

Introduction à DevOps

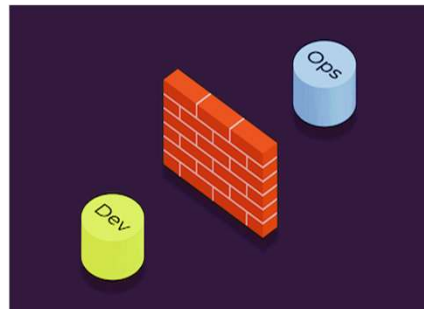
SAMIA NASIRI



Introduction

L'équipe de développement, a pour objectif principal de **faire évoluer l'application**

- **l'équipe des opérations**, a comme objectif de **maintenir l'application en conditions opérationnelles**.
- les **deux équipes** ont des objectifs respectifs antagonistes.



DEV vs OPS



- ▶ Relation entre **Dev** et **Ops**:
 - ▶ **Dev**: Equipes de développeurs logiciels
 - ▶ **Ops**: Equipes en charge de la mise en production des produits
- ▶ Antagonisme fort:
 - ▶ **Dev**: Modifications aux moindres coûts, le plus rapidement possible
 - ▶ **Ops**: Stabilité du système, qualité

DEV



Quels changements ?

- ▶ ajout ou évolution de **fonctionnalités**,
- ▶ correction de **bugs**,
- ▶ amélioration de la **performance**,
- ▶ ajustement suite au **feedback utilisateur**,
- ▶ adaptation à de **nouvelles exigences métier**.

OPS



Les équipes **OPS** sont responsables de :

- ▶ la **disponibilité** du système (ça doit marcher, tout le temps),
- ▶ la **stabilité** (pas de pannes, pas de régressions),
- ▶ la **qualité de service** (performance, sécurité, fiabilité).
- ▶ Objectif principale :

Garantir que le système fonctionne correctement en production.

Avant DevOps « une organisation en silos »

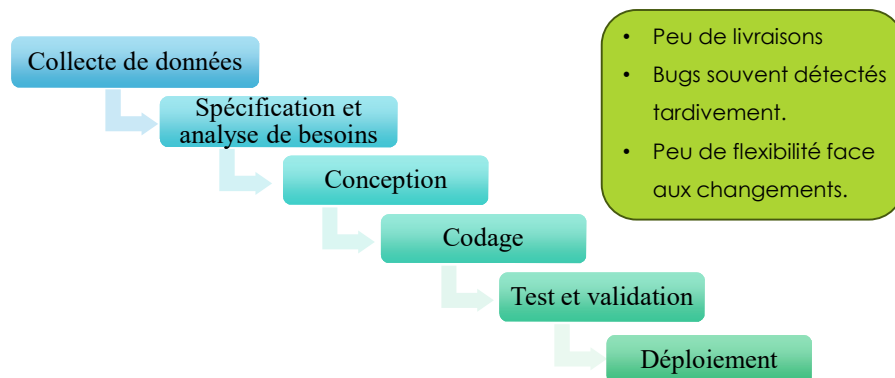


- ▶ Cloisonnement entre les équipes : Développement et Opérations.
 - ▶ **Développement** et **Opérations** travaillent de manière **séparée**, avec peu de collaboration et de communication entre elles.
 - ▶ Chacune se concentre uniquement sur ses propres objectifs, sans vision globale du cycle de vie du logiciel.
- ▶ Les développeurs livrent le code, puis la responsabilité de la mise en production est confiée aux équipes Ops.

DEV vs OPS

DEV	OPS
Développement continu et évolutif du logiciel	Eviter les incidents
Priorité : nouvelles fonctionnalités	Priorité : stabilité
Succès mesuré par la livraison	Succès mesuré par l'absence d'incidents

Cycle de vie classique



Limites du cycle classique

- ▶ Développement logiciel en plusieurs **phases séparées**.
- ▶ Long délai entre le besoin et la livraison.
- ▶ Déploiements lourds.
- ▶ Équipes cloisonnées : Développement / Exploitation.
- ▶ Communication limitée entre équipes.
- ▶ Beaucoup d'actions manuelles.
- ▶ Faible satisfaction client car **feedback tardif**.



Inefficacité globale et faible satisfaction client

Problème N°1 : Délais de livraison

- ▶ Application terminée côté développement: Le code est prêt, mais le produit n'est pas encore livré
- ▶ Serveur non prêt côté exploitation
- ▶ Mise en production reportée plusieurs fois



Retard, frustration, perte de valeur

Problème 2: Déploiements lents et risqués

- ▶ Avant DevOps :
 - ▶ Déploiement = tâche manuelle, souvent la nuit ou le weekend pour limiter l'impact sur les utilisateurs.
 - ▶ Scripts hétérogènes, documentations incomplètes.
 - ▶ Chaque mise en production = **stress** et **interruption du service** possible.
- ▶ Conséquence :
 - ▶ Peu de livraisons par an (parfois seulement 2–3 grosses releases annuelles).
 - ▶ Gros “big bang” déploiements, difficiles à maîtriser.

Problème 3: Bugs découverts tardivement

- ▶ Tests souvent effectués **après le développement complet**.
- ▶ Environnement de test \neq environnement de production (incohérences).
- ▶ Débogage long et coûteux → correction tardive.
- ▶ Conséquence :
 - ▶ Bugs critiques détectés seulement en production.
 - ▶ Perte de temps, augmentation des coûts, frustration client.

Problème 4 : Mauvaise communication entre équipes

- ▶ Équipes Dev et Ops travaillent séparément, sans réelle collaboration.
- ▶ Peu ou pas de partage de responsabilités.
- ▶ Chacun blâme l'autre :
 - ▶ Dev : “Ça marche sur ma machine !”
 - ▶ Ops : “Votre code casse la production !”
- ▶ Conséquence : perte de confiance, ralentissement global du projet.

Côté Développement (Dev)

« Ça marche sur ma machine ! »

- ▶ Le développeur a testé l'application **uniquement dans son environnement local**
- ▶ Il considère que le travail est terminé dès que le code fonctionne chez lui
- ▶ Les différences entre environnements (versions, configuration, serveurs) sont ignorées.

Côté Exploitation (Ops)

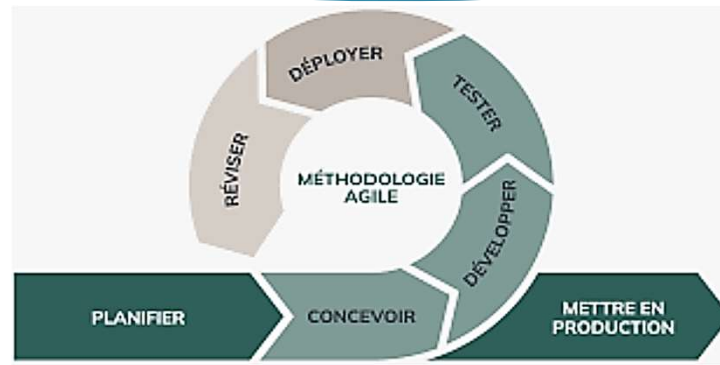
« **Votre code casse la production !** »

- ▶ Le code provoque des erreurs **en environnement réel**
- ▶ L'équipe Ops est responsable de la **stabilité du système**
- ▶ Elle reçoit le logiciel tardivement, sans implication préalable de l'exploitation.

Apparition d'Agile

- ▶ Réponse aux limites du cycle classique.
- ▶ **Principes** (Manifeste Agile) :
 - ▶ Collaboration avec le client.
 - ▶ Livraisons fréquentes (sprints courts).
 - ▶ Adaptabilité face aux changements.
- ▶ Exemples : **Scrum, Kanban.**
- ▶ *Tableau Kanban ou schéma de sprint Agile*

Agile



- ▶ **Itératif**: Répéter le processus à chaque **sprint** pour améliorer le produit
- ▶ **Incrémental**: Ajouter des fonctionnalités progressivement, livrables à chaque sprint

Agile:Notions essentielles

- ▶ **Sprint** : Itération de durée fixe (1 à 4 semaines)
 - ▶ Contient analyse, développement et tests
 - ▶ Produit un **incrément de logiciel fonctionnel**
- ▶ **Exemple**
 - ▶ Sprint 1 : connexion utilisateur
 - ▶ Sprint 2 : consultation du solde
 - ▶ Sprint 3 : virement bancaire
- ▶ Chaque **sprint** produit un **incrément**, **potentiellement livrable**, mais pas forcément une version complète.

Agile : Notions essentielles

- ▶ **User Story** : besoin simple exprimé du point de vue utilisateur

Exemple :

En tant que client, **je veux** consulter mon solde **pour** savoir combien je peux dépenser

- ▶ **Backlog** : liste priorisée des fonctionnalités
- ▶ **Product Owner (PO)** : Il représente les besoins du client et des utilisateurs,
 - ▶ priorise le backlog en fonction de la valeur apportée,
 - ▶ et s'assure que les fonctionnalités développées répondent aux objectifs du projet.
- ▶ Les réunions courtes, appelées **Daily meetings** ou Daily stand-up, sont des réunions quotidiennes et limitées dans le temps (10 à 15 minutes),
 - ▶ visant à aligner rapidement l'équipe,
 - ▶ à suivre l'avancement des travaux
 - ▶ et à identifier précocement les obstacles afin de favoriser une réaction immédiate.

Limites d'Agile seul

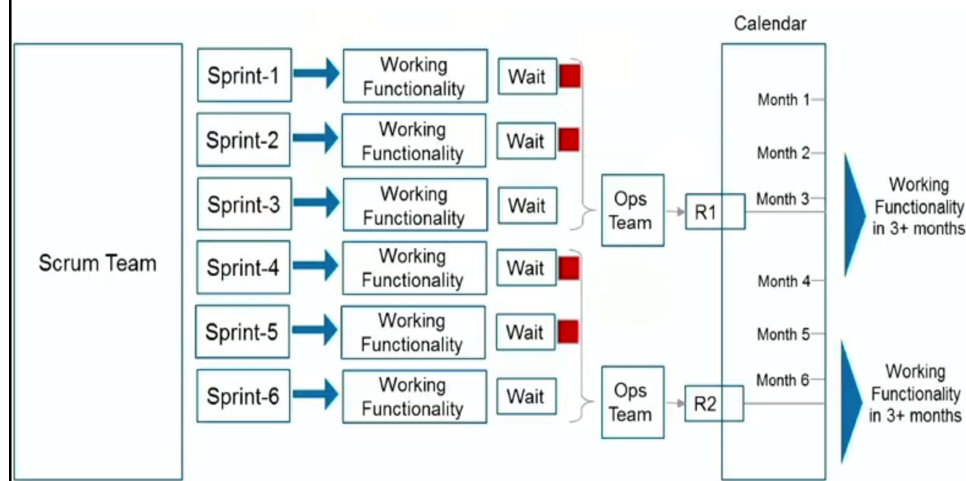
Agile améliore le **développement**, mais:

- ▶ Le **déploiement** reste souvent manuel.
- ▶ Les **Ops** (infrastructure, production) restent isolés.
- ▶ Risque de friction entre équipes Dev et Ops.

Problème : Agile limité sans Ops agiles

- ▶ L'équipe **Scrum** développe en **sprints courts**
- ▶ À chaque sprint, une **fonctionnalité fonctionnelle** est prête
- ▶ Pourtant, elle **n'est pas livrée immédiatement**
- ▶ **Où ça bloque ?**
 - Après chaque sprint : état **"Wait"**
 - Dépendance à l'**équipe Opérations**
 - Déploiement soumis à un **calendrier de releases** (mensuel / trimestriel)

Agile



Problème : Agile limité sans Ops agiles

► Conséquences

- Accumulation de fonctionnalités non livrées
- Livraison en production après **3+ mois**
- Feedback utilisateur très tardif

► Agilité stoppée après le développement

► Scrum rend le développement rapide, mais **sans Ops agiles, la valeur n'est pas livrée.**

Limites d'Agile seul

► Agile apporte:

- Développement itératif par **sprints**
- Collaboration Dev / Métier
- Livraison rapide du **code**
- Amélioration continue

► Les problèmes rencontrés

- Les **opérations restent traditionnelles**
- Déploiements via **tickets manuels**
- Temps d'attente après la fin du développement
- Rupture entre **fin du sprint** et **mise en production**
- Les développeurs **attendent après les Ops**

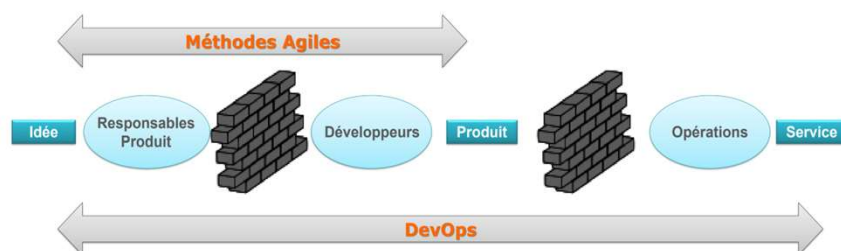
DevOps : la solution naturelle

Objectif de DevOps

Étendre l'agilité jusqu'à la production en alignant
Développement + Opérations.

De l'Agile au DevOps – Une continuité naturelle

Devops est un mouvement qui étend Agile en intégrant les opérations dans les itérations



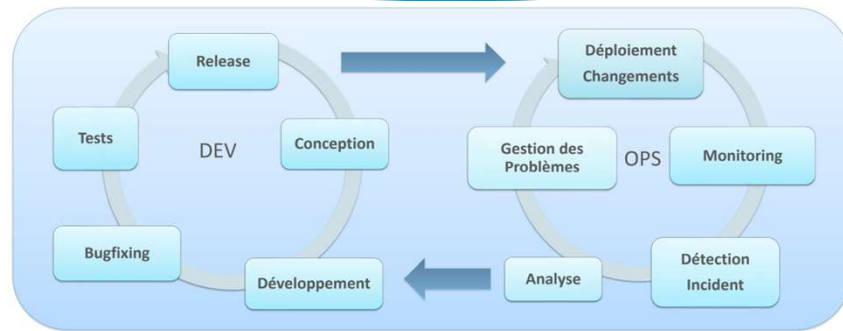
De l'Agile au DevOps – Une continuité naturelle

- ▶ Les méthodes **Agiles** favorisent la collaboration entre les responsables produit et les développeurs.
- ▶ Elles visent à livrer rapidement un produit fonctionnel et adapté aux besoins clients.
- ▶ Cependant, la phase d'exploitation reste souvent isolée du cycle Agile.
- ▶ Le mouvement **DevOps** vient prolonger cette logique en intégrant les **opérations** dans les itérations de développement.
- ▶ Objectif : créer un **flux continu de valeur**, du concept initial jusqu'au service rendu à l'utilisateur final.

Agile-Devops

- ▶ Dans DevOps, les Ops ne sont plus de simples exécutants de fin de chaîne. Ils deviennent **acteurs du cycle Agile**, co-responsables du produit **de la conception à la supervision**.
- ▶ Les équipes d'exploitation participent dès la phase de conception pour anticiper les contraintes d'infrastructure, automatiser les déploiements et garantir la stabilité du service dès les premiers sprints.

DevOps



- Le schéma représente le **cycle DevOps**, qui relie en continu les équipes **DEV** (développement) et **OPS** (opérations).
- **Deux boucles** : une boucle **DEV** + une boucle **OPS** reliées par des flèches → c'est la collaboration continue.

DevOps

- DevOps s'appuie sur un cycle continu dans lequel:
 - les équipes **DEV** développent, testent et livrent des versions du logiciel,
 - les équipes **OPS** les déploient, les surveillent et gèrent les incidents.
 - Les retours de la production alimentent ensuite le développement afin d'améliorer continuellement la qualité et la rapidité de livraison.
- L'objectif : **livrer plus vite, avec plus de qualité, et s'améliorer en continu** grâce aux retours de la production.

DevOps

- ▶ DevOps s'appuie sur un cycle de vie **continu** basé sur l'intégration, le déploiement et le feedback.
- ▶ DevOps fournit un cadre de pratiques qui accompagne tout le cycle de vie logiciel.

Devops

Devops Combine:

- ▶ **Culture organisationnelle** (collaboration Dev/Ops),
- ▶ **Pratiques** (automatisation, CI/CD, tests continus),
- ▶ **Outils** (Git, Docker, etc.).

Transition vers DevOps

- ▶ DevOps propose :
 - ▶ Plus de **collaboration** (Dev + Ops travaillent ensemble).
 - ▶ Plus d'**automatisation** (tests, déploiements, monitoring).
 - ▶ Une **approche continue** pour livrer plus vite et plus sûr.

Cycle de vie DevOps

Étapes principales :

1. Planification
2. Développement
3. Intégration et tests (CI)
4. Livraison et déploiement (CD)
5. Exploitation et supervision
6. Feedback et amélioration continue

Principes clés du DevOps

- ▶ Collaboration et communication
- ▶ Automatisation des processus répétitifs
- ▶ Intégration continue (CI)
- ▶ Déploiement continu (CD)
- ▶ Infrastructure as Code (IaC)
- ▶ Monitoring et feedback en continu

Composants technologiques du DevOps

- ▶ **Gestion de versions** : permet de centraliser le code et de collaborer efficacement « Git, GitHub, GitLab »
- ▶ **CI/CD** : automatisent l'intégration, les tests et le déploiement des applications « GitHub Actions, Jenkins, GitLab CI »
- ▶ **Conteneurisation** : garantit des environnements cohérents du développement à la production « Docker, Podman »
- ▶ **Orchestration** : permet de gérer automatiquement l'exécution, la coordination et la disponibilité de plusieurs services d'une application: Kubernetes, Docker Swarm
- ▶ **Monitoring** : permet de surveiller en continu le fonctionnement d'une application et de son infrastructure « Prometheus, Grafana »
- ▶ **Infrastructure as Code** : gérer et de reproduire les infrastructures de manière automatisée et fiable « Terraform, Ansible »

DevOps

- ▶ Cycle **classique** → **rigide** et **lent**.
- ▶ **Agile** → flexibilité dans le **développement**.
- ▶ **DevOps** → **automatisation** + **collaboration** → vision complète, **continue** et moderne.

Avec DevOps : implication des Ops dès le départ

- ▶ Le but est de **construire ensemble un produit qui fonctionne aussi bien en production qu'en développement**.
- ▶ **Les Ops participent à :**
 1. **La conception du produit**
 - ▶ Ils aident à **anticiper les besoins d'infrastructure** : hébergement, sécurité, performance, capacité réseau, monitoring...
 - Exemple :
 - ▶ Lorsqu'un nouveau service est conçu, les Ops peuvent recommander une architecture scalable (ex : conteneurs, microservices, cloud auto-scalable) pour éviter les problèmes futurs.
 - ▶ Ils s'assurent que le produit soit **"operable"** dès sa conception (facile à déployer, à surveiller, à réparer).

Avec DevOps : implication des Ops dès le départ

2. La planification des sprints

- ▶ Les Ops participent aux **réunions de planification** pour :
 - ▶ Identifier les **tâches d'infrastructure** à inclure dans le sprint (CI/CD, scripts, monitoring, logs...).
 - ▶ Synchroniser les priorités : les Devs livrent des fonctionnalités, les Ops automatisent les déploiements et la supervision.

▶ Exemple :

Pendant le sprint, on développe à la fois une nouvelle fonctionnalité **et** le pipeline d'intégration continue qui la déploiera automatiquement.

Avec DevOps : implication des Ops dès le départ

3. La définition des critères de livraison et de déploiement

- ▶ Ensemble, Devs + Ops définissent les **“Done” élargis** :
 - ▶ “Code compilé”
 - ▶ “Tests unitaires passés”
 - ▶ “Déploiement automatisé validé”
 - ▶ “Supervision configurée”
- ▶ Cela garantit que **la livraison inclut la production** dans sa définition du succès.

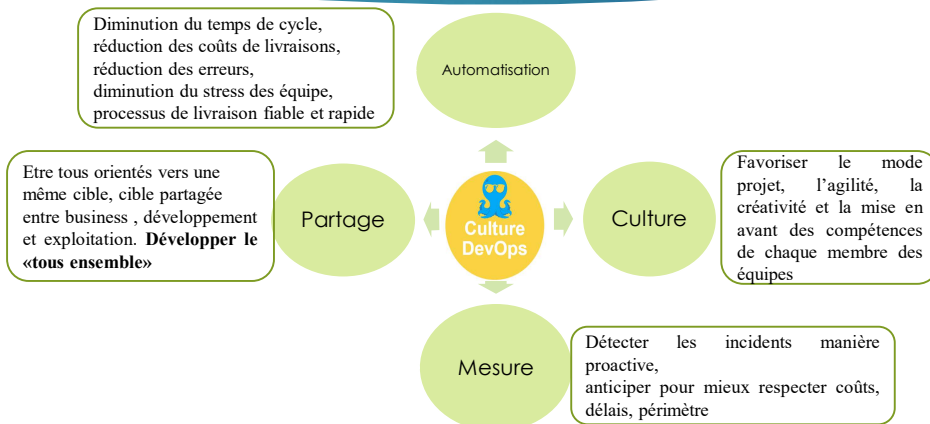
Exemple :

- ▶ Une fonctionnalité n'est pas considérée comme finie **“Done”** tant qu'elle n'est pas déployée et monitorée en environnement réel.

Avec DevOps : une chaîne fluide et intégrée

- Suppression des murs entre les équipes pour une **collaboration continue**.
- Les opérations sont intégrées dès la conception du produit.
- Les développeurs et les Ops partagent la **responsabilité du service en production**.
- Passage d'un modèle séquentiel à un modèle **itératif et automatisé (CI/CD)**.
- Meilleure synchronisation entre conception, développement, déploiement et supervision.

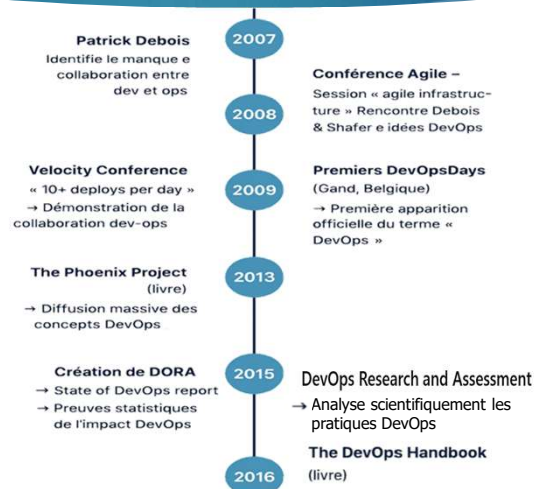
Définition



DevOps

- ▶ DevOps ne se limite pas à la collaboration entre les équipes Dev et Ops tout en restant dans des silos distincts.
- ▶ Le DevOps est un changement culturel dans lequel les ingénieurs de développement et d'exploitation **travaillent ensemble tout au long du cycle de développement.**

Devops: Historique



Avantages à adopter l'approche DevOps

- ▶ Accélération et amélioration de la livraison des produits
- ▶ Résolution plus rapide des problèmes et complexité réduite
- ▶ Stabilité accrue des environnements d'exploitation
- ▶ Meilleure utilisation des ressources
- ▶ Automatisation accrue
- ▶ Meilleure visibilité sur les résultats du système
- ▶ Innovation renforcée

Pourquoi gérer les versions ?

- ▶ Dans tout projet logiciel :
 - Plusieurs personnes travaillent en même temps.
 - Les fichiers changent constamment.
 - Il faut garder un **historique fiable**.
- ▶ Sans gestion de versions :
 - Risque de perdre des modifications.
 - Difficile de savoir *qui a changé quoi et quand*.
 - Collaboration compliquée.

Pourquoi gérer les versions ?

Identifier rapidement l'origine d'un problème

En cas de bug ou d'incident en production :

- ▶ **retracer le changement responsable,**
- ▶ comparer les différentes versions du code,
- ▶ comprendre la cause de la défaillance du système.

Git

- ▶ Git est un système de contrôle de version distribué utilisé pour **suivre les modifications apportées au contenu.**
- ▶ Git permet de suivre **l'évolution du code** et de **collaborer** facilement.
- ▶ Le projet est sauvegardé sous forme de versions appelées *commits*.
- ▶ Chaque **commit** = une **version du projet**.
- ▶ Chaque développeur a une copie complète du projet sur son ordinateur.
- ▶ Les modifications sont enregistrées localement et partagées entre dépôts.

Git

- ▶ Dans Git, chaque dépôt peut servir de point central d'échange. Lorsque Git est utilisé correctement, il existe une branche principale qui correspond au code déployable.
- ▶ La **branche principale** est la **version stable** et fiable du projet.
- ▶ Les équipes peuvent intégrer en continu les modifications prêtes à être publiées et travailler simultanément sur des branches distinctes entre les publications.
- ▶ Git seul permet de gérer les versions **localement**. Pour collaborer avec d'autres développeurs, on utilise un dépôt distant sur GitHub, GitLab...

Github

- ▶ GitHub est un service d'hébergement en ligne pour les dépôts Git.
- ▶ GitHub est hébergé par une filiale de Microsoft.
- ▶ GitHub propose des comptes gratuits, professionnels et d'entreprise.
- ▶ En août 2019, GitHub comptait plus de 100 millions de dépôts.
- ▶ Un dépôt (**repository**) est une structure de données permettant de stocker des documents, notamment le code source d'une application.

Gitlab

- ▶ GitLab:
 - une plateforme DevOps complète, fournie sous la forme d'une application unique.
 - donne accès à des référentiels Git, permettant la gestion du code source.
- ▶ GitLab permet aux développeurs :
 - collaborer,
 - réviser le code,
 - faire des commentaires et s'entraider pour améliorer le code de chacun.
 - Travailler à partir de leur propre copie locale du code.
 - Créer des branches et fusionner le code si nécessaire.
 - Rationaliser les tests et la livraison grâce à l'intégration continue (CI) et à la livraison continue (CD) intégrées.

Pourquoi utiliser Git ?

- ▶ Avant Git, les équipes géraient les versions de fichiers **manuellement** :
- ▶ Chaque membre enregistrait ses propres copies :
rapport_v1.doc, rapport_final_V3.doc,
rapport_final_definitif_v4(1).docx
- ▶ Difficile de savoir **qui a modifié quoi, quand, et pourquoi.**
- ▶ Le travail en équipe créait des conflits : deux personnes modifiaient le même fichier sans coordination.
- ▶ La traçabilité et la collaboration étaient difficiles à gérer: fichiers dupliqués, conflits de versions, pertes de modifications

Exemple 1

Imaginons une équipe de trois personnes travaillant sur le même projet.

- ▶ Chacun téléchargeait une **copie locale** du projet (par e-mail, clé USB ou serveur partagé).
- ▶ Chacun modifiait le code ou le document de son côté.
- ▶ En fin de journée, il fallait **rassembler les fichiers** pour créer une version commune.



- ▶ plusieurs fichiers nommés projet_final_v1.doc, projet_final_v2_Mohamed.doc, projet_final_def_v3_karam.docx...
- ▶ des pertes de modifications,
- ▶ des doublons,
- ▶ et beaucoup de confusion.

Exemple 2 : fichier partagé sur un serveur (accès distant simultané)

- ▶ Le projet (ex : rapport.docx) est stocké sur un **serveur commun**
- ▶ Deux personnes (par exemple, **Nada** et **Doha**) y accèdent **depuis leurs ordinateurs**.

Étapes du problème

- ▶ **Nada** ouvre le fichier et commence à écrire.
- ▶ **Doha** ouvre le même fichier avant que Nada n'ait enregistré ses changements.
- ▶ Chacun modifie une partie différente du document.
- ▶ **Nada enregistre** → les changements sont bien écrits sur le serveur.
- ▶ **Doha enregistre ensuite** → sa version remplace complètement celle de Nada.

Exemple 2 : fichier partagé sur un serveur - Résultat

- ▶ Les modifications de Nada sont effacées.
- ▶ Le fichier final contient uniquement le travail de Doha.
- ▶ Aucune alerte ni historique ne permet de savoir ce qui a été perdu.
- ▶ **Accès simultané à un fichier → risques de conflits et d'écrasement des modifications**

Sans gestion de versions - Problèmes

- ▶ **Scénario catastrophe :**
 - mon_projet.zip
 - mon_projet_v2.zip
 - mon_projet_v2_final.zip
 - mon_projet_v2_final_VRAIMENT_final.zip
- ▶ **Problèmes :**
 - Quelle est la bonne version ?
 - Qui a fait quelle modification ?
 - Comment revenir en arrière si quelque chose casse ?
 - Comment travailler à plusieurs sans tout écraser ?

La solution : Git

- ▶ **Git** est un système de gestion de versions qui permet de :
 - Enregistrer l'historique complet de votre projet
 - Revenir à n'importe quelle version précédente
 - Travailler en parallèle sur différentes fonctionnalités (branches)
 - Collaborer **sans conflit** avec d'autres développeurs
- ▶ **Analogie : Git** est comme une machine à remonter le temps pour votre code. Chaque "**photo**" (**commit**) capture l'état exact de votre projet à un **moment donné**.

Git

Pour chaque version d'un document, nous devons savoir :

- ▶ **Quand** le fichier a été modifié
- ▶ **Quelles modifications** ont été apportées
- ▶ **Pourquoi** il a été modifié
- ▶ **Qui** a effectué la modification

VCS (Version Control System)

- ▶ **Qu'est-ce qu'un VCS (Version Control System) ?**
- ▶ **Définition :**
Outil qui garde l'historique complet des fichiers d'un projet.
- ▶ Permet de :
 - ▶ Sauvegarder chaque modification.
 - ▶ Collaborer en équipe.
 - ▶ Comparer différentes versions.
 - ▶ Revenir à une version stable en cas de bug.

Avantages d'un VCS

- ▶ **Historique complet** → traçabilité des changements.
- ▶ **Collaboration** → plusieurs développeurs en parallèle.
- ▶ **Expérimentation** → possibilité de créer des branches.
- ▶ **Sécurité** → revenir facilement à une version plus stable

Concepts Fondamentaux de Git

- ▶ **Le Dépôt (Repository)**
- ▶ **Qu'est-ce qu'un dépôt Git ?**

Définition : Un dépôt (**repository**) est un espace de stockage qui contient :

- ▶ Tous les fichiers du projet
- ▶ L'historique complet des modifications
- ▶ Toutes les branches de développement
- ▶ Les configurations Git

Types de dépôts

1. **Dépôt Local:**
 - Sur votre ordinateur personnel
 - Accessible uniquement par vous
 - Créé avec : `git init`
2. **Dépôt Distant (Remote)**
 - Hébergé sur un serveur (GitHub, GitLab, etc.)
 - Accessible par toute l'équipe
 - Point de synchronisation central

Différence entre main et origin/main

Élément	Local ou Distant	Description
main	Locale	Branche sur ton ordinateur, où tu fais des commits.
origin/main	Locale (copie du distant)	Copie locale de la branche main du dépôt distant.
main (sur GitHub)	Distant	Branche hébergée sur GitHub (ou GitLab, etc.).

Initialiser un Dépôt

- Créer un nouveau dépôt:

```
mkdir mon-projet
cd mon-projet
git init
```

- Cloner un dépôt existant

```
git clone https://github.com/user/projet.git
```

Qu'est-ce qu'un commit ?

- ▶ Un commit est une "photographie" du projet à un instant T
- ▶ Imaginez votre projet comme un album photo :
- ▶ Chaque photo = un commit
- ▶ L'album = l'historique Git
- ▶ Chaque photo a une légende = le message du commit

Commit

Pourquoi les commits sont essentiels ?

- ▶ **Traçabilité** : Savoir qui a fait quoi et quand
- ▶ **Réversibilité** : Revenir en arrière si problème
- ▶ **Documentation** : Comprendre l'évolution du projet
- ▶ **Collaboration** : Partager les modifications avec l'équipe

Commit : message

Quelques bonnes pratiques lors de la rédaction d'un commentaire :

- ▶ ne terminez pas le message de commentaire par un **point**,
- ▶ limitez les messages de commentaire à moins de 50 caractères.

Commandes git

- ▶ Voir les modifications: `git status`

- ▶ Ajouter des fichiers au staging:

`git add` prépare les modifications avant la création d'un commit.

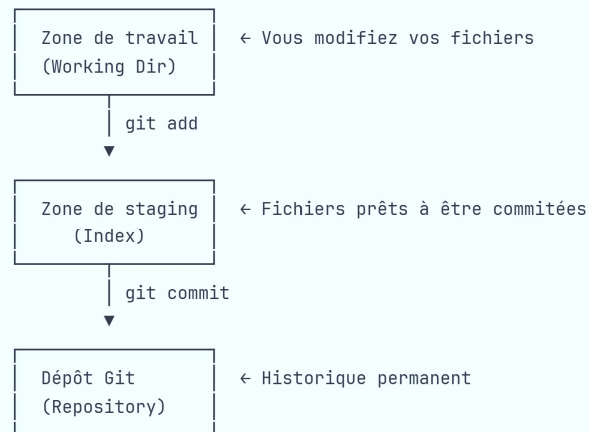
`git add` fichier.py

`git add .` # Tous les fichiers

- ▶ Créer le commit:

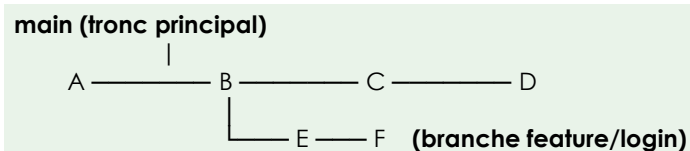
`git commit -m` "Message descriptif"

Le Cycle de Vie d'un Commit



Branche

- **Branche:** une **ligne de développement indépendante** qui permet de travailler sur plusieurs fonctionnalités en parallèle sans affecter le code principal.
- Tronc (main) : Version stable, production.
- Branches : Développements en cours
- Feuilles : Points de développement actifs



Branche

Le principe fondamental: Une branche permet d'expérimenter sans risque

- Si ça marche → fusion dans main
- Si ça ne marche pas → suppression de la branche

GIT: branch

- ▶ Une **branche (branch)** sert à ce que **chaque membre du projet puisse développer séparément**, puis à **fusionner (merge)** son travail avec celui des autres, sans casser le code des collègues ni la version stable.
- ▶ Quand plusieurs développeurs travaillent ensemble, il faut **éviter les interférences**.
- ▶ Sans branches, tout le monde modifierait les mêmes fichiers sur la même version

Résultat : **conflits, pertes de code, bugs**

Scénario sans branches

- **Problème** : Tout le monde travaille sur main

```
Développeur A : Ajoute une fonctionnalité (non testée)
↓
Code cassé poussé sur main
↓
Développeur B : Ne peut plus travailler
↓
Production : Application en panne
```

Scénario avec branches

```
main (toujours stable)
|
├── feature/paiement (Dev A travaille ici)
├── feature/profil (Dev B travaille ici)
└── hotfix/bug-urgent (Dev C corrige ici)
```

GIT: branch

Objectif	Ancienne commande	Nouvelle commande	Effet
Créer une branche sans changer	git branch nom	—	Crée seulement
Créer + basculer	git checkout -b nom	git switch -c nom	Crée et bascule
Juste changer de branche	git checkout nom	git switch nom	Bascule uniquement

Avantages de Branche

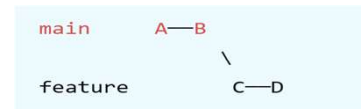
- ▶ *Main* reste **toujours stable**
- ▶ Chacun travaille **en isolation**
- ▶ Tests effectués **avant fusion**
- ▶ Production **jamais impactée** par le code en développement

Git: merge

- ▶ *Merge*: permet de **fusionner les modifications d'une branche** dans une autre tout en **préservant l'historique des commits**.
- ▶ Pourquoi *merger*?
 - Intégrer une nouvelle fonctionnalité,
 - Combiner le travail de plusieurs développeurs,
 - Mettre à jour la branche main,
 - Synchroniser les branches de développement

Git Merge – Fast-Forward

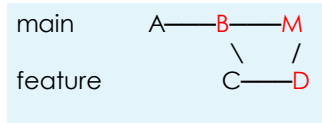
- ▶ **Fast-Forward**: merge automatique quand main n'a pas évolué pendant le travail sur la branche.
 - ▶ La commande de merge: `git merge feature`
- ↓
- ▶ Git **avance simplement** le pointeur de main.
 - ▶ **Aucun commit de merge** n'est créé.
 - ▶ Historique **linéaire et propre**.



Git: merge

- L'option de merge: **no-ff**

- Git merge feature no-ff



- Le commit M est un commit de merge ayant deux parents : le dernier commit de la branche *main* (B) et le dernier commit de la branche *feature* (D).
- Les commits intermédiaires (comme C) sont inclus implicitement.
 - Force la création d'un **commit de merge**
 - **Conserver la trace** de la branche

Git: merge

Type	Description	Visuel / Résumé
Fast-Forward	Pas de divergence → avance linéaire du pointeur	Historique linéaire (A→B→C→D)
No-ff	Branches divergentes → création d'un commit de merge	Nouveau commit M avec deux parents
Squash	Combine tous les commits en un seul	Historique simplifié (C+D → M)

Commandes principales:

git switch main

git merge develop

git merge develop **--no-ff**

git merge develop **--squash**

git merge develop **--ff-only**

-----> Force la création d'un commit de merge

-----> Combine tous les commits de la branche fusionnée en un seul

-----> Fusionne uniquement si le fast-forward est possible

Supprimer une branche

- ▶ Supprimer une branche locale fusionnée:
`git branch -d nom_branche`
- ▶ Supprimer une branche locale non fusionnée:
`git branch -D develop`
- ▶ Supprimer branche distante:
`git push origin --delete nom_branche`

Git remote

- ▶ Commande: `Git remote add origin url`
- ▶ Associe le dépôt local à un dépôt distant (GitLab, GitHub)
- ▶ origin : nom par défaut du dépôt distant
- ▶ url: adresse du dépôt distant
- ▶ Permet de partager le code avec d'autres collaborateurs
- ▶ Commande nécessaire avant d'utiliser **git push** ou **git pull**

Git Push

- ▶ **git push** permet d'envoyer les commits locaux vers un dépôt distant.
- ▶ Il met à jour la branche distante correspondante (souvent origin/main)
- ▶ **git push origin main** : envoie les commits de la branche locale *main* vers le dépôt distant *origin* afin de les partager avec les autres collaborateurs.
- ▶ L'option **-u** sert à faire le lien une fois pour toutes entre la branche locale et la branche distante.
- ▶ Après le premier push (**git push -u origin main**), les prochains envois peuvent être effectués simplement avec la commande : **git push**

git push

Élément	Rôle
git push	Envoyer les commits locaux vers le dépôt distant
-u	Définir la branche distante par défaut pour suivi futur
origin	Nom du dépôt distant (GitHub)
main	Branche locale que l'on pousse vers le distant

Git status

- ▶ `git status` permet d'afficher l'état actuel du dépôt Git.

Exemple:

- ▶ Elle indique :
- ▶ L'état par rapport au dépôt distant
- ▶ Les fichiers modifiés
- ▶ Les fichiers ajoutés
- ▶ Les fichiers non suivis
- ▶ La branche actuelle

```
On branch main
Your branch is up to date with 'origin/main'

Changes not staged for commit:
  modified:   index.html

Untracked files:
  style.css
```