SOMMAIRE

[Introduction 2](#_heading=h.gjdgxs)

[1.](#_heading=h.30j0zll) Compréhension des Microservices 2

[2.](#_heading=h.1fob9te) Compréhension du DevOps 3

[3.](#_heading=h.3znysh7) Intégration des Outils DevOps 4

[4.](#_heading=h.2et92p0) Intégration des outils de monitoring 5

[a.](#_heading=h.tyjcwt) Contexte et Importance du Monitoring 5

[b.](#_heading=h.3dy6vkm) Première Intégration : Prometheus + Grafana + Loki + Tempo 5

[c.](#_heading=h.1t3h5sf) Seconde Intégration : Prometheus + Grafana + ELK Stack + Zipkin 6

[d.](#_heading=h.4d34og8) Comparaison des Performances 7

[5.](#_heading=h.2s8eyo1) Conclusion 7

[6.](#_heading=h.17dp8vu) Références 8

# Introduction

Dans le cadre de notre travail de recherche en master, nous nous sommes penchés sur le sujet intitule **approches DevOps et surveillance distribuée des applications basées sur les architectures** microservices. Cette présentation vise à explorer ces concepts fondamentaux et à illustrer leur mise en pratique à travers le développement d'une application de **gestion des notes universitaires**.

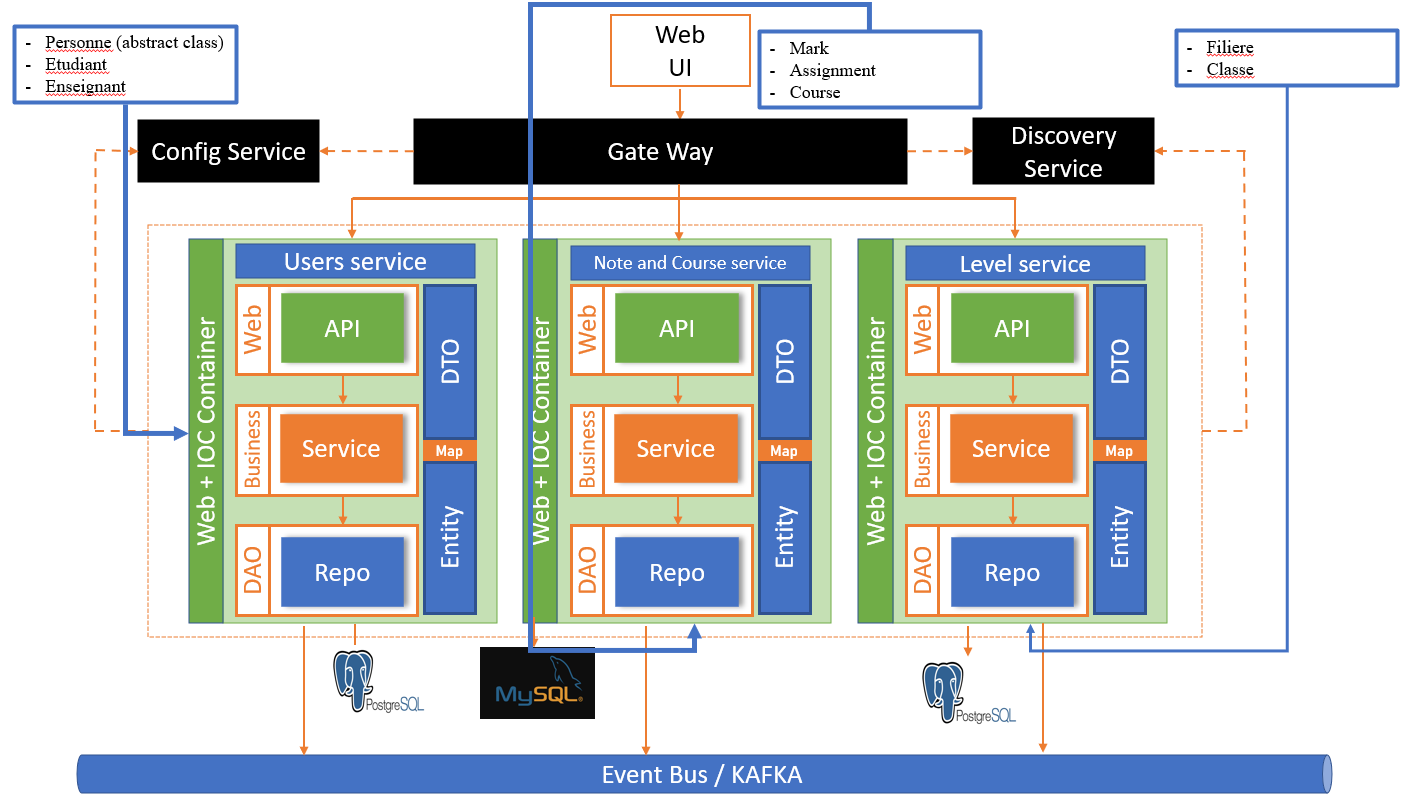
L'objectif principal de cette recherche est de comprendre comment les microservices et le DevOps peuvent **améliorer l'efficacité et la fiabilité** des applications distribuées. Pour ce faire, nous avons entrepris une phase expérimentale qui s'est déroulée en plusieurs étapes : la compréhension des microservices, leur implémentation dans une application concrète, et l'intégration d'un pipeline DevOps pour automatiser le développement et le déploiement.

# Compréhension des Microservices

Les **microservices** représentent une approche architecturale où une application est construite comme un ensemble de services indépendants et interconnectés. Chaque service, ou microservices, est responsable d'une tâche spécifique et communique avec les autres via des protocoles légers tels que **HTTP** ou des **mécanismes de messagerie**. Les avantages des microservices incluent la **scalabilité indépendante**, la **facilité de déploiement continu**, et la **résilience aux pannes**.

Dans le cadre de notre travail, nous avons développé une application de gestion des notes universitaires reposant sur une architecture de microservices. Trois principaux services ont été mis en œuvre :

* Un service de gestion des notes et des cours.
* Un service pour gérer les utilisateurs (étudiants et enseignants).
* Un service dédié à la gestion des filières et des programmes d'études.



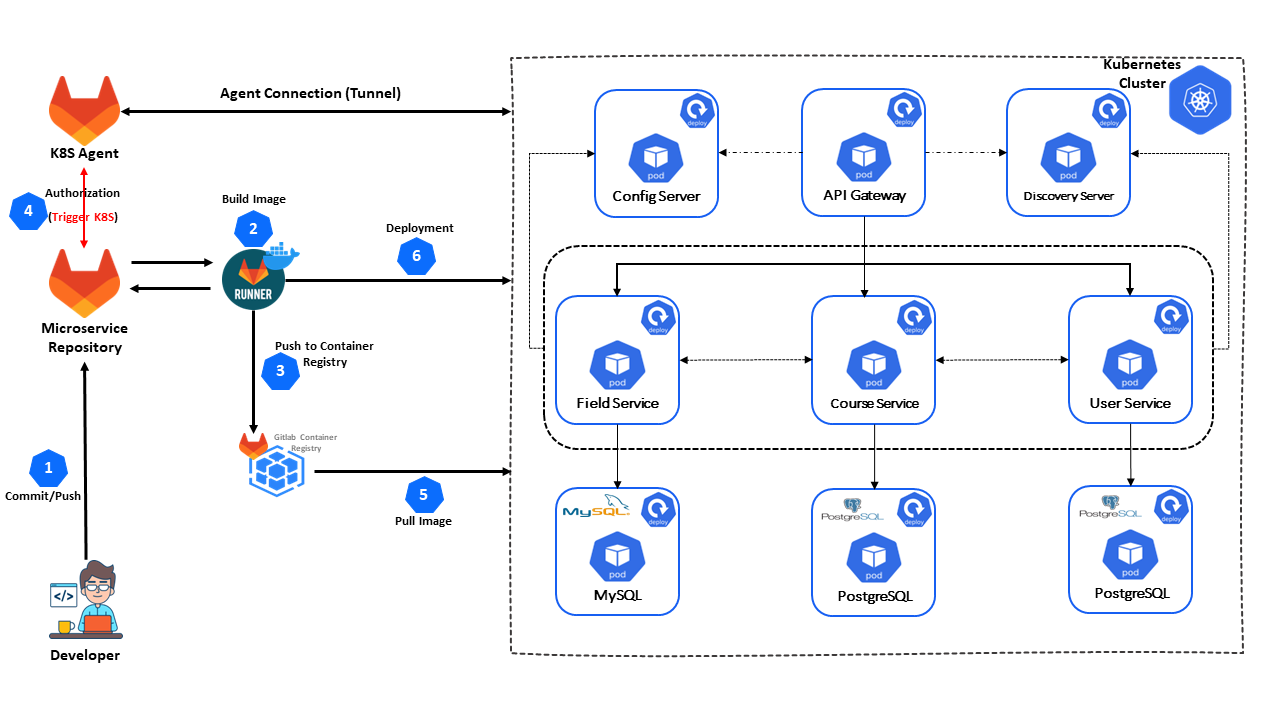
Chaque microservice a été conçu pour être autonome, avec sa propre base de données et son API dédiée, permettant ainsi une évolutivité et une maintenance plus efficaces.

# Compréhension du DevOps

Le DevOps est une culture et un ensemble de pratiques visant à rapprocher les équipes de développement (Dev) et d'opérations (Ops) pour automatiser et accélérer le processus de développement logiciel. Cette approche favorise la collaboration, l'intégration continue, le déploiement continu et une gestion efficace des infrastructures.

Pour notre projet, nous avons intégré un pipeline DevOps complet pour automatiser les différentes étapes du cycle de vie logiciel. Nous avons utilisé plusieurs outils clés :

* **Docker** : Pour la conteneurisation des microservices, permettant ainsi un environnement de développement cohérent et portable.
* **Kubernetes** : Pour l'orchestration des conteneurs, facilitant le déploiement, la mise à l'échelle et la gestion des microservices.
* **GitLab CI/CD** : Pour l'intégration continue et le déploiement continu des microservices, assurant une livraison rapide et fiable des nouvelles fonctionnalités.
* **GitLab Runner** : Pour l'exécution des pipelines CI/CD.
* **Registry** : Pour la gestion centralisée des images Docker.
* **Agent** : Pour l'exécution de tâches spécifiques dans le pipeline.



Cette intégration DevOps a permis d'automatiser les tests, le déploiement et la surveillance des microservices, favorisant ainsi une approche itérative et agile du développement logiciel.

# Intégration des Outils DevOps

L'intégration des outils DevOps dans le développement et le déploiement de notre application a été essentielle pour automatiser les processus et garantir la cohérence des déploiements. Chaque outil a joué un rôle spécifique dans le pipeline DevOps, contribuant ainsi à améliorer l'efficacité et la fiabilité de notre système.

* **Docker :** La conteneurisation avec Docker a permis d'encapsuler chaque microservice et ses dépendances dans des environnements isolés, assurant ainsi la portabilité entre les différentes étapes du cycle de vie logiciel. Les conteneurs Docker ont simplifié le déploiement en fournissant des images prêtes à être exécutées dans n'importe quel environnement compatible.
* **Kubernetes :** En utilisant Kubernetes pour l'orchestration des conteneurs, nous avons pu déployer et gérer nos microservices de manière évolutive et résiliente. Kubernetes a automatisé des tâches telles que le dimensionnement automatique en fonction de la charge, la gestion des mises à jour et le redémarrage des microservices en cas de défaillance.
* **GitLab CI/CD :** L'intégration continue (CI) et le déploiement continu (CD) avec GitLab ont été au cœur de notre pipeline DevOps. Chaque fois qu'un développeur a effectué une modification du code source, GitLab CI/CD a déclenché des workflows automatisés comprenant les builds des images dockers et les déploiements sur les environnements de test.
* **GitLab Runner :** Les runners GitLab ont été utilisés pour exécuter les jobs définis dans nos pipelines CI/CD. Ces runners ont permis d'exécuter les builds et les déploiements de manière distribuée, offrant ainsi des performances optimales même avec des charges de travail élevées.
* **Registry :** Nous avons utilisé un registre Docker privé pour stocker nos images de conteneurs, assurant ainsi un contrôle total sur la distribution et la sécurité de nos artefacts logiciels.
* **Agent :** Les agents GitLab ont été configurés pour exécuter des tâches spécifiques dans notre pipeline, comme le déploiement sur des environnements de production ou la collecte de métriques de performance.

# Intégration des outils de monitoring

## Contexte et Importance du Monitoring

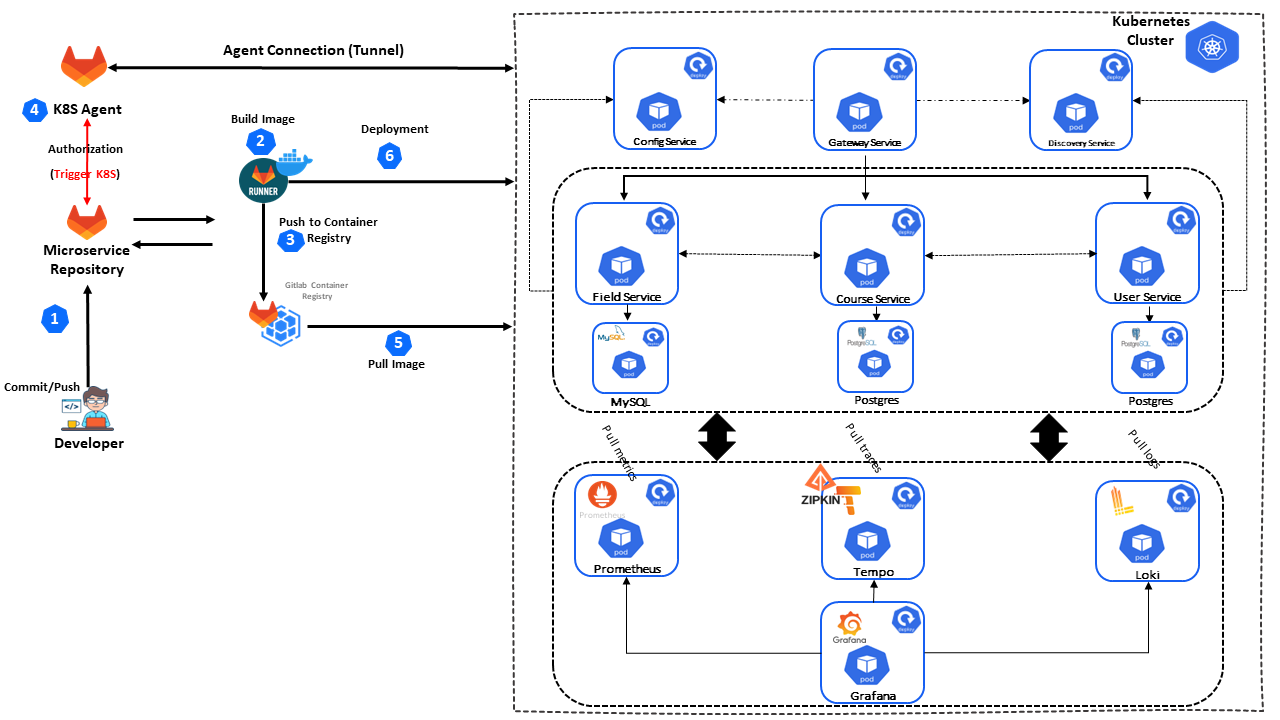
Le monitoring est un élément crucial dans le contexte des architectures distribuées, notamment celles basées sur des microservices. En raison de la nature décentralisée et souvent complexe des microservices, il devient essentiel de disposer d'une visibilité complète sur le comportement de chaque service ainsi que sur l'ensemble du système. Les outils de monitoring permettent non seulement de détecter les anomalies, mais aussi d'optimiser la performance, d'assurer la résilience, et de maintenir la continuité de service. Dans notre projet, nous avons choisi d'intégrer deux stacks principaux de monitoring pour évaluer leur efficacité : **Prometheus + Grafana + Loki + Tempo** et **Prometheu + Grafana + Zipkin + ELK Stack**.

## Première Intégration : Prometheus + Grafana + Loki + Tempo

La première intégration a impliqué la stack **Prometheus + Grafana + Loki + Tempo (Zipkin)**. Ces outils ont été choisis pour leur compatibilité avec les microservices, leur capacité à gérer de grandes quantités de données et leur facilité d'intégration avec Kubernetes et Docker.

1. **Prometheus** : Prometheus est un système de monitoring open-source qui collecte des métriques de performance à partir des services. Il scrappe régulièrement les données exposées par les microservices et stocke ces informations dans une base de données de séries temporelles.
2. **Grafana** : Grafana a été utilisé pour visualiser les données collectées par Prometheus. En créant des tableaux de bord personnalisés, nous avons pu surveiller en temps réel l'état de santé des microservices, les taux de requêtes, les erreurs et d'autres métriques critiques.
3. **Loki** : Pour la gestion des logs, Loki a été intégré. Contrairement à d'autres solutions de logging, Loki est conçu pour fonctionner de manière étroite avec Prometheus, en indexant les logs par labels associés aux métriques. Cela a permis de corréler facilement les logs avec les métriques de performance.
4. **Tempo (Zipkin)** : Enfin, Tempo a été ajouté pour le traçage distribué, permettant de suivre le parcours des requêtes à travers les différents microservices. Cette fonctionnalité est cruciale pour identifier les goulots d'étranglement et optimiser les performances.

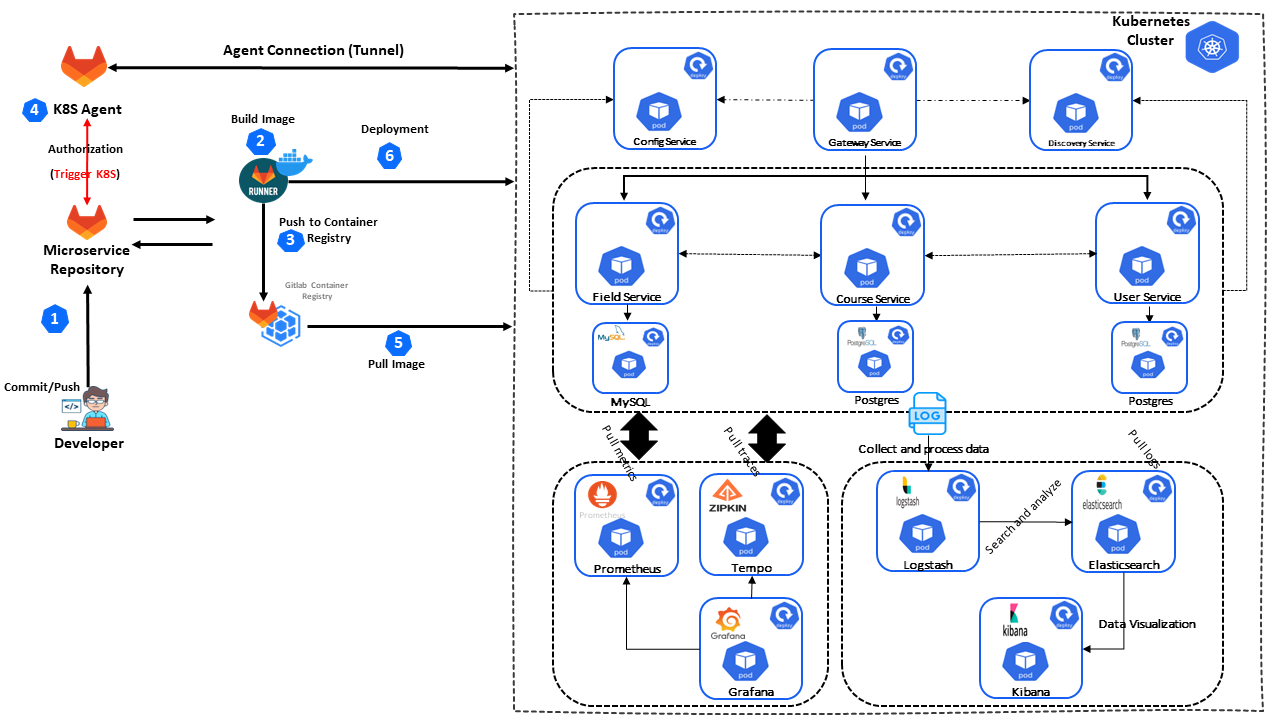
L'intégration de cette stack a grandement amélioré notre capacité à surveiller et diagnostiquer les problèmes au sein de l'architecture microservices.



## Seconde Intégration : Prometheus + Grafana + ELK Stack + Zipkin

La seconde phase d'intégration a impliqué **Prometheus + Grafana + Zipkin + ELK Stack (Elasticsearch, Logstash, Kibana)**. Bien que Prometheus et Grafana soient restés les principaux outils pour la collecte de métriques et la visualisation, l'intégration de l'ELK Stack a permis d'étendre la gestion des logs et l'analyse des données.

1. **Elasticsearch** : Elasticsearch a été utilisé pour stocker et indexer de grandes quantités de données de logs. Sa capacité à effectuer des recherches en texte intégral et à offrir des performances élevées même avec des volumétries importantes en a fait un choix idéal pour notre système.
2. **Logstash** : Logstash a été configuré pour collecter, transformer, et acheminer les logs vers Elasticsearch. Sa flexibilité dans le traitement des données (parsing, filtrage, enrichissement) a permis d'obtenir des logs plus structurés et exploitables.
3. **Kibana** : Kibana a été utilisé pour visualiser les logs stockés dans Elasticsearch. En plus des fonctionnalités de recherche avancée, Kibana a permis de créer des tableaux de bord pour l'analyse des logs, facilitant ainsi la détection d'erreurs.



Cependant, l'intégration de l'ELK Stack a présenté des défis, notamment en termes de configuration et d'optimisation des performances pour les environnements à grande échelle. Les ressources nécessaires pour faire fonctionner Elasticsearch de manière optimale, notamment en termes de mémoire et de CPU, ont nécessité une attention particulière dans notre infrastructure Docker et Kubernetes.

## Comparaison des Performances

En comparant les deux stacks, nous avons observé que **Prometheus + Grafana + Loki + Tempo** offrait une solution plus intégrée et plus simple à configurer pour les besoins de monitoring des microservices, en particulier pour la collecte de métriques et la gestion des logs. En revanche, **ELK Stack** a démontré une plus grande puissance pour la gestion et l'analyse des logs à grande échelle, bien que cela soit accompagné d'une complexité accrue en termes de gestion des ressources.

L'intégration de ces outils a permis d'améliorer considérablement notre capacité à surveiller et à diagnostiquer les performances des microservices dans notre application de gestion des notes universitaires. La combinaison de ces technologies nous a offert une visibilité complète sur l'infrastructure, contribuant ainsi à la fiabilité et à la performance globale du système.

# Conclusion

En conclusion, cette présentation a mis en lumière l'importance des microservices et du DevOps dans le contexte du développement d'applications distribuées. Nous avons exploré les concepts fondamentaux des microservices et du DevOps, en mettant en avant leur application concrète à travers notre projet de gestion des notes.

L'implémentation des microservices nous a permis de concevoir une architecture modulaire, favorisant la scalabilité et la flexibilité de notre application. En parallèle, l'intégration d'un pipeline DevOps complet a automatisé les processus de développement, de test et de déploiement, améliorant ainsi notre efficacité opérationnelle.

# Références

[1]. Giamattei et al., “Monitoring Tools for DevOps and Microservices.”