

# Cours Git

## Gestion de versions

Thierry Vaira

BTS SNIR

tvaira@free.fr © v0.1



# Sommaire

- 1 Présentation
- 2 Gestion de versions
- 3 Utilisation
- 4 Les branches
- 5 Git hébergé
- 6 Projet collaboratif
- 7 CLI vs IDE

# Présentation

**Git** est :

- un logiciel de **gestion de versions décentralisé** (DVCS)
- un logiciel libre créé par Linus Torvalds en 2005.
- Site officiel : <https://git-scm.com/>



C'est le logiciel de gestion de versions le plus populaire devant Subversion (svn) qu'il a remplacé avantageusement.

# Quelques ressources

- Manuel de référence : <http://book.git-scm.com/docs>
- Pro Git (fr) : <http://git-scm.com/book/fr/v2>
- Git Community Book (fr) : <https://alexgirard.com/git-book/>

# Système de gestion de versions (VCS)

- Le VCS (*Version Control System*) gère l'ensemble des versions (« **révision** » ou modification) d'un ou plusieurs fichiers (généralement en texte)
- Le VCS enregistre l'évolution d'un ensemble de fichiers au cours du temps dans un **historique**
- Les fichiers versionnés sont stockés sur un **dépôt** (*repository*)
- Essentiellement utilisée dans le **développement logiciel**, elle concerne surtout la gestion des codes source

Les différentes versions (ou révisions) sont nécessairement liées à travers des modifications : une modification est un ensemble d'ajouts, de modifications, et de suppressions de données.

# Mécanismes de base d'un VCS

La gestion de version repose sur deux mécanismes de base :

- un **calcul de la différence** entre deux versions
  - diff : Compare des fichiers ligne à ligne
  - patch : Utilise la différence entre deux fichiers pour passer d'une version à l'autre
- un **gestionnaire d'historique des diff** pour conserver les modifications

Le principe est le suivant : on passera de la version N à la version N+1 en appliquant une modification M. Un logiciel de gestion de versions applique ou retire ces modifications une par une pour fournir la version du fichier voulue.



# VCS vs DVCS

Un **système de gestion de version** ou **VCS** (*Version Control System*) :

- maintient l'ensemble des versions d'un logiciel ;
- conserve l'historique (les révisions successives) du projet dans un seul dépôt (*repository*) qui fait référence : possibilités de revenir en arrière, de voir les changements ;
- facilite la collaboration entre les intervenants : chacun travaille avec son environnement, plusieurs personnes travaillent sur les mêmes fichiers simultanément ;
- fournit des outils pour gérer le tout.

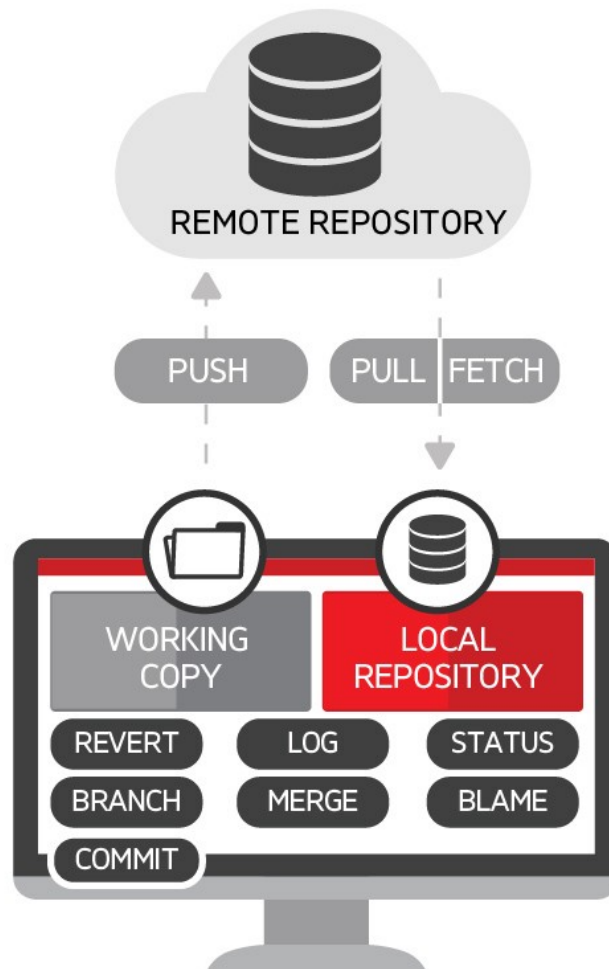
Un **DVCS** (*Distributed Version Control*) offre les mêmes services qu'un VCS sur une **architecture décentralisée** (ou distribuée).



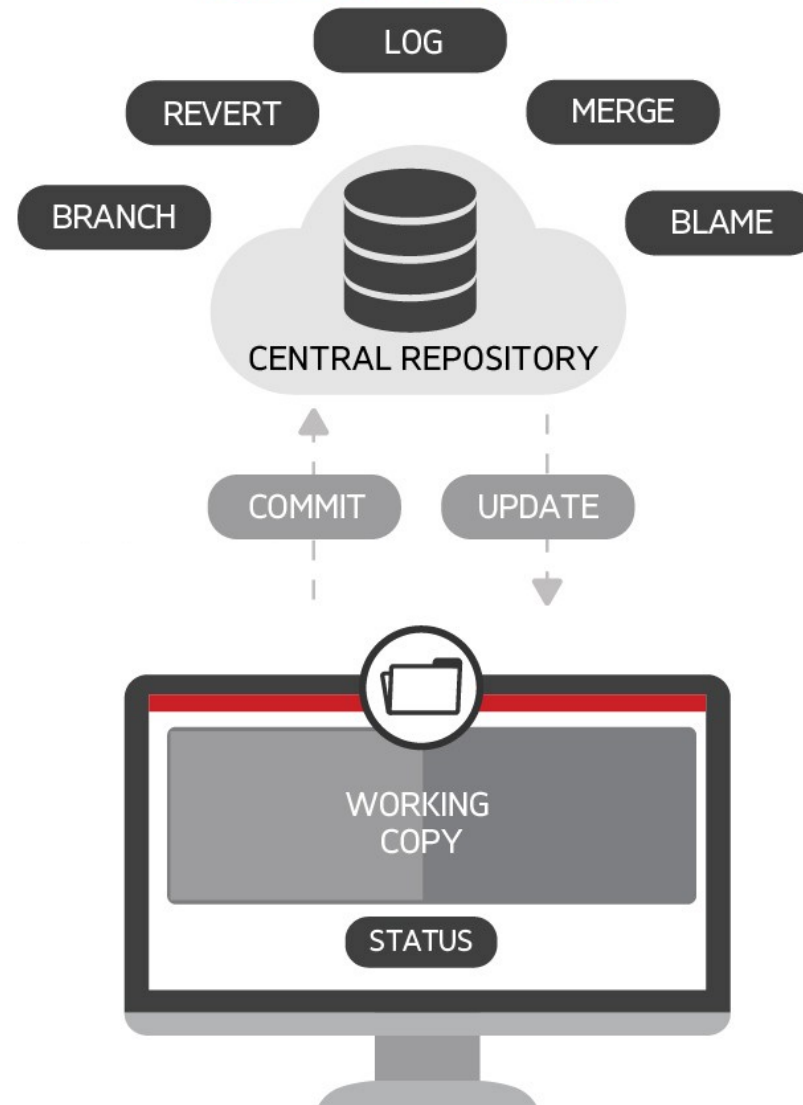
# Git vs Subversion

<https://www.tabnine.com/blog/svn-vs-git/>

## GIT



## SUBVERSION





# Fonctionnement interne

Le **dépôt Git** contient l'historique des instantanés (*commits*). C'est une base de "données" (d'objets) qui peut contenir **quatre types d'objets** :

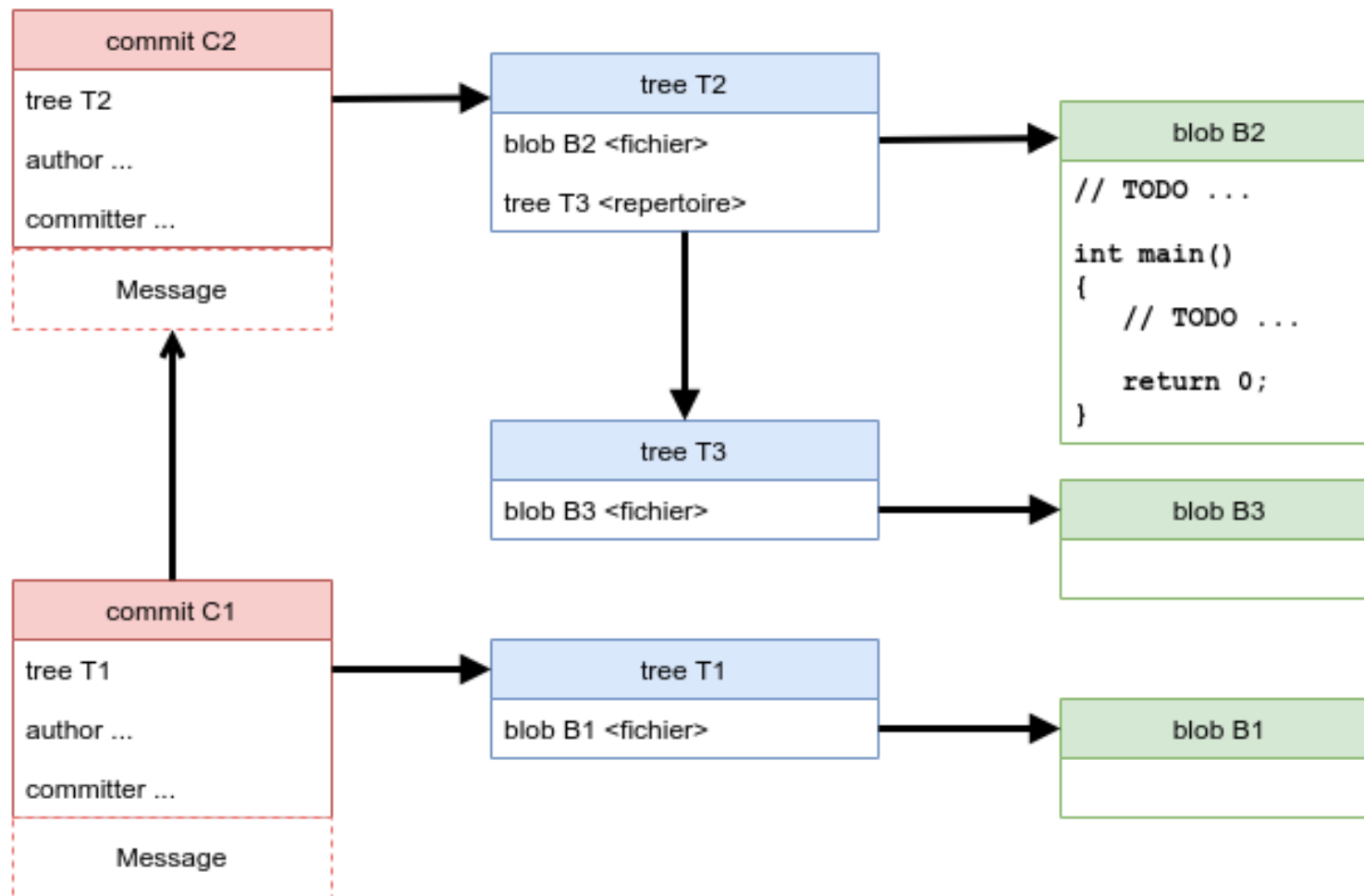
- L'objet **blob** (*binary large object*) représente le contenu d'un fichier (chaque révision d'un fichier = objet *blob* unique).
- L'objet **tree** décrit une arborescence de fichiers. Il est constitué d'une liste d'objets de type *blobs* (avec les informations tel que le nom du fichier et les permissions). Il peut contenir d'autres objets *trees* pour représenter les sous-répertoires.
- L'objet **commit** correspond à une arborescence de fichiers (*tree*) enrichie de métadonnées comme un message de description, le nom de l'auteur, etc.
- L'objet **tag** est une manière de nommer arbitrairement un *commit* spécifique pour l'identifier plus facilement.

Git utilise des **index** (une somme de contrôle calculée avec la fonction de hachage **SHA-1**) pour référencer les objets de la base.



# Historique

Git stocke un instantané (un *commit*) de la représentation de tous les fichiers du projet dans une structure hiérarchisée. L'instantané pointe également vers un ou plusieurs **objets commits parents**.



# Les commandes

- Git est un ensemble de commandes indépendantes dont les principales sont : `git init`, `git clone`, `git add`, `git status`, `git diff`, `git commit`, `git checkout`, `git merge`, `git log` etc.
- Liens :
  - <https://training.github.com/downloads/fr/github-git-cheat-sheet/>
  - <https://ndpsoftware.com/git-cheatsheet.html>
  - <https://www.julienkrier.fr/articles/git-cheat-sheet>
- Demander de l'aide : `man git`, `git help`, `git help <commande>`

# Premier pas

- Installation

## Sous GNU/Linux Ubuntu

```
$ sudo apt-get install git gitk  
$ git --version  
git version X.Y.Z
```

- Configuration

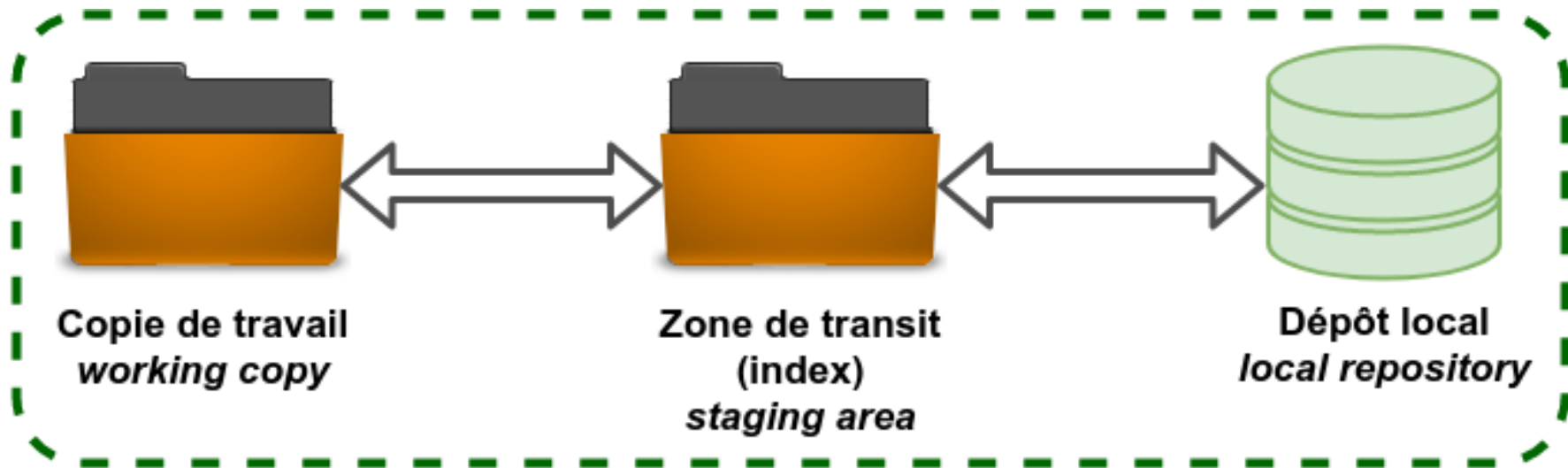
```
$ git config --global user.name "<votre nom>"  
$ git config --global user.email "<votre email>"  
$ git config --global core.editor vim  
$ git config --global color.diff auto  
$ git config --global color.status auto  
$ git config --global color.branch auto  
$ cat $HOME/.gitconfig  
$ git config --list
```

# En résumé

- La fonction principale de Git est de suivre les différentes versions d'un projet.
- Un projet est un ensemble de fichiers.
- Le *commit* (ou instantané) est l'élément central de Git.
- Un *commit* représente un ensemble cohérent de modifications sur le projet.

# Les zones

On distingue trois zones :



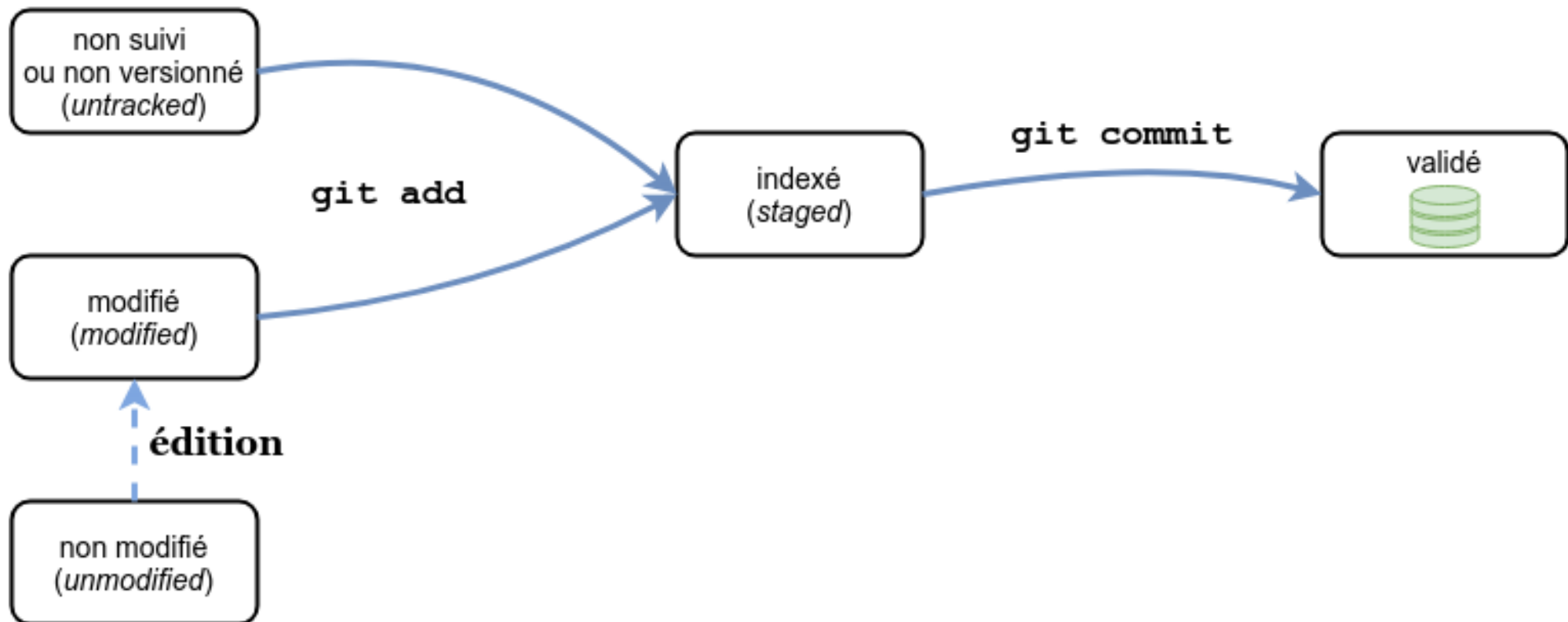
# Les différentes zones

- le **répertoire de travail** (*working directory*) : un répertoire du système de fichiers qui contient une extraction unique d'une version du projet pour pouvoir travailler
- l'**index** ou « zone de transit » (*staging area*) : un simple fichier (`.git/index`) qui stocke les informations concernant ce qui fera partie du prochain instantané (*commit*)
- le **dépôt local** (*local repository*) : un répertoire caché (`.git/`) qui stocke tout l'historique des instantannés (*commits*) et les méta-données du projet
- On peut considérer qu'il existe une quatrième zone nommée "remise" qui s'utilise avec la commande `git stash`.



# Les états d'un fichier

Les différents états d'un fichier :





# Les différents états d'un fichier

- **non suivi** ou non versionné (*untracked*) : aucun instantané existe pour ce fichier
- **non modifié** (*unmodified*) : non modifié depuis le dernier instantané
- **modifié** (*modified*) : modifié depuis le dernier instantané mais n'a pas été indexé
- **indexé** (*staged*) : modifié et ajouté dans la zone d'index
- **validé** (*committed*) : une version particulière d'un fichier

Pour obtenir l'état des fichiers du répertoire de travail (*working directory*), on utilise (très souvent) la commande `git status`

# Initialiser un dépôt git

- Création d'un répertoire

```
$ mkdir tp-git-sequence-1
mkdir: création du répertoire 'tp-git-sequence-1'

$ cd ./tp-git-sequence-1
```

- Initialisation d'un dépôt git

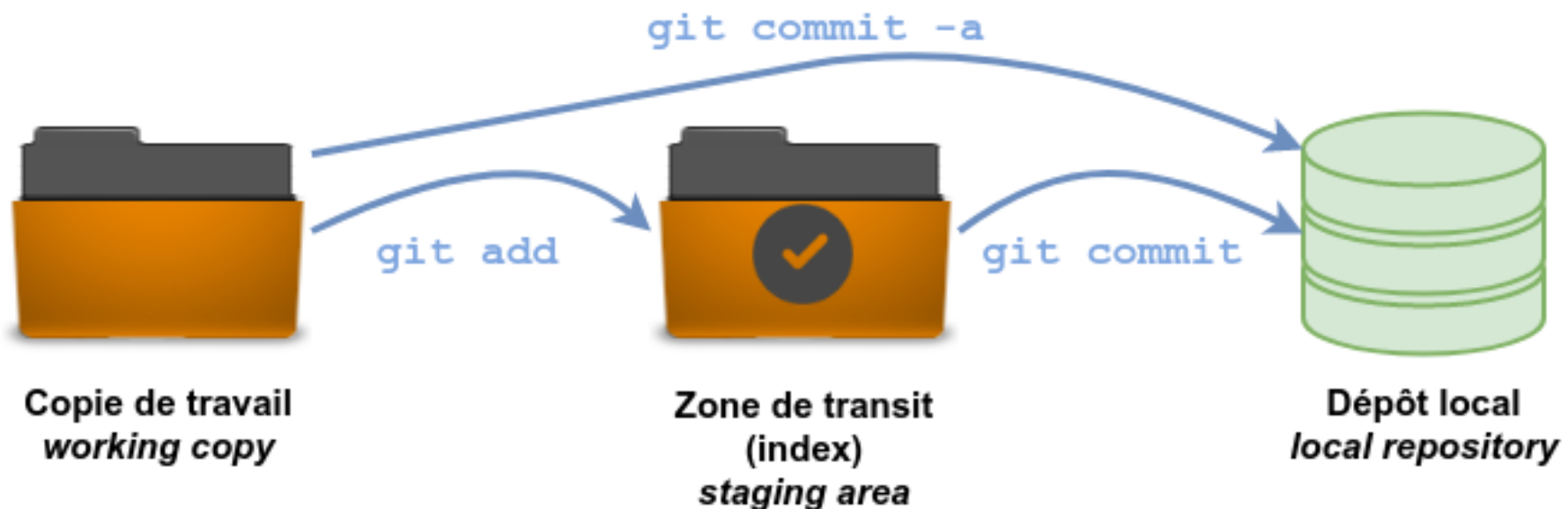
```
$ git init
Dépôt Git vide initialisé dans $HOME/tp-git-sequence-1/.git/

$ ls -al
...
drwxrwxr-x 7 tv tv 4096 juil. 28 10:58 .git

# Pour l'instant, aucun fichier n'est encore versionné.
```

# Travailler avec git

- on édite des fichiers dans le répertoire de travail (*working directory*) ;
- on indexe les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index (*staging area*) ;
- on valide les modifications, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans le dépôt local (*local repository*).



# Opérations sur les fichiers

- Ignorer des fichiers : il faut les ajouter dans un fichier spécial `.gitignore`

Pour nettoyer son répertoire de travail, on peut utiliser la commande `git clean` qui permet de supprimer les fichiers non-suivis qui ne sont pas ignorés.

- Effacer des fichiers : commandes `git rm` et `git rm --cached`
- Déplacer/Renommer des fichiers : commande `git mv`

# Visualiser des différences

La commande `git diff` (voir aussi `git difftool`) pour visualiser les lignes exactes qui ont été ajoutées, modifiées ou effacées :

- Voir les différences avec l'index

```
$ git diff
```

- Voir les différences avec le dernier *commit*

```
$ git diff --staged
```

La commande `git diff <commit>` sert à visualiser les modifications présentes dans le répertoire de travail par rapport au `<commit>` indiqué.



# Visualiser l'historique

Après avoir créé plusieurs instataneés (*commits*), il est possible de consulter l'historique avec la commande `git log`. C'est une commande importante et puissante disposant de nombreuses options :

- `git log -<nombre>` Limiter le nombre de *commits*
- `git log --oneline` Affiche chaque *commit* sur une seule ligne
- `git log -p` Affiche la différence complète de chaque *commit*
- `git log --graph --decorate` Affiche sous forme de graphe
- `git log --stat` Affiche avec des statistiques
- `git log -- <fichier>` Affiche uniquement les *commits* du fichier
- `git log <depuis>..<jusqu'à>` Affiche les validations qui se produisent entre deux *commits* en utilisant une référence

Voir aussi :

- `git blame <fichier>` Affiche qui a modifié le fichier et quand
- `git show <objet>` Affiche un objet du dépôt



# Annuler des actions

- `git commit --amend` pour modifier le dernier *commit*
- `git reset HEAD <fichier>` pour désindexer un fichier
- `git checkout -- <fichier>` pour annuler les modifications dans la copie de travail
- `git revert` pour inverser un *commit*
- Voir aussi :
  - `git reset --soft HEAD~` pour annuler le dernier `git commit`
  - `git reset --mixed HEAD~` pour annuler le dernier `git add` et `git commit`
  - `git reset --hard HEAD~` pour annuler les modifications dans la copie de travail et le dernier `git add` et `git commit`

# Étiqueter des versions

Git utilise deux types principaux d'étiquettes (*tags*) :

- **légère** : un pointeur sur un *commit* spécifique
- **annotée** (avec l'option `-a`) : un objet *tag* dans la base de données

```
$ git tag -a 1.0 -m 'La version 1.0'
```

```
$ git tag  
1.0
```

```
$ git show 1.0
```

Il est possible d'étiqueter après coup en spécifiant le *commit* : `git tag -a v1.2 <commit>`



# Publier une version

- Pour publier une version, il est nécessaire de créer une archive à partir d'un *commit* (généralement une étiquette de version).
- La commande dédiée à cette action est `git archive` :

```
# Modèle :  
# git archive --prefix=src-directory-name tag --format=zip > 'git describe master  
# .zip  
  
$ git archive --prefix='tp-git-sequence-1-vaira/' 1.0 | gzip > tp-git-sequence-  
-1-vaira.tar.gz  
$ git archive --prefix='tp-git-sequence-1-vaira/' 1.0 --format=zip > tp-git-  
sequence-1-vaira.zip
```

Il est possible de copier un dépôt avec la commande : `cp -Rf <source> <destination>`. Git fournit surtout la commande `git clone` pour cela.



# Utiliser le mode interactif

Git propose quelques scripts qui "guident" les opérations en ligne de commande avec l'option `-i` ou `--interactive`. Le mode interactif s'utilise principalement avec les commandes :

- `git add --interactive` : pour choisir les fichiers ou les parties d'un fichier à incorporer à un *commit*
- `git clean --interactive` : pour choisir les fichiers qui seront supprimés du répertoire de travail
- `git rebase --interactive` : pour choisir les *commits* à "rejouer"
  - Réordonner les *commits*
  - Écraser un *commit*
  - Diviser un *commit*
  - Supprimer un *commit*



# En résumé

- Éditer des fichiers : `vim` ou un EDI/IDE
- Ajouter les changements : `git add <fichier>`
- Valider les changements : `git commit -m "Message"`

Les commandes que l'on utilise tout le temps :

- `git status`
- `git log ...`

# Les branches

- Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne.
- Une branche représente une **ligne de développement indépendante**.
- Mais techniquement, une branche dans Git est simplement un pointeur déplaçable vers un *commit*.
- À chaque validation, le pointeur de la branche avance automatiquement pour pointer vers le dernier des *commits* réalisés.
- Le pointeur de la tête de la branche actuelle se nomme HEAD

La branche `master` ou `main` n'est pas une branche spéciale. Elle est identique à toutes les autres branches. La seule raison pour laquelle chaque dépôt en a une est que la commande `git init` la crée par défaut.

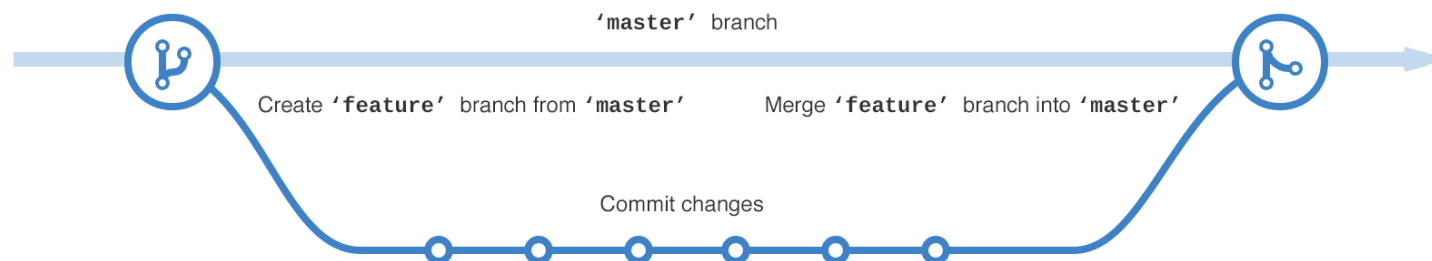
# Créer et basculer de branche

- Créer une nouvelle branche : `git branch <nom-branche>` (cela crée simplement un nouveau pointeur vers le *commit* courant)
- Basculer sur une branche existante : `git checkout <nom-branche>` (cela déplace HEAD pour le faire pointer vers la branche)
- En une seule commande : `git checkout -b <nouvelle-branche>`

Il est important de noter que lorsque l'on change de branche avec Git, les fichiers du répertoire de travail sont modifiés. Si la copie de travail ou la zone d'index contiennent des modifications non validées qui sont en conflit avec la branche à extraire, Git n'autorisera pas le changement de branche. Le mieux est donc d'avoir une copie de travail propre au moment de changer de branche.

# Fusionner une branche

- Une fois le travail réalisé (terminé et testé) dans la branche, il est prêt à être **fusionné** dans la branche `master`. On réalise ceci au moyen de la commande `git merge`.



- À présent que le travail a été fusionné, on n'a plus besoin de la branche. On peut la supprimer avec la commande `git branch -d <nom-branch>`

# Travailler avec les branches

Cela permet :

- de gérer plusieurs branches en parallèle et ainsi de cloisonner les travaux et d'éviter ainsi de mélanger des modifications du code source qui n'ont rien à voir entre elles.
- de conserver une version du logiciel prête à être livrée à tout instant puisqu'on ne fusionne que lorsque le développement d'une branche est bien terminé.

Dans Git, créer, développer, fusionner et supprimer des branches plusieurs fois par jour est un travail "normal". Un dépôt Git peut maintenir de nombreuses branches de développement.

# Différentes types de branches













On peut distinguer plusieurs types de branches :

- les **branches au long cours** (**permanentes**) : ce sont des branches ouvertes en permanence pour les différentes phases du cycle de développement.
- les **branches thématiques** (**temporaires**) : une branche thématique est une branche ayant une courte durée de vie créée et utilisée pour une **fonctionnalité** ou une **tâche particulière** (un correctif par exemple). On y réalise quelques *commits* et on supprime la branche immédiatement après l'avoir fusionnée dans la branche principale. Les branches thématiques sont utiles quelle que soit la taille du projet.
- Voir aussi : les branches de suivi

De nombreux développeurs travaillent avec Git en utilisant une méthode de développement basée sur les branches (par exemple Gitflow).



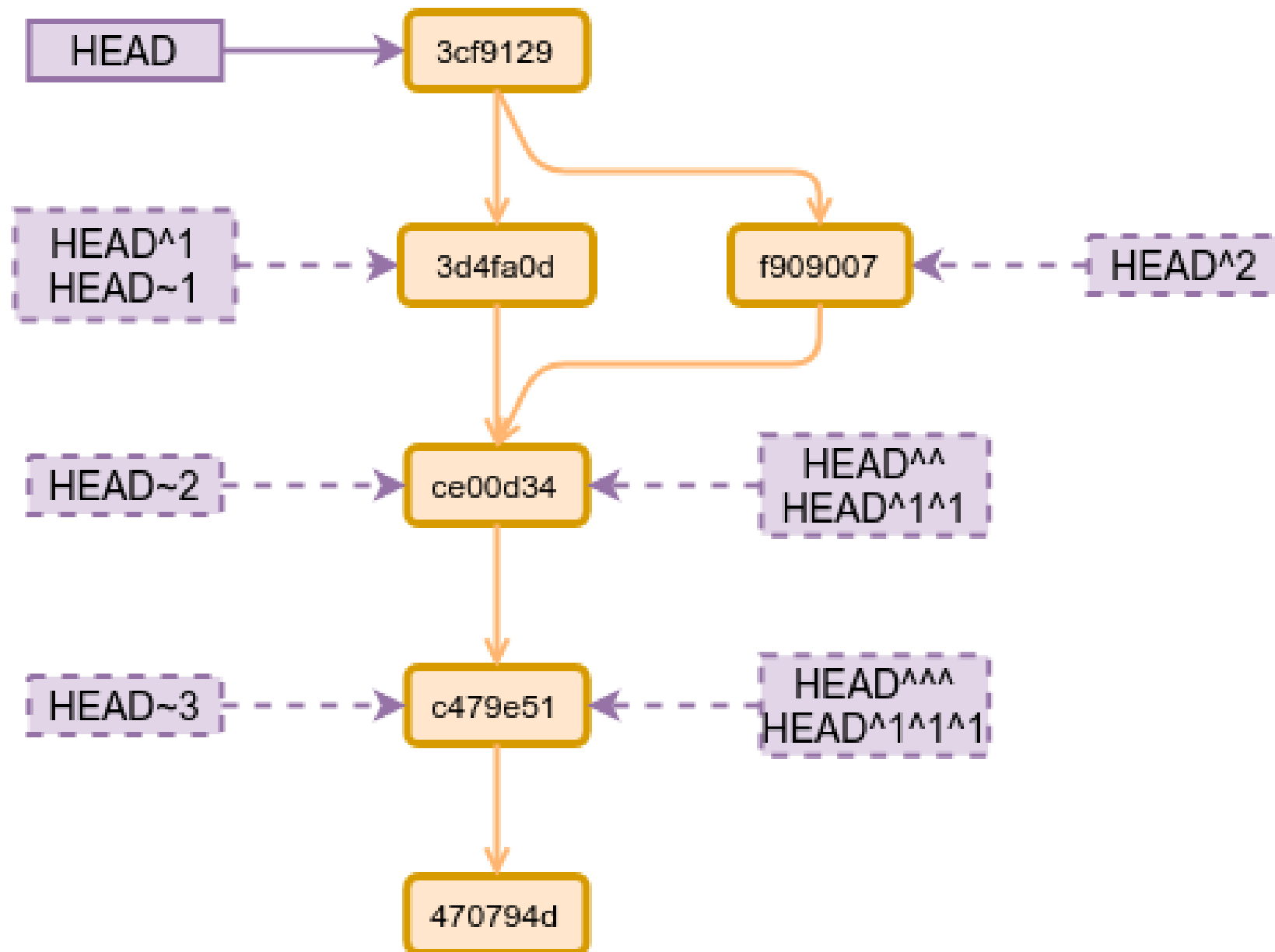
# Graphe d'historique

Graph	Description	Date	Author	
	 <b>main</b> Merge branch 'modification-fonction' into main	16 Aug 2021 16:18	tvaira	3cf9129d
	 modification-fonction Modification de la fonction afficherBienvenue() qui affiche le message "Bienvenu..."	16 Aug 2021 15:51	tvaira	f9090073
	 correctif-readme Modification README.md	16 Aug 2021 15:34	tvaira	3d4fa0d0
	 origin/correctif-readme  origin/main  1.1 Modification afficherBienvenue en afficherMessage	15 Aug 2021 11:20	tvaira	ce00d344
	Renommage README.md	11 Aug 2021 17:14	Thierry VAIRA	c479e51a
	Modification du fichier README	11 Aug 2021 17:04	tvaira	470794d3

# Le pointeur de référence HEAD

- HEAD est une référence symbolique pointant vers l'endroit (un *commit*) où l'on se trouve dans l'historique. Si on fait un *commit*, HEAD se déplacera.
- HEAD~ désigne le premier ancêtre de la pointe de la branche actuelle. HEAD~ est l'abréviation de HEAD~1. HEAD~n désigne le n-ième ancêtre.
- HEAD^ désigne le premier parent immédiat de la pointe de la branche actuelle. HEAD^ est l'abréviation de HEAD^1. HEAD^2 désigne le deuxième parent lorsqu'il y a un *commit* de fusion.
- Pour un *commit* avec un seul parent, HEAD~ et HEAD^ signifient la même chose.

# Exemple : HEAD



# En résumé

- Créer une branche thématique et basculer dessus : `git branch <branche>` puis `git checkout <branche>` (ou `git checkout -b <branche>`)
  - Éditer des fichiers : `vim` ou un EDI/IDE
  - Ajouter les changements : `git add <fichier>`
  - Valider les changements : `git commit -m "Message"`
- Basculer sur la branche principale et fusionner la branche thématique : `git checkout master` puis `git merge <branche>`
- Supprimer la branche thématique : `git branch -d <branche>`

Les commandes que l'on utilise tout le temps :

- `git status`
- `git log ...`
- `git branch ...`



# Git hébergé

Il est possible d'héberger des projets Git sur un site externe dédié à l'hébergement. Quelques hébergeurs :

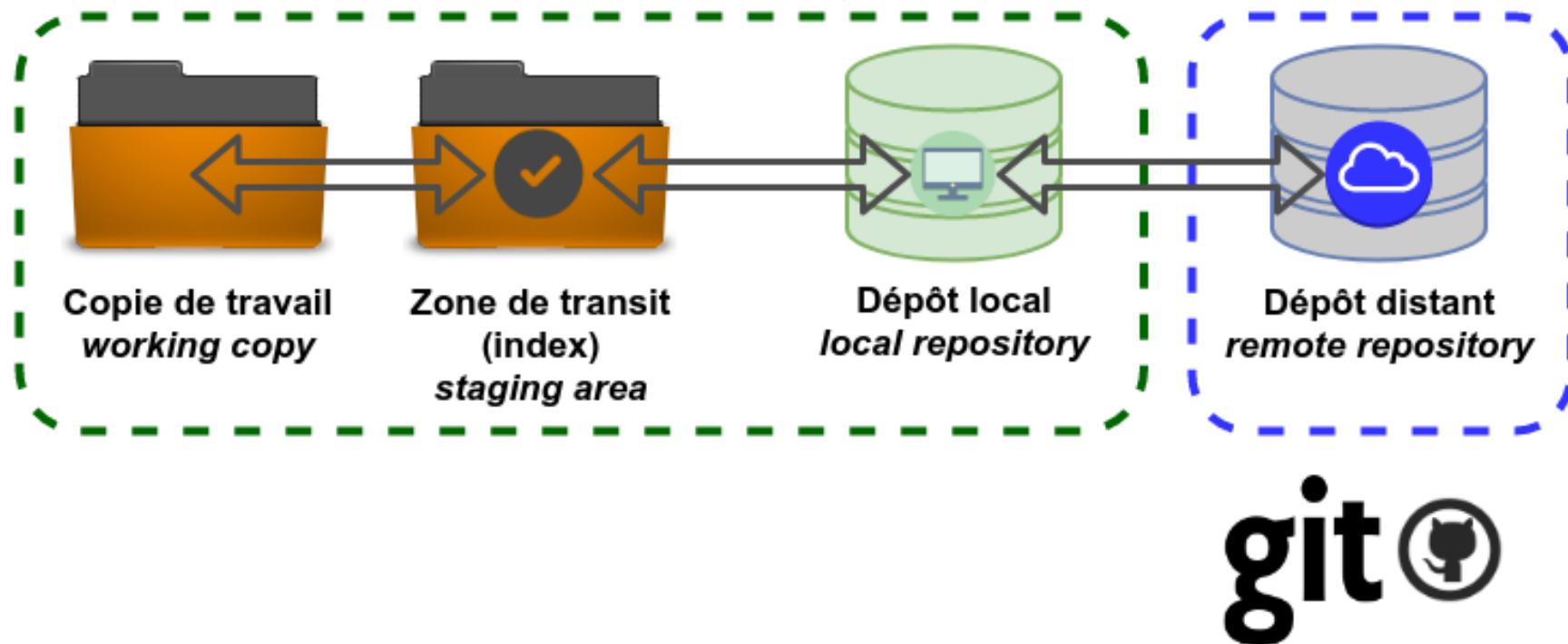
- **GitHub** est un service web d'hébergement (lancé en 2008) et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git. Site officiel : <https://github.com/>
- **GitLab** est un logiciel libre de forge basé sur Git proposant les fonctionnalités de wiki, un système de suivi des bugs, l'intégration continue et la livraison continue. Site officiel : <https://about.gitlab.com/>
- **Bitbucket Cloud** est un service web d'hébergement et de gestion de développement logiciel utilisant le logiciel de gestion de versions Git. Site officiel : <https://bitbucket.org/>

Liste : <https://git.wiki.kernel.org/index.php/GitHosting>



# Notion de dépôt distant

Un **dépôt distant** est un dépôt hébergé sur un serveur, généralement sur Internet.

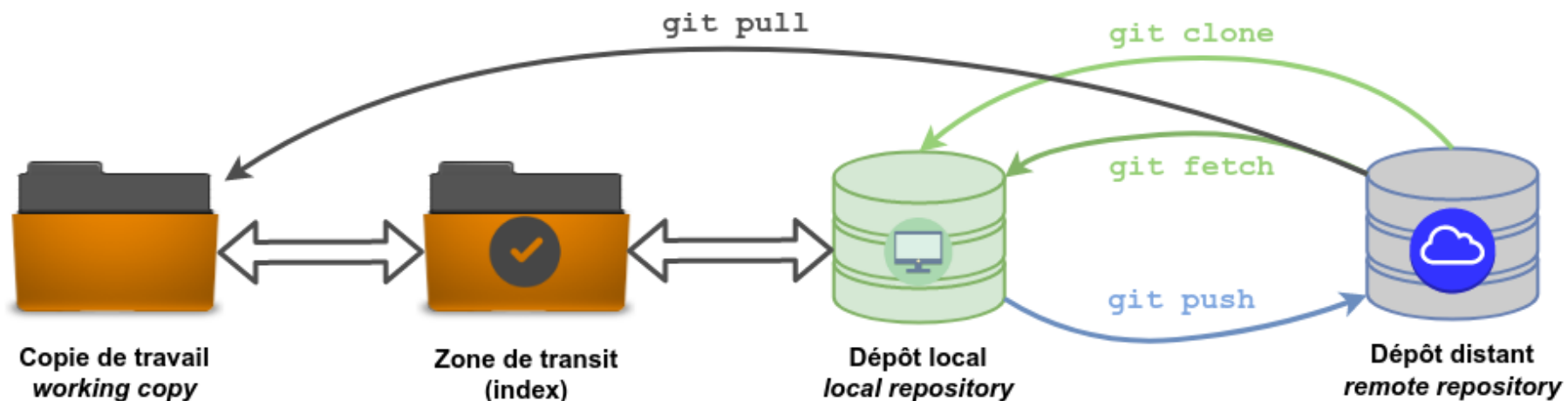


Un dépôt distant peut servir à la sauvegarde et/ou au partage du code d'un projet.

# Interagir avec un dépôt distant

Des commandes spécifiques seront utilisées pour synchroniser les dépôts local et distant :

- `git clone` permet d'obtenir une copie d'un dépôt Git existant.
- `git push` publie ("pousse") les nouveaux *commits* du dépôt local sur le dépôt distant.
- `git fetch` récupère l'ensemble des *commits* présents sur le le dépôt distant et met à jour le dépôt local. Elle ne modifie pas le répertoire de travail.
- `git pull` consiste essentiellement en `git fetch` suivi par `git merge`. Le répertoire de travail peut donc être modifié.



# Interagir avec GitHub

Il est possible d'interagir avec un dépôt sur GitHub de plusieurs manières :

- L'URL d'accès au dépôt en **SSH** sera de la forme :

`git@github.com:<utilisateur>/<depot>.git`

`https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/connecting-to-github-with-ssh`

- L'URL d'accès au dépôt en **HTTPS** sera de la forme :

`https://github.com/<utilisateur>/<depot>.git`

`https:`

`//docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token`

- La commande `gh` permet l'utilisation de GitHub en la ligne de commande (CLI) : `gh repo clone <utilisateur>/<depot>.git`

`https://cli.github.com/`





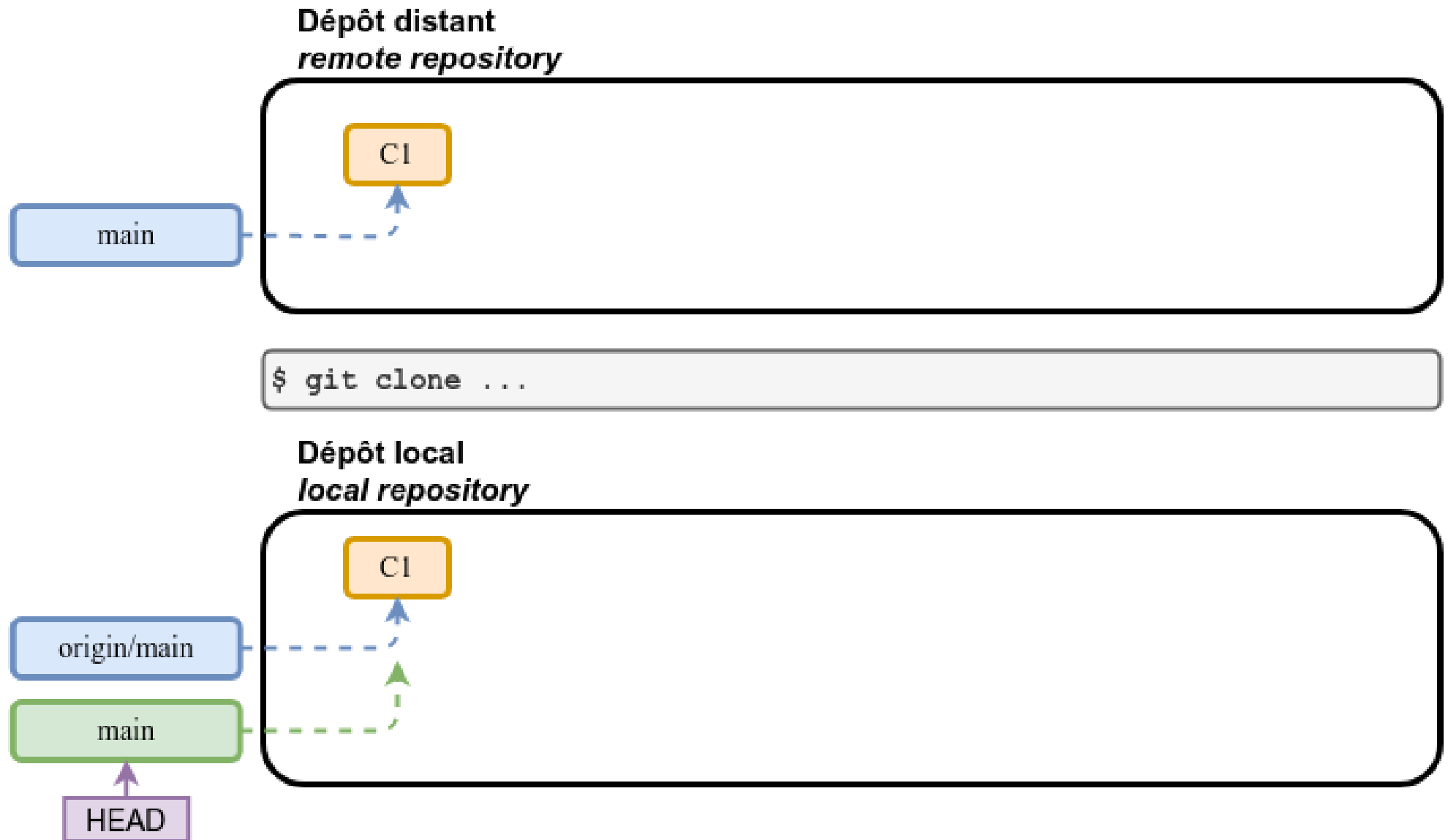
# Cloner un dépôt distant

```
$ git clone URL
```

La commande `git clone` effectuera les actions suivantes :

- crée un répertoire du nom du dépôt existant, initialisé avec un répertoire `.git` à l'intérieur,
- nomme automatiquement le serveur distant (*remote*) `origin`,
- tire l'historique,
- crée un pointeur sur l'état actuel de la branche `main` et l'appelle localement `origin/main`
- crée également une branche locale `main` qui démarre au même endroit que la branche `main` distante

# Cloner un dépôt distant



# Associer un dépôt distant existant

Un dépôt distant (*remote repository*) doit exister sur GitHub par exemple :

```
$ mkdir <depot>
$ cd ./<depot>
$ git init

$ git remote add origin git@github.com:<utilisateur>/<depot>.git
$ git branch -M main
$ git push -u origin main

$ git remote -v
origin git@github.com:<utilisateur>/<depot>.git (fetch)
origin git@github.com:<utilisateur>/<depot>.git (push)
```

# Branche de suivi

Une **branche de suivi** (*tracking branch*) est une branche locale qui est en relation directe avec une branche distante (*upstream branch*).

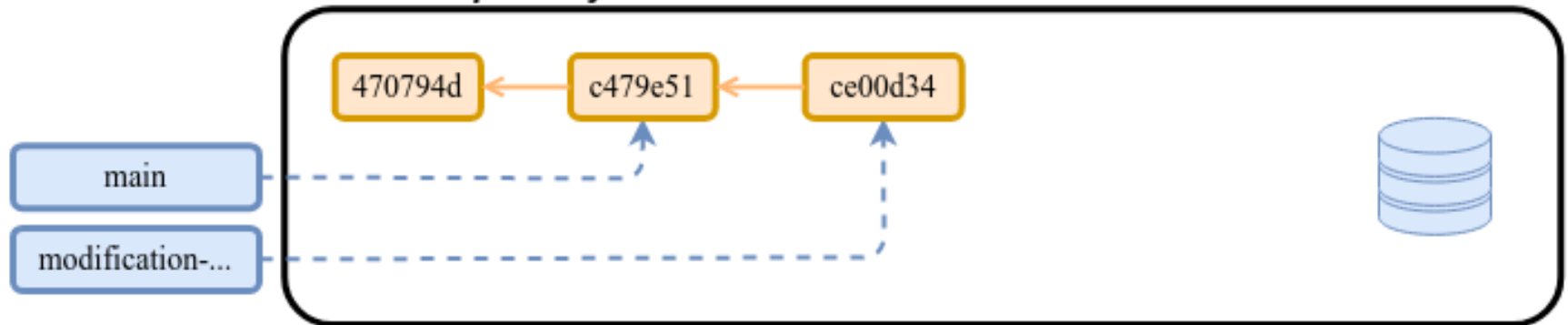
Les branches de suivi peuvent servir :

- à sauvegarder son travail sur la branche dans un dépôt distant
- partager son travail sur la branche avec d'autres développeurs

Voir aussi les options `--track`, `-u` ou `--set-upstream-to`, `--set-upstream` des commandes `checkout`, `branch` et `push`.

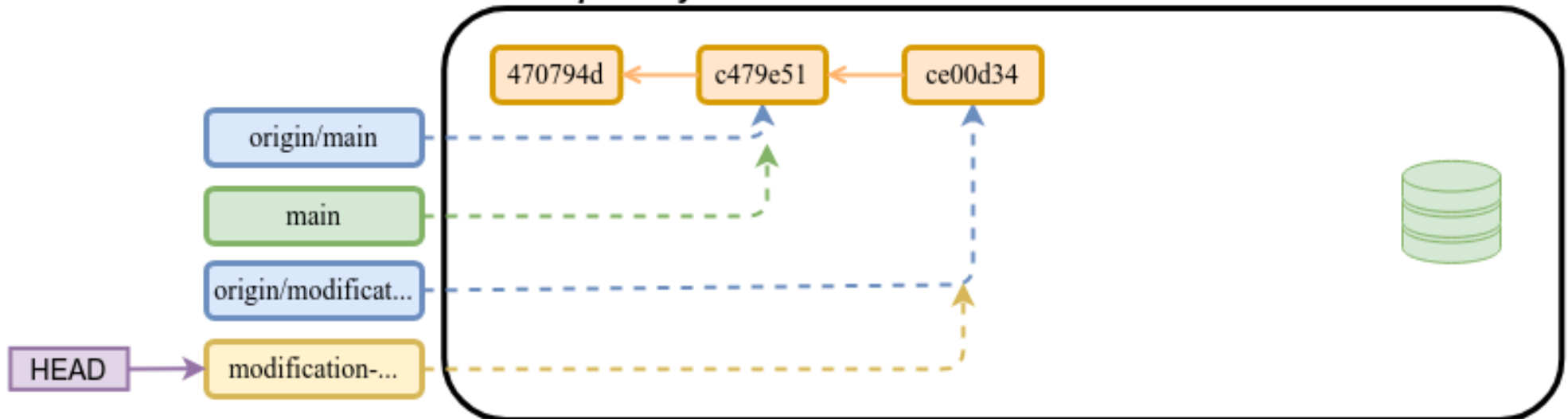
# Branche de suivi

Dépôt distant  
*remote repository*



```
$ git push --set-upstream origin modification-fonction
```

Dépôt local  
*local repository*



alle  
rignon

# Pull Request et Révision de code

**Pull Request** peut être traduit par « **Proposition de révision** » (PR) : c'est-à-dire une demande de modification ou de contribution.

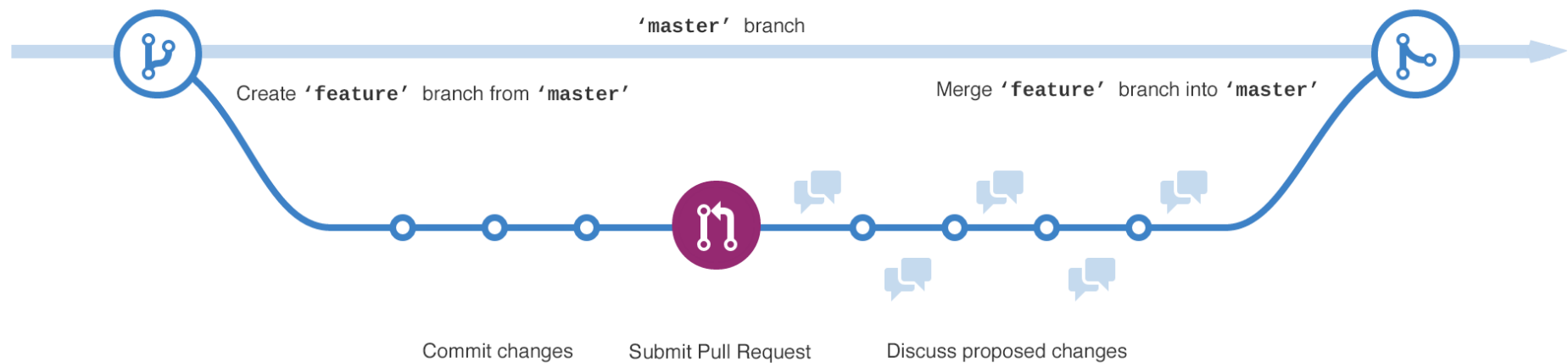
Les *Pull Requests* sont :

- une fonctionnalité facilitant la collaboration des développeurs sur un projet.
- un mécanisme permettant à un développeur d'informer les membres de l'équipe qu'il a terminé un « travail » (une fonctionnalité, une version livrable, un correctif, ...) et de proposer sa contribution au dépôt central.
- une notification aux développeurs pour qu'ils révisent le code puis le fusionnent (*merge*).

Pendant une **révision de code**, les développeurs peuvent discuter de la fonctionnalité (commenter le code, poser des questions, ...) et proposer des adaptations de la fonctionnalité en publiant des *commits* de suivi.



# Pull Request



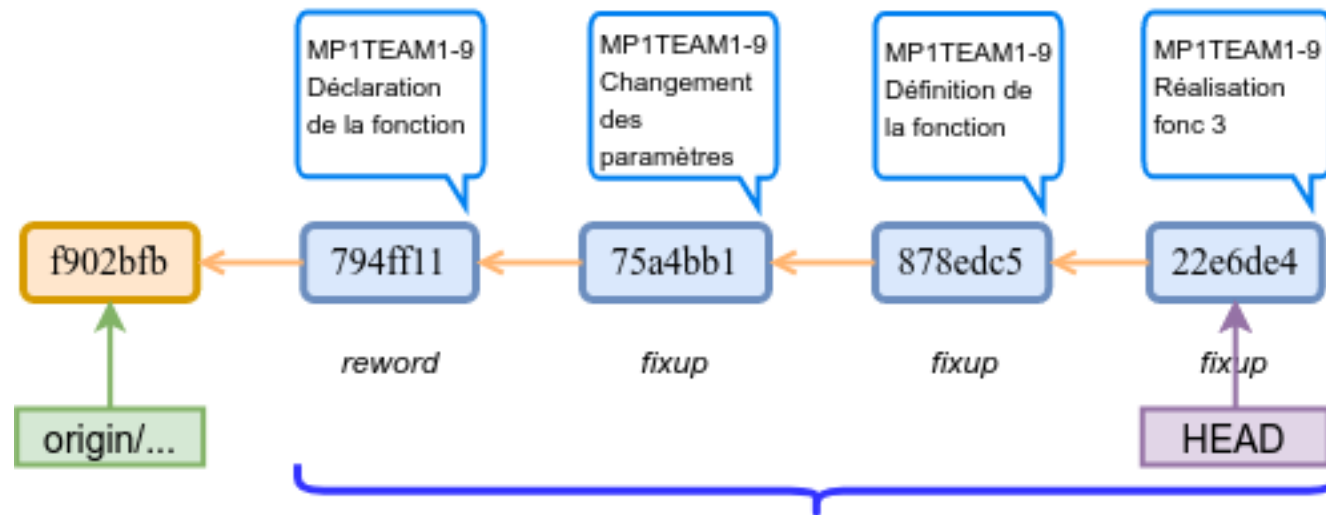
# Travail collaboratif

Attention :

- Nettoyer son historique local avant de publier en effectuant un rebasage interactif avec `git rebase -i @upstream`
- Travailler à plusieurs sur une branche de fonctionnalité : faire `git pull --rebase` sur une branche de suivi obsolète
- Supprimer (si besoin) toutes ses modifications et commits locaux et récupérer un dépôt distant « propre » :  
`git fetch origin && git reset --hard origin/main`



# Rebasage interactif



Avant de publier

```
git rebase -i @{upstream}
```

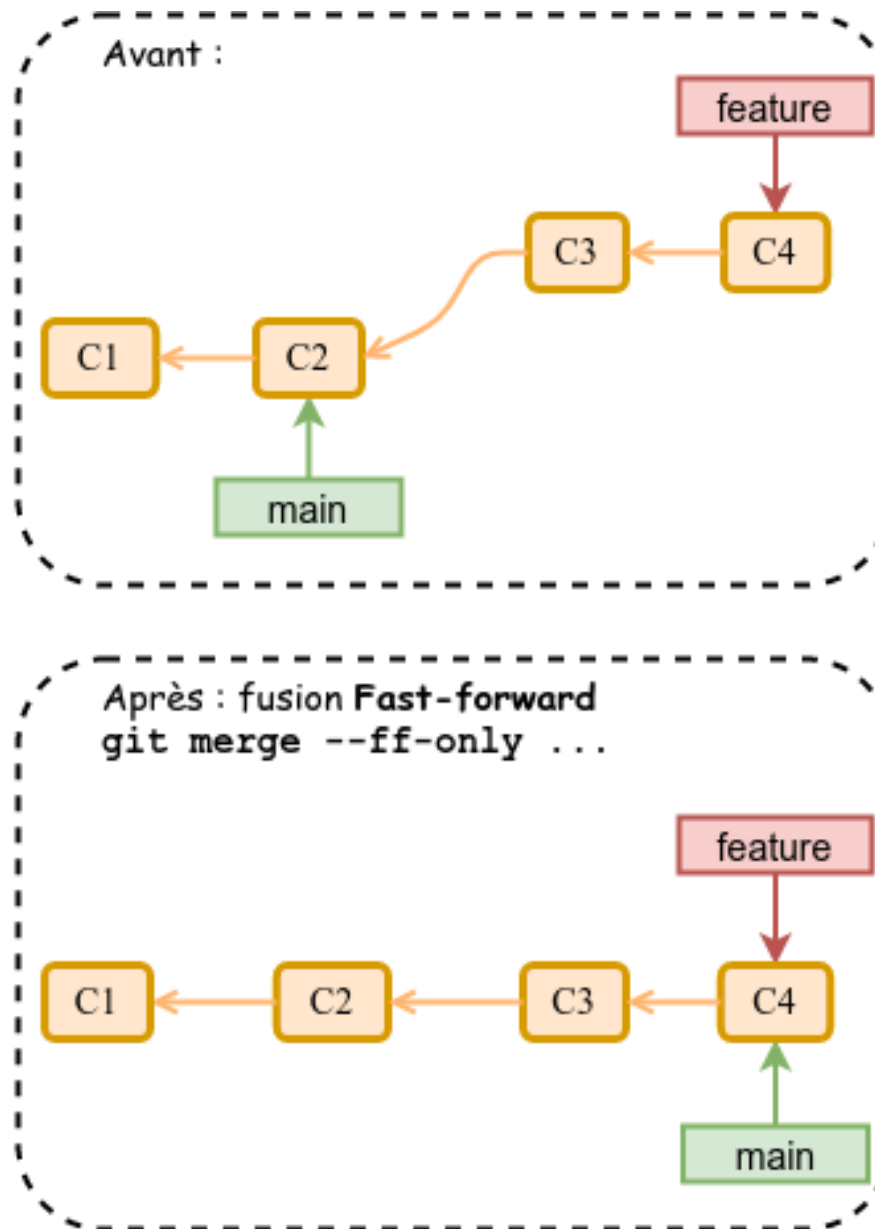


# Stratégies de fusion

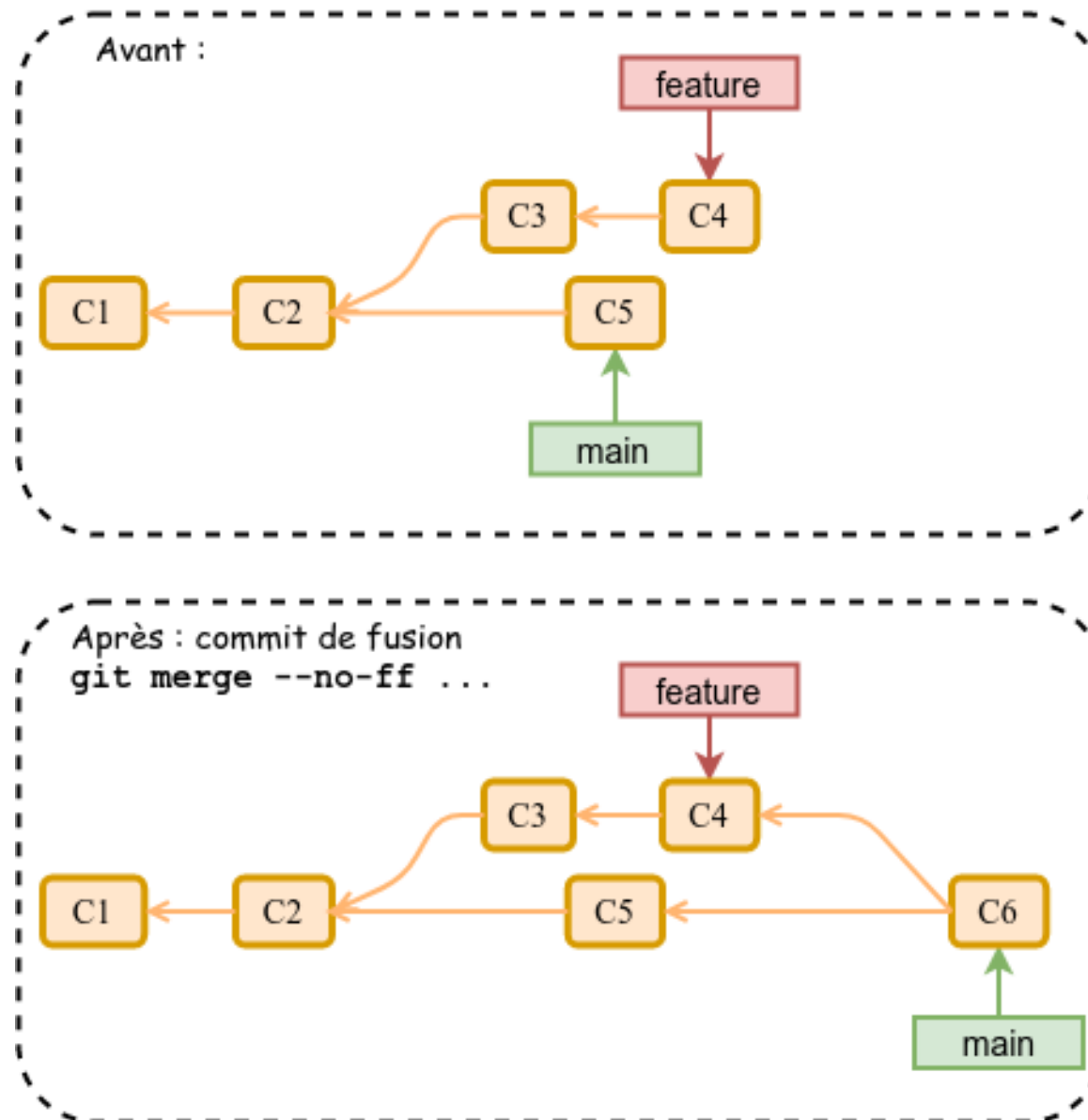
- **Avance rapide (*Fast Forward*)** : c'est la fusion utilisée **par défaut** par `git merge` si c'est possible. Git déplace (vers l'avant) les *commits* de la branche *feature* vers la branche destination *main* si il n'y a pas eu de nouveaux *commits* sur cette branche. On peut réaliser cette fusion avec l'option `--ff-only`.
- **Commit de fusion (*merge commit*)** : lorsque l'historique de développement a divergé, `git merge` réalise une fusion à trois sources (*three-way merge*) en utilisant les deux *commits* au sommet des deux branches (C4 et C5) ainsi que leur plus proche ancêtre commun (C2) pour créer un nouveau *commit* (C6). On peut réaliser cette fusion avec l'option `--no-ff`.
- **Squash** : on obtient un nouveau *commit* qui regroupe tous les *commits* de la branche. Pour réaliser cette fusion, il faut ajouter l'option `--squash`.



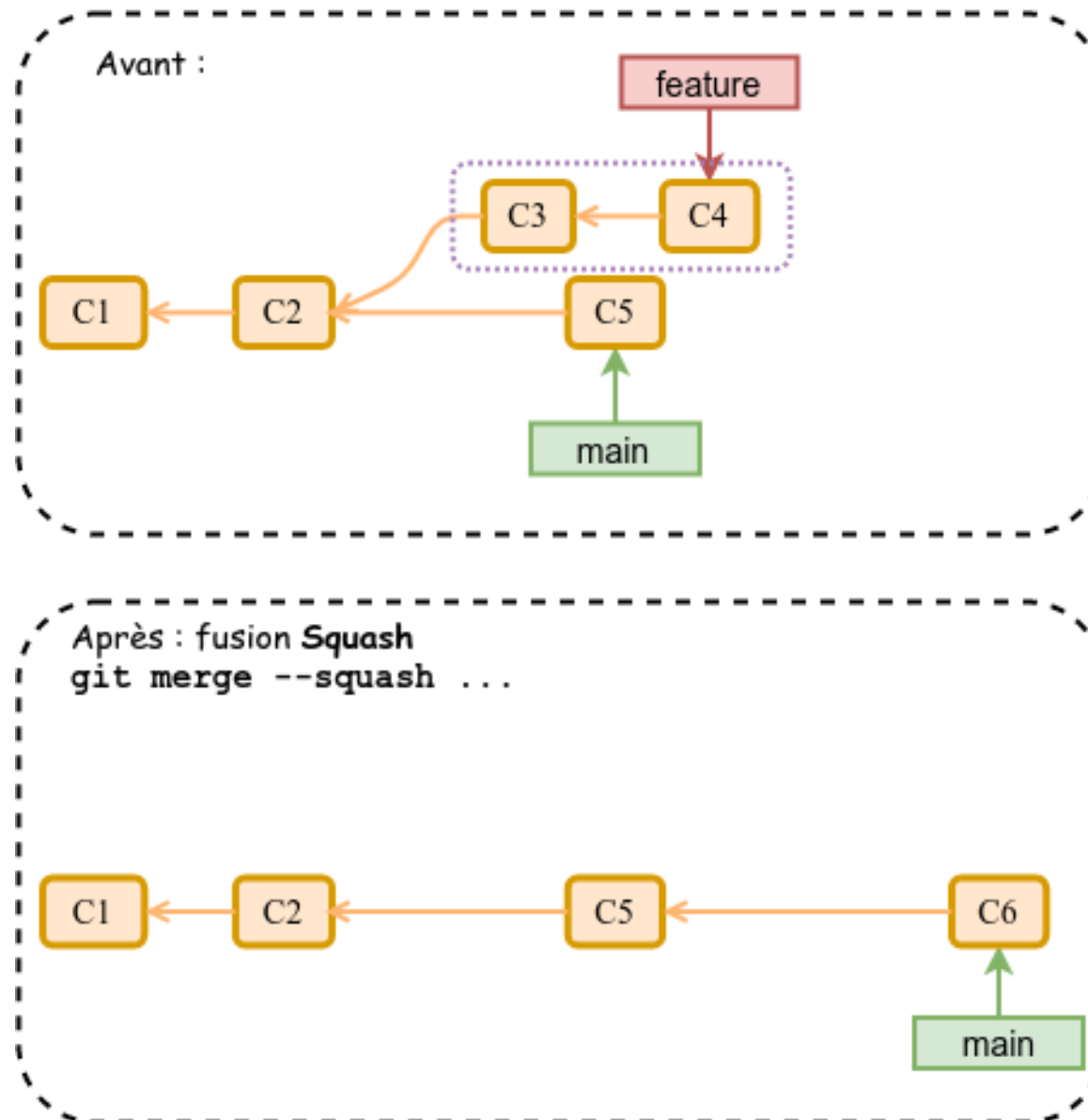
# Avance rapide (*Fast Forward*)



# Commit de fusion



# Squash



# Conflit de fusion

- Il est possible qu'une fusion (*merge*) ne puisse pas être réalisée automatiquement par Git.
- Cela arrive lorsqu'une même partie d'un fichier a été modifiée dans deux branches distinctes.
- Lorsque Git rencontre un conflit au cours d'une fusion, il l'indique dans les fichiers concernés avec des **délimiteurs** (<<<<<<<, ===== et >>>>>>>) qui marquent les deux côtés du conflit.
- Pour résoudre le conflit, il faut choisir une partie ou l'autre ou bien fusionner les deux contenus "à la main".
- On peut ensuite terminer la fusion (suivre les indications de `git status`) en faisant un `git commit`.



# workflow git

- Un **workflow Git** est une méthode, un processus de travail, une recette ou une recommandation sur la façon d'utiliser Git pour accomplir un travail de manière cohérente et productive.
- Il n'existe pas de processus standardisé sur la façon d'interagir avec Git.
- Il est important de s'assurer que l'équipe de projet est d'accord sur la façon dont le flux de modifications sera appliqué. Un *workflow* Git doit donc être défini.
- Il existe plusieurs *workflow* Git connus qui peuvent être utilisés : *workflow* centralisé, *workflow* de branche de fonctionnalité, *workflow Gitflow*, etc.

[www.atlassian.com/fr/git/tutorials/comparing-workflows](http://www.atlassian.com/fr/git/tutorials/comparing-workflows)



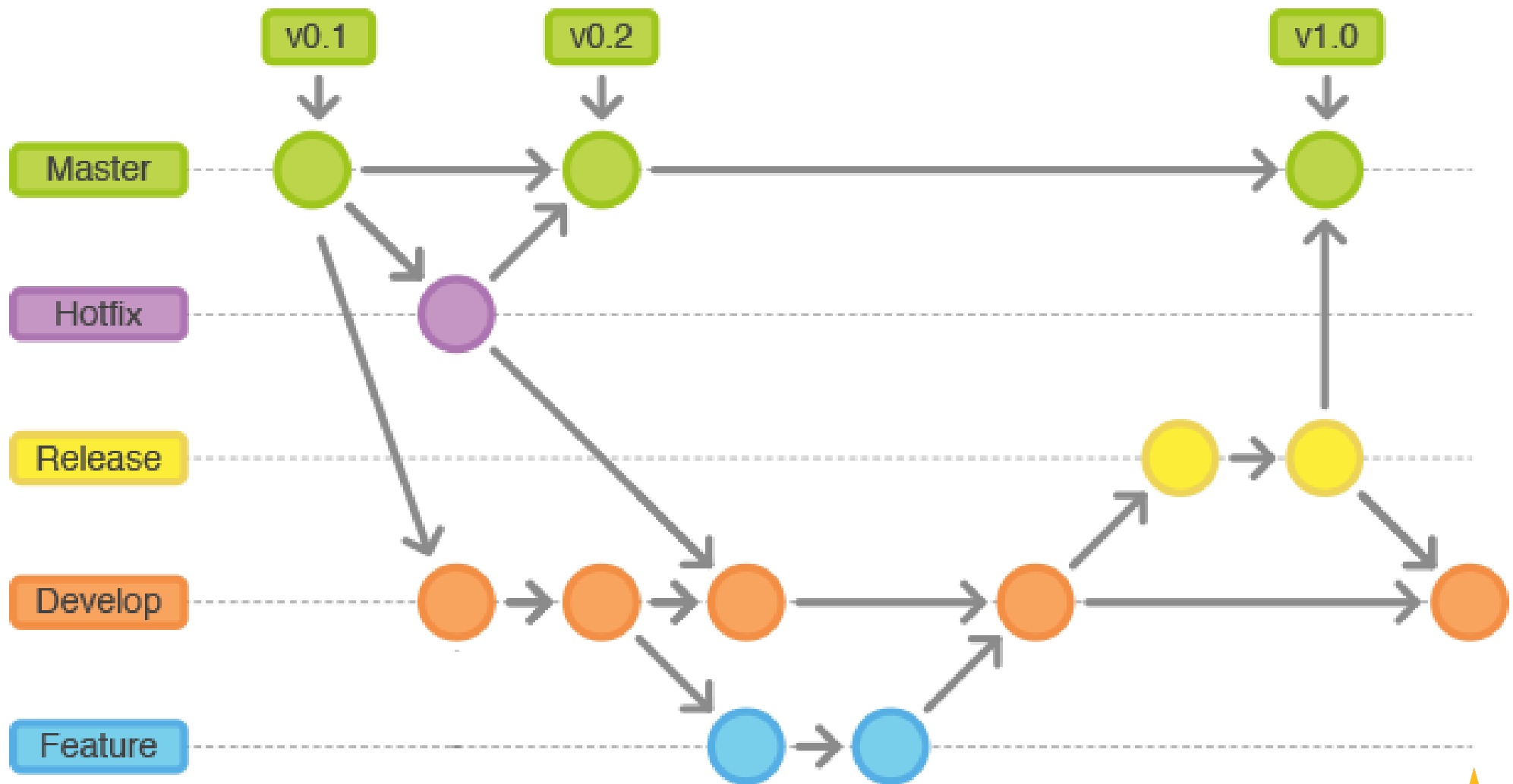
# Gitflow

- Le *workflow* **Gitflow** définit un modèle de branchement strict conçu autour de la version du projet.
- Ce *workflow* n'ajoute pas de nouveaux concepts ou commandes.
- **Gitflow** permet de gérer les bugs (*issues*), les nouvelles fonctionnalités (*features*) et les versions (*releases*) en attribuant des rôles très spécifiques à différentes branches et définit comment et quand elles doivent interagir.

Il existe des extensions `git-flow` à Git pour intégrer le *workflow* Gitflow.



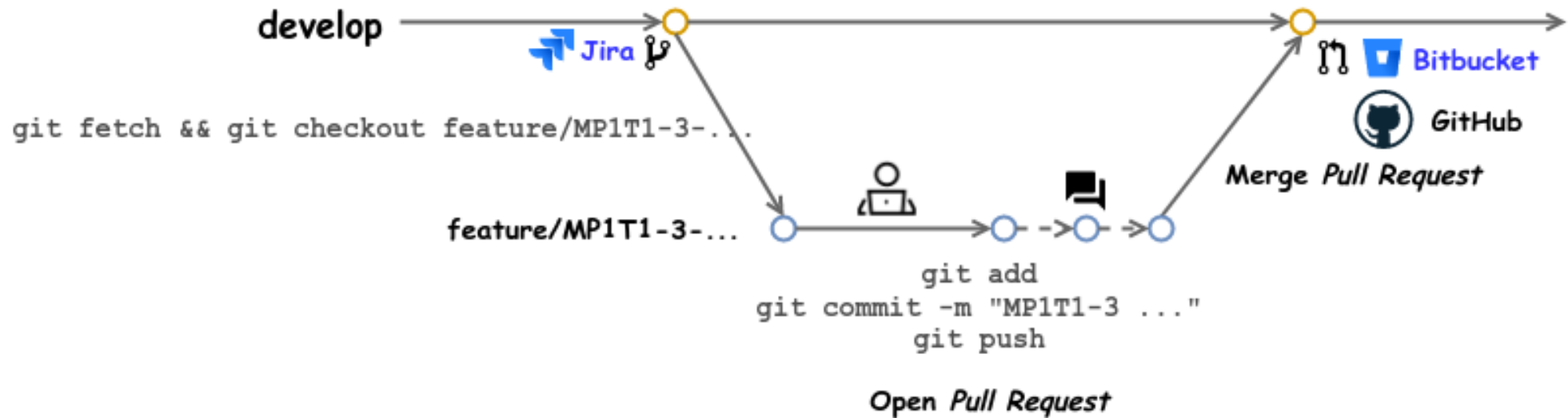
# Modèle de branches



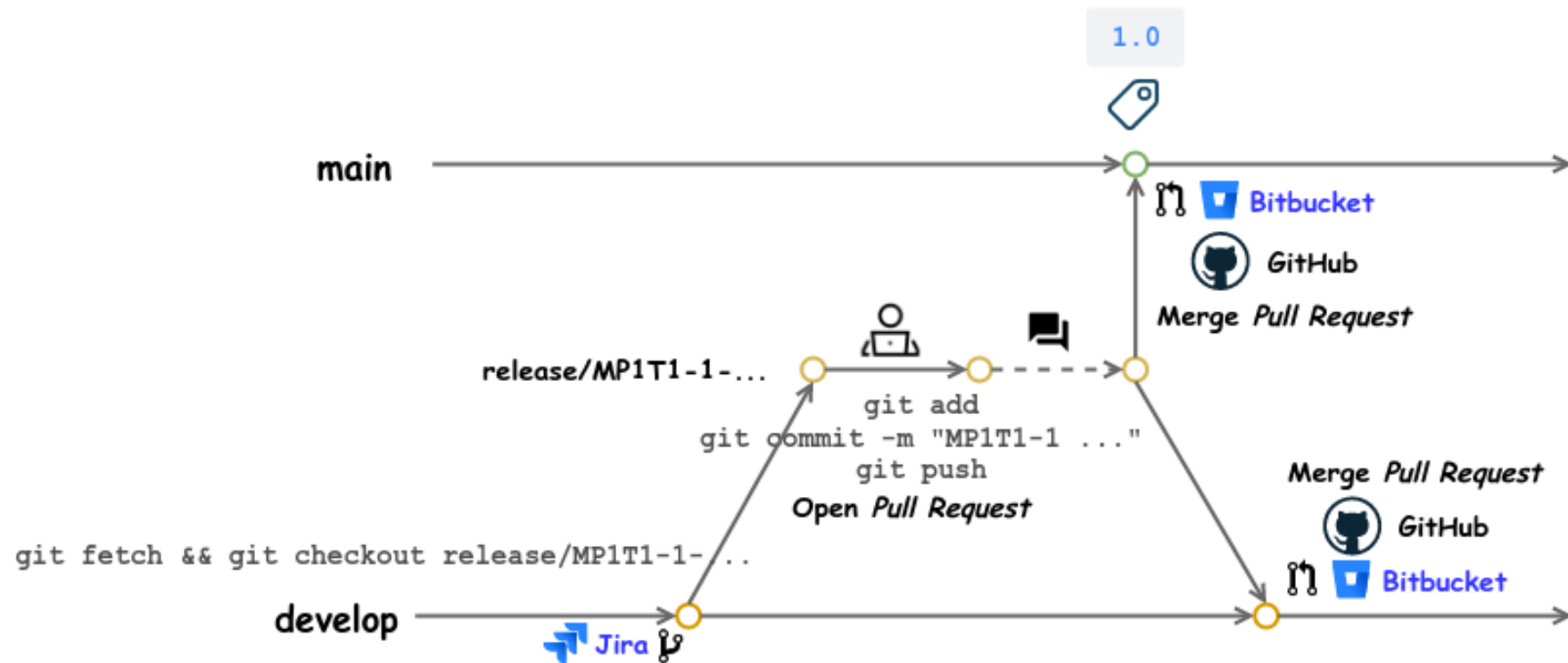
# Rôles des branches

- pour les **branches permanentes** :
  - La **branche *master*** stocke l'historique des versions officielles. Tous les *commits* de cette branche sont étiquetés avec un numéro de version (*tags*).
  - La **branche *develop*** est créée à partir de la branche *master*. Elle sert de branche d'intégration pour les fonctionnalités. Cette branche contiendra l'historique complet du projet.
- pour les **branches temporaires** :
  - Les **branches *features-xxxx*** permettent de travailler sur des nouvelles fonctionnalités. Elles sont créées directement à partir de la branche *develop* et une fois le travail fini, fusionnées vers la branche *develop*.
  - Les **branches *release-xxxx*** permettent de travailler sur une livraison (généralement des tâches dédiées à la documentation). On les crée à partir de *develop* puis on les fusionne dans *master* en leur attribuant un numéro de version (*tag*).
  - Les **branches *hotfix-xxxx*** permettent de publier rapidement (*hot*) une correction (*fix*) depuis la branche *master*. Ces branches seront ensuite fusionnées vers la branche *master* et *develop*.

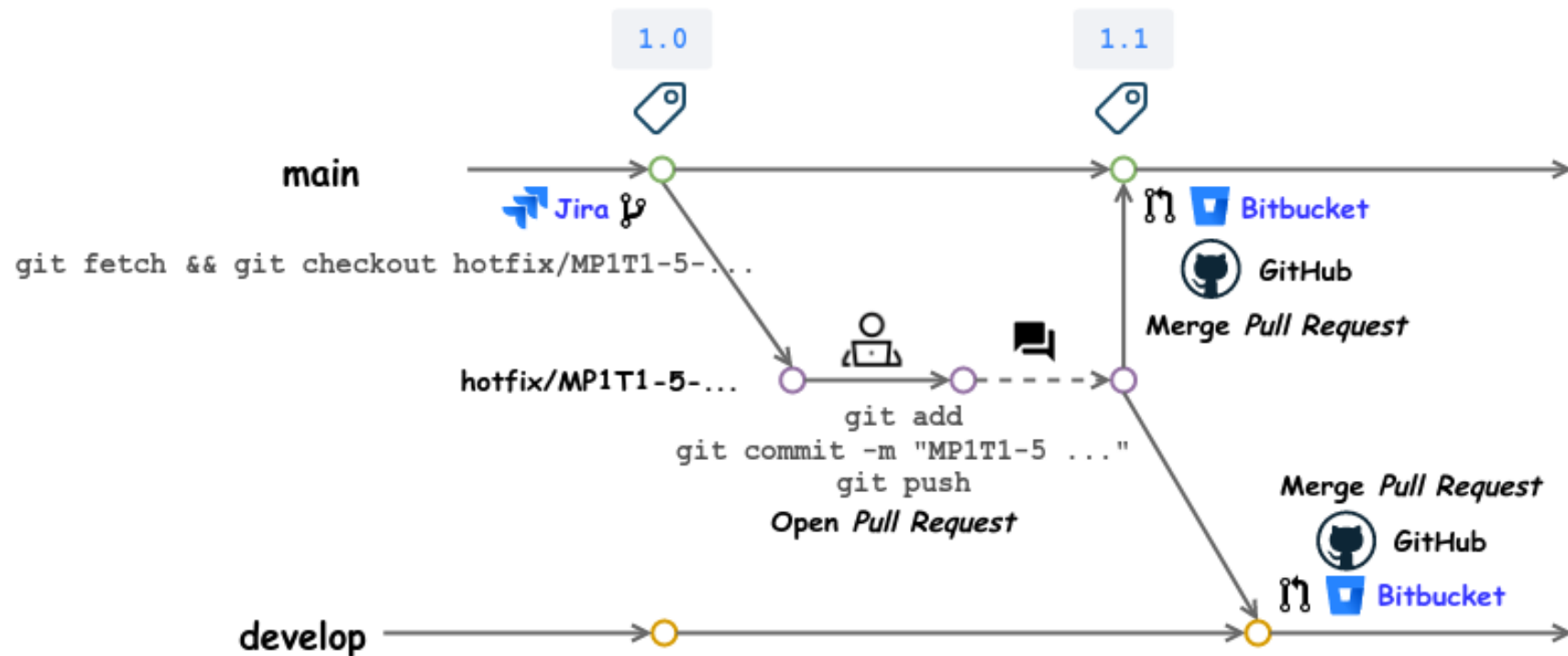
# Réalisation d'une fonctionnalité



# Réalisation d'une *release*



# Correction d'un *bug*



# Projets BTS SN

- Les branches (*feature*, *release* et *hotfix*) seront créées dans **Jira à partir d'un ticket**.
- Les fusions seront réalisées lors d'une revue de code en utilisant les ***Pull Requests* dans GitHub ou Bitbucket**.

Lorsqu'elle désigne un travail bien identifié du projet (une fonctionnalité, une *release* ou un correctif), il est préférable (obligatoire) que cela reste visible dans le graphe d'historique, même lorsque la branche est supprimée. Pour éviter que Git utilise par défaut une avance rapide (*Fast Forward*) si c'est possible, il faudra réaliser un *commit* de fusion avec l'option `--no-ff`.

# Jira / GitHub

📢 Donnez votre avis 1 👍 🔄 ... ✕

← GitHub



Create Branches PRs Tags

Repository\*

btssn-lasalle84/mp1-team0

Base branch\*

🔗 develop

Branch name\*

feature/MP1T0-2-initialisation-du-projet

⚙️ Branch Name Template

Create branch

← GitHub



Branch **feature/MP1T0-2-initialisation-du-projet** has been successfully created.

Use this command to check out your branch:

git fetch && git checkout feature/MP1T0-2-initialis



# Jira / Bitbucket

## Create branch

Repository

btssn-avignon/miniprojet1-team1

Type ⓘ

Feature

From branch

develop

Branch name

feature/ MP1T1-2-initialisation-du-projet



develop

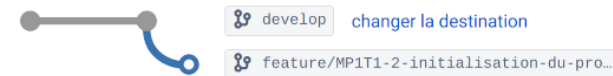
feature/MP1T1-2-initialisation-du-proj...

Create

Cancel

btssn-avignon / miniprojet1-team1 / miniprojet1-team1 / Branches

### feature/MP1T1-2-initialisation-du-projet



develop changer la destination

feature/MP1T1-2-initialisation-du-proj...

#### Consultez cette branche

Cette branche ne contient aucune modification pour l'instant. Consultez-la depuis votre ordinateur local pour la modifier.

[Consulter via Sourcetree](#)

You can also use this command to check out your branch:

```
git fetch && git checkout feature/MP1T1-2-initialisation-du-projet
```



# Choisir *commit* de fusion (*Pull Request*)

## Bitbucket

Merge strategy

Merge commit

Merge commit  
git merge --no-ff

Squash  
git merge --squash

Fast forward  
git merge --ff-only

## GitHub

Merge pull request

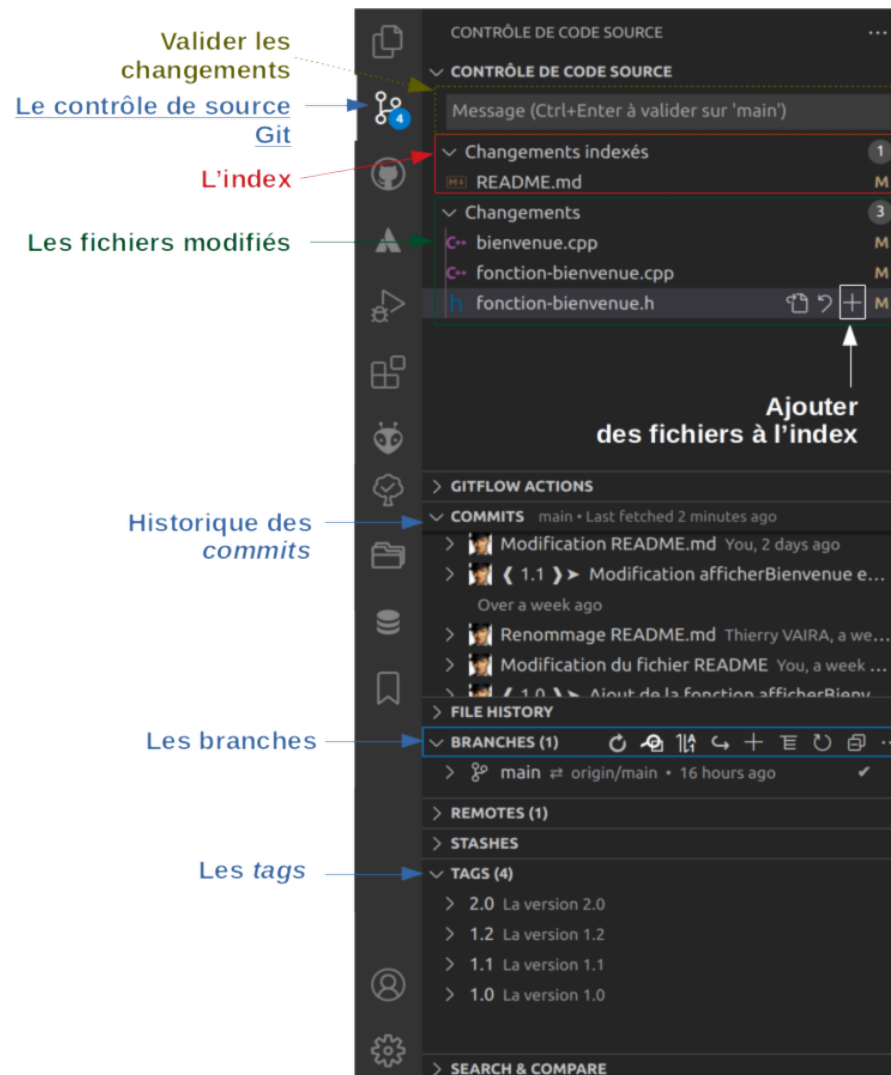
✓ **Create a merge commit**  
All commits from this branch will be added to the base branch via a merge commit.

**Squash and merge**  
The 1 commit from this branch will be added to the base branch.

**Rebase and merge**  
The 1 commit from this branch will be rebased and added to the base branch.

# Environnement de développement intégré (EDI ou IDE)

La plupart des environnements de développement intègre Git ou propose des extensions pour le faire. Le contrôle de source (SCM) dans VS Code :



# Les outils graphiques

Il existe de nombreuses interfaces graphiques permettant de gérer des projets Git :

- une interface web avec GitWeb
- une interface de visualisation détaillée et graphique avec gitk

Il existe également de nombreuses autres applications :

- qgit, Gigggle, GitExtensions, TortoiseGit, SourceTree, GitEye, ...

# GitEye

The screenshot displays the CollabNet GitEye application window. The interface is divided into several panes:

- Left Pane (Git Repositories):** Shows the repository structure for 'team-0'. It includes a 'Local' section with 'origin/develop' and 'origin/master', and a 'Remote Tracking' section with 'origin/feature/T0-1-coder-fonc-1' and 'origin/feature/T0-2-coder-fonc-2'. The 'Working Tree' section shows files like '.gitignore', '.project', and 'projet.txt'.
- Top Pane (Dashboard):** Contains tabs for 'Dashboard', 'Git Files', 'History', 'Task List', and 'Builds'. The 'History' tab is currently selected.
- Right Pane (Commit History):** Displays a table of commits with columns for 'Id', 'Message', 'Author', 'Authored Date', 'Committer', and 'Committed Date'. The table shows a series of commits, with the most recent one (90cd7a7) highlighted in orange.
- Bottom Pane:** Shows the commit details for the selected commit (90cd7a7), including the commit message, author, committer, and a list of files changed.

The commit history table is as follows:

Id	Message	Author	Authored Date	Committer	Committed Date
90cd7a7	1.0 develop master release/1.0 origin/HEAD origin/develop origin/master origin/release/1.0 HEAD	Vaira Thierry	12 hours ago	Thierry Vaira	12 hours ago
f2ea333	réalisation de la fonc 2	Vaira Thierry	12 hours ago	Vaira Thierry	12 hours ago
4f8e9e8	Merged develop into feature/T0-2-coder-fonc-2	Thierry Vaira	13 hours ago	Thierry Vaira	13 hours ago
9f5d94c	Revert "réalisation de la fonc 2"	tvaire	13 hours ago	tvaire	13 hours ago
9e5ff5e	réalisation de la fonc 2	tvaire	2 days ago	tvaire	2 days ago
0dd01b3	origin/feature/... réalisation de la fonc 1	tvaire	2 days ago	tvaire	2 days ago
e504848	origin/feature/... Ajout du fichier projet.txt	vaira	3 days ago	vaira	3 days ago
ff88740	Initial commit	Thierry Vaira	3 days ago	Thierry Vaira	3 days ago

The commit details for the selected commit (90cd7a7) are as follows:

```

commit 90cd7a745004cf06db097aa7e971d96306063740
Author: Vaira Thierry <rk42nftjrm@privaterelay.appleid.com> 2021-07-09 21:03:52
Committer: Thierry Vaira <vaira@lasalle84.org> 2021-07-09 21:03:52
Parent: 0dd01b30c83b0e819150193cc49e6cdd79fba232 (réalisation de la fonc 1)
Parent: f2ea33352ac5786cbb5f9823116a010dd432ef09 (réalisation de la fonc 2)
Branches: develop, master, release/1.0, origin/HEAD, origin/develop, origin/master, origin/release/1.0
Tags: 1.0

Fusion effectuée feature/T0-2-coder-fonc-2 (pull request #1)

réalisation de la fonc 2

Approuvé par : Thierry Vaira
  
```

# Les principales commandes I

Git est un ensemble de commandes indépendantes dont les principales sont :

- `git init` crée un nouveau dépôt ;
- `git clone` clone un dépôt distant ;
- `git add` ajoute le contenu du répertoire de travail dans la zone d'index pour le prochain *commit* ;
- `git status` montre les différents états des fichiers du répertoire de travail et de l'index ;
- `git diff` montre les différences ;
- `git commit` enregistre dans la base de données (le dépôt) un nouvel instantané avec le contenu des fichiers qui ont été indexés puis fait pointer la branche courante dessus ;
- `git branch` liste les branches ou crée une nouvelle branche ;



# Les principales commandes II

- `git checkout` permet de basculer de branche et d'en extraire le contenu dans le répertoire de travail ;
- `git merge` fusionne une branche dans une autre ;
- `git log` affiche la liste des *commits* effectués sur une branche ;
- `git fetch` récupère toutes les informations du dépôt distant et les stocke dans le dépôt local ;
- `git push` publie les nouvelles révisions sur le dépôt distant ;
- `git pull` récupère les dernières modifications distantes du projet et les fusionne dans la branche courante ;
- `git tag` liste ou crée des *tags* ;
- `git stash` stocke de côté un état non commité afin d'effectuer d'autres tâches.

