

# Introduction au DevSecOps

Université de Nantes, MIAGE, M2

© 2025-2026 Bertrand Florat

Ce document est sous licence Creative Commons Attribution - Partage dans les Mêmes  
Conditions 4.0 International (CC BY-SA 4.0)

Révision : 9419494 du 2026-02-04 - Version PDF

API

API

CI/CD



## Contribuer à ce support

Pousser une Pull Request [ici](#), ce support est *as code* (sources en Markdown).



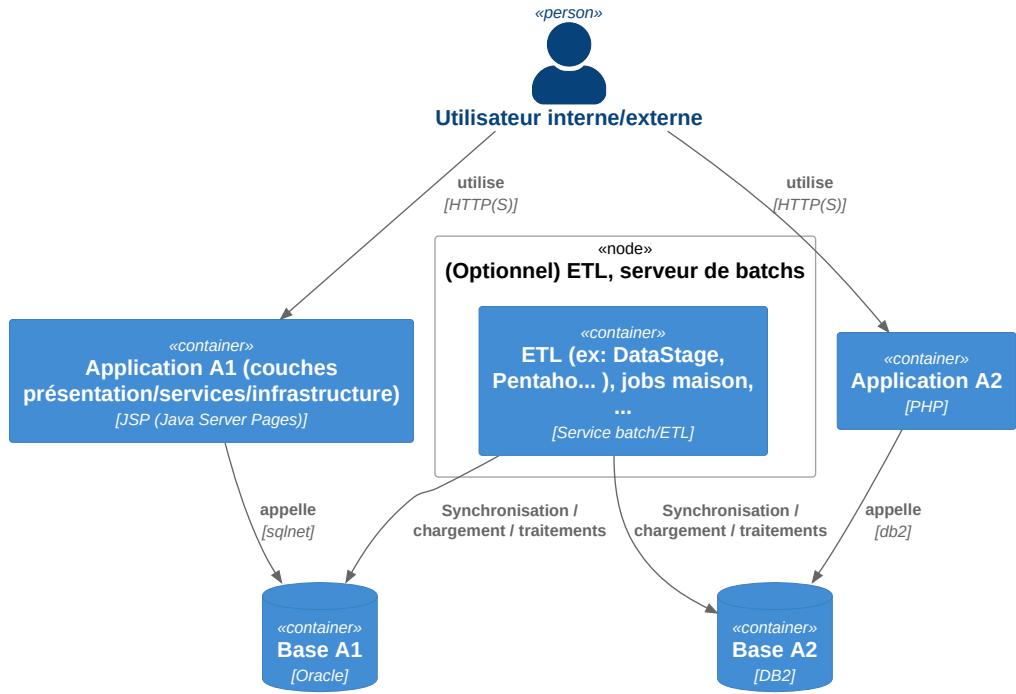
**Note :** Toutes les contributions sur le fond comme sur la forme sont appréciées.



# Séance 1 – Fondations du DevSecOps (1h20)

- 1 - Rappel sur les typologies d'architectures modernes
- 2 - Rappel sur les ENF
- 3 - Contexte des Organisations Traditionnelles
- 4 - DevOps : culture, principes et objectifs
- 5 - Bonnes pratiques DevOps (architecture et livraison)
- 6 - Le DevSecOps

## **1.1 Rappel sur les typologies d'architecture modernes**

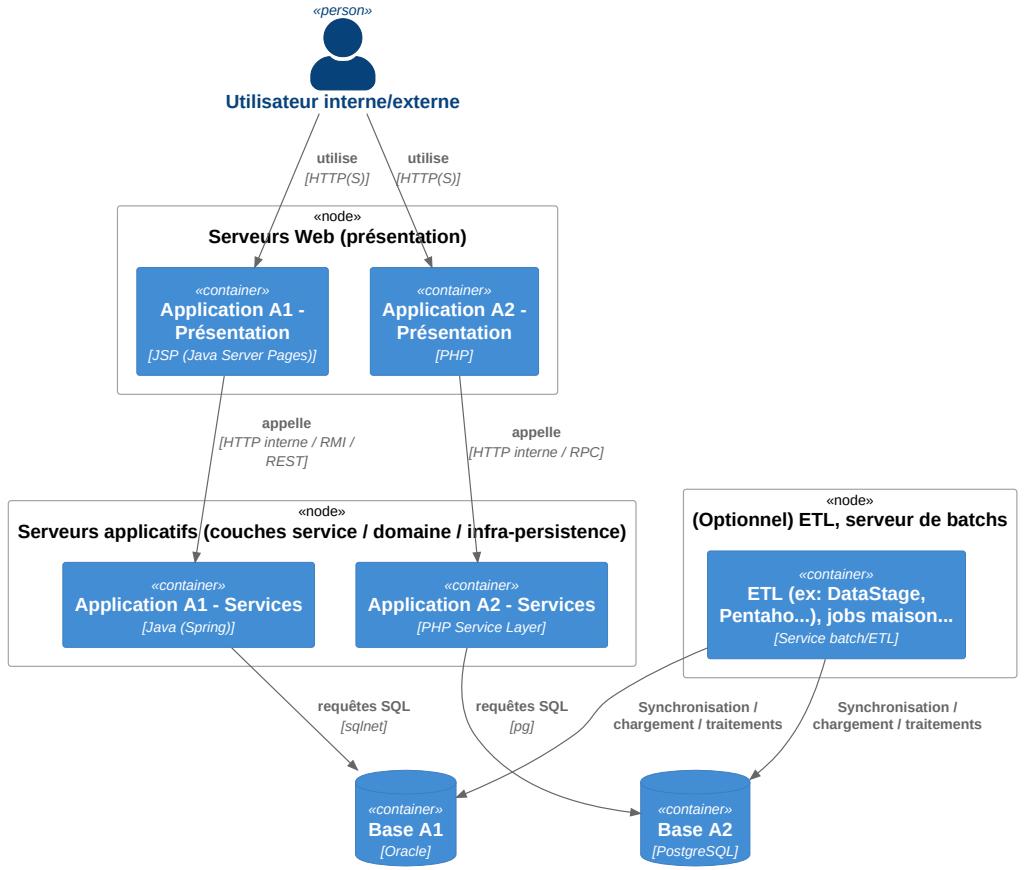


## Le monolithe (années 1990 - aujourd'hui)

- Applications Web (Server Side Rendering)
- Unité de déploiement unique: Toute l'application est déployée en un bloc
- Base de code commune: Tous les modules partagent le même codebase
- Couplage fort: Les composants sont étroitement liés et interdépendants
- En général, découpage en couches (au sein d'un seul tiers)

## Le monolithe : avantages / inconvénients

- **Web**: simple à déployer
- **Architecture simple**
- **Technologies homogènes**: Utilisation généralement d'un seul langage/framework
- **Vendor Locking** selon les technologies retenues mais acceptable (ex: JEE)
- **Scalabilité** verticale ET horizontale possible mais indifférenciée par fonctionnalité
- **Difficile à maintenir** (couplage fort, code complexe)
- **Collaboration difficile** (conflits de merge...)
- **Difficile à tester**
- **Stack technologique** : presque impossible à migrer (il faut tout réécrire)
- **Lourd à démarrer / déployer**



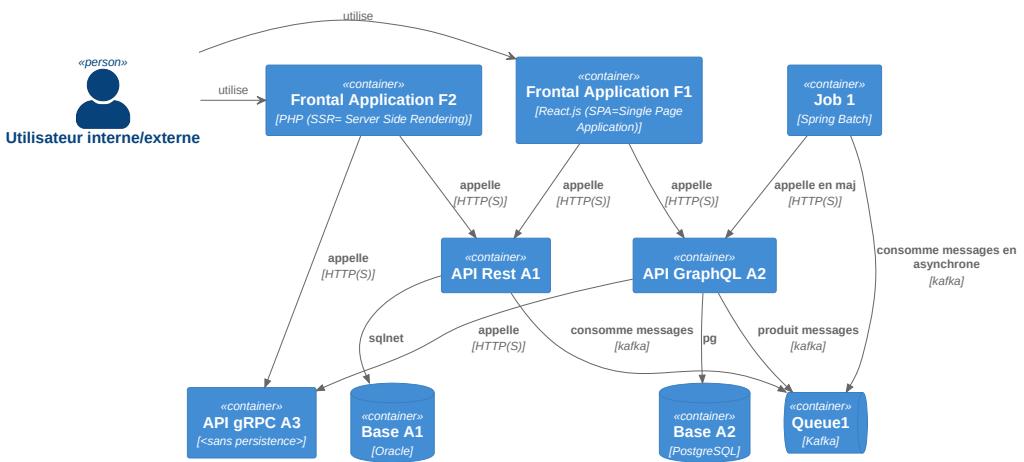
## Le n-tiers (années 2000 - aujourd'hui)

- Applications Web (Server Side Rendering)
- Proche du monolithe mais unité de déploiement par tiers (en général, un tiers présentation / un tiers service / un tiers persistence)

## Le n-tiers : avantages / inconvénients

- ✓ **Architecture assez simple**
- ✓ **Scalabilité**: Verticale ET horizontale (mais limitée sur les BDD)
- ✓ **Scalabilité individuelle** de chaque tiers (ex: 2 serveurs de présentation, 3 serveur de service)
- ✓ **Découplage** présentation / services
- ✓ **RH** : Possible d'avoir deux équipes : une frontend et une backend
- ⌚ **Maintenance** quelque fois difficile à maintenir (couplage fort, code complexe dans chaque tiers)
- ⌚ **Vendor Locking** : Selon les technologies retenues mais acceptable (ex: JEE)
- ⌚ **Testabilité** : Plus simple à tester (ex: bouchonnage du tiers services)
- ⌚ **Déploiement**: Peut être lourd à démarrer / déployer. Démarrage à faire dans l'ordre.
- ⚠ **Stack technologique** : presque impossible à migrer (il faut tout réécrire)
- ⚠ **Pas de réutilisation** des services par d'autres applications

## Le micro-services (années 2010 - aujourd'hui)



- Découpage par **services**
- Organisation autour des **capacités métier**
- **Produits** plutôt que projets
- **Points de terminaison intelligents et tuyaux simples**
- Gouvernance et gestion des données **décentralisées**
- **Automatisation** de l'infrastructure
- Conception pour **tolérer les défaillances**
- Conception **évolutive**
- Écriture **polyglotte** (languages de programmation des services possiblement différents)



**Note :** L'architecture microservice a été précédée du **SOA** (Service-Oriented Architecture) dans les années 2000

- Le SOA est une architecture similaire mais s'appuie en général sur un Enterprise Service Bus (ESB) centralisé et non des appels directs (même on trouve en architecture microservices de plus en plus d'API Gateway)
- En SOA, les WebServices se basaient sur le standard SOAP (et non REST, GraphQL ou gRPC)
- Les WebServices étaient principalement développés en Java (Java Enterprise Edition = JEE) et non multi-languages (polyglotte)
- Le SOA était principalement représenté par des solutions propriétaires complexes et coûteuses

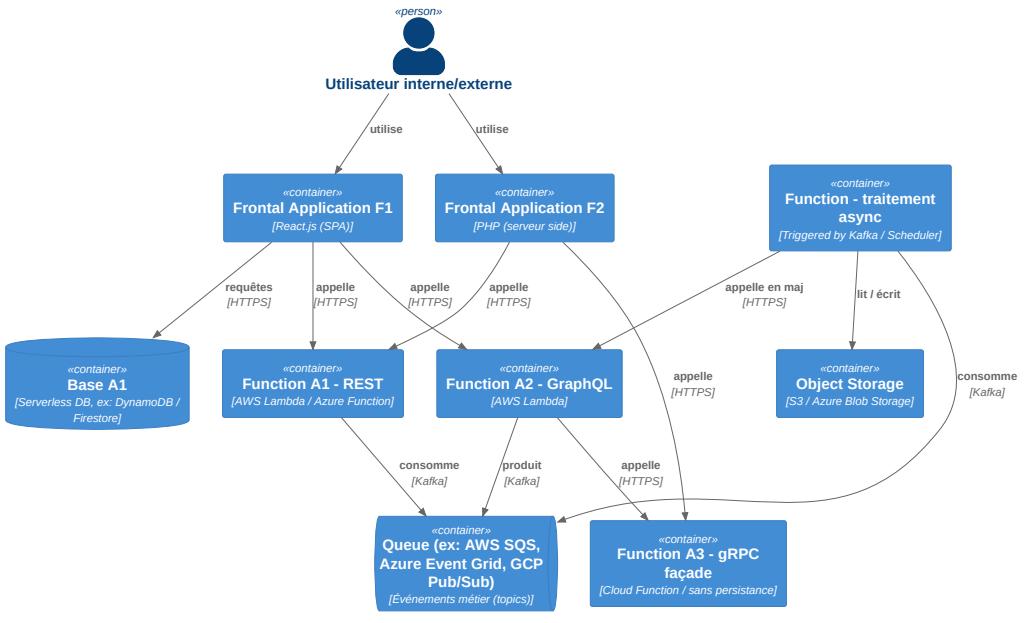
## Le micro-services : avantages / inconvénients

- ✓ **Facilité d'évolution** : les services peuvent être remplacés, réécrits ou supprimés
- ✓ **Autonomie/parallélisation des équipes** : chaque équipe peut développer et déployer ses services
- ✓ **Réutilisation des services** entre applications
- ✓ **Déploiement indépendant** de chaque service : maj plus fréquentes et ciblées
- ✓ **Scalabilité verticale et horizontale granulaire**
- ✓ **DEV plus simple** : petits périmètres, plus faciles à comprendre et à tester, **très adapté à l'IA**
- ✓ **Peu de vendor locking** : technologies Open Source et standard principalement
- ⌚ **Code polyglotte** : avantage RH mais aussi un **risque sur la maintenabilité**
- ⌚ **Surcoût en infrastructure** : orchestrateurs, API Gateway, monitoring, observabilité...
- ⌚ **Tests d'intégration** plus simples mais **tests système** plus complexes
- ⚠ **Architecture complexe** : code plus simple mais intégration plus complexe et nécessite des mécanismes robustes et une gestion des erreurs (rejeux...)
- ⚠ **Surdimensionné** pour certaines organisations (nécessite DevOps, CI/CD, observabilité, résilience...)
- ⚠ **Consistance des données plus difficile** : chaque service gère sa propre base → cohérence eventualisée

# Le serverless (années 2015 - aujourd'hui)

Traits principaux (source: Thoughtworks) :  
serverless :

- **Faible Barrière à l'entrée**
- **Sans hôtes** (que l'on gère soit-même en tout cas)
- **Sans états (Stateless)**
- **Élasticité**
- **Distribué**
- **Orienté événements (Event-Driven)**



## Le serverless : avantages / inconvénients

- ✓ Fin de l'infrastructure à gérer et nécessitant des compétences élevées, avec **HA native**
- ✓ Fin de la scalabilité à gérer, élasticité automatique, mais **latence à froid**.
- ✓ Autonomie / parallélisation des équipes : chaque équipe peut développer, tester et déployer son propre service
- ⌚ Coût raisonnable en théorie (paiement à l'utilisation) prévoir **surveillance importante** pour éviter les surcoûts
- ⌚ Développement en théorie rapide et simple, mais **débogage complexe et risque sur la cohérence globale**
- ⚠ **Vendor-locking** très élevé (forte dépendance aux plateformes Cloud)
- ⚠ **Surface d'attaque** plus large, surtout si on multiplie les fournisseurs
- ⚠ **Architecture complexe** : code plus simple mais intégration plus complexe, nécessitant des mécanismes robustes et une gestion des erreurs (rejeux...).
- ⚠ **Consistance des données plus difficile** : chaque service gère sa propre base → cohérence eventualisée

 **Note :** Les architectures n-tiers, microservices et serverless représentent **l'état de l'art** actuel.

- L'architecture microservice présente de nombreux avantages mais **nécessite** une force de frappe technique considérable
- Utiliser une architecture n-tiers classique en couches/hexagonale + présentation en SPA (React.js...) pour les projets simples ou les organisations plus modestes
- Le serverless est **peu adapté** aux environnements **souverains** (administrations) ou **sensibles**
- On le rencontre plus fréquemment dans les **structures agiles** (startups, PME) ou celles disposant de **moins d'exigences techniques internes**
- Ces architectures sont souvent **utilisées de façon complémentaire**. Elles ne s'excluent pas mutuellement.
- En général, je recommande pour les applications **de taille moyenne à importante de limiter le serverless aux fonctions périphériques** (envoi d'emails, traitement de fichiers, BI...)

Plus de détail :

[Cours de M1, thème « typologies d'architectures »](#)

## 1.2 Rappel sur les ENF

Une Exigence Non Fonctionnelle est une **exigence** portant sur une **aptitude d'un système informatique** (exemple: la confidentialité).

Les exigences fonctionnelles précisent ce que doit faire le système (le quoi) : règles de gestion, IHM, traitements, ...) alors que les ENF précisent les **attributs de qualité du système**.

## ENF d'exploitabilité

- **Robustesse** : Capacité du système à fonctionner correctement malgré des erreurs
- **Résilience** : Aptitude à se remettre automatiquement d'une panne ou d'un incident
- **Observabilité** : Possibilité de diagnostiquer l'état du système via logs, métriques et traces



## ENF de performances

- **Scalabilité** : Capacité à maintenir les performances en augmentant les ressources
- **Rapidité**: Aptitude d'un système à présenter un temps de réponse (TR) acceptable pour un niveau charge donné
- **Élasticité** : Aptitude d'un système à provisionner /dé-provisionner automatiquement des ressources en fonction de la charge afin de conserver des temps de réponses acceptables tout en optimisant le coût



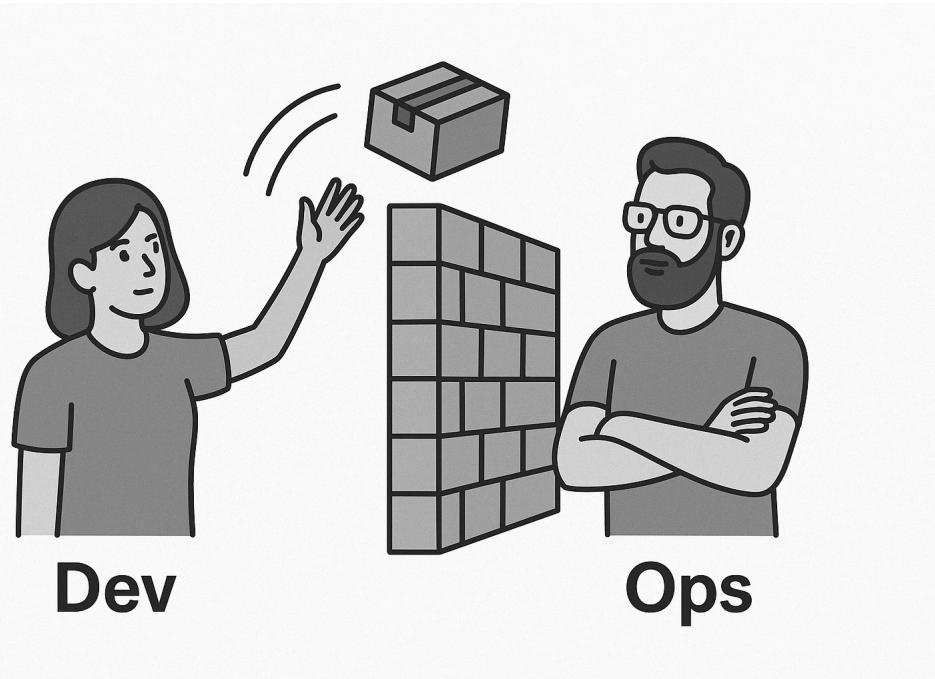
## ENF de sécurité

- **Disponibilité** : Le système est accessible et fonctionnel quand nécessaire
- **Confidentialité** : Les données sont protégées contre les accès non autorisés
- **Intégrité** : Les données ne sont ni altérées ni corrompues



**Note :** Voir au besoin [le cours de M1](#) sur les Exigences Non Fonctionnelles (ENF).

## 1.3 Contexte des Organisations Traditionnelles



- **Structure Organisationnelle :** Hiérarchique et cloisonnée par rôles :
  - **Devs :** Développement de nouvelles fonctionnalités
  - **Ops :** Gestion de l'infrastructure et des déploiements

**Processus :** Souvent rigides et basés sur des cycles longs

# Équipes de Développement (Devs)

- **Objectif Principal** : Livrer rapidement de nouvelles fonctionnalités
- **Priorités** :
  - **Innovation** : Introduire de nouvelles fonctionnalités et améliorations
  - **Réactivité** : Adapter rapidement le produit en fonction des retours utilisateurs
  - **Cycle de Développement Court** : Utiliser des méthodologies agiles
- **Défis** :
  - Ne pas introduire des bugs ou des instabilités (régressions, nouvelles fonctionnalités instables)

# Équipes d'Opérations (Ops)

- **Objectif Principal** : Assurer la stabilité et la sécurité des systèmes
- **Priorités** :
  - **Fiabilité** : Garantir que les applications fonctionnent sans interruption
  - **Sécurité** : Protéger les systèmes contre les menaces
  - **Maintenance Préventive** : Effectuer des mises à jour pour éviter les pannes
- **Défis** :
  - Gérer les changements fréquents tout en maintenant la stabilité et la sécurité.

## Défis de la Collaboration Devs-Ops

- **Silos Organisationnels** : Communication limitée entre équipes
- **Processus Lents** : Déploiements longs et complexes (Time To Market = TTM allongé)
- **Résistance au Changement** : Culture d'entreprise parfois réticente à l'innovation
- **Impact sur l'Innovation** : Difficulté à répondre rapidement aux besoins du marché



## 1.4 DevOps : culture, principes et objectifs

Livrer des logiciels **plus rapidement, plus fréquemment et plus fiablement**



Le DevOps est un **ensemble de pratique agiles** appliquées à l'**intégration** et non à la gestion de projet (comme Scrum) ou au code (comme XP)

## Convergence avec DevOps

- **DevOps** : Aligner les objectifs des Devs et des Ops.
- **Avantages :**
  - **Amélioration de la Communication** : Réduction des silos
  - **Déploiements Plus Rapides et Sécurisés** : Automatisation avec CI/CD
  - **Culture de Collaboration** : Responsabilité partagée pour la qualité et la stabilité

# Les piliers du succès

Réduire  
les silos  
organisationnels

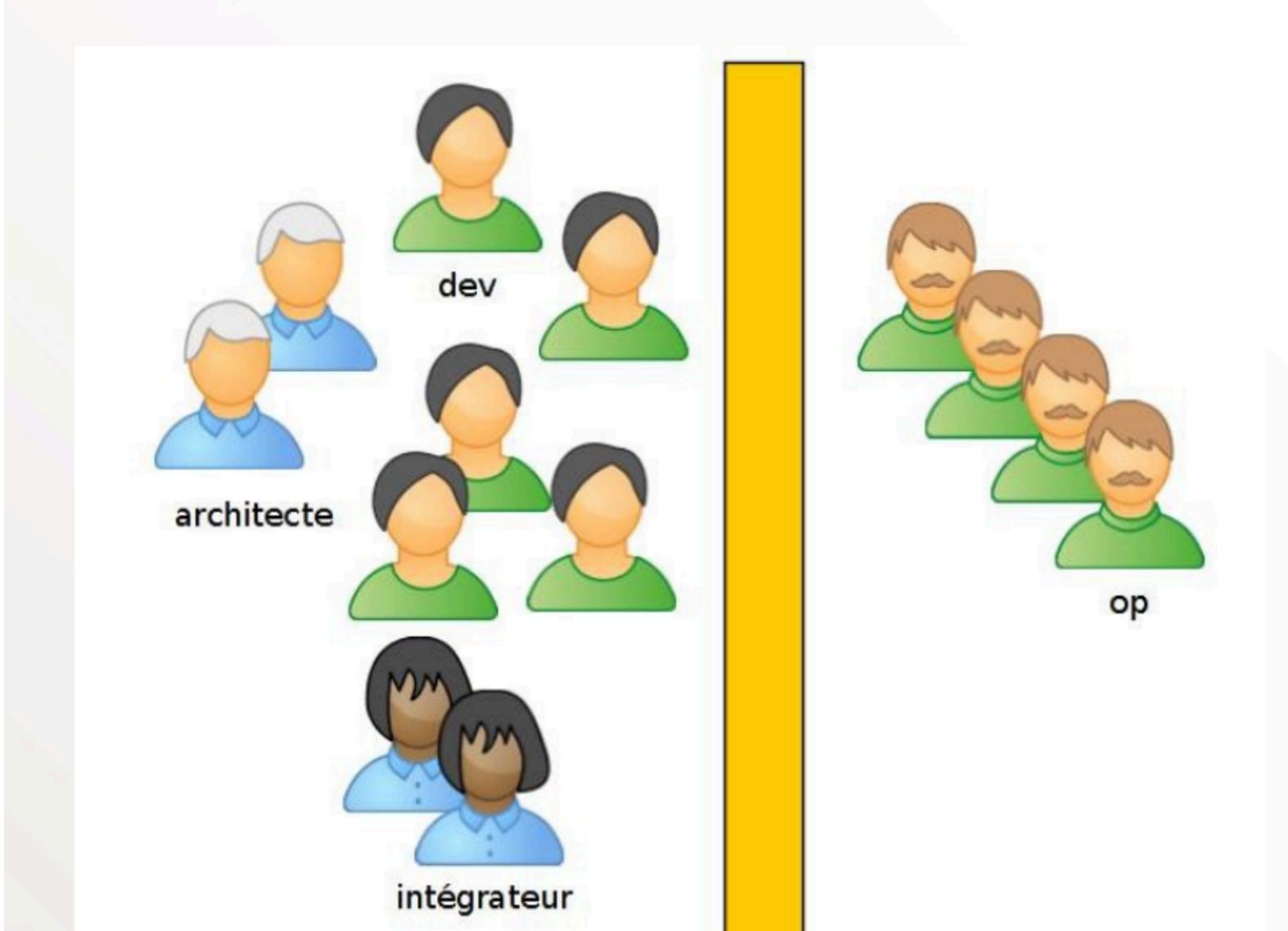
Accepter  
l'erreur

Mettre en place  
les changements  
graduellement

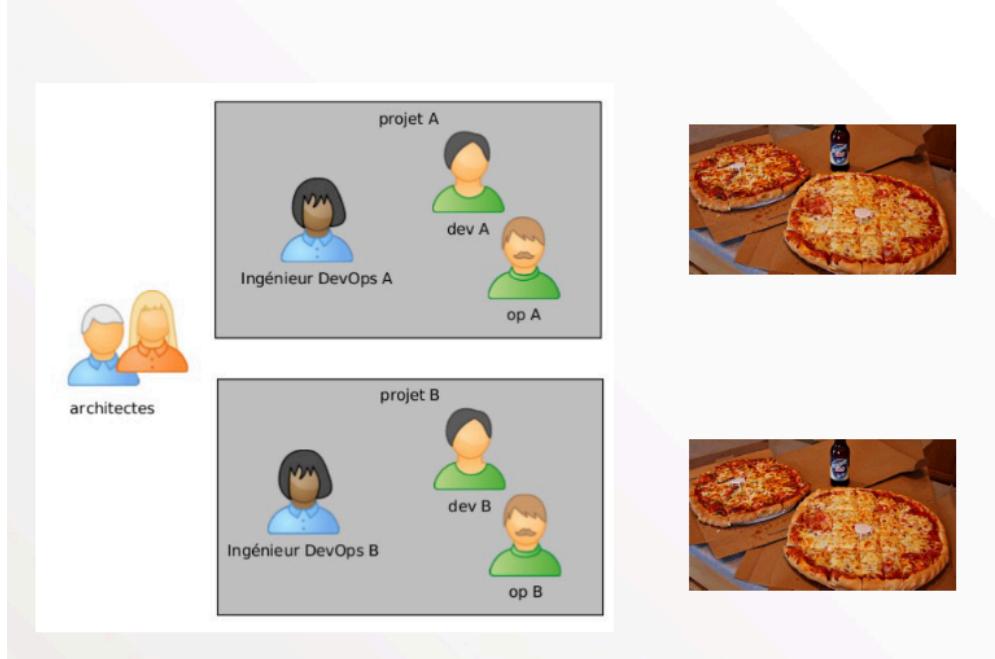
S'appuyer sur  
les outils  
et  
l'automatisation

Tout  
mesurer

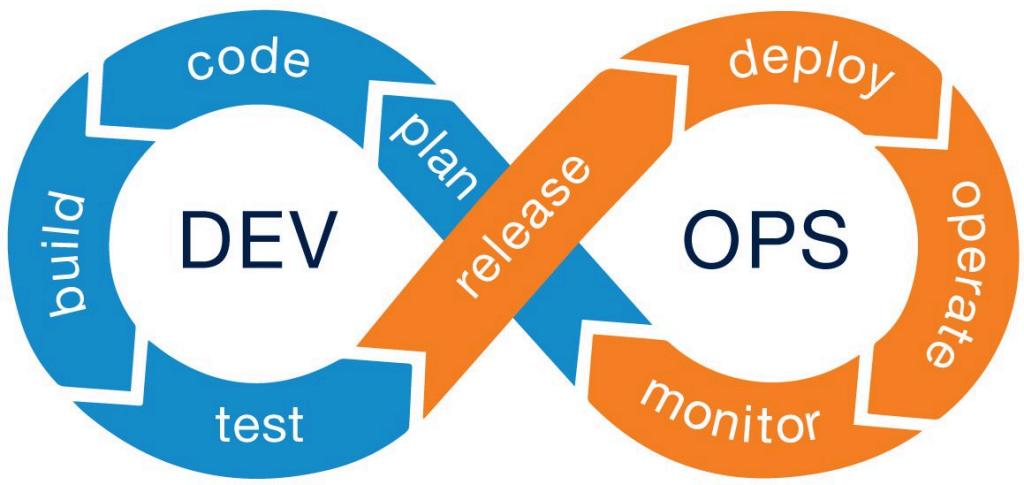
## Passer de cette organisation :



# A cette organisation (two pizza teams) :



- **Équipes petites** (5 à 8 personnes max), **autonomes, multi-compétences**
- **Responsables** à la fois du code et de l'**exploitation**
- **Communication fluide** entre Dev et Ops



## Le cycle d'amélioration continue DevOps

# Étapes clés du cycle DevOps

-  **Plan** : prioriser les besoins (fonctionnels et non fonctionnels)
-  **Code** : Écrire le code source et les tests unitaires — souvent collaboratif (Git, MR...)
-  **Build** : Compiler, empaqueter l'application et générer des artefacts
-  **Test** : Exécuter des tests automatisés (unitaires, d'intégration, de sécurité...)
-  **Release** : Préparer la MEP, valider le déploiement dans des environnements de test
-  **Deploy** : Déployer automatiquement (CD, canary...)
-  **Operate** : Bon fonctionnement (patches, ajouter ressources, modifier configuration)
-  **Monitor** : Observer performances, superviser, collecter métriques, détecter anomalies

# Comment mieux collaborer ?

## Penser système, pas équipe isolée

- Communication multi-équipes via messagerie instantanée de préférence

## Feedback

- Source d'information commune (ex: daily)

## Innovation et amélioration continue

- Ateliers

## Partager les responsabilités, culture du blameless.

## Casser les carcans

- Les devs exploitent (**You build it, you run it** » Amazon)
- Les ops codent (IaC : Infrastructure as Code)

# Qu'est-ce qu'on entend par automatisation ?

## CI : Intégration Continue

 Tests automatisés (TU, TI, E2E, spécifications exécutables...)

 Livraison continue

 Infrastructure as Code (IaC): Provisioning des serveurs et composants

 GitOps : Si ce n'est pas dans Git, ça n'existe pas

 CD (Livraison Continue voire Déploiement Continu)

## Relations entre DevOps et Cloud-native

| DevOps et Cloud-native sont étroitement liés et complémentaires

 DevOps = Méthode et culture

 Cloud-native = Architecture des applications à déployer sur un cloud

- **Microservices**, conteneurs, orchestration (ex. Kubernetes)
- **Scalabilité**, tolérance aux pannes, portabilité
- **Infrastructure dynamique** et éphémère

DevOps est un prérequis naturel au cloud-native : impossible d'exploiter des architectures distribuées modernes sans automatisation et collaboration

Cloud-native amplifie les bénéfices de DevOps : scalabilité automatique, testabilité, déploiement continu, observabilité native

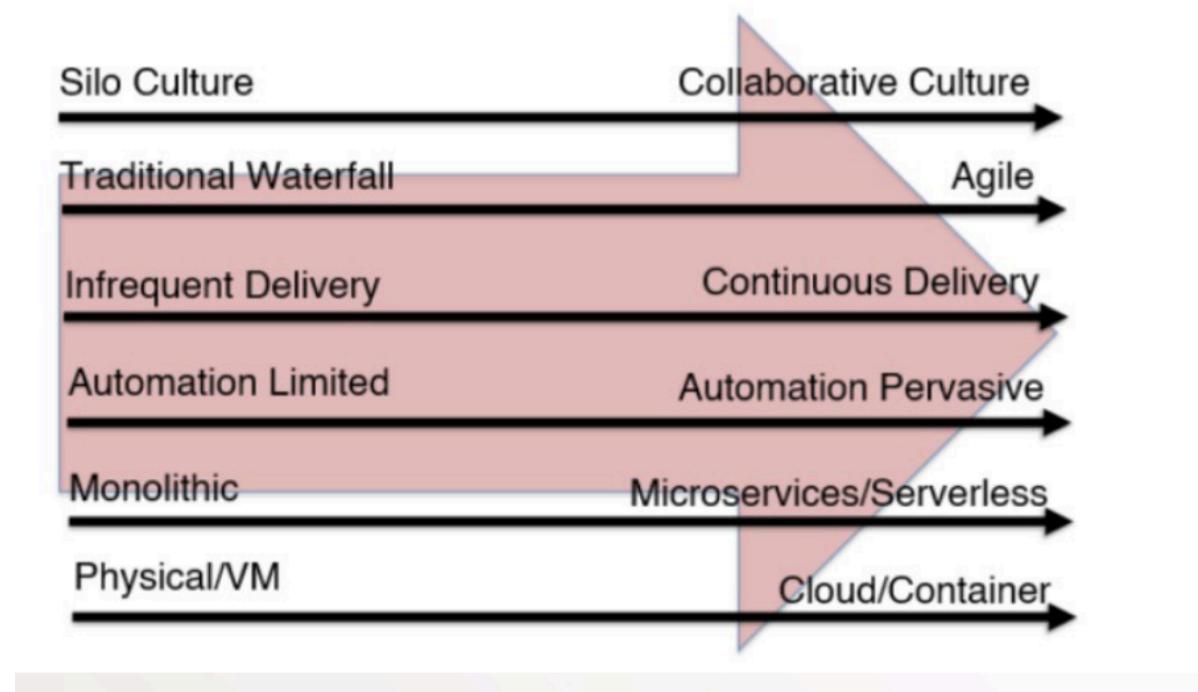
## Pets vs Cattle : deux visions de l'infrastructure

Cette métaphore illustre le **changement de culture** entre l'infrastructure traditionnelle (manuelle) et l'infrastructure cloud-native (automatisée, jetable, scalable).

Pets (animaux de compagnie)	Cattle (bétail)
Nommés individuellement (ex: svr_bdd )	Identiques et numérotés (ex: node-001 , node-002 )
On les soigne, on les répare	On les remplace automatiquement (permet de repartir d'une situation connue rapidement)
Configuration manuelle	Crées par script / IaC
Chers à maintenir	Jetables, scalables, automatisés
Tendance "on-premise"	Tendance cloud / conteneurs / DevOps

 L'approche "Cattle" favorise la **scalabilité, la résilience et l'automatisation**, au cœur des pratiques **cloud-native et DevOps**.

# En résumé: un mix de changements architecturaux, organisationnels et humains



 **Quiz !** Niveau DevOps de la pratique (de 0 à 5) :

- 1 : Les ops fournissent aux devs un **dashboard Zabbix** mais seuls les Ops reçoivent les alertes par mail
- 2 : Un **canal de messagerie instantanée** a été mis en place pour remonter les problèmes de PROD mais la plupart du temps les Ops ne signalent que les indisponibilités programmées
- 3 : Chaque équipe produit dispose d'un.e **DevOps dédié** qui automatise tous les aspects du cycle de vie (build déploiement, alerting...) et dispose des droit de PROD sur le provider Cloud
- 4 : Les modules sont **conteneurisés en Docker** mais lancés manuellement (pas d'orchestrateur)
- 5 : L'équipe de Dev rédige un **document d'installation manuelle en Word** et l'envoie aux Ops par mail lorsque la version du produit est publiée



## 1.5 Bonnes pratiques DevOps (architecture et livraison)

## ⚙️ Indépendance de la Configuration

### Principe :

Ne jamais embarquer de **configuration plate-forme dépendante** dans les paquets applicatifs.

### Pourquoi ?

- ✗ Risque de couplage fort avec l'environnement
- ✗ Difficulté à déployer sur plusieurs cibles (dev, prod, cloud, etc.)
- ✗ Impossibilité de reproduire des déploiements de manière fiable
- ✗ Gaspillage d'espace disque pour stocker les livrables

## Bonne pratique

-  Séparer **binnaire applicatif** et **configuration d'environnement**
-  Utiliser des mécanismes externes (fichiers de configuration, **variables d'environnement**, systèmes de configuration)
-  Favoriser des déploiements **portables** et **prévisibles**



## Automatiser (presque) tout

**Principe :**

Automatiser le **déploiement**, le **rollback**, le **provisionnement**, etc.

**Objectifs**

- Gagner en **fiabilité** et en **rapidité**
- Réduire les erreurs humaines
- Faciliter les répétitions et les rollbacks

## ⚠️ Attention à l'overkill

- Pas besoin d'automatiser **absolument tout !**
- Autoriser quelques actions manuelles **quand le coût d'automatisation est supérieur au gain attendu**

Voir [cette section](#) de "The Site Reliability Workbook" de Google: "It's important to note that eliminating toil isn't always the best solution".

## La gestion des branches en Trunk-Based Development (TBD)

- Approche Trunk-Based Development (TBD) + FF (Feature Flags)
- **Une seule branche** (ex : `main`)
- Variante conseillée : **Short-Lived Feature Branches** : branches à vie courte (quelques h à quelques j) pour les **Merge Requests (topics)** uniquement
- Feature-Flags pour activer/désactiver facilement les nouvelles fonctionnalités
- Objectif : simplifier l'intégration, **réduire les conflits** et écarts
- Configurer le **Fast-Forward only** (impose rebases) pour simplifier l'historique

## Canary Testing

### Définition :

Déploiement progressif d'une nouvelle version logicielle à une fraction restreinte des utilisateurs (les « canaries ») pour détecter rapidement les anomalies.

### Objectifs

-  Déetecter précocement les anomalies en prod
-  Minimiser les risques lors des déploiements
-  Réagir rapidement grâce à un monitoring renforcé

## Fonctionnement

1. Déploiement à un groupe restreint (les FF peuvent servir de filtre)
2. Surveillance des métriques clés (erreurs, latence, UX)
3. Extension progressive ou rollback selon les résultats

## Avantages

-  Limite l'impact d'un incident
-  Feedback rapide et fiable
-  Améliore la confiance et réduit le stress des équipes

## Dark Deployment

### Définition :

Déploiement d'une nouvelle version ou de nouveaux modules en production **sans les exposer aux utilisateurs finaux**

### Objectifs

-  Tester l'infrastructure et les performances
-  Identifier les problèmes d'intégration en avance de phase
-  Préparer un basculement rapide (feature toggle)



## Blue-Green Deployment

### Définition :

Technique de déploiement où **deux environnements identiques** (Blue et Green) sont utilisés pour minimiser les interruptions

### Fonctionnement

1. **Blue** : environnement en production actuel
2. **Green** : nouvelle version déployée en parallèle
3. Test sur Green → bascule du trafic → mise à jour terminée !

Le déploiement suivant sera donc en Green -> Blue

## Objectifs

-  Déployer sans downtime
-  Pouvoir revenir rapidement à l'ancienne version en cas de problème
-  Sécuriser les mises en production

## Avantages

-  Bascule instantanée
-  Rollback facile et rapide (hors données)

 Challenges au niveau des évolutions des modèles de données et de leur compatibilité...

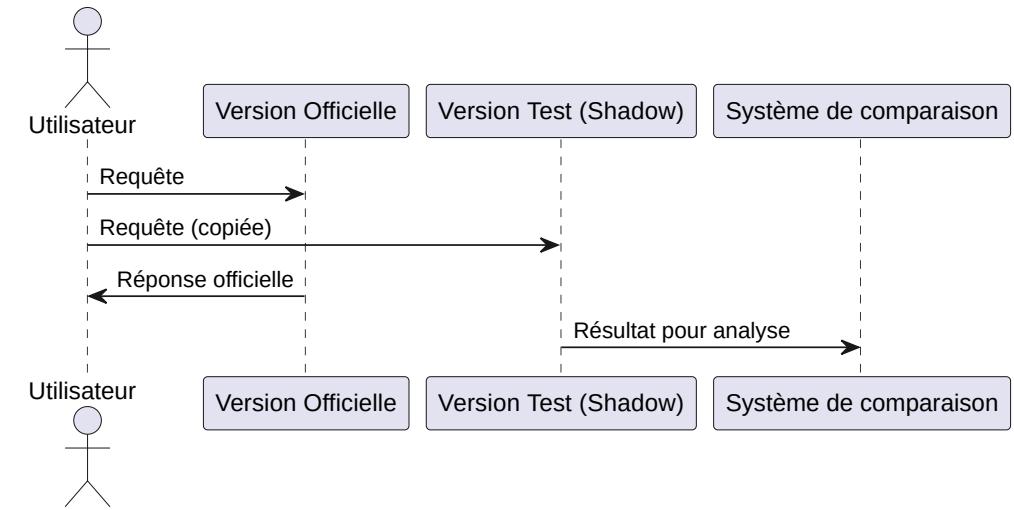
## » Shadow Traffic / Double Commande

### Définition :

Technique où **le trafic utilisateur est dupliqué** vers une nouvelle version sans impacter la réponse envoyée.

### Objectifs

- Valider le comportement de la nouvelle version
- Comparer les résultats avec l'ancienne version
- Déetecter les écarts sans risque pour les utilisateurs



## Fonctionnement

1. Le trafic réel est envoyé aux deux versions
2. Seule la réponse de la version officielle est utilisée
3. Les réponses sont comparées en interne

## Avantages

-  Permet de vérifier l'intégration (configuration, tuning...)
-  Tests réalistes en conditions de production
-  Aucune perturbation pour les utilisateurs
-  Détection précoce d'erreurs ou de régressions fonctionnelles



## Définition :

Approche où **Git est la source unique de vérité** pour décrire l'état désiré d'un système, souvent pour du déploiement cloud/Kubernetes.

**Si ce n'est pas dans Git, ÇA N'EXISTE PAS !**

77% des organisations déclarent adopter les principes GitOps (CNCF Annual Survey 2024) [[Source](#)]

## Fonctionnement

- Les configurations (infra, apps) sont **stockées dans Git**
- Les modifications sont faites par **pull | merge requests** (PR/MR)
- Les personnes sont identifiées via un **compte dédié**, pas un compte de service



## GitOps - Avantages

-  Déploiements traçables et auditables
-  Rollbacks facilités
-  Déploiements reproducibles et fiables
-  Séparation claire entre développement et opérations

## 🔧 Convention over Configuration (ou 'on rails')

### Principe :

Privilégier des **standards explicites** et les valeurs par défaut plutôt que laisser de nombreuses options manuelles.

### Exemples :

- Maven qui "opinionnalise" la structure des sources
- Comptes de service en dur déductibles par convention (ex: `cs-monapi` pour l'API `monapi`)

## 🔧 Objectifs

- Simplifier les décisions techniques
- Gagner en **productivité** et en **prévisibilité**
- Accélère le **onboarding**
- Réduire les erreurs et les écarts de pratique

## ✖ Quelques anti-patterns d'intégration à éviter

### Interventions humaines non maîtrisées

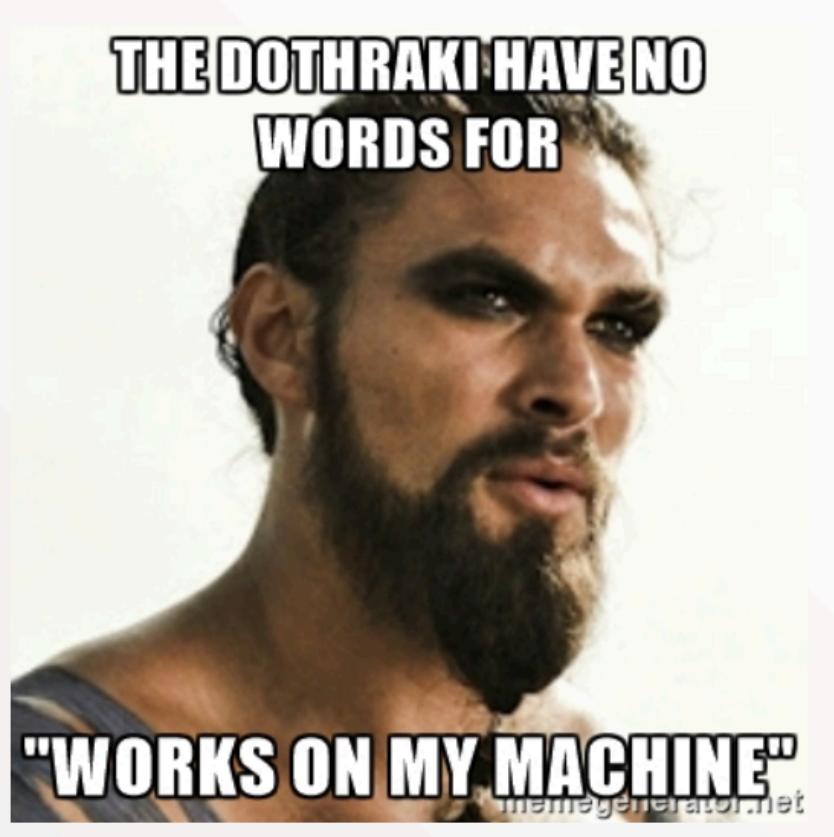
- Déploiements manuels, procédures non automatisées
- Dépendance à des **workflows informels ou oraux**
- Risques d'erreurs, de non-reproductibilité et de ralentissement

## ⚠️ Intégration continue défaillante

- **Builds lancés à la main**, sans trigger automatique
- **Tests trop longs** ( $> 10$  mins) ou non-reproductibles
- **Perte de confiance** dans les pipelines CI/CD
- **Temps de feedback trop élevé** → ralentit la boucle de développement
- **Contrôles** (qualimétrie, sécurité, performances...) **contournés** ou shuntés par les devs

## ⌚ Chaos dans la gestion de version

- Trop de **branches parallèles** → complexité, conflits, dérives
- **Absence de tags** -> difficile de retrouver la version qui a été livrée
- **Branches actives longtemps sans rebase** = dette technique
- **Absence de convention de nommage** des messages de commits et branches (voir le [Conventional Commit](#))



## ➡ Écart (ou impédance) DEV-PROD

- Des **binaries différents** selon les environnements  
(ex : DEV, TEST, PROD)
- **Spécificité de l'environnement de DEV**
  - (ex: développement sous Windows, déploiement sous Linux)
- Génération locale, configuration manuelle, "ça marche chez moi / **Works on my machine**"

## 1.6 Le DevSecOps

DevSecOps intègre **la sécurité** dès le début du cycle DevOps, au lieu de l'ajouter à la fin.

C'est l'**évolution naturelle du DevOps** et l'**état de l'art actuel**.

## Fondamentaux du DevSecOps

### 1. Intégration de la sécurité dès le départ (Secure By Design)

La sécurité n'est plus un "étape finale", elle est intégrée dès la conception et le développement.

### 2. Culture collaborative Dev + Sec + Ops

Les développeurs, les équipes sécurité et les opérations travaillent ensemble, en continu.

### 3. Automatisation des contrôles de sécurité

Scans de vulnérabilités, tests statiques, politiques de sécurité automatisées dans les pipelines CI/CD.

#### 4. "Shift Left"

Déetecter les failles **le plus tôt possible**, dès l'écriture du code, pour éviter des corrections tardives coûteuses.

#### 5. Observabilité et réponse aux incidents

Surveillance continue, journaux centralisés, alertes, détection d'anomalies et réponse rapide aux menaces.

#### 6. Politique de moindre privilège et de séparation des responsabilités

Chaque composant n'a que les droits nécessaires à son fonctionnement. Isolation stricte des rôles.

## 7. Sensibilisation à la sécurité dans les équipes

| Formation continue, bonnes pratiques, revues de code avec une vision sécurité.

## 8. Conformité outillée

| Outils de suivi de conformité (RGPD, ISO, etc.) et corrélation de logs (SIEM) : ex. Splunk.

## DevOps vs DevSecOps

	DevOps	DevSecOps
 Objectif	Livrer plus vite et plus souvent	Livrer vite <b>et</b> en toute sécurité
 Focus	Automatisation, CI/CD, collaboration	Sécurité intégrée dans tout le cycle
 Pratiques	CI/CD, Monitoring, IaC, observabilité	+ Scans, tests de sécurité, politique zero-trust
 Acteurs	Dev + Ops	Dev + Sec + Ops
 Sécurité	Souvent en fin de cycle	Intégrée dès le début (Shift Left)

## ? Lequel choisir ?

-  **DevSecOps est une évolution naturelle du DevOps**
-  Choisissez **DevSecOps** dès que des données sensibles, un contexte réglementaire ou une exposition web sont en jeu.
-  Pour des projets internes simples, DevOps peut suffire (au départ)... mais sécuriser dès le début reste toujours préférable.

 Aujourd'hui, dans un contexte de plus en plus dangereux (attaques, DDOS, rançongiciels, hameçonnage...), l'approche **DevSecOps** est à privilégier dans la plupart des cas.



## **Principes de Secure by Design : sécurité = critère d'architecture comme la performance**

- **Moindre privilège** : uniquement droits strictement nécessaires
- **Séparation des responsabilités** : fonctions critiques isolées pour limiter les impacts
- **Surface d'attaque minimale** : réduction des points d'entrée exposés
- **Défense en profondeur** : plusieurs couches de sécurité superposées
- **Comportement sûr par défaut** : si pas explicitement autorisé : refusé
- **Validation stricte des entrées** : toute donnée externe considérée comme non fiable
- **Auditabilité** : les actions critiques sont traçables et exploitables en cas d'incident

## 🔥 Les CVE les plus critiques de ces dernières années

Score CVSS (Common Vulnerability Scoring System) =~ 10/10 :

CVE	Produit/Système affecté	Pourquoi c'est marquant
<b>CVE-2021-44228 — Log4Shell</b>	Apache Log4j	Exécution de code à distance via JNDI, exploité à grande échelle ; considéré comme l'une des vulnérabilités les plus critiques de la décennie.
<b>CVE-2022-30190 — Follina</b>	Microsoft MSDT	Exploit facile via document Office, très utilisé en phishing/attaques post-compromission.
<b>CVE-2022-1388 — F5 BIG-IP iControl</b>	F5 BIG-IP	Failles dans les consoles d'administration réseau exposées en entreprise.
<b>CVE-2022-26134 — Confluence RCE</b>	Atlassian Confluence	Exploité massivement dans des attaques opportunistes sur serveurs Confluence.
<b>CVE-2025-55182 — React2Shell</b>	React / Next.js	Nouvelle vulnérabilité critique touchant le rendu serveur de composants React.
<b>CVE-2025-32756 / CVE-2025-32800 /</b>	Divers serveurs & systèmes (Apache, MySQL, PostgreSQL, etc.)	En continuum 2025, de nombreuses vulnérabilités



## Défense en profondeur (Defense in Depth)

Principe fondamental de la sécurité : **multiplier les couches de protection** pour réduire le risque d'intrusion ou de compromission, même en cas de faille.

💡 Certaines normes et meilleures pratiques recommandent la **diversité technologique** (dans la mesure du raisonnable) pour éviter les monocultures technologiques

Exemple: utiliser des firewalls de vendeurs différents

## Couches typiques de défense

1. **Contrôles physiques** : data centers sécurisés, badges, portiques, caméras...
2. **Réseau** : firewalls, segmentation, VPN, sécurité des flux (TLS)
3. **Infrastructure** : durcissement des OS, patchs de sécurité, monitoring système
4. **Accès et identités** : authentification forte, RBAC, MFA
5. **Applications** : validation des entrées, protections contre les injections
6. **Données** : chiffrement, gestion des secrets, contrôle d'accès aux bases
7. **Surveillance et réponse** : journaux, alertes, Intrusion Detection System (IDS),  
Intrusion Prevention System (IPS), plans d'intervention

## Etude de cas : rôle du DevSecOps dans le cadre d'une CVE majeure ?

**Log4shell (CVE-2021-44228) :**

**Impact** : Exécution de code à distance (RCE) sur les serveurs vulnérables

**Score CVSS** : 10.0 / 10 (critique)

**Composant affecté** : Log4j 2 (versions < 2.15.0)

**Vecteur** : Chaînes de log contenant des expressions \${jndi:ldap://...} injectées depuis des requêtes utilisateur

**Exploitabilité** : Très simple — aucun accès authentifié requis



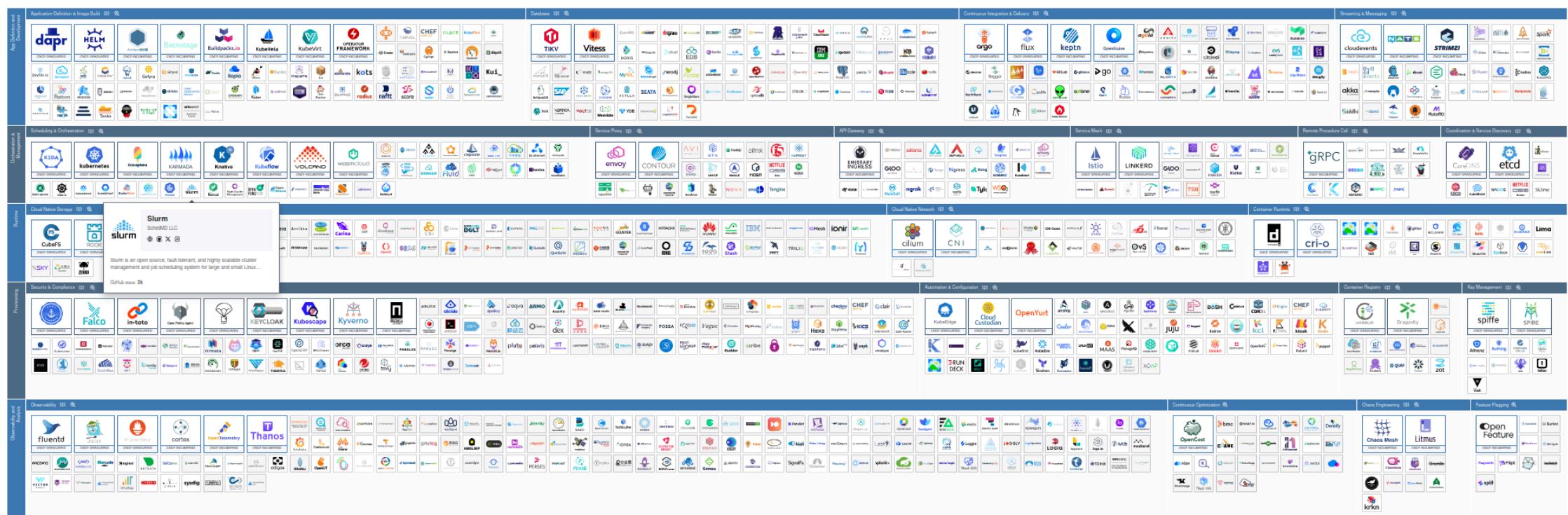
## Séance 2 – Écosystème technologique : IaC, CI/CD & sécurité (1h20)

- 1 - L'outillage et l'infrastructure DevOps / Cloud Native...
- 2 - Rappel sur les conteneurs
- 3 - Kubernetes : le système nerveux du cloud
- 4 - L'Infrastructure as Code (IaC)
- 5 - Notions de déploiement sécurisé
- 6 - La CI / CD

# 💡 2.1 L'outillage et l'infrastructure DevOps / Cloud Native...

Voir le [Landscape CNCF](#) (Cloud Native Computing Foundation)

- Un écosystème foisonnant...
- ... en évolution rapide et constante





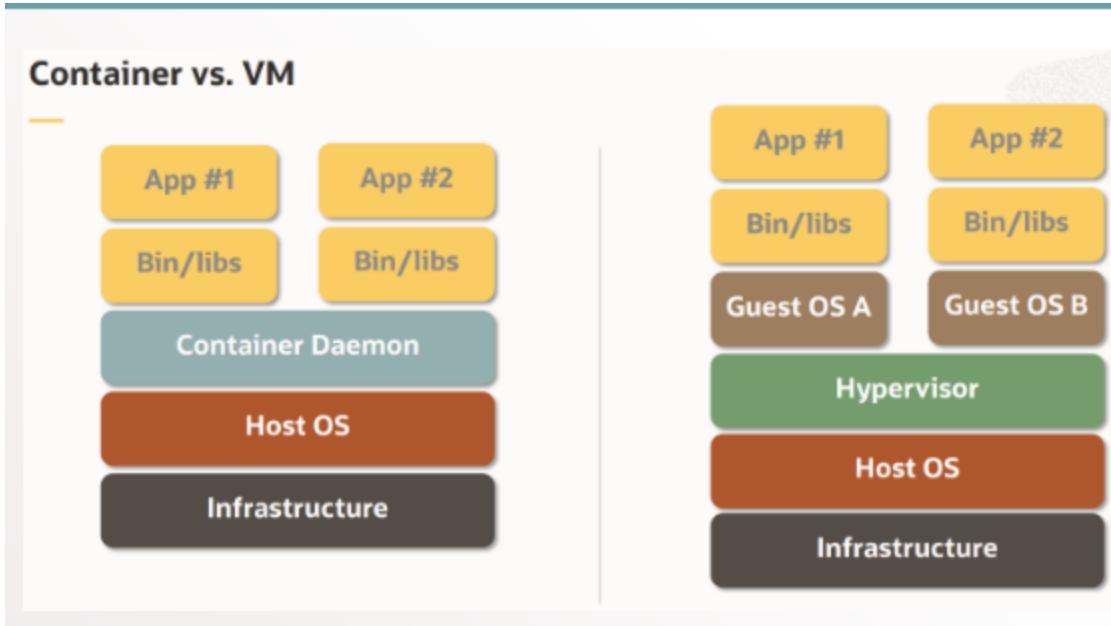
## 2.2 Rappel sur les conteneurs

Un **conteneur** est un environnement léger, isolé et portable, permettant d'exécuter une application avec toutes ses dépendances.

### ✓ Caractéristiques

- **Partage le noyau** de l'hôte (contrairement à une VM)
- **Étanche** (processus isolés)
- **Démarrage rapide**, faible empreinte mémoire
- **Image immuable** = mêmes résultats sur tous les environnements
- Facile à versionner, tester, déployer

# Conteneurs et VMs



**i** Ne pas opposer VM et conteneurs :

- VM : premier découpage pour grosses machines
- Les conteneurs tournent très souvent dans des VMs...
- Mais pas toujours (**bare metal**)



## Les images de conteneurs

Une **image de conteneur** est une archive contenant tout le nécessaire pour lancer une application (les binaires mais aussi ses libraries et les librairies système dont il a besoin).

### Norme de référence : [OCI \(Open Container Initiative\)](#)

- Standard ouvert pour les **formats d'image** et les **runtimes**
- Assure la **portabilité** entre outils : Docker, Podman, containerd, etc.



## Notion d'"onions" (couches)

- Une image est construite **par couches successives**
- Chaque instruction (RUN, COPY...) ajoute une **nouvelle couche**
- Les couches sont **en cache et partagées** entre images
- On peut **reconstruire partiellement** l'image si une couche change
- Consolidées sous la forme d'une archive ( `.tar.gz` )
- **Publiées (push) dans un registre d'image** (ex: Artifactory, GitLab Container Registry, GitHub Container Registry, ...) pour pouvoir être récupérées (pull)

## 🐳 Exemple de Dockerfile (Node.js)

```
# Image de base officielle
FROM node:18-slim

# Répertoire de travail
WORKDIR /app

# Dépendances
COPY package*.json ./
RUN npm install

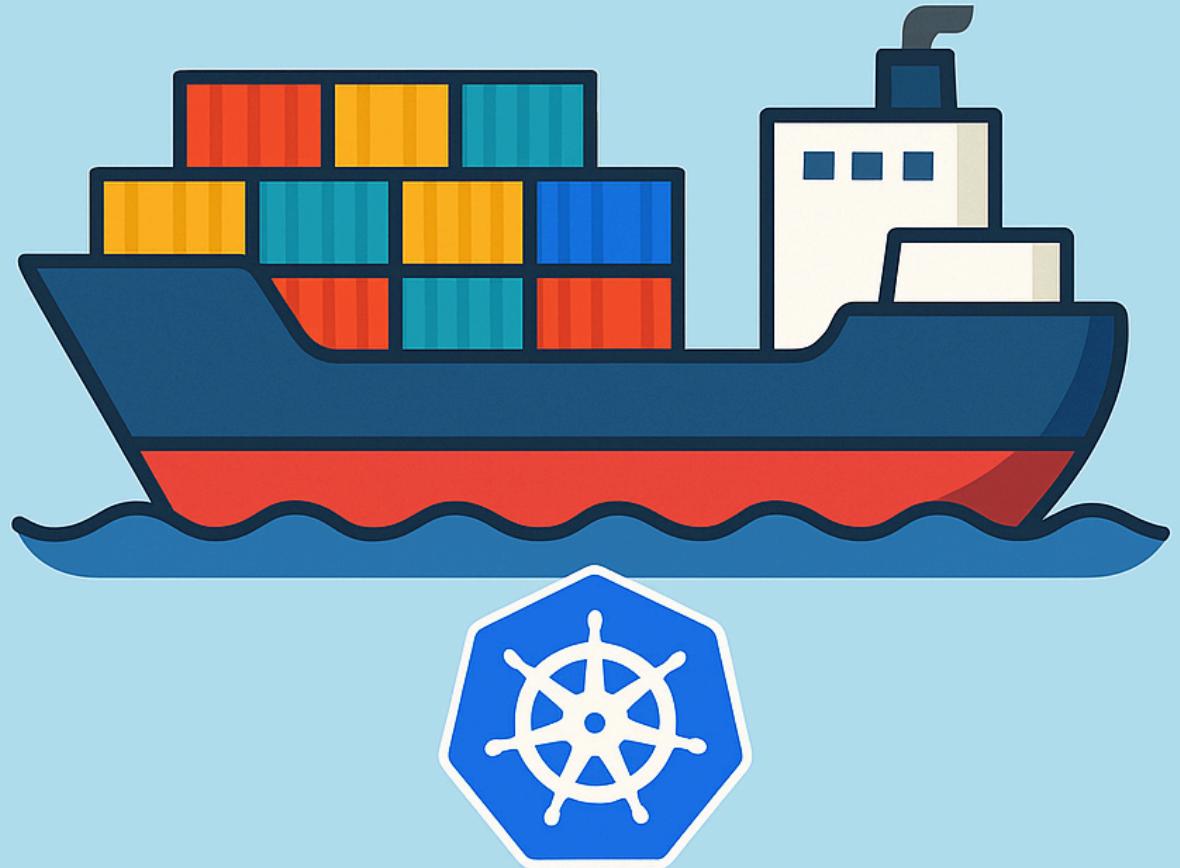
# Code source
COPY . .

# Port exposé
EXPOSE 3000

# Commande de démarrage
CMD ["npm", "start"]
```

## Exemples d'outils de construction et execution des conteneurs

- **Docker** : création et exécution de conteneurs (via un démon RESTful)
- **Podman** : alternative sans démon
- **Containerd / CRI-O** : autres moteurs d'exécution de haut niveau (gèrent le cycle de vie, le pull, l'exécution des conteneurs)
- **Kaniko/Buildah** : construction d'images OCI en userspace (hors conteneur)
- **runc** (par défaut, écrit en Go) / **crun** (écrit en C) / **youki** (en Rust) : moteurs d'exécution bas niveau



# KUBERNETES

## 2.3 Kubernetes : le système nerveux du cloud

“Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.”

— [kubernetes.io](https://kubernetes.io)

## ★ Kubernetes : un projet open source majeur

- Amorcé par Google en 2014
- Inspiré de Borg (orchestrateur interne de Google)
- Donné à la CNCF en 2015
- Principaux contributeurs : Google, Red Hat, VMware, Microsoft, Amazon



## Chiffres clés

- ≈ **120k stars** sur GitHub (projet Kubernetes)
- ≈ **8 600 contributeurs actifs** (période récente, Linux Foundation Insights)
- **91%** des organisations utilisent des **conteneurs en production** (CNCF Annual Survey 2024)
- **80% des grandes organisations dans le monde** utilisent K8S en 2025 (contre 63% en 2024) ([source CNCF](#))

# Pourquoi Kubernetes ?

- 💪 Orchestration des conteneurs à grande échelle
- 🌐 Portabilité cloud (AWS, Azure, GCP, on-premise)
- 🔧 Communauté et écosystème très riches

## Une infrastructure managée au-dessus de l'infrastructure

-  Conteneurs = unités de calcul (compute)
  -  Volumes = stockage persistant
  -  Services/Ingress/Network Policies = réseau intelligent et sécurisé
-  Kubernetes orchestre tout automatiquement



## Développement & exploitation

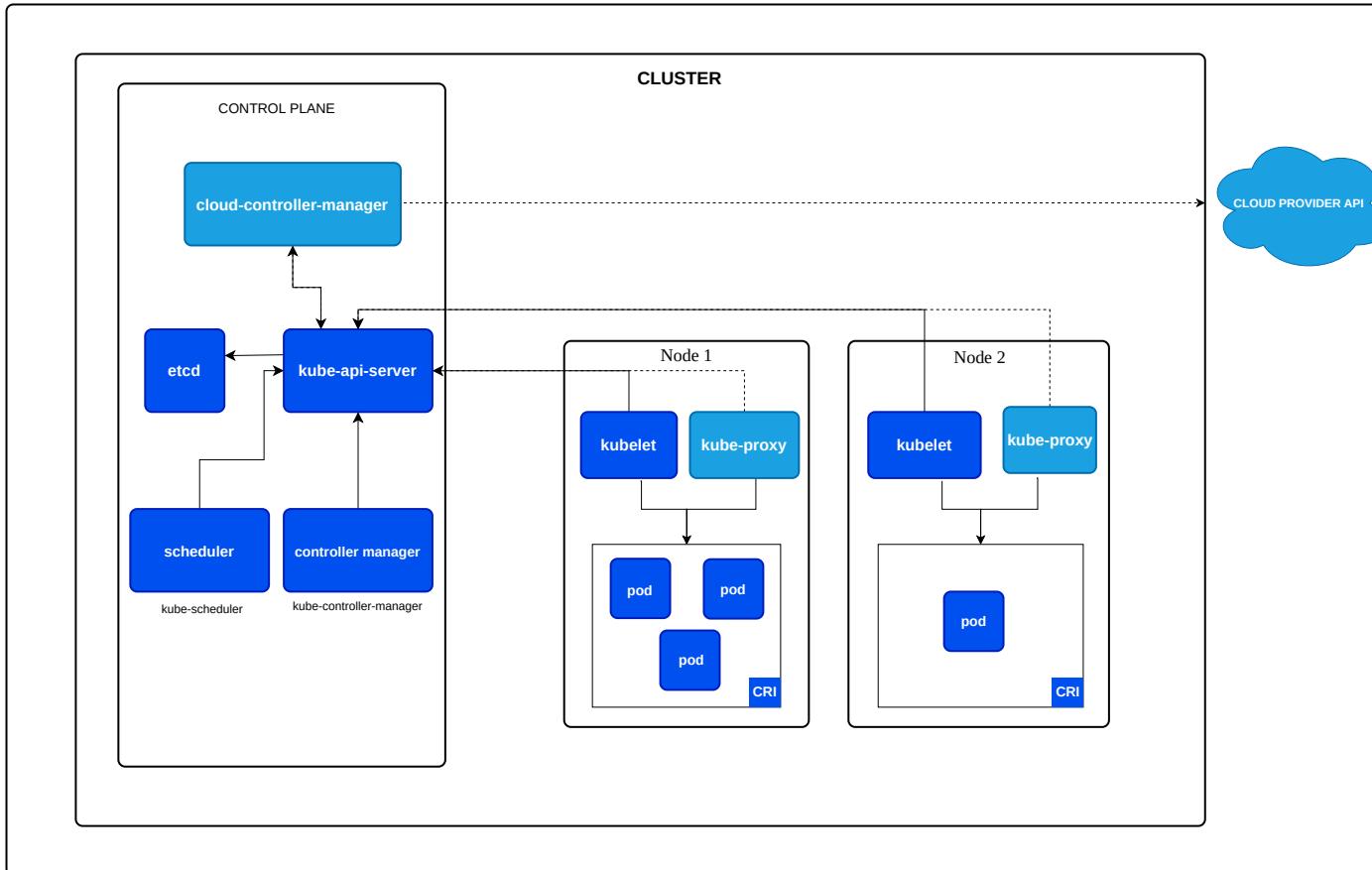
- Déploiement déclaratif (YAML, GitOps)
- Scalabilité automatique selon la charge
- Déploiement avancé : rolling updates
- Résilience intégrée : redémarrage, réPLICATION
- Observabilité native (logs, métriques, events)



## Kubernetes = moteur des apps cloud-native

- Déploiement continu, microservices, tolérance aux pannes
- Utilisé par : Google, Spotify, BlaBlaCar, OVH, etc.
- Standard de fait pour les plateformes modernes (AKS, EKS, GKE...)

## 🧩 Architecture de K8S



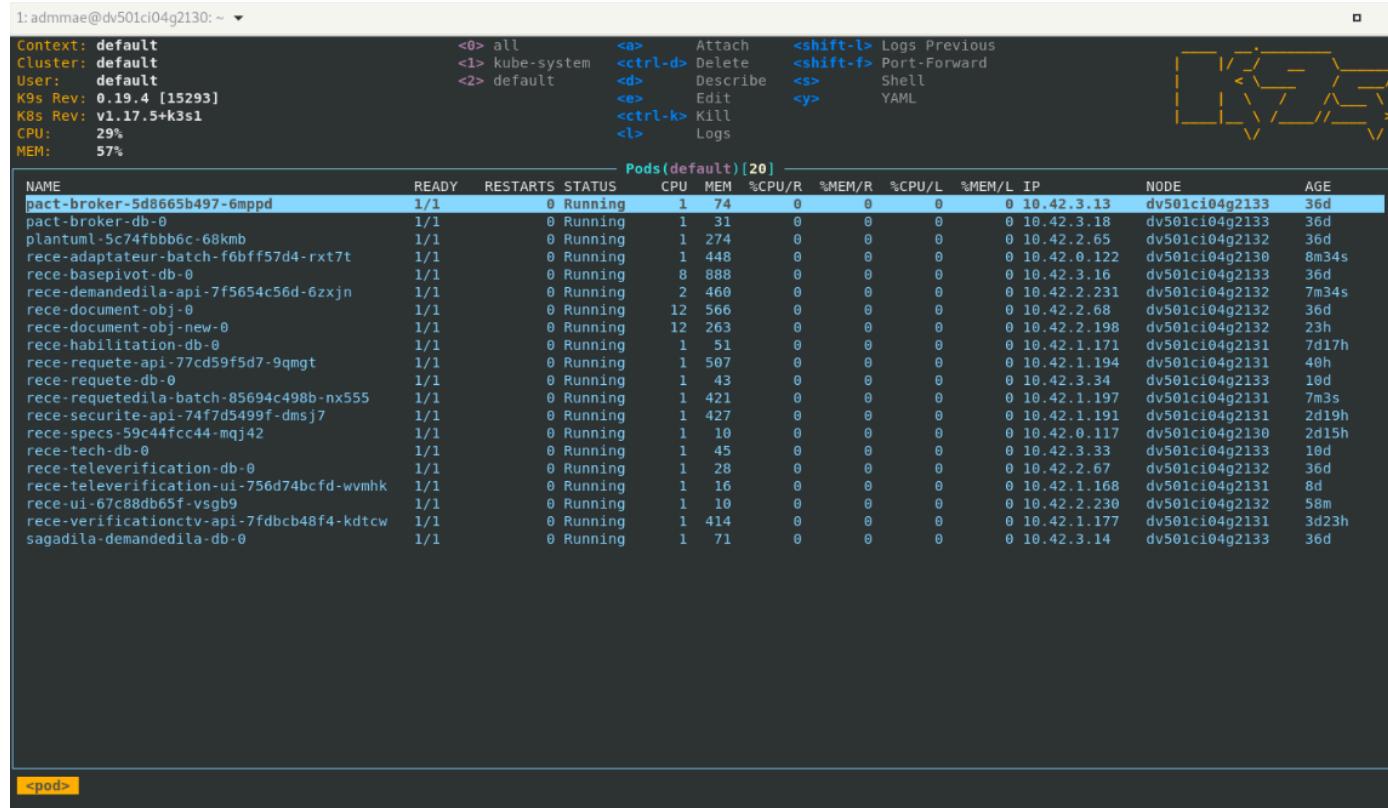
Source: <https://kubernetes.io/docs/concepts/architecture/>



## Exemple de déploiement Kubernetes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
```

# 💡 K9S, un navigateur Kubernetes



The screenshot shows a terminal window titled '1: admmae@dv501ci04g2130: ~'. The window displays the K9S (Kubernetes Terminal) interface. At the top, there's a context menu with options like 'all', 'Attach', 'Logs', 'Previous', 'Delete', 'Describe', 'Shell', 'Edit', 'YAML', 'Kill', and 'Logs'. Below the menu, it shows 'Context: default', 'Cluster: default', 'User: default', 'K9s Rev: 0.19.4 [15293]', 'K8s Rev: v1.17.5+k3s1', 'CPU: 29%', and 'MEM: 57%'. The main area is titled 'Pods (default) [20]' and lists 20 pods with columns for NAME, READY, RESTARTS, STATUS, CPU, MEM, %CPU/R, %MEM/R, %CPU/L, %MEM/L, IP, NODE, and AGE. The pods listed include pact-broker, pact-broker-db, plantuml, rece-adaptateur-batch, rece-basepivot-db, rece-demandedila-api, rece-document-obj, rece-document-obj-new, rece-habilitation-db, rece-requette-api, rece-requette-db, rece-requestedila-batch, rece-securite-api, rece-specs, rece-tech-db, rece-televerification-db, rece-televerification-ui, rece-ui, and sagadila-demandedila-db. The bottom left of the terminal has a yellow bar with the text '<pod>'. To the right of the terminal, there is a small cluster map icon.

💡 Outil presque indispensable et permettant d'accélérer grandement la gestion d'un cluster K8S

## Objectifs

- Déployer des environnements **de façon reproductible**
- **Versionner** l'infrastructure (comme du code)
- **Automatiser** la création, modification et suppression de ressources
- Réduire les **erreurs manuelles** et les écarts entre environnements



## La notion d'état désiré

En Infrastructure as Code, on décrit **l'état final attendu** de l'infrastructure, et non les étapes pour y parvenir

- L'**état désiré** représente ce que **l'infrastructure doit être** : types de ressources, configurations, relations
- L'outil IaC (ex : Terraform) se charge de **comparer l'état réel** à l'état désiré et applique les **changements nécessaires**

## Exemples

- "Je veux **2 serveurs** EC2 avec 8 Go de RAM"
- "Je veux un **load balancer** devant mes pods Kubernetes"
- "Je veux une **base PostgreSQL** avec sauvegarde quotidienne"
- "Je veux **4 instances** à tout moment de mon serveur Web et situées sur des noeuds différents"



## Avantages

- **Automatisation** des changements
- **Détection des dérives** entre ce qui est déployé et ce qui est attendu
- **Idempotence** : rejouer le code n'a pas d'effet s'il n'y a rien à changer

## En résumé

Les outils IaC ne spécifient pas *comment faire*, mais ce que l'on veut obtenir

 Tous les outils d'IaC, quelle que soit leur niveau d'abstraction (voir plus loin) utilisent ce concept mais à des niveaux différents.

- Dans un déploiement **Kubernetes**, ce sera par exemple « je désire au moins **cinq instances** (pods) de mon application »
- Dans un **playbook Ansible**, ce sera : « Je veux que le fichier /etc/xyz.conf existe" ou "le **service SystemD ABC** est démarré »



## Exemples d'outils IaC orientés provisionnement

Outil	Éditeur / Origine	Description principale
Terraform	HashiCorp	Provisionnement multi-cloud déclaratif en HCL
Pulumi	Pulumi Corp.	IaC en langages classiques (Python, TS, Go...)
CloudFormation	Amazon Web Services (AWS)	Déploiement d'infra AWS en JSON/YAML
ARM / Bicep	Microsoft (Azure)	Provisionnement pour Azure (ARM = JSON, Bicep = DSL)
Deployment Manager	Google Cloud	IaC pour GCP avec YAML/Jinja
Crossplane	Upbound	Provisionnement Kubernetes via CRDs



## Exemple de fichier Terraform (HCL)

```
provider "aws" {  
    region = "eu-west-3" # Paris  
}  
  
resource "aws_instance" "web" {  
    ami              = "ami-0c55b159cbfafe1f0" # Amazon Linux 2  
    instance_type   = "t2.micro"  
  
    tags = {  
        Name = "MonServeurWeb"  
    }  
}
```

# Outils IaC orientés Gestion de Configuration

Outil	Mode	Particularités
<b>Ansible</b>	Agentless (push)	Simple, basé sur SSH, YAML déclaratif
<b>Puppet</b>	Agent + serveur	DSL propre, très utilisé en entreprise
<b>Chef</b>	Agent + serveur	Basé sur Ruby, flexible mais complexe
<b>SaltStack</b>	Push + Pull hybride	Rapide, orienté événements

 Ces outils permettent de **configurer et maintenir l'état des serveurs**, une fois provisionnés par des outils d'IaC de provisionnement comme Terraform ou Pulumi

 Une approche **conteneurs** (voir plus loin) reste **préférable** quand possible

## ⚙️ Exemple de fichier Ansible (YAML)

```
- name: S'assurer que nginx est démarré
hosts: webservers
become: true

tasks:
  - name: Démarrer et activer nginx
    ansible.builtin.service:
      name: nginx
      state: started
      enabled: true
```



## Exemple de configuration Pulumi (TypeScript)

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

// Create an S3 bucket
const bucket = new aws.s3.Bucket("my-bucket", {
    acl: "private",
    tags: {
        Environment: "Dev",
        Name: "MyPulumiBucket",
    },
});

// Export the bucket name
export const bucketName = bucket.id;
```

## 🐳 Outils IaC orientés conteneur

Outil	Rôle principal	Particularités
Dockerfile	Décrit l'image du conteneur	Définition du build, base de tout conteneur
Docker Compose	Décrit des stacks multi-conteneurs en local	Idéal pour le DEV, moins adapté à la PROD
Helm	Gestion de packages Kubernetes (charts)	Templating puissant, logique de versionnage
Kustomize	Overlays et variantes d'objets Kubernetes	Natif K8s, sans langage de templating
CDK8s	IaC Kubernetes via des langages de programmation	Basé sur JS/TS/Python, alternative à YAML (jeune)

 Ces outils permettent de **configurer et maintenir l'état des serveurs**, une fois provisionnés par des outils d'IaC de provisioning comme Terraform ou Pulumi.

 Nous préconisons plutôt Kustomize pour les applications internes de l'organisation et Helm pour les applications externe (déjà packagées en Helm).

## ❖ Exemple de fichier Kustomize

```
# production/kustomization.yaml

resources:
- deployment.yaml
- service.yaml

commonLabels:
  app: demo-app

patches:
- target:
    kind: Deployment
    name: mon-app
  patch: |
    - op: replace
      path: /spec/relicas
      value: 3
```



## 2.5 Notions de déploiement sécurisé

Un déploiement sécurisé repose sur plusieurs **couches complémentaires** de protection



## Gestion des secrets

- Ne jamais stocker de secrets (jetons, clés...) en clair dans le code source (sauf éventuellement en DEV)
- Utiliser des outils dédiés : **Vault, Sealed Secrets, AWS Secrets Manager**
- Injecter les secrets via des variables d'environnement ou des volumes sécurisés



## Réseaux et isolation

- Segmenter (VLAN) les flux réseau : **externe / interne / administration**
- Appliquer des politiques de réseau (ex. : **NetworkPolicy**)
- Réduire la surface d'exposition des services (ex. : ingress, firewall, proxy)

## Contrôle d'accès basé sur les rôles (RBAC)

- Appliquer le principe du **moindre privilège**
- Définir les rôles avec précision : développeur, opérateur, administrateur, observateur...
- Implémenter le RBAC au niveau de **Kubernetes**, du **cloud**, des **pipelines CI/CD** et des **registries**

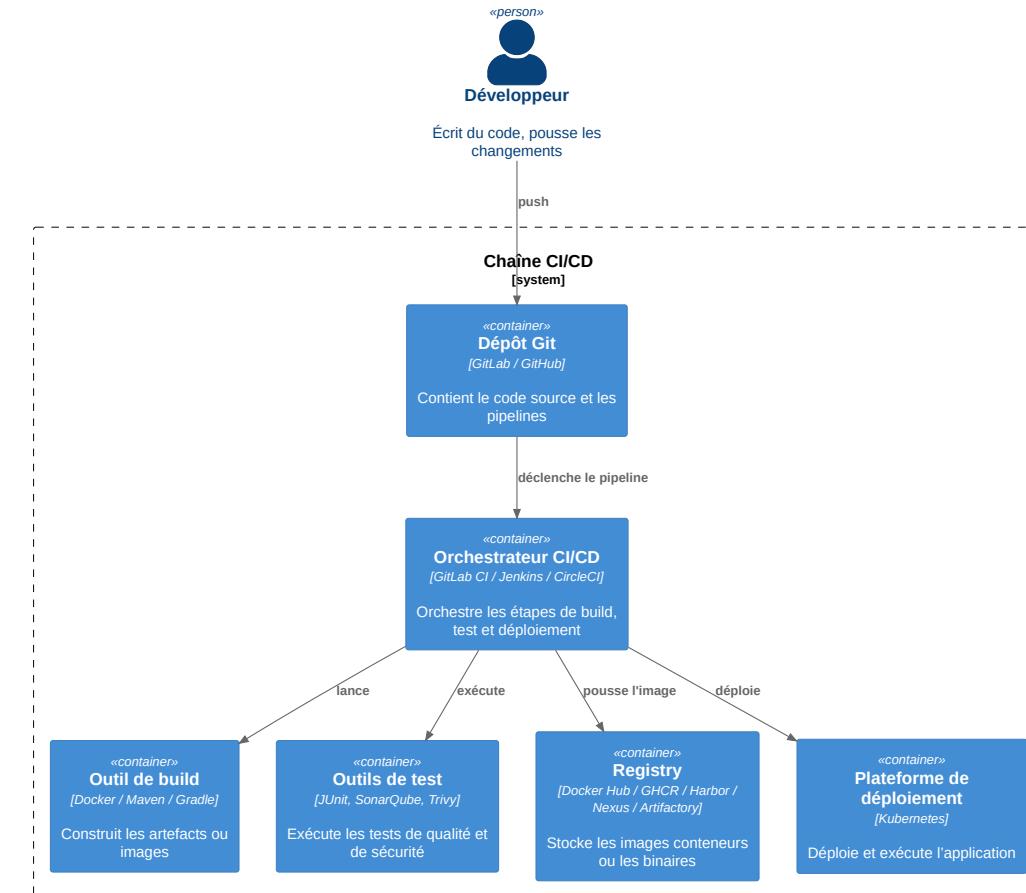
## Exemple de RBAC dans Kubernetes

```
# Role : lecture des Pods dans un namespace
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: mon-namespace
  name: lecture-pods
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]

# RoleBinding : association du rôle à un utilisateur
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: liaison-lecture-pods
  namespace: mon-namespace
subjects:
  - kind: User
    name: alice
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: lecture-pods
  apiGroup: rbac.authorization.k8s.io
```

## 2.6 La CI / CD

La CI/CD automatise la **construction**, les **tests** et le **déploiement** des applications, afin de **livrer plus vite, plus souvent, et avec plus de fiabilité**



## ⚙ CI – Intégration Continue (*Continuous Integration*)

- **Construit** les livrables finaux selon les technologies et langages et via diverses opérations : **Compilation, Transpilation, Minification, Bundling, Optimisation, Obfuscation, Linkage, Packaging** (zip, image OCI, war, deb, msi, jar,...), etc.
- Automatiser les **tests à chaque modification du code**
- Peut intégrer des tests de performance
- ... et des **tests de sécurité**
- Déetecter rapidement les régressions (**feedback rapide**)
- Analyse la qualité du code (Qualimétrie) : **inspection continue**
- Favoriser les **petites livraisons fréquentes**
- Exemples d'outils : GitLab CI, Jenkins, GitHub Actions, CircleCI, Drone

## GitLab CI/CD en bref

 **GitLab CI/CD** = système d'intégration et de déploiement continu intégré à GitLab

- Exécute des **pipelines** dès qu'un commit/push/MR est effectué
- Décrit le pipeline dans un fichier `.gitlab-ci.yml` à la racine du dépôt
- Utilise des **jobs** parallélisables, organisés en **stages** (build, test, deploy...) en série
- Les scripts exécutés par les jobs sont simplement des commandes bash (simple et puissant)

 Fonctionne avec des **runners** auto-hébergés ou cloud. En général, les **jobs** sont exécutés sous la forme de **conteneurs** (beaucoup plus reproductible)



## Exemple de pipeline GitLab CI simple

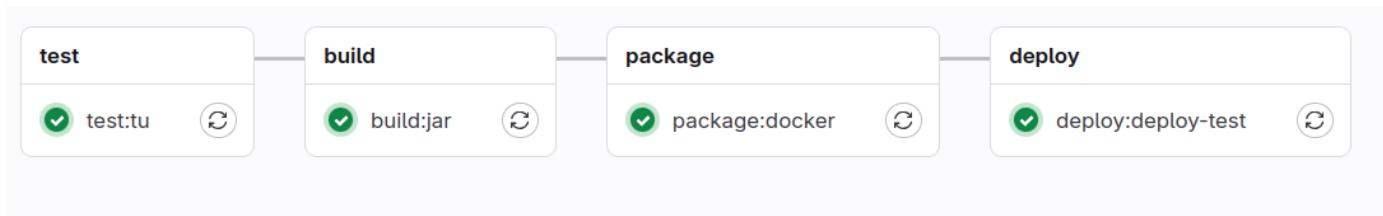
```
stages:
- build
- test
- deploy

build-job:
  stage: build
  script:
    - echo "Compilation..."
    - make

test-job:
  stage: test
  script:
    - echo "Tests unitaires"
    - make test

deploy-job:
  stage: deploy
  script:
    - echo "Déploiement en staging"
    - ./deploy.sh
only:
- main
```

## Un pipeline Gitlab-CI simple avec CD en test



# Un pipeline Drone-Cl simple avec CD en PROD

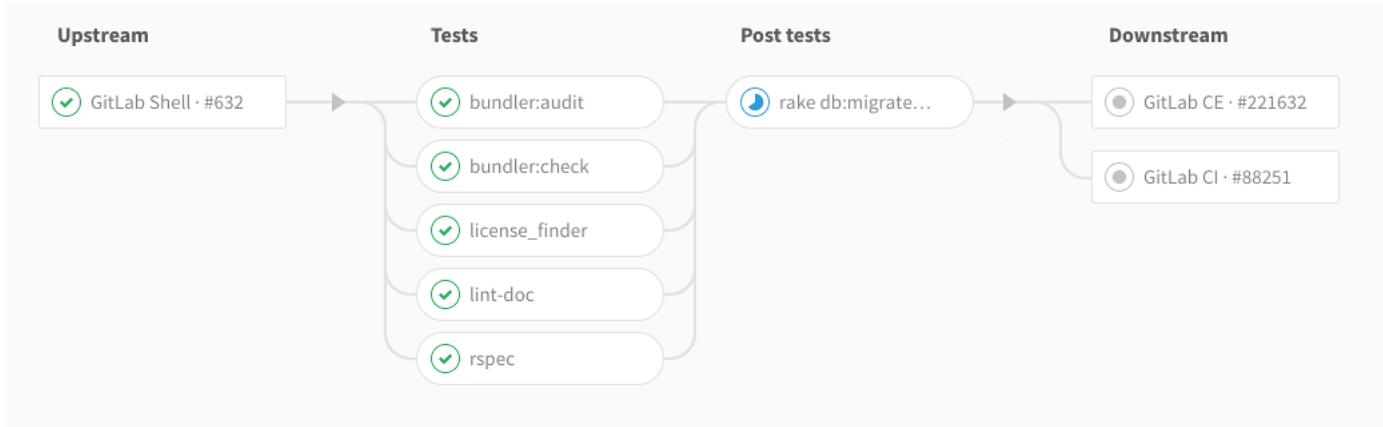
The screenshot shows a Drone CI build interface. At the top, it displays "Builds > #37 >" and the repository name "site-perso". A message indicates a push to the "main" branch. The pipeline consists of one stage named "default" with four steps: "clone" (00:08), "build\_site" (00:05), "build\_nginx\_image" (00:45), and "deploy\_k8s" (00:02). All steps are marked as successful (green checkmarks). The "LOG VIEW" tab is selected, while "GRAPH VIEW" is also available. A large portion of the log output is redacted.

PIPELINE STAGES		1 stage
default		01:00
STEPS		
clone	00:08	✓
build_site	00:05	✓
build_nginx_image	00:45	✓
deploy_k8s	00:02	✓

CONSOLE LOGS

```
1 Cloning with 0 retries
2 Initialized empty Git repository in /drone/src/.git/
3 + git fetch origin +refs/heads/main:
4 From https://git.florat.net/bflorat/site-perso
5 * branch      main      -> FETCH_HEAD
6 * [new branch] main      -> origin/main
7 + git checkout 73954d67035a393b4de63d0402f429bb7e2253d7 -b main
8 Switched to a new branch 'main'
9
```

## Un pipeline plus complexe avec parallélisation et pipeline downstream



Source : [gitlab.com](https://gitlab.com)

## ⚙ Jenkins : le vétéran de la CI/CD

- Jenkins est un outil open-source d'intégration continue (CI)
- Très modulaire, basé sur des **plugins** (plus de 1800 !)
- Pipelines décrits par un DSL (Domain Specific Language) basé sur Groovy
  - DSL déclaratif (plus simple, structuré) → `pipeline { ... }`
  - DSL scripté (plus flexible, moins lisible) → `node { ... }`
- Déploiement sur serveur (local ou cloud), UI web
- Configuration possible par UI **ou via code (`Jenkinsfile`)**

 A moins d'être sur une CI-CD historique, préférer Gitlab-CI ou d'autres CI plus modernes pour éviter le plugin-hell (plugins instables et incompatibles) bien qu'il soit possible de coder des jobs en bash...



## Exemple de Jenkinsfile simple (Déclaratif)

```
pipeline {  
    agent any  
  
    stages {  
        stage('Build') {  
            steps {  
                echo 'Compilation du projet...'  
                sh 'make'  
            }  
        }  
        stage('Tests') {  
            steps {  
                echo 'Lancement des tests...'  
                sh 'make test'  
            }  
        }  
        stage('Déploiement') {  
            when {  
                branch 'main'  
            }  
            steps {  
                echo 'Déploiement en production...'  
                sh './deploy.sh'  
            }  
        }  
    }  
}
```

## La qualimétrie, exemple avec SonarQube



Utilisez le plugin Sonarlint sur votre IDE pour améliorer votre code avant même d'être commité



## CD – Déploiement Continu (*Continuous Delivery / Deployment*)

- **Continuous Delivery** : le code est toujours prêt à être déployé (déploiement manuel)
- **Continuous Deployment** : le code validé est déployé automatiquement (en environnement de test voire en production pour les organisations très matures)



## Ajout de sécurité dans les pipelines CI/CD

L'intégration de la sécurité dans la chaîne CI/CD est un pilier de l'approche **DevSecOps**

Elle permet de détecter les failles **tôt, automatiquement, et de manière répétable**

## 🔍 Analyse de code et dépendances

### ✓ SAST (Static Application Security Testing)

- Analyse du **code source** avant l'exécution
- Déetecte les failles connues (ex : injections, erreurs de logique)
- Exemple : SonarQube, Semgrep

### ✓ DAST (Dynamic Application Security Testing)

- Analyse l'application en **exécution**, sans connaître le code
- Exemples : OWASP ZAP, Burp Suite

### ✓ SCA (Software Component Analysis)

- Vérifient les CVEs dans les dépendances
- Exemples : **OWASP Dependency Check, npm audit, Trivy, Gryspe, Artifactory XRay**

## Intégrité des artefacts : signature & attestation

### Signatures de build

- Assurent que les images / artefacts **proviennent bien de votre pipeline**
- Permettent la **vérification en production** avant déploiement

### Outils courants

- **cosign** : signature et vérification d'images conteneurs
- **Sigstore** : chaîne complète pour signer, publier et vérifier les artefacts

 Ajoute de la **tracabilité** et de la **confiance** à la chaîne logicielle



## Sécuriser les artefacts : SBOM & provenance



### SBOM (Software Bill of Materials)

- Liste exhaustive des composants d'une image ou d'un build
- Permet de retrouver facilement les dépendances, leurs versions et leurs licences **en production** (alors que le SCA ne s'applique que lors du build)
- Requis par de plus en plus de normes/réglementations (NIS2, ISO 27001, DORA, ...)



### Provenance

- Informations sur **qui a construit quoi, quand, et avec quoi**
- Permet l'**auditabilité** complète du build

Outils : **Syft, DependencyTrack, CycloneDX**, ...

## Exemple de SBOM au format CycloneDX

```
{  
  "bomFormat": "CycloneDX",  
  "specVersion": "1.4",  
  "version": 1,  
  "components": [  
    {  
      "type": "library",  
      "name": "spring-boot-starter-web",  
      "version": "3.2.5",  
      "licenses": [  
        { "license": { "id": "Apache-2.0" } }  
      ],  
      "purl": "pkg:maven/org.springframework.boot/spring-boot-starter-web@3.2.5"  
    },  
    {  
      "type": "library",  
      "name": "log4j-api",  
      "version": "2.23.1",  
      "licenses": [  
        { "license": { "id": "Apache-2.0" } }  
      ],  
      "purl": "pkg:maven/org.apache.logging.log4j/log4j-api@2.23.1"  
    },  
    ...  
  ]  
}
```



## Exercice

Comment intégrer une analyse SCA d'un logiciel JavaScript dans un pipeline ?



## Séance 3 – Marché du travail, métiers du DevSecOps, rôle de l'encadrant (1h20)

- 1 - Les principaux métiers de l'intégration
- 2 - De nouvelles exigences transverses
- 3 - La formation continue
- 4 - Sécurité, culture d'équipe, et responsabilité technique
- 5 - Ressources pour approfondir

## 3.1 Les principaux métiers de l'intégration

 Intégrateur applicatif

 Intégrateur d'exploitation / SysOps

 Site Reliability Engineer (SRE)

 Métier : Ingénieur DevOps

## Métier : Intégrateur applicatif

 **Objectif principal** : Assurer que les dépendances sont compatibles, à jour et sûres

### Activités principales

- Assemble les applications avec leurs dépendances : **Maven, npm, Gradle, PyPI**, etc.
- Gère les conflits de version et maintient la **cohérence globale**
- Met en œuvre l'analyse de sécurité des composants (SCA)
- Maintient une **documentation claire et traçable**

 Compétences clés : outils de build, dépendances, sécurité logicielle, veille technologique



## Métier : Intégrateur d'exploitation / SysOps

🎯 **Objectif principal** : Assurer le bon fonctionnement quotidien des systèmes informatiques en environnement Cloud

💡 Le SysOps est l'évolution naturelle du SysAdmin dans un environnement cloud.

### 🔧 Activités principales

- Surveille et gère les systèmes, services et applications
- Résout les incidents et problèmes techniques au quotidien
- Met en œuvre les **mises à jour et correctifs**
- Code et exécute des opérations manuelles ou planifiées d'**Infrastructure as Code (IaC)**
- Gère les **sauvegardes et les restaurations**
- Collabore avec les équipes DEV pour planifier les déploiements
- Maintient la **documentation opérationnelle**

## Métier : Site Reliability Engineer (SRE)

 **Objectif principal** : Améliorer la **fiabilité** des systèmes et applications grâce à des pratiques d'**ingénierie logicielle**

### Activités principales

- Développe des outils de **supervision**, **alerting** et **automatisation**
- Fiabilise le SI via l'**Infrastructure as Code (IaC)**
- Configure les environnements pour assurer **robustesse** et **résilience**
- Conduit des **tests de résilience** (ex. : **chaos engineering**)
- Optimise les performances, réalise des **benchmarks**
- Propose aux équipes DevOps les **bonnes pratiques IaC** (et les documente)
- Rédige les **post-mortems** après incidents

## Métier : Ingénieur DevOps/DevSecOps

 **Objectif principal** : Favoriser la collaboration entre les équipes **développement et exploitation** (SysOps, SRE...), pour **accélérer et fiabiliser** le cycle de développement et de déploiement logiciel

### Activités principales

- Automatise les processus de **développement, tests et déploiement** (CI/CD)
- Écrit et maintient l'**Infrastructure as Code** (Terraform, Kubernetes, Ansible...)
- Aide les DEV à appliquer les bonnes pratiques infra et sécurité (souvent issues du SRE)  
**cloud-native, haute disponibilité, rejeux, résilience**
- Met en place des démarches d'**amélioration continue**
- Participe à la **documentation technique et process**
- Intègre la **sécurité dans la CI/CD** (SAST, DAST, ...)

## ❖ Comparatif des rôles liés à l'intégration et à l'exploitation

Rôle	Objectif principal	Proximité Devs	Posture
Intégrateur applicatif	Assurer la compatibilité et la sécurité des dépendances	+++	Proactif
SysOps	Garantir le fonctionnement quotidien des systèmes	+	Réactif
Ingénieur DevOps / DevSecOps	Fluidifier et sécuriser le cycle Dev / Ops	++	Proactif++
SRE	Améliorer la fiabilité via l'ingénierie logicielle	+	Proactif+++

## Compétences clés pour devenir DevOps

### Compétences techniques indicatives

- **Langages de programmation** : Python, Go, Bash
- **Systèmes d'exploitation** : Linux, Windows
- **Conteneurisation & Orchestration** : Docker, Kubernetes
- **Gestion de configuration** : Ansible, Puppet, Chef
- **CI/CD** : GitLab CI, Jenkins, ArgoCD, ...
- **Infrastructure as Code** : Terraform, CloudFormation, Vagrant
- **Cloud** : AWS, OVH Public Cloud, Azure, GCP, ...
- **Surveillance & Logs** : Prometheus, Grafana, ELK Stack

 Prêt à démarrer ?

Guide des technologies à connaître pour devenir DevOps :

👉 <https://roadmap.sh/devops>



## Compétences humaines

- Communication & collaboration
- Résolution de problèmes
- Adaptabilité
- Culture DevOps & mindset agile



## En résumé

- L'**intégrateur applicatif** agit **en amont**, sur la qualité des dépendances
- Le **SysOps** assure **la continuité de service** au quotidien dans le cloud
- Le **DevOps/DevSecOps** crée **les ponts et les automatismes** entre Devs et Ops
- Le **SRE** agit en **ingénieur de la fiabilité**, avec des outils et une culture de production avancée

## Des métiers en pleine croissance

### Peu délocalisables

-  Développeurs parfois remplacés par l'IA, des progiciels ou du SaaS
-  Sysadmins souvent remplacés par des solutions cloud

mais:

 **SysOps / DevOps / SRE** : essentiels à la fiabilité

 **Multi-rôles recherchés** → Développement, automatisation, infra, observabilité

## L'impact de l'IA sur les métiers SecDevOps

### On accélère fortement sur certaines activités

- Développement et mise au point **beaucoup plus rapide** de scripts (SRE, Ops)
- Création de **petits outils, IHM, scripts** en *vibe programming*
- Génération accélérée de manifests (Terraform, Kubernetes, Dockerfiles, pipelines CI/CD, ...)
- Aide au **pentesting** (exploration, payloads, idées d'attaque)
- Analyse de **volumes massifs de logs / traces / alertes**
- Rédaction assistée de post-mortems / Root Cause Analysis, documentations opérationnelles, comptes rendus...
  - choses souvent *repoussées ou jamais faites*

### Réduction drastique du coût cognitif et du temps d'exécution

## ✖ Risques réels

- **Perte de maîtrise technique**
  - scripts copiés sans compréhension → risque de catastrophe
  - manifests “qui marchent” mais non compris
- **Illusion de compétence**
  - l'IA masque les lacunes fondamentales
- **Biais de confiance**
  - réponses plausibles mais fausses
  - erreurs subtiles en sécurité / réseau / concurrence
- **Standardisation dangereuse** (comme tout le monde utilise les mêmes IA)
  - mêmes patterns → mêmes failles → mêmes attaques

Ce fragment de manifest Kubernetes pour une base de données générée fonctionne à première vue mais en PROD, ça va faire mal...

```
resources:  
limits:  
  memory: "256Mi"
```

## Bonne posture face à l'IA

### Utiliser l'IA comme :

- **Précepteur** (explication, reformulation, exemples)
- **Assistant de réflexion**, pas décideur
- **Accélérateur**, pas substitut

### Pour rester compétent :

- Savoir écrire un script **sans IA**
- Lire et comprendre : logs bruts, manifests YAML, erreurs kernel / réseau...
- Travailler son **intuition opérationnelle**
  - l'IA est **faible** sur :
    - signaux faibles
    - incidents atypiques
    - contextes humains / organisationnels

## 3.2 Nouvelles exigences transverses

Les organisations modernes doivent intégrer de nouvelles préoccupations **dès la conception et dans tous les métiers du cycle de vie logiciel.**

En tant que futur encadrant, vous serez également responsable :

- De la **maîtrise des coûts** (surtout Cloud)
- De la **qualité de l'observabilité** et de sa **conformité réglementaire**
- Des **arbitrages entre sécurité, performance et vie privée**
- De l'**impact environnemental** des pratiques techniques (**écoconception**)

 L'objectif est de concevoir des systèmes à la fois **robustes, responsables et durables.**



## FinOps – Optimisation des coûts cloud

- Suivi et **pilotage des dépenses cloud** par les équipes techniques
- Budgétisation, alertes, accountability, optimisation multi-cloud
- Collaboration Dev + Finances + Ops



## Intérêt économique des tests et de la CI/CD

### ✓ Le coût d'un bug augmente fortement avec le temps

- Idée clé : plus un défaut est détecté tard, plus il est coûteux (contexte, coordination, correctifs, rollback, impact clients)
- Les ratios exacts varient beaucoup (domaines, criticité, orga) ; on voit souvent des ordres de grandeur **x5 à x100** selon les phases
- Les boucles longues favorisent l'accumulation de dette technique invisible

## \$ Coût d'un bug avec la phase du projet

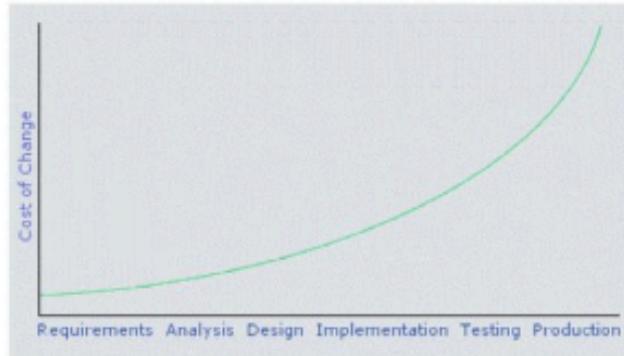
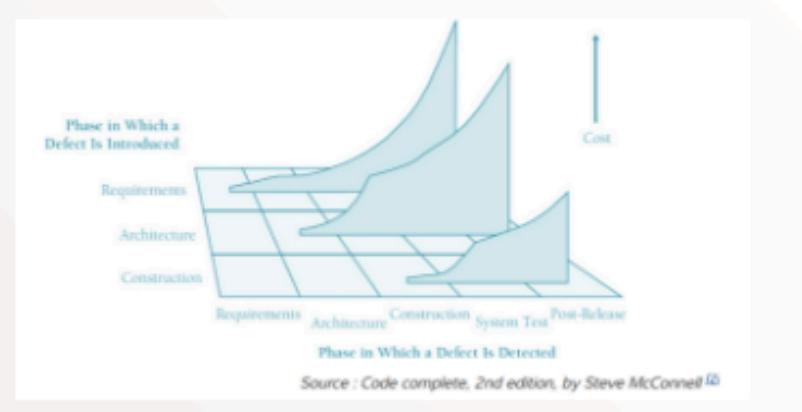


Fig. 4. The cost of change rising exponentially over time (Beck [1999])



Applied Software Measurement, Capers Jones, 1996  
Building Security Into The Software Life Cycle, Marco M. Morana, 2006

Règle des 10 mins max : boucle de rétroaction maximale pour un pipeline



Source : Code complete, 2nd edition, by Steve McConnell<sup>15</sup>

## Règle d'or : la **rétroaction rapide**

 "10 minutes max" : temps idéal pour un retour complet dans un pipeline

 Objectif :

- Identifier les erreurs **immédiatement après le commit**
- Corriger à chaud → développeur encore “dans le flow”
- Réduire les context switches, les bugs en prod, et la frustration

 Une CI/CD bien conçue = investissement qui **diminue les coûts cachés**

## Éthique de l'observabilité

**“Ce que l'on mesure et stocke doit être justifié, maîtrisé et proportionné.”**

- Les **logs** contiennent souvent des **données personnelles** (IP, identifiants, comportements)
- Le **RGPD** impose une **justification claire**, une **durée de conservation limitée**, et un **usage proportionné**
- L'**observabilité** doit soutenir le **bon fonctionnement** du système, **sans surveillance excessive**
- **Journaliser uniquement** ce qui est **utile** à la supervision ou à l'analyse post-mortem
- **Éviter** les logs contenant des **données sensibles** ou **nominatives**
- Mettre en place des **politiques de rétention** et de **rotation des journaux**
- **Restreindre l'accès** aux journaux et **auditer** leur consultation

## GreenOps – Réduction de l'empreinte environnementale

- Réduction de la **consommation énergétique des services IT**
- Optimisation des ressources (CPU, stockage, cloud idle, etc.)
- Sensibilisation à la **sobriété numérique**

Le rôle de l'encadrant inclut également une **responsabilité environnementale** :

- **Réduire les volumes de logs, métriques et traces** à l'essentiel
- **Limiter la durée de conservation** des données de monitoring et sauvegardes
- **Éviter la redondance fonctionnelle** entre outils
- **Favoriser des solutions sobres** en énergie et en ressources

## 3.3 La Formation continue

⚠ Dans le monde du DevSecOps, les technologies évoluent encore plus vite que dans le domaine du DEV. Des formations régulières sont indispensables. Prévoir *a minima* des **MOOCs** (Udemy, Coursera, edX...) réguliers.

### Certifications clés dans le cloud, le DevOps et la sécurité

De nombreuses certifications permettent de **valoriser ses compétences techniques**

et de se spécialiser selon son profil : Cloud, Infrastructure, Sécurité ou DevSecOps.



## Cloud & Infrastructure

- **AWS Certified Solutions Architect / DevOps Engineer**
- **CKA** (Certified Kubernetes Administrator)
  - Pour administrer et gérer des clusters Kubernetes en production
- **CKAD** (Certified Kubernetes Application Developer)
  - Pour concevoir et déployer des applications sur Kubernetes
- **Terraform Associate** (HashiCorp Certified)



## Sécurité

- Formations de l'**ANSSI** comme la formation **ESSI** "Expert en Sécurité des systèmes d'information" ouverte aux organisations vitales pour la nation.
- **CISSP** (Certified Information Systems Security Professional) – niveau expert
- **CEH** (Certified Ethical Hacker) – sécurité offensive



## DevSecOps

- **DevSecOps Foundation** (DevOps Institute)
- **Practical DevSecOps** – plus technique, orienté pipelines CI/CD

## S'auto-évaluer sur la sécurité

### Forces possibles :

- Connaissance des failles classiques (OWASP top 10...)
- Capacité à auditer du code ou des configurations
- Sensibilisation aux enjeux RGPD, traçabilité, logs

### Lacunes fréquentes :

- Sécurité réseau ou infrastructure (TLS, firewalls, cloud IAM)
- CI/CD sécurisé, gestion des secrets
- Modélisation des menaces, tests d'intrusion, audit



## 3.4 Sécurité, culture d'équipe, et responsabilité technique

Aujourd'hui, un encadrant technique ne peut plus ignorer la **sécurité**

Elle se construit dans la **culture d'équipe** et les **choix technologiques**



## Gouvernance de la sécurité dans l'équipe

- Clarifier les responsabilités : **qui fait quoi en sécurité ?**
- Mettre en place des **rituels sécurité** : revue de code, threat modeling, revue des exceptions CVE, ...
- **Anticiper les risques humains** : erreurs, négligence, turnover
- Être le garant des **bonnes pratiques** (pas le pompier)



## Rôles dans l'équipe

- **Développeurs** : appliquent les bonnes pratiques de codage sécurisé
- **DevOps / SRE** : gèrent la sécurité infra, secrets, CI/CD
- **Encadrant / Tech Lead** : impulse la direction, arbitre les choix
- **Product Owner** : comprend les enjeux sécurité liés au métier

## Choix technologiques orientés sécurité

- Frameworks avec protections intégrées (CSRF, XSS, injections...)
- Authentification et chiffrement dès la conception
- Traçabilité : surveillance, logs, audit trail
- Choix d'outils maintenus, documentés, et **audités**



## Culture sécurité by design

- Sécurité = qualité → **intégrée dès le départ** (« Shift to the Left »)
- **User stories** incluant sécurité (ex : rôles, gestion des données sensibles)
- Vérifications automatiques SAST/DAST/SCA dans **CI/CD**
- **Formation continue** : Capture The Flag (CTF) internes, katas sécurité (cas d'école à corriger), analyse de post-mortems, ...



## Sécurité au quotidien

- Exemples de **rituels** :
  - Atelier “**Sécurité du mois**” (présentation d'un problème de PROD par exemple)
  - Focus sécurité en **rétrospective**
  - **Mini audits en peer review**
- Montrer l'exemple : **ne jamais merger un code douteux**, même sous pression



## Posture de l'encadrant responsable

- Cultiver un climat de confiance : “on peut parler de faille sans crainte”, **blameless**.
- Rendre la sécurité **visible** : KPIs, alertes, dashboards
- Défendre les **chantiers sécurité/dette technique** face aux priorités business
- **Valoriser les efforts invisibles** autour de la sécurité



## 3.5 Ressources pour approfondir

- [OWASP.org](#) : Top 10, Cheat Sheets, outils de test
- [DevSecOps.org](#) : principes, pratiques, retours terrain
- [CNCF Security TAG](#) : bonnes pratiques cloud native
- [HackTricks](#) : encyclopédie sécurité offensive/défensive
- Cybrary, Root Me, TryHackMe... : entraînement pratique (CTF, labs)
- Micode "[la Fabrique à idiots \(2026\)](#)" : Documentaire sur l'utilisation de l'IA pour coder