المعهد الوطني للبريد والمواصلات
ⵜⴰⵙⴷⴰⵡⵜ ⵏⵜⵉⵎⵓⵔⴰⵜ ⴰⵎⵙⵙⵓⵔⵉ ⵏ ⵜⵉⵎⵓⵔⴰⵜ ⴰⵏⴰⵎⵓⵔ
Institut National des Postes et Télécommunications

Rapport technique

Cycle de vie DevOps complet : Intégration, Déploiement GitOps et Observabilité totale d'une application microservices.

Élaboré par :
Zakarya Belaid
Adil El Hihi

Sous la direction de :
Hassan El Farssi

14 novembre 2025

Résumé

Ce projet technique s'inscrit dans le cadre de la mise en place d'une plateforme SRE (Site Reliability Engineering) complète pour une application e-commerce (simulée) basée sur une architecture microservices. L'objectif principal est de garantir la fiabilité, la performance et l'observabilité de bout-en-bout d'un système distribué, en automatisant son déploiement et sa supervision sur Kubernetes.

L'architecture applicative repose sur cinq services spécialisés (API Gateway, Service de Commandes, Service d'Inventaire, Service Utilisateurs et Frontend) développés en Node.js. La communication inter-services est assurée par des appels HTTP directs, orchestrés par l'API Gateway. Chaque service est nativement instrumenté pour exposer les trois piliers de l'observabilité : des métriques de performance (/metrics via Prometheus), des logs structurés (via console.log), et des traces distribuées (via OpenTelemetry). Des sondes de santé (/healthz) garantissent la robustesse du déploiement (liveness/readiness probes).

Le projet implémente une culture SRE et une chaîne de déploiement GitOps. Une pipeline CI/CD sur Azure DevOps automatise la construction des images Docker, leur publication sur Docker Hub, et la mise à jour des manifestes Kubernetes (tagging d'image) dans un dépôt Git. Argo CD, agissant comme opérateur GitOps, détecte ces changements et synchronise automatiquement l'état du cluster Kubernetes. La supervision est assurée par une stack d'observabilité complète : Prometheus pour l'agrégation des métriques, Loki pour la centralisation des logs, et Jaeger pour le traçage distribué. Grafana sert d'interface de visualisation unifiée pour les dashboards (corrélant métriques et logs) et gère la configuration des alertes. Celles-ci sont traitées par Alertmanager et routées vers des canaux externes tels que Slack pour une gestion proactive des incidents.

Ce projet met ainsi en oeuvre les principes fondamentaux du DevOps moderne : l'instrumentation applicative, l'automatisation du déploiement (GitOps), et une observabilité complète (métriques, logs, traces) permettant un dépannage rapide et une surveillance en temps réel de la fiabilité du système.

Mots clés : Kubernetes, Azure DevOps, Argo CD, Docker, Prometheus, Grafana, Loki, Jaeger, Alertmanager.

Table des figures

Table des matières

Introduction générale

Avec l'adoption massive des architectures microservices, les entreprises ont gagné en agilité et en scalabilité. Cependant, cette distribution des fonctionnalités a introduit une complexité opérationnelle exponentielle. La supervision traditionnelle, le débogage et la garantie de la performance ne sont plus suffisants. Dans ce contexte, assurer la fiabilité des services devient un enjeu critique, nécessitant une plateforme d'observabilité complète capable de corréler les métriques, les logs et les traces en temps réel.

Ce projet a pour objectif la conception et le déploiement d'une plateforme SRE complète pour une application e-commerce reposant entièrement sur Kubernetes. Cette architecture, conçue pour être "nativement observable", instrumente chaque microservice pour exposer les trois piliers de l'observabilité : les métriques, les logs et les traces distribuées, tout en assurant sa robustesse via des sondes de santé.

L'implémentation du projet s'appuie sur les principes fondamentaux du SRE et du GitOps. Le déploiement est automatisé via une chaîne CI/CD sur Azure DevOps, qui gère la construction et la publication des images Docker vers Docker Hub. Un opérateur GitOps, Argo CD, assure ensuite la synchronisation continue et le déploiement des manifestes Kubernetes. La supervision est assurée par une stack d'observabilité complète : Prometheus pour l'agrégation des métriques, Loki pour la centralisation des logs, et Jaeger pour le traçage. Grafana centralise la visualisation (dashboards corrélant métriques et logs), tandis qu'Alertmanager gère les alertes proactives vers des canaux externes comme Slack.

L'ensemble de ce travail illustre l'intégration des pratiques SRE modernes dans un cycle de vie DevOps. Il démontre comment l'instrumentation applicative, couplée à une boucle de déploiement GitOps (Argo CD), permet de construire un système non seulement résilient et scalable, mais surtout entièrement "observable", facilitant la détection, le diagnostic et la résolution d'incidents complexes en temps réel.

Chapitre 1

État de lart

Ce chapitre présente l'ensemble des concepts, technologies et outils qui ont été étudiés et utilisés dans le cadre de la mise en place de la plateforme et de son écosystème d'observabilité. Il s'agit notamment des principes de l'orchestration de conteneurs avec Kubernetes, des modèles de déploiement GitOps via Argo CD, ainsi que de la stack complète d'observabilité (Prometheus, Grafana, Loki, Jaeger) pour la supervision de microservices.

À travers cette revue de létat de lart, nous cherchons à contextualiser les choix technologiques effectués, en exposant leurs fondements, avantages et pertinence dans le projet. Ce travail constitue ainsi une base de référence pour la phase dimplémentation présentée dans le chapitre suivant.

1.1 Concepts fondamentaux

1.1.1 Cloud Computing

Le terme cloud désigne les serveurs accessibles sur Internet, ainsi que les logiciels et bases de données qui fonctionnent sur ces serveurs. Les serveurs situés dans le cloud sont hébergés au sein de datacenters répartis dans le monde entier. L'utilisation de l'informatique cloud (cloud computing) permet aux utilisateurs et aux entreprises de s'affranchir de la nécessité de gérer des serveurs physiques eux-mêmes ou d'exécuter des applications logicielles sur leurs propres équipements.

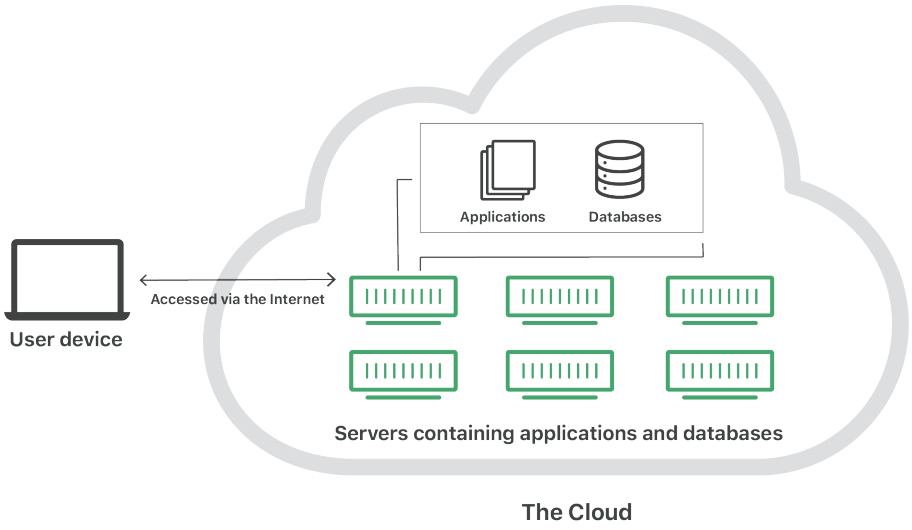

FIGURE 1.1 – Le cloud

Services du cloud

Les ressources disponibles dans le cloud sont connues sous le nom de services, car elles sont gérées activement par un fournisseur de cloud. Les services cloud comprennent l'infrastructure, les applications, les outils de développement et le stockage de données, entre autres produits. Ces services se répartissent en différentes catégories ou modèles de services.

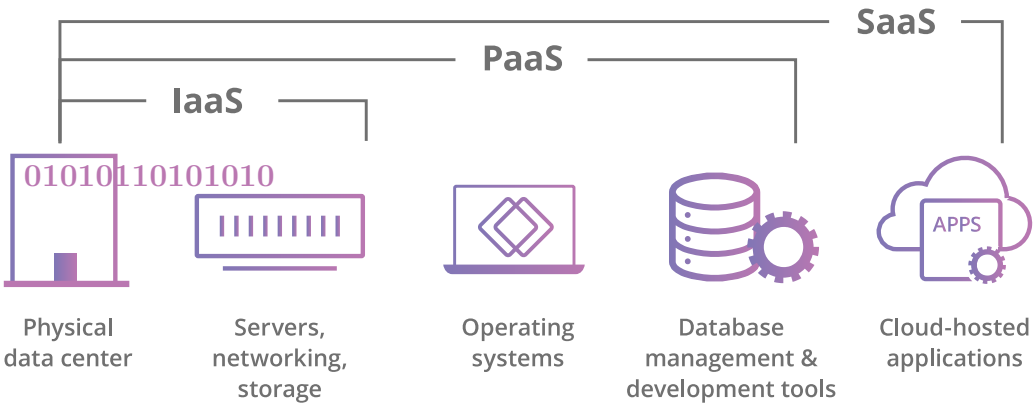

FIGURE 1.2 – Types de services cloud

1.1.2 Microservices

Les microservices sont un style architectural populaire pour concevoir des applications résilientes, hautement évolutives, déployables de manière indépendante et capables dévoluer rapidement. La mise en place réussie dune architecture microservices exige un changement fondamental de mentalité. Elle ne se limite pas à découper une application en services plus petits ; il faut également repenser la conception, le déploiement et lexploitation des systèmes.

Une architecture microservices est constituée dun ensemble de petits services autonomes. Chaque service est autonome et doit implémenter une seule capacité métier dans un contexte délimité. Un contexte délimité représente une division naturelle au sein dune entreprise et définit explicitement les frontières à lintérieur desquelles un modèle métier existe.

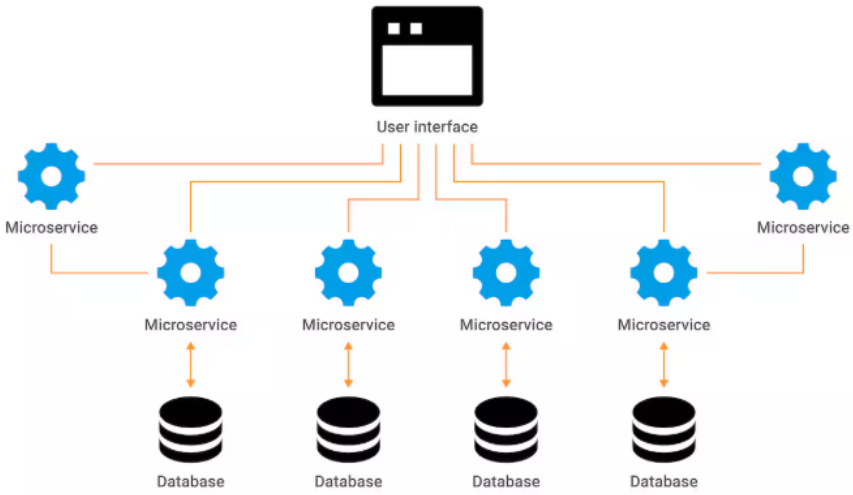

FIGURE 1.3 – l'architecture microservices

Les migrations vers microservices sont motivées par la maintenabilité, scalabilité et rapidité de déploiement ; tests et gestion de la cohérence des données restent des défis majeurs.

1.1.3 DevSecOps

Le DevSecOps est un modèle qui englobe le développement (Dev), la sécurité (Sec) et l'exploitation (Ops). Cette approche de la culture, de l'automatisation et de la conception des plateformes intègre la sécurité en tant que responsabilité partagée tout au long du cycle de vie informatique.

Les conteneurs ont amélioré l'évolutivité et le dynamisme du développement et du déploiement, ce qui a transformé la manière dont la plupart des entreprises innovent. Pour cette raison, les pratiques de sécurité DevOps doivent s'adapter au nouveau paysage et se conformer aux instructions de sécurité propres aux conteneurs.

Les technologies cloud-native ne sont pas compatibles avec des politiques et des listes de contrôle de sécurité statiques. Il est nécessaire que la sécurité soit continue et intégrée à chaque étape du cycle de vie de l'application et de l'infrastructure.

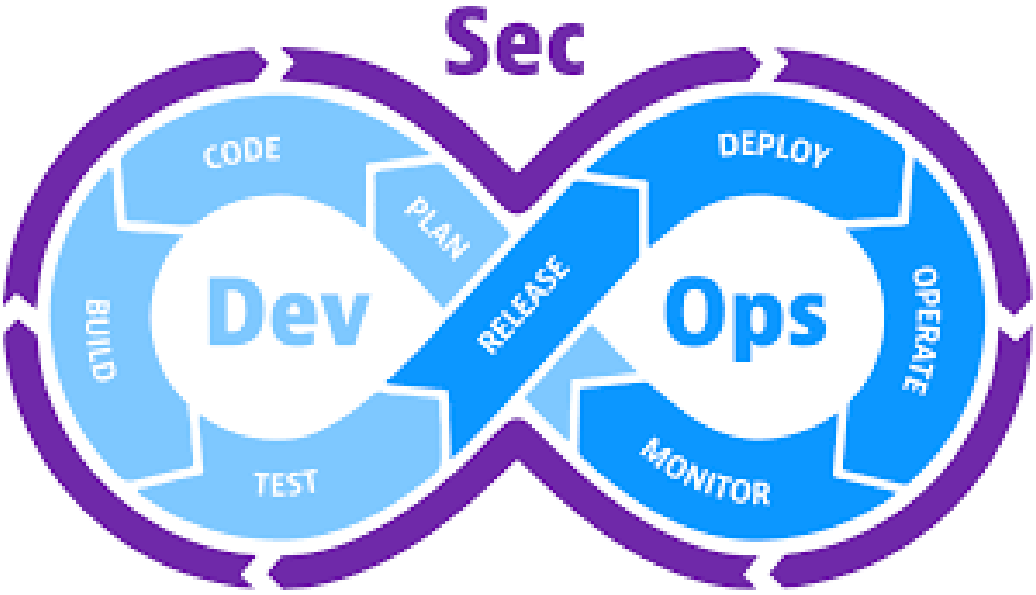

FIGURE 1.4 – Cycle de vie d'un microservice

1.1.4 Conteneurisation et Orchestration

Kubernetes (K8s) : Kubernetes est le système d'orchestration de conteneurs qui gère le déploiement, la mise à l'échelle et la résilience de notre application. Il automatise le placement des conteneurs (Pods) et gère le réseau interne (Services) ainsi que les sondes de santé (liveness/readiness probes). Pour ce lab, Minikube est utilisé pour simuler un cluster Kubernetes complet en local.

FIGURE 1.5 – Kubernetes

1.1.5 La Stack d'Observabilité Complète

L'observabilité est la capacité de comprendre l'état interne d'un système en se basant sur ses sorties externes. Elle repose sur quatre piliers, Surveillance, traçage, logging et visualisation.

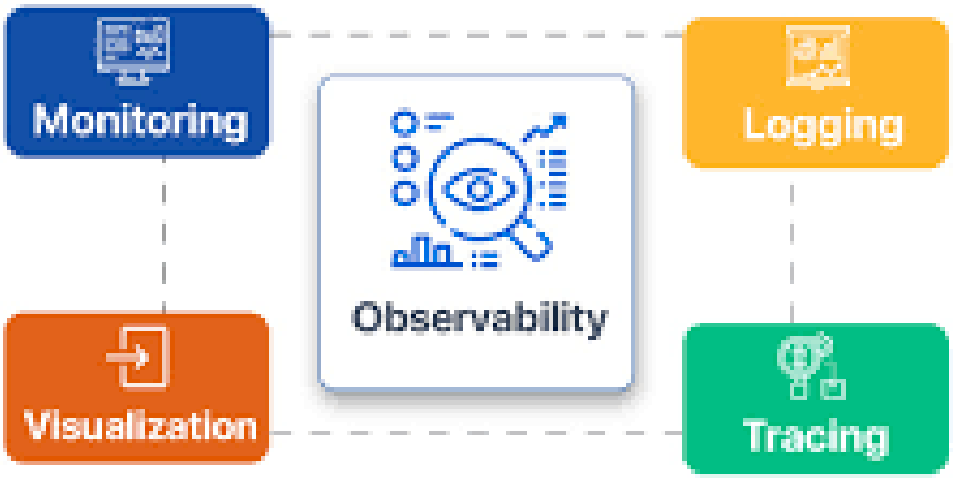

FIGURE 1.6 – Piliers

1.2 Analyse Comparative des Outils d'Observabilité

Si notre projet utilise la stack "PLG" (Prometheus, Loki, Grafana) et Jaeger, il est important de la situer par rapport aux autres solutions majeures du marché.

1.2.1 La stack PLG (Prometheus, Loki, Grafana) + Jaeger

C'est l'approche "cloud-native" open-source, soutenue par la Cloud Native Computing Foundation (CNCF).

— **Description :** Elle combine Prometheus pour les métriques, Loki pour les logs (avec une indexation par labels similaire à Prometheus), et Grafana pour la visualisation unifiée des deux. Jaeger est ajouté pour le traçage distribué (bien que Grafana Tempo soit une alternative plus récente).

— **Avantages :** Open-source, standard de facto pour Kubernetes, communauté très active, pas de "vendor lock-in". L'approche de Loki (indexation par labels uniquement) est extrêmement efficace en termes de coût et de ressources par rapport à l'indexation full-text.

— **Inconvénients :** Nécessite d'assembler, de configurer et de maintenir plusieurs composants distincts (Prometheus, Loki, Jaeger, Promtail), comme démontré dans ce projet.

1.2.2 La stack ELK (Elasticsearch, Logstash, Kibana)

Historiquement, la stack ELK (ou Elastic Stack) a été la solution open-source dominante, principalement pour la gestion des logs.

— **Description :** Elle utilise Elasticsearch (un moteur de recherche full-text) pour stocker et indexer les logs, Logstash (ou Fluentd/Beats) pour les collecter et les transformer, et Kibana pour la visualisation.

— **Avantages :** Extrêmement puissante pour la recherche textuelle complexe sur de grands volumes de logs.

— **Inconvénients :** Plus lourde et plus gourmande en ressources que Loki, car elle indexe l'intégralité du contenu de chaque ligne de log. Kibana est moins mature que Grafana pour la visualisation de métriques et de traces, rendant l'interface moins unifiée.

1.2.3 Les plateformes SaaS (Datadog, New Relic, Splunk)

Ce sont des solutions commerciales "tout-en-un" qui fournissent l'observabilité en tant que service (SaaS).

— **Description :** Un agent unique est installé sur les nuds et collecte automatiquement les métriques, les logs et les traces, puis les envoie à la plateforme du fournisseur.

— **Avantages :** Simplicité extrême (zéro maintenance d'infrastructure), interfaces très soignées, et les trois piliers (métriques, logs, traces) sont corrélés nativement sans configuration.

— **Inconvénients :** Coût (généralement basé sur le volume de données ou le nombre d'hôtes), "vendor lock-in" (dépendance à un fournisseur), et moins de flexibilité de configuration que les outils open-source.

1.3 Analyse Comparative des Outils GitOps

De même, pour le déploiement continu, nous avons utilisé Argo CD, qui s'inscrit dans l'écosystème GitOps.

1.3.1 Argo CD (Approche "Pull-based" avec UI)

C'est la solution que nous avons implémentée.

— **Description :** Un outil GitOps déclaratif pour Kubernetes, membre de la CNCF. Il s'exécute comme un opérateur dans le cluster, surveille un dépôt Git et "tire" (pull) les changements pour les appliquer.

— **Avantages :** Fournit une interface utilisateur (UI) web très complète pour visualiser l'état de la synchronisation, voir les "diffs" et gérer les applications. Très populaire dans l'écosystème.

1.3.2 Flux CD (Approche "Pull-based" native)

Flux est l'autre projet majeur de GitOps de la CNCF, souvent considéré comme le "rival" d'Argo CD.

— **Description :** Également un opérateur "pull-based", Flux est souvent perçu comme plus léger et plus "natif" de Kubernetes. Il s'appuie fortement sur les CRD (Custom Resource Definitions) pour tout configurer.

— **Inconvénients :** Historiquement, il ne disposait pas d'interface graphique (bien que des solutions tierces existent), rendant le débogage et la visualisation de l'état plus complexes pour les débutants (tout se fait en ligne de commande ou en YAML).

Conclusion

L'étude de l'état de l'art a permis de mettre en évidence les concepts et technologies clés qui définissent la gestion des systèmes distribués modernes. L'orchestration de conteneurs via **Kubernetes** apporte résilience et scalabilité, tandis que les principes du **Site Reliability Engineering (SRE)** fournissent une méthodologie éprouvée pour garantir la fiabilité. Le **GitOps** (implémenté par Argo CD) prolonge cette logique en automatisant les déploiements de manière déclarative et sécurisée. Enfin, la **stack d'observabilité** complète (Prometheus, Loki, Jaeger) offre une visibilité approfondie sur l'état interne du système en corrélant métriques, logs et traces.

Ces éléments forment un socle conceptuel et technologique solide qui a guidé l'implémentation de la plateforme d'observabilité et de la chaîne de déploiement CI/CD présentées dans les sections suivantes de ce rapport.

Chapitre 2

Présentation et Conception de la Plateforme

Ce chapitre présente de manière détaillée la conception et la mise en place de la plateforme dans le cadre de ce projet. Après avoir défini les fondements théoriques dans le chapitre précédent, il sagit ici de décrire larchitecture globale, les choix techniques et les outils mis en uvre pour réaliser la solution.

Le chapitre est structuré de façon progressive. Il commence par la description de l'application (une simulation de microservices e-commerce) qui sert de contexte pour notre travail. Il détaille ensuite la conception et la mise en oeuvre de la plateforme qui l'entoure. Cela inclut la chaîne de déploiement CI/CD GitOps (Azure DevOps, Docker Hub, Argo CD) et l'implémentation de la stack d'observabilité complète (Prometheus, Grafana, Loki, Jaeger) au sein de l'environnement Kubernetes.

Chaque section est accompagnée de diagrammes darchitecture, de tableaux de bord (dashboards) et dextraits de code ou de configuration YAML illustrant les principales étapes de conception. Lensemble vise à démontrer la cohérence, la modularité et la robustesse de la plateforme d'observabilité proposée, tout en assurant la fiabilité, la maintenabilité et la supervision automatisée du système.

2.1 Contexte : L'Application à Superviser

2.1.1 Architecture des microservices applicatifs

Pour construire et valider une plateforme d'observabilité, il est indispensable de disposer d'une application à superviser. Dans le cadre de ce lab, une application de microservices Node.js simulant un système e-commerce simple a été développée.

Cette application est volontairement conçue pour modéliser les interactions complexes d'un système distribué. Elle se compose des cinq services suivants, chacun conteneurisé avec Docker :

- **Frontend :** Une interface utilisateur Nginx qui permet à l'utilisateur de générer du trafic.
- **API Gateway :** Le point d'entrée unique (basé sur Express.js) qui reçoit les requêtes du frontend et les route vers les services internes appropriés.
- **Order Service :** Gère la création de commandes.
- **Inventory Service :** Gère l'état des stocks des articles.
- **User Service :** Gère les informations des utilisateurs (optionnel).

Le flux de requêtes simule une dépendance de service : le Frontend appelle l'API Gateway, qui à son tour appelle les services Order, Inventory et User. De plus, le service Order appelle lui-même le service Inventory pour vérifier la disponibilité d'un article avant de valider une commande. C'est cette architecture distribuée qui sera instrumentée, déployée et supervisée.

Le diagramme ci-dessous illustre le flux de communication entre les différents microservices de notre application.

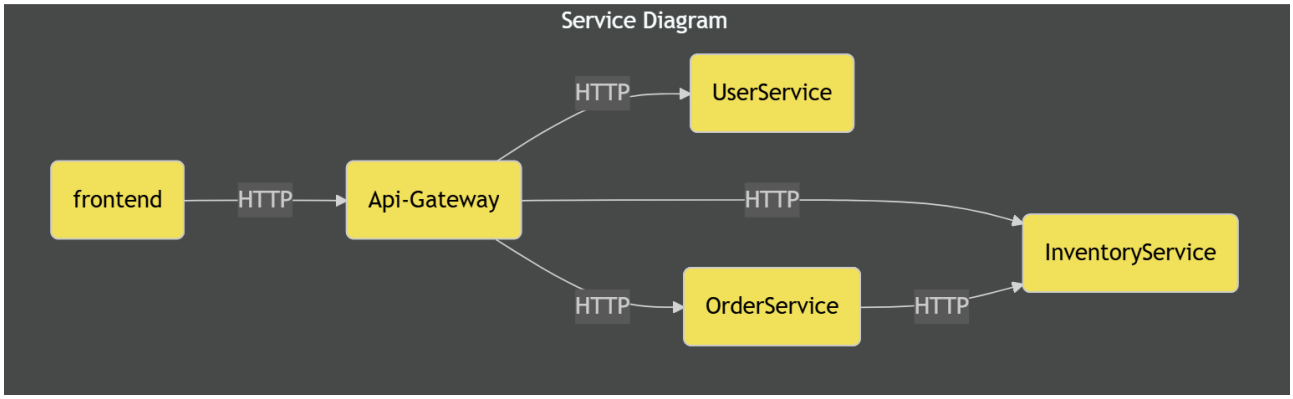

FIGURE 2.1 – Architecture

2.1.2 Conteneurisation des microservices (Docker)

Une étape fondamentale de ce projet est la conteneurisation de chaque microservice (API Gateway, Order Service, etc.). La conteneurisation, via Docker, permet d'encapsuler le code de l'application, ses dépendances (comme les bibliothèques Node.js) et sa configuration dans une image portable et reproductible.

Cette image garantit que l'application s'exécutera de manière identique, quel que soit l'environnement (développement local, CI/CD, ou production sur Kubernetes). Tous les services Node.js du projet partagent un Dockerfile similaire, dont la structure est la suivante :

```
FROM node:latest
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8000
CMD ["npm", "start"]
```

FIGURE 2.2 – Docker file de service d'Api

Explication technique des instructions Docker

Chaque ligne de ce Dockerfile exécute une commande spécifique pour construire l'image :

— **FROM node :latest** : Définit l'image de base. Nous partons d'une image officielle fournie par Node.js (version latest), qui contient déjà le runtime Node.js et l'outil npm.

— **WORKDIR /app** : Définit le répertoire de travail à l'intérieur du conteneur. Toutes les commandes suivantes (comme 'COPY' et 'RUN') s'exécuteront à partir de ce dossier '/app'.

— **COPY package*.json ./** : Copie les fichiers 'package.json' et 'package-lock.json' (grâce au '*') de notre machine locale vers le répertoire '/app' du conteneur. Cette étape est cruciale pour la mise en cache : elle n'est ré-exécutée que si ces fichiers changent.

— **RUN npm install** : Exécute la commande 'npm install' à l'intérieur du conteneur. En se basant sur les fichiers 'package.json' copiés à l'étape précédente, 'npm' télécharge et installe toutes les dépendances du projet (comme 'express', 'axios', 'prom-client', etc.).

— **COPY . .** : Copie *tout le reste* du code source (comme 'node.js', 'metrics.js', etc.) de notre dossier local vers le répertoire '/app' du conteneur.

— **EXPOSE 8000** : Informe Docker que l'application à l'intérieur du conteneur écoutera sur le port '8000' (dans notre cas). C'est une métadonnée qui facilite la configuration réseau.

— **CMD ["npm", "start"]** : Définit la commande par défaut qui sera exécutée lorsque le conteneur démarre. Cela lance notre application en exécutant le script 'start' défini dans notre 'package.json' (ex : 'node -r ./tracing.js node.js').

2.1.3 Validation locale avec Docker Compose

Avant de déployer l'application sur un orchestrateur complexe comme Kubernetes, une première validation de l'architecture et des interactions entre services a été effectuée localement à l'aide de **Docker Compose**.

```
services:
 user-service:
   image: zakaryab2003/fsdevopsuser:latest
   networks:
     - app-network

 order-service:
   image: zakaryab2003/fsdevopsorder:latest
   networks:
     - app-network
   environment:
     - INVENTORY_SERVICE_URL=http://inventory-service:3002

 inventory-service:
   image: zakaryab2003/fsdevopsinventory:latest
   networks:
     - app-network

 api-gateway:
   image: zakaryab2003/fsdevopsapi:latest
   ports:
     - "8000:8000"
   networks:
     - app-network
   depends_on:
     - user-service
     - order-service
     - inventory-service
   environment:
     - ORDER_SERVICE_URL=http://order-service:3001
     - INVENTORY_SERVICE_URL=http://inventory-service:3002
     - USER_SERVICE_URL=http://user-service:3003

 frontend:
   image: zakaryab2003/fsdevopsfrontend:latest
   ports:
     - "80:80"
   networks:
     - app-network
   environment:
     - GATEWAY_URL=http://localhost:8000
   depends_on:
     - api-gateway

networks:
  app-network:
    driver: bridge
```

FIGURE 2.3 – Fichier `docker-compose.yaml` pour le test local

Explication technique générale

Ce fichier `docker-compose.yaml` définit l'ensemble de notre application :

— **services :** Chaque bloc (comme api-gateway, order-service, etc.) définit un conteneur.

— **image :** Spécifie l'image Docker à utiliser pour ce service, ici tirée de Docker Hub (ex : zakaryab2003/fsdevopsapi :latest).

— **networks :** Tous les services sont connectés à un réseau privé virtuel commun (app-network). Cela permet aux conteneurs de communiquer entre eux en utilisant simplement leur nom de service (ex : http ://order-service) comme s'ils étaient sur un réseau réel.

— **environment :** Permet d'injecter des variables d'environnement dans les conteneurs. C'est ainsi que l'api-gateway apprend les adresses internes des autres services (ex : ORDER_SERVICE_URL=http ://order-service :3001).

— **ports :** Expose les ports d'un conteneur sur la machine hôte. Seuls le frontend (port 80) et l'api-gateway (port 8000) sont exposés, car ce sont les points d'entrée externes.

— **depends_on :** Contrôle l'ordre de démarrage, garantissant que les services backend (comme order-service) sont démarrés avant l'api-gateway qui en dépend.

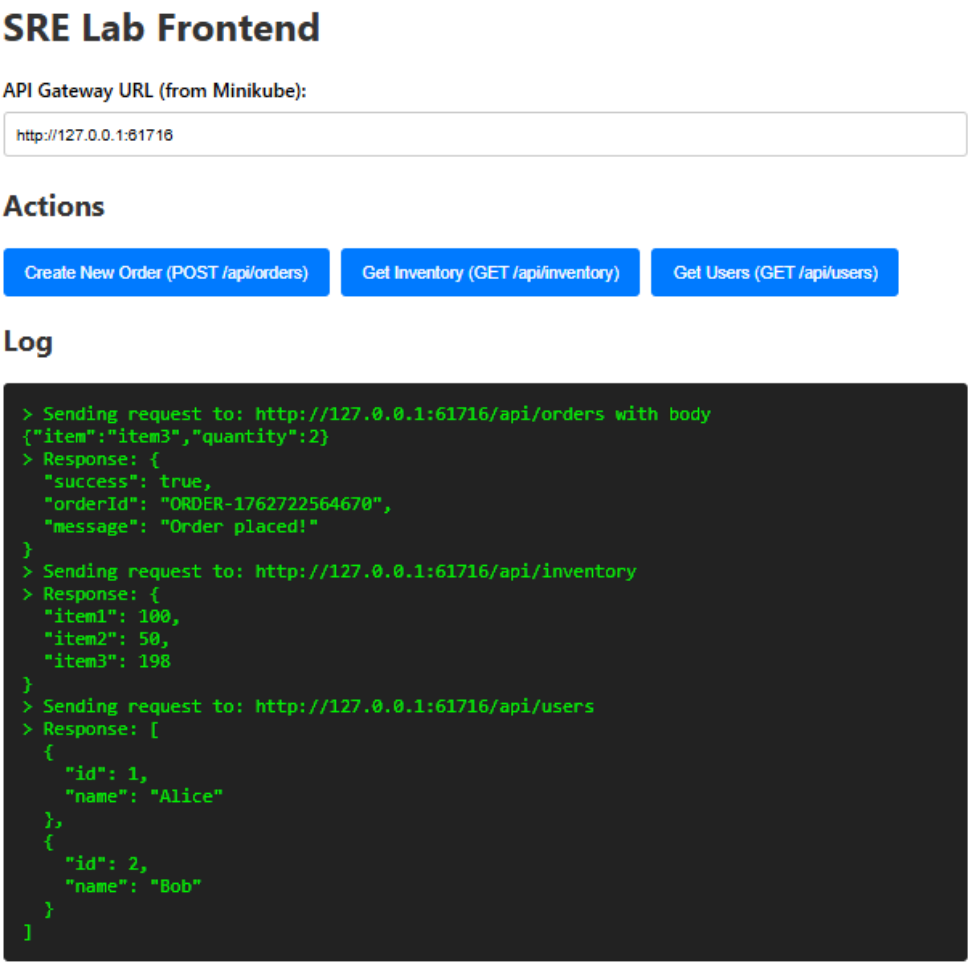

FIGURE 2.4 – Test d'application

2.1.4 Validation des endpoints d'observabilité avec Postman

Une fois les services démarrés localement avec Docker Compose, l'étape suivante consiste à valider que l'instrumentation (les endpoints de supervision) fonctionne correctement. Cette vérification a été effectuée à l'aide de l'outil **Postman** pour interroger directement les endpoints /healthz et /metrics exposés par chaque service.

Cette étape est cruciale : elle confirme que les services sont non seulement opérationnels, mais qu'ils exposent également les données nécessaires à notre future stack d'observabilité (Prometheus, Kubernetes).

Test du endpoint /healthz

Le endpoint /healthz (pour "health check") est une sonde de santé de base. Son unique rôle est de répondre OK si l'application est démarrée et fonctionne, ou de ne pas répondre (ou de renvoyer une erreur) en cas de crash.

— **Requête :** Une simple requête GET est envoyée, par exemple, à l'API Gateway sur http ://localhost :8000/healthz.

— **Résultat attendu :** Le service doit répondre avec un statut 200 OK et le corps de la réponse doit contenir le texte "OK".

— **Importance :** Ce test valide la sonde de "liveness" (vivacité) que Kubernetes utilisera plus tard pour redémarrer automatiquement un conteneur qui ne répond plus.

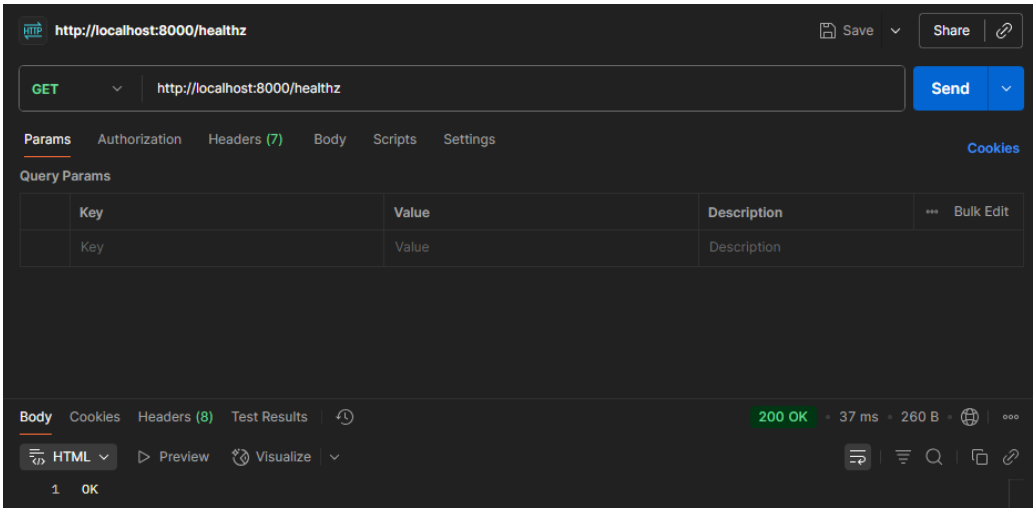

FIGURE 2.5 – Test d'application

Test du endpoint /metrics

Le endpoint /metrics est le cur de notre instrumentation pour Prometheus. C'est ici que notre librairie prom-client expose toutes les métriques que nous avons définies (compteurs, gauges, histogrammes).

— **Requête :** Une requête GET est envoyée à l'endpoint des métriques, par exemple http ://-localhost :8000/metrics.

— **Résultat attendu :** Le service doit répondre avec un statut 200 OK et un corps de réponse en format texte brut. Ce texte est la liste de toutes les métriques au format d'exposition Prometheus (ex : http_requests_total...).

— **Importance :** Ce test confirme que l'application expose activement ses métriques et que Prometheus sera capable de les "scraper" (collecter) une fois déployé.

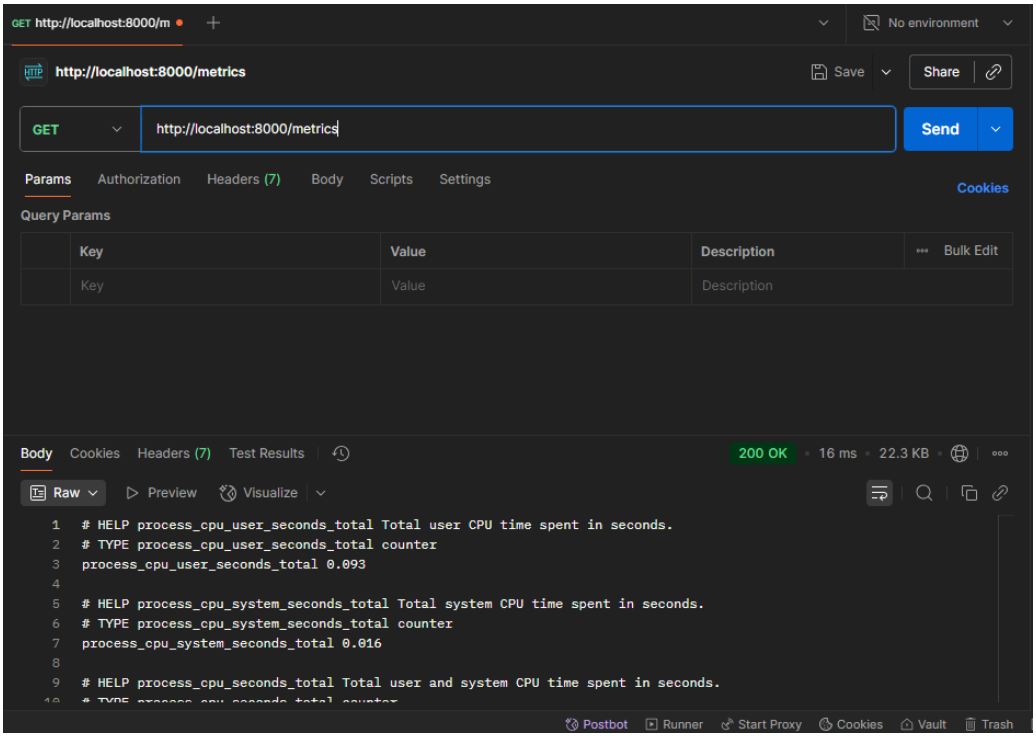

FIGURE 2.6 – Test d'application

Implémentation des endpoints d'instrumentation

La validation des endpoints /healthz et /metrics est rendue possible par leur implémentation explicite dans le code de chaque service (par exemple, gateway.js), en utilisant la bibliothèque prom-client.

Le code suivant illustre comment ces routes sont définies dans une application Express.js, ainsi que la définition d'une métrique de type Counter :

```
 // --- 1. Définition d'une métrique (Counter) ---
const httpRequestsTotal = new client.Counter({
  name: 'http_requests_total',
  help: 'Total number of HTTP requests',
  labelNames: ['service', 'method', 'route', 'status_code'],
  registers: [register],
});

// --- 2. Implémentation du Health Check (/healthz) ---
app.get('/healthz', (req, res) => {
  // In a real app, you might check DB connections, etc.
  res.status(200).send('OK');
});

// --- 3. Implémentation du Metrics Endpoint (/metrics) ---
 app.get('/metrics', async (req, res) => {
  try {
    res.set('Content-Type', register.contentType);
    res.end(await register.metrics());
  } catch (ex) {
    res.status(500).end(ex);
  }
});
```

FIGURE 2.7 – Extraits de code pour l'implémentation des métriques et health checks

Comme le montre ce code :

— La métrique httpRequestsTotal est un **Counter** qui s'incrémente à chaque requête.

— L'endpoint /healthz est une route simple qui renvoie OK pour confirmer que le serveur est en ligne.

— L'endpoint /metrics utilise la fonction register.metrics() de prom-client pour collecter et exposer toutes les métriques définies (y compris httpRequestsTotal et les métriques système par défaut) au format textuel attendu par Prometheus.

Note : Il faut integrer un exporteur de traces pour que jaeger collecte les traces, il existe un exporteur officiel *@opentelemetry/exporter-jaeger*

2.2 Mise en place de la Boucle GitOps (CI/CD)

La première étape de la conception de notre plateforme technique consiste à préparer l'environnement Kubernetes. Une infrastructure bien organisée est fondamentale pour la maintenabilité, la sécurité et la gestion des ressources.

2.2.1 Isolation des ressources avec les Namespaces

Plutôt que de déployer l'ensemble de nos outils et de notre application dans le namespace default, une segmentation a été mise en place via des **Namespaces** Kubernetes. Un namespace est un "cluster virtuel" qui permet d'isoler logiquement des groupes de ressources.

Cette approche permet de séparer les différentes responsabilités :

— **sre-lab :** Le namespace dédié à l'application que nous supervisons (API Gateway, Order Service...).

— **argocd :** Un namespace isolé pour l'outil de déploiement GitOps (Argo CD).

— **monitoring :** Un namespace pour la stack de métriques (Prometheus, Grafana, Alertmanager).

— **logging :** Le namespace pour la stack de journalisation (Loki et Promtail).

— **tracing :** Le namespace pour le système de traçage distribué (Jaeger) et (sre-lab) pour les microservices.

La création de ces namespaces a été effectuée à l'aide de la commande kubectl de base, comme illustré ci-dessous :

```
kubectl create namespace $(Nom_Namespace)
```

FIGURE 2.8 – Commandes de création des namespaces

Cette séparation garantit que les outils de supervision ne sont pas mélangés avec l'application supervisée et permet d'appliquer des politiques de sécurité (RBAC) et des quotas de ressources spécifiques à chaque périmètre.

2.2.2 Déploiement sur Kubernetes : Manifestes de l'application

Une fois les images Docker construites et poussées sur Docker Hub, l'étape suivante consiste à les déployer sur notre cluster Kubernetes. Pour ce faire, nous utilisons des fichiers de configuration déclaratifs en YAML, appelés **Manifestes**.

Pour chaque microservice, deux objets Kubernetes principaux sont requis :

— **Deployment :** Un objet qui décrit l'état souhaité de l'application. Il définit quelle image conteneur utiliser, le nombre de répliques (instances de pod), les variables d'environnement, les sondes de santé, et les ressources à allouer.

— **Service :** Un objet qui définit un point d'entrée réseau stable pour un ensemble de pods (gérés par un Deployment). Il permet aux différents services de communiquer entre eux (Service de type ClusterIP) ou d'exposer un service à l'extérieur du cluster (Service de type NodePort ou LoadBalancer).

Les manifestes suivants pour l'api-gateway servent d'exemple représentatif pour tous les autres services.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway-deploy
  namespace: sre-lab
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-gateway
  template:
    metadata:
      labels:
        app: api-gateway
      annotations:
        prometheus.io/scrape: 'true'
        prometheus.io/path: '/metrics'
        prometheus.io/port: '8000'
    spec:
      containers:
        - name: api-gateway
          image: zakaryab2003/fsdevopsapi:71
          imagePullPolicy: Always
          ports:
            - containerPort: 8000
          env:
            - name: ORDER_SERVICE_URL
              value: "http://order-service-svc.sre-lab:3001"
            - name: INVENTORY_SERVICE_URL
              value: "http://inventory-service-svc.sre-lab:3002"
            - name: USER_SERVICE_URL
              value: "http://user-service-svc.sre-lab:3003"
            - name: OTEL_SERVICE_NAME
              value: "api-gateway"
            - name: OTEL_EXPORTER_JAEGER_ENDPOINT
              value: "http://jaeger-all-in-one.tracing.svc.cluster.
    local:14268/api/traces"
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8000
            initialDelaySeconds: 30
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8000
            initialDelaySeconds: 15
            periodSeconds: 5
          resources:
```

```
            requests:
              memory: "128Mi"
              cpu: "250m"
            limits:
              memory: "256Mi"
              cpu: "500m"
```

Listing 2.1 – Exemple de manifeste de Deployment pour l'API Gateway

```
apiVersion: v1
kind: Service
metadata:
  name: api-gateway-svc
  namespace: sre-lab
spec:
  selector:
    app: api-gateway
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 8000
      nodePort: 30007
  type: NodePort
```

Listing 2.2 – Service d'api-gateway

2.2.3 Mise en place de la Boucle GitOps (CI/CD)

L'automatisation du déploiement est un pilier de ce projet. L'objectif est de créer un flux de travail où une modification du code déclenche un déploiement de manière automatique, sécurisée et traçable. Ceci est réalisé en combinant une pipeline d'intégration continue (CI) avec une approche de déploiement continu (CD) basée sur GitOps.

Intégration Continue (CI) avec Azure DevOps

Le pipeline d'intégration continue est le moteur qui transforme le code source en un artefact déployable (une image Docker). Pour ce projet, **Azure DevOps** a été utilisé pour orchestrer ce processus.

Un pipeline YAML distinct a été créé pour chaque microservice, configuré pour se déclencher uniquement lorsque des modifications sont détectées dans le dossier du service concerné (ex : api-gateway/). Le pipeline exécute deux étapes principales : "BuildPush" et "Update".

```
# Docker
# Build a Docker image
# https://docs.microsoft.com/azure/devops/pipelines/languages/docker

trigger:
  paths:
    include:
      - api-gateway/*
```

```
9
10 resources:
11 - repo: self
12
13 variables:
14   - group: 'variable group for the project pipelines'
15   - name: tag
16     value: '$(Build.BuildId)'
17   - name: imageRepository
18     value: 'zakaryab2003/fsdevopsapi'
19   - name: dockerfilePath
20     value: 'api-gateway/Dockerfile'
21   - name: buildContext
22     value: 'api-gateway/'
23   - name: scriptPath
24     value: 'Scripts/UpdateK8sManifests.sh'
25   - name: serviceName
26     value: 'api-gateway'
27
28 stages:
29 - stage: Build
30   displayName: Build image
31   jobs:
32   - job: Build
33     displayName: Build
34     pool:
35       vmImage: ubuntu-latest
36     steps:
37     - task: Docker@2
38       inputs:
39         containerRegistry: 'image-registery'
40         repository: 'zakaryab2003/fsdevopsapi'
41         command: 'buildAndPush'
42         Dockerfile: 'api-gateway/Dockerfile'
43         buildContext: 'api-gateway/'
44         tags: |
45           $(tag)
46
47 - stage: Update
48   displayName: 'Update Manifest'
49   dependsOn: Build
50   jobs:
51     - job: Update
52       displayName: 'Update Manifest in Git'
53       pool:
54         vmImage: ubuntu-latest
55       steps:
56         - task: ShellScript@2
57           displayName: 'Run update-manifest.sh'
58           inputs:
```

```
            scriptPath: '$(scriptPath)'
            args: '$(serviceName) $(imageRepository) $(tag)'
          env:
            # This correctly maps the secret variable from your group
            # to the $GITHUB_TOKEN env variable for the script.
            GITHUB_TOKEN: $(GITHUB_TOKEN)
```

Listing 2.3 – Pipeline d'integration

```
#!/bin/bash
set -x


REPO_URL="https://x-access-token:${GITHUB_TOKEN}@github.com/
    zakaryadev03/Full-stack-DevOps.git"
MANIFEST_PATH="k8s"
TMP_REPO_PATH="/tmp/temp_repo"

# Git bot configuration
GIT_USER_NAME="Azure DevOps CI"
GIT_USER_EMAIL="ci-bot@dev.azure.com"

# --- Argument Validation ---
if [ "$#" -ne 3 ]; then
    echo "ERROR: Illegal number of parameters."
    echo "Usage: $0 <service-name> <docker-image-name> <image-tag>"
    echo "Example: $0 api-gateway zakaryab2003/fsdevopsapi v1.0.1"
    exit 1
fi

# Assign arguments to readable variables
SERVICE_NAME="$1" # e.g., "api-gateway"
IMAGE_NAME="$2"   # e.g., "zakaryab2003/fsdevopsapi"
IMAGE_TAG="$3"    # e.g., "latest" or "v1.0.1" or $CI_COMMIT_SHA

# Define the full path to the manifest file
MANIFEST_FILE="$MANIFEST_PATH/$SERVICE_NAME-deployment.yaml"

# --- Git Operations ---

# Cleanup previous runs just in case
rm -rf "$TMP_REPO_PATH"

# Clone the git repository
git clone --depth 1 "$REPO_URL" "$TMP_REPO_PATH"
cd "$TMP_REPO_PATH"


# Configure Git user for this commit
git config user.name "$GIT_USER_NAME"
git config user.email "$GIT_USER_EMAIL"
```

```
42
43 # --- Manifest Update ---
44
45 # Check if the file exists before trying to modify it
46 if [ ! -f "$MANIFEST_FILE" ]; then
47     echo "ERROR: Manifest file not found at $MANIFEST_FILE"
48     exit 1
49 fi
50
51 echo "Updating $MANIFEST_FILE to use image $IMAGE_NAME:$IMAGE_TAG"
52 sed -i "s|image: .*|image: $IMAGE_NAME:$IMAGE_TAG|g" "$MANIFEST_FILE"
53
54 # --- Git Commit & Push ---
55
56 # Add only the modified file
57 git add "$MANIFEST_FILE"
58
59 # Commit the changes with a specific message
60 git commit -m "CI: Update $SERVICE_NAME image to $IMAGE_NAME:
       $IMAGE_TAG"
61
62 # Push the changes back to the repository (to the main branch)
63 git push
64
65 # Cleanup: remove the temporary directory
66 echo "Successfully pushed manifest update. Cleaning up."
67 rm -rf "$TMP_REPO_PATH"
```

Listing 2.4 – Script de mettre a jour les manifests k8s

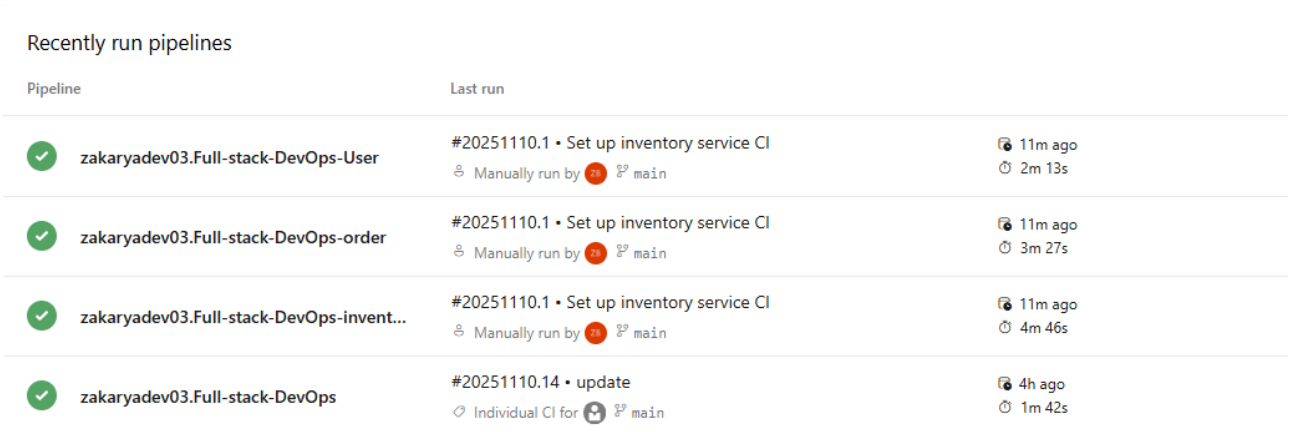

FIGURE 2.9 – Les pipelines pour les services

2.3 Conception de l'Infrastructure de Déploiement et de Supervision

Le déploiement continu est géré par **Argo CD**, un outil de GitOps "pull-based" (basé sur l'extraction). Son rôle est de surveiller en permanence notre dépôt Git (la "source de vérité") et de s'assurer que l'état de notre cluster Kubernetes correspond exactement à ce qui est défini dans les fichiers de manifestes.

Installation d'Argo CD

La première étape consiste à installer Argo CD dans son propre namespace (argocd) pour l'isoler de nos applications et de nos outils de supervision.

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/
    argo-cd/stable/manifests/install.yaml
```

Configuration de l'Application GitOps

Une fois Argo CD installé et accessible via son interface utilisateur (UI), nous avons configuré une nouvelle "Application" pour gérer notre projet. Cela consiste à dire à Argo CD : "Regarde ce dépôt Git et déploie-le à cet endroit".

Les paramètres clés de la configuration sont les suivants :

— **Source (Source de vérité) :** Le dépôt Git de notre projet et le chemin d'accès (Path) pointant vers notre dossier de manifestes (k8s).

— **Destination :** Le cluster Kubernetes sur lequel Argo CD est lui-même installé (kubernetes.default.svc) et le namespace de destination où les manifestes doivent être déployés, c'est-à-dire sre-lab.

— **Politique de Synchronisation :** Nous avons activé le **sync automatique** (automated).

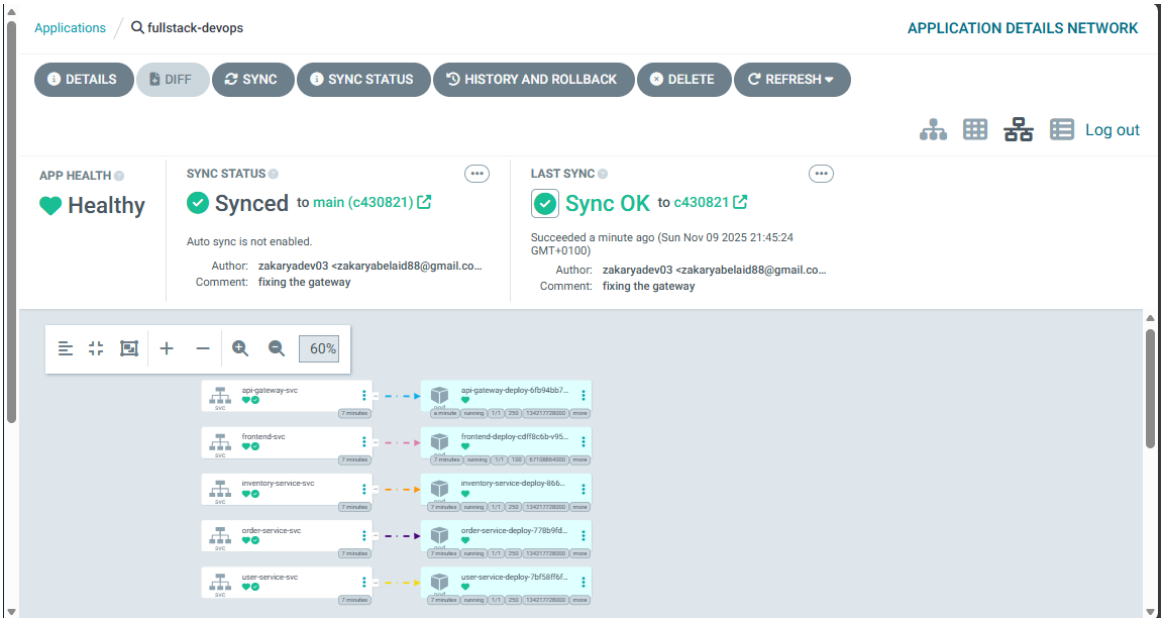

FIGURE 2.10 – Deployemnt d'application

Avec cette configuration, la boucle GitOps est complète. Lorsque notre pipeline CI (Azure DevOps) pousse un nouveau commit dans le dossier k8s/ (par exemple, pour changer un tag d'image image : ... :72), Argo CD le détecte. Grâce au "sync automatique", il compare l'état du cluster à l'état de Git, voit la différence, et applique automatiquement le changement au cluster.

2.3.1 Mise en place de la Stack d'Observabilité

Pour atteindre une supervision complète de notre application, une stack d'observabilité composée d'outils spécialisés a été déployée. L'objectif est de collecter trois types de télémétrie (les "trois piliers de l'observabilité") :

- **Métriques (Metrics) :** Des mesures numériques aggrégées (ex : http_requests_total). Collectées par **Prometheus**.
- **Logs (Logs) :** Des événements textuels horodatés (ex : [Gateway] Request to...). Collectés par **Loki** et **Promtail**.
- **Traces (Traces) :** Une vue détaillée du cycle de vie d'une requête à travers plusieurs services. Collectées par **Jaeger**.

Grafana sert d'interface utilisateur unifiée pour visualiser ces trois sources de données.

Installation via Helm

L'installation de ces outils a été entièrement gérée par **Helm**, le gestionnaire de paquets pour Kubernetes. Helm simplifie le déploiement d'applications complexes en "packagant" tous les manifestes Kubernetes, configurations et dépendances nécessaires dans un seul "chart".

La première étape consiste à ajouter les dépôts Helm officiels pour chaque outil :

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo add grafana https://grafana.github.io/helm-charts
helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
helm repo update
```

Ensuite, chaque composant de la stack a été installé via une commande helm install dédiée, en spécifiant le namespace approprié :

```
helm install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring
helm install loki grafana/loki -f values.yaml --namespace logging
helm upgrade --install promtail grafana/promtail --values values1.yaml --namespace logging
helm install jaeger jaegertracing/jaeger-all-in-one --namespace tracing
helm install grafana grafana/grafana --namespace monitoring
```

Configuration des sources de données (Data Sources) dans Grafana

Une fois tous les services déployés, l'étape finale consiste à configurer l'interface Grafana pour qu'elle puisse interroger ces différentes sources de données.

Depuis l'interface d'administration de Grafana, nous avons ajouté trois "Data Sources" :
.

— **Prometheus :** http ://prometheus-server.monitoring.svc.cluster.local :80
— **Alert manager :** http ://prometheus-alertmanager.monitoring.svc.cluster.local :9093
— **Jaeger :** http ://jaeger-all-in-one.tracing.svc.cluster.local :16686
— **Loki :** http ://loki-gateway.logging.svc.cluster.local :80

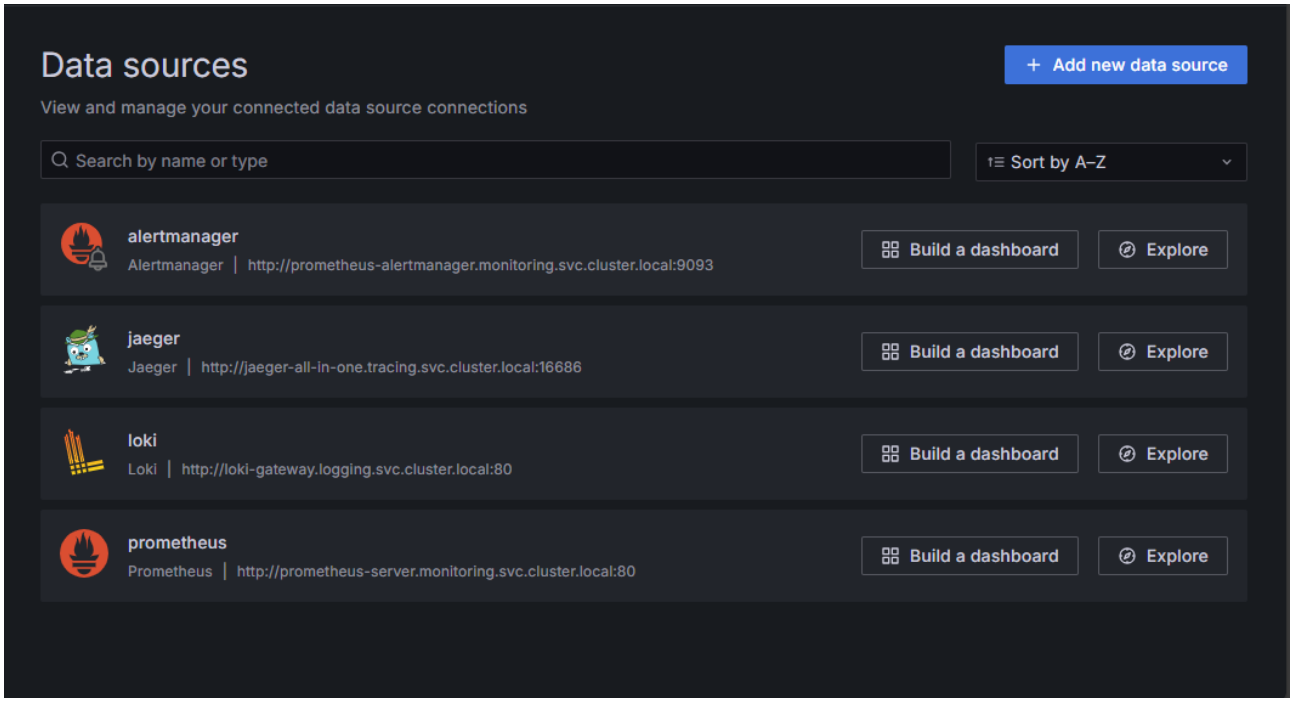

FIGURE 2.11 – Data sources

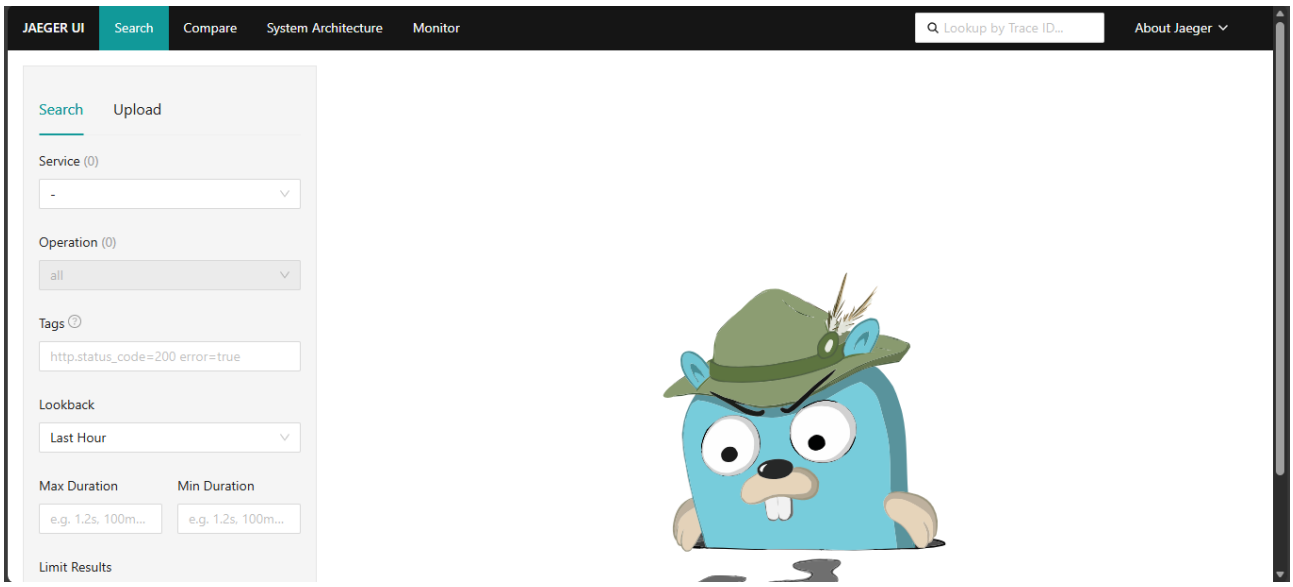

FIGURE 2.12 – Jaeger UI

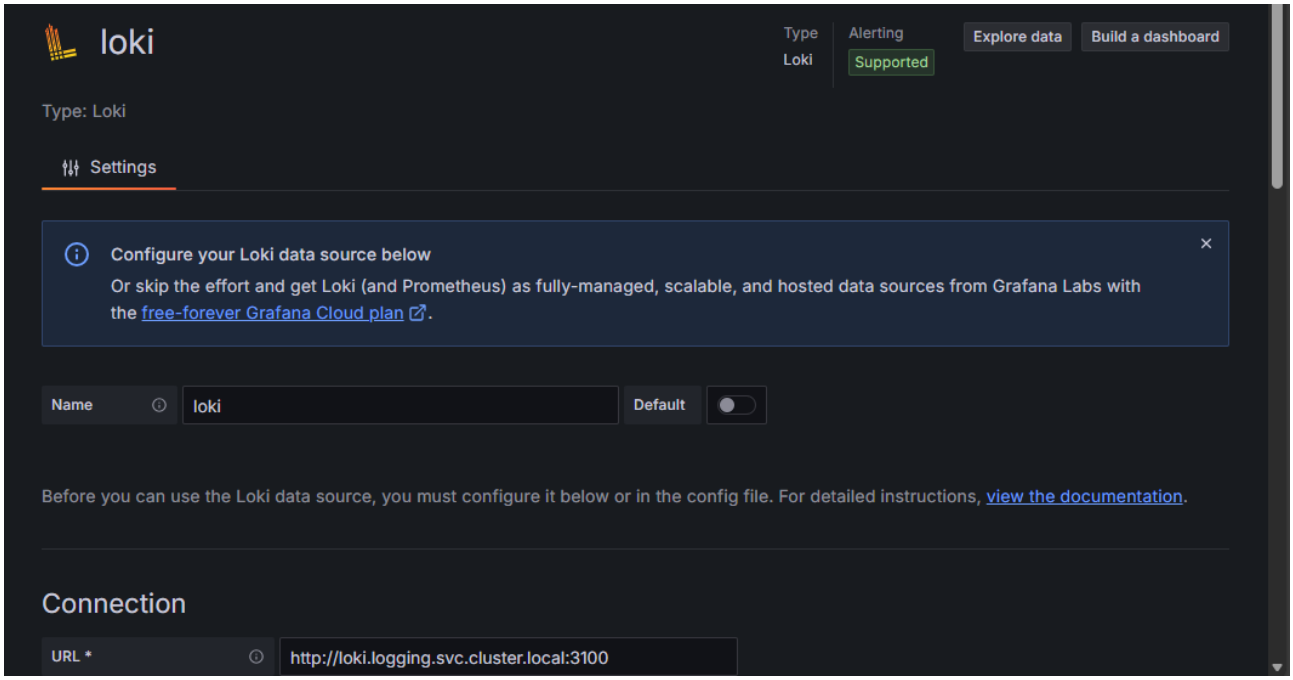

FIGURE 2.13 – Loki

Collecte des traces

On a testé jaeger en simulant des requetes entre les services de notre application.

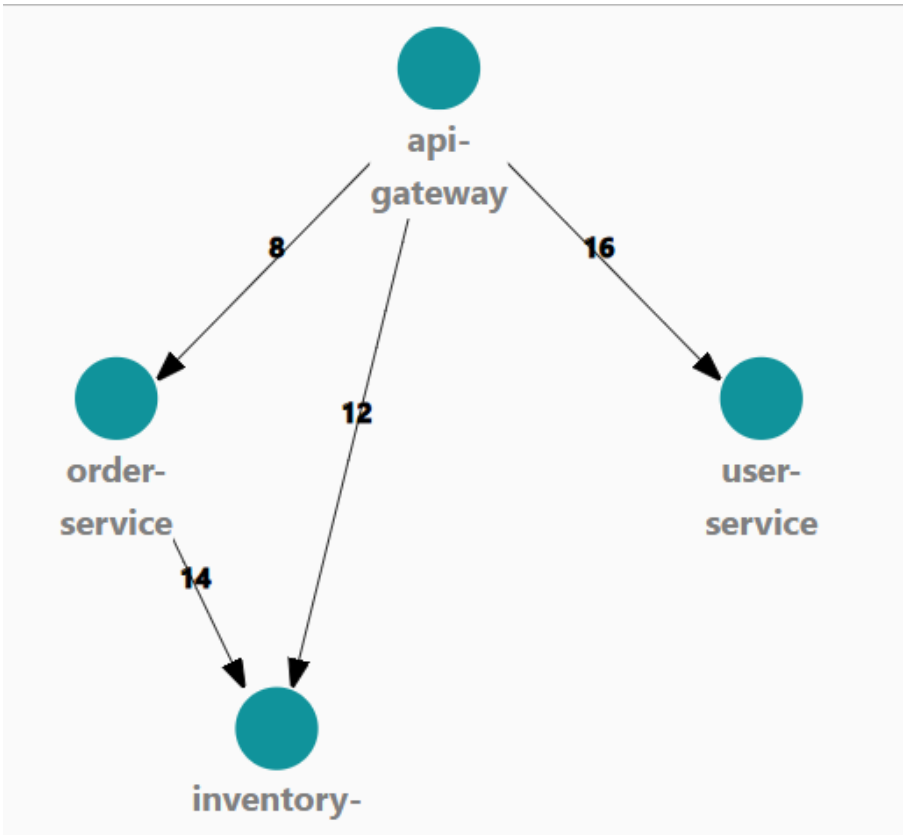

FIGURE 2.14 – Traces d'application

Collecte des logs

Avant d'utiliser Loki, il faut configurer Promtail. Promtail est un agent qui transfère le contenu des journaux locaux vers une instance privée de Grafana Loki. Par défaut, promtail ne collecte pas les logs des namespaces ajouté, il faut les preciser dans le fichier de configuration yaml et faire une mettre a jour a l'aide du commande *install –upgrade*.

```
config:
  clients:
    # This is the Loki "push" URL and tenant ID
    - url: http://loki-gateway.logging.svc.cluster.local:80/loki/api/v1/push
      tenant_id: 1

  scrape_configs:
    # This is the default job to scrape Kubernetes pods
    - job_name: kubernetes-pods
      kubernetes_sd_configs:
        - role: pod
          namespaces:
            # This is the key you found:
            # We are explicitly telling Promtail which namespaces to scrape.
            names:
              - "sre-lab"
              - "logging"

      # This part adds all the useful Kubernetes labels (app, pod, etc.)
      pipeline_stages:
        - docker: {}
      relabel_configs:
        - source_labels: [__meta_kubernetes_pod_node_name]
          target_label: __host__
        - action: replace
          source_labels: [__meta_kubernetes_pod_name]
          target_label: pod
        - action: replace
          source_labels: [__meta_kubernetes_pod_namespace]
          target_label: namespace
        - action: replace
          source_labels: [__meta_kubernetes_pod_container_name]
          target_label: container
        - action: labelmap
          regex: __meta_kubernetes_pod_label_(.+)
```

Listing 2.5 – Fichier de configuration

2.3.2 Visualisation et Tableaux de Bord

Grafana est l'outil central de notre plateforme de supervision. Il sert d'interface unifiée qui se connecte à nos différentes sources de données (Prometheus, Loki et Jaeger) pour agréger et visualiser l'ensemble de la télémétrie sur une seule interface.

Pour ce projet, une approche hybride de "dashboarding" a été utilisée :

— **Dashboards Prédéfinis (Community) :** Pour la supervision générale de l'infrastructure Kubernetes (état des nuds, utilisation CPU/Mémoire des pods, etc.), des dashboards standards ont été importés depuis la communauté Grafana. Ces dashboards (comme "Kubernetes All-in-One") fonctionnent "clés en main" avec les métriques exposées par le chart kube-prometheus-stack.

— **Dashboard Personnalisé :** Pour superviser la logique métier et les performances de notre application, un dashboard personnalisé a été créé. Ce dashboard cible spécifiquement les métriques que nous avons nous-mêmes définies dans notre code (ex : orders_total, inventory_stock_level_total) et filtre les données sur notre namespace (sre-lab).

Importation du Dashboard Applicatif (JSON)

Pour garantir la portabilité et la reproductibilité de notre dashboard applicatif, il n'a pas été créé manuellement dans l'interface, mais importé directement depuis un fichier **JSON**. Cette approche permet de versionner notre dashboard dans Git, tout comme notre code ou nos manifestes.
Les fichiers .json sont dans le dossier grafana avec le code source

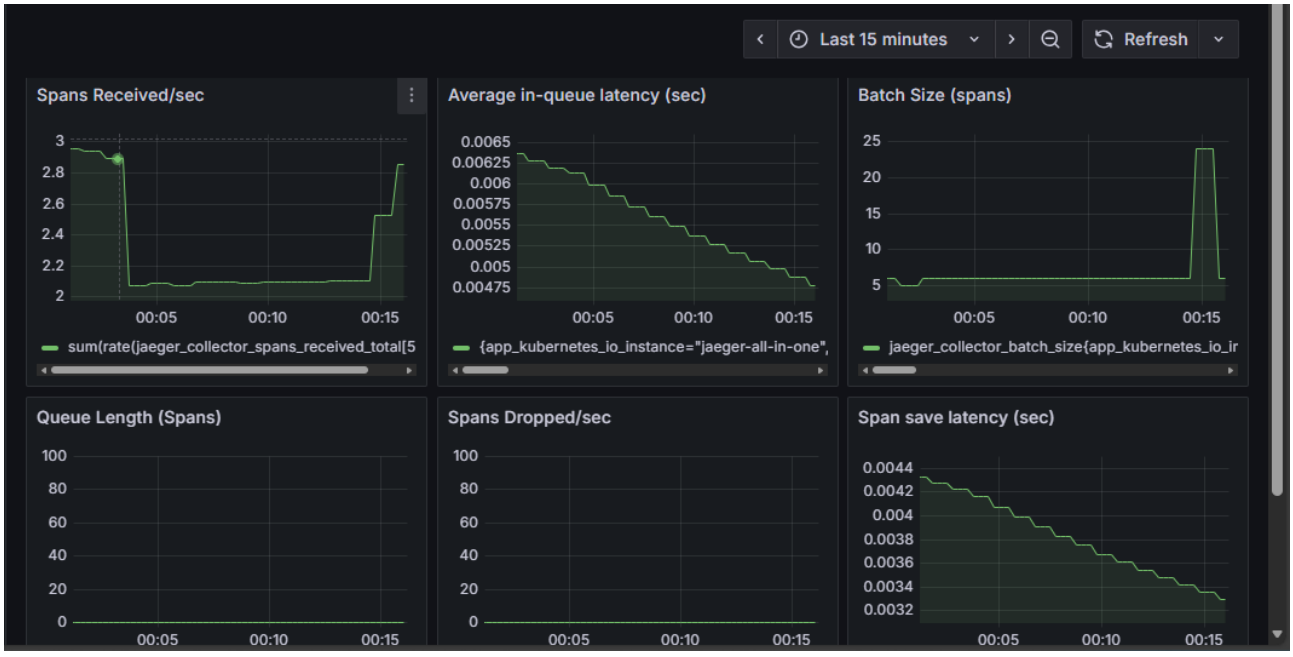

FIGURE 2.15 – Dashboard Jaeger

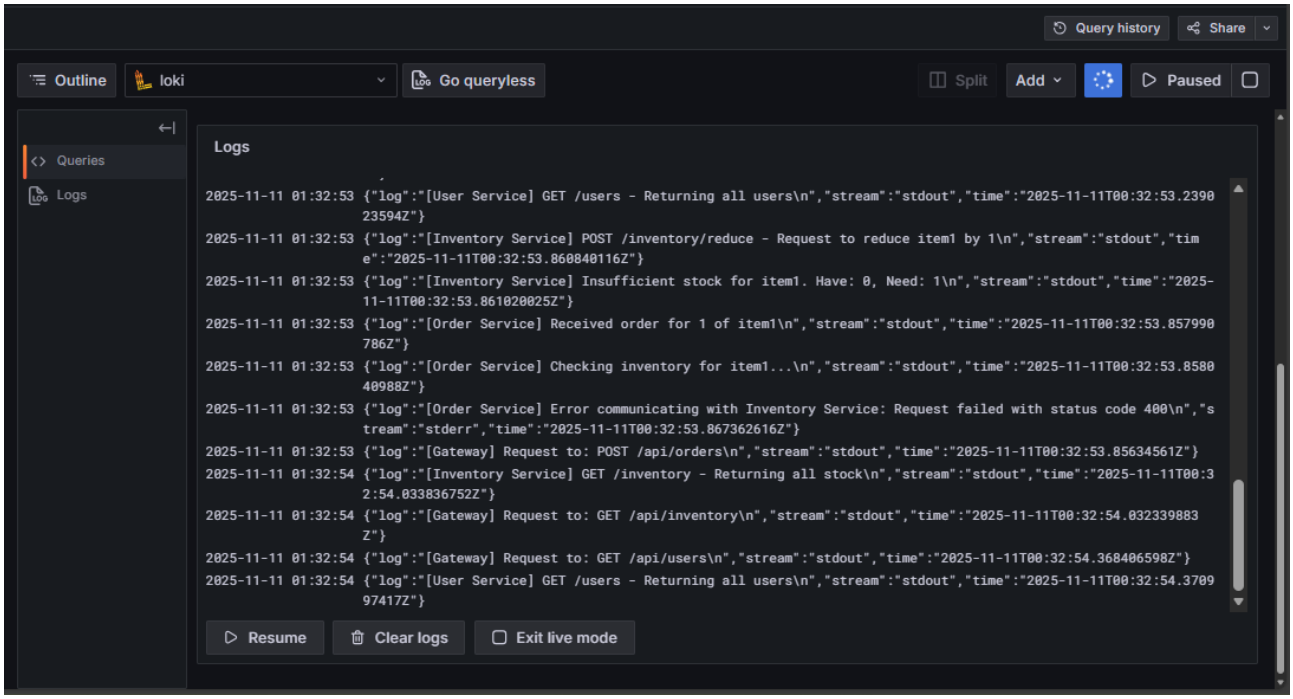

FIGURE 2.16 – Les logs d'application e-commerce

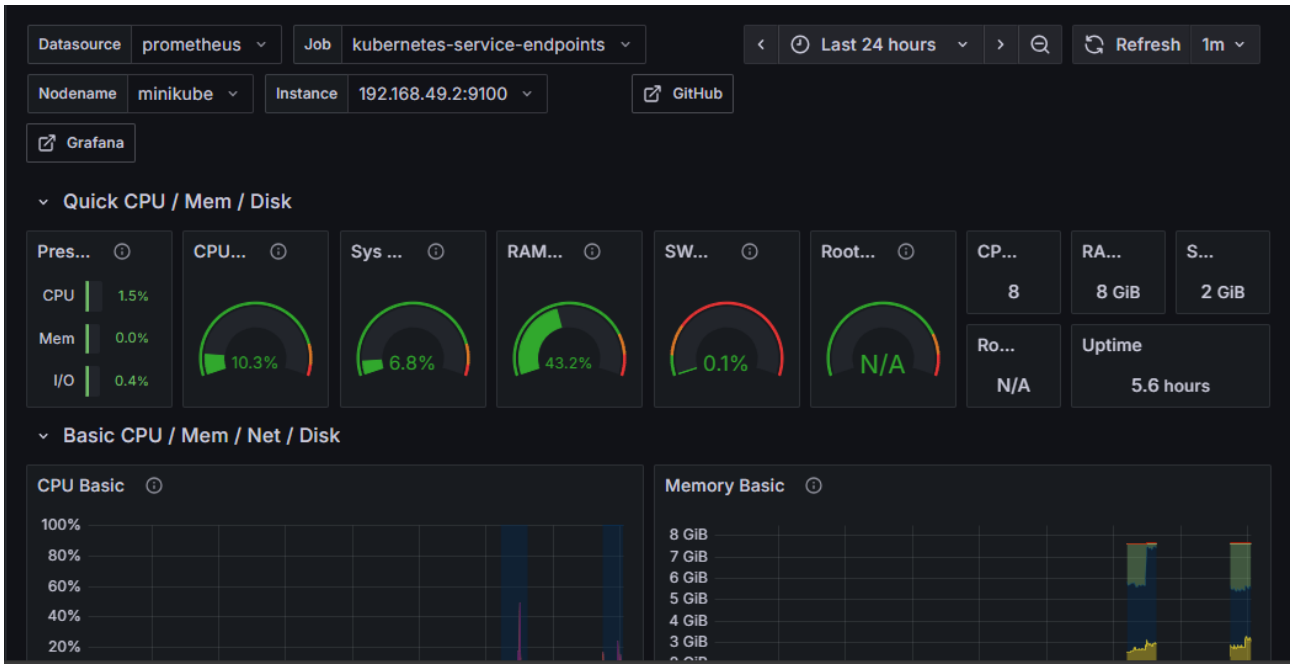

FIGURE 2.17 – metrics des nodes

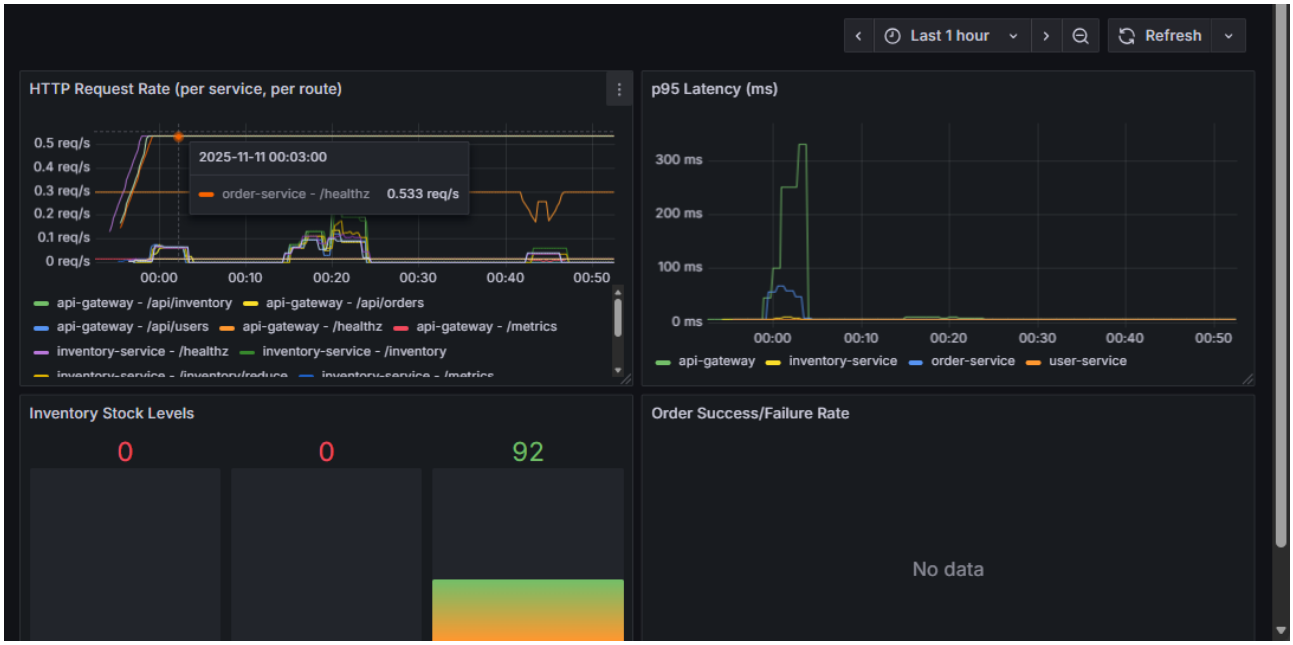

FIGURE 2.18 – Visualisation des metrics d'application

2.4 Configuration d' Alertmanager

La mise en place d'une plateforme de supervision ne se limite pas à la visualisation des données ; elle doit également être capable de notifier de manière proactive les équipes en cas d'anomalie. Cette section décrit la configuration de la chaîne d'alertage complète, de la détection du problème dans Prometheus à la notification sur Slack, en utilisant le "Unified Alerting" de Grafana qui pilote Alertmanager.

Étape 1 : Création de la Règle d'Alerte

La première étape consiste à définir la condition qui doit déclencher une alerte. Pour ce projet, nous avons implémenté une règle fondamentale : **"Service Indisponible"**. Une nouvelle règle d'alerte a été créée dans Grafana, basée sur la métrique up que Prometheus génère automatiquement pour chaque service qu'il supervise (1 si le service est joignable, 0 s'il est injoignable).

- **Nom de la règle :** Service Down
- **Source de données :** Prometheus
- **Requête (Query) :** upnamespace="sre-lab". Cette requête vérifie le statut de tous les services dans notre namespace applicatif.
- **Condition :** L'alerte se déclenche (Firing) si la condition WHEN last() OF B IS BELOW 1 est vraie. Cela signifie qu'une alerte est créée dès qu'un service (n'importe lequel) a une valeur up inférieure à 1.
- **Évaluation :** L'alerte est évaluée toutes les 1m (1 minute) et doit être active pendant 1m (for : 1m) avant de se déclencher. Cela évite les "fausses alertes" (flapping) lors d'un simple redémarrage de pod.

— **Labels :** Un label crucial severity = critical a été ajouté à la règle. Ce label est la clé qui permettra à notre politique de notification de router cette alerte.

Étape 2 : Création du Point de Contact

Ensuite, nous devons indiquer à Grafana où envoyer les notifications. Nous avons utilisé la fonctionnalité "Incoming Webhooks" de Slack pour obtenir une URL unique. Dans Grafana,

sous l'onglet "Alerting" → "Contact points", un nouveau point de contact a été créé :

— **Nom :** Slack

— **Intégration :** Slack

— **Webhook URL :** L'URL sécurisée fournie par Slack a été collée ici.

— **Test :** Le bouton "Test" de Grafana a été utilisé pour envoyer un message de test, validant ainsi la connexion entre Grafana et Slack.

Étape 3 : Création de la Politique de Notification

Enfin, il faut connecter la Règle d'Alerte au Point de Contact. C'est le rôle de la "Notification Policy" (Politique de Notification). Une nouvelle politique a été créée pour intercepter nos alertes

critiques :

— **Matcher (Filtre) :** La politique est configurée pour ne s'appliquer qu'aux alertes ayant le label severity avec la valeur critical.

— **Contact Point (Destination) :** Le point de contact de destination a été défini sur SRE Slack Channel (celui créé à l'étape 2).

Cette configuration implémente la logique de routage : **SI** une alerte se déclenche **ET** qu'elle a le label severity=critical, **ALORS** envoie la notification au canal Slack.

Étape 4 : Test de la boucle d'alertage complète

Pour valider l'ensemble de la chaîne, une panne a été simulée manuellement, comme requis par le laboratoire. Nous avons volontairement "tué" l'API Gateway en mettant à l'échelle son déploiement à 0 répliques.

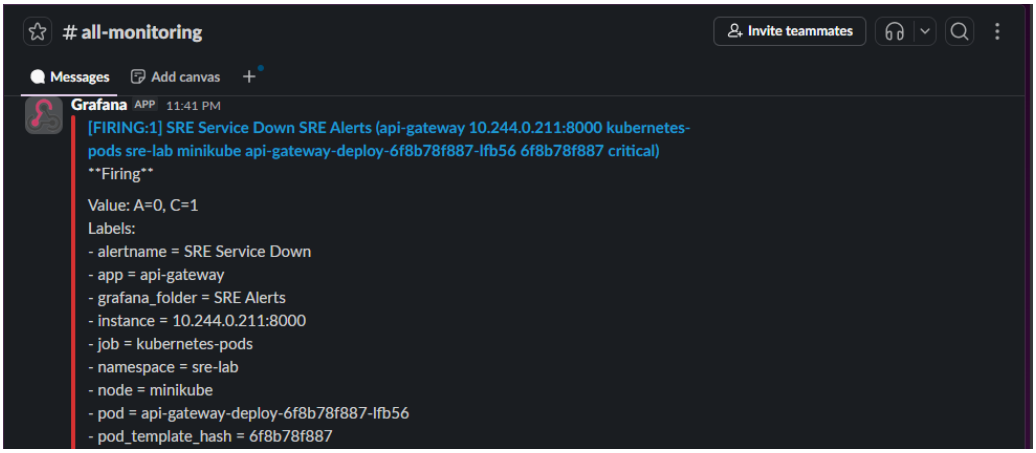

FIGURE 2.19 – Alert sur slack

Chapitre 3

Conclusion et Perspectives

Ce projet de laboratoire a porté sur la conception, l'implémentation et le déploiement d'une plateforme complète d'observabilité et de GitOps, appliquée à une architecture microservices déployée sur Kubernetes. L'objectif n'était pas seulement de déployer une application, mais de construire l'infrastructure de supervision, de fiabilité et d'automatisation de niveau production qui l'entoure.

Lensemble des travaux réalisés a permis de :

— Concevoir et déployer une application de microservices Node.js (API Gateway, Order Service, etc.) conteneurisée avec Docker et servant de cas d'étude pour la supervision.

— Implémenter une stack d'observabilité complète ("Les 3 Piliers") : **Prometheus** (métriques), **Loki/Promtail** (logs), et **Jaeger** (traces), le tout centralisé dans **Grafana**.

— Instrumenter en profondeur le code applicatif avec **OpenTelemetry** (pour les traces), **prom-client** (pour les 4 types de métriques Prometheus) et des console.log structurés (pour Loki).

— Construire une chaîne CI/CD GitOps entièrement automatisée : **Azure DevOps** (CI) pour builder, tagguer et pousser les images Docker, et **Argo CD** (CD) pour le déploiement "pull-based" sur Kubernetes.

— Configurer une chaîne d'alertage proactive avec **Grafana Unified Alerting** et **Alertmanager**, intégrant des notifications critiques (ex : "Service Down") directement sur Slack via webhook.

Difficultés rencontrées :

— **La configuration de la stack d'observabilité :** L'interfaçage des outils a été complexe, notamment :

  — **Loki :** Le débogage du "No data" causé par l'absence de Promtail, puis la configuration correcte de values.yaml pour le tenant_id et le filtrage des namespaces.

  — **Jaeger :** La gestion de l'incompatibilité de protocole (OTLP vs. Thrift), le port (14268 vs. 4318) et les dépendances npm manquantes (Resource is not a constructor).

— **L'authentification CI/CD :** La configuration du GITHUB_TOKEN (Personal Access Token) entre Azure DevOps et le dépôt GitHub pour permettre au script shell de "committer" les changements de tag d'image.

— **La gestion du cache d'images Kubernetes :** Le diagnostic du problème où les services (sauf api-gateway) n'implémentaient pas le traçage, causé par l'utilisation du tag :latest qui restait en cache sur le nud Minikube, nécessitant un rollout restart ou l'utilisation de tags uniques (via le Build.BuildId).

— **Le débogage PromQL :** La correction des requêtes Grafana suite au "brouillage" des labels (ex : job="kubernetes-pods" vs. app="api-gateway") et la correction du bug [object Object] dans metrics.js.

Perspectives dévolution :

— **Implémenter des SLO/SLI formels :** Utiliser nos métriques (ex : p95 latency) pour définir des Objectifs de Niveau de Service (SLO) formels (ex : "99% des requêtes de l'API Gateway doivent être < 200ms") et créer des alertes basées sur ces SLO.

— **Corrélation des données :** Enrichir les dashboards Grafana pour lier les "3 piliers" : par exemple, en cliquant sur un pic de latence (métrique), afficher automatiquement les logs et les traces de cet instant précis.

— **Infrastructure as Code (IaC) :** Automatiser le déploiement du cluster lui-même et de l'ensemble de la stack de supervision (Prometheus, Loki, etc.) à l'aide d'outils comme Terraform.

— **Métriques basées sur les logs :** Utiliser Loki pour analyser les logs (ex : compter les 404 ou 500) et générer de nouvelles métriques qui peuvent être ensuite visualisées et utilisées pour l'alertage dans Prometheus.

En conclusion, ce projet de laboratoire a permis d'acquérir une expérience pratique significative dans la construction d'une plateforme d'ingénierie de fiabilité de bout en bout, tout en consolidant les compétences en GitOps, orchestration (Kubernetes), instrumentation applicative et observabilité complète.

Références

— Argo CD. (n.d.). *Argo CD - Declarative GitOps CD for Kubernetes*. Retrieved November 13, 2025, from https ://argo-cd.readthedocs.io/

— Azure DevOps. (n.d.). *What is Azure DevOps ?*. Microsoft. Retrieved November 13, 2025, from https ://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops

— Grafana Labs. (n.d.). *Grafana Documentation*. Retrieved November 13, 2025, from https ://-grafana.com/docs/grafana/latest/

— Grafana Labs. (n.d.). *Loki : Like Prometheus, but for logs*. Retrieved November 13, 2025, from https ://grafana.com/docs/loki/latest/

— Jaeger Tracing. (n.d.). *Jaeger : open source, end-to-end distributed tracing*. Retrieved November 13, 2025, from https ://www.jaegertracing.io/docs/

— Kubernetes. (n.d.). *Production-Grade Container Orchestration*. Retrieved November 13, 2025, from https ://kubernetes.io/docs/home/

— OpenTelemetry. (n.d.). *An observability framework for cloud-native software*. Retrieved November 13, 2025, from https ://opentelemetry.io/docs/

— Prometheus. (n.d.). *Prometheus - Monitoring system time series database*. Retrieved November 13, 2025, from https ://prometheus.io/docs/introduction/overview/