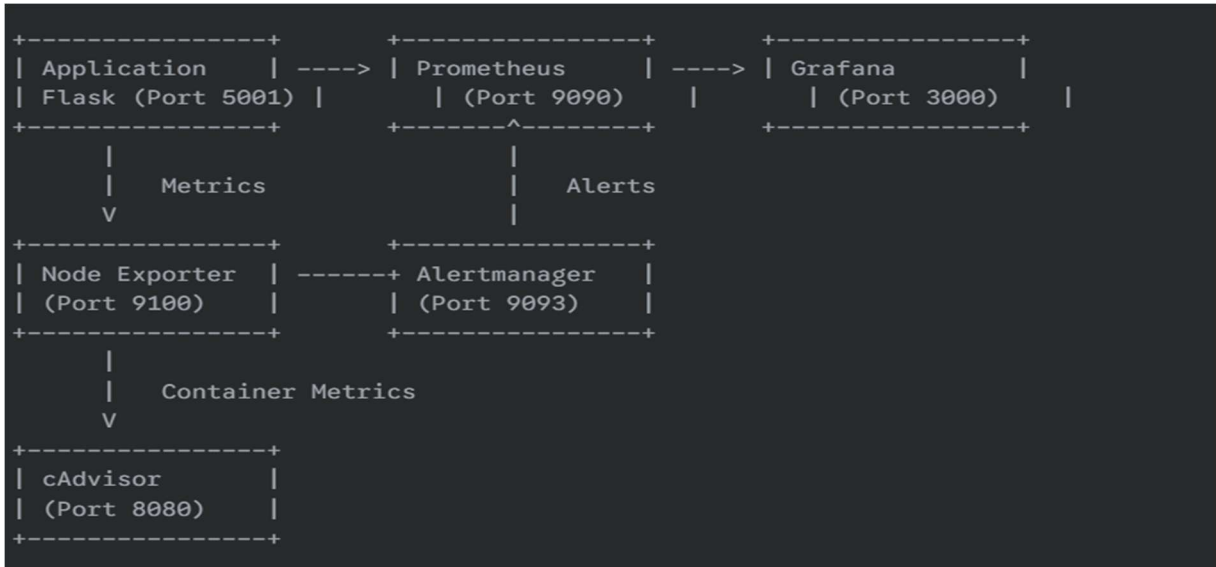
Showing 7 items

Engine running | RAM 1.80 GB | CPU 1.13% | Disk: 12.71 GB used (limit 1006.85 GB) | Terminal | v4.42.0

ARCHITECTURE DE LA SOLUTION

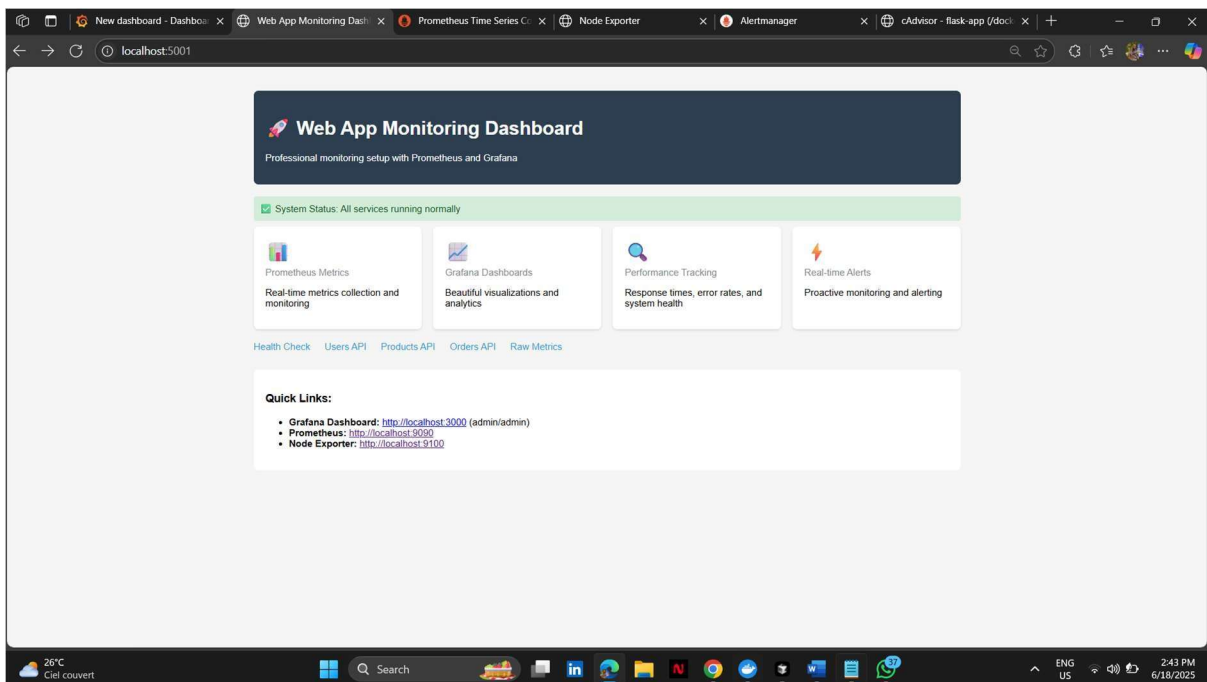
L'architecture de la solution est conçue pour être modulaire, robuste et facile à déployer, s'appuyant fortement sur la conteneurisation.

Schéma Général

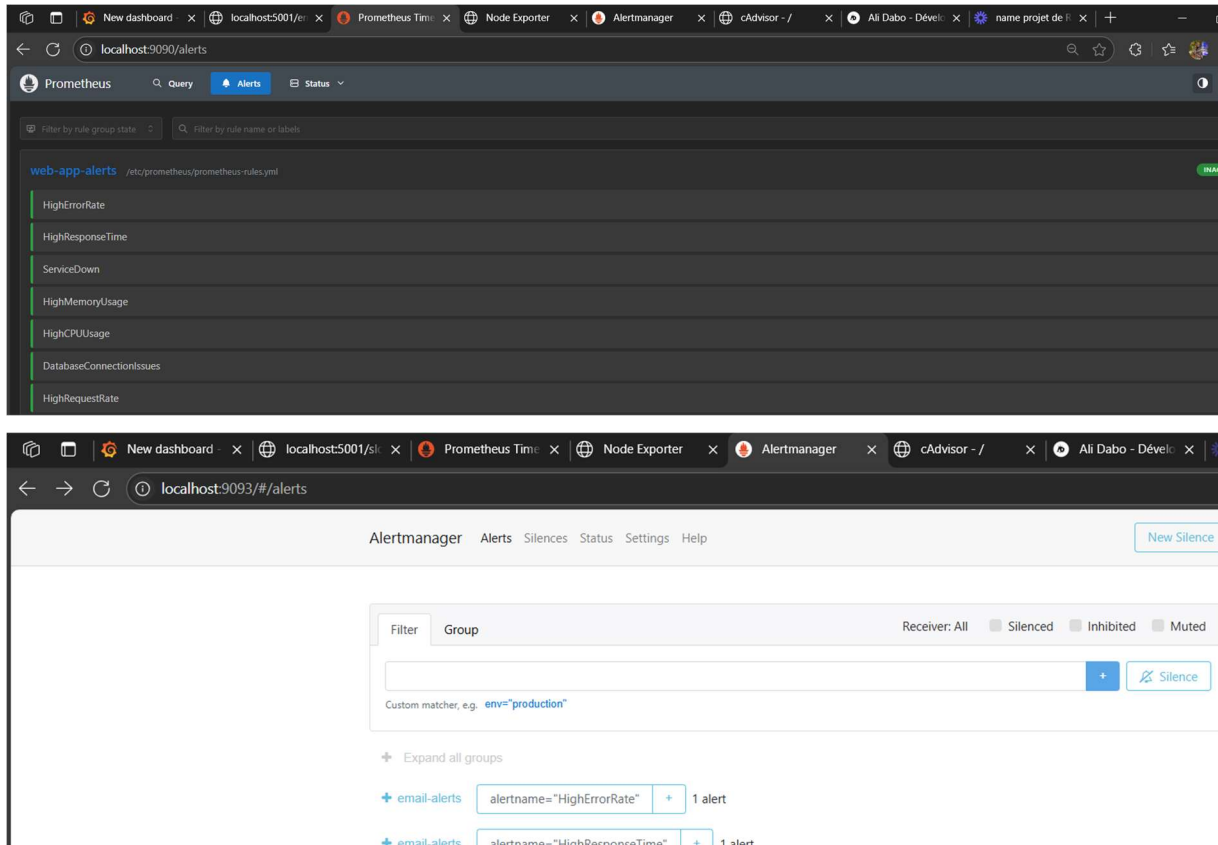


Description Détaillée

1. Application Flask: Notre application web est le service principal à monitorer. Elle expose un endpoint `/metrics` compatible Prometheus, instrumenté pour collecter des données sur le taux de requêtes, la latence, les erreurs, l'utilisation CPU/mémoire et les requêtes actives.
L'application Flask ([Web App Monitoring Dashboard](#)).



2. Prometheus: C'est le cœur du système de monitoring. Lui-même ([Prometheus Time Series Collection and Processing Server](#)) pour sa propre santé. Il stocke ces données et évalue les règles d'alerte définies dans prometheus-rules.yml.

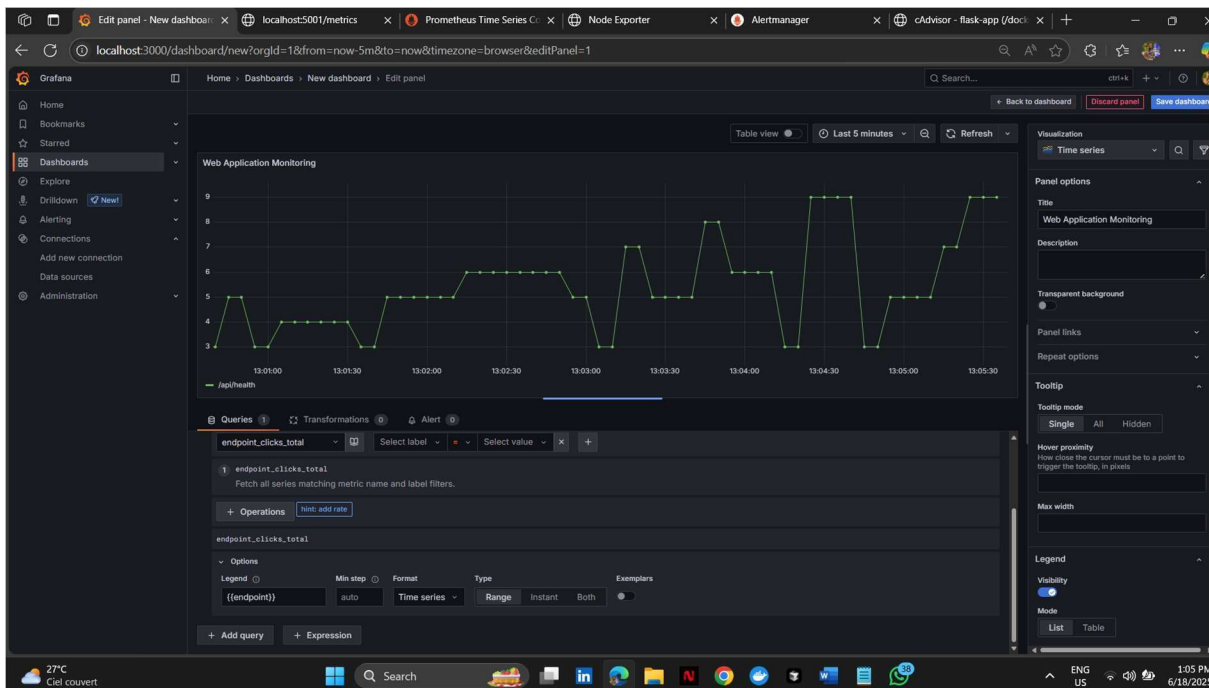


Prometheus est configuré pour scraper les métriques de :

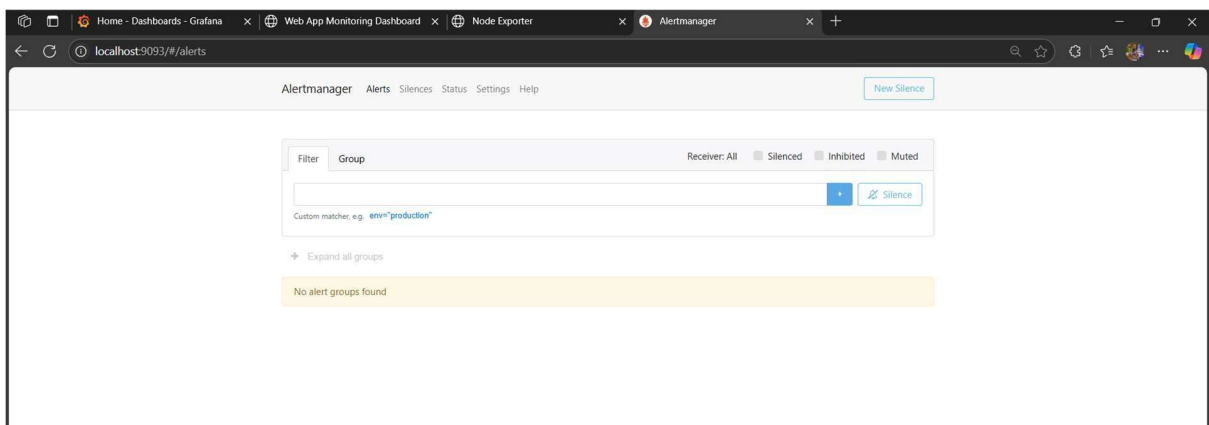
Node Exporter ([Node Exporter](#)) pour les métriques système de la machine hôte.

cAdvisor ([cAdvisor - /](#)) pour les métriques d'utilisation des ressources des conteneurs Docker.

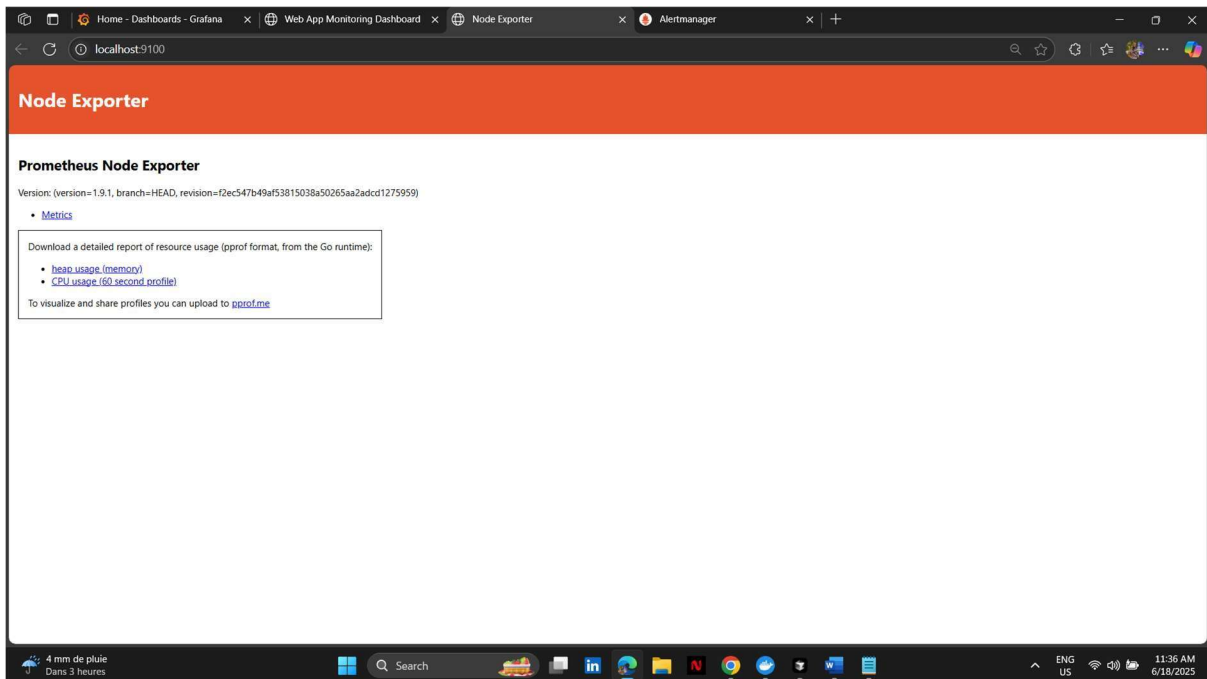
- Grafana: Se connecte à Prometheus en tant que source de données. Il est utilisé pour créer des tableaux de bord dynamiques qui visualisent les métriques collectées en temps réel. Le provisioning automatique via `datasources/datasource.yml` et `dashboards/dashboard.yml` assure que Grafana est prêt à l'emploi avec un dashboard prédéfini (`web-appdashboard.json`).



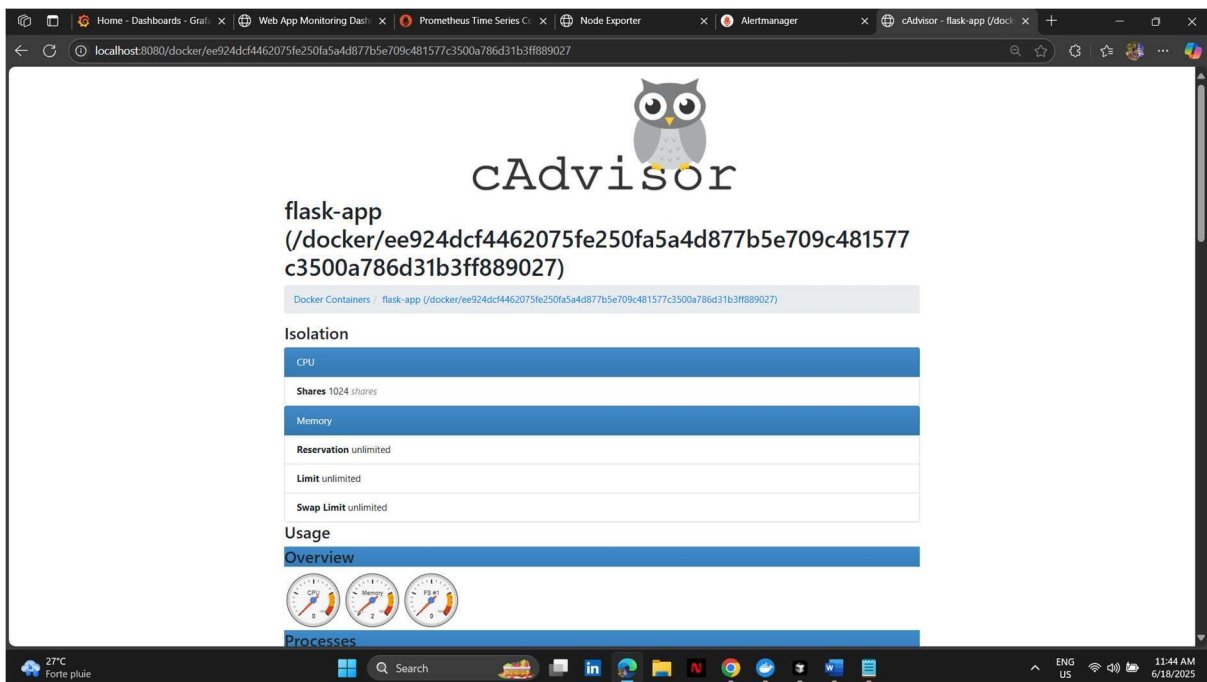
- Alertmanager : Reçoit les alertes de Prometheus, les déduplique, les regroupe et les achemine vers les récepteurs configurés (ex: Slack, webhooks), selon les règles définies dans `alertmanager.yml`.



5. Node Exporter: Un agent léger qui expose des métriques système de la machine hôte.



6. cAdvisor : Un agent qui collecte et agrège les métriques d'utilisation des ressources (CPU, mémoire, réseau, système de fichiers) de tous les conteneurs Docker en cours d'exécution.



7. Docker Compose: Orchestre tous ces services. Le fichier `dockercompose.yml` définit chaque service, ses dépendances, ses volumes persistants (pour Prometheus et Grafana afin de conserver les données), et son réseau dédié `'monitoring-network'` pour une communication interne sécurisée

Configuration des fichiers clés

`docker-compose.yml`: Définit les services (prometheus, alertmanager, grafana, flask-app, node_exporter, cadvisor), leurs images Docker, les ports exposés, les volumes persistants et le réseau.

`prometheus.yml`: Contient la configuration globale de Prometheus et les `scrape_configs` pour découvrir les cibles de monitoring.

`prometheus-rules.yml`: Définit les règles d'alerte basées sur PromQL, avec des seuils et des périodes de déclenchement. `alertmanager.yml`: Configure les routes d'alerte et les récepteurs (ex: Slack).

`grafana/provisioning/datasources/datasource.yml`: Configure automatiquement Prometheus comme source de données dans Grafana.

grafana/provisioning/dashboards/dashboard.yml et
grafana/provisioning/dashboards/web-app-dashboard.json : Gèrent le
provisioning automatique des dashboards Grafana.

ÉTAPES DE RÉALISATION

Le projet a été mené à bien à travers les étapes suivantes :

Initialisation du projet et création du dépôt Git : Mise en place de la structure de base du projet et gestion de version.

Développement de l'application Flask instrumentée avec Prometheus :

- Création de l'application web app.py avec des endpoints d'API (utilisateurs, produits, commandes, santé).
- Intégration de la bibliothèque prometheus_client pour exposer des métriques personnalisées (taux de requêtes, latence, erreurs) et système (CPU, mémoire).
- Ajout de fonctionnalités pour simuler des erreurs /api/error et des latences /api/slow afin de tester le système d'alerte.

Rédaction des Dockerfile et docker-compose.yml :

- Création du Dockerfile pour l'application Flask, assurant une image légère et sécurisée (utilisateur non-root).
- Élaboration du docker-compose.yml pour orchestrer tous les services (Flask, Prometheus, Grafana, Alertmanager, Node Exporter, cAdvisor) avec leurs configurations, volumes et réseau dédié.

Configuration de Prometheus:

- Création du fichier prometheus.yml pour définir les cibles de scraping (Flask, Node Exporter, cAdvisor, Prometheus lui-même).
- Définition des règles d'alerte dans prometheus-rules.yml (ex: HighErrorRate, HighLatency, HighCPULoad).

Provisioning automatique de Grafana :

- Configuration des fichiers datasources/datasource.yml pour ajouter Prometheus comme source de données.
- Mise en place des fichiers dashboards/dashboard.yml et web-appdashboard.json pour charger automatiquement un dashboard de monitoring complet au démarrage de Grafana.

Mise en place de l'Alertmanager:

- Création du fichier alertmanager.yml pour définir les récepteurs (webhook générique pour Slack ou autres) et les routes d'alerte.

Tests de charge et simulation d'erreurs:

- Développement d'un script load-test.py pour générer du trafic sur l'application web et simuler des conditions réelles.
- Utilisation des endpoints /api/error et /api/slow pour déclencher manuellement les alertes et vérifier leur bon fonctionnement.

Documentation et scripts de démarrage multiplateforme:

- Rédaction d'un fichier README.md détaillé.
- Création de scripts de démarrage start.sh (Linux/Mac) et start.bat (Windows) pour simplifier le déploiement et inclure des vérifications de santé.
- Mise en place d'un fichier gitignor` pour exclure les fichiers sensibles et temporaires.

RÉSULTATS OBTENUS

Les résultats obtenus démontrent la réussite du projet dans la mise en œuvre d'une solution de monitoring complète et fonctionnelle :

Supervision en temps réel: Capacité à suivre en direct la performance et la santé de l'application Flask et de l'infrastructure sous-jacente (CPU, mémoire, Docker).

Dashboards Grafana dynamiques et personnalisés: Visualisation claire et intuitive des métriques clés (taux de requêtes, latence, erreurs, utilisation des ressources) à travers un tableau de bord pré-configuré, facilitant l'identification rapide des tendances et des anomalies.

Alertes automatiques et proactives: Déclenchement fiable des alertes sur des événements critiques (taux d'erreur élevé, lenteur des réponses, saturation CPU/mémoire) grâce à Prometheus et Alertmanager, permettant une intervention rapide.

Déploiement reproductible et simplifié : L'ensemble de la stack est conteneurisée et orchestrée via Docker Compose, permettant un déploiement rapide et identique sur n'importe quelle machine compatible Docker. Les scripts start.sh et start.bat facilitent grandement le démarrage.

Validation des alertes: Le script load-test.py a permis de simuler du trafic et de valider que les alertes se déclenchent correctement lorsque les seuils sont atteints.

Accessibilité: Tous les services sont accessibles via des URLs locales dédiées :

- Grafana Dashboard: ``http://localhost:3000`` (admin/admin)
- Prometheus: ``http://localhost:9090``
- Alertmanager: ``http://localhost:9093``
- Web Application: ``http://localhost:5001``
- cAdvisor : ``http://localhost:8080``
- node exporter: ``http://localhost:9100``

DIFFICULTÉS RENCONTRÉES ET SOLUTIONS

Au cours de la réalisation de ce projet, plusieurs défis techniques ont été rencontrés et surmontés :

Problème d'encodage du fichier .env sous Windows:

- Problème: Les fichiers .env générés automatiquement sous Windows pouvaient avoir des problèmes d'encodage (ex: BOM) ce qui rendait la lecture des variables d'environnement difficile pour les applications conteneurisées.
- Solution: Le script `start.bat` a été ajusté pour forcer l'encodage ASCII ou UTF-8 sans BOM lors de la création du fichier .env, assurant une compatibilité multiplateforme.

Erreur de configuration Prometheus (`scrape_timeout` > `scrape_interval`) :

- Problème: Une configuration initiale où le `scrape_timeout` de Prometheus était supérieur ou égal au `scrape_interval` pouvait entraîner des comportements inattendus ou des alertes erronées.
- Solution: Ajustement minutieux des intervalles et des délais (`scrape_interval`, `scrape_timeout`) dans `prometheus.yml` pour garantir une collecte de métriques cohérente et éviter les faux positifs.

Dépendances Python manquantes ou incorrectes dans le Dockerfile:

- Problème: Omission de certaines bibliothèques (psutil, gunicorn) ou versions incompatibles dans `requirements.txt` lors de la construction de l'image Docker de l'application Flask.
- Solution: Révision et mise à jour rigoureuse du fichier `requirements.txt` pour inclure toutes les dépendances nécessaires et spécifier des versions compatibles, suivie d'une reconstruction de l'image Docker.

Simulation de charge sous Windows sans openssl pour la clé de l'API:

- Problème: La génération d'une clé d'API hexadécimale pour le fichier .env via openssl n'était pas nativement disponible sous Windows.
- Solution: Adaptation du script `start.bat` pour utiliser des commandes PowerShell alternatives ([System.Convert]::ToHexString((Get-Random -Count 10 -Maximum 256 | ForEach-Object { [byte]\$_ }))) pour générer une clé sécurisée et aléatoire.

Problèmes de permissions pour les volumes Docker:

- Problème: Des erreurs d'accès aux volumes persistants de Prometheus et Grafana sous Linux ou Docker Desktop (Windows/Mac) en raison de problèmes de permissions de fichiers.
- Solution: S'assurer que les utilisateurs au sein des conteneurs (ex: prometheus, grafana) ont les permissions nécessaires sur les dossiers montés, ou utiliser des options de volume Docker qui gèrent automatiquement les permissions (:z ou :Z pour SELinux, ou s'assurer que les UIDs/GIDs correspondent).

CONCLUSION ET PERSPECTIVES

Ce projet a permis de mettre en œuvre une stack de monitoring professionnelle, facilement déployable et extensible, offrant une visibilité complète sur la performance et la santé d'une application web et de son infrastructure conteneurisée. Les outils open source utilisés (Prometheus, Grafana, Docker) offrent une grande flexibilité et une forte valeur ajoutée pour la supervision d'applications modernes.

L'architecture robuste et la facilité de déploiement via Docker Compose en font une base solide pour de futurs développements.

Perspectives

Intégration d'un pipeline CI/CD pour le déploiement automatisé:

Automatiser le build des images Docker, le déploiement de la stack et la mise à jour des configurations via des outils comme Jenkins, GitLab CI/CD ou GitHub Actions.

Ajout de notifications vers d'autres canaux: Étendre les récepteurs d'Alertmanager pour inclure Teams, email, PagerDuty, ou SMS, afin d'adapter les notifications aux préférences des équipes.

Extension à la supervision multi-applications ou microservices: Adapter la configuration de Prometheus pour scraper plusieurs applications Flask ou des microservices différents, gérés par un seul tableau de bord Grafana.

Déploiement sur le cloud (AWS, Azure, GCP) avec Kubernetes : Migrer la stack Docker Compose vers une orchestration plus avancée avec Kubernetes pour une gestion de la haute disponibilité, de l'auto-scaling et du déploiement à l'échelle de production.

Intégration de Loki pour la gestion des logs: Ajouter Loki (avec Promtail) à la stack pour une collecte centralisée des logs, permettant de corréler les métriques et les logs pour un débogage plus efficace.

Annexes :

Lien vers le dépôt GitHub

https://github.com/alsondab/Prometheus_Grafana_Web_Flask.git

Manuel d'installation

Dans le README.md du fichier zip

Guide d'utilisation

Fichiers joints au rapport :

Code source (archive ZIP)

Présentation PowerPoint

Journal de bord du projet (PDF)

Ce document constitue le rapport final du projet réalisé dans le cadre du cours CloudOps. Tous les éléments techniques décrits sont opérationnels et testés.