



# Le versioning avec Git

Elies Jebri



---

# Plan du module

# Git

---

- Introduction & Contexte
- Bases de la gestion de versions
- Concepts fondamentaux de Git
- Commandes essentielles (usage quotidien)
- Travailler en équipe avec Git
- Stratégies de branches
- Git avancé (zones à risque)
- Bonnes pratiques & erreurs courantes
- Git dans l'écosystème DevOps (CI/CD)
- Conclusion & ressources

# Introduction & Contexte

---

## **Objectif du module**

Mettre tout le monde au même niveau **avant** d'entrer dans Git :

Pourquoi Git existe

À quoi il sert **concrètement en entreprise**

Ce qu'il **n'est pas**

# Version Control System



# Pourquoi Git est indispensable aujourd'hui

## Pourquoi a-t-on besoin de Git ?

- Le développement logiciel est un **travail collaboratif**
- Les fichiers évoluent en permanence
- Plusieurs personnes modifient le **même code en parallèle**

## Sans système de gestion de versions :

- Perte d'historique
- Écrasement de code
- Impossible de savoir *qui a fait quoi et pourquoi*

## Git permet de :

- Conserver l'historique complet d'un projet
- Travailler à plusieurs sans se gêner
- Revenir à un état stable à tout moment

# Git dans la vraie vie (entreprise)

- **Objectif**

- Faire le lien entre **Git théorique** et **le quotidien d'un développeur** :
- montrer que Git n'est jamais utilisé seul
- préparer mentalement les juniors à GitLab / GitHub / CI/CD

# Git en environnement professionnel

- Git est rarement utilisé seul. Il est intégré dans un **écosystème d'outils** :
- Plateformes Git :
  - GitLab
  - GitHub
  - Bitbucket
- Travail collaboratif :
  - Branches
  - Pull Request / Merge Request
  - Revue de code
- Intégration avec les outils DevOps :
  - Déclenchement automatique de pipelines CI/CD
  - Tests automatiques
  - Déploiement continu

**Chaque action Git peut avoir un impact :**

- un push peut lancer un pipeline
- un merge peut déclencher un déploiement
- un tag peut créer une release



# Bases de la gestion de versions

## **Objectif du module**

Avant de parler de Git, s'assurer que les juniors comprennent :

le **problème initial**

ce qu'apporte un **système de gestion de versions**

pourquoi Git est un DVCS

# Qu'est-ce qu'un système de gestion de versions (VCS) ?

- **Sans système de gestion de versions**
- Sauvegardes manuelles  
(project\_final\_v3\_definitif.zip)
- Perte de l'historique
- Impossible de savoir :
  - qui a modifié un fichier
  - quand
  - pourquoi
- **Avec un système de gestion de versions (VCS)**
- Historique complet des modifications
- Traçabilité :
  - auteur
  - date
  - message
- Possibilité de revenir à une version antérieure
- Un VCS **enregistre l'évolution d'un projet dans le temps**

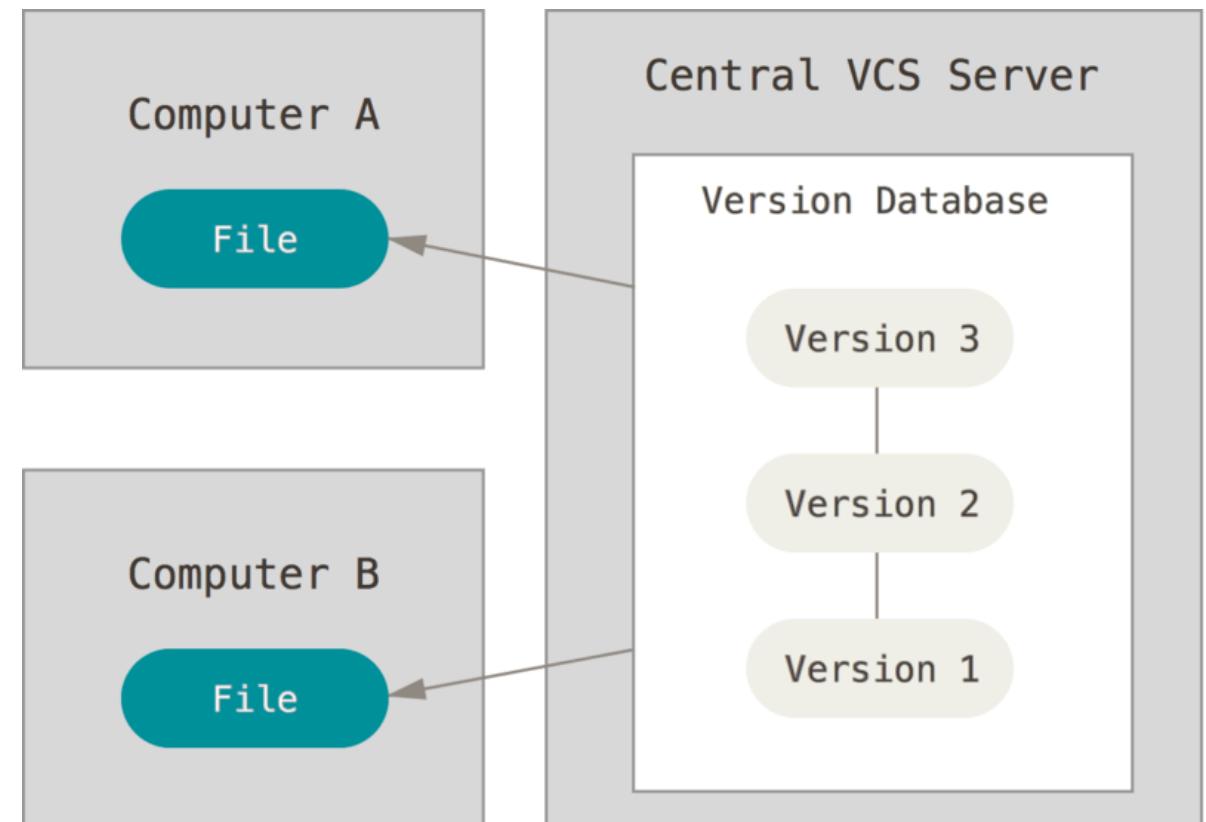
# Systèmes de gestion de versions : centralisé vs distribué

## Systèmes de gestion de versions centralisés (CVCS)

- Un **serveur central** contient le code source
- Les développeurs récupèrent les fichiers depuis ce serveur
- Exemples : SVN, CVS

### Limites :

- Dépendance forte au serveur central
- Si le serveur est indisponible → travail bloqué
- Historique incomplet côté client



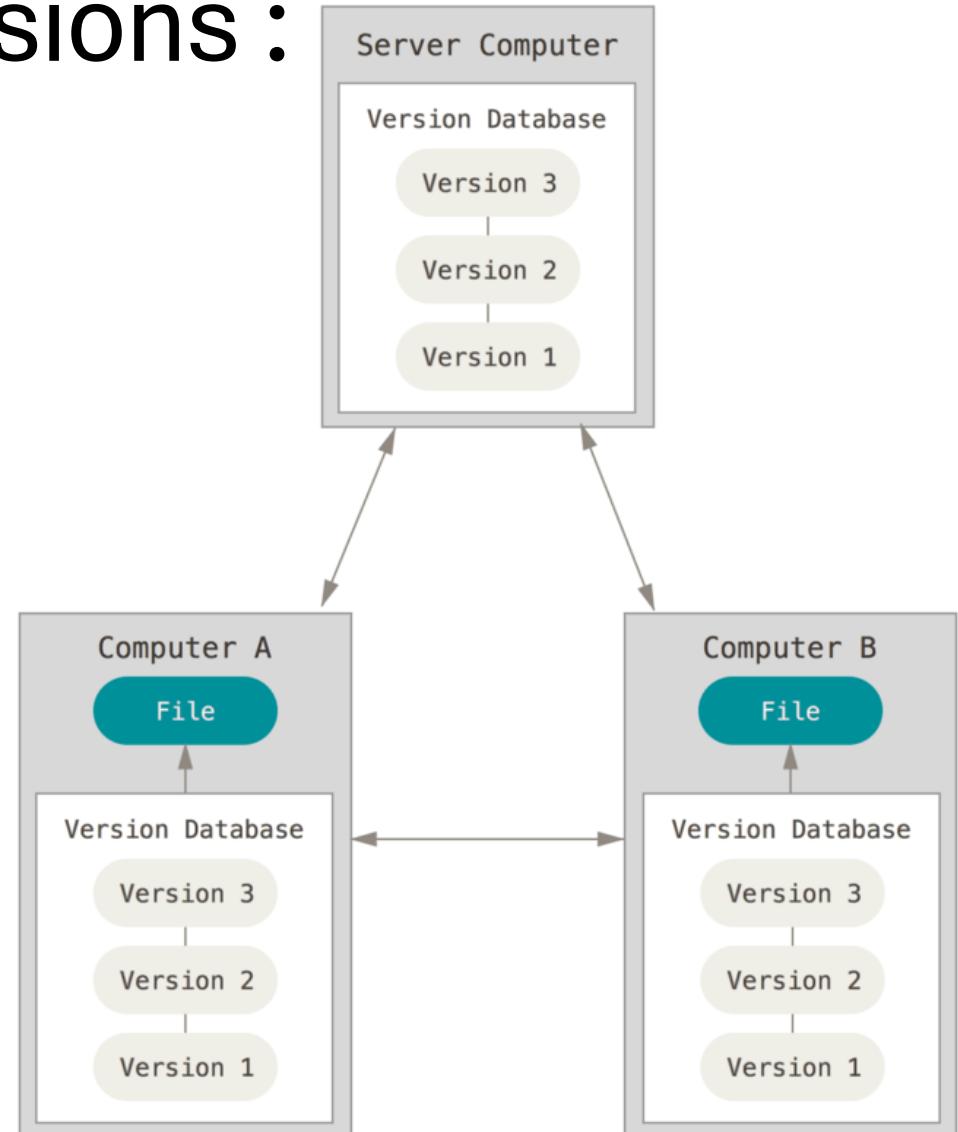
# Systèmes de gestion de versions : centralisé vs distribué

## Les systèmes de gestion de version distribués (DVCS)

- Chaque développeur possède une **copie complète du dépôt**
- Historique complet disponible localement
- Exemples : Git, Mercurial

### Avantages :

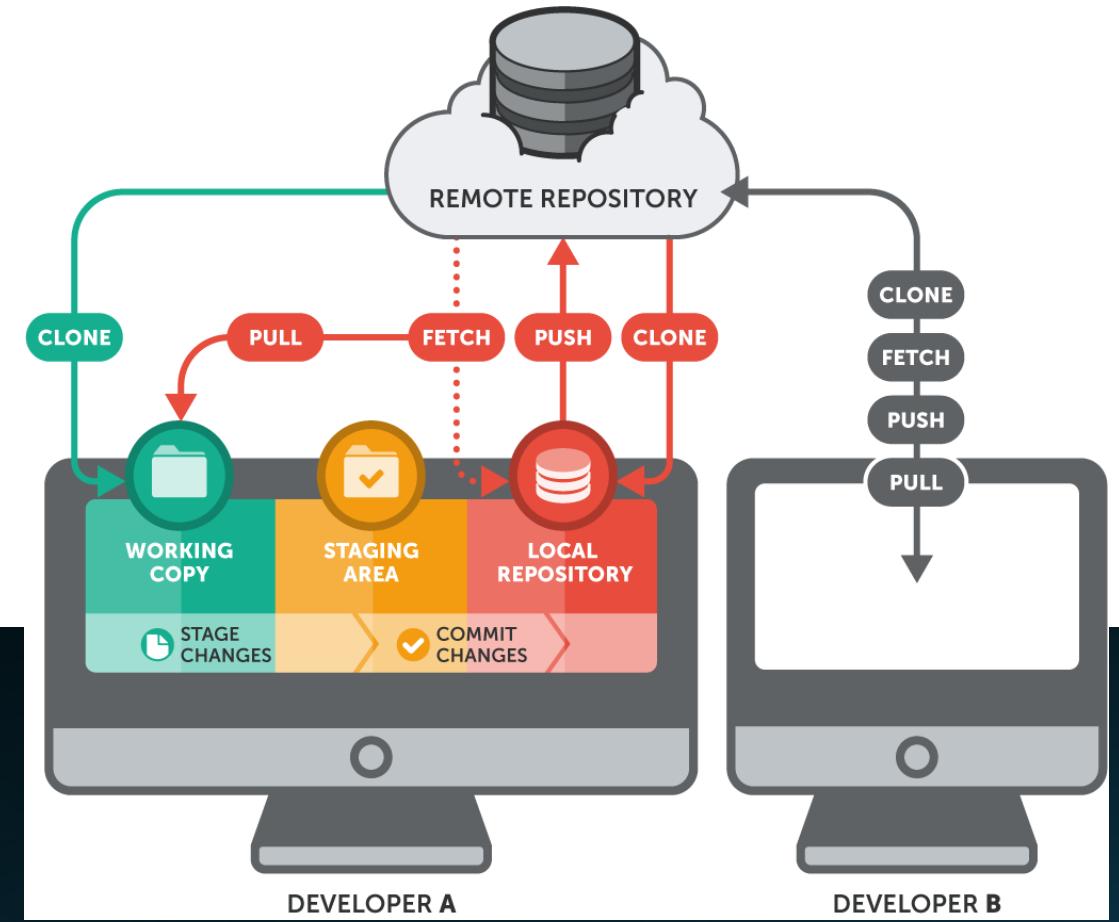
- Travail possible hors connexion
- Meilleure performance
- Chaque clone est une sauvegarde complète



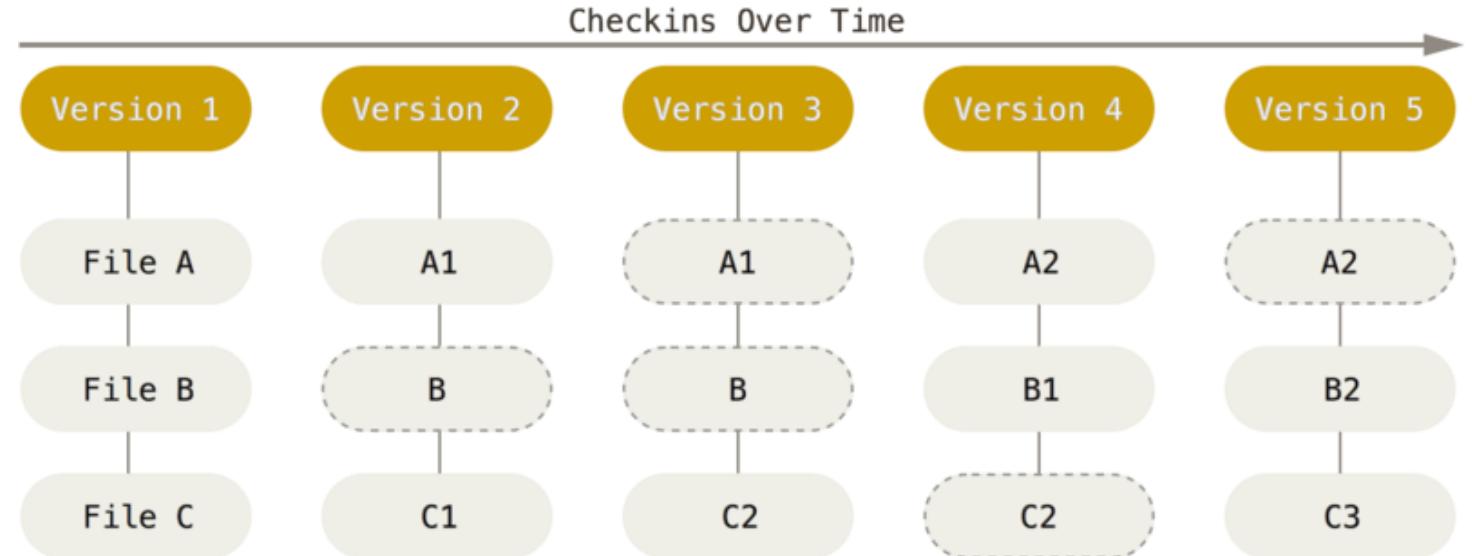
# Concepts fondamentaux de Git

## Objectif du module

À la fin, un junior doit **comprendre ce que fait Git**, pas juste taper des commandes.



# Les snapshots de Git



- Git ne stocke pas les différences ligne par ligne
- À chaque commit, Git enregistre un **instantané (snapshot)** du projet
- Chaque snapshot représente l'état complet du projet à un instant donné

Si un fichier n'a pas changé :

- Git ne le recopie pas
- Git référence simplement la version précédente

# Les trois zones de travail Git

- Git organise le travail en trois zones :
- Working Directory
  - Répertoire de travail
  - Fichiers modifiés localement
- Staging Area (Index)
  - Zone intermédiaire
  - Contient les fichiers prêts à être commités
- Repository (Git directory)
  - Historique des commits
  - Stocké dans .git

## Working Copy

Your Project's Files



Git watches tracked files  
for new local modifications...

## Tracked (and modified)



If a file was modified since it  
was last committed, you can  
stage & commit these changes

stage



Changes that are **not staged** will  
not be committed & remain as  
local changes until you stage &

## Staging Area

Changes included in  
the Next Commit



Changes that were added to  
the Staging Area will be  
included in the next commit

commit

## Local Repository

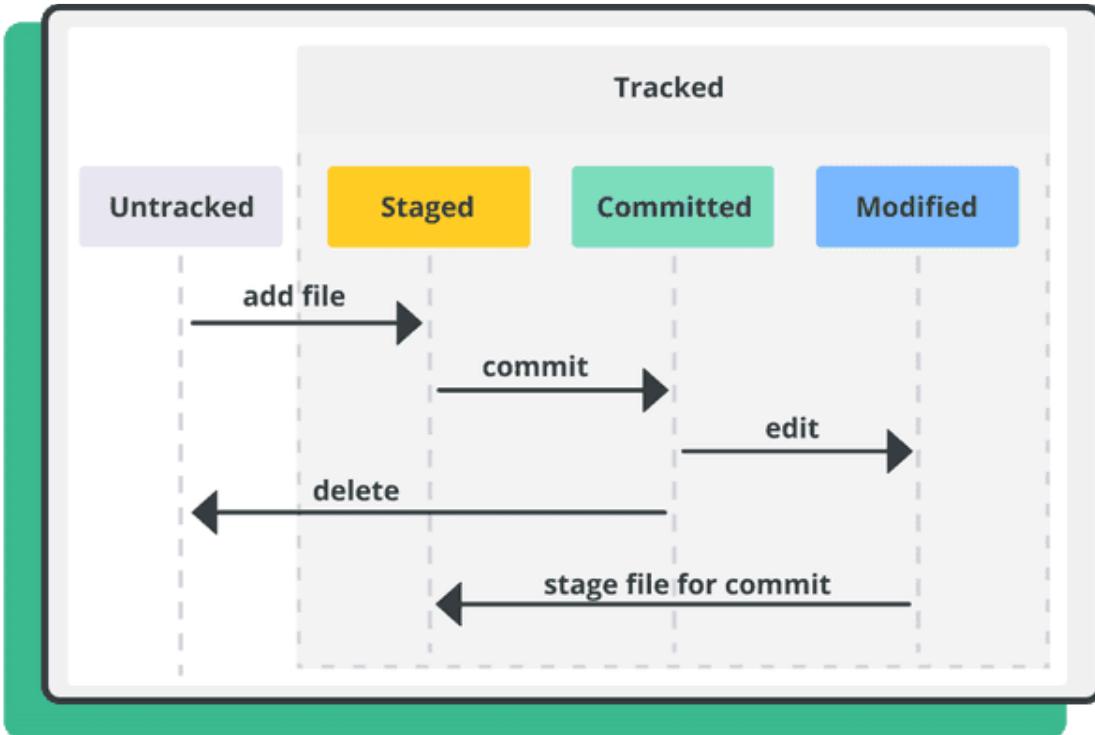
The ".git" Folder



All changes contained in a  
commit are saved in the local  
repository as a new revision

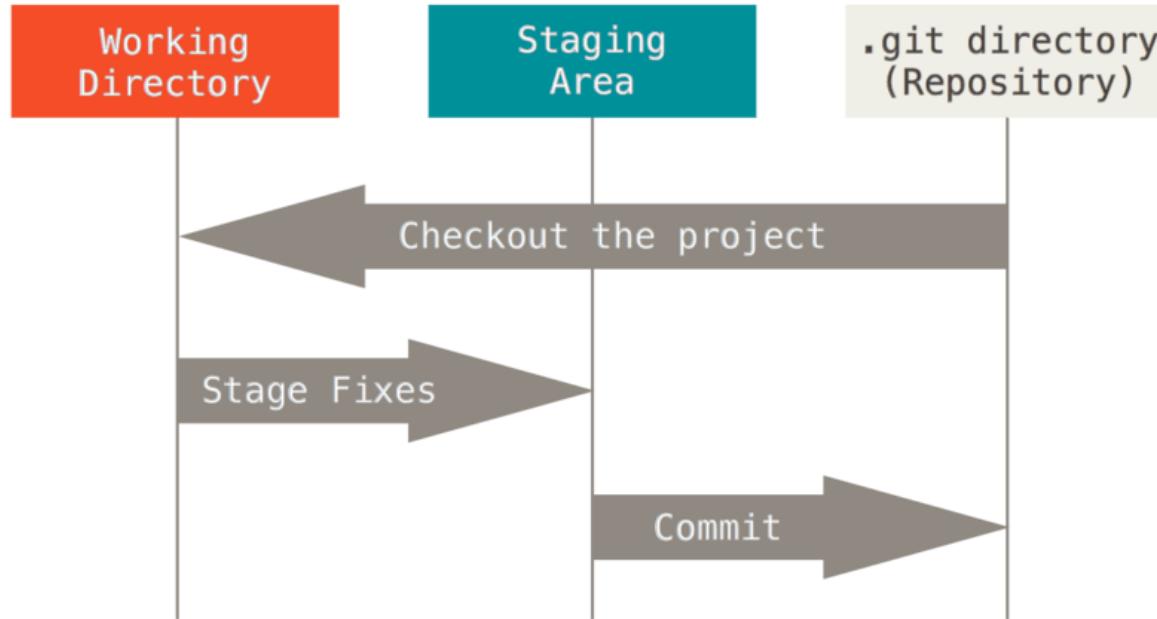
Git ne fait jamais un commit **directement** depuis le working directory.

# Les trois états d'un fichier dans Git



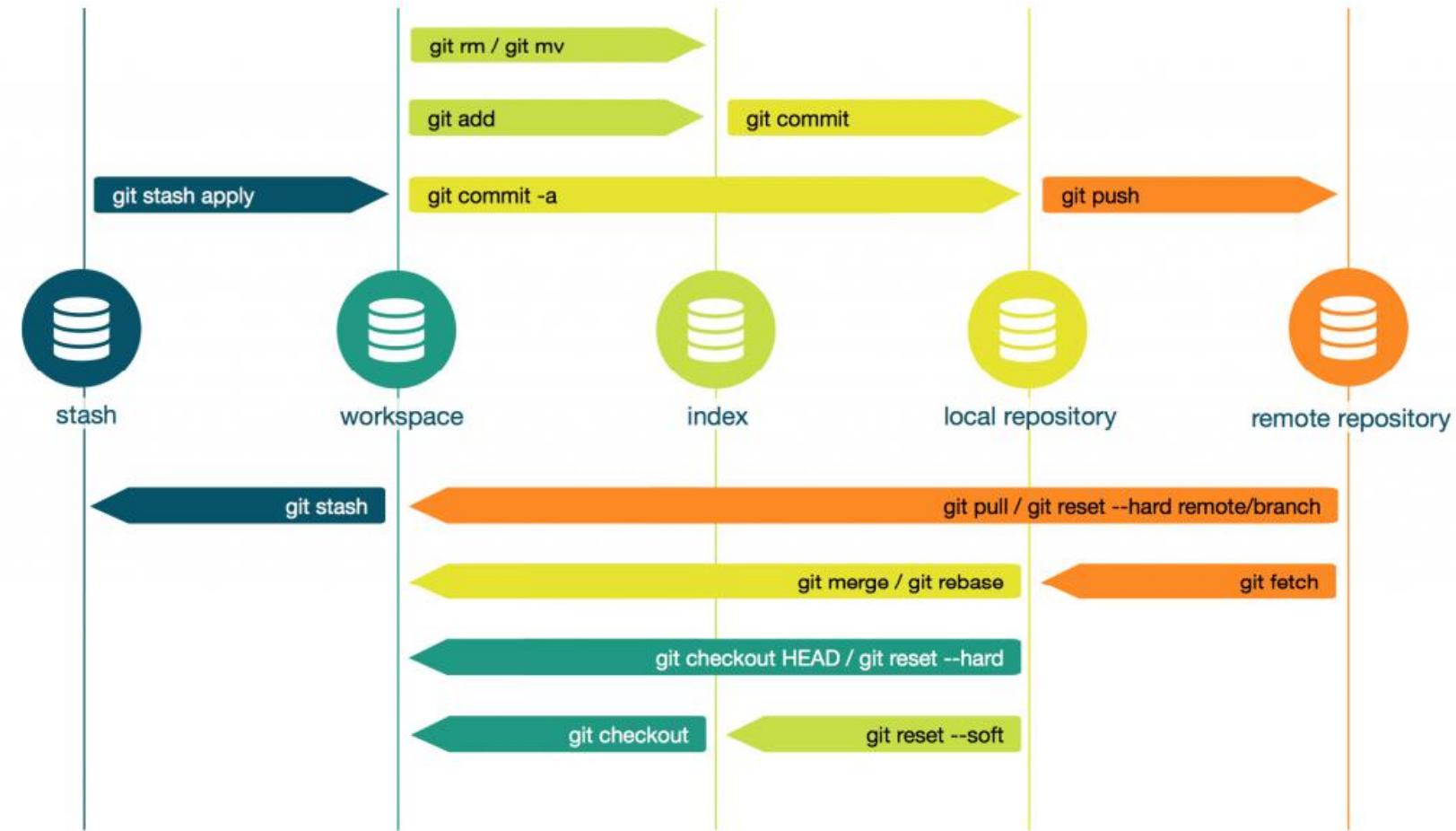
- Un fichier Git peut être dans trois états :
- **Untracked**
  - Fichier inconnu de Git
  - N'est ni suivi, ni versionné
  - Exemple : nouveau fichier créé
- **Modified**
  - Le fichier a été modifié
  - Git ne l'a pas encore pris en compte
- **Staged**
  - Le fichier est dans la staging area
  - Il sera inclus dans le prochain commit
- **Committed**
  - Le fichier est enregistré dans l'historique Git
  - Il fait partie d'un snapshot
- Le passage d'un état à un autre est **contrôlé par le développeur**

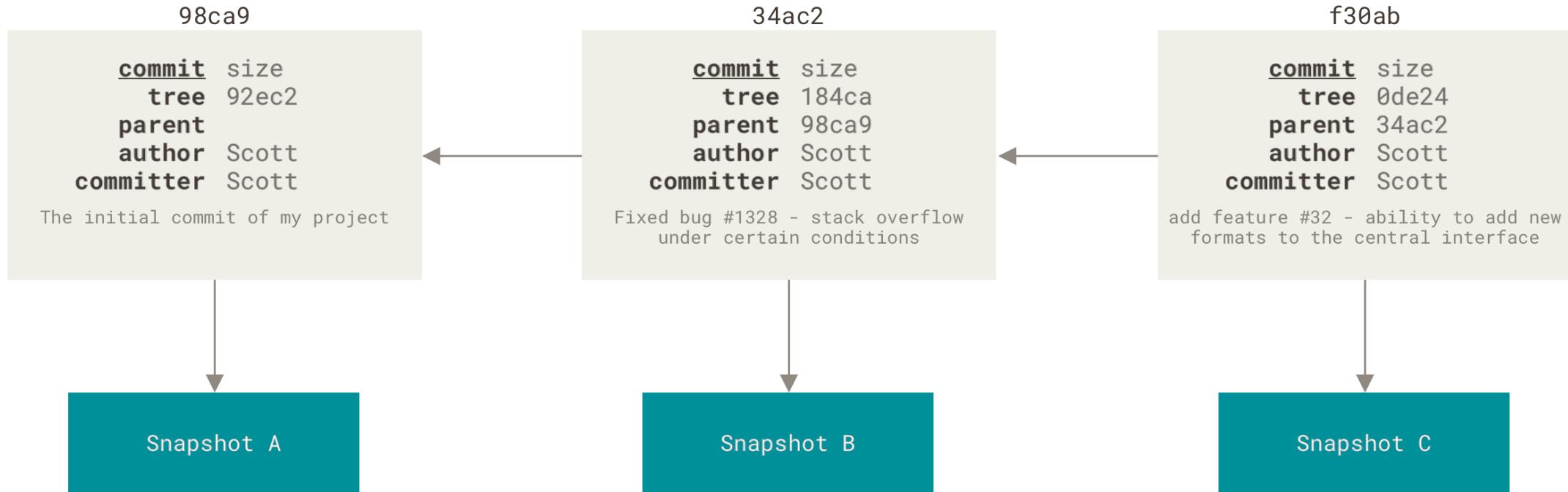
# Cycle de vie d'un fichier Git



1. Création du fichier  
→ **Untracked**
2. Ajout au suivi  
→ `git add`  
→ **Staged**
3. Validation  
→ `git commit`  
→ **Committed**
4. Modification du fichier  
→ **Modified**
5. Nouvelle indexation  
→ `git add`  
→ **Staged**

# Git cheatsheet



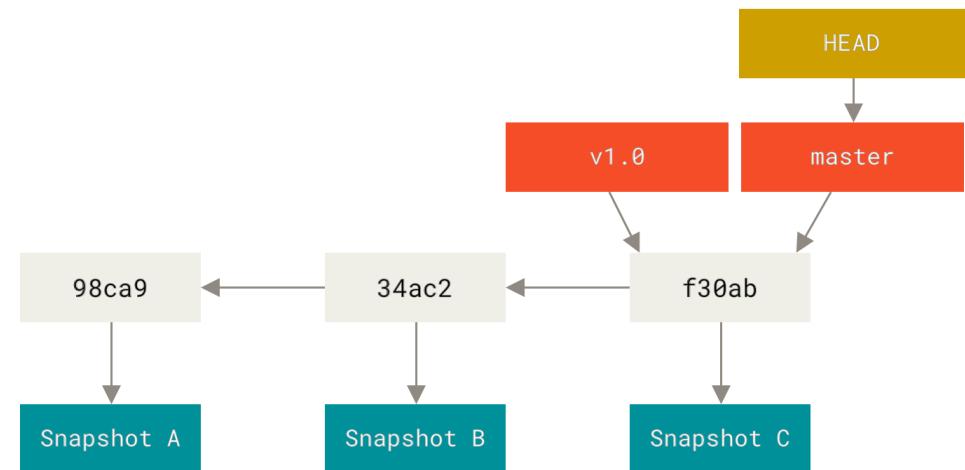


# Ce qu'est réellement un commit

- **Un commit Git contient :**
  - Un **snapshot** du projet
  - Un **auteur** (nom + email)
  - Une **date**
  - Un **message de commit**
  - Un ou plusieurs **parents**
  - Un identifiant unique (**SHA-1**)
- **Types de commits :**
  - Commit initial : aucun parent
  - Commit normal : un parent
  - Commit de merge : plusieurs parents
- Un commit est **un objet immuable**

# Branches et HEAD

- **Les branches dans Git**
  - Une branche est un **pointeur** vers un commit
  - Elle avance à chaque nouveau commit
  - Créer une branche est une opération légère
- **HEAD**
  - HEAD est un pointeur spécial
  - Il indique **la branche ou le commit courant**
  - Toutes les commandes agissent là où pointe HEAD
- Changer de branche = déplacer HEAD



# Commandes essentielles (usage quotidien)

## **Objectif du module**

Rendre les juniors **opérationnels** :

comprendre **ce qu'ils tapent**

savoir **quand** utiliser chaque commande

éviter les erreurs courantes

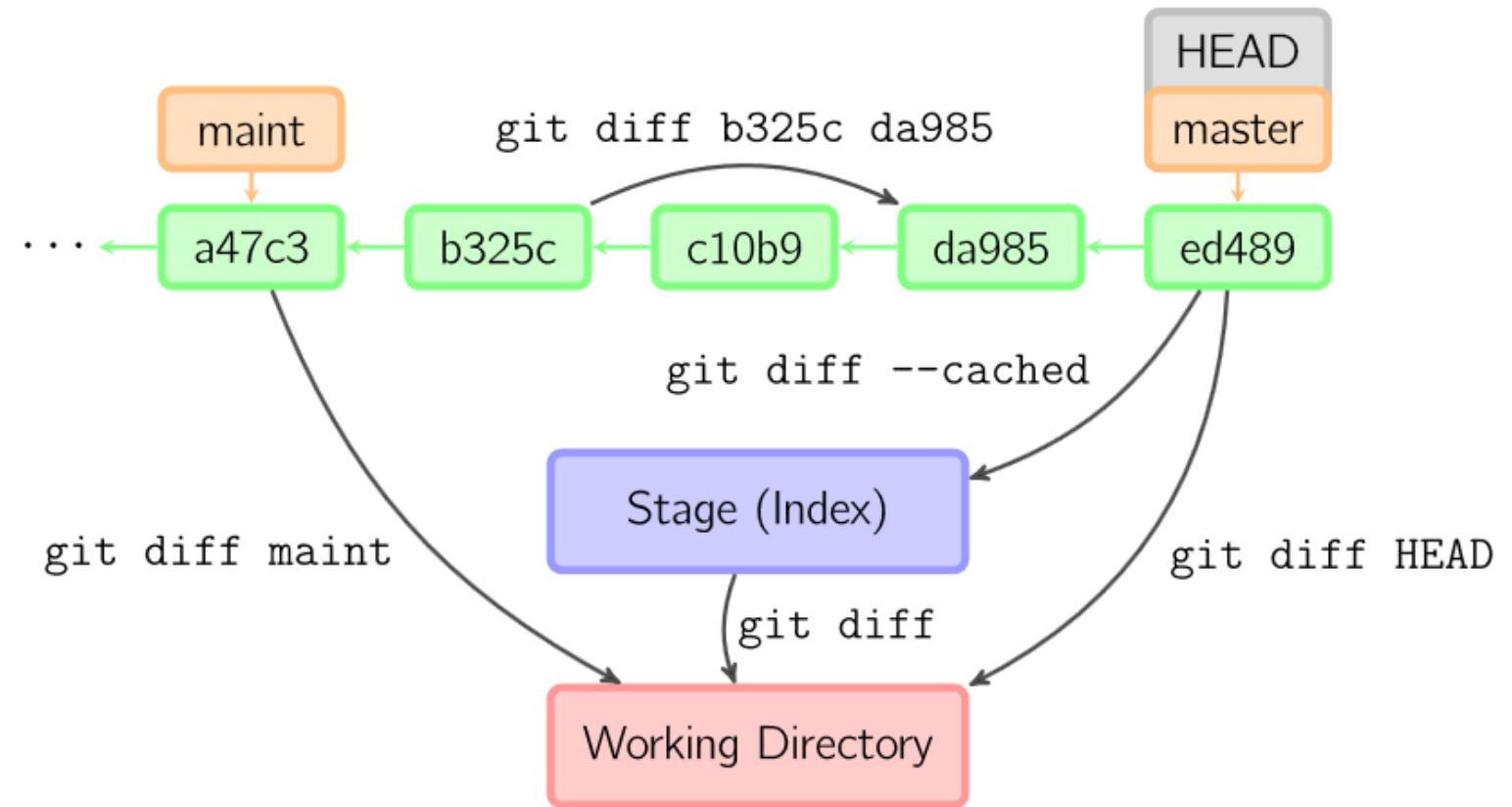
# Initialiser et cloner un dépôt

- **Créer ou récupérer un dépôt Git**
  - Initialiser un nouveau dépôt :  
`git init`
  - Cloner un dépôt existant :  
`git clone <url>`
- **Différence :**
  - `git init` → nouveau projet
  - `git clone` → projet existant + historique complet
- Un clone contient **tout l'historique**

# Observer l'état du dépôt

- **Commandes pour comprendre ce qui se passe**
  - Vérifier l'état du dépôt :  
`git status`
  - Voir l'historique des commits :  
`git log`
  - Voir les différences :  
`git diff`
  - Ces commandes **ne modifient rien**, elles informent.

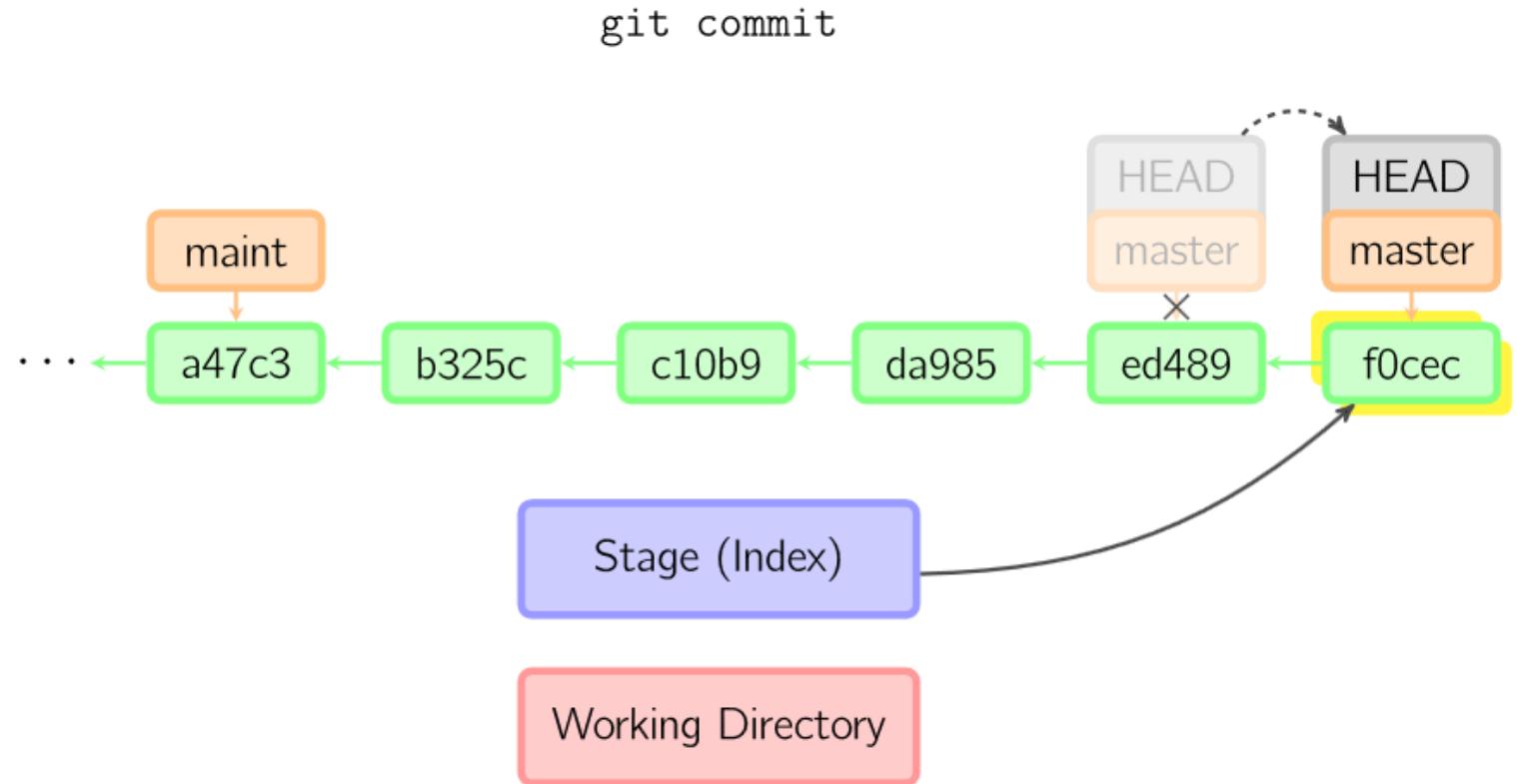
diff



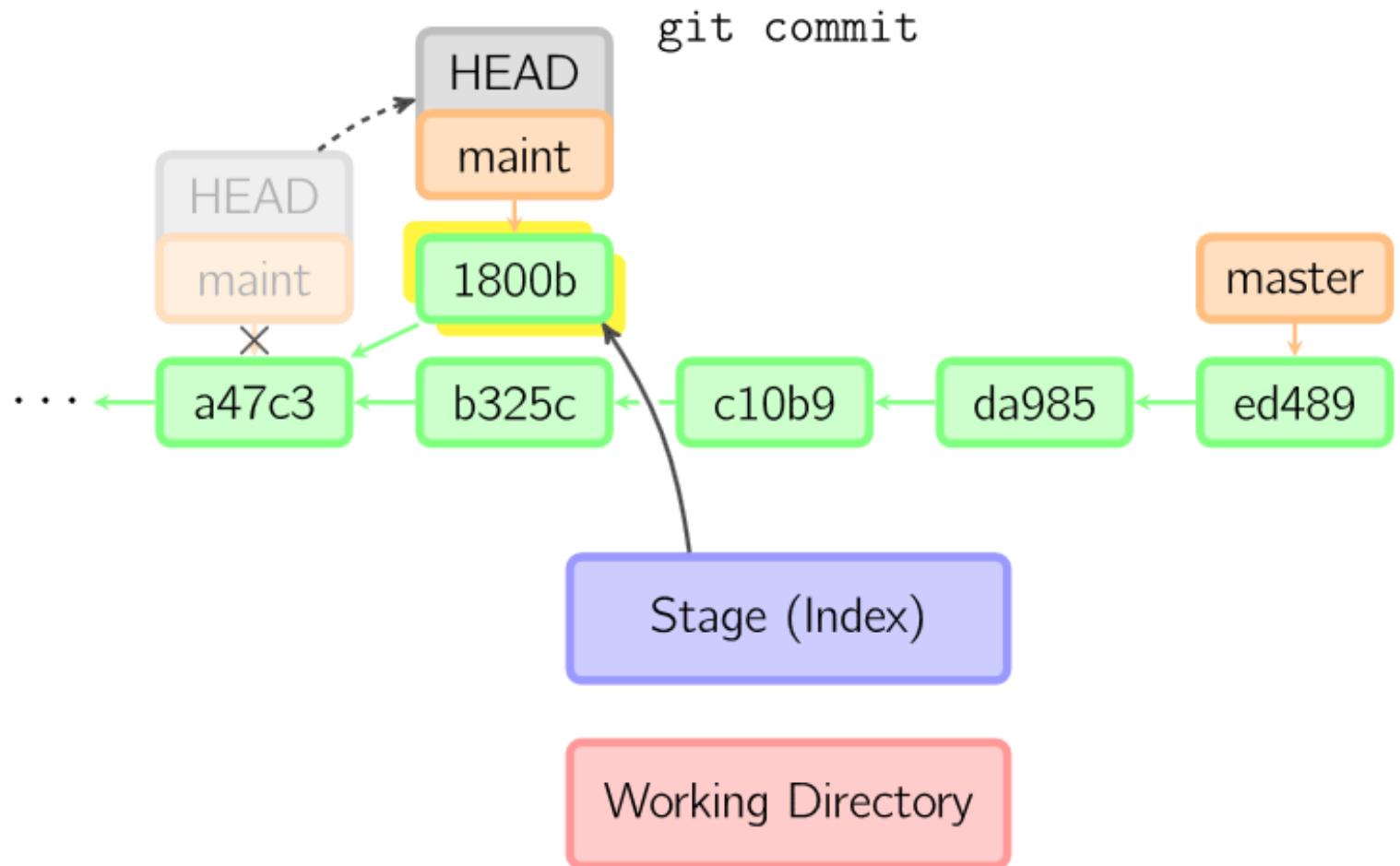
# Enregistrer son travail (add & commit)

- **Étapes pour enregistrer des modifications**
- Ajouter des fichiers à la staging area :  
`git add <fichier>`  
`git add .`
- Valider les changements :  
`git commit -m "message de commit"`
- Le commit enregistre **ce qui est dans le stage**, pas tout le projet.

# commit



# commit



# Modifier le dernier commit (git commit --amend)

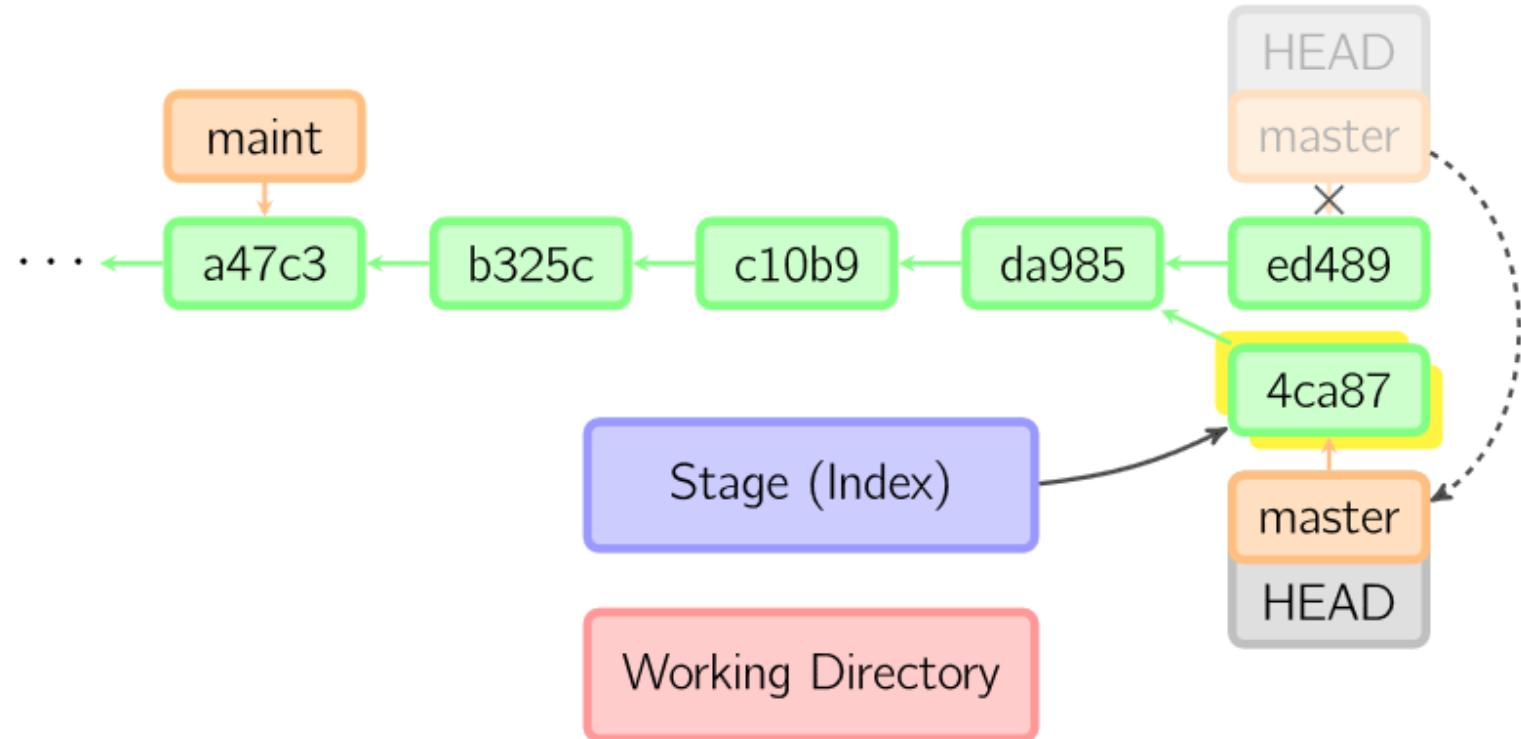
## Corriger le dernier commit

git commit --amend

- Permet de :
  - Modifier le message du dernier commit
  - Ajouter des fichiers oubliés
- Le commit précédent est **remplacé**, pas modifié.
- **⚠ Règle importante**
  - OK si le commit **n'a pas été poussé**
  - À éviter sur un commit déjà partagé

# commit --amend

git commit --amend

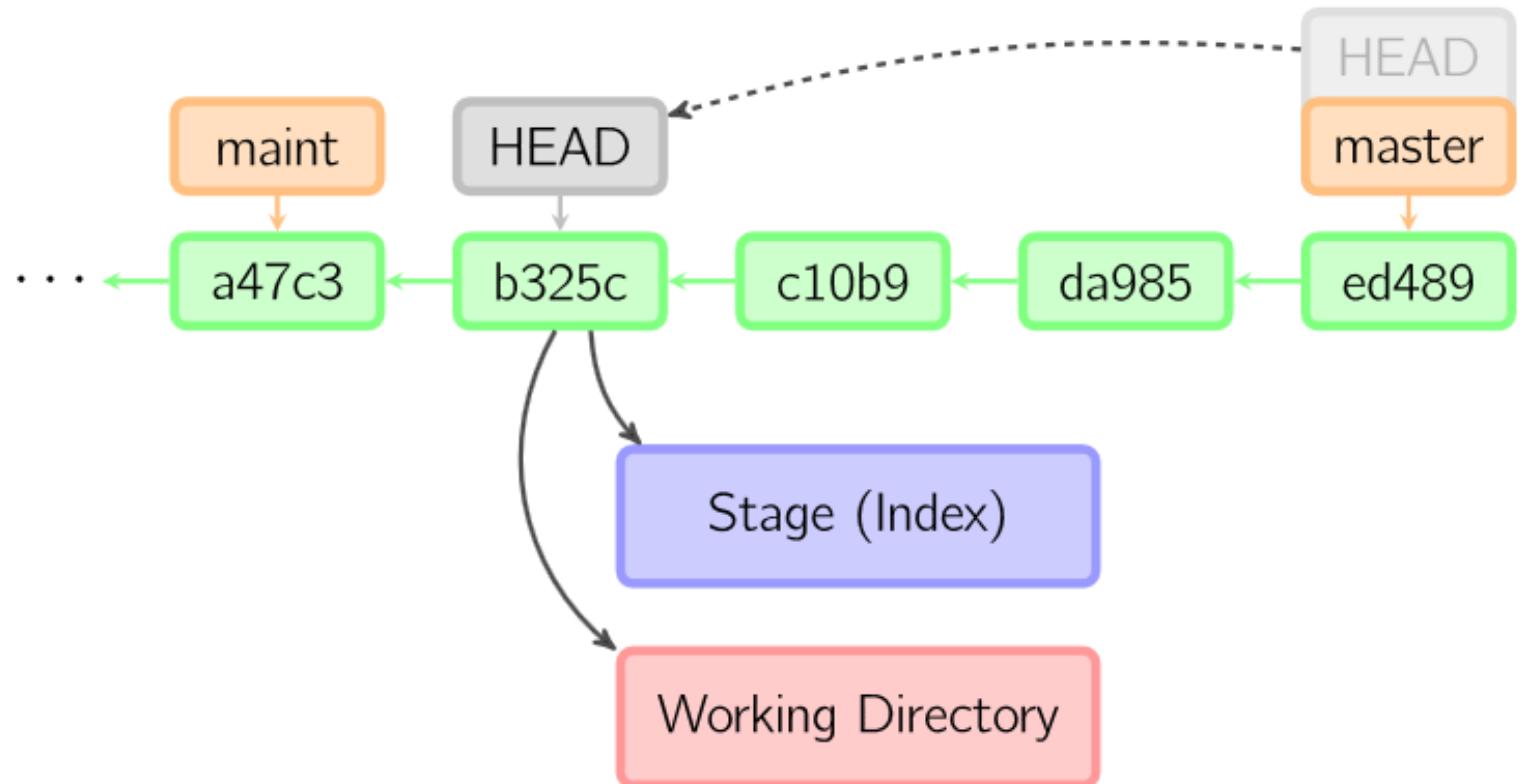


# Travailler avec des branches

- Créer une branche :  
`git branch feature-login`
- Changer de branche :  
`git checkout feature-login`  
ou  
`git switch feature-login`
- Créer et basculer en une seule commande :  
`git checkout -b feature-login`
- Chaque fonctionnalité doit avoir sa branche.

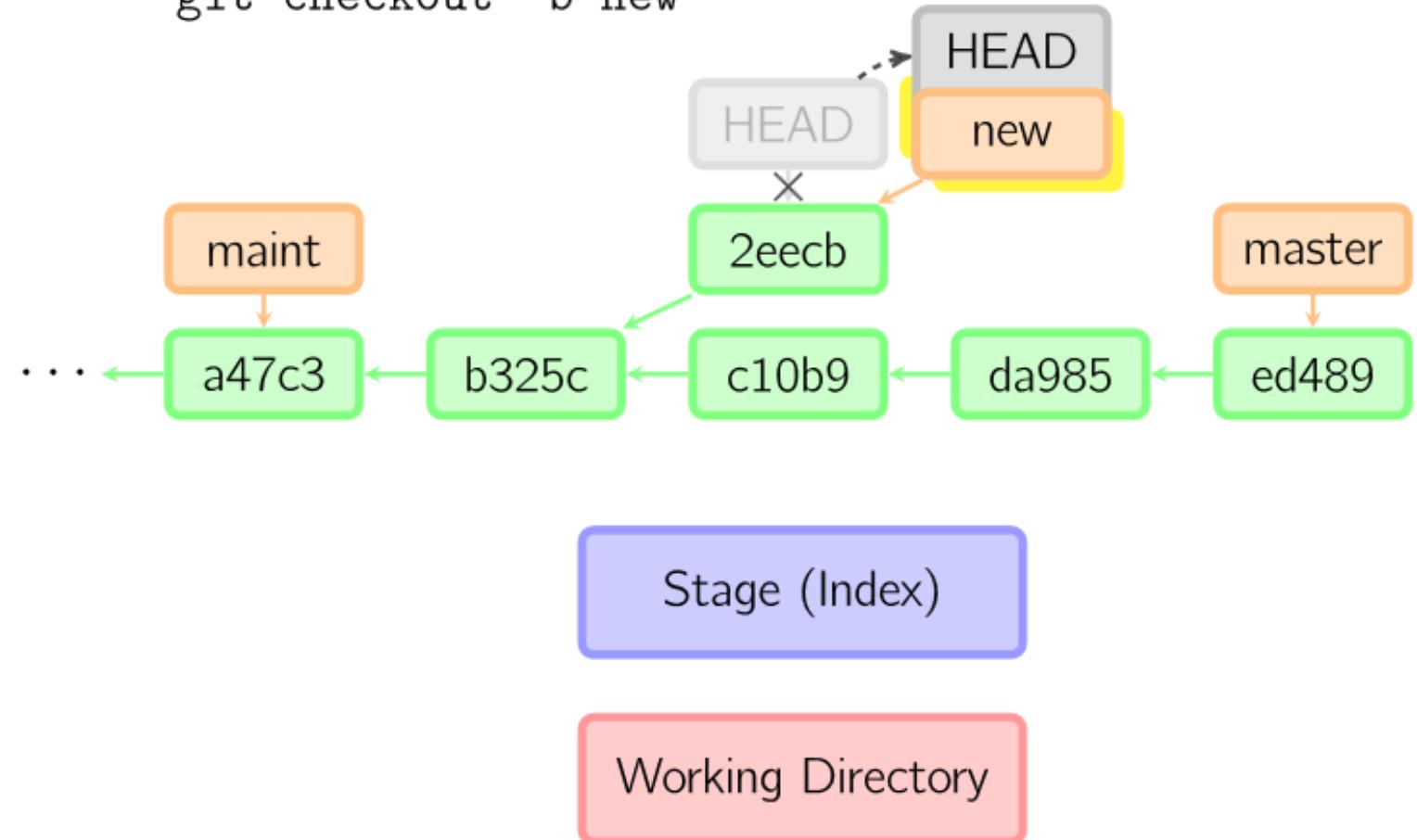
# checkout (branche anonyme ou detached HEAD)

git checkout master~3

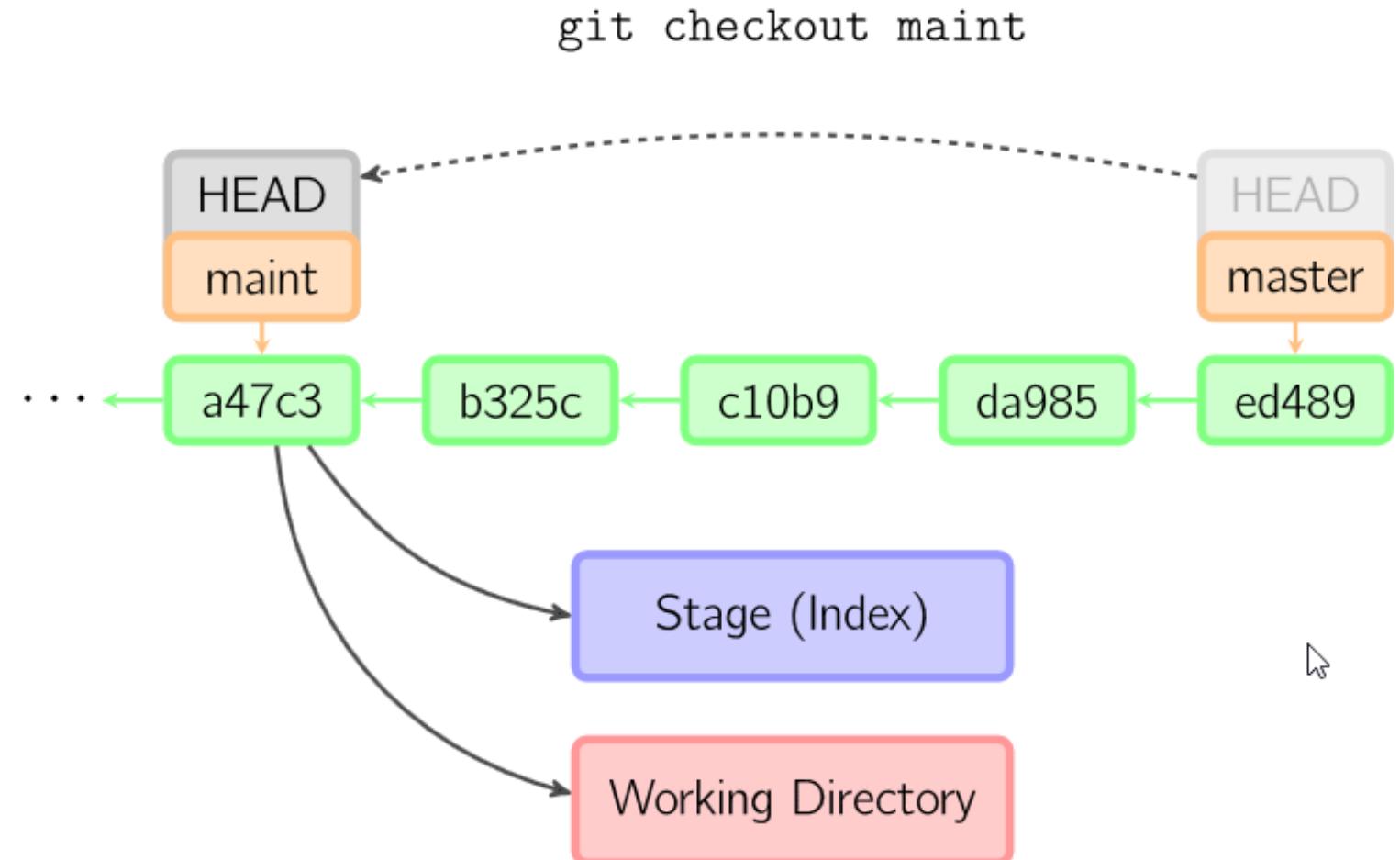


# Créer et basculer sur une branche

```
git checkout -b new
```

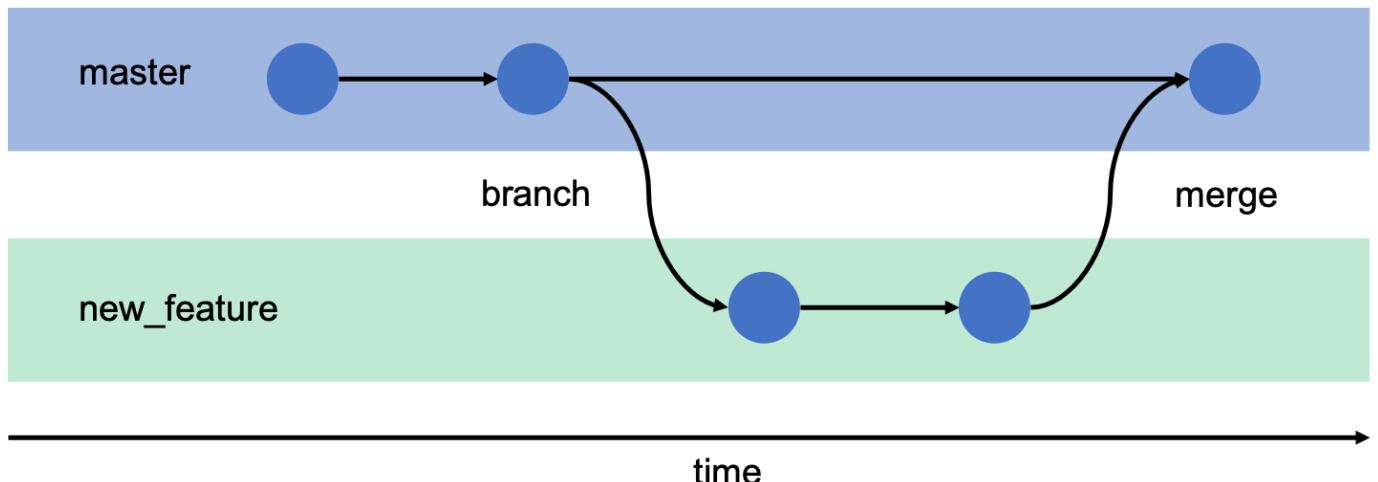


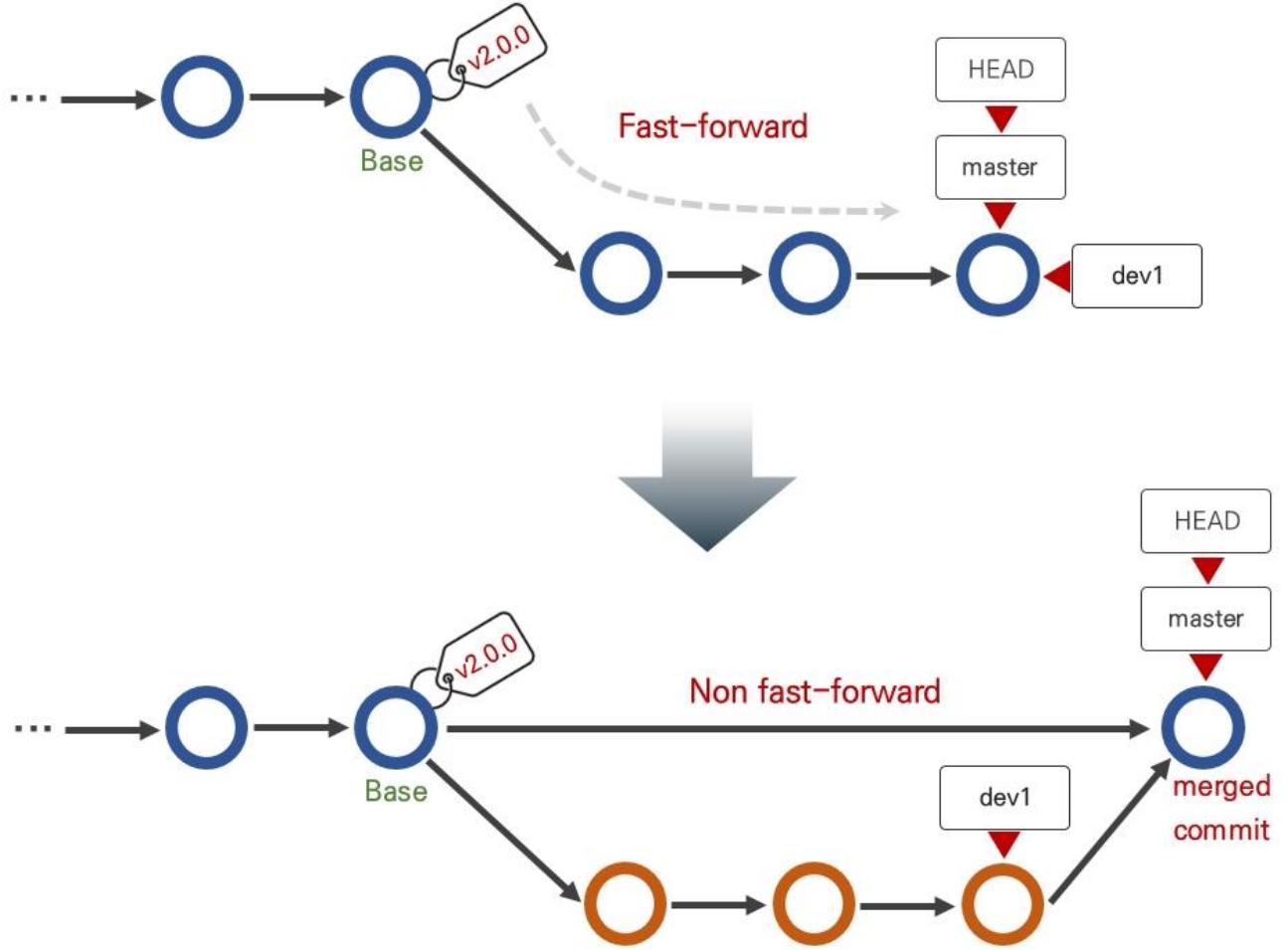
# Changer de branche



# Fusionner du code (git merge)

- **Fusionner une branche dans une autre**
  - git merge feature-login
  - Combine le contenu de deux branches
  - Peut créer un **commit de merge**
  - Git tente de fusionner automatiquement
  - Si deux personnes ont modifié la même partie du fichier :
    - Git signale un **conflit**
    - Le développeur doit le résoudre manuellement

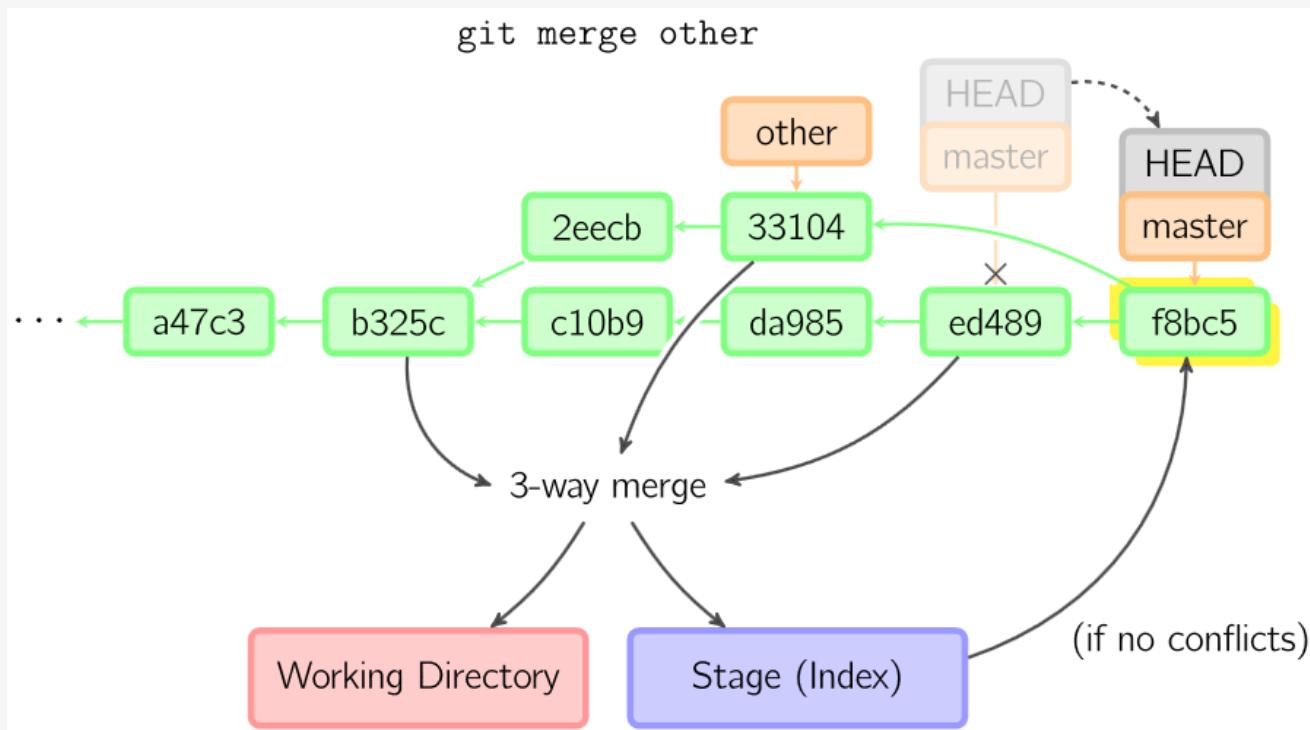




## Fast-forward merge (ou pas)

- Type de fusion Git qui se produit lorsqu'il existe un chemin linéaire entre la pointe de la branche courante et la branche cible.
- Au lieu de créer un nouveau commit de fusion, Git déplace simplement (ou « avance rapidement ») le pointeur de la branche courante pour qu'il pointe vers le même commit que la branche cible.
- `--no-ff` : Crée systématiquement un commit de fusion, même si une avance rapide est possible. Cela préserve l'historique de la branche.

## 3 way merge



- Branches divergentes
- Création d'un commit de merge
- Peut générer des conflits
- Le développeur utilise **toujours git merge**

👉 Git choisit automatiquement le type de merge

# Travailler en équipe avec Git

## **Objectif**

Comprendre :

où vit le code

ce que représente origin

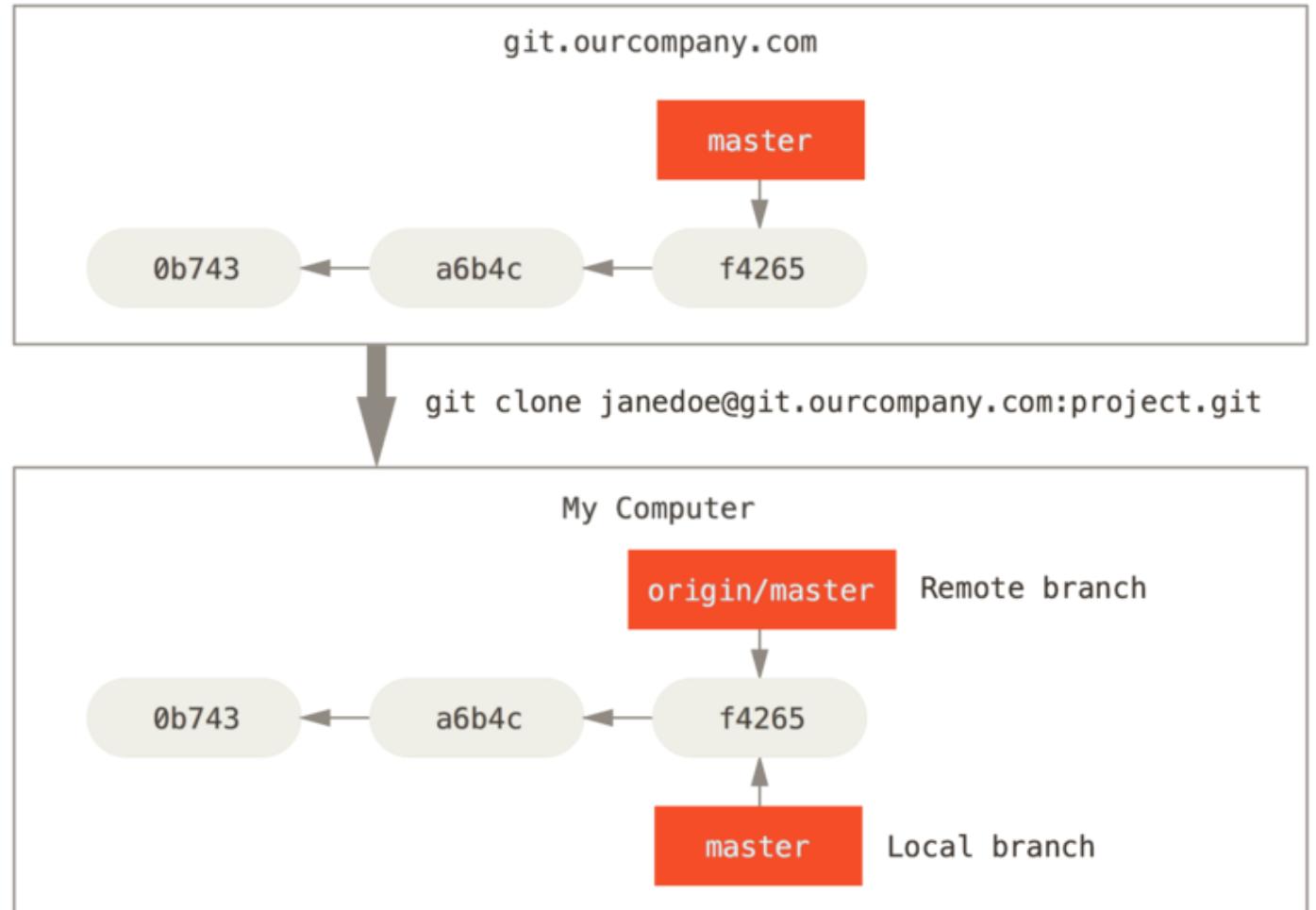
pourquoi Git est distribué

# Dépôts locaux et dépôts distants

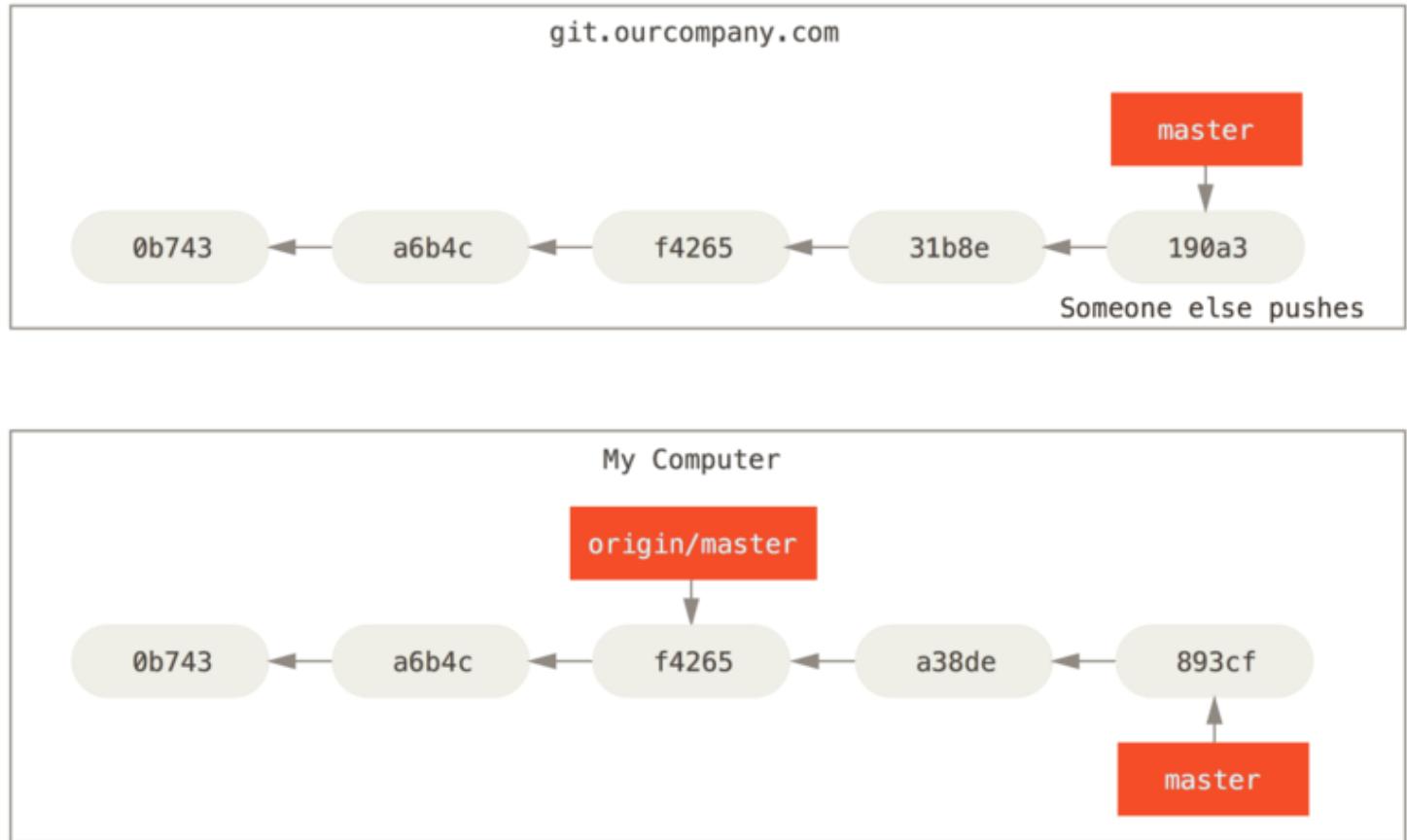
- **Dépôt local**
  - Sur la machine du développeur
  - Historique complet
  - Travail possible hors connexion
- **Dépôt distant**
  - Dépôt partagé (GitLab, GitHub, Bitbucket)
  - Synchronisation de l'équipe
  - Appelé par défaut origin
- **Le dépôt distant n'est pas obligatoire pour Git, mais indispensable pour travailler en équipe.**

# git clone <url>

- Récupérer le projet (une seule fois)
- Crée un dépôt local
- Configure automatiquement le dépôt distant origin
- Récupère tout l'historique

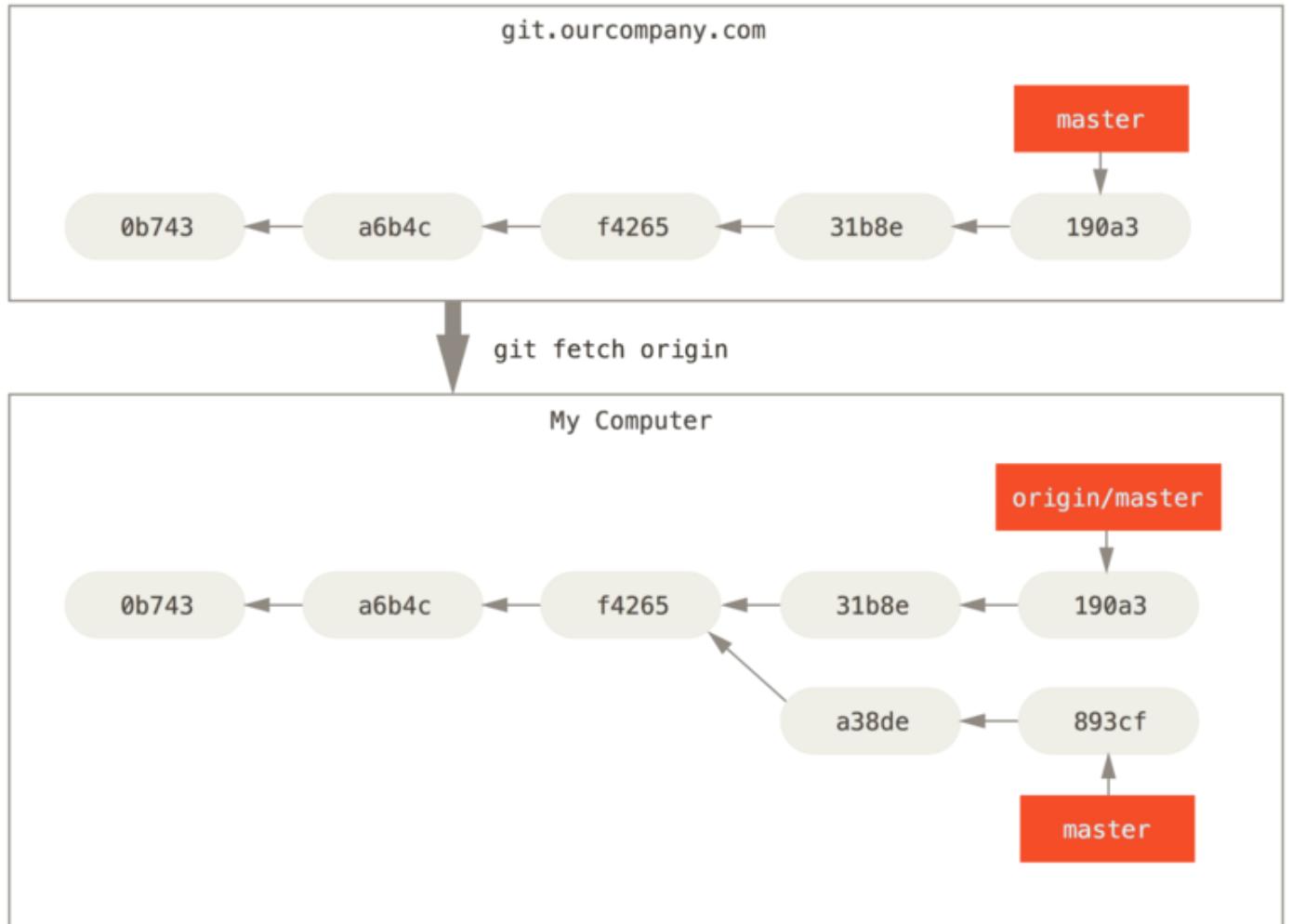


# Les travaux locaux et distants peuvent diverger



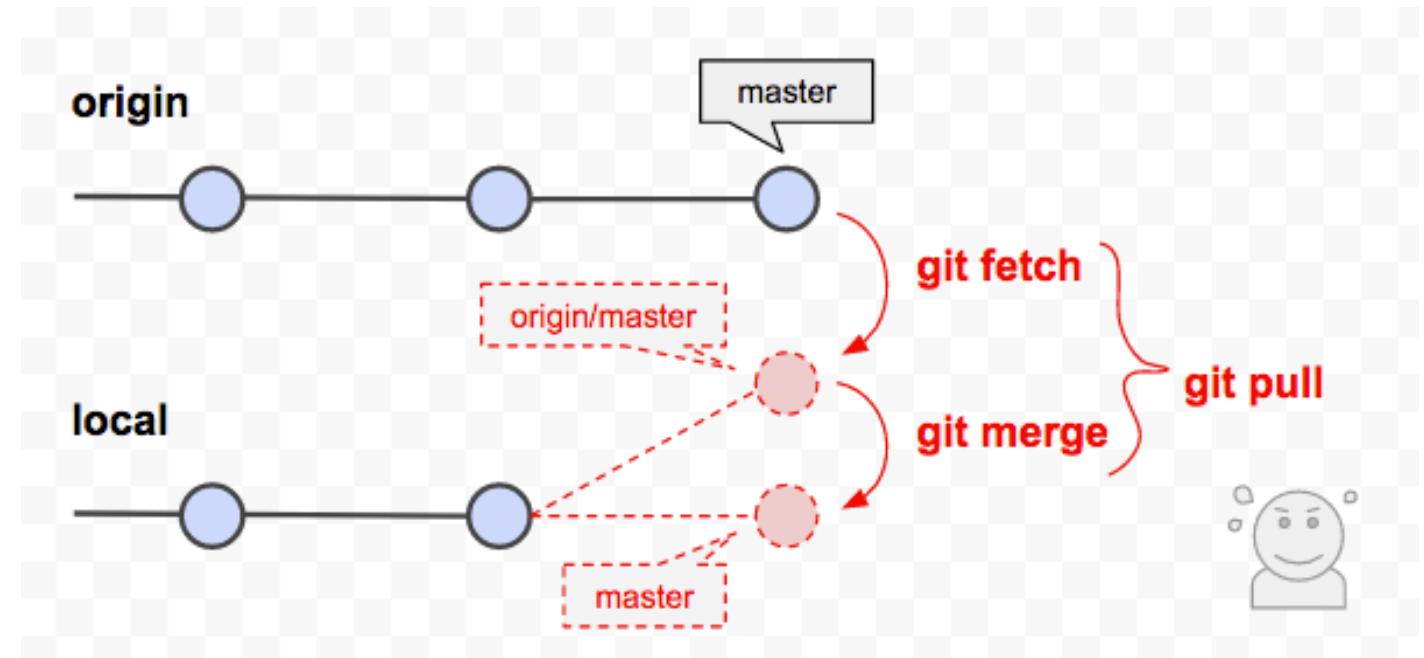
# git fetch

- Récupère les changements du dépôt distant
- Met à jour les branches de suivi à distance
- ✗ Ne modifie pas la branche locale
- ➡ Commande **sans risque**



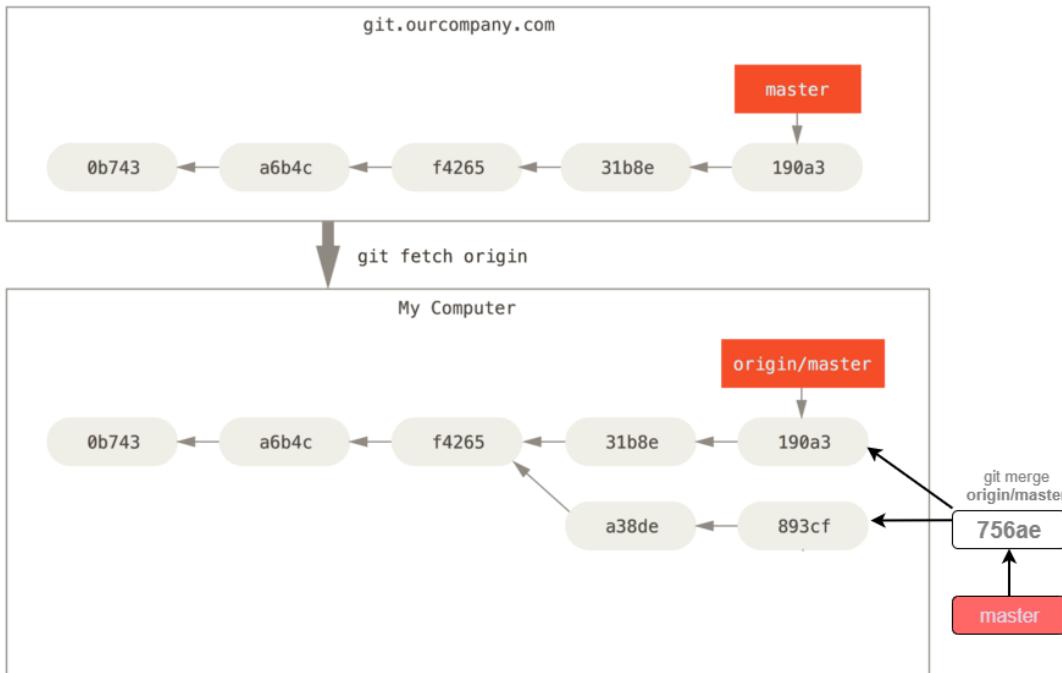
# git pull

- Équivaut à :
  - git fetch + git merge
- Met à jour la branche locale
- Peut créer un commit de merge
- ➡ À utiliser **en connaissance de cause**

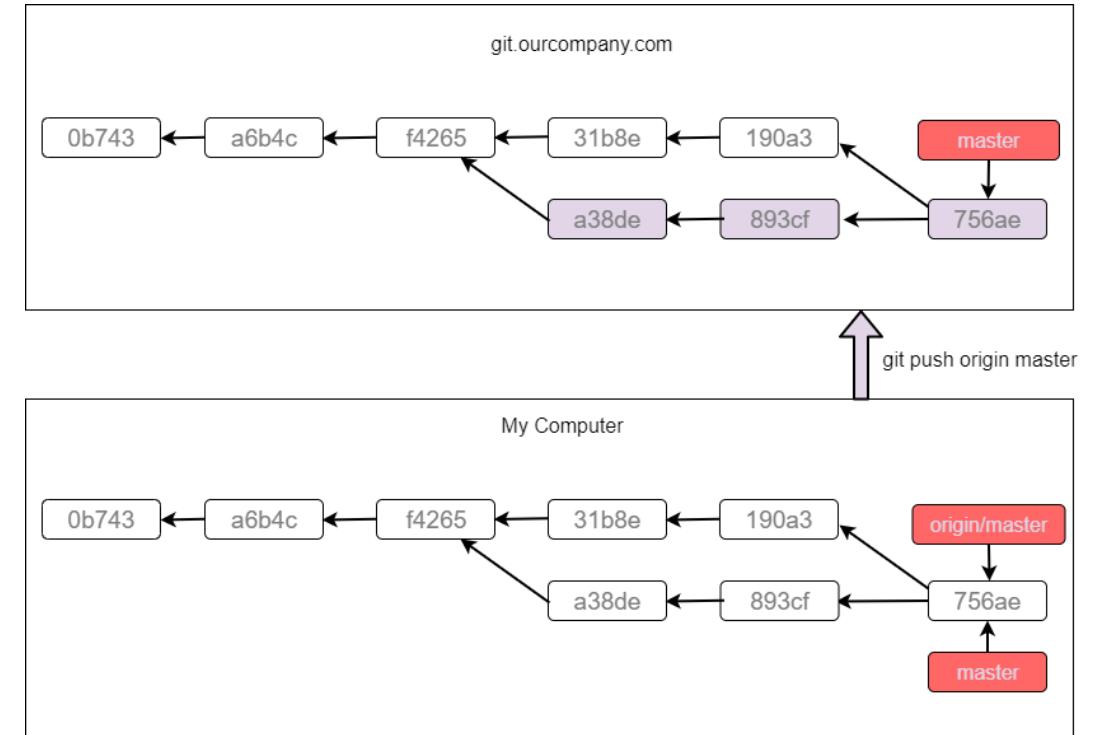


# git push

## Avant push



## Après push



# Cycle de travail standard d'un développeur

## Cycle de travail typique

- 1 Cloner le projet (une seule fois)

```
git clone <url>
```

- 2 Créer une branche de travail

```
git checkout -b feature-x
```

- 3 Développer, ajouter et commiter

```
git add .
```

```
git commit -m "feat: add feature x"
```

- 4 Mettre à jour sa branche

```
git pull
```

- 5 Pousser son travail

```
git push origin feature-x
```

- 6 Créer une Merge Request / Pull Request

- **On ne pousse jamais directement sur la branche principale**

# Les conflits de merge

## Pourquoi un conflit apparaît ?

- Deux branches modifient la **même partie d'un fichier**
  - Git ne peut pas décider quelle version garder
  - Ce n'est **pas une erreur**, c'est une situation normale en équipe.
- **Comment Git signale un conflit ?**
- Le merge s'arrête
  - Git marque les zones en conflit dans le fichier
- ```
<<<<< HEAD  
code local  
=====  
code distant  
>>>>> feature-x
```

# Résoudre un conflit

1. Lire et comprendre les deux versions
2. Choisir ou combiner le code correct
3. Supprimer les marqueurs
4. git add
5. git commit

# Pull Request / Merge Request

## Qu'est-ce qu'une Pull / Merge Request ?

- Demande formelle pour intégrer une branche
- Permet :
  - revue de code
  - discussions
  - validation avant intégration

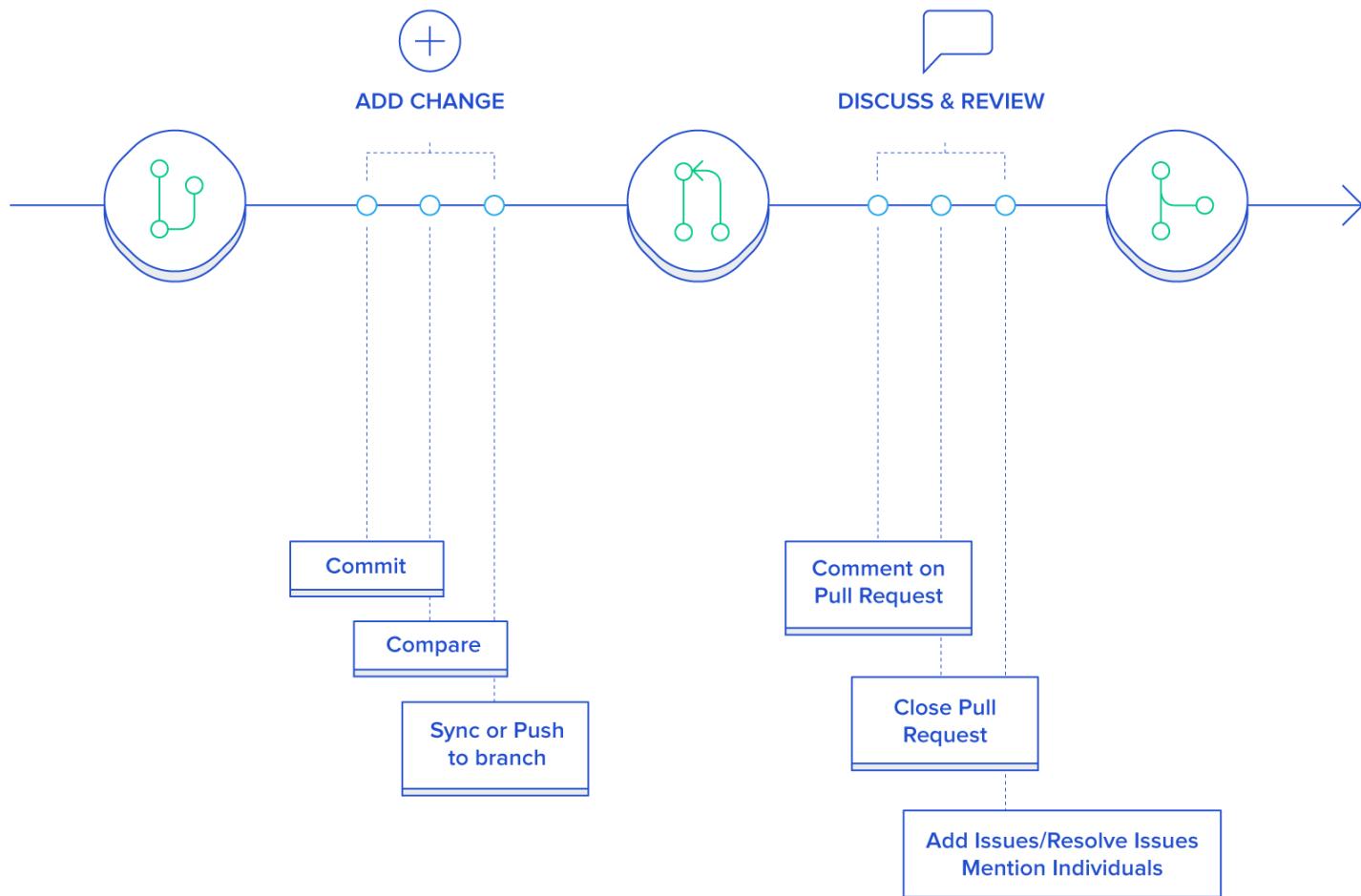
## Avantages

- Amélioration de la qualité du code
  - Partage de connaissances
  - Réduction des bugs
  - Une MR est un **outil de collaboration**, pas une formalité.
- 
- **Pas de MR = pas de travail en équipe**

# Règles d'équipe à ne jamais enfreindre

## Règles fondamentales en équipe

-  Ne jamais pousser directement sur la branche principale
-  Ne jamais réécrire l'historique d'une branche partagée
-  Ne jamais forcer un push sans accord (--force)
-  Une fonctionnalité = une branche
-  Toujours passer par une Pull / Merge Request
-  Toujours relire le code avant de merger
- Ces règles protègent **le projet et l'équipe**



# Styles de développement

- En général, un développeur effectue des demandes d'extraction (Pull Request) pour combiner les modifications qu'il a créées avec le projet principal.
- Un processus d'examen de ces changements est lancé par les réviseurs qui peuvent insérer des commentaires sur chaque élément qu'ils pensent pouvoir être amélioré ou considérer comme inutile.
- Après avoir reçu des commentaires, le créateur peut y répondre, créer une discussion, ou simplement la suivre et modifier son code en conséquence.



# Stratégies de branches

## **Objectif du module**

Montrer qu'en entreprise :

on ne branche pas au hasard

une stratégie évite le chaos

Git s'adapte à l'organisation, pas l'inverse

# Pourquoi une stratégie de branches ?

## Sans stratégie de branches

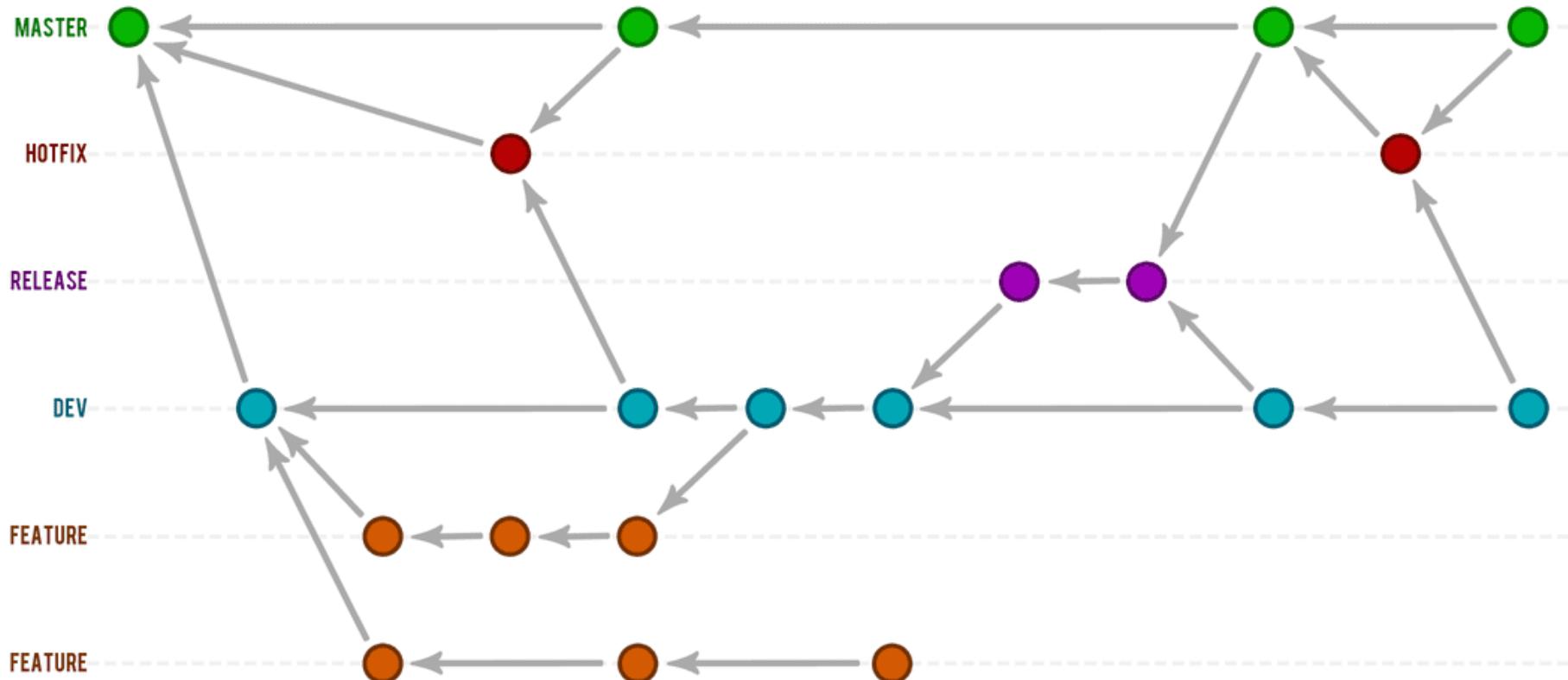
- Branches longues et obsolètes
- Conflits fréquents
- Code instable
- Déploiements risqués

## Avec une stratégie claire

- Règles connues de tous
- Intégration maîtrisée
- Code plus stable
- Livraisons plus fréquentes

Une stratégie de branches est une **règle d'équipe**, pas une préférence personnelle.

# Le modèle GitFlow



# Principe du GitFlow

- master / main : code en production
- develop : intégration des fonctionnalités
- feature/\* : développement de nouvelles fonctionnalités
- release/\* : préparation des versions
- hotfix/\* : corrections urgentes en production

Chaque type de branche a un rôle clair.

## Avantages

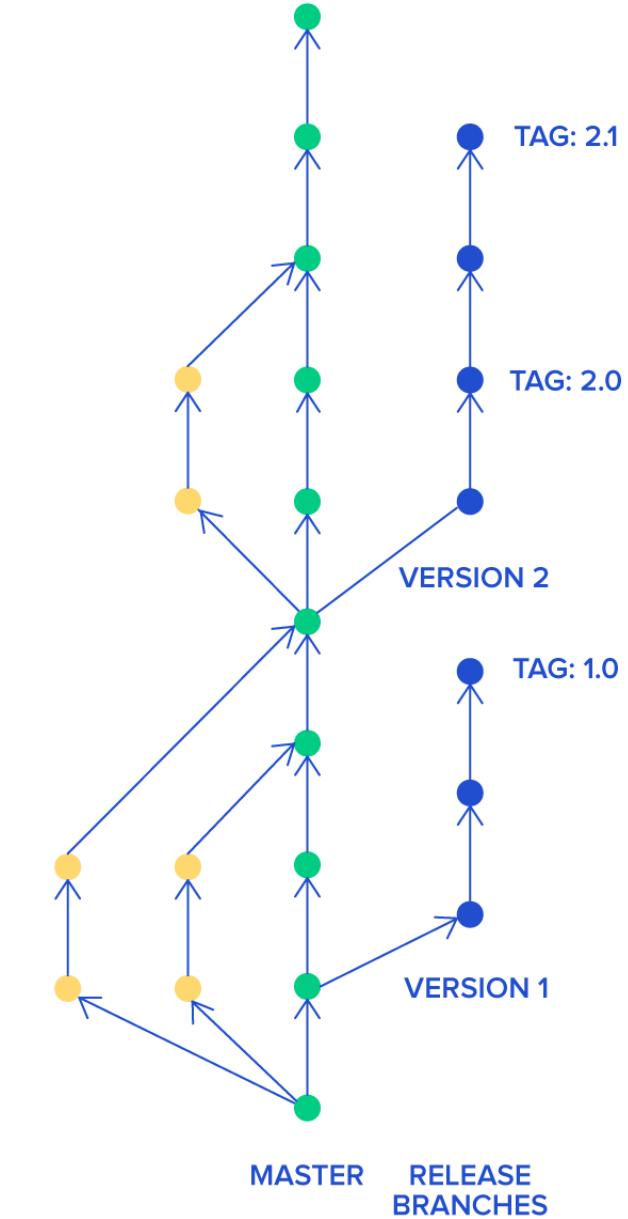
- Structure claire
- Bon pour équipes nombreuses
- Historique lisible

## Inconvénients

- Workflow complexe
- Trop de branches
- Peu adapté aux livraisons fréquentes

# Trunk-based Development Workflow

- tous les développeurs travaillent sur une seule branche avec un accès ouvert à celle-ci. Souvent, c'est simplement la masterbranche. Ils y commettent du code et l'exécutent. C'est ultra simple.



# Principe du Trunk-Based Development

- Une branche principale (main)
- Branches très courtes
- Intégration fréquente
- Déploiements rapides

Tous les développeurs travaillent proches du **tronc commun**.

## Avantages

- Historique simple
- Moins de conflits
- Livraison continue facilitée

## Inconvénients

- Discipline forte requise
- Tests automatisés indispensables
- Moins tolérant aux erreurs

# Quelle stratégie de branches choisir ?

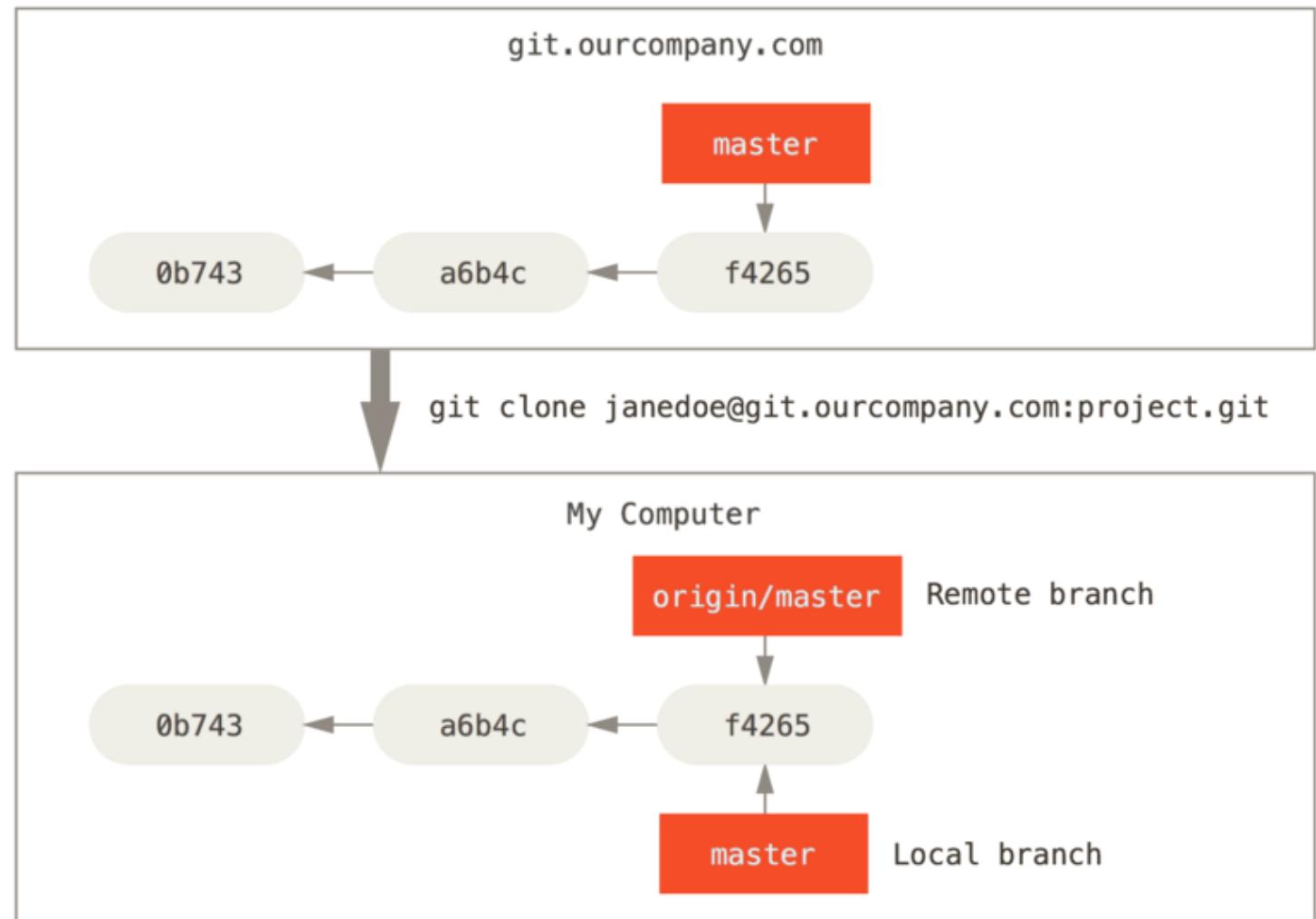
## Le choix dépend de :

- Taille de l'équipe
- Fréquence de livraison
- Niveau de maturité DevOps
- Existence de tests automatisés

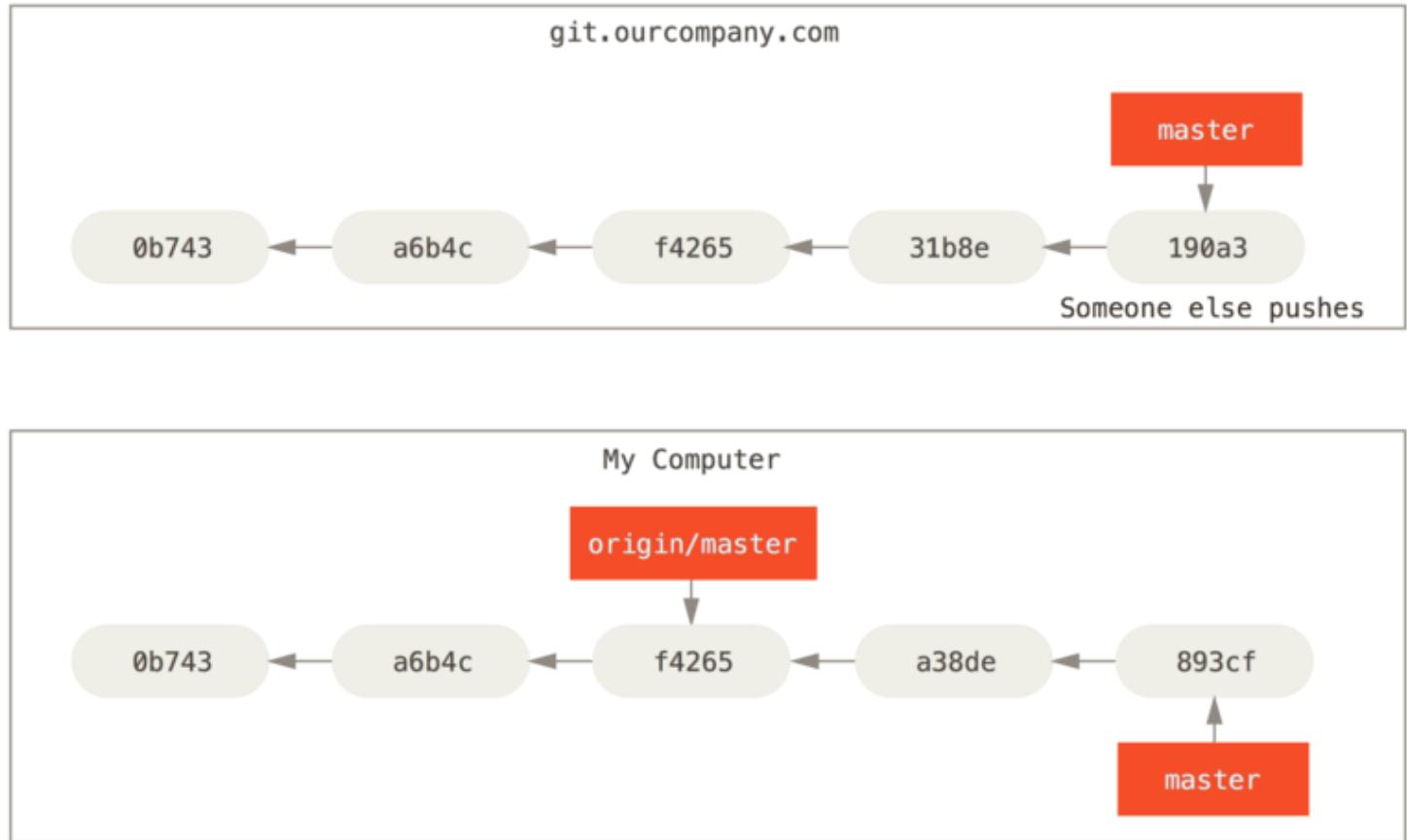
## Exemples

- **Petite équipe / livraisons fréquentes**  
→ Trunk-Based Development
- **Grande équipe / releases planifiées**  
→ GitFlow
- **Projet critique / legacy**  
→ Workflow hybride

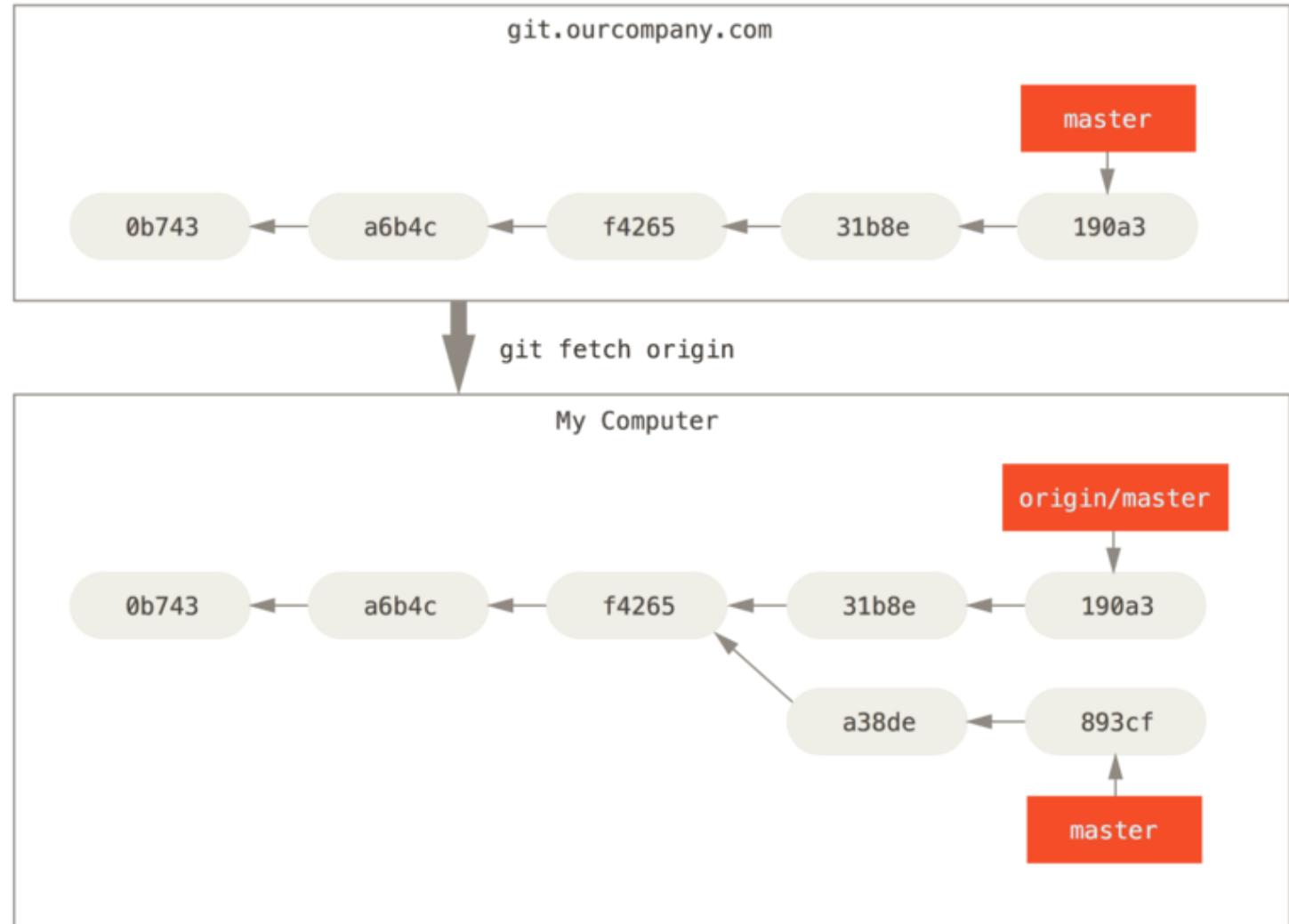
# Dépôts distant et local après un clone



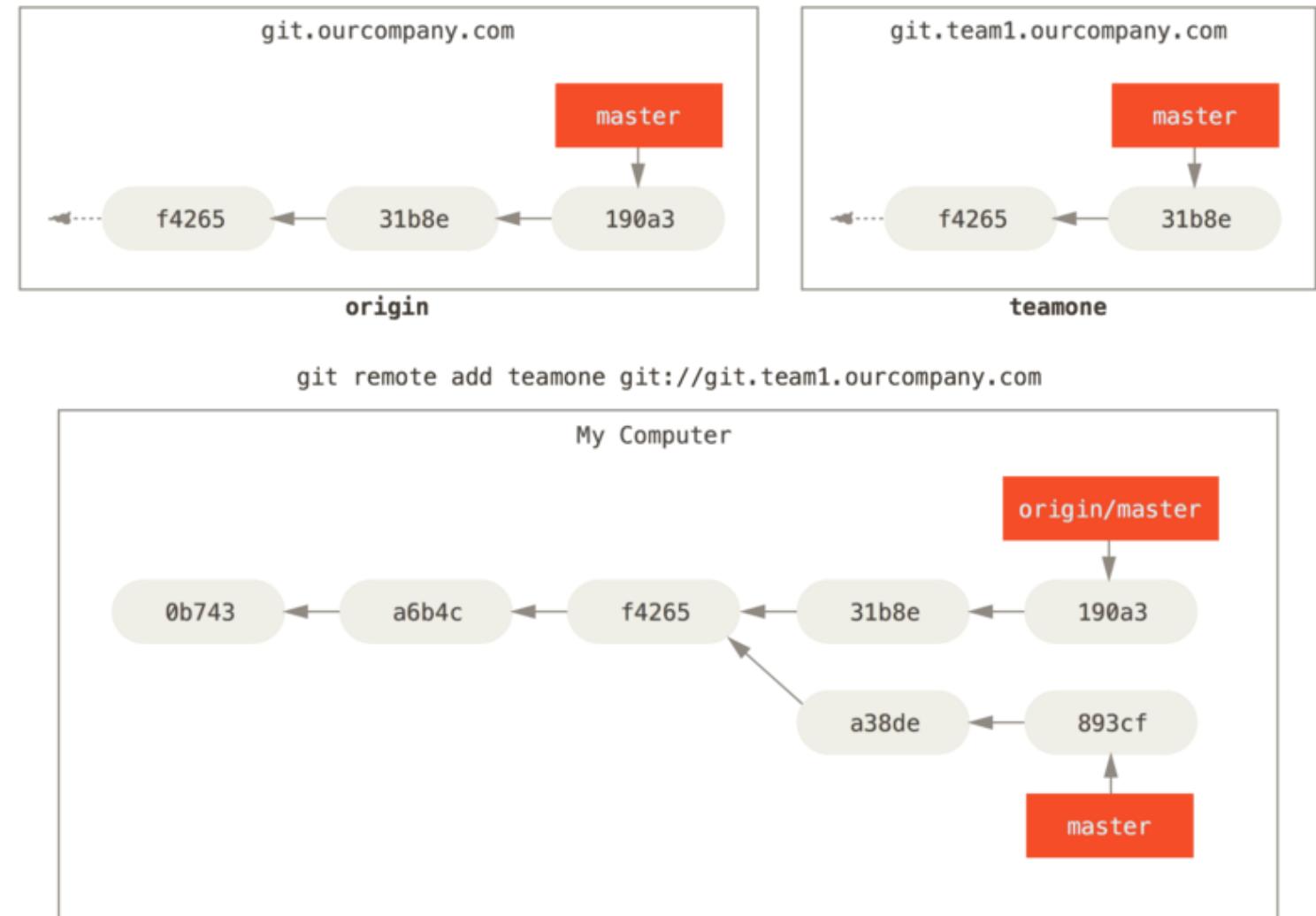
# Les travaux locaux et distants peuvent diverger



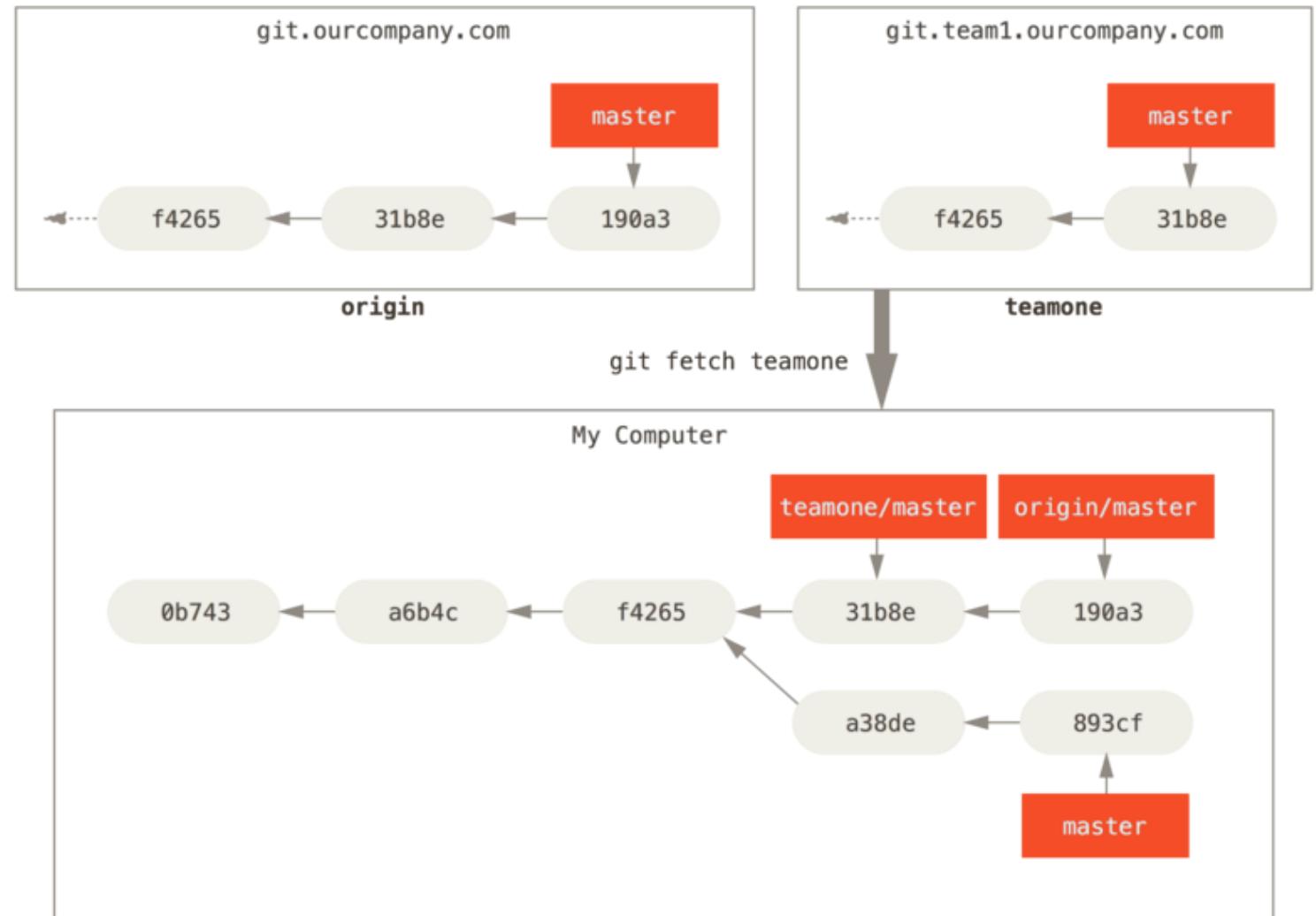
# git fetch met à jour vos branches de suivi à distance



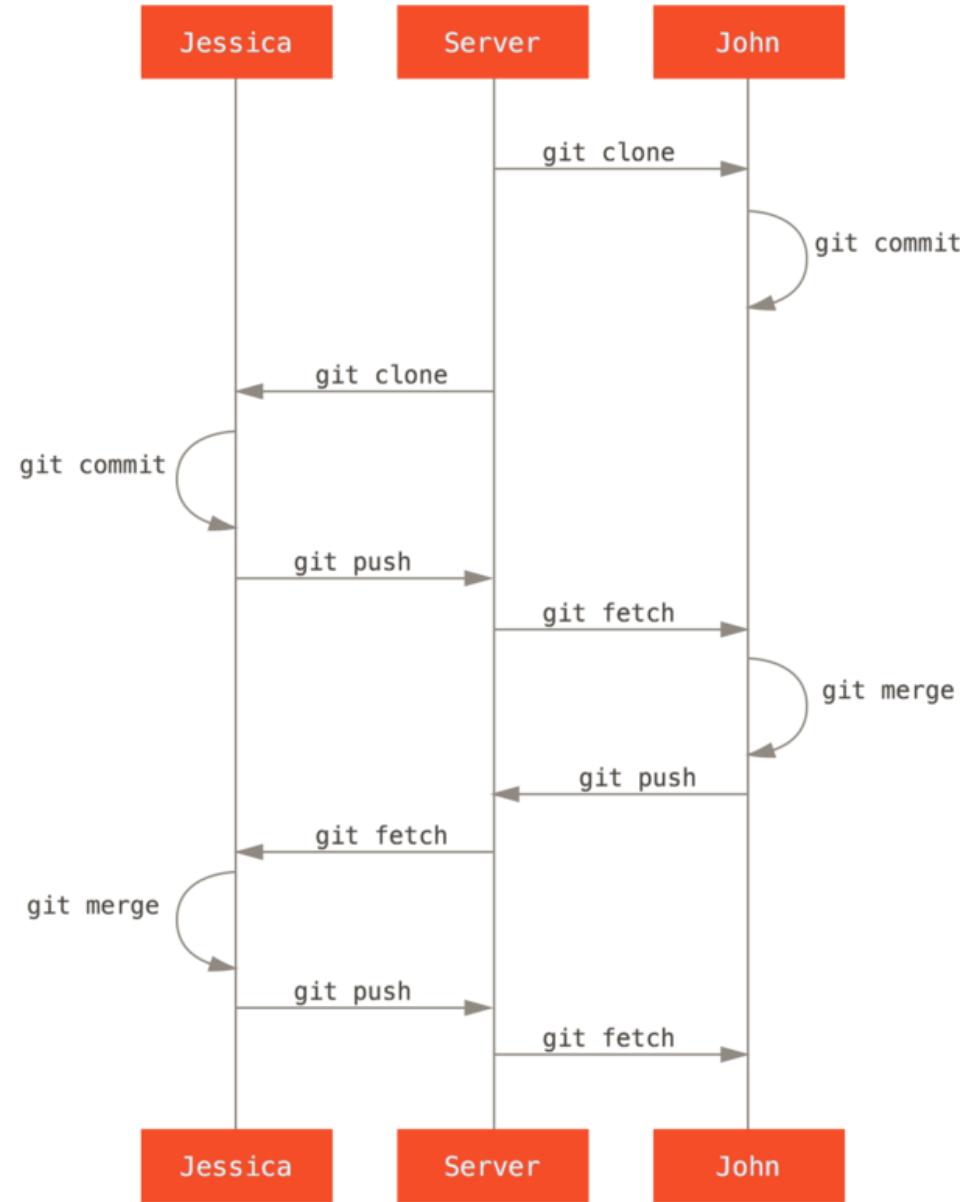
# Ajout d'un nouveau serveur en tant que référence distante



# Branche de suivi à distance teamone/master



# Séquence générale des événements pour une utilisation simple multi-développeur de Git.



# Flux de travail typique sur la machine d'un développeur

## #1. Clone the project repository

```
developer$ git clone  
https://gitlab.com/git_divya/structuralStrategy.git
```

## #2. Checkout 'dev'(or 'uat') branch and switch to the branch with the "-b" flag

```
developer:[release]$ git checkout -b dev
```

### # Add code changes on dev branch

```
developer:[dev]$git add . && git commit -m "Adding navigation  
code"
```

## #3. Create 'feature1' branch from an older project snapshot; say commit id-"62fc03f"

```
developer:[dev]$git checkout -b feature1 62fc03f
```

### #Develop the code for feature1 changes, add to index and commit the changes

```
developer:[feature1]$git add . && git commit -m "Adding  
feature1 code functions"
```

## #4. Create 'bugfix\_940' branch from an initial project snapshot with the tagid-"r1.0"

```
developer:[feature1]$git checkout -b bugfix_940 r1.0
```

### #Write and test the code and commit to this branch

```
developer:[bugfix_940]$git add . && git commit -m "fixed the  
bug#940"
```

## #5. Checkout 'dev' branch

```
developer:[bugfix_940]$git checkout dev
```

### # Merge 'feature1' and 'bugfix\_940' branches into dev resolving the conflicts if any

```
developer:[dev]$git merge feature1
```

```
developer:[dev]$git merge bugfix_940
```

### #6. Update local 'dev' branch with remote 'dev' branch using 'git pull' command to be in synch

```
developer:[dev]$git pull
```

### #7. Push local 'dev' branch merged changes to the Project remote repo

```
Developer:[dev]$git push
```

# Flux de travail typique sur la machine d'un Mainteneur

## #1. Clone the project repository

Maintainer:~ \$git clone https://gitlab.com/git\_divya/structuralStrategy.git

## #2. Checkout 'release' branch

Maintainer: [dev] \$git checkout release

## #3. Merge 'dev'(or 'uat') branch codes, resolve conflicts if any

Maintainer: [release] \$git commit -m "Merge dev changes"

## #4. Tag the latest release code

Maintainer:[release] \$git tag -a r1.6 -m "Tag latest dev changes" HEAD

## #5. Send to 'QA' team for approval, merge approved commits into 'release'

Maintainer: [release] \$git merge QA

## #6. Update 'dev' branch with latest commits on 'release' branch

Maintainer: [release] \$git checkout dev

Maintainer: [dev] \$git merge release

## #7. Maintainer push all new commits to the repository

Maintainer: [release] \$git push --all

# Git avancé

## **Objectif du module**

Donner de la puissance **sans danger** :

expliquer ce qui existe

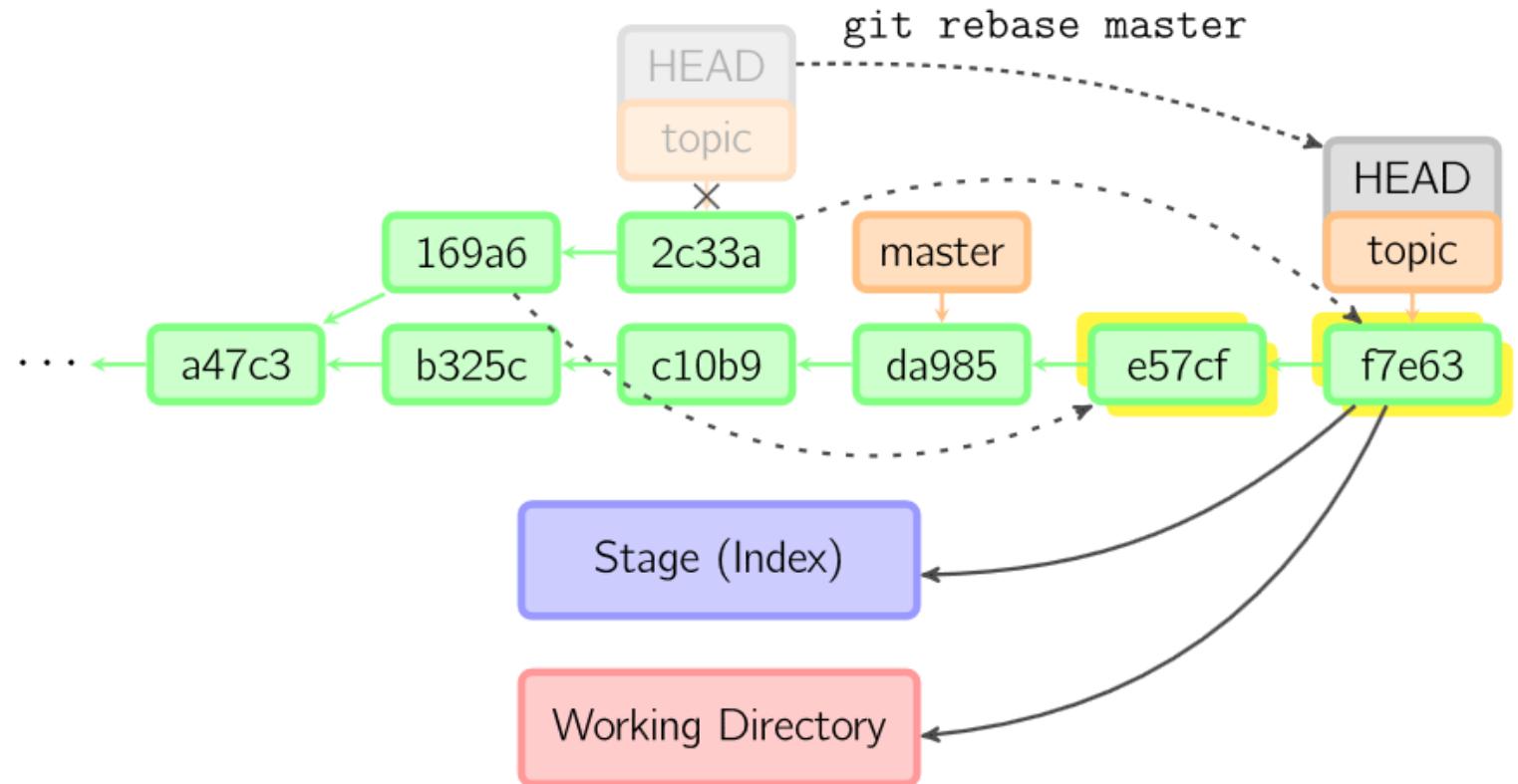
cadrer strictement l'usage

éviter les erreurs irréversibles

# merge vs rebase

- **git merge**
  - Conserve l'historique
  - Crée un commit de merge
  - Sûr pour les branches partagées
- **git rebase**
  - Réécrit l'historique
  - Crée un historique linéaire
  - À utiliser **localemement uniquement**
- **✗ Never rebase a shared branch**

# rebase



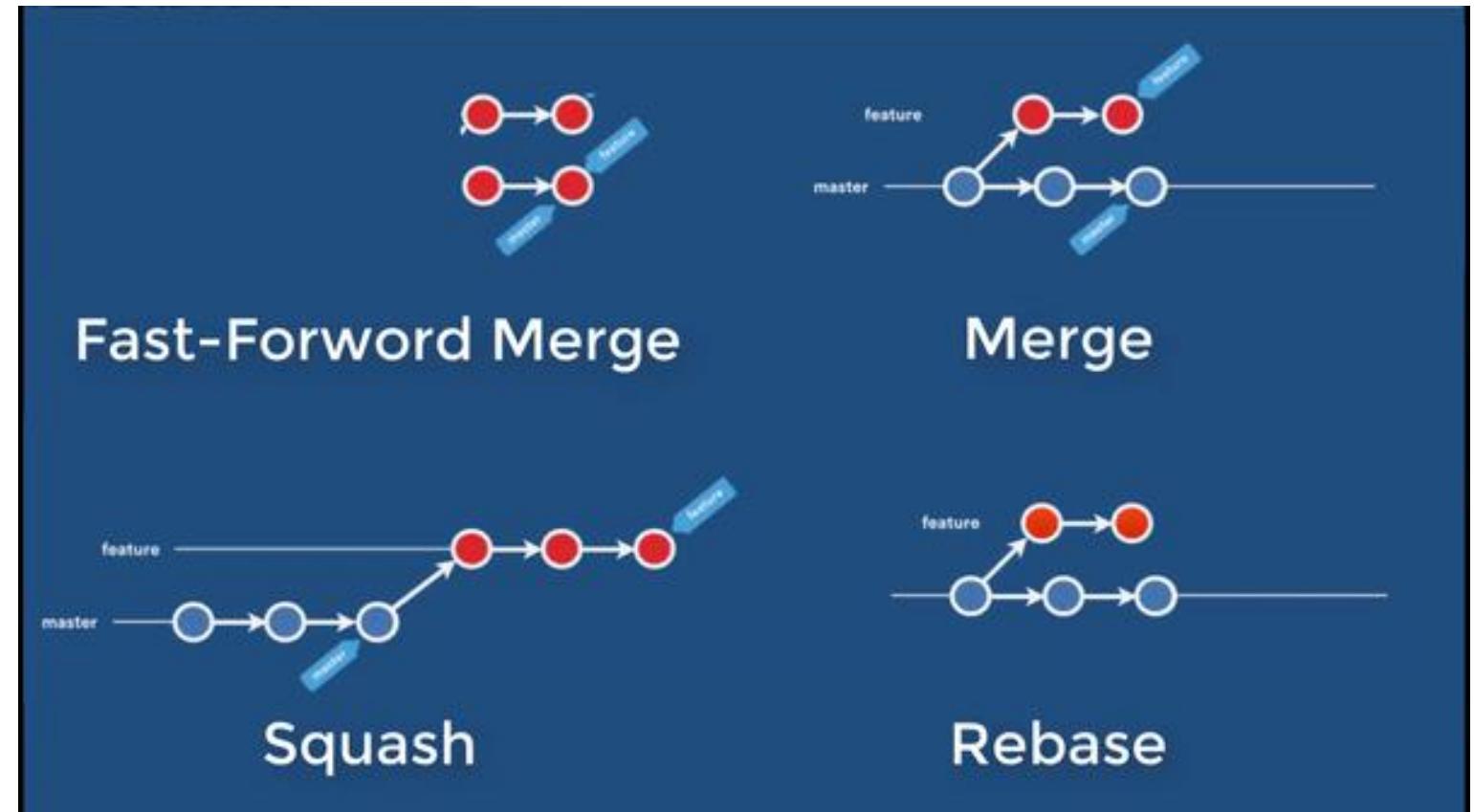
## The golden rule of rebase

“No one shall rebase a shared branch”

Eliès Jebri

66

# Fusion des branches



# git reset vs git revert

## git reset

- Déplace un pointeur de branche
- Réécrit l'historique
- Efface ou modifie des commits

`git reset --hard <commit>`

À utiliser **localement uniquement**

## git revert

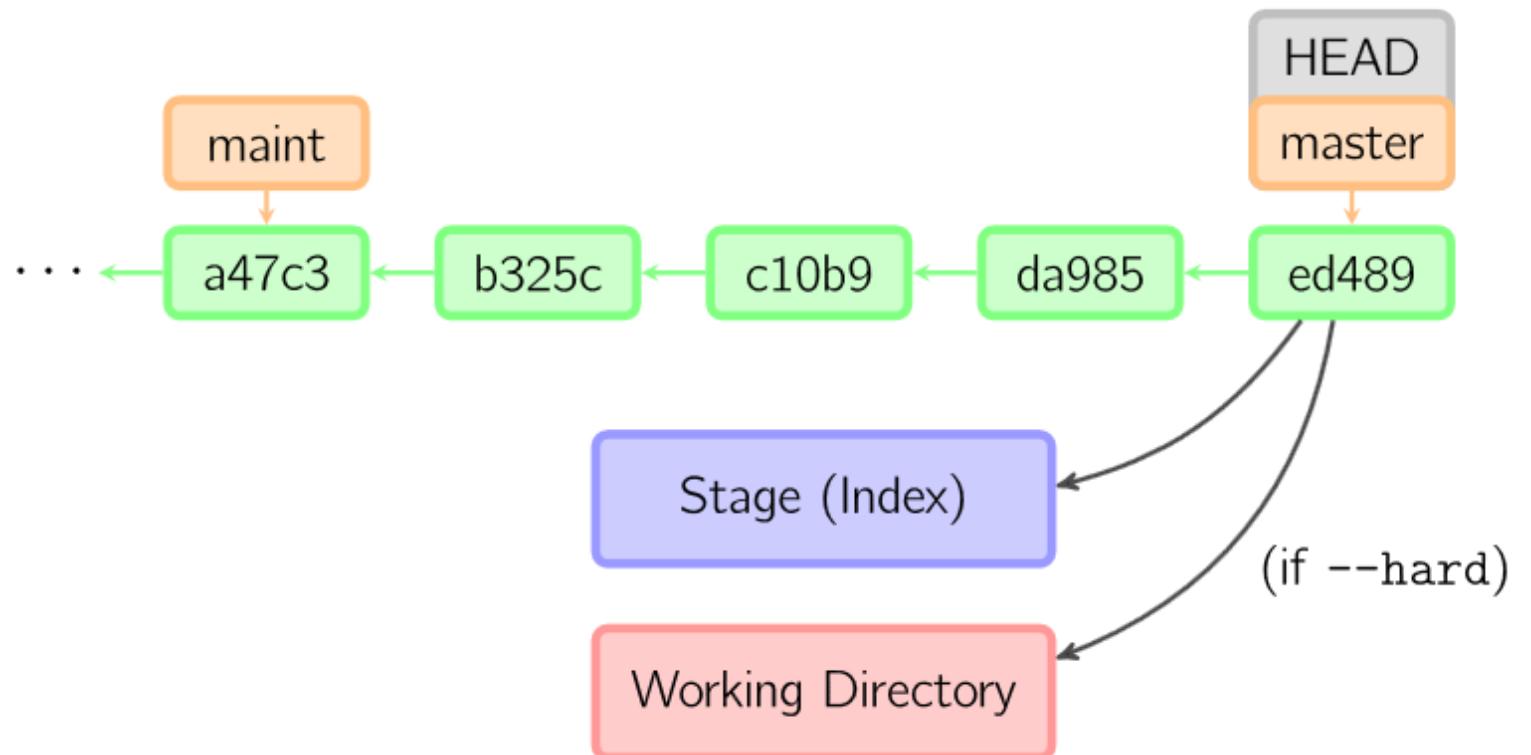
- Crée un **nouveau commit**
- Annule l'effet d'un commit précédent
- Historique conservé

`git revert <commit>`

À utiliser sur les branches partagées / production

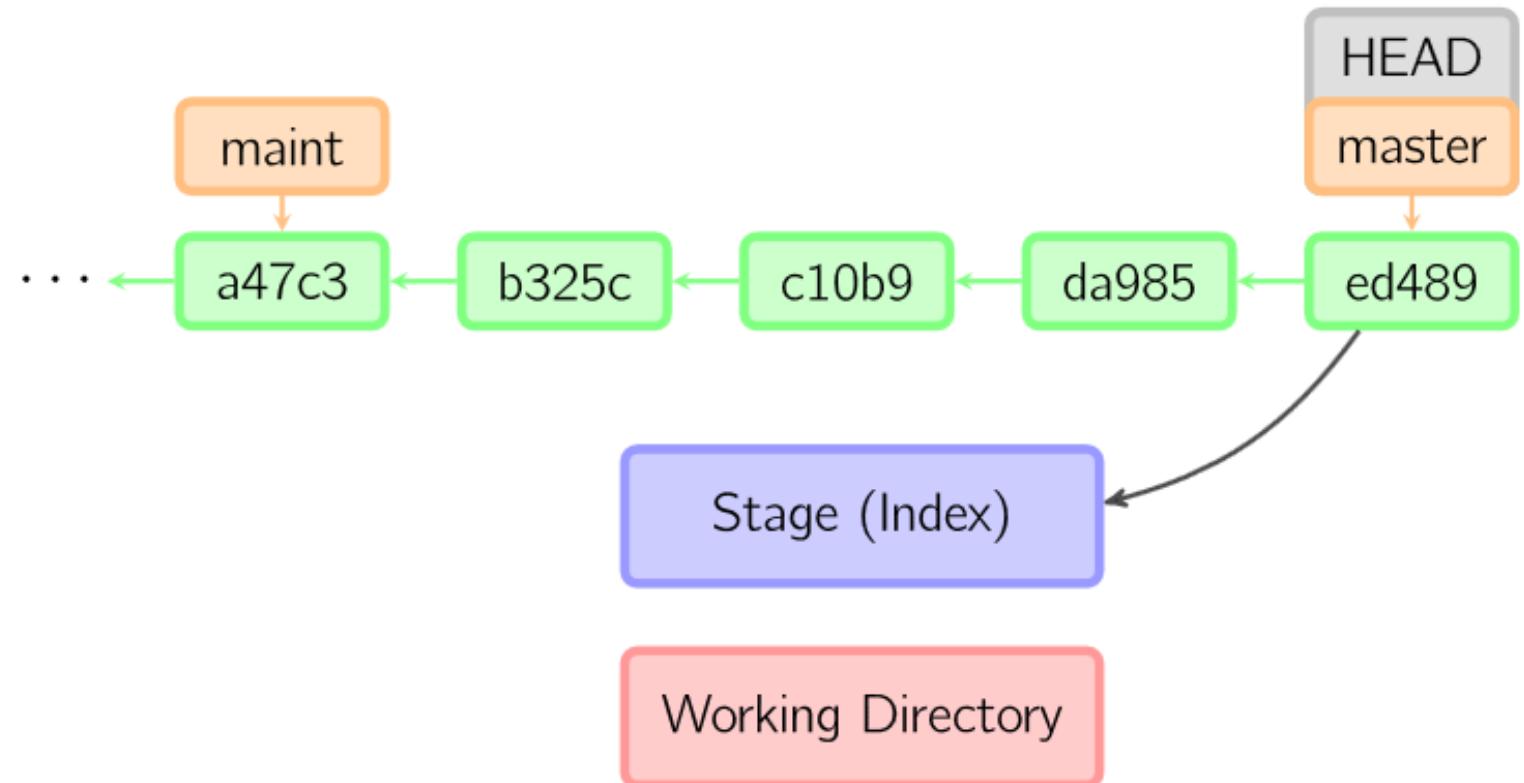
# reset

git reset

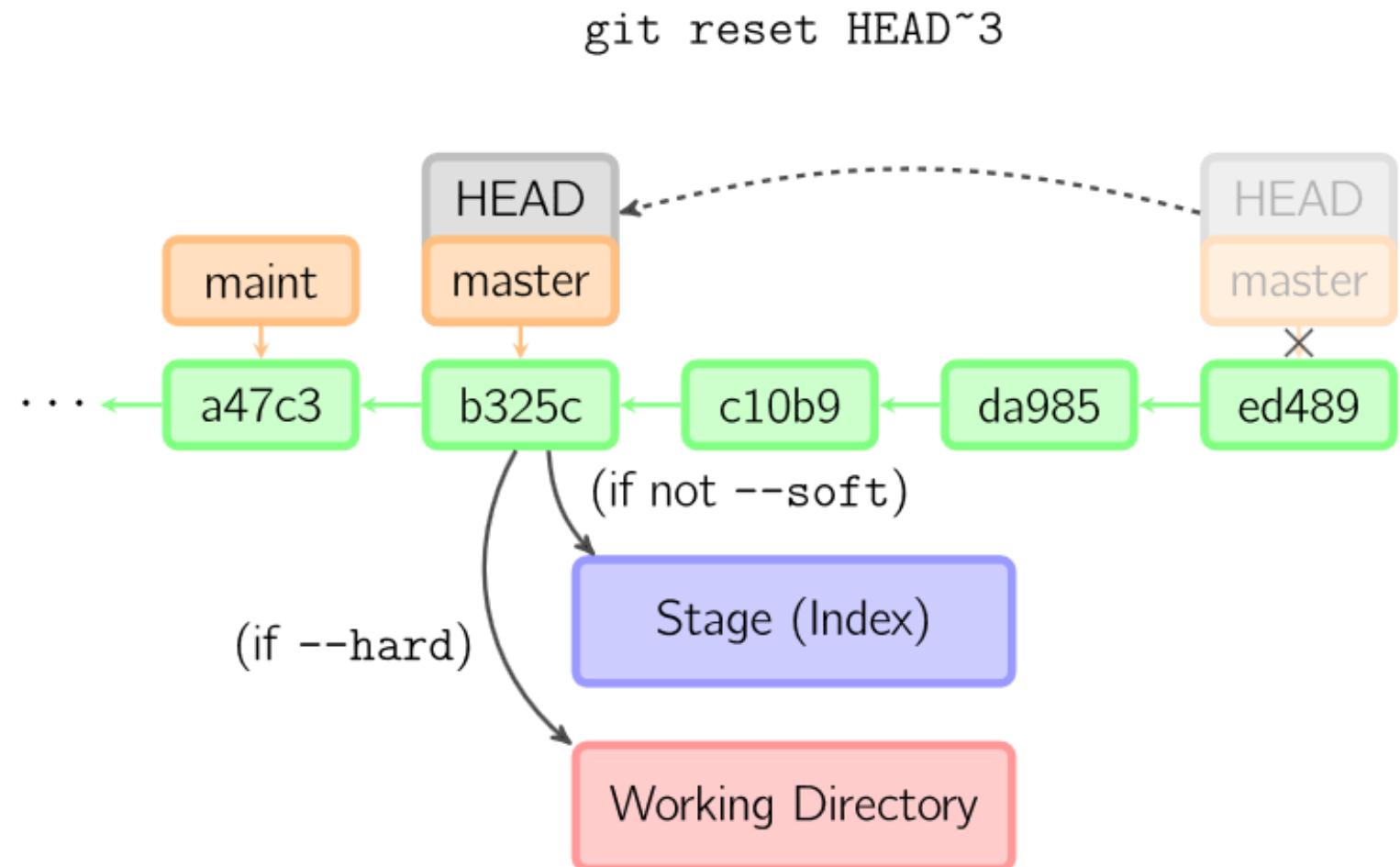


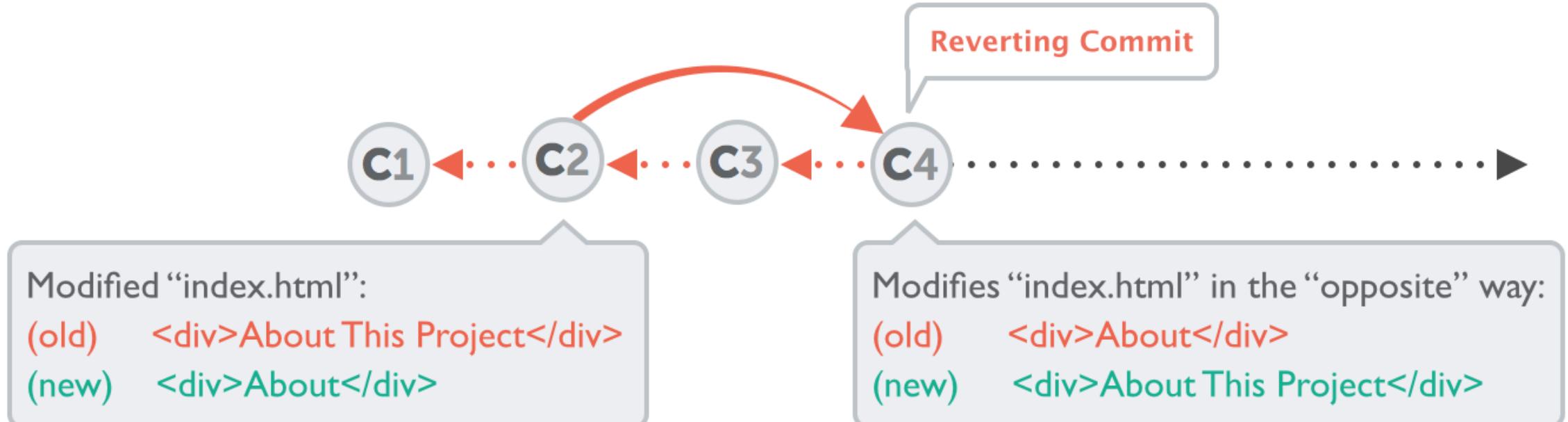
# reset fichier

```
git reset -- files
```



# reset destination



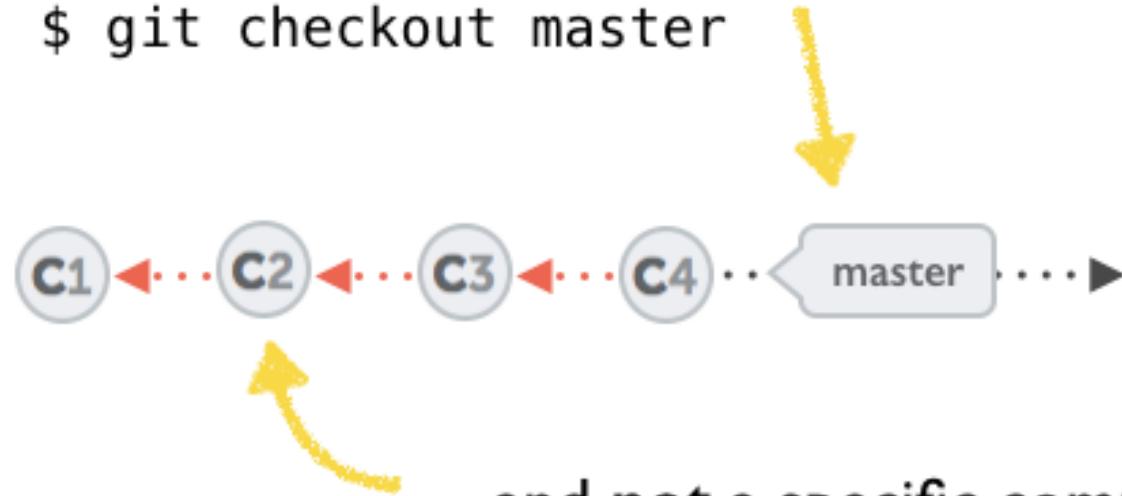


# git revert – Annuler sans réécrire l'historique

- **Principe**
- Annule un commit **en créant un nouveau commit**
- L'historique reste intact
- Méthode sûre pour les branches partagées / production

**usually, you check out a branch:**

```
$ git checkout master
```



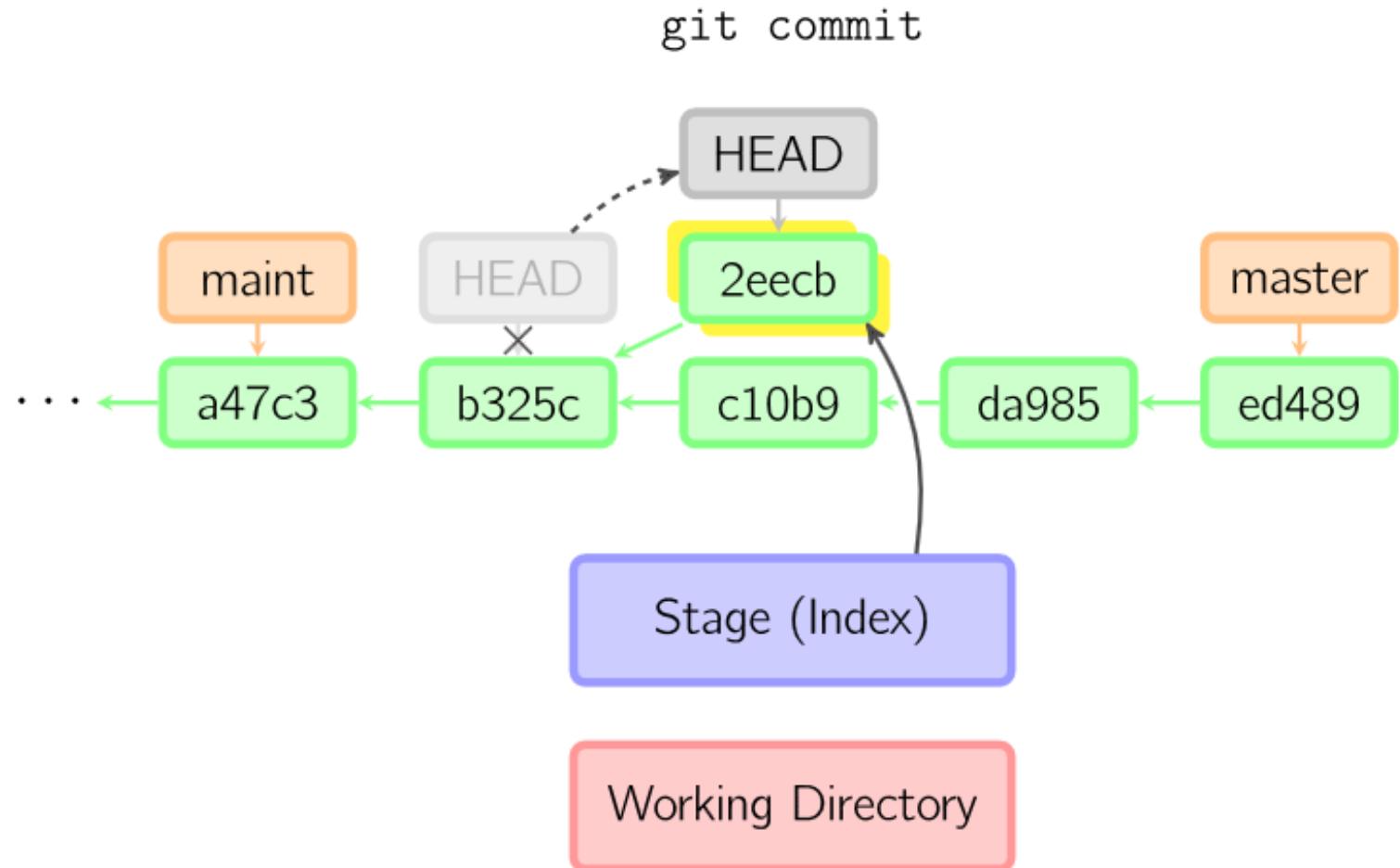
**...and not a specific commit:**

```
$ git checkout a05ef02
```

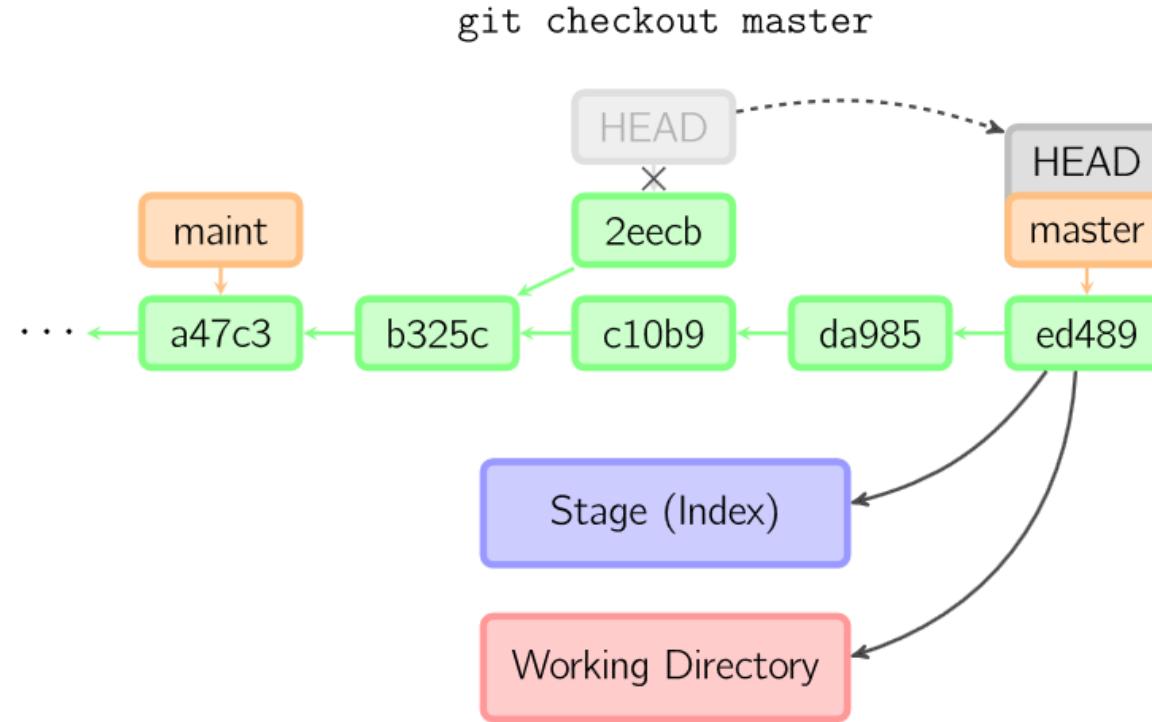
## Detached HEAD

- **Qu'est-ce que le Detached HEAD ?**
- HEAD ne pointe plus vers une branche
- Git est positionné sur un **commit précis**
- Les commits faits ici peuvent être perdus

# Committer avec une "Detached HEAD"

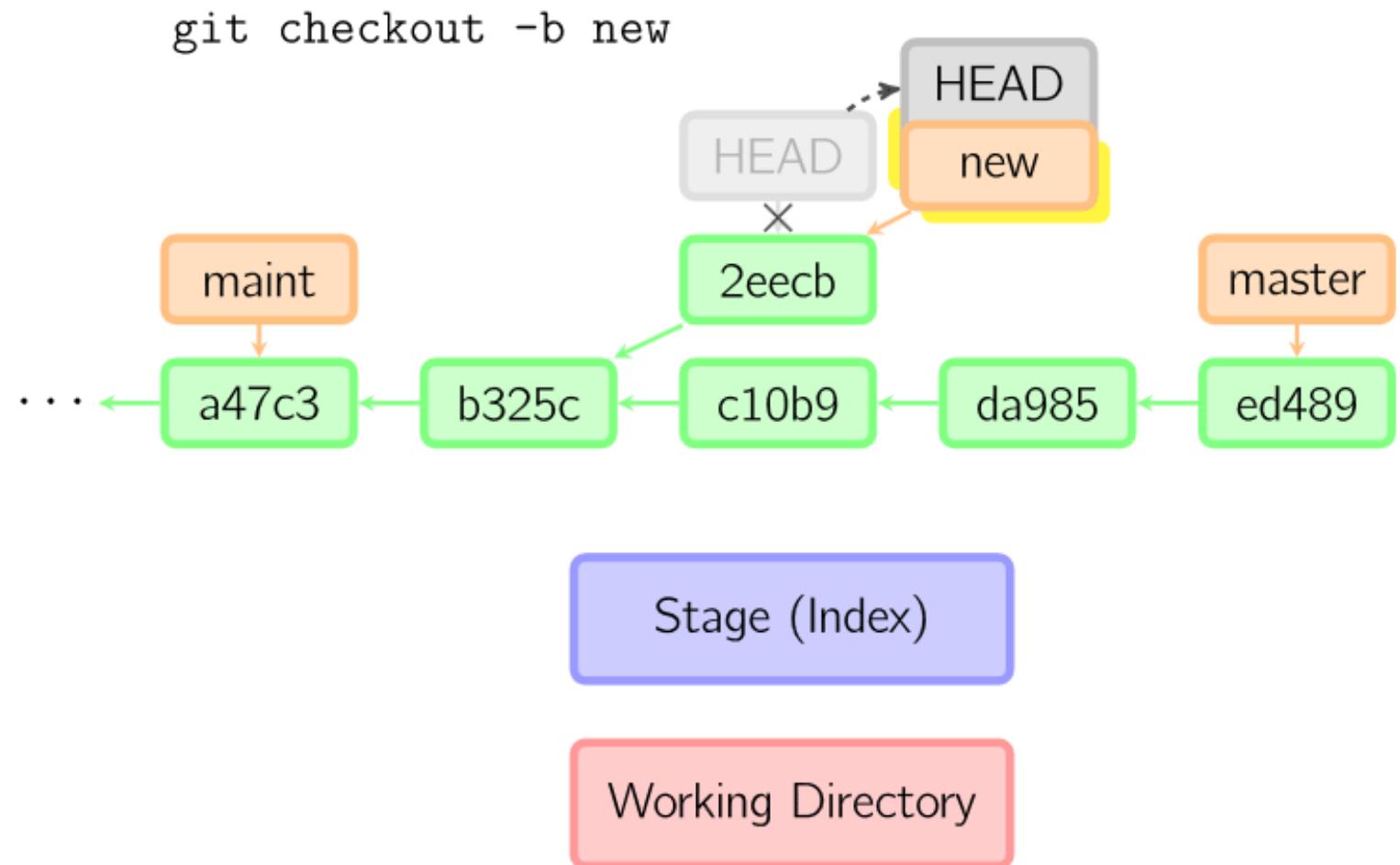


# Commit non référencé et perdu



- **Pourquoi c'est dangereux ?**
  - Les commits ne sont attachés à aucune branche
  - Risque de perte si on change de branche

# Conserver un detached HEAD



# git stash

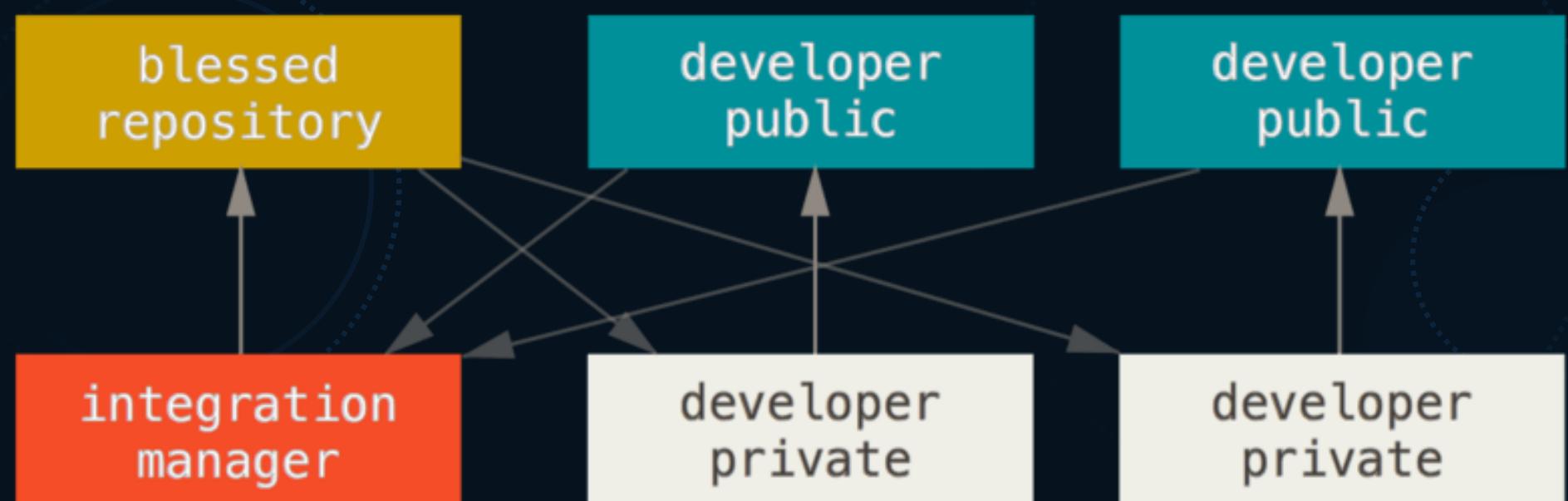
- Sauvegarder temporairement des modifications
- Nettoyer le working tree
- Changer de branche rapidement
  - git stash
- Récupérer son travail
  - git stash pop
  - OU
  - git stash apply

# Git distribué Qui commande?

Développements distribués

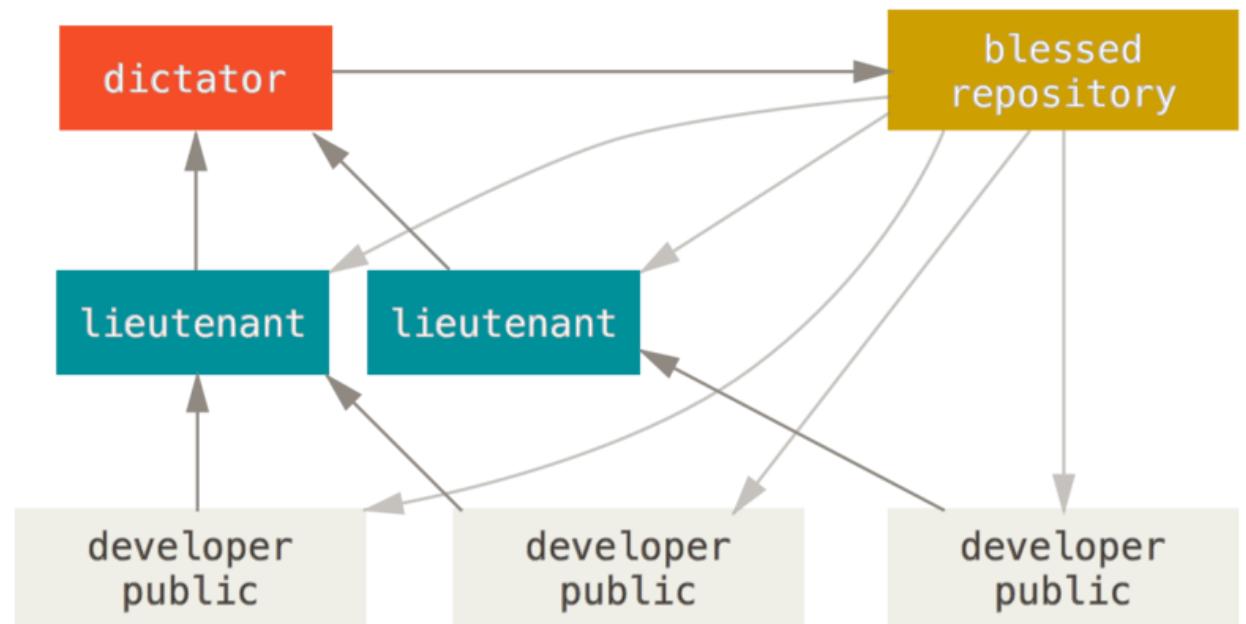
# Mode du gestionnaire d'intégration

- Le mainteneur du projet pousse vers son dépôt public.
- Un contributeur clone ce dépôt et introduit des modifications.
- Le contributeur pousse son travail sur son dépôt public.
- Le contributeur envoie au mainteneur un e-mail de demande pour tirer ses modifications depuis son dépôt.
- Le mainteneur ajoute le dépôt du contributeur comme dépôt distant et fusionne les modifications localement.
- Le mainteneur pousse les modifications fusionnées sur le dépôt principal.



# Mode dictateur et ses lieutenants

- Les simples développeurs travaillent sur la branche thématique et rebasent leur travail sur master. La branche master est celle du dictateur.
- Les lieutenants fusionnent les branches thématiques des développeurs dans leur propre branche master.
- Le dictateur fusionne les branches master de ses lieutenants dans sa propre branche master.
- Le dictateur pousse sa branche master sur le dépôt de référence pour que les développeurs se rebasent dessus.



# Références

- <https://git-scm.com/book/fr/v2/>
- <https://rogerdudler.github.io/git-guide/>
- <https://www.daolf.com/posts/git-series-part-1/>
- <https://www.jquery-az.com/git-github-tutorials/>
- <https://openclassrooms.com/fr/courses/1233741-gerez-vos-codes-source-avec-git>