



make it clever



GitOps
avec GitHub et ArgoCD

Plan du séminaire

- **Introduction à GitOps**

- Origines de GitOps
- Problèmes adressés par GitOps
- Présentation des concepts GitOps

- **Gestion de versions avec Git**

- Principes de Git
- Référentiel
- Commit
- Configuration
- Synchronisation avec le référentiel
- Commandes essentielles
- Gestion des branches et des conflits

- **La CI/CD avec GitHub Actions**

- Présentation de GitHub Actions
- Automatisation avec GitHub Actions
- Déploiement avec GitHub Actions

- **Implémenter le déploiement avec ArgoCD**

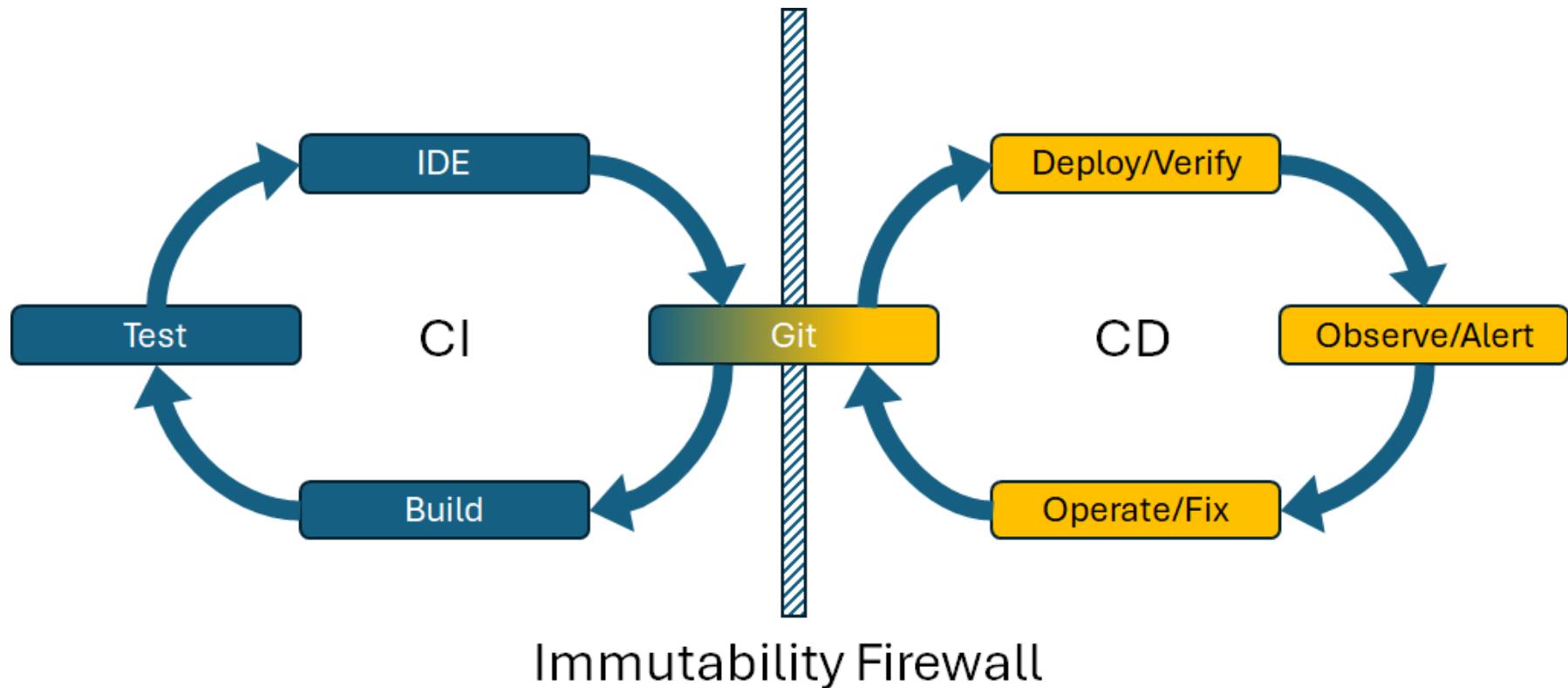
- Introduction à ArgoCD
- Déploiement avec ArgoCD
- Gestion de la configuration avec Helm
- GitOps et la sécurité

- **Conclusion**

- Intérêt de la démarche GitOps
- Difficultés de l'implémentation

Introduction à GitOps

- **DevOps + Git = GitOps**
- La démarche DevOps centrée sur l'utilisation de Git



- **Avant 2013**

- Virtualisation lourde KVM/Xen/VMWare
- Ou légère via conteneurs OpenVZ / LXC
- Isolation de processus avec chroot

- **2013**

- Naissance de Docker
- Infra déclarative, reproductibilité, séparation code/données

- **Avant 2014**

- On utilise pas vraiment Docker en production
- Load Balancing, Failover, Scaling “à l’ancienne”

- **2014-2015**

- Avènement des orchestrateurs de conteneurs
- Naissance de Kubernetes



vmware[®]
by Broadcom



docker



kubernetes

- **2016**
 - Un ingénieur Weaveworks supprime toute l'infra par erreur
 - L'équipe parvient à tout reconstruire en 40min
 - En bénissant le fait que toutes les configurations étaient tracées dans Git !
- **2018**
 - Adoption progressive de la philosophie par l'industrie et les cloud providers
 - AWS, Azure, Google Cloud
- **2019**
 - Weaveworks publie les outils de livraison Flux et Flagger
- **2020**
 - Fondation d'un groupe de travail GitOps : <https://opengitops.dev/>
- **⇒ 4 grands principes**



- **Déclaratif**
 - L'état souhaité du système doit être exprimé de manière **déclarative**
- **Versionné et immuable**
 - L'état souhaité est **stocké** pour garantir **immuabilité, versions et historique**
- **Tiré (pull) automatiquement**
 - Cet état est extrait **automatiquement** de la source par des **agents logiciels**
- **Continuellement réconcilié**
 - Ces agents logiciels **observent en continu** l'état réel du système
 - Ils **tentent d'appliquer** l'état souhaité

- **Plein d'outils, pleins de fonctionnalités !**

- Docker
- Kubernetes (k8s)
- Terraform
- Ansible / Chef / Puppet
- Kustomize / Helm
- Git
- Gitlab / Github
- Jenkins / Github Actions
- Flux / ArgoCD
- Prometheus / Grafana
- ELK
- Packaging devops/applicatif
- Cluster, gestion conteneurs
- Provisioning “déclaratif”
- Infrastructure
- Système
- Gestion Sources
- Gestion projet
- CI
- CD
- Job Scheduler
- Monitoring

- **DevOps / Git**
- **Git utilisé pour :**
 - Dépôt applicatif
 - Code application uniquement (périmètre dev)
 - Dépôt devops/système
 - Quelques scripts et conf (périmètre devops)
 - Beaucoup de conf non versionnées
 - Transformée en “donnée” à sauvegarder
- **Mode déploiement**
 - **Push**, surtout pour l'application
- **DevOps + Git**
- **Git utilisé pour :**
 - Dépôt applicatif
 - Conf Docker
 - Instructions de build/déploiement
 - Dépôt devops
 - Versionning de toute l'infra et configurations
 - Chaînes de build/déploiement
 - Gestion des workflows (dev / projet)
 - Merge Requests / Reviews : permet à tout le monde de contribuer de manière sûre sur n'importe quel repo
 - Tests syntaxe / unitaires / fonctionnels
- **Mode déploiement**
 - **Pull**, au moins pour l'application

- **Consistance et fiabilité**
 - Comment assurer la **cohérence** entre les environnements (dev / test / prod...) ?
 - Comment réduire les **erreurs humaines** ?
- **Gestion des changements**
 - Comment assurer la **tracabilité** de **tous** les changements ?
 - Comment assurer qu'ils ont tous été **revus** et **traités** ?
- **Automatisation / Reproductibilité**
 - Comment assurer l'**automatisation** entre :
 - La **validation** d'un changement <—— et ——> Le processus de **mise à jour** de l'infrastructure ?
 - Comment **reproduire** facilement un environnement déployé ?

- **Collaboration entre les équipes Dev et Ops**
 - Comment favoriser la **culture DevOps** ?
 - Comment faire participer les dev aux **configurations** de l'environnement ?
- **Gestion de la complexité**
 - Comment faire face à la **complexité croissante** des architectures ?
 - En particulier avec les architectures **microservices Cloud** ?

- **Infrastructure as Code (IaC)**
 - Gestion de l'infrastructure par des scripts de code
 - Voir description de toute l'infrastructure via du code !
- **Augmentation de la fiabilité**
- **Réduction du risque / erreur humaine**
- **Réduction du temps de déploiement**
 - Coûts \$ ☺
- **Amélioration de la cohérence et reproductibilité de l'infrastructure**
 - Voir reproductibilité complète de toute l'infrastructure !



- **LE gestionnaire de code source / versions**
 - Crée par Linus Torvalds en 2005 pour gérer le code de Linux
 - **Décentralisé**, en remplacement de BitKeeper, devenu payant
- **Première version en 2 semaines (avril 2005)**
 - Maintenu depuis par Juno Hamano
 - 2008 : ouverture de la plateforme GitHub
 - 2011 : fondation de GitLab
- **Dans l'écosystème Git : bien plus que du versionning !**
 - 2018 : Microsoft rachète GitHub 7.5 milliards
 - 2023 : Plus de 119 millions de repos publics en septembre

- **Git en tant que Source de vérité**
 - Un principe fondamental de GitOps
 - Git en tant que source de vérité **unique**
 - Pour décrire l'état désiré de votre système
- **Merge/pull requests !!**

- **Intégration Continue - Continuous Integration (CI)**

- Pilier de la philosophie DevOps et donc de GitOps
- Intégrer régulièrement les modifications de code pour éviter les conflits de fusion
- Accélère le rythme de livraison
- Améliore la qualité grâce aux tests



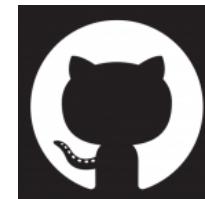
GitHub Actions

- **Il peut être pertinent de découpler CI et CD**
 - La pipeline **CI** produit un livrable/artefact
 - La **CD** déploie l'artefact sur un environnement
- **On peut vouloir relancer la CI sans forcément déployer**
- **De même, on peut vouloir déployer plusieurs fois le même artefact**
 - Sur environnements différents ou re-déploiement “à blanc”

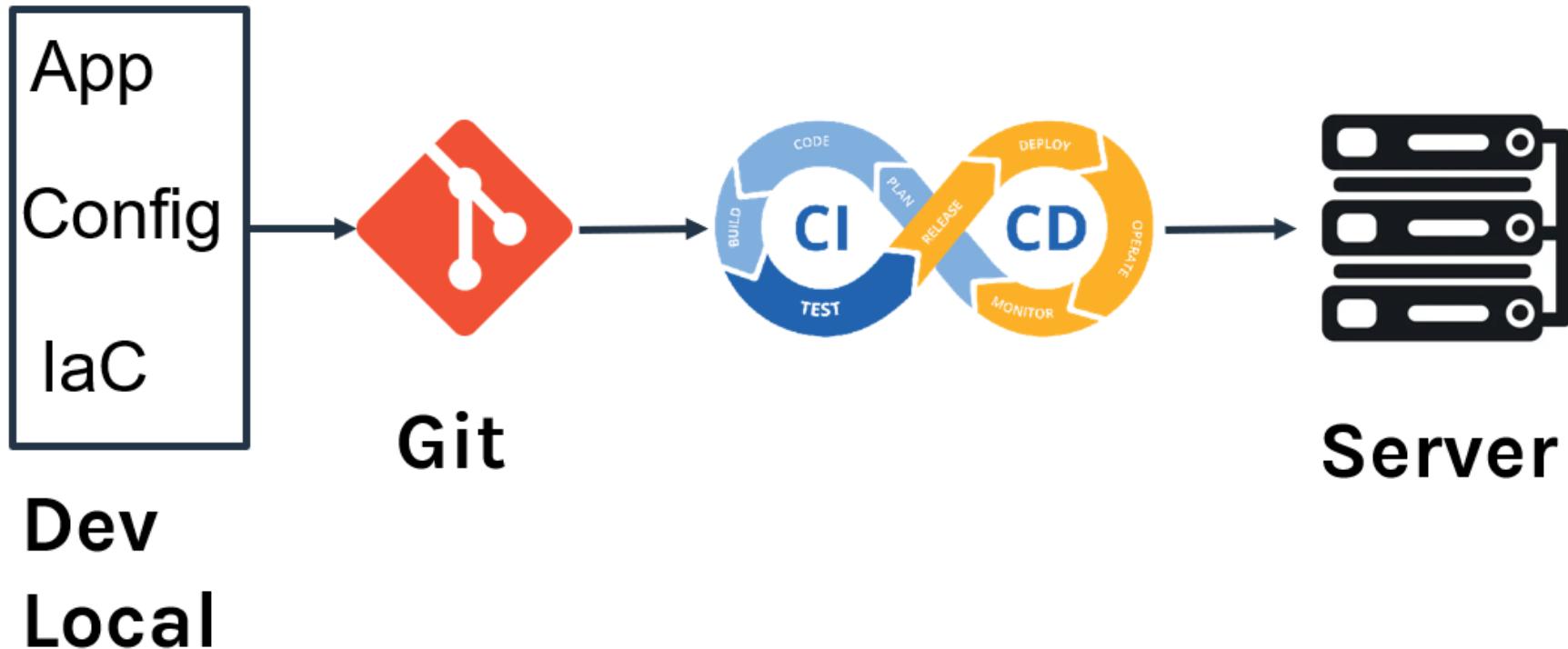
1 run CI → 1 run CD

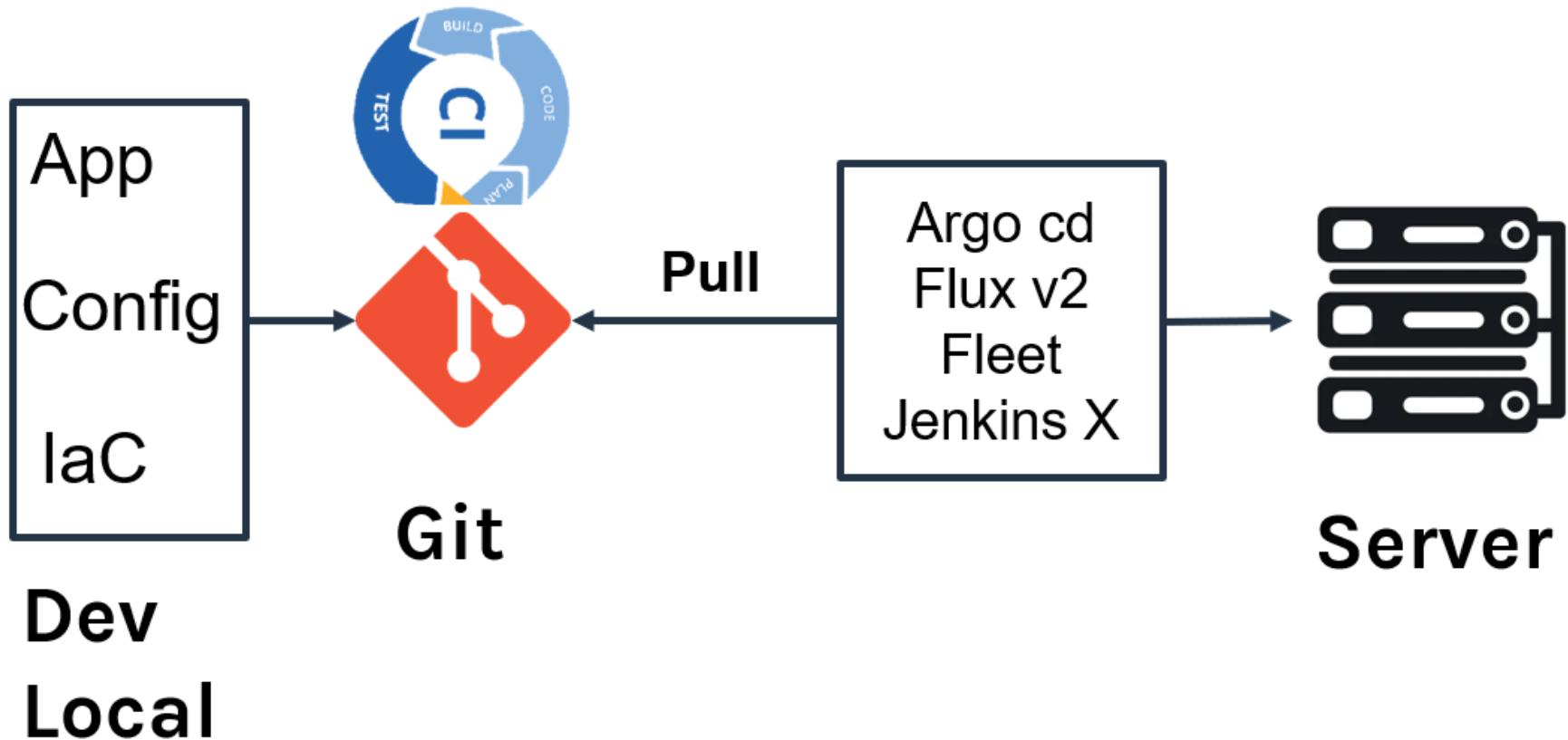


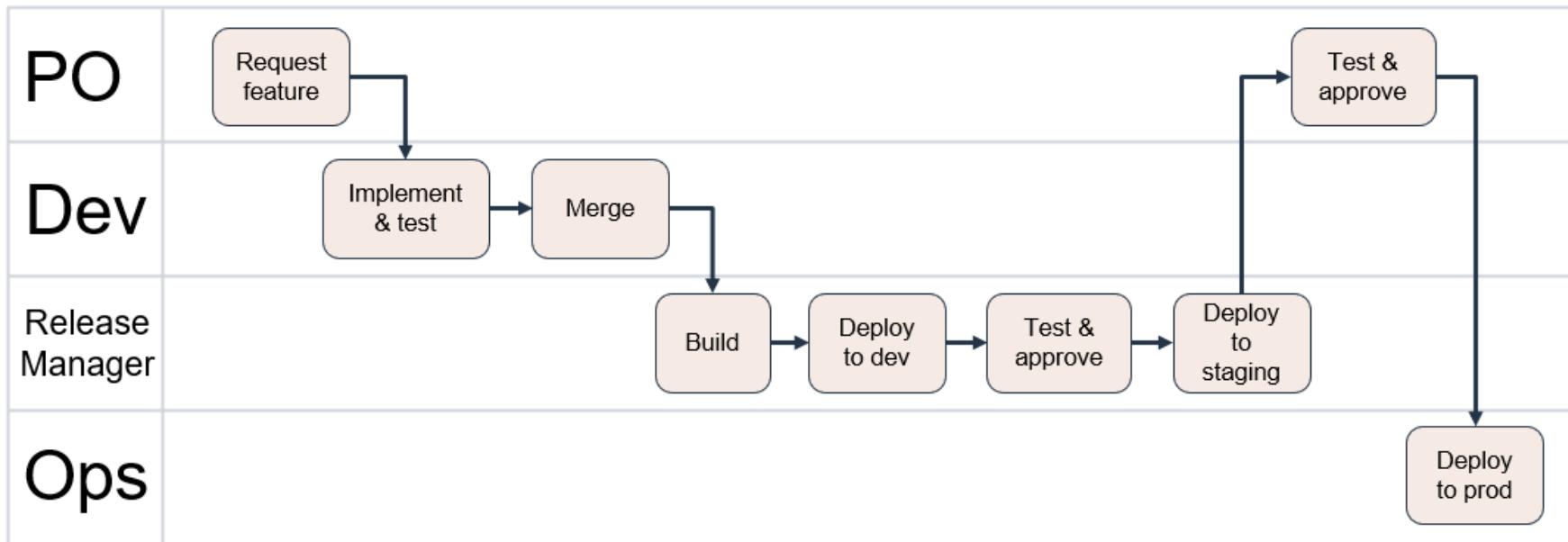
Travis CI

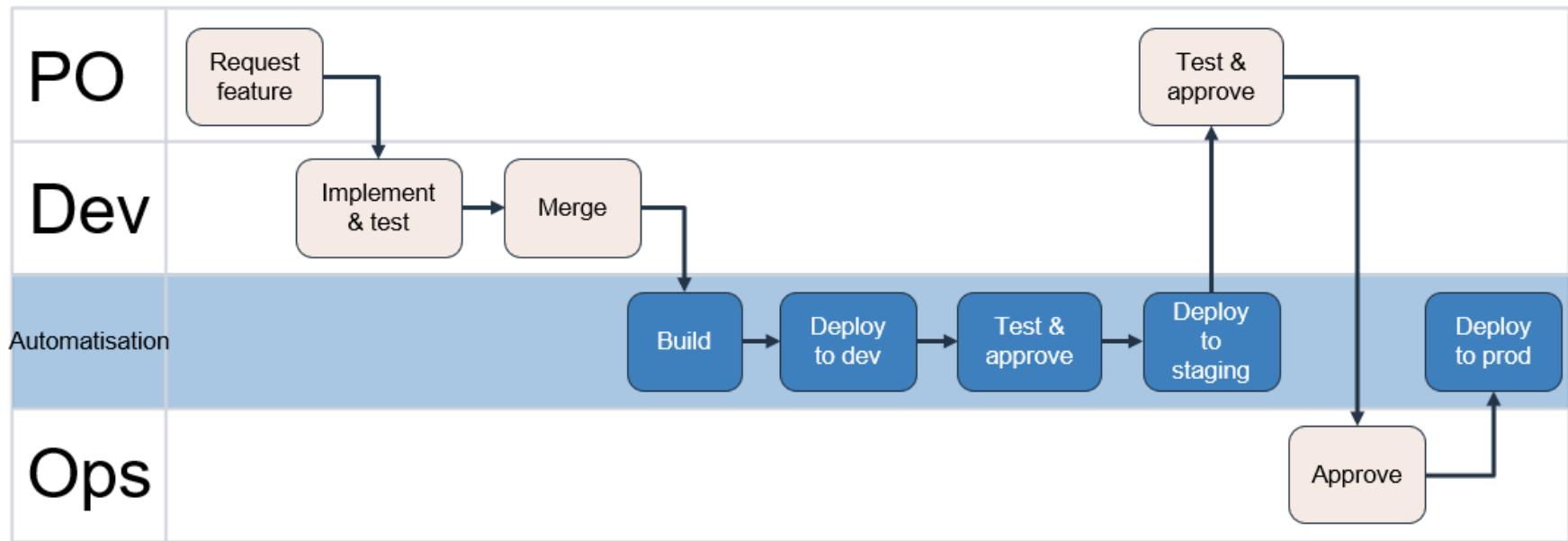


GitHub Actions









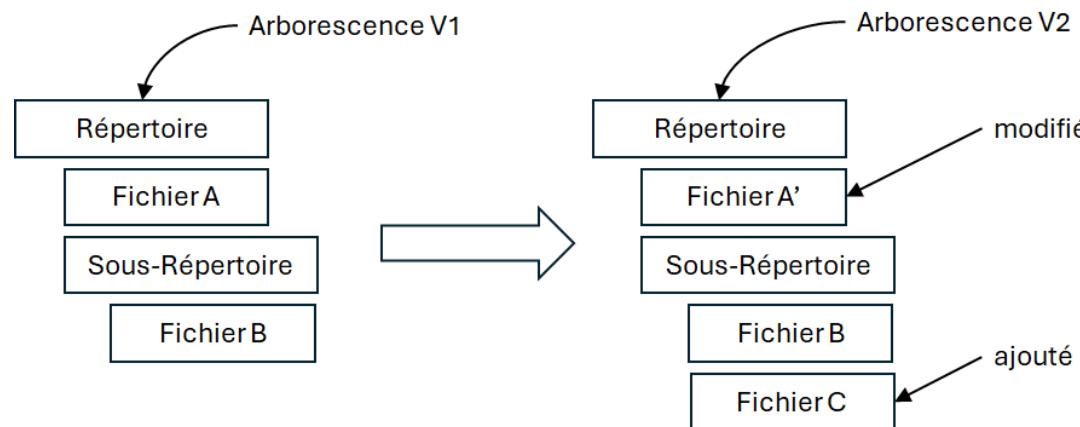
Gestion de versions avec Git

- **Git est un DVCS (Distributed Version Control System)**

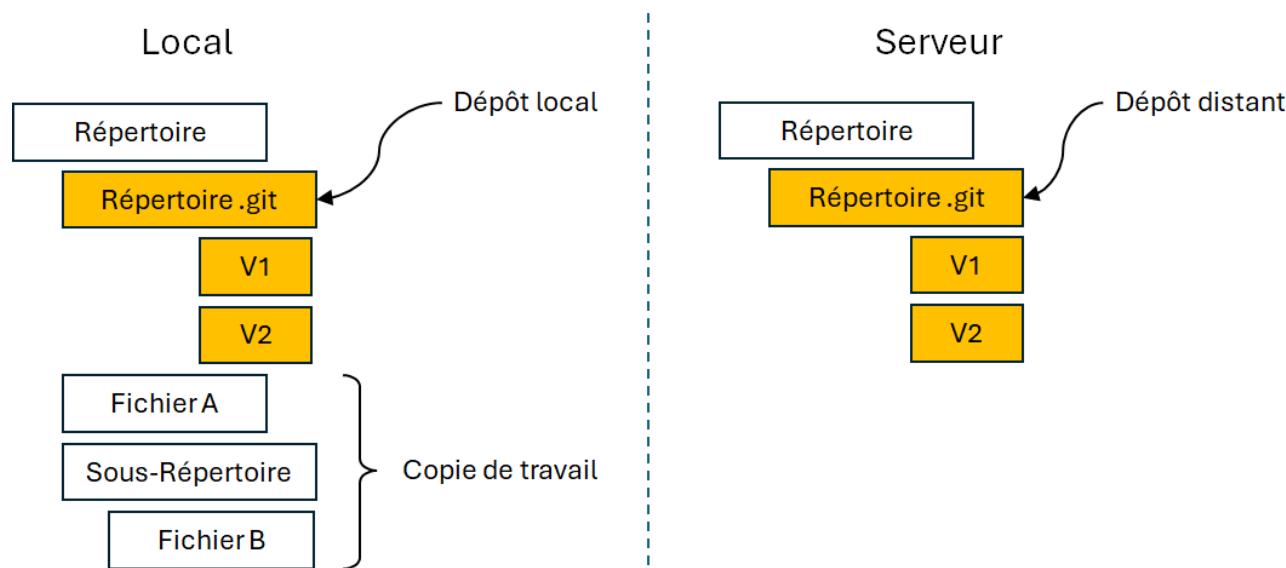
- Il sert à gérer des **versions** (d'arborescences) de fichiers
- Il est **distribué** (contrairement à la plupart des VCS)
 - Ce qui signifie que sa conception originale n'implique **pas** un serveur unique et central
 - Chaque copie locale contient une copie de **tout l'historique** et se synchronise

- **Il peut s'utiliser :**

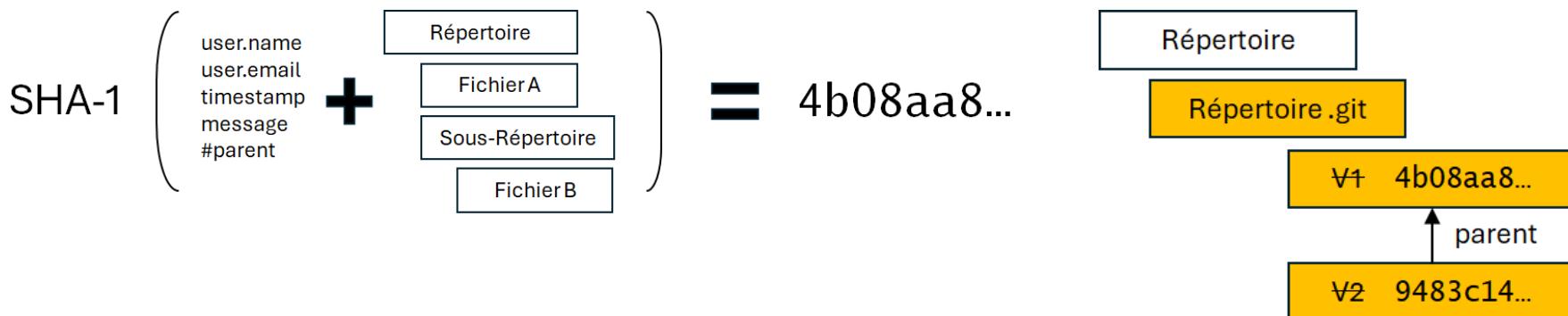
- En **local**, de manière individuelle et déconnectée
- A **plusieurs**, chacun se resynchronisant quand il le souhaite
- Ou avec un **serveur** hébergeant le **référentiel** officiel (plus naturel en entreprise)



- **Git utilise la notion de référentiel (repository) ou dépôt**
 - En **local** : matérialisé par un répertoire **.git** dans le dossier suivi racine
 - À **distance** : sur le serveur
 - Il n'y a pas de différence notable entre dépôt local et distant
 - Mais le serveur (GitHub, GitLab...) amènent des fonctionnalités complémentaires
- **Le dépôt conserve l'historique des révisions de l'arborescence**
 - L'arborescence courante est appelée **copie de travail (working copy)**



- **Commit = un instantané de l'arborescence**
 - Il s'agit d'un objet binaire interne **immutable** (une photo de l'arborescence)
- **Il contient notamment :**
 - L'**auteur** (nom, adresse mail), le **timestamp**, le **message** (description textuelle)
 - Le **contenu** de l'arborescence (de manière optimisée)
 - L'identifiant du commit **parent** (parfois plusieurs)
- **Ces informations servent à calculer son identifiant (hash)**
 - hash = 40 digits hexadécimaux identifiant de manière quasi unique le commit



- **Le hash ne dépend pas du serveur**
 - Il ne dépend que du contenu et du contexte (auteur, timestamp...)
 - Pas de numéro à incrémenter, pas de séquence
- **Unicité globale**
 - Tout commit de n'importe quel dépôt a ainsi un id unique
 - Que ce soit un dépôt public, privée ou même local
- **Identifiant abrégé**
 - Quelques caractères suffisent pour identifier un commit dans un dépôt
 - La plupart des commandes affichent et acceptent cet identifiant abrégé
- **Exemple**
 - `git show 4b08aa8` ⇐ montre les informations du commit
 - `git show 4b08aa87591e53336542bbb08766f62c3b6d360d` ⇐ équivalent

- **Le message de commit devrait respecter les conventions :**
 - Une ligne courte décrivant l'intention du commit (ce que l'on cherche à faire)
 - Elle devrait commencer par un **verbe à l'impératif** ou un **nominatif**
 - Ajoute, Corrige, Ajout, Correction
 - **Une ligne vide** sépare ensuite d'autres lignes (détails justifiant ce commit)
 - Un **footer optionnel** peut préciser des références (ids de commit, tickets, ...)

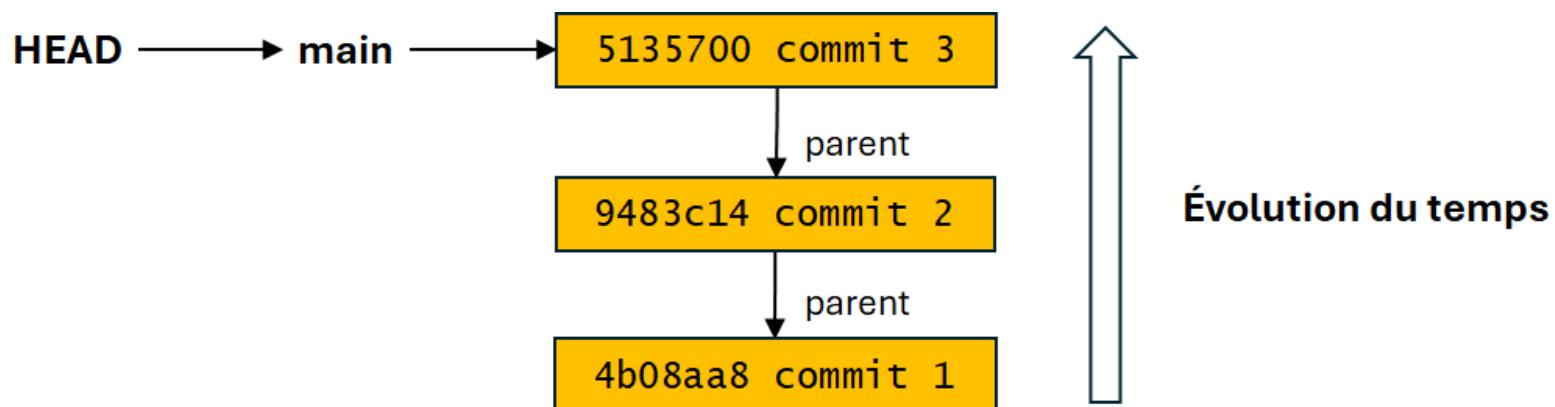
Corrige le bug n°87456 - modification du mot de passe impossible

Ajoute un lien à la page de connexion permettant de modifier son mot de passe.
Ce lien n'est actif que si l'utilisateur est connecté

Dépend du commit 7e56df3

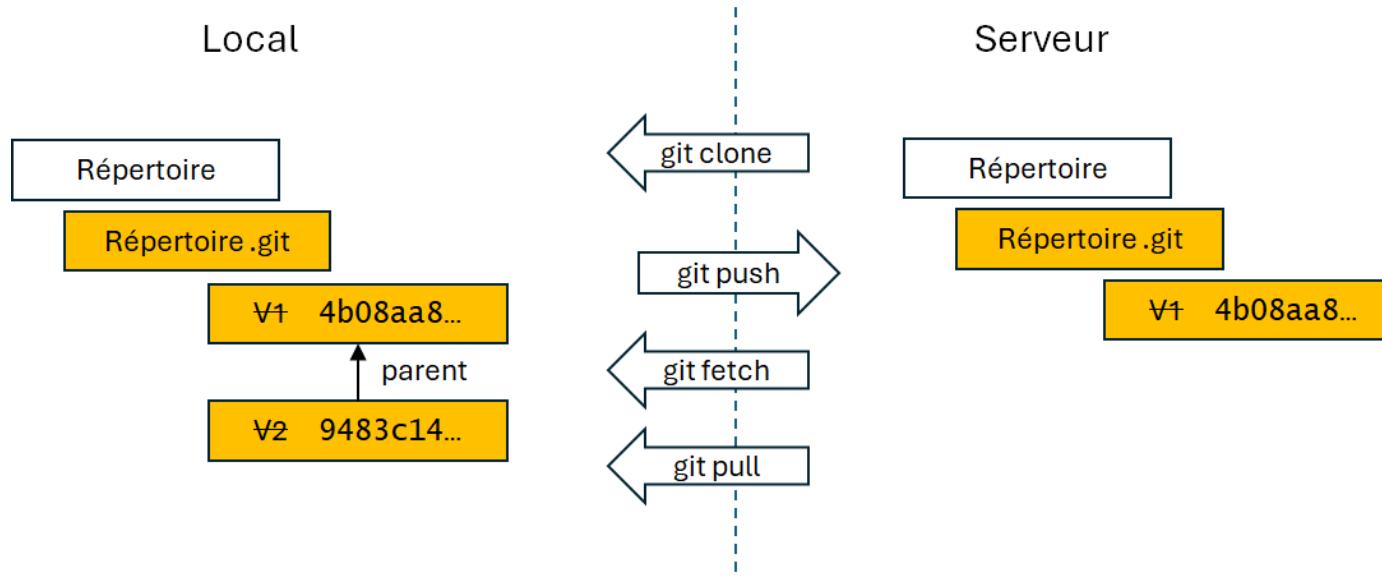
- **Une spécification plus formelle : conventional commit**
 - Cette spécification vise à formaliser encore plus pour permettre d'automatiser
 - Voir le site <https://conventionalcommits.org> (il y a une version française)
 - Elle n'est pas largement adoptée...

- L'historique des commits est une chaîne de commits
 - Chaque commit référence son commit **parent** (parfois plusieurs)
 - La **branche** courante (**main**) garde une référence vers le dernier commit
 - La référence symbolique **HEAD** pointe vers cette branche courante



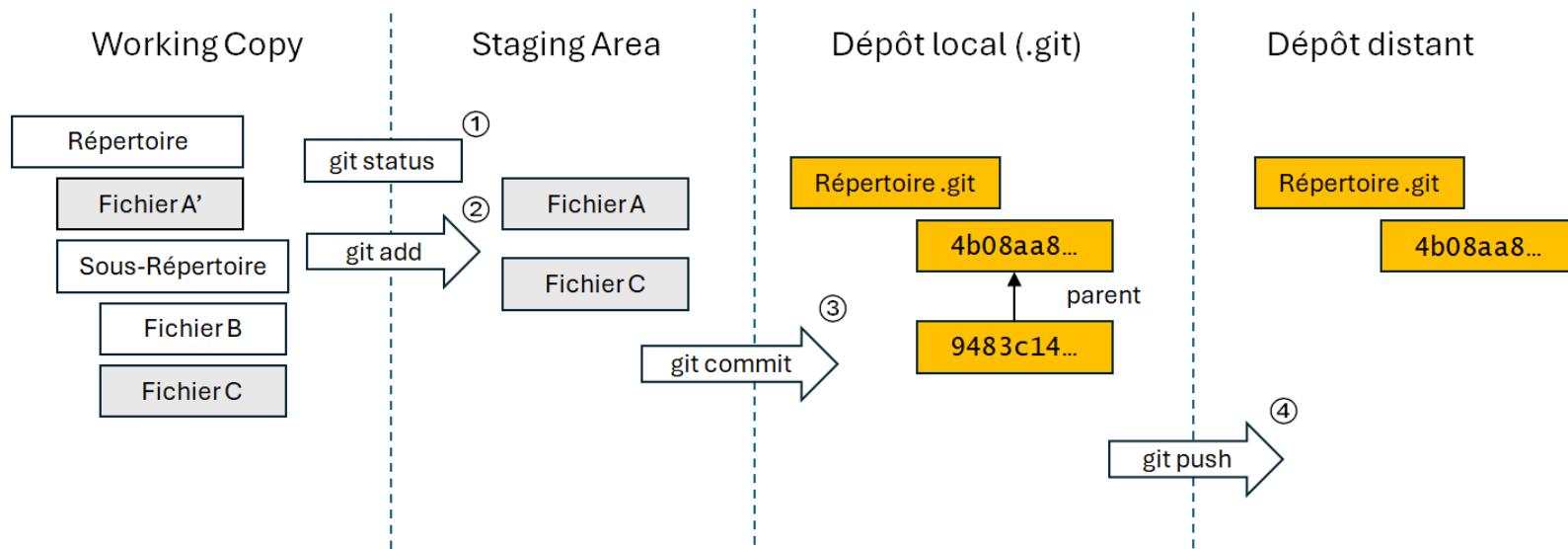
- La commande **git log** affiche cette liste chainée
 - Elle part du commit courant (**HEAD**) et remonte la chaîne pour afficher

- **La synchronisation des dépôts n'est pas automatique**
 - Plusieurs commandes pour déclencher explicitement une synchronisation



- **git clone <url_du_dépot_distant>** : pour copier un dépôt distant en local
- **git push** : pousse les commits **vers** le dépôt distant
- **git fetch** : tire les commits **depuis** le dépôt distant
- **git pull** : fait un **git fetch** et resynchronise la copie de travail locale

- **Créer un commit fait intervenir la staging area (ou index en français)**
 - C'est la zone de préparation du prochain commit
 - Elle permet de sélectionner précisément les modifications à inclure



- **git status** permet de connaître l'état actuel ①
- **git add <file|dir>** permet d'ajouter dans la staging area ②
- **git commit** fabrique un commit avec le contenu de la staging area ③
- **git push** pousse le commit vers le dépôt distant ④

- **Il faut être identifié (auteur) pour créer un commit**
 - Requiert de configurer les propriétés **user.name** et **user.email**
 - Ces informations se configurent avec la commande **git config**
- **La configuration peut se faire à trois niveaux :**
 - Local à un dépôt avec **--local** (option par défaut)
 - Global à un utilisateur avec l'option **--global**
 - Global au système avec l'option **--system**

```
*** Please tell me who you are.
```

Run

```
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"
```

to set your account's default identity.

Omit --global to set the identity only in this repository.

- **Plusieurs possibilités pour se connecter à un dépôt distant**
 - **HTTPS ou SSH** (élimine la saisie de mot de passe)
- **La connexion SSH nécessite une paire de clés privée/publique**

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
...
Your identification has been saved in /home/ubuntu/.ssh/id_rsa
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub
...
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2E= ubuntu@ip-172-31-10-41
```

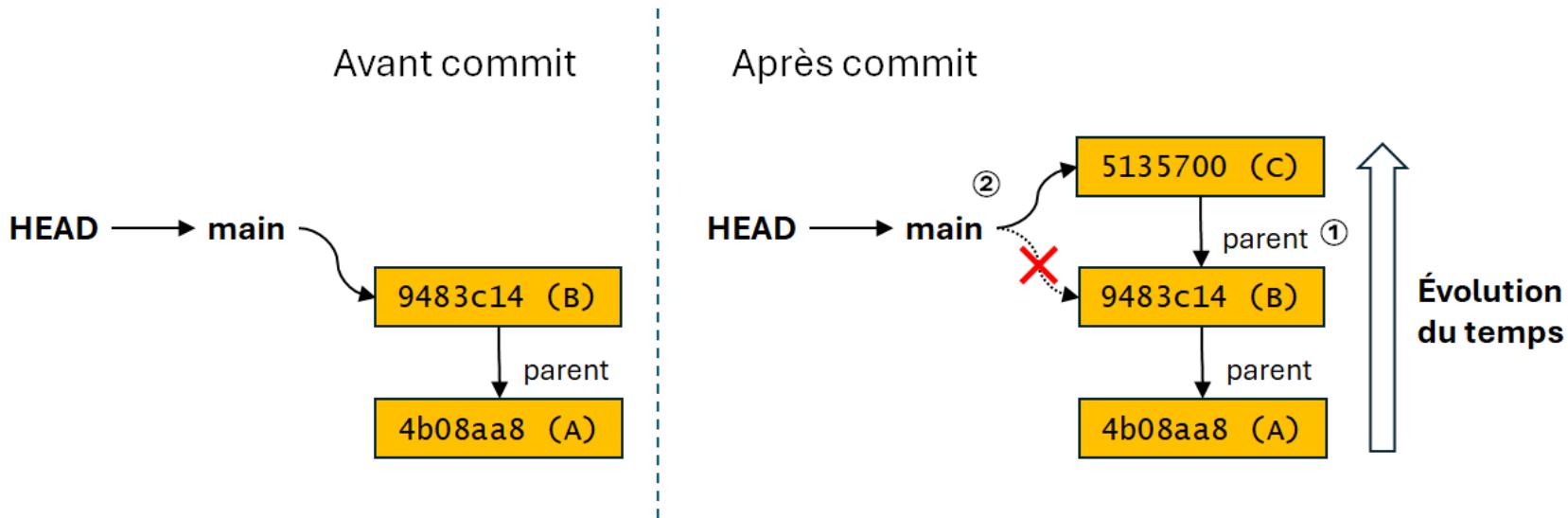
- **La clé publique doit être déclarée dans le compte GitHub**
 - Sur l'avatar > **Settings** > **SSH and GPG keys** > **New SSH key**

TP 1

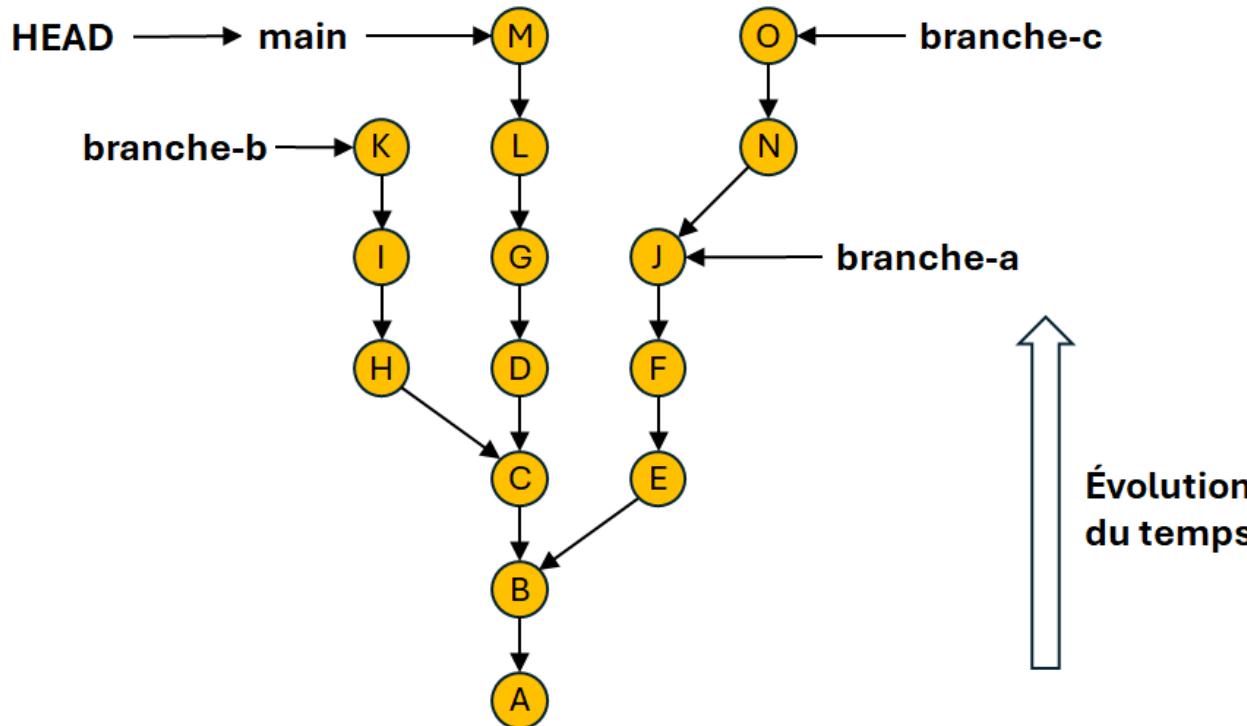
- **Premier résumé des commandes essentielles**

- Commandes d'initialisation et diverses :
 - **git help** : afficher l'aide (**git help <commande>** pour l'aide sur une commande particulière)
 - **git clone <url>** : copier un dépôt distant en local
 - **git config** : gérer la configuration, en particulier **user.name**, **user.email** et **core.editor**
- Commandes avec le dépôt local :
 - **git status** : afficher l'état local (working copy et staging area)
 - **git log** : afficher l'historique (**--oneline**, **--all...**)
 - **git show** : afficher les détails d'un commit
 - **git diff** : afficher les différences entre deux commits / working copy
 - **git add <file|dir>** : ajouter un fichier dans la staging area (cf. **restore**, **rm**, **mv...**)
 - **git commit** : créer un commit (demande un message de commit ou **-m <message>**)
 - **git checkout** : restaurer la copie de travail à un état particulier (commit, branche...)
- Commandes avec le dépôt distant :
 - **git remote** : afficher les informations des dépôts distants (**-v**)
 - **git push** : pousser les commits locaux vers le dépôt distant
 - **git fetch** : tirer les commits du dépôt distant
 - **git pull** : tirer les commits et resynchroniser la working copy

- **Une branche est techniquement une référence vers un commit**
 - Et donc vers une chaîne de commits (lien parent)
 - D'un point de vue fonctionnel, elle représente un **fil d'évolutions**
- **La branche active est gérée spécialement lors d'un commit**
 - La staging area définit le contenu du nouveau commit
 - **HEAD** indique le commit parent ①
 - La branche courante est mise à jour pour référencer ce nouveau commit ②



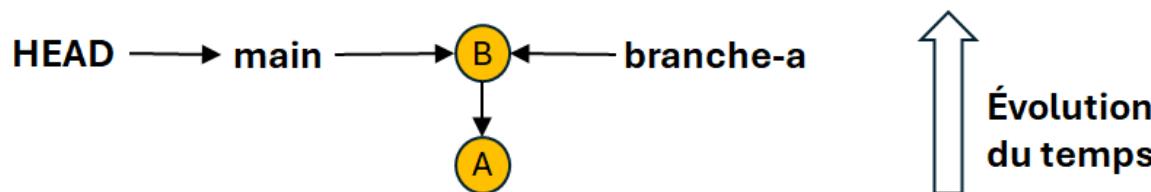
- **Les branches de Git sont très légères et rapides à créer**
 - Une branche est un simple fichier contenant un hash
- **Elles servent à divers usages et leur durée de vie est très variable**
 - En général, elles servent à définir des flux d'évolutions séparés



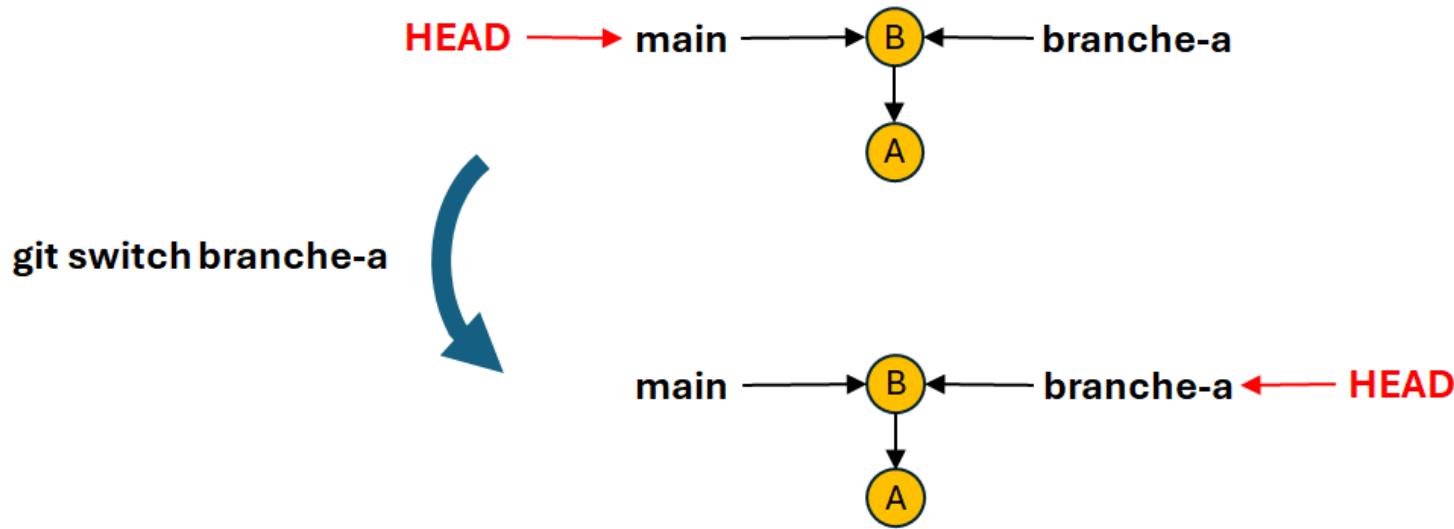
- **La commande git branch pour manipuler les branches**
 - **git branch** : afficher les branches locales et la branche active (avec *)
 - **git branch --all** : afficher aussi les branches distantes (remote)

```
$ git branch --all
* main
  branche-a
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

- **git branch <nom>** : créer une nouvelle branche à partir de **HEAD**
 - Attention aux caractères spéciaux et à la casse : une branche est un fichier et doit être portable

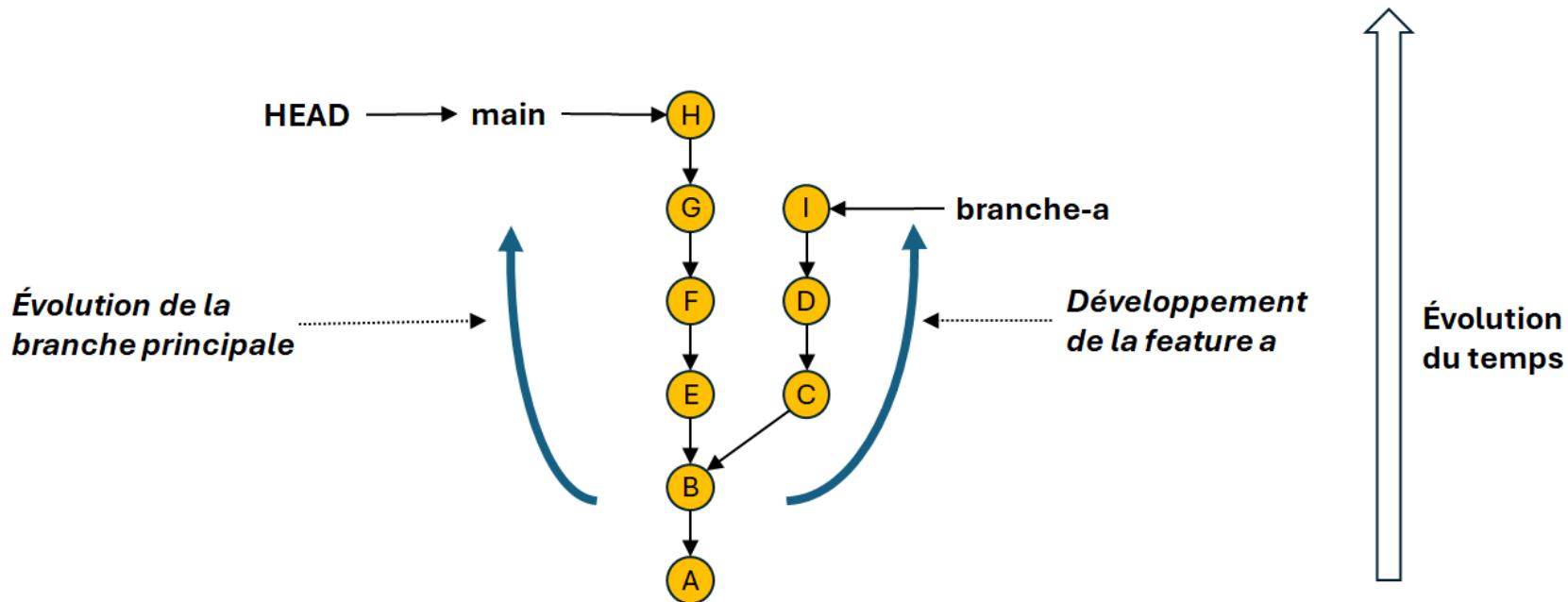


- **Changer de branche modifie la référence HEAD**
 - **git switch <branche>** : la branche devient la branche **active**
 - La commande **git checkout <branche>** est équivalente (mais sert aussi à restaurer un fichier)

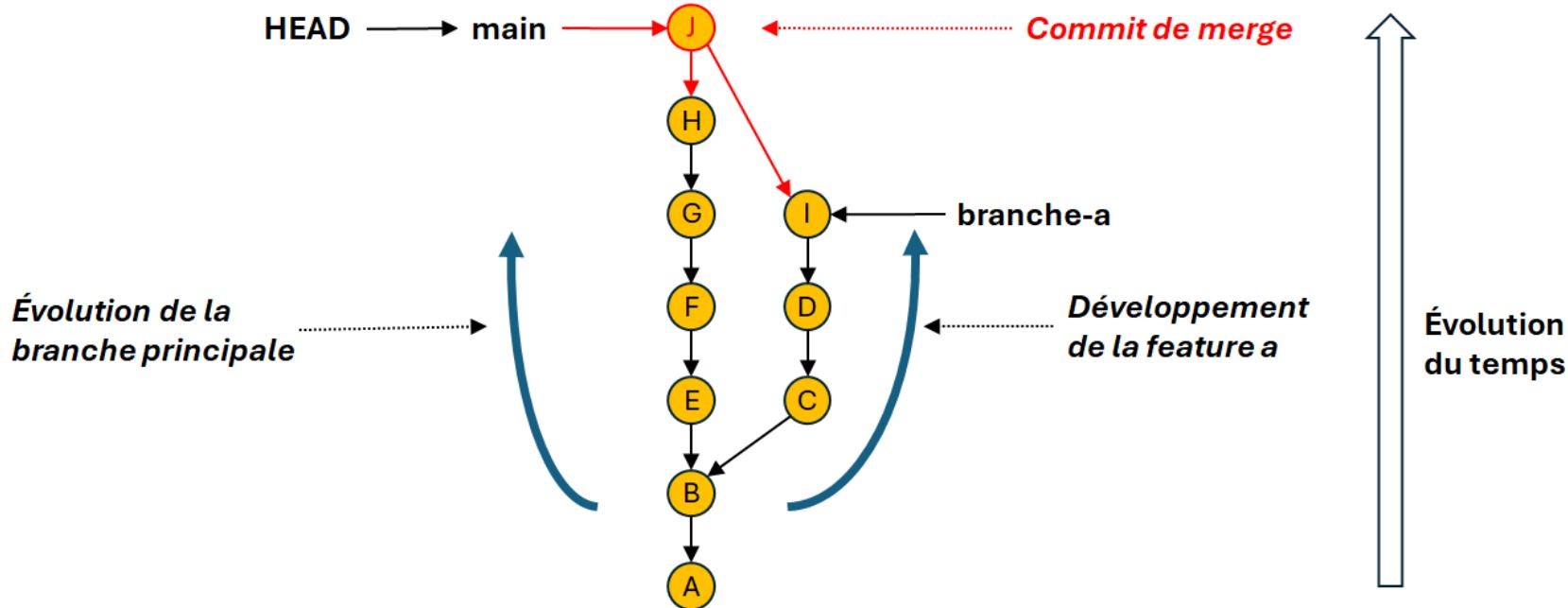


- **Remarque : la working copy est actualisée**
 - Pour refléter l'état des fichiers du nouveau commit pointé
 - Ici, les fichiers sont les mêmes mais on peut switcher de n'importe où
 - Attention en cas de modification en cours sur la working copy

- **Les branches permettent de séparer des flux d'évolutions**
 - Typiquement, des flux de développement (feature, patch, exploration...)
 - Les flux doivent parfois se rejoindre (livraison, fusion de travaux parallèles...)
- **git merge <branche> = fusionner la branche vers la branche active**
 - Correspond à livrer les modifications pour les mettre en commun

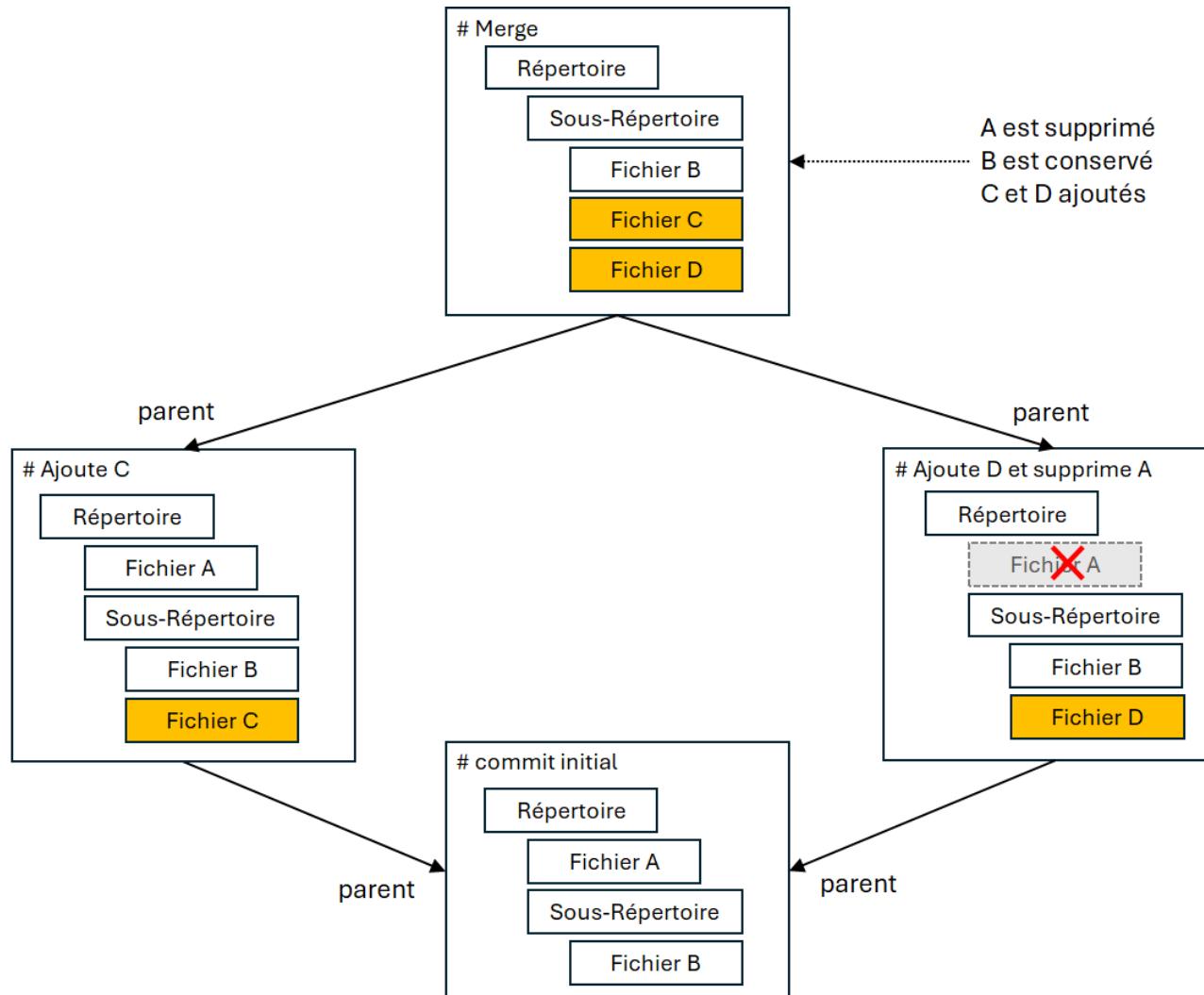


- **Le merge fabrique un commit avec plusieurs parents**
 - En général deux parents mais ça peut être plus
- **Rappel : commit = instantané d'une arborescence de fichiers**
 - Le merge fusionne ces arborescences et crée un commit avec deux parents



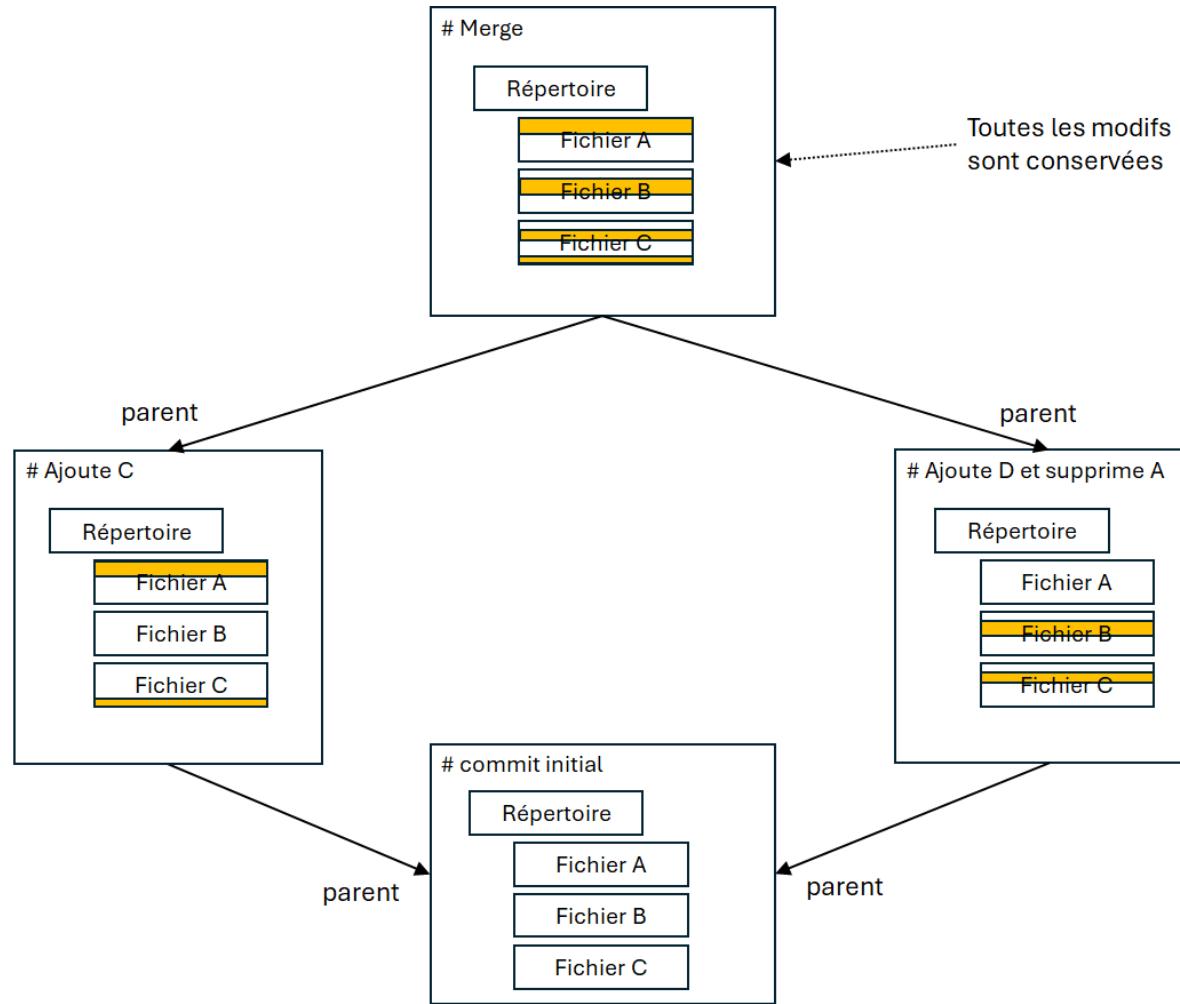
Merge d'arborescences

- Cas résolu automatiquement : ajout et suppression de fichiers



Merge des fichiers modifiés

- Cas résolu automatiquement : modifications indépendantes

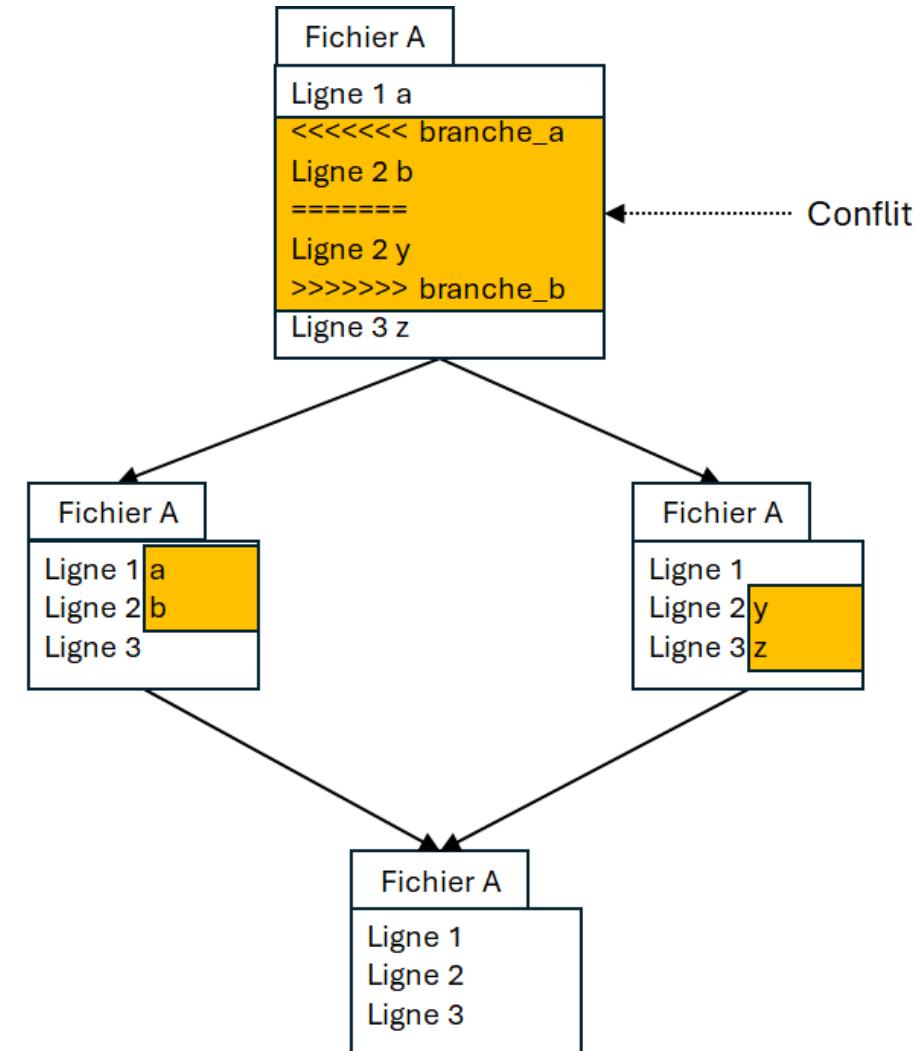


- **Cas de conflit**

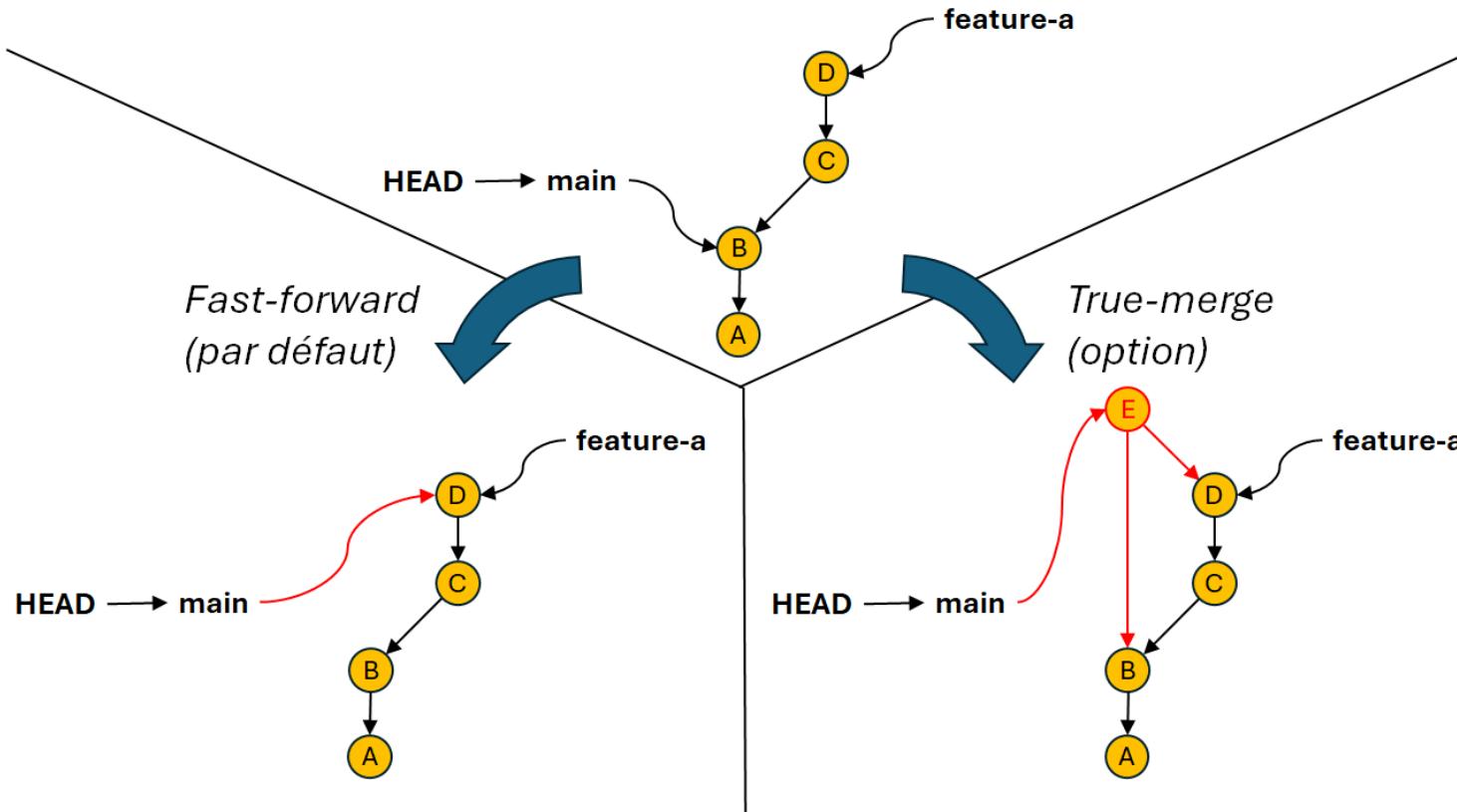
- À résoudre à la main
- Les lignes en conflit sont recopiées
 - Entre chevrons <<<< ... ===== ... >>>>
- Il faut décider comment résoudre :
 - Ligne 2 b ?
 - Ligne 2 y ?
 - Ligne 2 by ?
 - Ligne 2 yb ?
 - Ligne 2 b puis Ligne 2 y ?
 - Ligne 2 y puis Ligne 2 b ?
 - Autre chose ?
- Les lignes sans conflit sont traitées

- **Terminer la résolution**

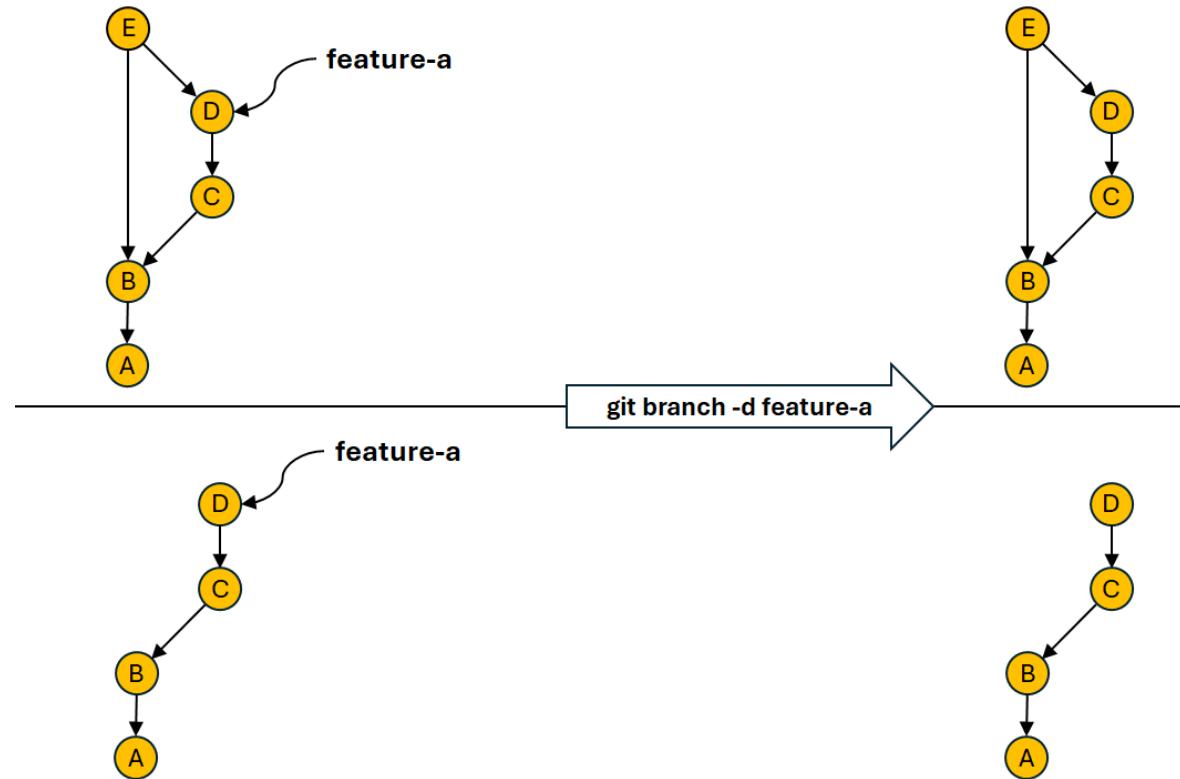
- **git add** des fichiers résolus
- **git merge --continue**



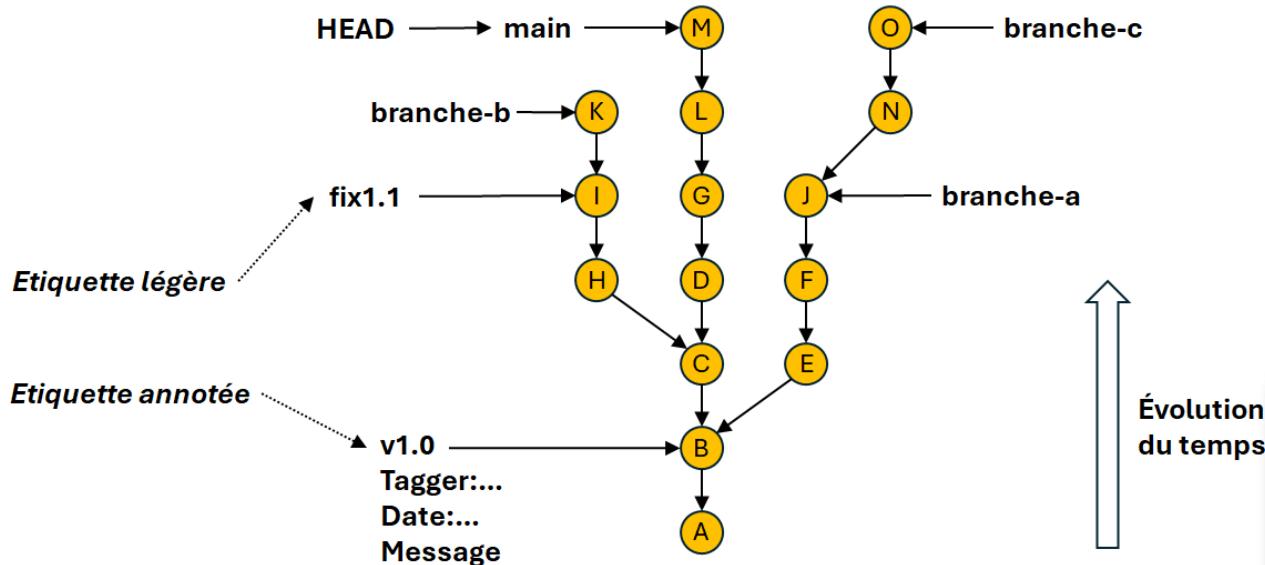
- **Le merge ne fabrique pas toujours un commit (option à choisir)**
 - Une branche **feature-a** évolue puis est mergée (`git merge feature-a`)
 - **Fast-forward** : déplace simplement la référence **main** (historique linéaire)
 - **True-merge** : force à créer un commit avec deux parents (ici commit **E**)



- **Élaguer régulièrement l'arbre en supprimant les branches inutiles**
 - `git branch -d <nom>` : supprime la branche
 - **Remarque** : ne supprime que la référence, pas les commits
 - On peut toujours les retrouver, au pire avec leur hash

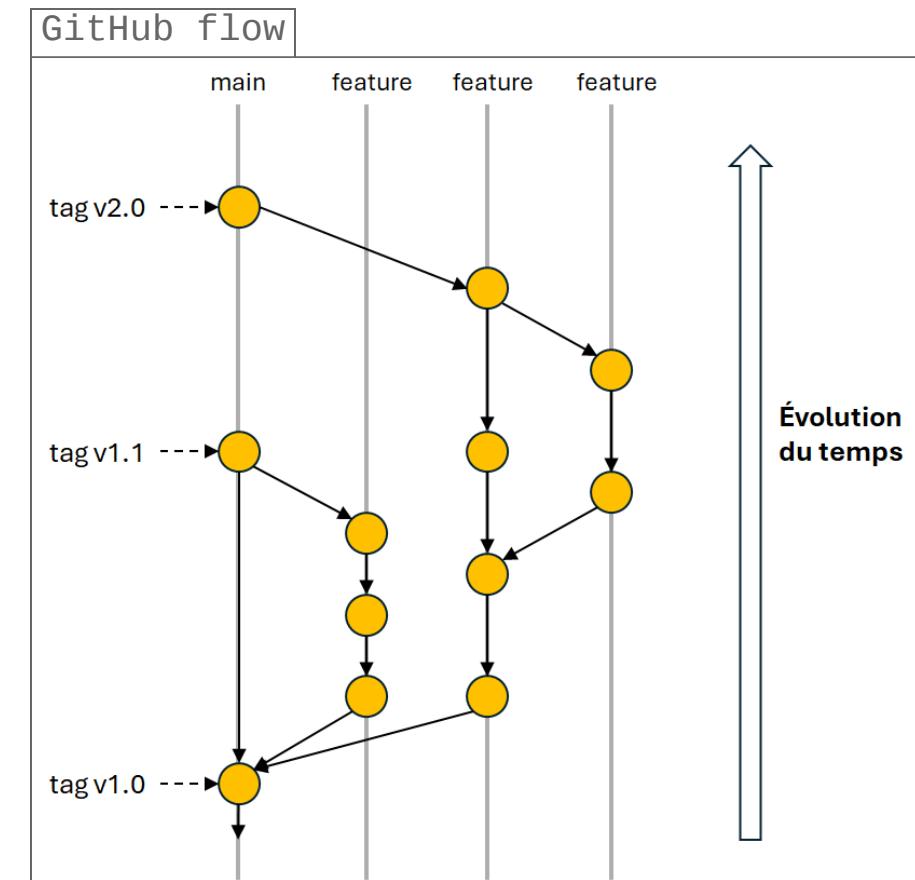


- **Repérer un commit par son hash n'est évidemment pas pratique**
 - Nous avons souvent besoin de faire référence à une révision particulière
 - Et une branche est prévue pour évoluer à chaque commit
- **Git propose la notion de tag (étiquette) sous deux formes :**
 - **Légère** : une simple référence vers un commit, comme pour une branche
 - **Annotée** : un objet (comme un commit, avec son auteur, date, message...)

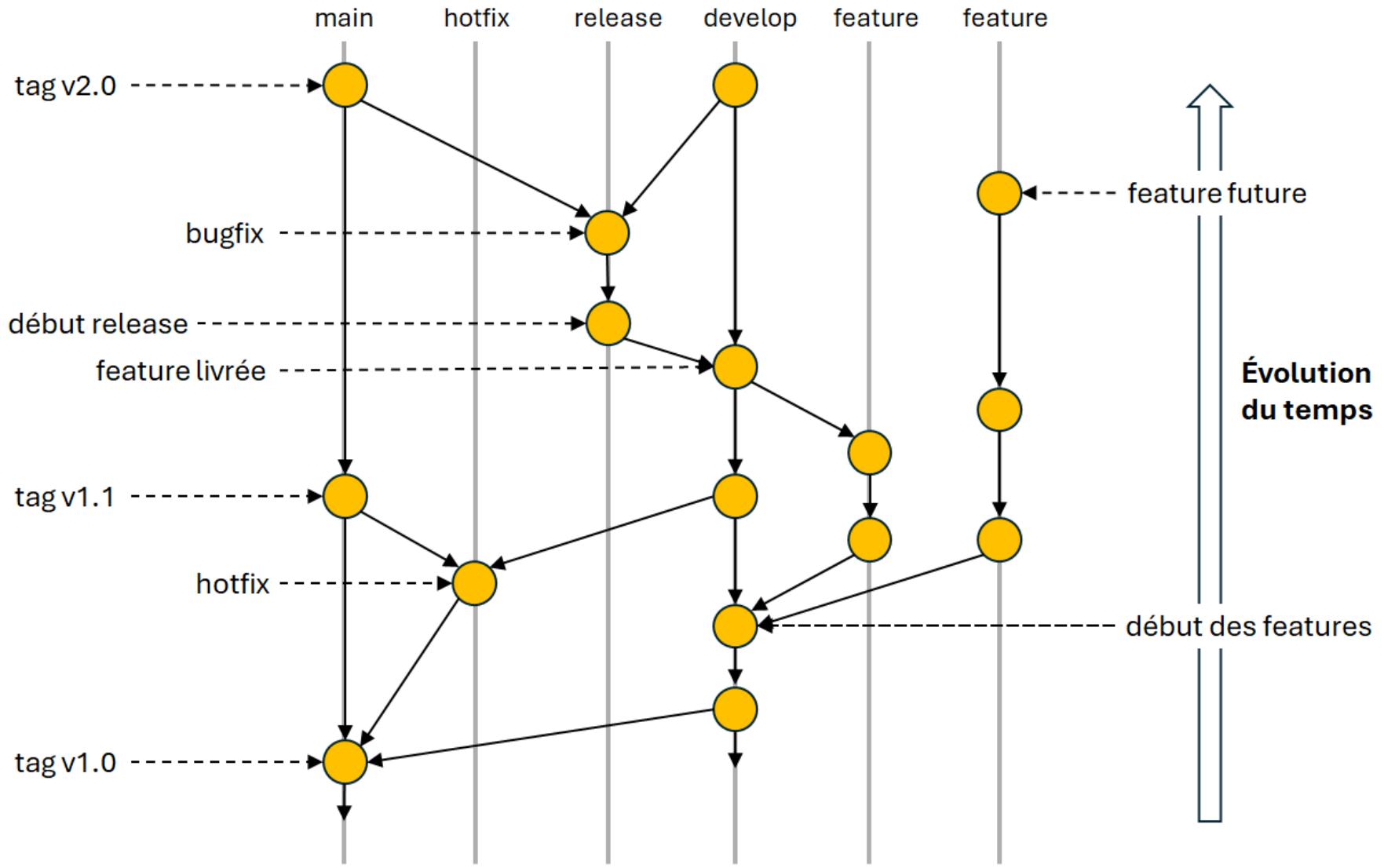


TP 2

- **Les branches/tags offrent diverses possibilités d'organisation**
 - Dépend de la taille de l'équipe, du type de projet, des préférences de chacun...
 - Liberté totale en local : c'est rapide et ça ne coute rien
- **En équipe**
 - Définition d'un **Flow** documenté
 - git flow, github flow, gitlab flow...
 - Branche = flux de développement
 - development branch
 - feature branch
 - hotfix branch
 - release branch
 - ...
 - Conventions de nommage strictes
 - Restrictions sur les branches

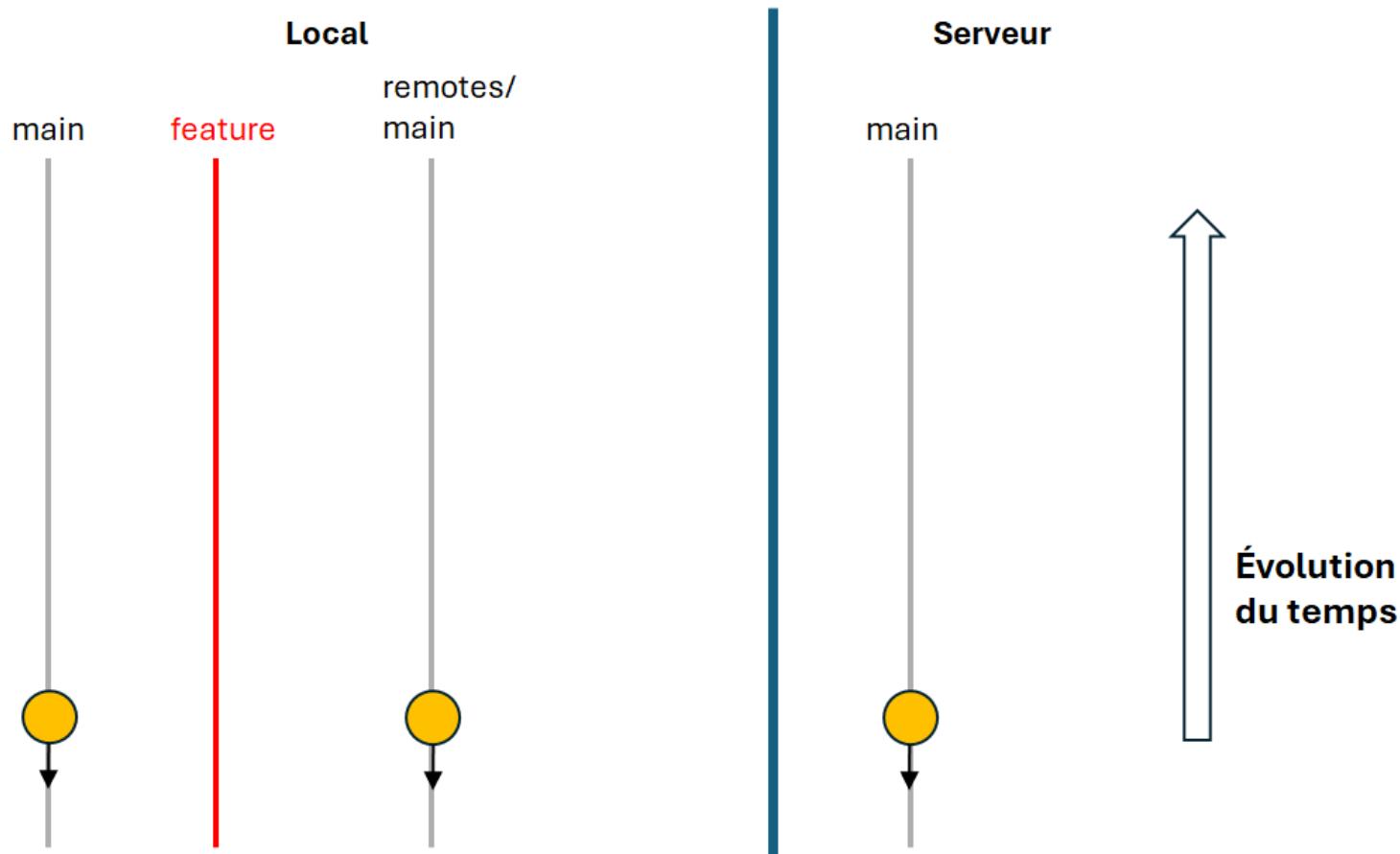


Exemple de flow complexe Git flow



- **Pull Request (PR) = demande d'intégration (merge) d'une branche**
 - Quand la branche cible est protégée (**main, release...**)
 - Garantit que tout changement a été approuvé
 - Appelée aussi Merge Request (MR) dans certains systèmes (Gitlab...)
- **Outilage pour :**
 - Revue de code
 - Discussion
 - Évolution
 - Automatisation (ticket...)
 - Conclusion (merge et clôture ou abandon)

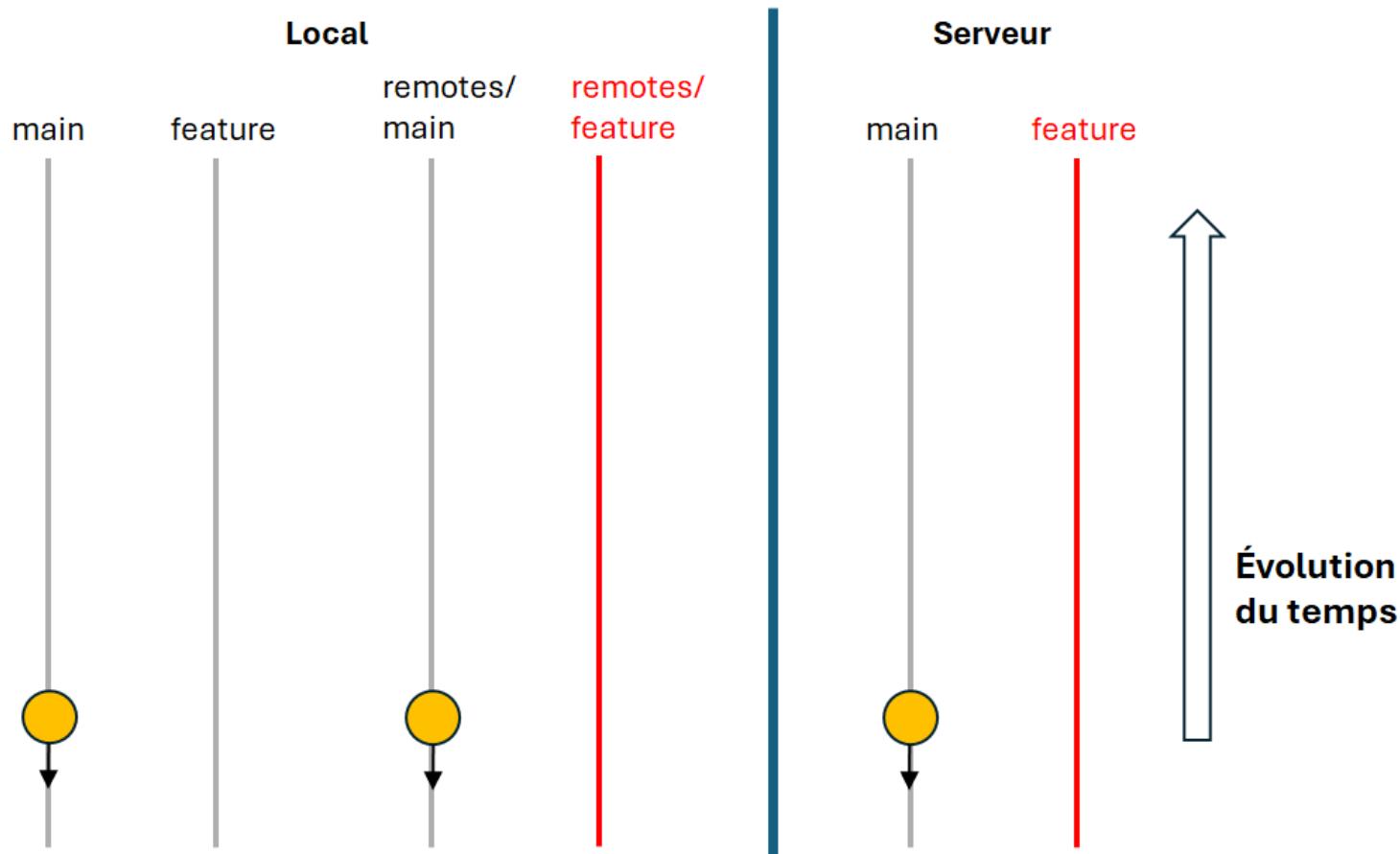
- **Création en local d'une branche pour un changement (feature, fix...)**
 - **git switch -c feature** ⇐ création de la branche puis switch dessus



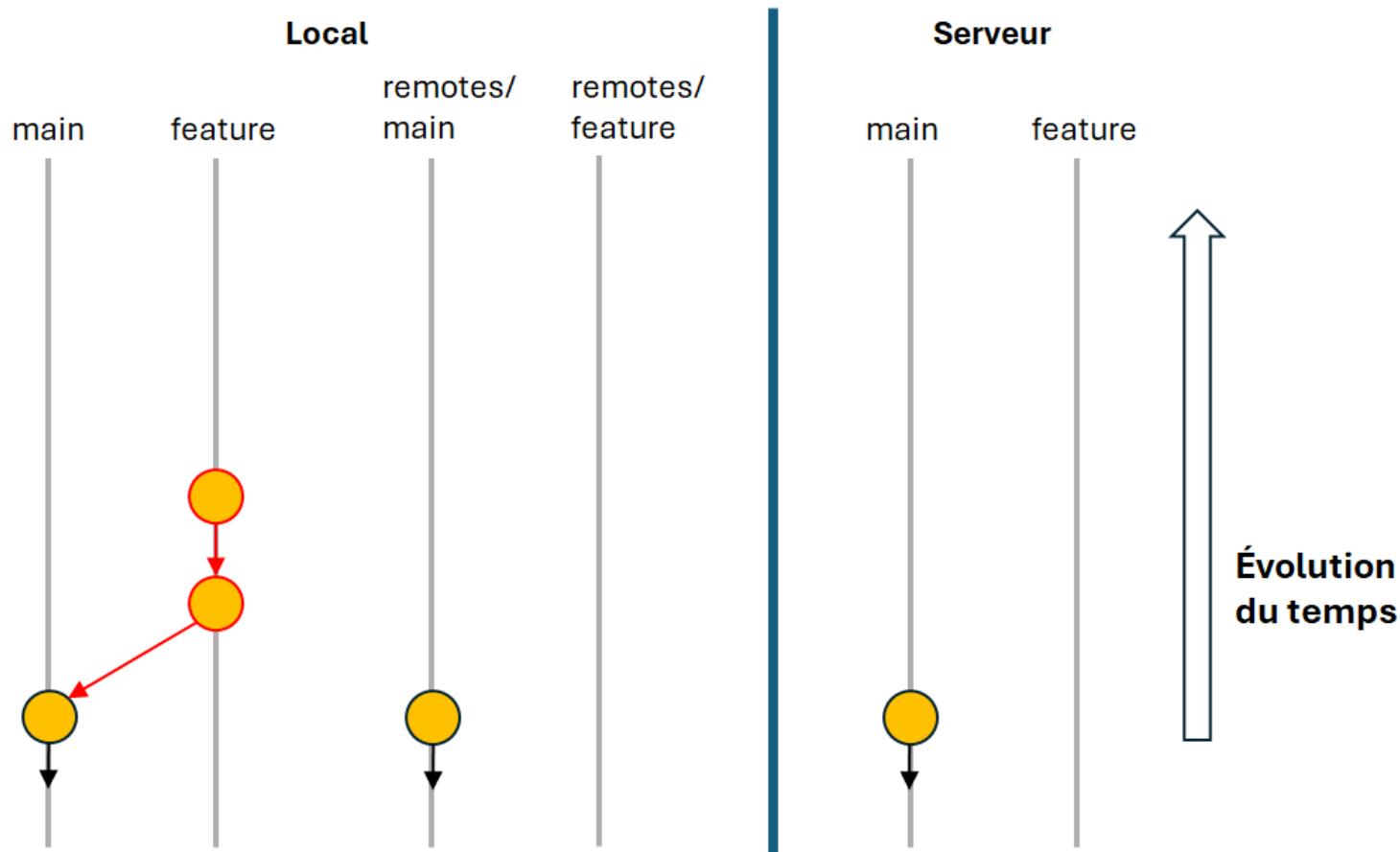
Pull Request

Etape 2

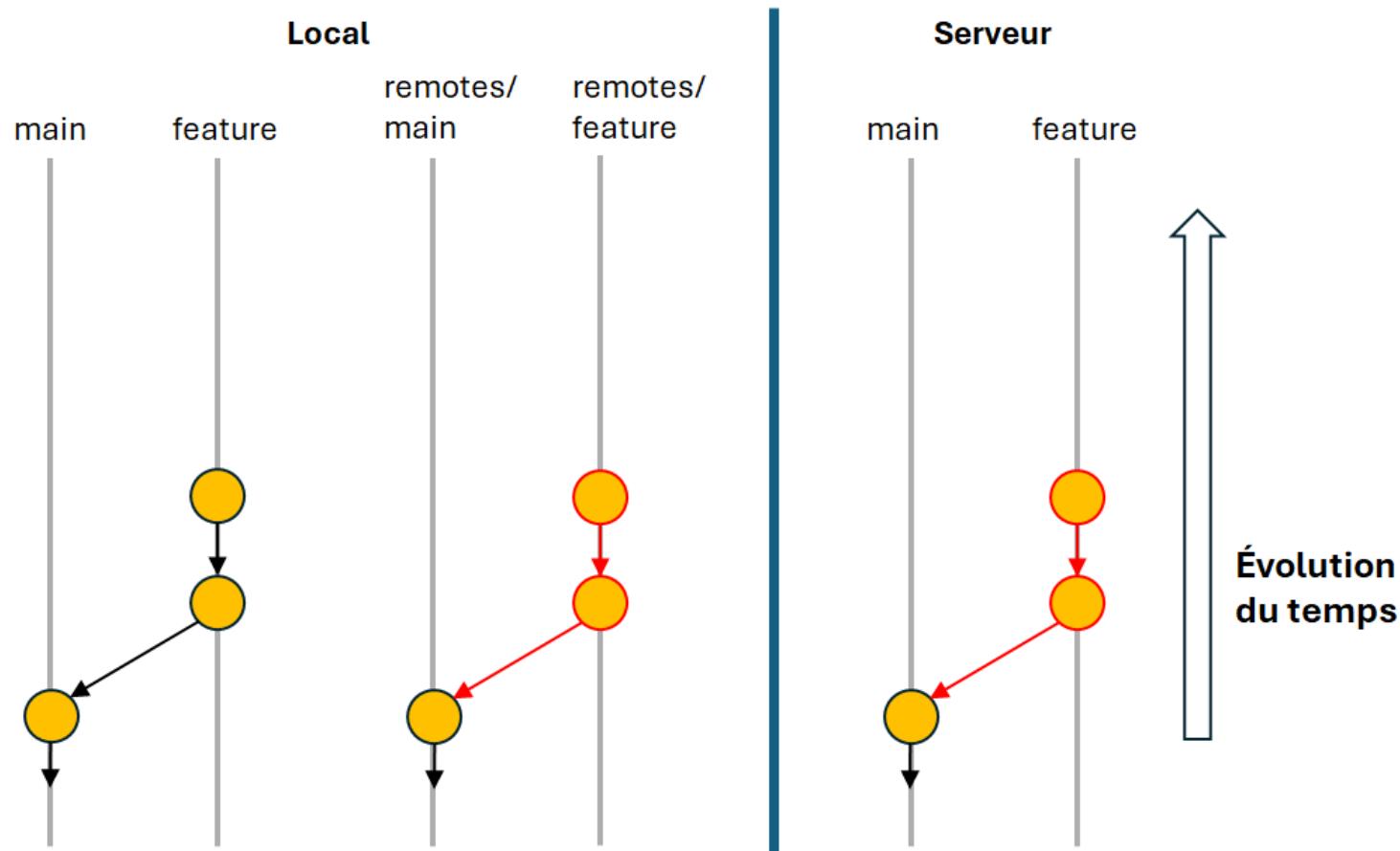
- **Pousser la branche sur le serveur**
 - Pas obligatoire mais a le mérite d'informer et de réserver le nom de branche
 - **git push --set-upstream origin feature**



- Développer les modifications en local
 - `git add ...` et `git commit ...`



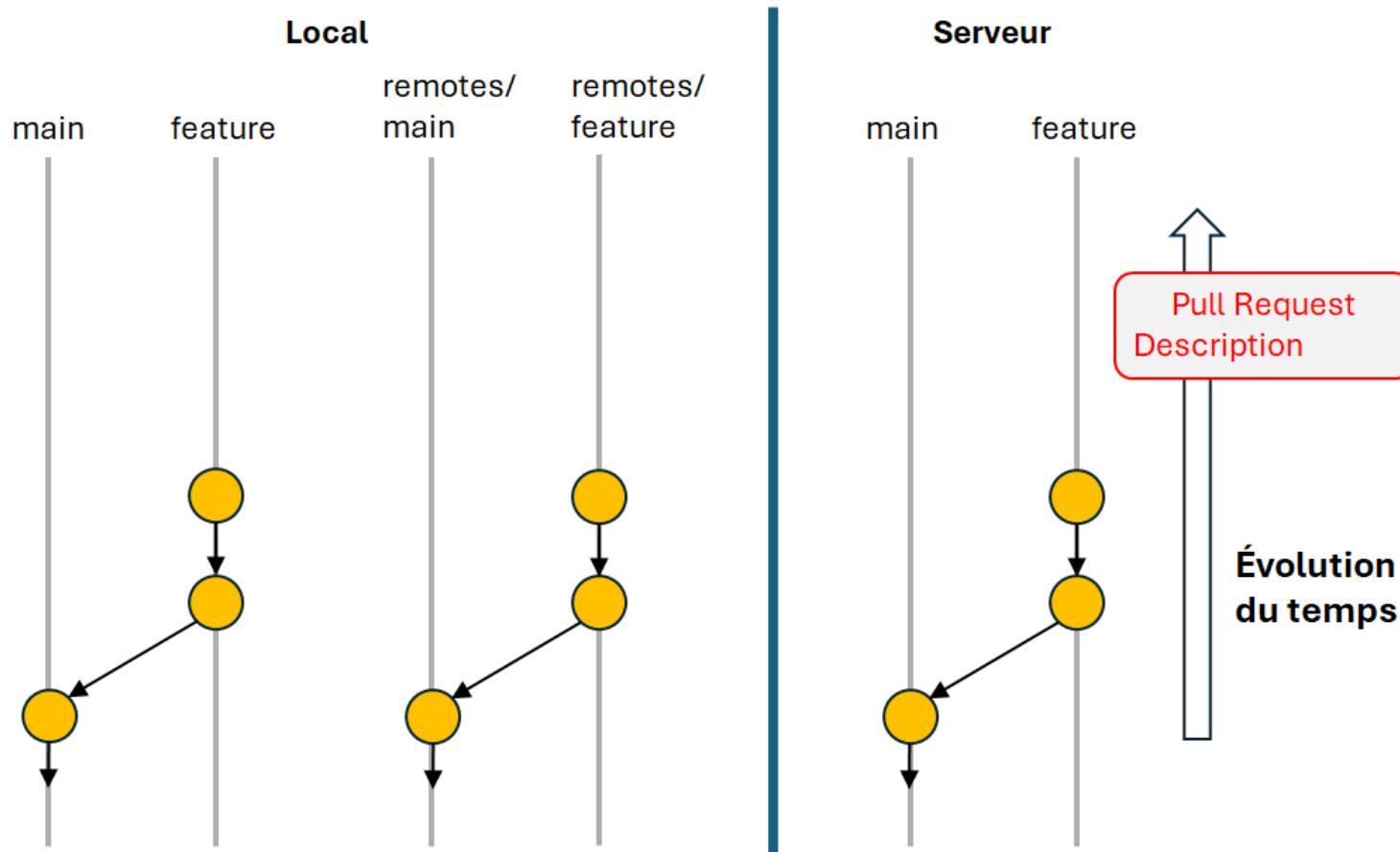
- Pousser les modifications sur le serveur (permet de collaborer)
 - `git push`



Pull Request

Etape 5

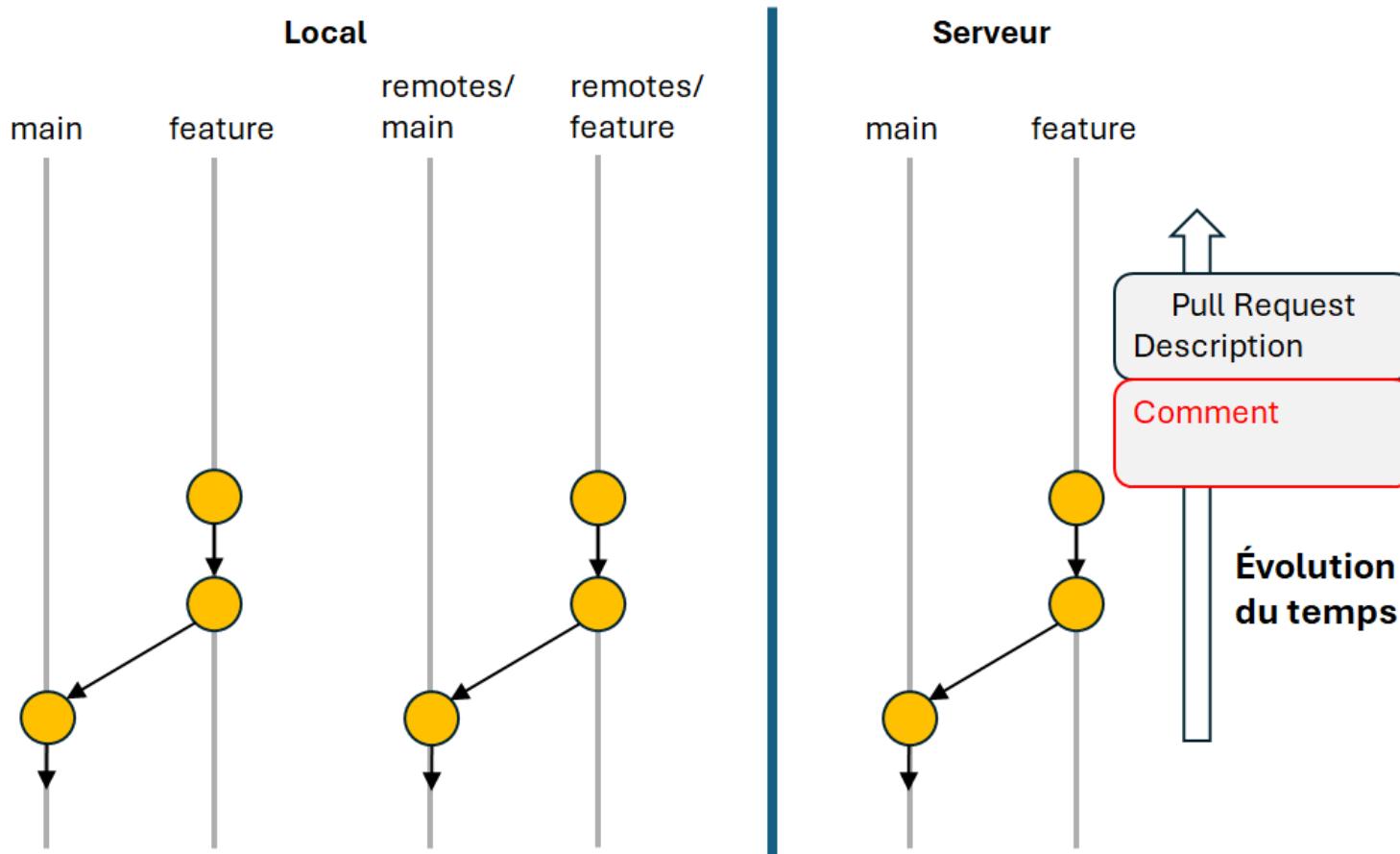
- **Créer la Pull Request dans l'interface web ou dans GitHub CLI**
 - Décrire la demande, indiquer la branche cible et source (celle à merger)



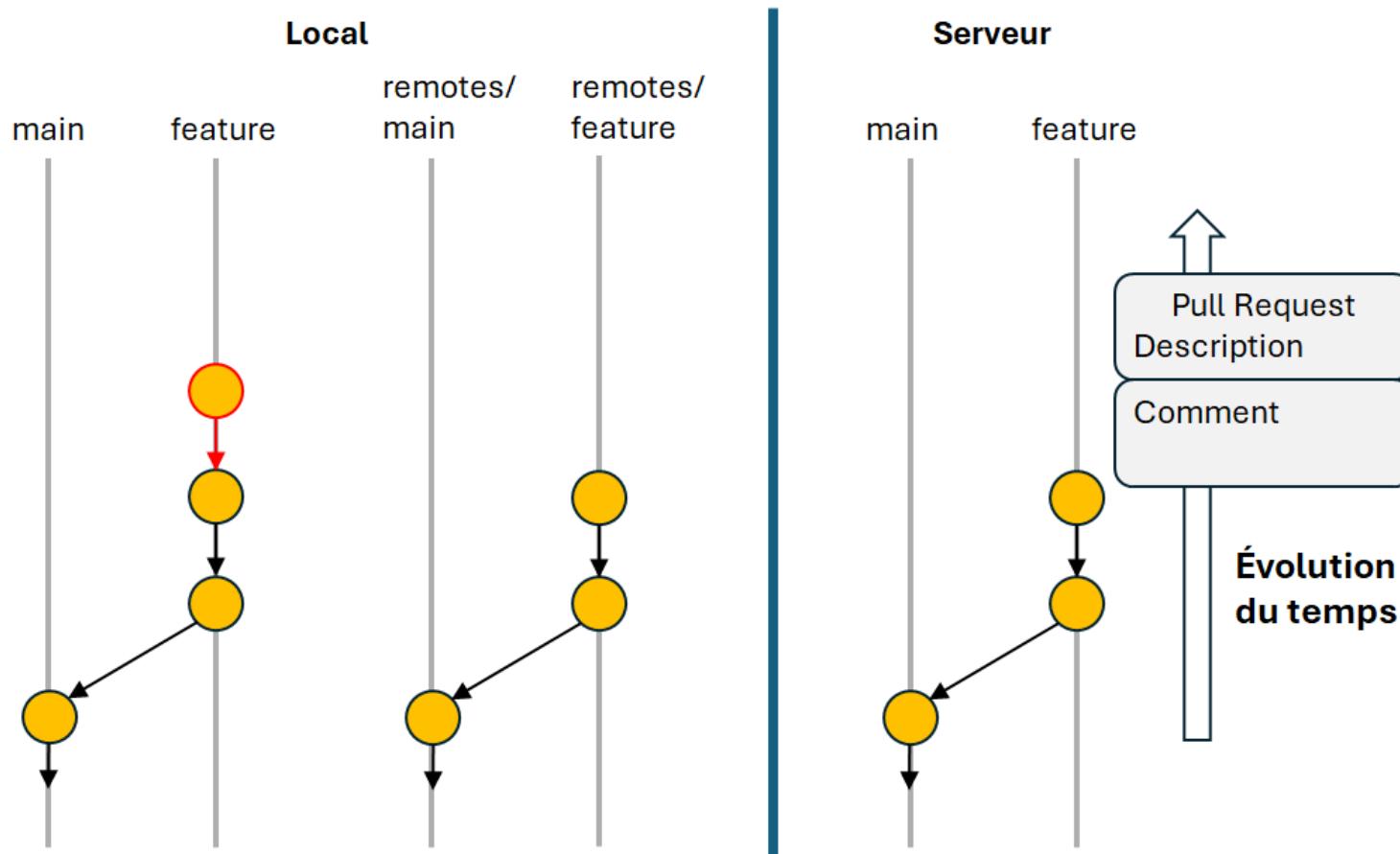
Pull Request

Etape 6

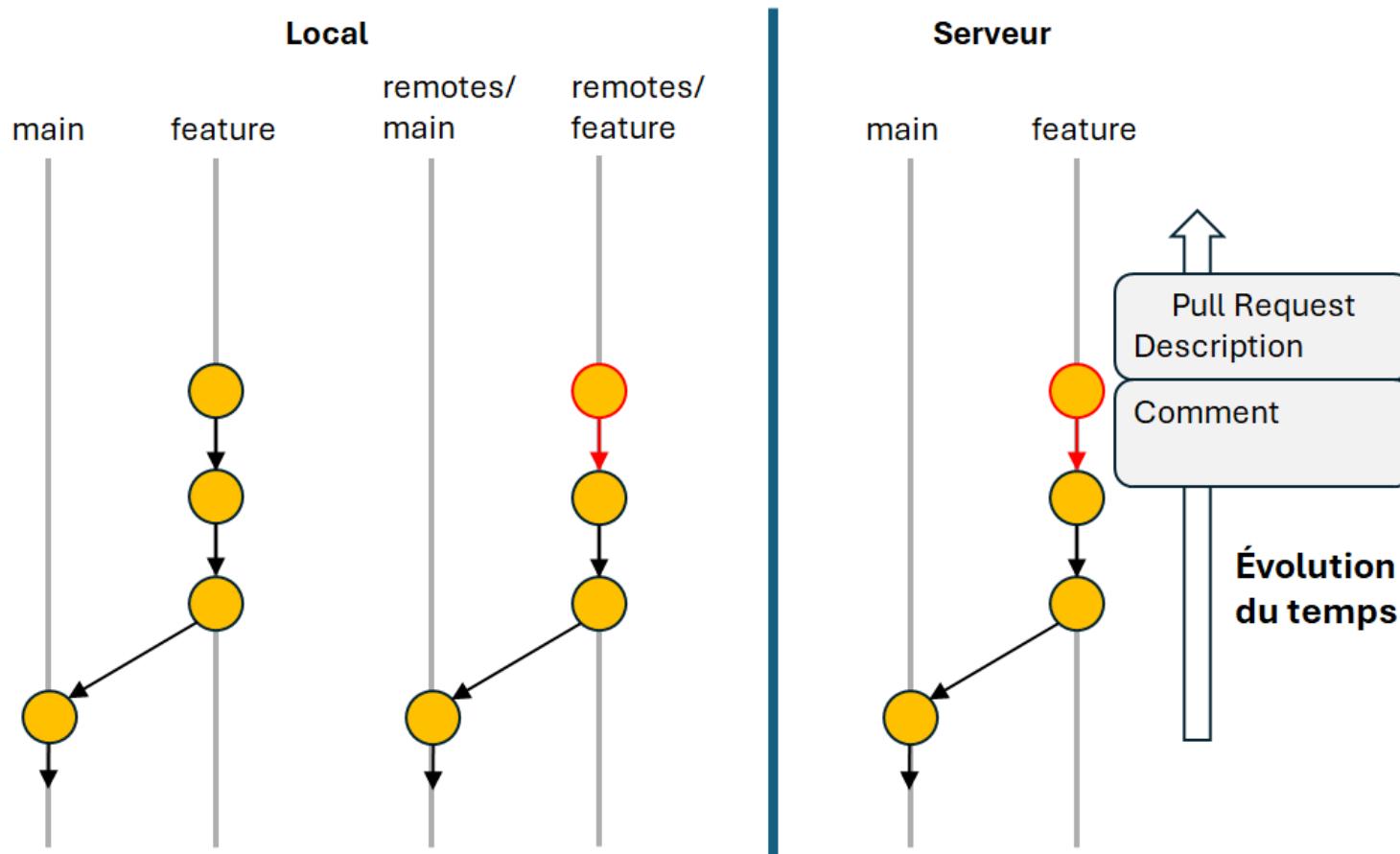
- La Pull Request peut conduire à des commentaires
 - Ou à des demandes d'améliorations de la part du reviewer (discussion)



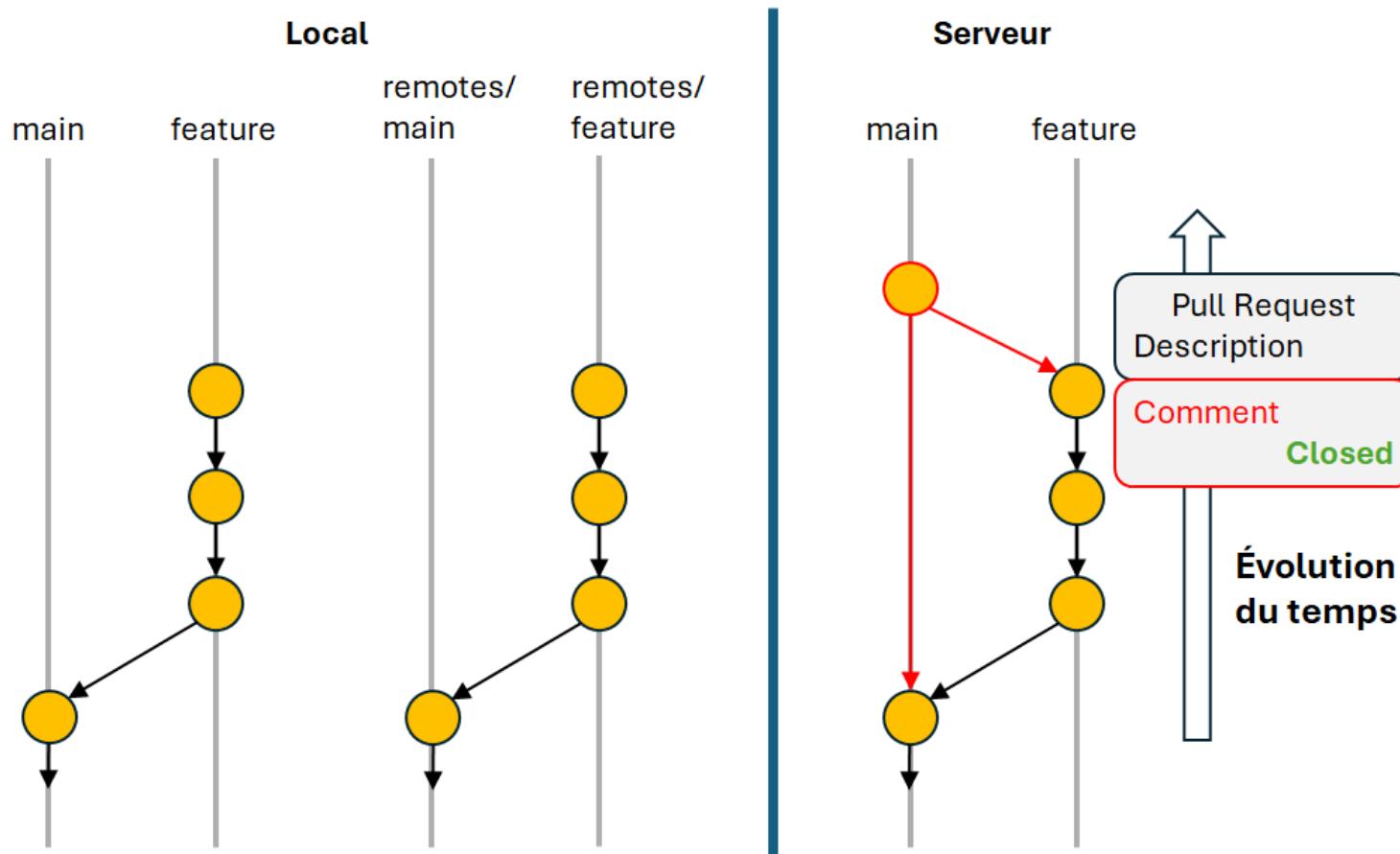
- **Prise en compte des demandes d'amélioration**
 - Génère de nouveaux commits sur la branche (`git add ...` et `git commit ...`)



- Pousser les nouvelles améliorations
 - `git push`



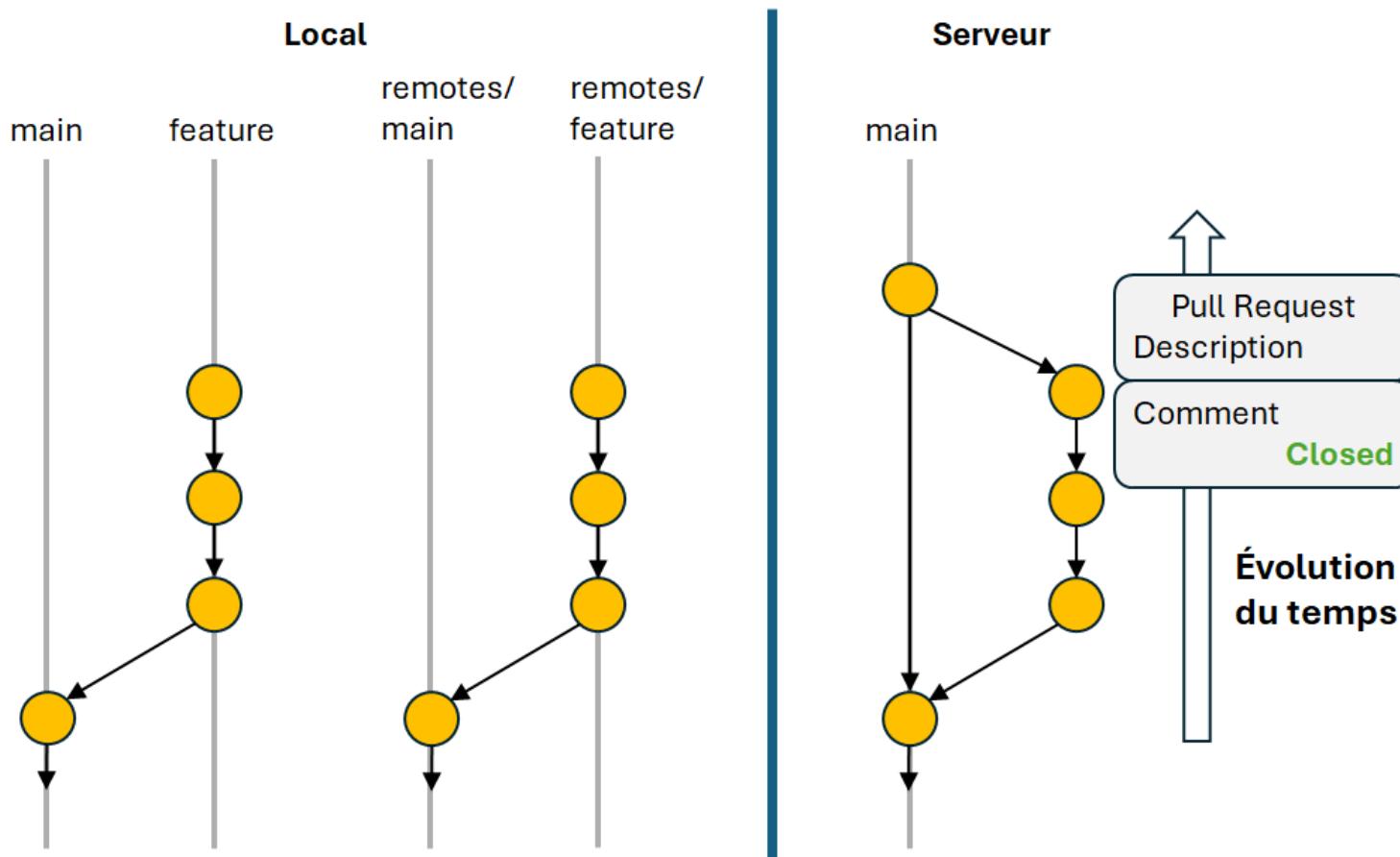
- **Acceptation et fermeture de la Pull Request**
 - Conduit à un merge de la branche (ou un abandon)
 - Plusieurs options possibles (voir plus loin **merge**, **squash** et **rebase**)



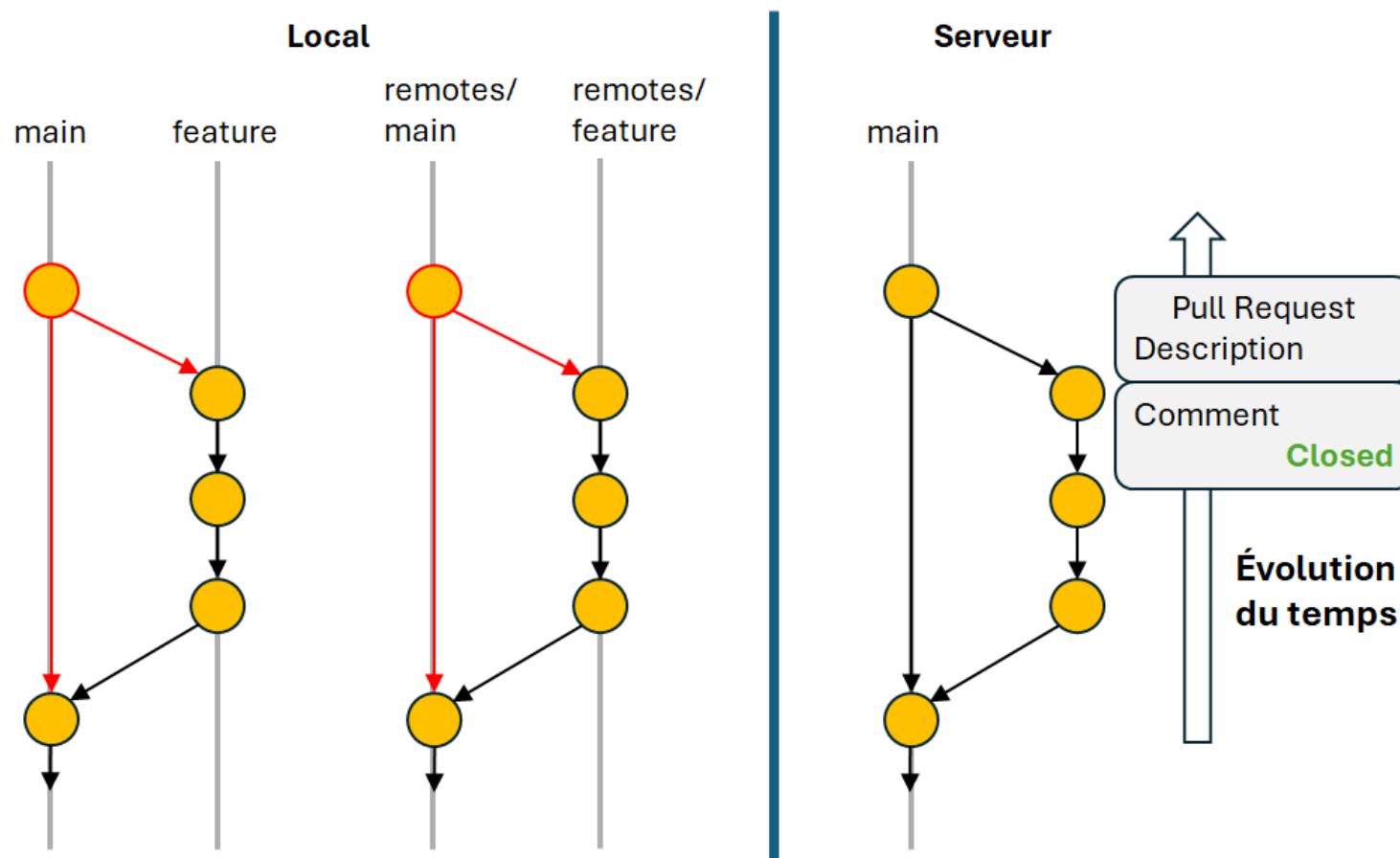
Pull Request

Etape 10

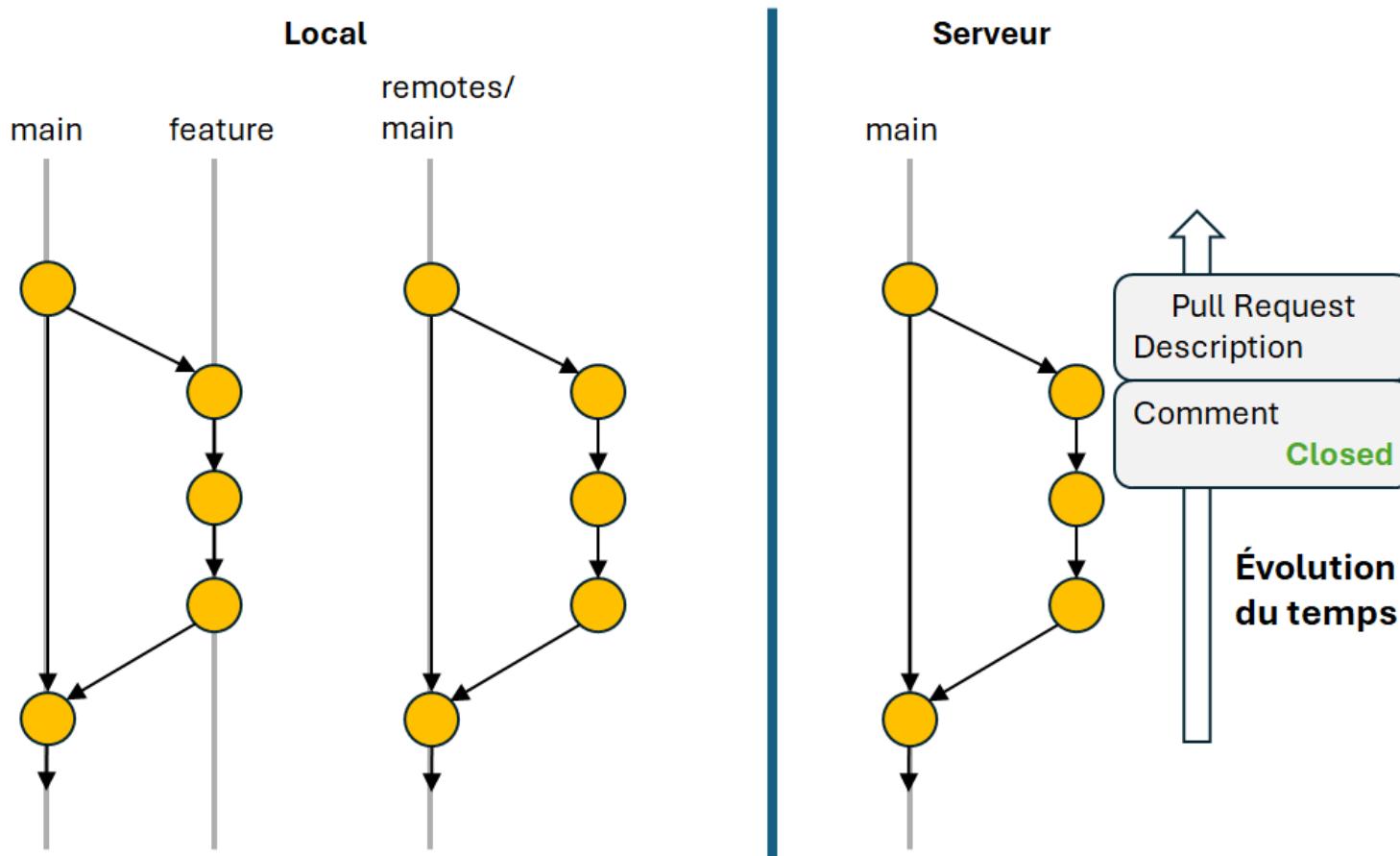
- Suppression (sur le serveur) de la branche devenue inutile



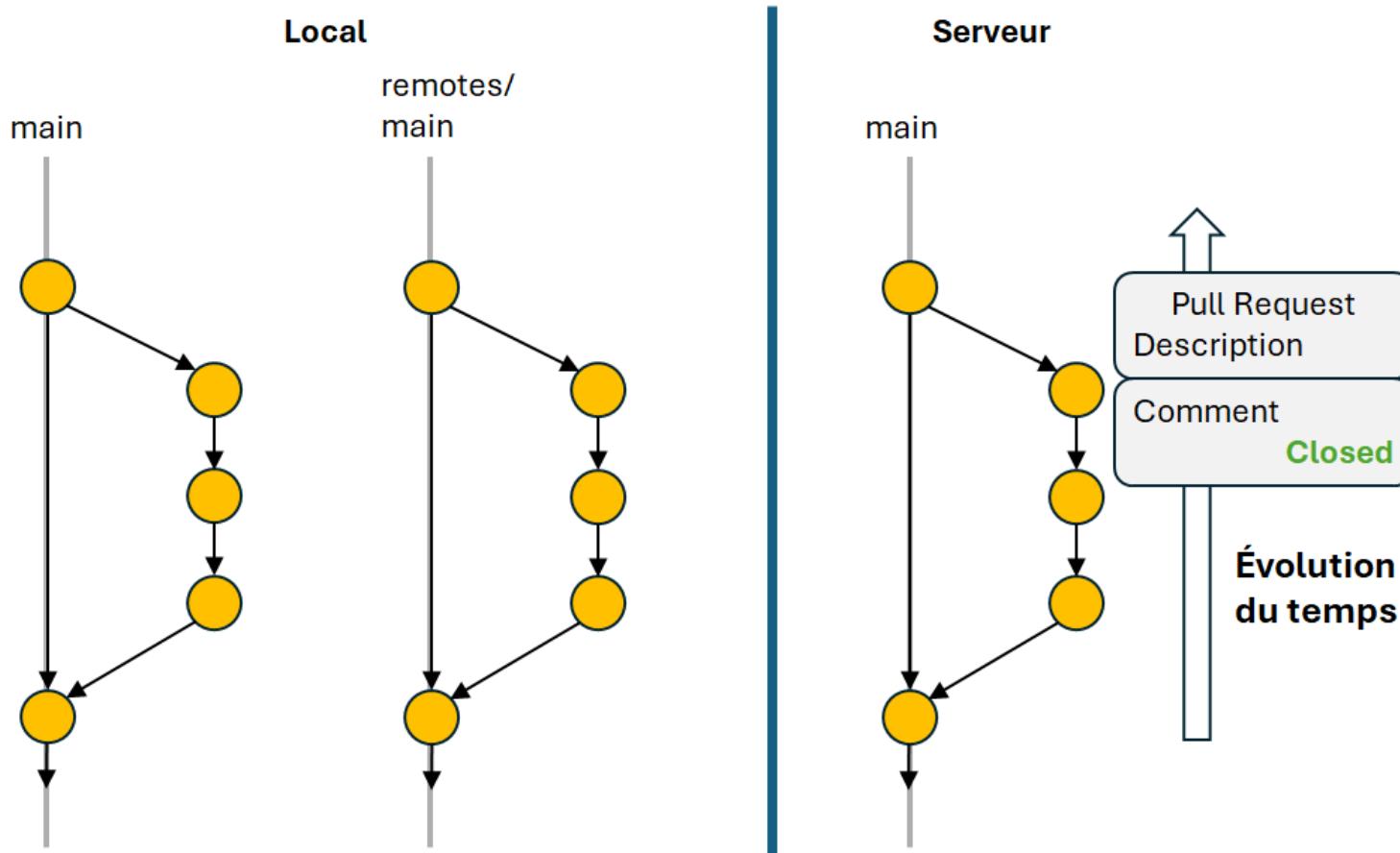
- Récupération en local du merge
- `git switch main`
- `git pull`



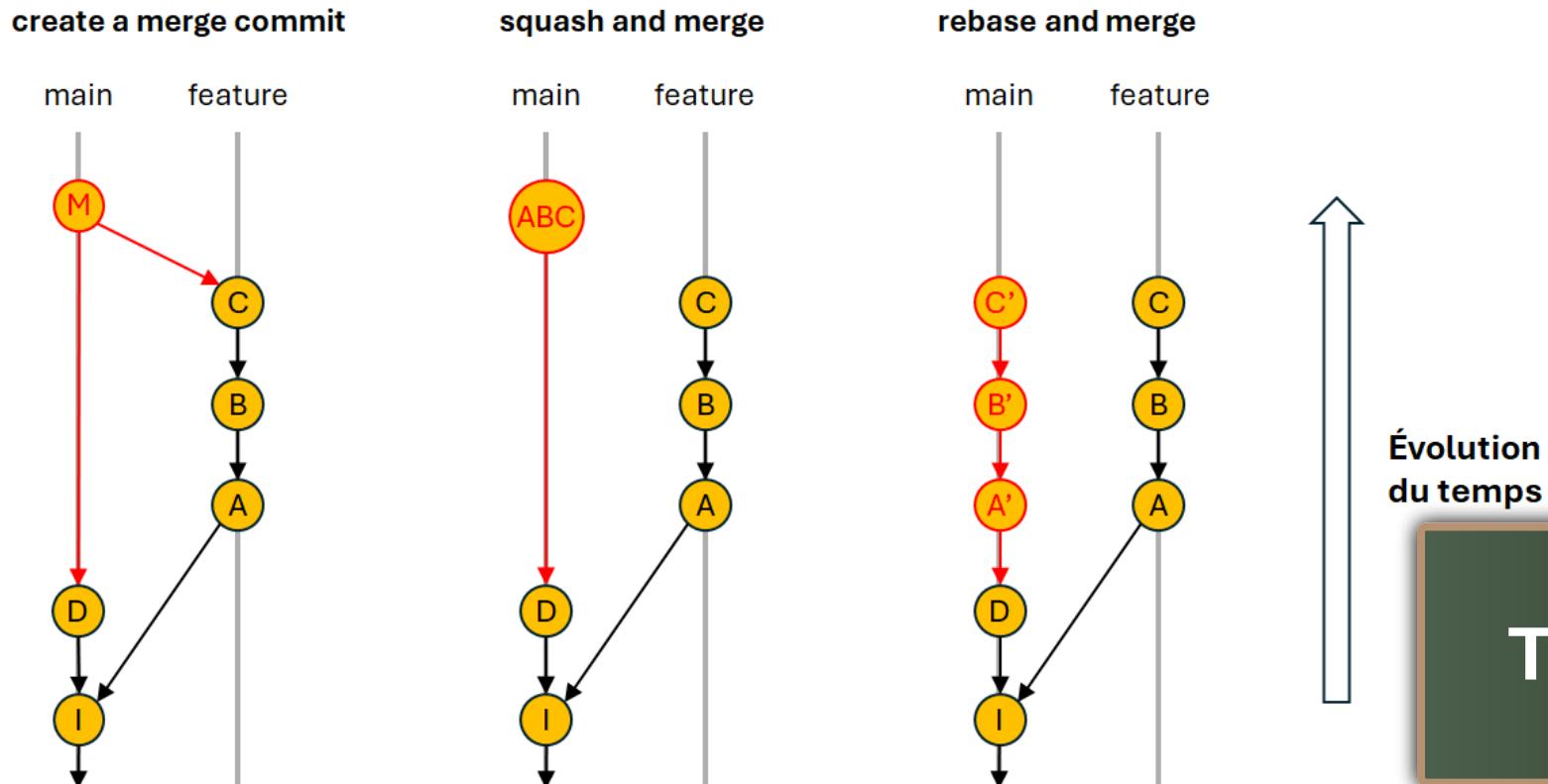
- **Resynchronisation des branches remotess**
 - `git fetch --prune` ou bien `git pull --prune`



- Suppression de la branche locale devenue inutile
 - `git branch -d feature`



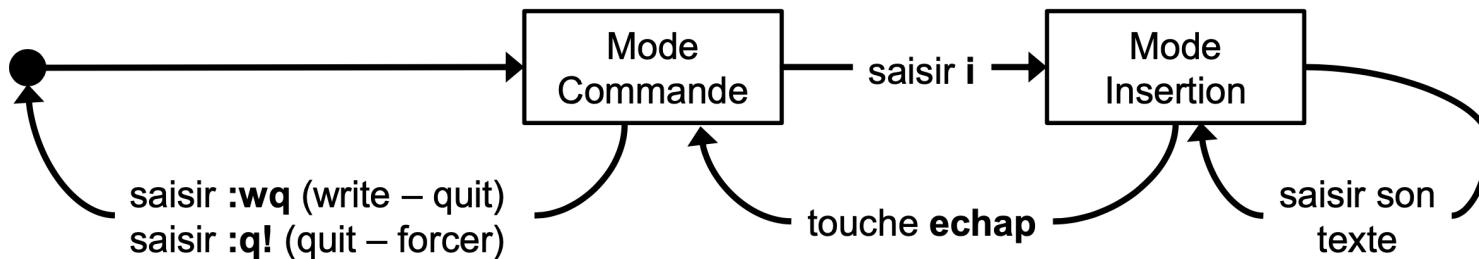
- **Conclure la Pull Request, trois options :**
 - **create a merge commit** : force un true-merge (`git merge --no-ff`)
 - **squash and merge** : produit un commit équivalent (`git merge --squash`)
 - **rebase and commit** : recopie les commits dans la branche cible (`git rebase`)



Git avancé

- **Git peut être configuré pour ignorer certains fichiers**
 - Les résultats de compilations, les logs, les fichiers techniques, ...
- **Un fichier pour cela : .gitignore**
 - Généralement à la racine du projet
 - Peut être dans n'importe quel répertoire, à partir duquel il prend alors effet
- **Syntaxe du fichier**
 - Les lignes qui commencent par **#** sont des commentaires
 - **/TODO** : ignorer le fichier **TODO** à la racine du répertoire
 - **tmp/** : ignorer les fichiers du sous-répertoire **tmp**
 - ***.log** : ignorer les fichiers **.log** dans ce répertoire et en dessous
 - **doc/**/*txt** : ignorer tous les fichiers **.txt** sous le répertoire **doc/**
 - ***.[oa]** ignorer les fichiers **.o** et **.a**
 - **!lib.o** : ne pas ignorer **lib.o** malgré la règle précédente

- **Controverse sur les options --message ou -m du commit**
 - Favorisent des commits courts, sans détail, ne respectant pas les conventions
- **On conseille plutôt de laisser Git ouvrir l'éditeur de message**
 - On saisit alors son message complet sur plusieurs lignes et on ferme l'éditeur
 - Git utilise **vim** par défaut (l'éditeur standard d'unix/linux basé sur **vi**)
 - Ce qui nécessite de connaître les bases minimales de survie à cet outil



- **Sous Windows, configurer directement l'éditeur notepad**
 - On peut bien sûr configurer l'éditeur de son choix

```
$ git config --global core.editor notepad
```

Extraire un tag ?

- **Tag = référence donc checkout produira un HEAD détaché**
 - Ce qui posera des soucis si on commite ensuite

```
$ git checkout v0.9  
Note: switching to 'v0.9'.  
  
You are in 'detached HEAD' state. You can look around,  
make experimental  
...
```

NON !!!

- **La bonne approche est de créer une branche**

- L'ancienne commande : **git checkout -b <branch> <tag>**
- La commande moderne : **git switch -c <branch> <tag>**

```
$ git switch -c brancheTag v0.9  
Switched to a new branch 'brancheTag'
```

OUI !!!

Revenir à une situation précédente sur son dépôt local

- Annuler les modifications en cours avec **git reset**
- Il existe plusieurs types de reset :
 - **--soft** : repositionne le **HEAD** vers le commit indiqué (et la branche courante)
 - Cette opération ne touche pas à l'index ni à la copie de travail
 - **--mixed** (par défaut) : repositionne le **HEAD** et vide l'index
 - **--hard** (dangereux) : repositionne le **HEAD**, l'index et la copie de travail
 - Le seul vraiment dangereux : on peut perdre ses modifications locales
 - Remarque : ne prend pas en compte les fichiers inconnus (**git clean -fd** pour les enlever)
- Attention : ne pas faire de reset sur un historique public
 - C'est-à-dire sur les dépôts partagés
- **git reset est une commande dangereuse => lire la doc !**

```
$ git reset --hard HEAD~
```

Annuler complètement le dernier commit et les dernières modifications. Repositionne (en local) la branche et le HEAD sur le commit précédent

- **Deux cas classiques :**
 - Oubli d'une modification dans le commit
 - Fréquent avec le **commit -a** qui ne prend que les fichiers modifiés par exemple
 - Erreur, oubli, faute de frappe dans le message de commit
- **Il faudrait annuler le dernier commit et le refaire**
 - Faisable avec **git reset** mais un peu lourd
- **Modifier le dernier commit local git commit --amend**
 - Après avoir mis à jour l'index ou pas
 - L'éditeur s'ouvre et propose le dernier message de commit pour le modifier
- **Attention : ça recrée un nouveau commit qui remplace le précédent**
 - Car la date et le message de commit participe au calcul du hash
 - Ne pas utiliser si on a déjà poussé le commit sur un dépôt distant

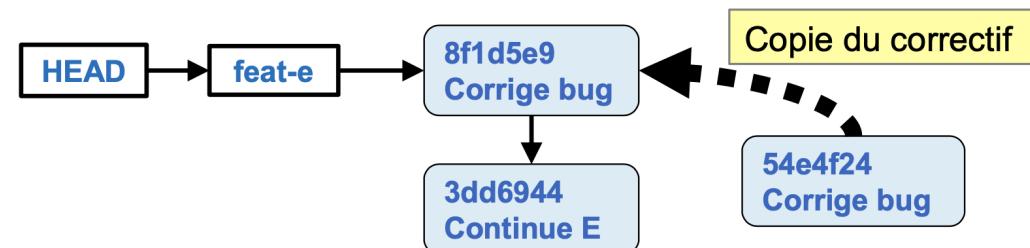
```
$ git add nouveau-fichier.txt && git commit --amend -m "Nouveau message"
```

- **Annoter les lignes d'un fichier avec git blame <fichier>**
 - Affiche l'identifiant court, le nom de l'auteur et la date devant chaque ligne
 - Les lignes qui étaient dans le premier commit sont précédées d'un ^

```
$ git blame hello.txt
^a9fa86c (LEFEVRE Jean-Francois 2020-05-22 15:45:29 +0200 1) Hello World!
e5f19eb0 (LEFEVRE Jean-Francois 2020-05-29 17:48:00 +0200 2) finalisation
a31be6f3 (LEFEVRE Jean-Francois 2020-06-03 12:11:33 +0200 3) correction bug-41
ca4f3ca0 (LEFEVRE Jean-Francois 2020-06-04 15:05:49 +0200 4) feature C
391576d1 (LEFEVRE Jean-Francois 2020-06-04 21:25:28 +0200 5) feature Dajout origine
54e4f249 (LEFEVRE Jean-Francois 2020-06-05 11:21:37 +0200 6) correction bug
```

- **Un bug est détecté et corrigé en parallèle d'une branche**
 - Il empêche de tester la nouvelle fonctionnalité en cours de développement
 - On ne peut pas merger cette branche (elle n'est pas prête et elle n'est pas testée)
- **Il y a tentation de récupérer le correctif dans la branche**
 - Un **diff** et un copier/coller des modifications ? (un peu trop manuel)
- **Cueillir les cerises avec git cherry-pick <id> ?**
 - La commande permet de copier un commit dans la branche courante
 - **Remarque** : cette commande est controversée, ne pas l'utiliser !!!
 - Elle crée un nouveau commit (nouveau hash) avec les mêmes infos (diff, auteur, date, ...)
 - On obtient des commits semblables dans différentes branches, cela amène de la confusion

```
$ git cherry-pick 54e4f24
[feat-e 8f1d5e9] Corrige bug
Date: Fri Jun 5 11:21:37 2020 +0200
1 file changed, 1 insertion(+)
```



- **La commande git rebase pour déplacer une branche**
 - Comme son nom l'indique, elle change la base de la branche
 - Déplacer la branche courante avec **git rebase <branche-cible>**
 - On peut préciser la branche à déplacer en second paramètre (fera un checkout de cette branche)
- **Exemple de la documentation (avec topic la branche active)**



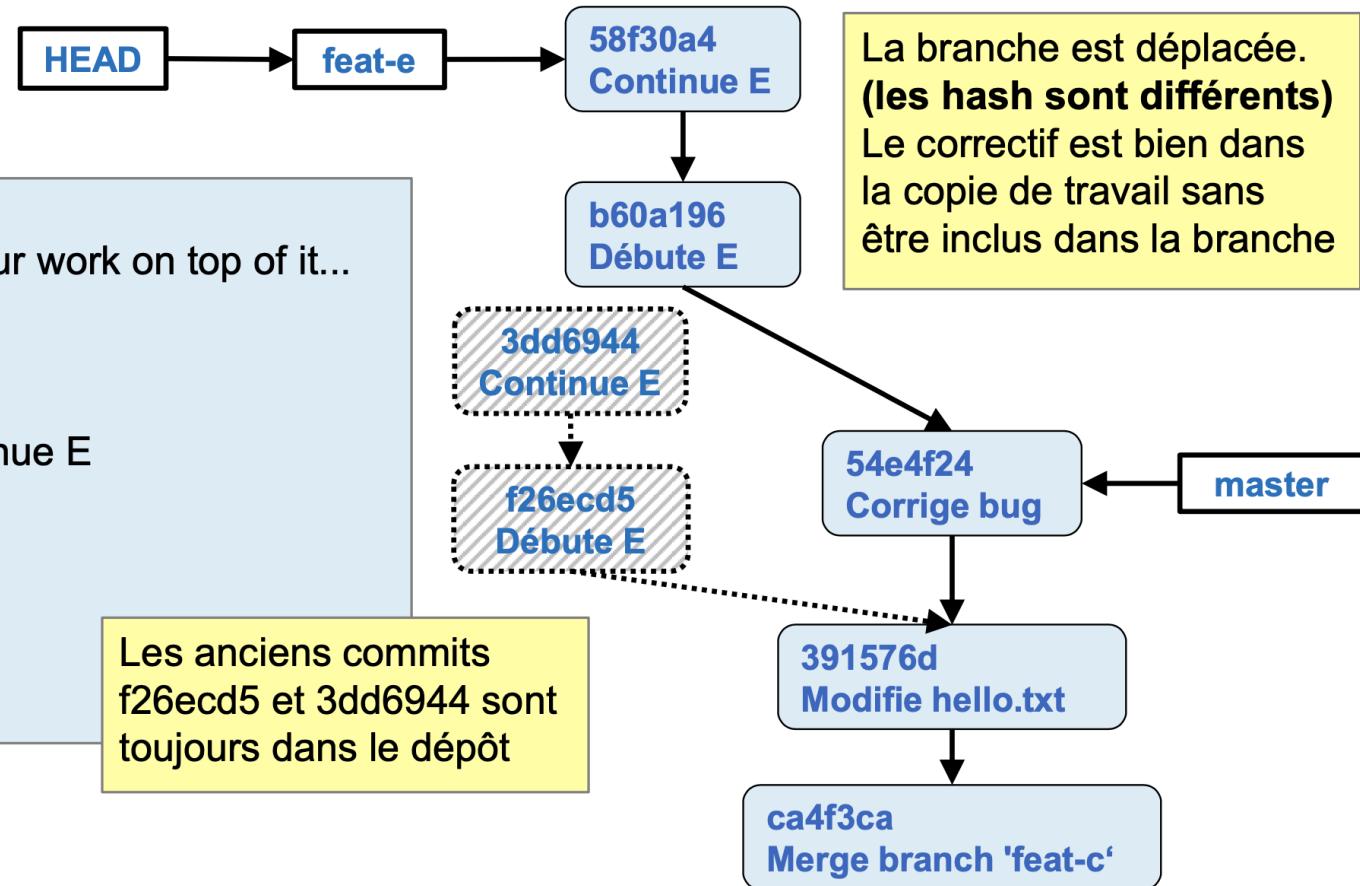
La branche **topic** est déplacée et raccrochée au bout de la branche **master**.
La commande remonte les commits depuis **C** jusqu'à trouver un ancêtre commun avec **master** (**E**)
Elle recopie alors ces commits (sans l'ancêtre commun, il est déjà dans **master**) au bout de **master**.
Noter les **A'**, **B'** et **C'** (ce sont des copies comme avec un cherry-pick)
La branche **topic** pointe maintenant **C'** (la branche contient le même code mais avec **F** et **G** inclus)

- **La commande est plus riche, et offre d'autres possibilités**
 - C'est un véritable scalpel pour extraire des branches et les greffer ailleurs
 - Voir la documentation avec **git help rebase**

- **Appliquons la commande sur notre exemple**

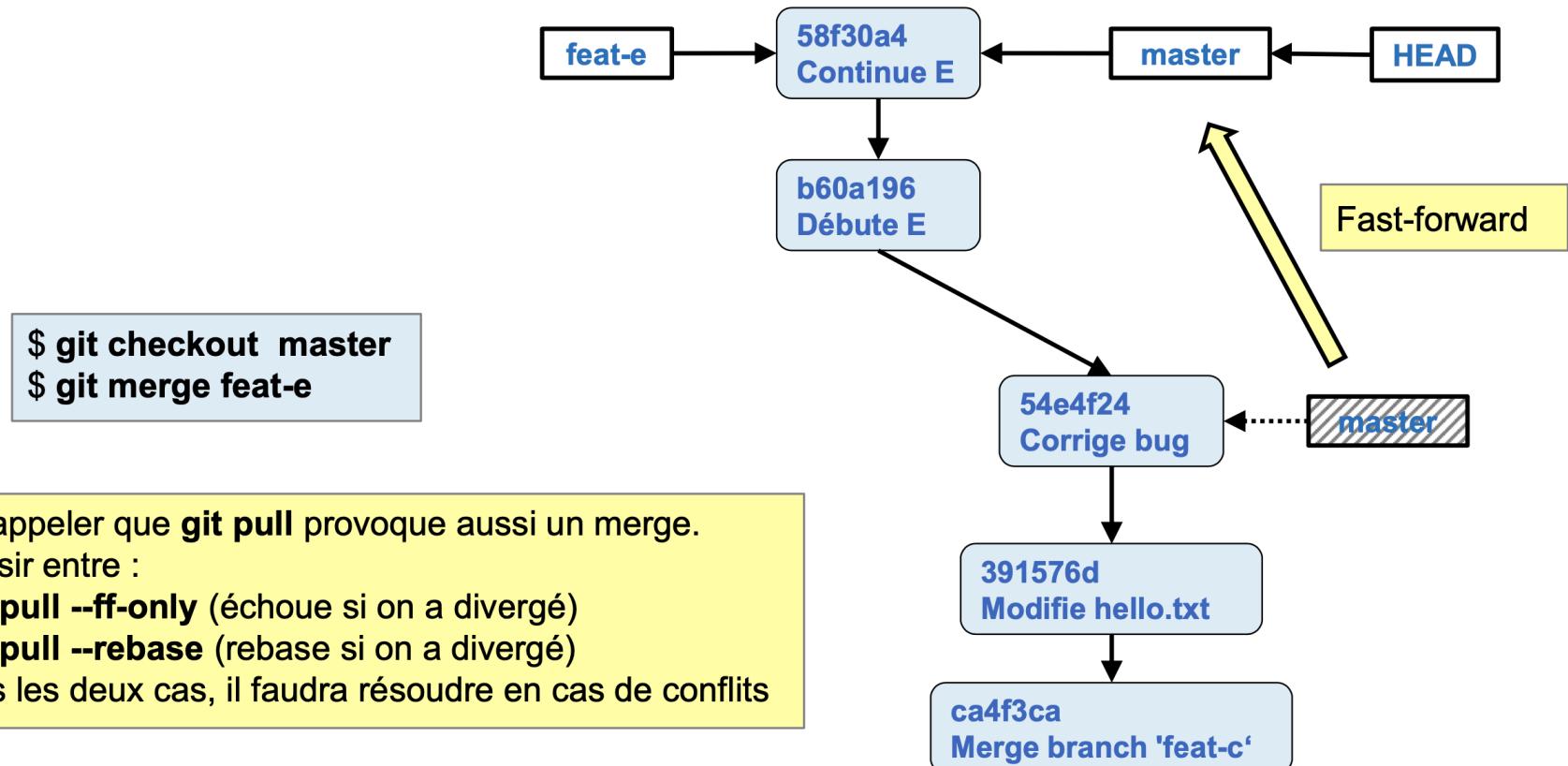
- On rebase la branche **feat-e** sur **master** pour la raccrocher derrière

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Débute E
Applying: Continue E
$ git log --oneline --graph -5
* 58f30a4 (HEAD -> feat-e) Continue E
* b60a196 Débute E
* 54e4f24 (master) Corrige bug
* 391576d Modifie hello.txt
* ca4f3ca Merge branch 'feat-c'
\\
```



- **Le rebase rend possible un historique linéaire**

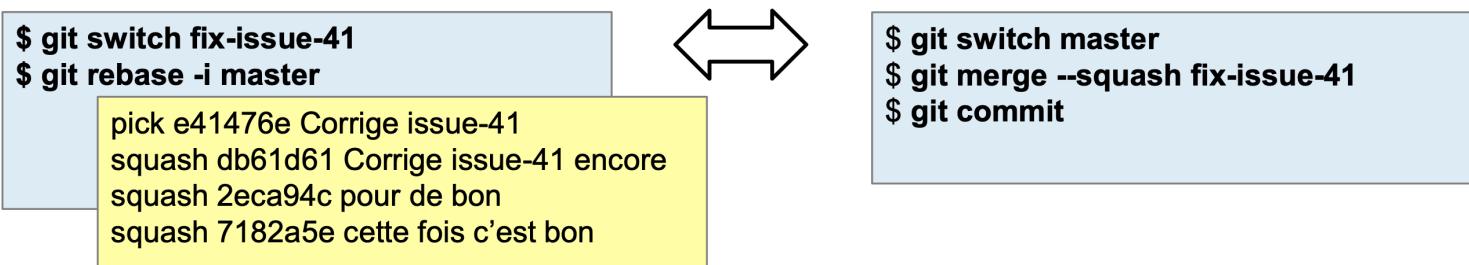
- Si combiné avec le merge **--ff-only** (interdiction de créer des true merge)
- Le rebase permet d'actualiser la branche et de la merger avec un Fast-forward



- **Le rebase peut aussi permettre de réécrire l'historique**
 - Insérer un correctif dans une ancienne version
 - Remettre au propre l'historique de son développement local sur une branche
 - Déplacer une sous-branche, réordonner des commits, les regrouper, ...
- **Le rebase interactif (`git rebase -i`)**
 - Ouvre l'éditeur avec la liste des commits à rebaser (dans l'ordre chronologique)
 - Pour chacun d'eux, on précise une action
 - **pick** = garde le commit (**cherry-pick**)
 - **reword** = garde et modifie le message
 - **edit** = garde mais stoppe pour modifier le commit
 - **squash** = fusionne son contenu avec le précédent
 - **fixup** = comme squash mais efface le message
 - **exec** = exécute une commande avec un shell
 - **drop** = supprime le commit (et son contenu)

```
pick e41476e Corrige issue-41
pick db61d61 Corrige issue-41 encore
pick 2eca94c pour de bon
pick 7182a5e cette fois c'est bon
```

- **Grace au rebase, il est possible de simplifier l'historique**
 - Développer en toute liberté et sécurité (commits fréquents, granularité faible)
 - Remettre au propre à la fin avant de livrer un historique clair
- **Idéalement, après livraison**
 - 1 fonctionnalité (ou correction) = 1 commit (granularité plus grosse)
 - Clairement décrit par un message conventionnel
 - Et clairement détaillé et justifié
 - **git rebase -i <commit>** permet cela (avec l'action **squash**)
 - **git merge --squash <branche>** est une autre solution
 - Nécessite ensuite un **git commit**



- **Le rebasing est dangereux**
 - Il « supprime » des commits existants (en réalité, ils existent toujours)
 - Puis les recrée avec le même message, mais pas la même signature
 - Ou il en crée de nouveaux (fusion de commits)
 - Tout cela est possible en local (peu dangereux si on sait ce qu'on fait)
 - Mais devient problématique si on a poussé vers un autre dépôt
- **Si on rebase des commits déjà publiés sur d'autres dépôts :**
 - On introduit des incohérences
 - Certains développeurs auront pu baser leur travail sur les commits disparus
 - Ils devront donc fusionner leur travail avec des modifications existantes
 - Et réintroduiront des anciens commits déjà existants... en boucle...
 - Il faut être prudent, il n'y a pas de message d'avertissement
- **Il ne faut jamais rebaser des commits publiés**

- **Parfois, on a des modifications qu'on ne veut pas commiter**
 - Pas terminées, le commit ne serait pas à la bonne place, ...
- **Une pile d'enregistrements de modifications : git stash**
 - Cela enregistre (remise) toutes les modifications locales (pas de commit)
 - Nettoie la copie de travail et permettra de les restituer quand on veut

```
$ echo "nouvelle information" >> README
$ git pull
error : your local changes to the following files will be
overwritten by merge :
      readme.txt
$ git stash
$ git status
On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean $ git pull
$ git stash pop
$ tail -n 1 readme.txt
nouvelle information
```

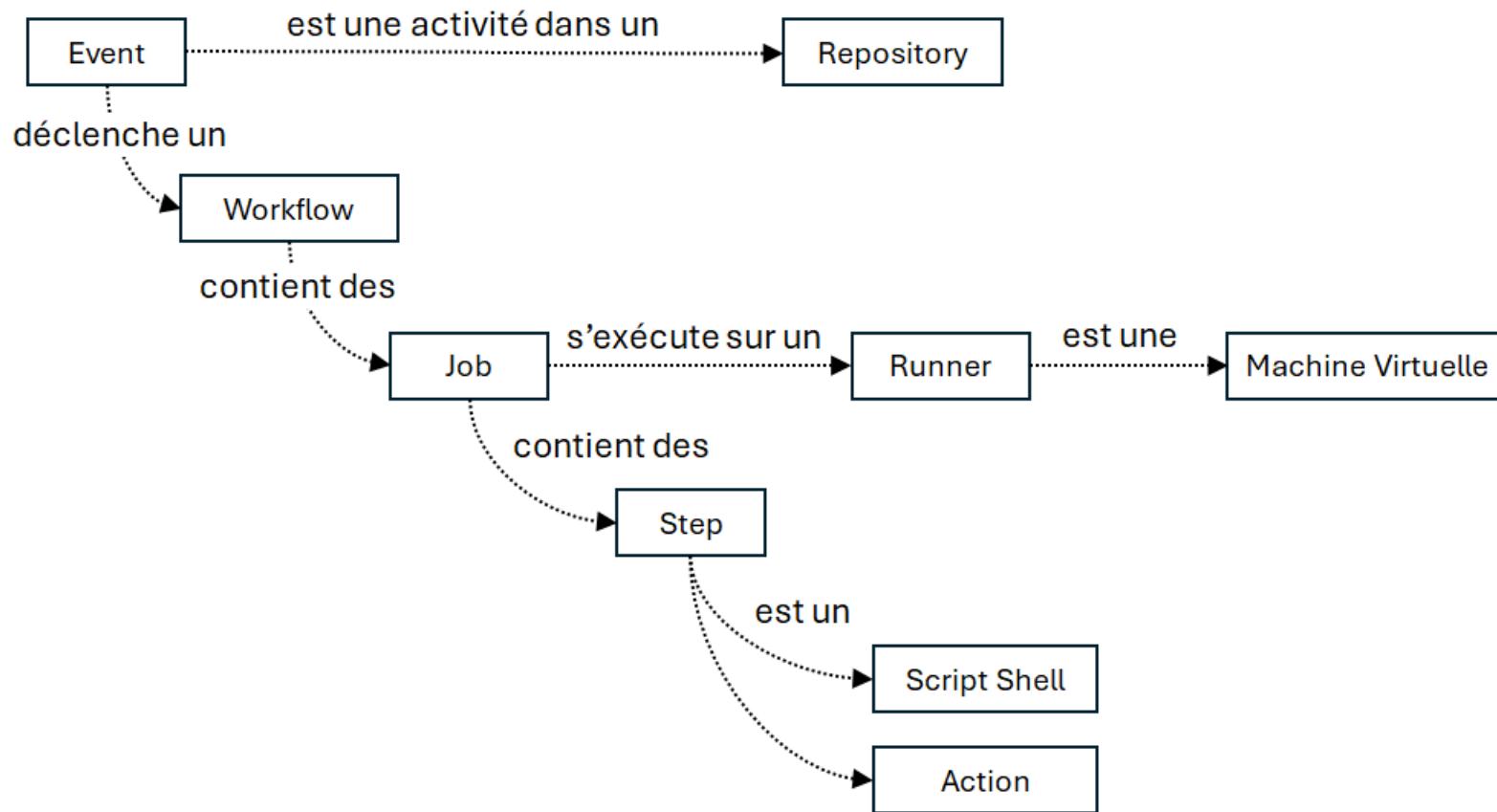
- **git bisect permet une recherche de bug par dichotomie**
 - Elle aide à trouver par dichotomie dans quel commit a été introduit un bug
 - On lance la procédure de recherche avec **git bisect start**
 - On indique que le **HEAD** est **bad** (présente le bug) avec **git bisect bad**
 - On indique un commit **good** (sans le bug) avec **git bisect good <commit>**
 - On lance ses tests et on indique si c'est good ou bad (ou skip si pas testable)
 - A chaque réponse l'intervalle des commits est divisé, jusqu'à trouver
- **La procédure peut même être automatisée (git bisect run)**
 - Il faut juste une commande renvoyant **0** si **OK**, autre chose si **KO**

```
$ git bisect start
$ git bisect bad
$ git bisect good 905d84bb
$ git bisect run test.sh
83a8fe288c6ef6989f96145499de87c3c58b3ec0 is the first bad commit ...
$ git bisect reset
```

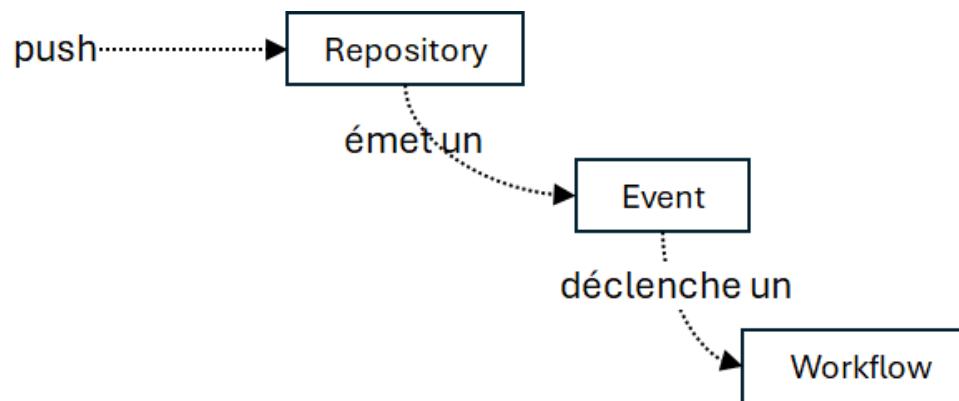
La CI/CD avec GitHub Actions

- **GitHub Actions = Plateforme CI/CD**

- Automatise les pipelines de génération, de test et de déploiement
- Mais permet de réagir à toute sorte d'événements dans le dépôt



- **Correspond à une activité sur le dépôt**
 - Exemple : création/suppression d'une branche, push, pull request, issue...
- **Peut aussi correspondre à :**
 - Une heure **planifiée**
 - Un déclenchement **manuel** ou via un appel REST
 - Un **événement externe** se propageant jusqu'au dépôt
 - Evénement `repository_dispatch`
- **Lorsque l'événement se produit :**
 - Recherche et exécute le ou les workflow associés



- Décrit dans un fichier **YAML** dans `.github/workflows/`
 - Indique sur quels événements il doit être déclenché (**on:**)
 - Contient des **jobs**

```

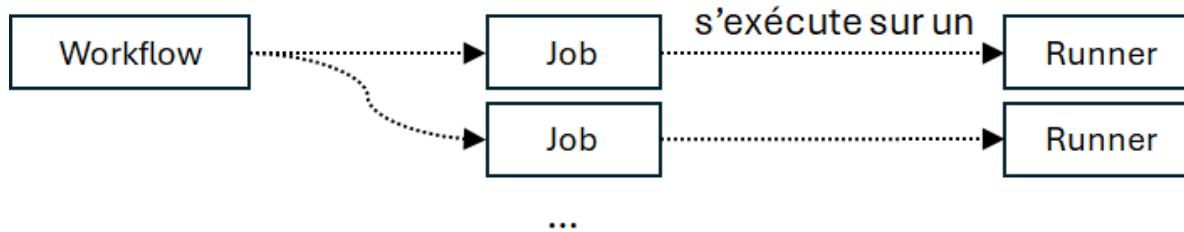
name: CI                      # nom du workflow

on:                                # section des déclencheurs
  push:                            # déclenché sur un push
    branches: [ "main" ]           # mais seulement sur la branche main
  pull_request:                   # déclenché aussi sur une pull_request
    branches: [ "main" ]           # mais seulement sur la branche main

  workflow_dispatch:              # déclencheable aussi manuellement

jobs:                               # définit les jobs du workflow
...

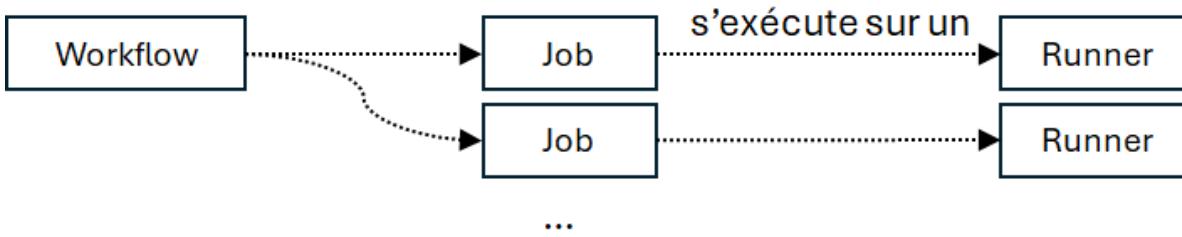
```



- S'exécute sur un runner (**runs-on:**)

- En parallèle par défaut
- En séquence quand il dépend d'un autre job (**needs:** <id du job à attendre>)

```
...
jobs:                                # définit les jobs du workflow
  job1:                                # id du job
    runs-on: ubuntu-latest
    steps: ...
  job2:                                # deuxième job en parallèle
    runs-on: ubuntu-latest
    steps: ...
  job3:                                # troisième job en séquence
    runs-on: ubuntu-latest
    needs: job1, job2                  # s'exécutera après job1 et job2
    steps: ...
...
...
```



- **En séquence dans un job**

- Sur le même **runner** pour tout le job (permet le partage de données)
- Soit un script shell (**run:**) : une ou plusieurs commandes shell
- Soit une action GitHub réutilisable (**uses:**)
 - Permet de diminuer la quantité de code répétitif
 - Disponible sur le GitHub Marketplace mais on peut aussi créer les siennes

```
...
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3          # réutilise l'action checkout

      - name: Run a one-line script      # un step exécutant une commande
        run: echo Hello, world!

      - name: Run a multi-line script   # un step exécutant plusieurs commandes
        run: |
          echo Add other actions to build,
          echo test, and deploy your project.
```

- Définit le type de machine sur laquelle exécuter le job (**runs-on:**)
 - Soit un runner hébergé par GitHub
 - Avec un ou plusieurs **label** permettant de sélectionner le runner (ex : **ubuntu-latest**)
 - Il existe des runners avec plus de mémoire ou de disque pour les entreprises
 - Soit un groupe de runners (**group: ubuntu-runners**)
 - Soit un runner auto-hébergé (**runs-on: [self-hosted, linux]**)
- On peut réclamer une exécution dans un conteneur (**container:**)

```
...
jobs:
  container-job:
    runs-on: ubuntu-latest
    container: node:18          # raccourci pour spécifier l'image
    steps:
      - name: print versions
        run: |
          node -v                 # s'exécute dans le conteneur
          npm -v                  # v18.19.1
                                    # 10.2.4
```

- GitHub Actions permet d'automatiser les flux de travail
- Possibilité d'utiliser des actions prédéfinies
- Possibilité de créer des actions personnalisées
- Peut être utilisé pour automatiser des tâches
 - Les tests
 - Le déploiement
 - Les notifications
 - Et bien d'autres

TP 4

Implémenter le déploiement avec ArgoCD

- **Bref Historique**

- 1972 : IBM VM/370
- 1999-2000 : FreeBSD Jails
- 2004 : Linux Vservers, Solaris Containers
- 2007 : projet Linux Containers (LXC)
 - Principe de cloisonnement de sous-environnements sans virtualisation mais dans un conteneur
- 2013 : dotCloud crée Docker

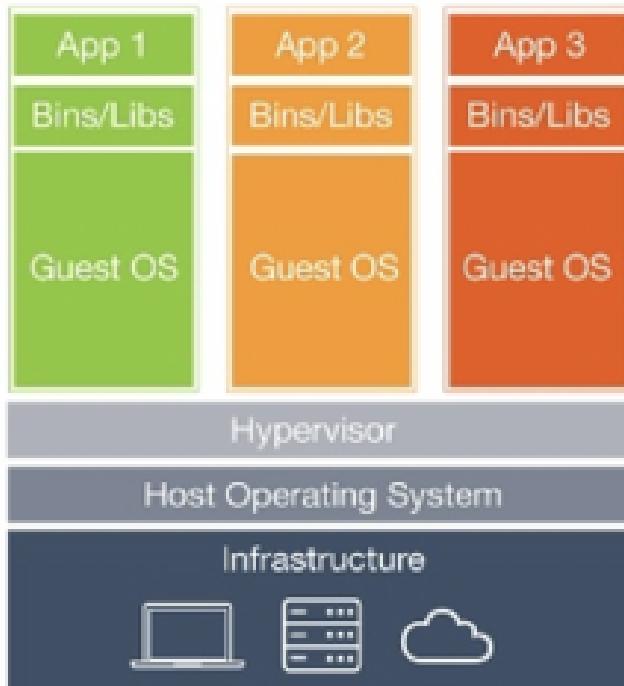
- **Engouement envers la technologie**

- Au centre des technologies les plus significatives
- Implémentation du support par les fournisseurs Cloud
- Retour rapide sur la technologie
- Correction des bugs très rapide
- Ajouts de fonctionnalités

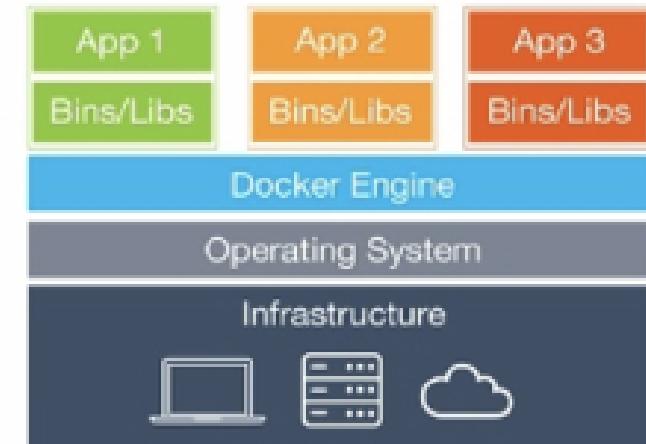
- **La force de Docker repose sur :**
 - Utilisation d'une technologie de type **conteneur**
 - Dont la mise en œuvre est simple
 - Le concepts d'images
 - Encapsule les fichiers requis pour l'exécution
 - Des méta-données
 - Une **image** est exécutée dans un **conteneur**
- **Docker simplifie la création de conteneurs au travers de :**
 - Son propre système de **Dockerfile**
 - Images de base (Ubuntu, Debian, Fedora, Centos, Alpine, ...)
 - Images de technologie prête à l'emploi (php, python, java...)
 - Application clef en main (MySQL, Apache, Nginx, ...)

- **Les conteneurs sont plus légers**

- Ils reposent sur l'OS hôte
- Moins de ressources
 - (CPU, RAM, disque, ...)



Virtual Machines



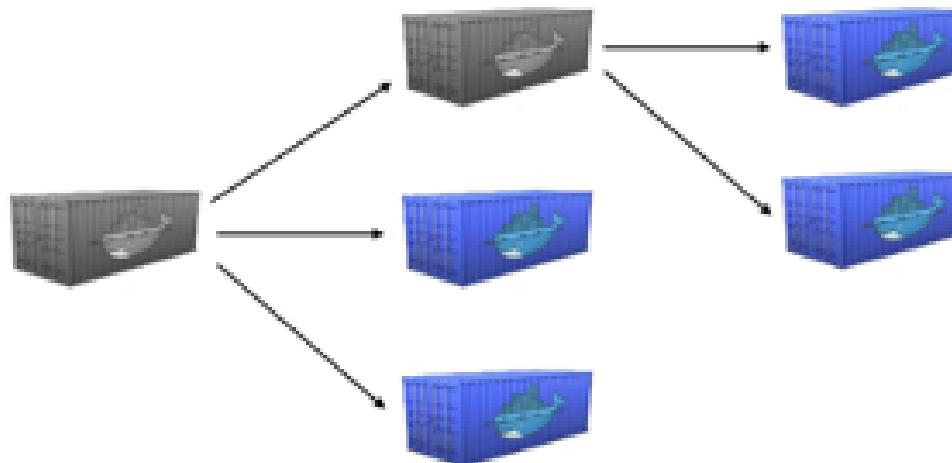
Containers

- **Image**

- Il s'agit de la représentation statique du conteneur
- Est écrite sous la forme de Dockerfile
- Est conçue pour être distribuée
- Est stockée dans un registre (Docker hub, Nexus, ...)

- **Conteneur**

- Instance d'une image
- N instances / environnement



```
# lister les conteneurs
docker ps # -a même ceux qui sont stoppés

# lancer un conteneur
docker run --name myapache -d -p 8080:80 -v
"${pwd}/data:/usr/local/apache2/htdocs" httpd:2.4
docker run -it nginx bash

docker exec -it myapache bash

# afficher les logs
docker logs -f ID_CONTENEUR

# stopper/supprimer
docker stop/rm

# construire une image docker
docker build -t TAG .
docker tag SOURCE TARGET
docker push my.docker.repo.com/my/image/name:1.2.3
```

- **Un Dockerfile est un fichier de configuration**
 - Il contient la description de l'image à créer sous forme d'une suite d'instructions
 - Un commentaire commence par le caractère #
- **Document de référence**
 - <https://docs.docker.com/engine/reference/builder/>
- **Avantages :**
 - Automatiser la création d'une image
 - Reproductible
 - Maintenable
 - Rapide

```
FROM alpine:3.3
ENV LANG C.UTF-8
RUN { echo '#!/bin/sh'; \
      echo 'set -e'; \
      echo; \
      echo 'dirname "$(dirname "$(readlink -f \
      "$(which javac || which java)")")"'; \
    } > /usr/local/bin/docker-java-home \
    && chmod +x /usr/local/bin/docker-java-home
ENV JAVA_HOME /usr/lib/jvm/java-1.8-openjdk
ENV PATH $PATH:$JAVA_HOME/bin
ENV JAVA_VERSION 8u72
ENV JAVA_ALPINE_VERSION 8.72.15-r2
RUN set -x && apk add --no-cache \
      openjdk8="$JAVA_ALPINE_VERSION" \
      && [ "$JAVA_HOME" = "$(docker-java-home)" ]
```

- **Un build multi-stage contient plusieurs instructions FROM**
 - Chacune débute un nouveau stage pouvant utiliser une base différente
- **Tout peut alors se faire avec un unique Dockerfile**
 - Etapes intermédiaires de construction
 - Image final construite en copiant les artefacts des étapes précédentes

```
# l'image build est temporaire, juste pour le build
FROM java:8 AS build
COPY Hello.java .
RUN javac Hello.java

# création de l'image de production
FROM openjdk:8-jre-stretch
COPY --from=build Hello.class .
```

- **Framework open source créé par Google**
 - Pas besoin d'installer Docker
 - Pas besoin de Dockerfile
- **Avantages**
 - Pure Java
 - Rapidité
 - Reproductivité
- **Possibilité de l'ajouter comme plugin dans le pom.xml**

- **Buildah et Podman**
 - Alternative à Docker mais rootless
- **Buildpacks**
 - Pas besoin de dockerfile
 - Détection du language utilisé
 - Construction d'une image avec un entry point et des scripts de démarrage
- **Shipwright et Kaniko**
 - Construction de l'image dans Kubernetes

TP 5

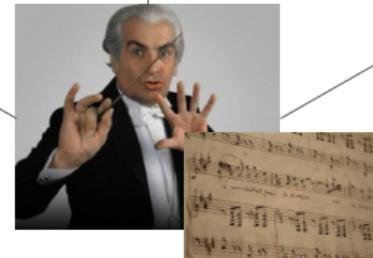
- Docker en production ? mais comment ?
- Comment :
 - Démarrer un projet (un ensemble de service) ?
 - Gérer les containers arrêtés suite à une erreur ?
 - Gérer les pannes des machines/hosts ?
 - Gérer la maintenance de vos machines/hosts ?
 - Gérer les mises à jour de vos composants ?
 - Gérer les mises à l'échelle (scale up/down) ?
 - Gérer la multiplication du nombre de composants à déployer ?

- **Docker-compose ?**
 - docker-compose est tout-à-fait capable de faire tout ça mais...
 - ... la notion de production n'a pas forcément la même signification pour tous
- **Comment :**
 - Ajouter de la ressource (capacité de calcul, de stockage, ...) à un environnement, tout en conservant une disponibilité de service ?
 - Uniformiser ?
 - “Déployer” sans indisponibilité de service de la même façon ?
 - Faire un “rollback” sans indisponibilité de service de la même façon ?

- **L'orquestrateur de conteneurs est la solution**
 - Il connaît les partitions de chaque instrument (**service dans un conteneur**) nécessaire à l'accomplissement de l'oeuvre intégrale (**tous les services / projet**)



- **Un bon chef d'orchestre connaît tous ses musiciens**
- **Il est capable de leur demander de jouer moins fort / plus fort**
- **Il est capable de :**
 - Déetecter quand un seul violon parmi N violons est en incapacité de jouer
 - (en retard / pas prêt / malade)
 - Le sortir de l'orchestre, le temps qu'il soit prêt ou qu'il soit remplacé



- **D'un point de vue "utilisateur"**
 - Vous n'allez que transmettre les partitions (**descripteurs**)
 - De chaque instrument (**service / conteneur**)
 - Au chef d'orchestre (**l'orchestrateur**)
 - Et celui-ci va se débrouiller

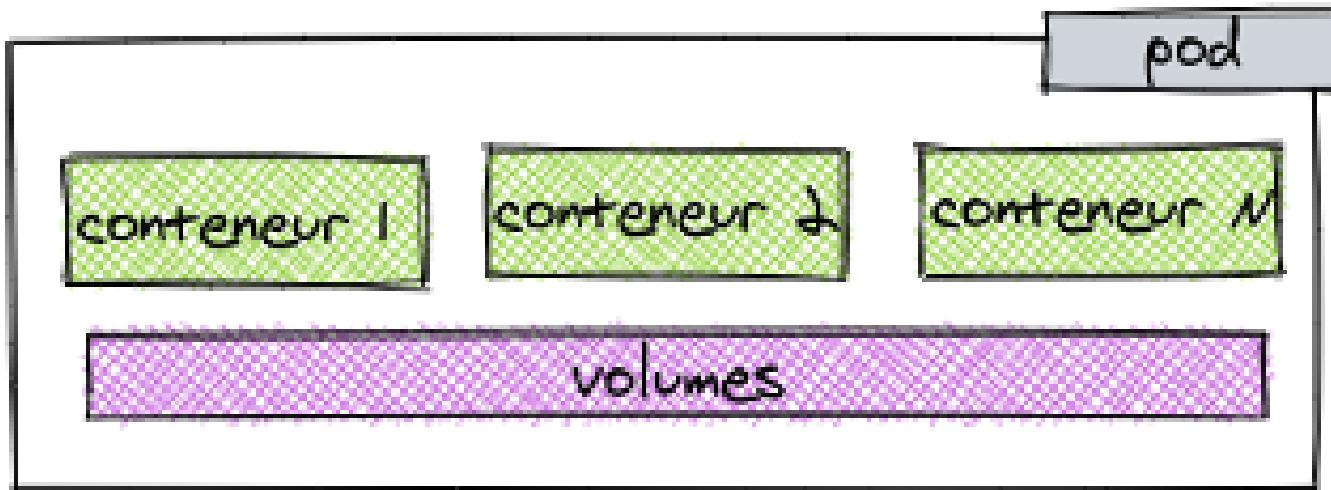


- **Google a initié le projet Kubernetes en 2014**
 - Kubernetes s'appuie sur 15 années d'expérience pendant lesquelles Google a fait tourner en production des applications à grande échelle
- **Kubernetes a été enrichi au fil du temps**
 - Par les idées et pratiques mises en avant par une communauté **très active**
- **Kubernetes vient du grec : signifie Timonier ou Pilote**
 - Qui tient le gouvernail
- **k8s est une abréviation**
 - Dérivée du remplacement des 8 lettres centrales de “K-ubernetes” par un 8
 - Prononciation anglophone : **koo-ber-nay'-tis**
 - On dit aussi “Kube”

Kubernetes

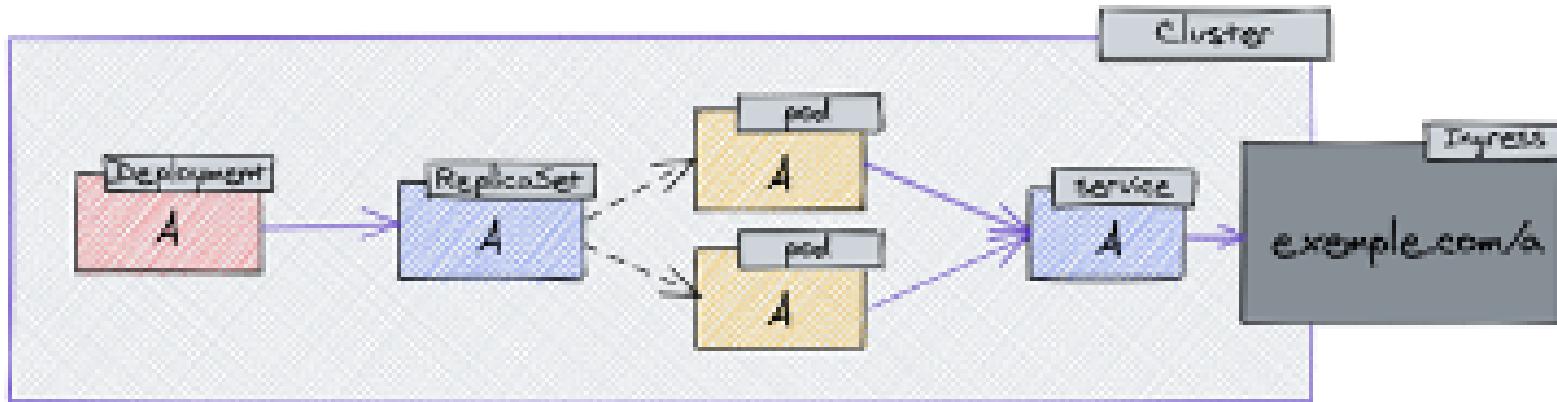
Qu'est-ce-qu'un pod ?

- **Le pod est un groupe de N conteneurs (1 à N)**
 - Qui peuvent partager des ressources entre eux
- **Souvent, il y a un seul conteneur par pod... mais pas toujours...**

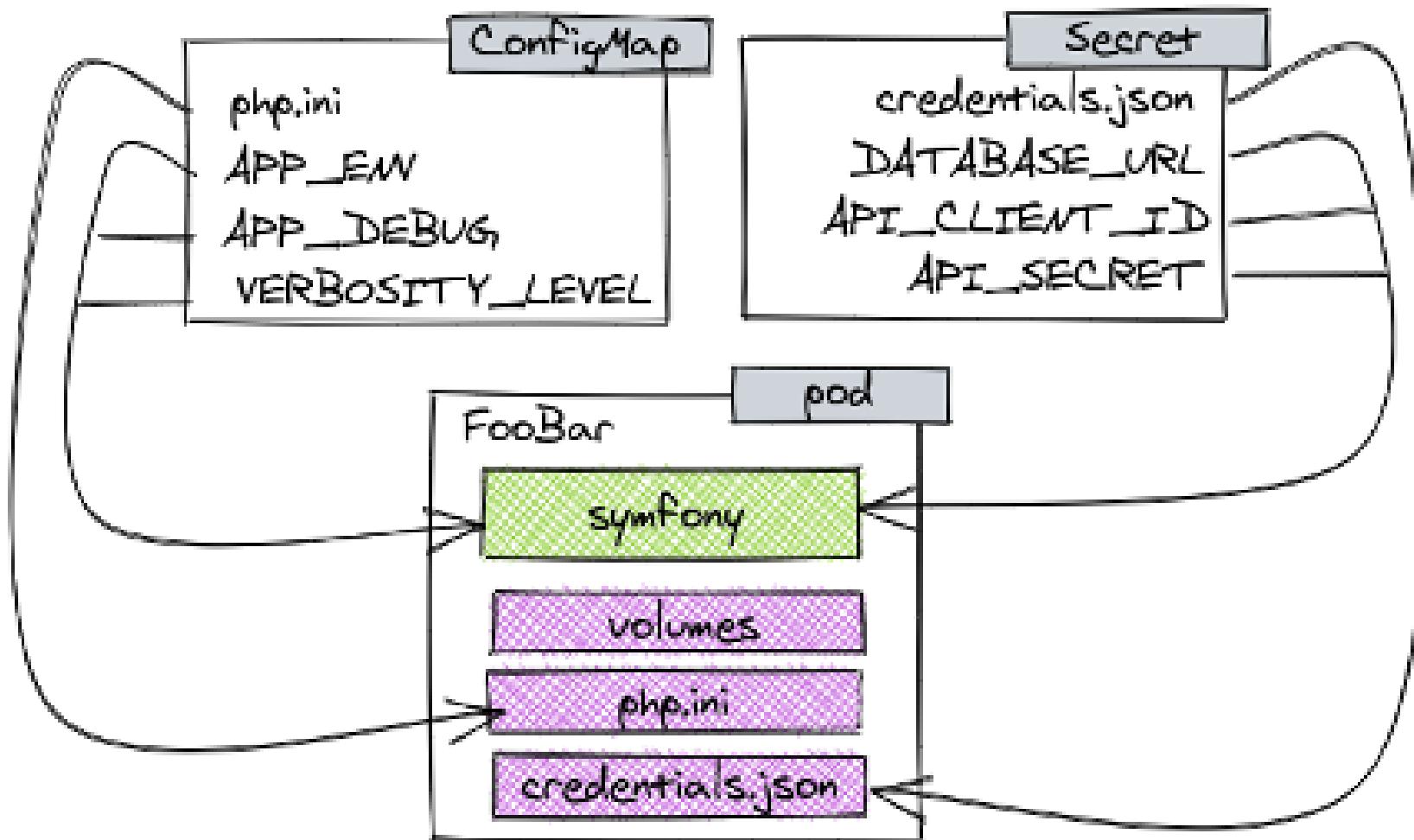


- **Deployment**
 - Permet de gérer le déploiement d'une application et ses mises à jour
 - Gère un ReplicaSet
- **ReplicaSet**
 - Crée un ensemble de Pods identiques
- **Service**
 - Load balance / assemble des Pods pour exposer le service sur un port réseau
- **Ingress**
 - Un reverse proxy HTTP (souvent utilisé pour rendre le Service public)

- **ConfigMap**
 - Permet d'injecter variables d'environnement et fichiers dans les Pods
- **Secret**
 - Similaire au ConfigMap en plus sécurisé
- **Cronjob**
 - Pour programmer le lancement d'un Job/conteneur Docker



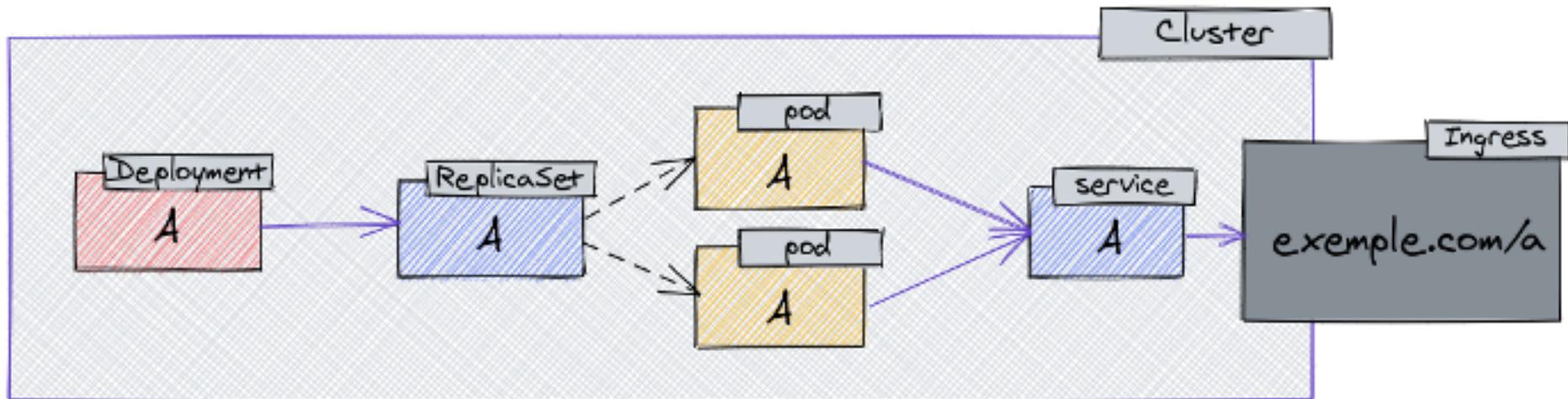
Kubernetes ConfigMap et Secrets



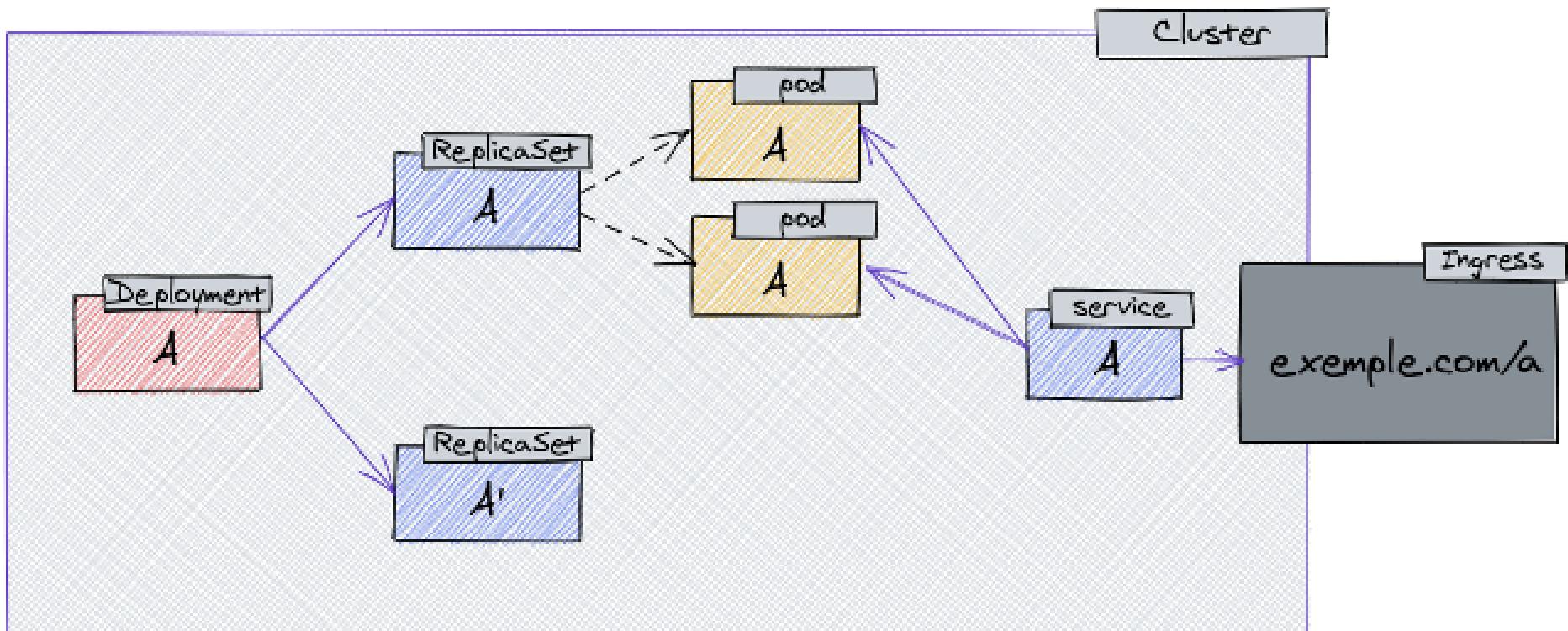
- Exemple de descripteur

```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: clinic-deploy  
  labels:  
    app: clinic-front  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      app: clinic-front  
  template:  
    metadata:  
      labels:  
        app: clinic-front  
    spec: # specification du pod  
      containers:  
        - name: springboot  
          image: spring-petclinic:3.1.0-SNAPSHOT
```

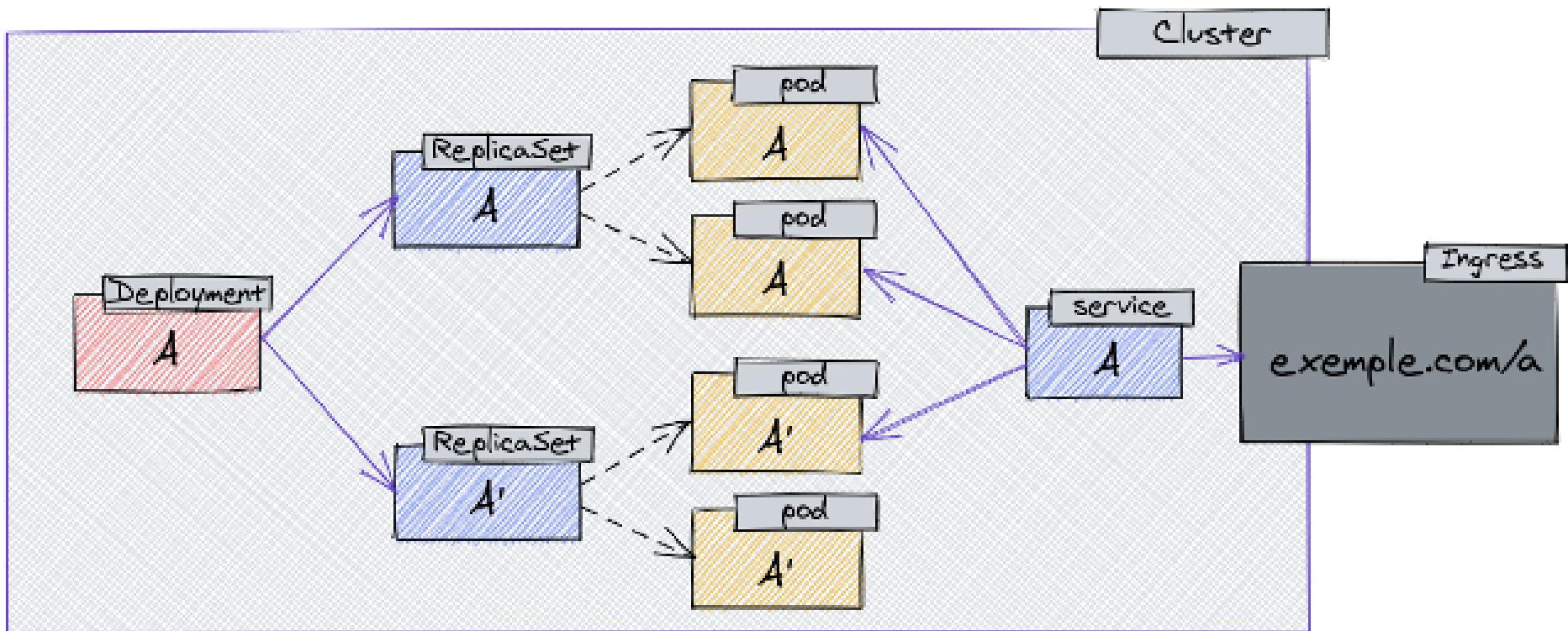
Kubernetes Rolling Update



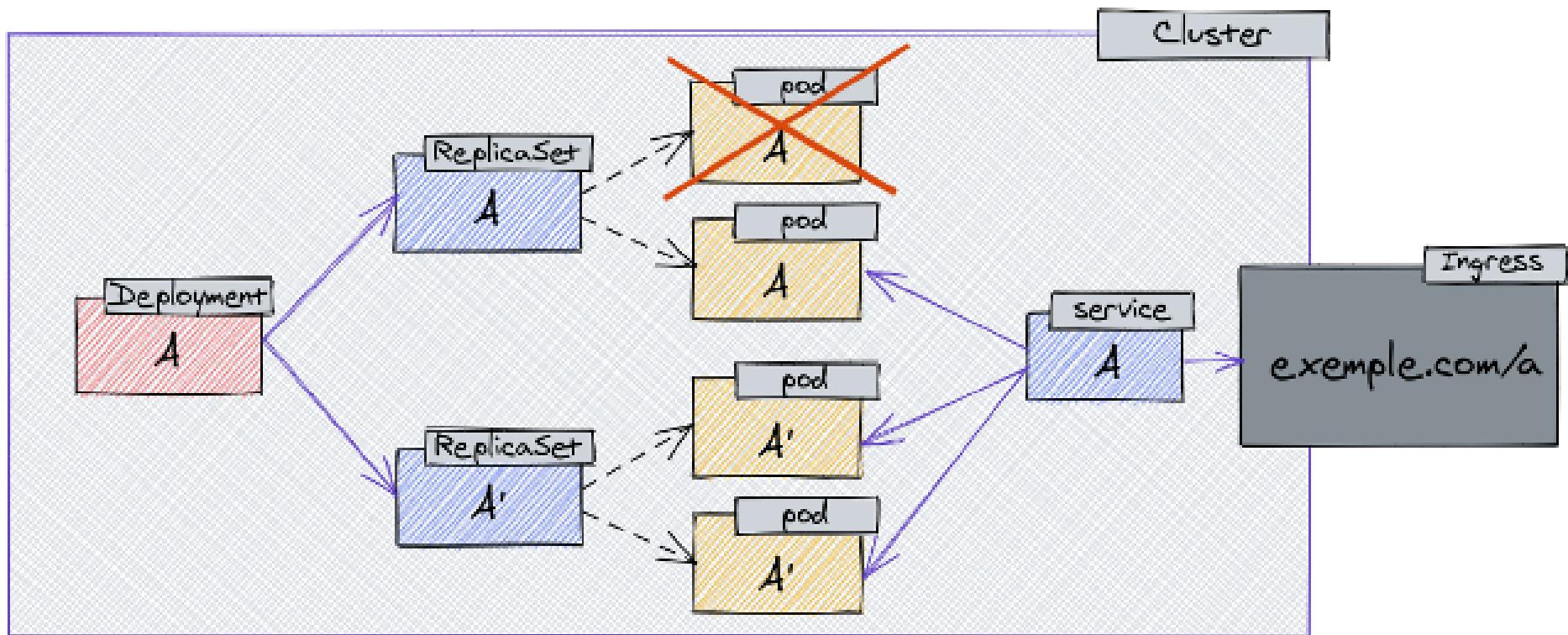
Kubernetes Rolling Update



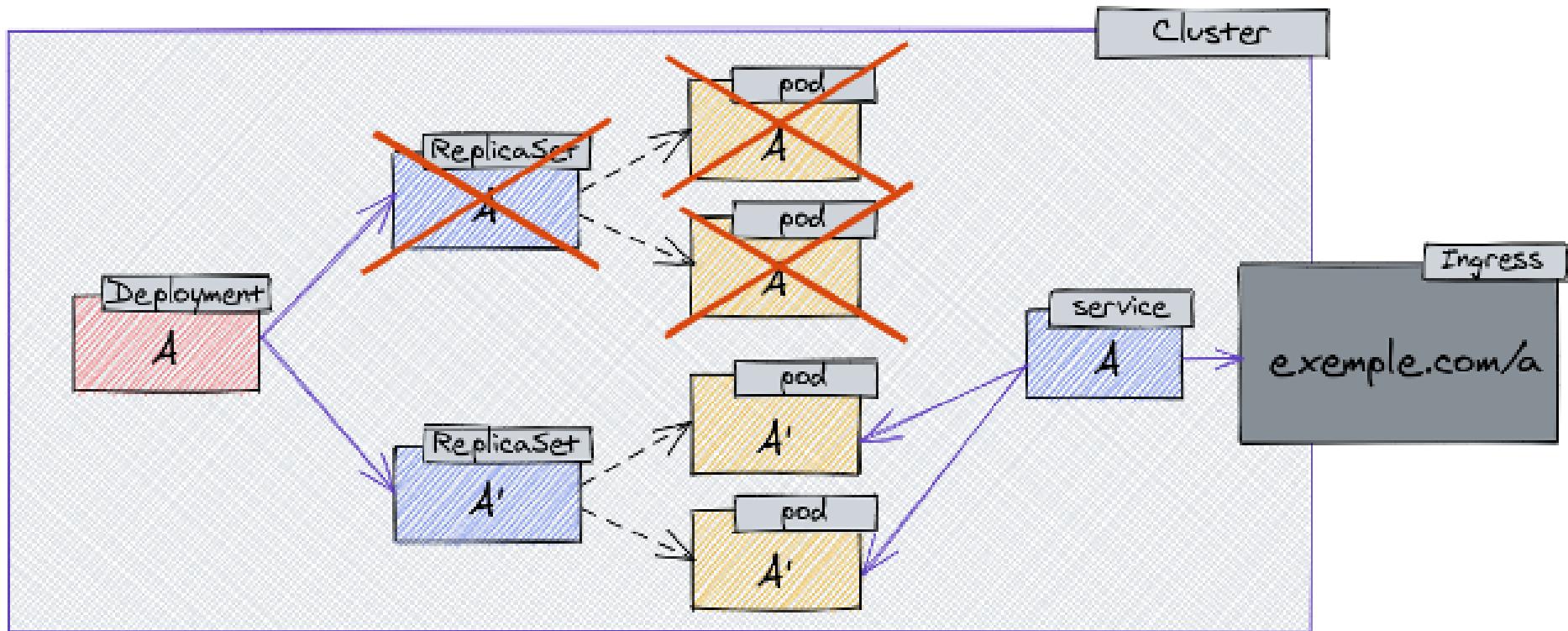
Kubernetes Rolling Update



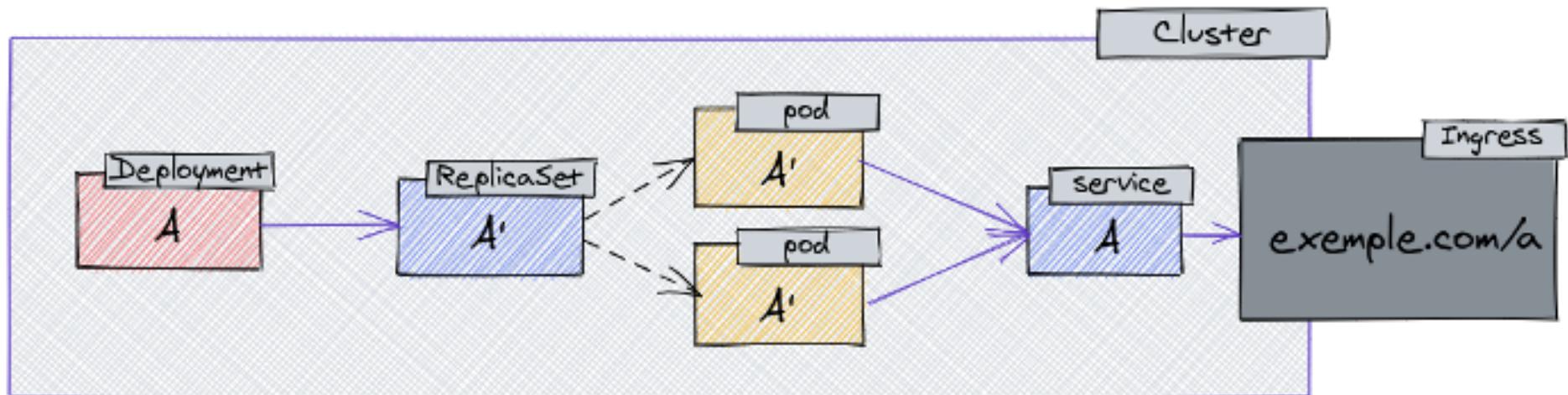
Kubernetes Rolling Update



Kubernetes Rolling Update



Kubernetes Rolling Update



```
kubectl get po # lister les pods
kubectl get po mypod -o yaml
kubectl describe po mypod # plus synthétique

kubectl get all -A # presque tous les objets sur tous les namespaces
kubectl get all,cm,secret,ing # ajouter configmaps/secret/ingress

kubectl config get-contexts
kubectl config use-context CONTEXT

kubectl apply -f fichier.yaml # ou une URL
kubectl delete po mypod

kubectl port-forward mypod 8080:80

kubectl get events -w # récupération des événements en temps réel, du namespace courant

kubectl logs -f mypod [-c CONTENEUR]
kubectl exec -it mypod [-c CONTENEUR] -- bash # ouvrir un shell ou lancer une commande

# deployment
kubectl rollout restart deployment mypod # détruit et recrée les pods, sans coupure de service
kubectl rollout history deployment mypod
kubectl rollout history deployment mypod --revision=4 | grep Image
kubectl rollout undo deployment/nginx-fpm [--to-revision=2]
```

- **Chacune de ces sondes peut-être utilisée pour savoir si :**
 - Le conteneur est en vie (**Liveness probe**)
 - Si KO, le conteneur est détruit et relancé (check ne doit pas dépendre de services ext)
 - L'application du conteneur est démarrée (**Startup Probe**)
 - Seulement **si long à démarrer**
 - Uniquement au start si le liveness est trop "sensible". Si OK, le liveness prend le relai
 - Le conteneur est prêt à travailler (**Readiness probe**)
 - Si KO, conteneur maintenu mais on ne lui envoie plus de traffic (check peut dépendre de services ext)
- **C'est plus qu'une bonne pratique !**
 - Permet au cluster de se réparer automatiquement (GitOps ++)
 - Et d'assurer les déploiements sans coupure

```
containers:
  - name: varnish
    image: ...

livenessProbe:
  exec:
    command:
      - varnishstat -1I MAIN.n_lru_nuked
  initialDelaySeconds: 30
  timeoutSeconds: 5

readinessProbe:
  httpGet:
    path: /readiness
    port: 8080
  initialDelaySeconds: 30
  timeoutSeconds: 5
```

TP 6

- **Personnalisation de manifests K8s**
 - Outil de contextualisation d'un ensemble de ressources Kubernetes
 - Permet de patcher vos ressources au moment du déploiement
 - Intégré nativement depuis Kubernetes 1.14
 - Avec la commande **kubectl apply -k**
 - <https://kustomize.io/>
- **Rien à voir avec Helm, plus simple, et parfois bien suffisant**
 - De plus en plus utilisé avec Helm
 - <https://itnext.io/helm-is-not-enough-you-also-need-kustomize-82bae896816e>
 - <https://blog.container-solutions.com/using-helm-and-kustomize-to-build-more-declarative-kubernetes-workloads>



- Démarrage par la création d'un répertoire de base
- Contenant des ressources standards Kubernetes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 3
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
```

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
```

- **Un fichier kustomization.yaml**

```
resources:  
- deployment.yaml  
- service.yaml
```

- **On obtient ce type de structure de fichiers**

```
base  
| deployment.yaml  
| kustomization.yaml  
| service.yaml
```

Kustomize exemple de patch

```

└── base
    └── deployment.yaml
    └── kustomization.yaml
    └── service.yaml
└── dev
    └── kustomization.yaml
    └── lower_memory.yaml
    └── lower_replicas.yaml

```

```

resources:
- ./base

patches:
- path: lower_memory.yaml
- path: lower_replicas.yaml
  target:
    kind: Deployment
    name: my-nginx

```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
        - name: my-nginx
          resources:
            limits:
              memory: 128Mi

```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 1

```

TP 7

- **Un gestionnaire de paquets pour K8s**
 - Packager plusieurs ressources Kubernetes
 - Dans un seul artefact de déploiement : le **Chart**
- **Propose une gestion de dépendances et permet (au choix) :**
 - D'être utilisé en moteur de templating (sans objectif d'abstraction des ressources)
 - De rendre les ressources abstraites par l'intermédiaire d'un moteur de Templating
- **URLs**
 - <https://helm.sh/>
 - <https://artifacthub.io/>

- **Chart**
 - Un package de l'ensemble de ressources Kubernetes nécessaires
 - Deployment, ConfigMap, Secret, Service...
- **Release**
 - Une instance de **Chart** qui a été déployée
- **Dépôt**
 - Une collection de **Charts**
- **Template**
 - Un modèle de ressource **Kubernetes** s'appuyant sur **Golang** et **Sprig**

Helm

Exemple de Chart

```
.  
+-- sf  
|   +-- charts  
|   |   Chart.yaml  
|   +-- templates  
|       +-- deployment.yaml  
|       +-- _helpers.tpl  
|       +-- ingress.yaml  
|       +-- NOTES.txt  
|       +-- serviceaccount.yaml  
|       +-- service.yaml  
|       +-- tests  
|           └── test-connection.yaml  
+-- values.yaml  
+-- values_dev.yaml  
+-- values_prod.yaml  
  
4 directories, 11 files
```

```
1  apiVersion: v1  
2  kind: Service  
3  metadata:  
4      name: {{ include "sf.fullname" . }}  
5  🔍 labels:  
6      {{- include "sf.labels" . | nindent 4 }}  
7  spec:  
8      type: {{ .Values.service.type }}  
9      ports:  
10     - port: {{ .Values.service.port }}  
11         targetPort: http  
12         protocol: TCP  
13         name: http  
14  ➔ selector:  
15      {{- include "sf.selectorLabels" . | nindent 4 }}
```

TP 8

- **Déploiement continu pour K8s en mode pull**
 - <https://argoproj.github.io/cd>
 - Niveau de maturité “Graduated” à la Cloud Native Computing Foundation
- **pull : surveille une branche Git**
 - Déploie automatiquement à la moindre modification
 - Proche fonctionnellement de Flux
- **Développé par Akuity**
 - Interface graphique incluse



- Capture d'écran

The screenshot shows the ArgoCD UI interface. At the top, there are tabs for APP DETAILS, APP Diffs, SYNC, SYNC STATUS, HISTORY AND ROLLBACK, DELETE, and REFRESH. Below these are sections for APP HEALTH (Healthy) and CURRENT SYNC STATUS (Synced). The sync status is shown as "Sync OK" with a timestamp of "To 3.33.2 (3.33.2)" and a note about being last updated 11 seconds ago. The main area displays a hierarchical tree of application components under the root "argocd-dev-server". The components include "argocd-dev-server", "argocd-nginx-ingress", "argocd-dev-application-controller", "argocd-dev-lease-controller", "argocd-dev-secret-controller", "argocd-dev-service-controller", "argocd-dev-repository", and "argocd-cm-server". Each component has a status icon (green for healthy) and a deployment progress bar. A sidebar on the left contains icons for Log, App Health, Sync, and Refresh. On the right, there are icons for Application Details, Application Diffs, Sync, History and Rollback, Delete, Refresh, and Log Out. A large green box in the bottom right corner contains the text "TP 9".

- **Une traçabilité et une auditabilité accrues des changements**
 - Chaque changement est effectué par un commit
- **Le contrôle d'accès et les workflow des dépôts**
 - Maitrise de l'envoi des changements
- **Les déploiements en mode pull peuvent améliorer la sécurité**
 - en évitant d'avoir à donner des accès en production aux développeurs
- **Les outils de scan des vulnérabilités peuvent être intégrés dans la CI/CD**

Conclusion

- **Amélioration de la collaboration**
 - Entre les équipes de développement et d'exploitation.
- **Raccourcissement des cycles de développement**
 - Grâce à l'automatisation et l'intégration continue
- **Livraison plus rapide et plus fiable**
 - Des fonctionnalités aux utilisateurs
- **Réduction du risque d'erreurs et de problèmes**
 - Lors du déploiement en production
- **Meilleure réactivité face aux changements et aux incidents**

- Nécessite une maîtrise approfondie des outils de CI/CD et de gestion de configuration
- Peut être difficile à mettre en place pour les grandes entreprises avec des processus existants rigides
- Peut nécessiter une refonte complète de la façon dont les équipes de développement et d'exploitation travaillent ensemble
- Peut nécessiter une formation et un temps d'adaptation pour les membres de l'équipe