Workflows de collaboration Git : modèles de branches, revue de code et intégration continue

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Objectifs et vocabulaire des workflows de collaboration 1
1.1 Vocabulaire 2
2 Six modèles de workflow 3
2.1 Workflow centralisé 4
2.2 Feature branch workflow 4
2.3 Gitflow 5
2.4 GitHub Flow 6
2.5 Forking workflow 6
2.6 Trunk-based development 6
3 Comparaison des six modèles 7
4 Intégration continue et revue de code 8
4.1 Gabarit de pull request et checklist de revue 8
5 Rebase et fusion : ce que fait chaque commande 9
6 Automatiser l'interface distante avec GitHub CLI 10
7 Exercice intermédiaire : forker et synchroniser 10
8 Exercice final : fusionner à deux et résoudre un conflit de fusion 17
9 Section bonus : simuler un flux Gitflow 22
10 Conclusion et ressources 24

1 Objectifs et vocabulaire des workflows de collaboration

Un workflow est un jeu de règles qui décrit comment les modifications circulent dans une équipe : quelles branches existent, combien de temps, qui fusionne quoi, sous quel contrôle. Git n'impose aucun de ces règlements — il les permet tous. Le choix d'un workflow est donc une décision de coordination, pas une contrainte technique : un modèle se juge au problème d'échelle qu'il résout et au coût qu'il impose en échange.

Quatre objectifs structurent la leçon. Ils sont formulés comme des critères vérifiables, en miroir du barème de l'exercice final : ce qui est exigé ici prépare ce qui y sera évalué.

- **Clarté** — chaque participant sait sur quelle branche il travaille et où va sa modification. Critère : le nom de la branche et sa cible se déduisent du seul énoncé du modèle, sans question à poser.
- **Collaboration** — aucune modification n'atteint la branche partagée sans avoir été lue. Critère : chaque fusion est précédée d'une relecture consignée dans la pull request.
- **Qualité** — l'état reçu est validé par une exécution automatique, non par une déclaration. Critère : la pipeline s'exécute sur chaque pull request et bloque la fusion en cas d'échec.
- **Livraison** — le rythme de publication est choisi, pas subi. Critère : la date de mise en production se déduit du modèle (à chaque fusion, à chaque branche de release, à chaque tag).

> ! Important
>
> Un workflow ne produit ses effets qu'à trois conditions : formalisé (dans le README du projet), automatisé (par la CI et la protection des branches), appliqué sans exception — y compris par ses auteurs.

1.1 Vocabulaire

Chaque terme reçoit une définition bornée à l'usage de la leçon ; quand Les bases de Git a déjà posé le concept, la définition est renvoyée, pas réenseignée.

> i Définition — workflow (de collaboration)
>
> Ensemble de conventions régissant la création, la fusion, la relecture et la publication des branches dans un dépôt partagé. Le modèle est indépendant de l'outil de gestion de versions ; Git se contente de le rendre exécutable.

> i Définition — pull request (PR) et merge request (MR)
>
> Demande de fusion d'une branche dans une autre, déposée sur une forge et assortie d'une relecture, d'une discussion et d'une pipeline. Le mécanisme est unique, le nom dépend de l'écosystème : pull request sur GitHub, merge request sur GitLab. Les deux désignations s'abrègent PR et MR dans la suite.

ℹ Définition — intégration continue (CI)

Exécution automatique des vérifications de qualité (compilation, tests, lint) déclenchée par un événement de la forge — poussée ou pull request. Son résultat conditionne l'autorisation de fusionner.

ℹ Définition — livraison continue (CD)

Déploiement automatisé d'un état validé par la CI, selon une règle fixée à l'avance (chaque fusion sur la branche principale, chaque tag). Elle se distingue du déploiement continu en ce qu'une décision humaine peut subsister sur le chemin de la production.

ℹ Définition — protection de branche

Ensemble de règles côté serveur appliquées à une branche : push direct interdit, relecture et pipeline exigées avant fusion. Configurée sur la forge, pas dans Git — d'où son efficacité, qui vaut pour tous les contributeurs sans exception.

ℹ Définition — Conventional Commits

Convention de format des messages de validation (`feat:`, `fix:`, `docs:`…). Posée dans Les bases de Git (section « Convention — Conventional Commits ») ; elle sert ici de critère de la checklist de revue.

ℹ Définition — rebase et merge

Deux manières d'intégrer une branche dans une autre, analysées en section « Rebase et fusion : ce que fait chaque commande ». À retenir d'emblée : la fusion ajoute un commit à l'histoire, le rebassage rejoue les commits ailleurs sous des empreintes neuves.

2 Six modèles de workflow

Les six modèles suivent une grille commune : le problème d'échelle traité, le mouvement des réfs induit (modèle posé dans Les bases de Git, section « Branches et fusions »), le patron de commandes, les conditions d'adéquation énoncées en critères. Les blocs de commandes décrivent des patrons — ils citent des dépôts

distants propres à chaque projet et ne s'exécutent pas ici ; les deux simulations de la leçon, seules exécutées, travaillent dans des dépôts locaux.

2.1 Workflow centralisé

Problème traité. Une équipe très petite veut un point de convergence unique : tout le monde travaille sur la même branche, à la queue leu leu.

Mouvement des réfs. Une seule réf locale, `main`, qui glisse sur la réf distante du même nom ; aucune seconde branche à fusionner.

```
Developers → commit → push → origin/main
```

```
git switch main
git pull --rebase
# travail : git add, git commit
git push origin main
```

Adéquation. Le modèle fonctionne à condition que la confiance soit totale — disons deux personnes au plus — car rien n'impose de relecture : la CI, si elle existe, s'exécute après la poussée, plus avant. Coût : les conflits se règlent au moment du pull, donc après le travail et dans la précipitation.

2.2 Feature branch workflow

Problème traité. Plusieurs sujets avancent en parallèle sans s'attendre mutuellement ; chacun doit pouvoir être intégré indépendamment des autres.

Mouvement des réfs. Une réf `feature/<sujet>` bifurque depuis `main` ; la ramène soit un commit de fusion à deux parents, soit, en cas de squash (réduction de toute la branche à un commit unique à l'intégration — analysé §5), un commit neuf posé sur `main`.

```
main ---o-----o---------o
        \         /
 feature1 o-----o
```

```
git switch -c feature/connexion
git push --set-upstream origin feature/connexion
# pendant le travail, pour rester proche de main :
git fetch origin
git rebase origin/main
# intégration après relecture (typiquement via PR) :
git switch main
```

```
git merge --no-ff feature/connexion
git push origin main
```

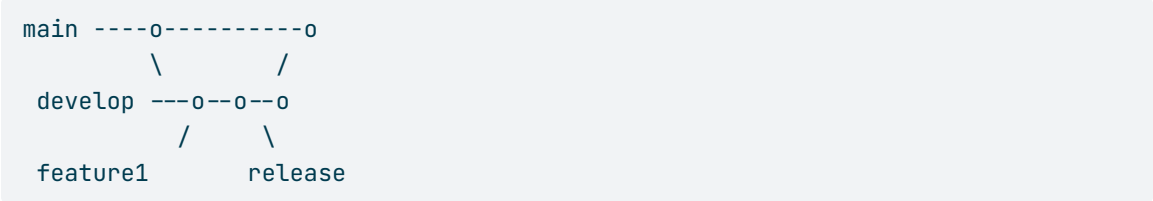
Adéquation. Compromis généraliste dès qu'une relecture est exigée : chaque sujet est isolé, la CI s'exécute sur la pull request. Coût : une branche qui traîne diverge et doit être resynchronisée — c'est le rôle du rebassage analysé plus bas.

2.3 Gitflow

Problème traité. Un produit livré par versions numérotées doit préparer une release sans interrompre le développement de la suivante, et pouvoir corriger une version ancienne déjà en production.

Mouvement des réfs. Deux réfs permanentes : `main` (l'état publié) et `develop` (l'intégration). Les `feature/*` naissent de `develop` et y reviennent ; les `release/*` naissent de `develop` et retournent aux deux ; les `hotfix/*` naissent de `main` et retournent aux deux. Tous ces mouvements sont des fusions ordinaires : le modèle n'ajoute aucun mécanisme, seulement une discipline de nommage et de chronologie.

Vincent Driessen a formalisé cette organisation en 2010 sous le nom de Gitflow ; le modèle précède l'ère du déploiement continu, ce qui explique sa lourdeur dans les projets web.




```
git switch develop
git switch -c feature/export-csv
git switch develop
git merge --no-ff feature/export-csv
git switch -c release/1.2 develop
git switch main
git merge --no-ff release/1.2
git tag -a v1.2 -m "Version 1.2"
git switch develop
git merge --no-ff release/1.2
```

Adéquation. Pertinent pour les logiciels livrés en paquets (application mobile, installation sur site), où une version a un début et une fin. Coût : chaque intégration tient une double comptabilité (`develop`, puis `main`) ; le modèle devient une entrave pour un produit déployé quotidiennement.

2.4 GitHub Flow

Problème traité. Publier souvent sans perdre la relecture : la chorégraphie de releases de Gitflow devient une gêne dès que la production est à un clic.

Mouvement des réfs. Une seule réf permanente, `main`, continuellement déployable ; les branches sont courtes, naissent de `main` et y meurent par fusion.

```
main ---o---o----o---o
         \ /      \ /
         PR       PR
```

Adéquation. Équipes dotées d'une CI rapide et fiable, pull requests de taille réduite. C'est le modèle que met en œuvre l'exercice final de cette leçon : branche, pull request, relecture, fusion.

2.5 Forking workflow

Problème traité. Accepter les contributions d'auteurs sans droit d'écriture sur le dépôt central : logiciels libres, projets de cours, échanges internes entre services.

Mouvement des réfs. Deux dépôts distants : l'amont (`upstream`) et le fork du contributeur. La branche locale glisse sur `upstream/main` par fetch puis rebase ; la branche de contribution est poussée vers le fork, d'où part la pull request vers l'amont.

```
upstream (org/repo)
   ^
   |  pull request
fork (contributeur/repo) ⟵ clone de travail
```

```
git remote add upstream git@github.com:org/repo.git
git fetch upstream
git switch main
git rebase upstream/main
git push origin main
```

Adéquation. Modèle de référence dès que les contributeurs n'ont aucun droit : le contrôle d'accès se joue à la pull request, que le mainteneur relit chez lui. Coût : la double synchronisation (`origin` et `upstream`), exécutée en simulation dans la section qui suit.

2.6 Trunk-based development

Problème traité. Supprimer la divergence à la source : moins il y a de branches, moins il y a de fusions, et plus tôt les erreurs d'intégration apparaissent.

Mouvement des réfs. Les commits aboutissent sur `main` (le tronc) plusieurs fois par jour, directement ou par branches d'un à deux jours de vie ; les fonctions inachevées sont masquées par des interrupteurs de fonctionnalité, non par des branches.

Adéquation. Exige trois critères réunis : couverture de tests élevée, CI rapide, discipline d'équipe. Sans eux, le modèle dégénère en workflow centralisé sans relecture — le pire des deux.

3 Comparaison des six modèles

Les caractérisations qui précèdent tiennent dans un tableau à critères fermés : chaque case se vérifie comme vrai ou faux, ce qui rend le choix argumentable plutôt qu'opinionnel.

Modèle	Branches permanentes	Relecture avant fusion	Déclencheur de déploiement	Coût de coordination
Centralisé	`main`	Non	Poussée sur `main`	Très faible
Feature branch	`main`	Oui (PR)	Choix explicite	Faible
Gitflow	`main`, `develop`	Oui	Fusion dans `main` via `release/*`	Élevé
GitHub Flow	`main`	Oui (PR)	Fusion dans `main`	Faible
Forking	`main` (amont)	Oui (PR depuis le fork)	Au bon vouloir du mainteneur	Moyen (double synchronisation)
Trunk-based	`main`	Oui, continue	Chaque fusion	Élevé (tests, interrupteurs)

Deux correspondances concluent la comparaison : un projet open source accueillant des contributeurs extérieurs relève du forking, sans discussion ; une organisation qui livre plusieurs fois par jour et dispose des tests correspondants relève du trunk-based. Entre ces deux pôles, feature branch et GitHub Flow couvrent l'essentiel des situations de cours et de produit ; Gitflow répond à un rythme de versions numérotées ; le centralisé reste cantonné à l'essai personnel ou au binôme de dépannage.

4 Intégration continue et revue de code

Intégration continue. Chaque événement de la forge (poussée, pull request) déclenche une pipeline : compilation, tests, linteurs. Deux mécanismes méritent d'être nommés, car la CI est souvent présentée comme une magie : un déclencheur — la forge appelle l'exécuteur de la pipeline sur l'événement — et une barrière — le résultat alimente la protection de branche côté serveur, qui refuse la fusion tant que la pipeline est rouge.

Le cours lui-même fonctionne sur ce modèle : le document que vous lisez est produit par une pipeline qui le rend à chaque poussée ; une leçon qui casse le rendu n'atteint pas le site. La vérification y est automatisée et bloquante — exactement ce que la CI signifie pour du code.

Livraison continue. Une fois la CI verte, la CD déploie selon la règle choisie dans le modèle (chaque fusion, chaque tag). La CD présuppose la CI : déployer sans vérification exécutée, c'est déployer plus vite ses propres erreurs.

Le slogan fondateur de la CI — test early, fail fast, fix quickly — n'est pas une incitation mais une définition d'un rythme : tester tôt, c'est exécuter les vérifications à chaque intégration et non en fin de cycle ; échouer vite, c'est garder une pipeline assez courte pour que la faute reste imputable à la dernière contribution ; corriger aussitôt, c'est considérer un état rouge comme bloquant, car une intégration cassée masque la suivante.

4.1 Gabarit de pull request et checklist de revue

Une pull request n'est relisible que si elle dit ce qu'elle fait. Le gabarit — le template de la forge, un fichier `PULL_REQUEST_TEMPLATE.md` prérempli dans chaque nouvelle demande — borne la description à quatre rubriques :

```
# Objet
Une phrase : ce que la modification change et pourquoi.

# Modifications
Liste des fichiers touchés ; une ligne de justification par fichier.

# Vérification
Commande exécutée localement et résultat observé (ex. `make test`).

# Références
Issue liée, le cas échéant (`Fixes #42`).
```

La relecture s'appuie sur une checklist fermée, chaque item se décidant à la seule lecture de la demande :

1. Le titre suit la convention Conventional Commits (Les bases de Git, section homonyme).
2. Le périmètre est restreint : le diff tient dans environ 40 lignes ; au-delà, la demande est scindée.
3. Chaque fichier modifié apparaît dans la rubrique Modifications, avec sa justification.
4. La rubrique Vérification énonce une commande que le relecteur peut réexécuter.
5. La pipeline de CI est verte.
6. La contribution ne contient aucun secret ; aucune ligne commentée pour faire passer un build ; aucune commande destructrice (`reset --hard`, `clean -f`, push forcé) sans justification écrite dans la description (Les bases de Git, section « Commandes destructrices »).

Cette checklist constitue la partie revue de la grille de l'exercice final ; les étapes d'exécution de la consigne en constituent l'autre partie : relire son binôme avec ces critères, c'est déjà appliquer une moitié du barème.

5 Rebase et fusion : ce que fait chaque commande

La fusion crée un commit à deux parents qui devient la nouvelle position de la réf cible ; les commits fusionnés conservent leurs empreintes. C'est l'opération vérifiée dans Les bases de Git (section « Branches et fusions ») par la lecture de l'objet : la ligne `parent` y apparaît deux fois. L'option `--no-ff` impose cette trace de groupe même lorsque Git pourrait s'en passer — elle répond à la question « cette branche a-t-elle existé ? » longtemps après sa suppression.

Le rebassage ne crée aucun commit de fusion. Git détermine le dernier état commun aux deux branches, extrait les commits propres à la branche rebasée sous forme de correctifs, déplace la réf sur la nouvelle base, puis rejoue les correctifs un par un. Chaque commit est recréé : même message, mêmes modifications, mais parent différent — donc empreinte différente. La conséquence se lit directement dans la section « Travailler à plusieurs » de Les bases de Git : une branche rebasée ne contient plus l'histoire que le distant connaît déjà, et le refus `non-fast-forward` protège précisément celle-ci. Le squash proposé par les forges généralise ce mouvement : toute la contribution devient un correctif unique rejoué sur `main`, qui garde un historique linéaire.

Le choix employé dans ce cours se borne à trois règles :

- sur une branche personnelle dont les commits n'ont pas encore été poussés : rebase, pour rester proche de `main` ;
- pour l'intégration d'une pull request relue : merge (`--no-ff` ou squash), décidé sur la forge ;

- `git rebase -i` (nettoyage interactif de l'historique local) est nommé ici, non enseigné : il appartient à un autre moment du cursus.

6 Automatiser l'interface distante avec GitHub CLI

GitHub CLI (`gh`) n'est ni un client Git ni une couche magique : c'est un **client de l'API de la forge**. Les opérations qu'il accomplit sont celles de l'interface web — ouvrir une pull request, consulter une pipeline, créer un dépôt — exécutées par appels HTTP, donc scriptables. Le dépôt lui-même reste régi par Git seul : `gh` ne touche l'arbre de travail que par délégation, en invoquant les commandes vues dans la leçon précédente.

```
# Bloc non exécuté par le rendu — patron de commandes.
gh repo clone utilisateur/repo
gh repo create mon-repo --private
gh pr create --fill --base main --head feature/x
gh pr checkout 123
```

Deux précisions. `gh pr checkout` est un verbe de la forge qui récupère et affiche la branche d'une pull request par l'intermédiaire de Git ; il n'a rien à voir avec l'ancienne commande `git checkout` que ce cours a retirée de son vocabulaire. Et `gh` est un outil distinct de Git : sa disponibilité dans un environnement se vérifie par `gh --version`, sans présumer de sa présence.

7 Exercice intermédiaire : forker et synchroniser

> ⚠️ Exercice : circuit complet d'une contribution
>
> Sur une forge, forkez un petit dépôt, clonez votre fork, déclarez le dépôt d'origine sous le nom `upstream`, synchronisez votre branche principale, puis proposez une contribution par pull request. Si la forge n'est pas disponible au moment du rendu, la simulation ci-dessous produit exactement les mêmes objets et les mêmes échanges entre distants.

Simulation. Quatre dépôts dans `/tmp/git-wf` : un amont nu, son fork nu, le clone du mainteneur, le clone du contributeur. La première cellule définit les variables `WF`, `UPSTREAM`, `FORK`, `SEED`, `CONTRIB` : en cas de réexécution isolée, réexécuter-la d'abord.

```
WF=/tmp/git-wf
UPSTREAM=$WF/upstream.git
FORK=$WF/fork.git
SEED=$WF/seed
```

```
CONTRIB=$WF/contrib
rm -rf $WF && mkdir -p $WF
git config --global init.defaultBranch main
git config --global user.name "Atelier Git"
git config --global user.email "atelier.git@example.invalid"
```

L'amont est créé et pourvu d'un contenu initial par le mainteneur :

```
git init --bare $UPSTREAM
git clone $UPSTREAM $SEED
cd $SEED
printf '# Projet atelier\n' > README.md
git add README.md
git commit -m "docs: initialiser le dépôt amont"
git push -u origin main
```

```
Initialized empty Git repository in /tmp/git-wf/upstream.git/
Cloning into '/tmp/git-wf/seed'...
warning: You appear to have cloned an empty repository.
done.
[main (root-commit) 178dcb3] docs: initialiser le dépôt amont
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
Enumerating objects: 3, done.
Counting objects:  33% (1/3)
Counting objects:  66% (2/3)
Counting objects: 100% (3/3)
Counting objects: 100% (3/3), done.
Writing objects:  33% (1/3)
Writing objects:  66% (2/3)
Writing objects: 100% (3/3)
Writing objects: 100% (3/3), 256 bytes | 256.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To /tmp/git-wf/upstream.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

Le fork est un clone nu de l'amont ; le contributeur clone son fork et déclare l'amont en second distant :

```
git clone --bare $UPSTREAM $FORK
git clone $FORK $CONTRIB
```

```
git -C $CONTRIB remote add upstream $UPSTREAM
git -C $CONTRIB remote -v
```

```
Cloning into bare repository '/tmp/git-wf/fork.git'...
done.
Cloning into '/tmp/git-wf/contrib'...
done.
origin  /tmp/git-wf/fork.git (fetch)
origin  /tmp/git-wf/fork.git (push)
upstream    /tmp/git-wf/upstream.git (fetch)
upstream    /tmp/git-wf/upstream.git (push)
```

L'amont pendant que le contributeur a le dos tourné :

```
cd $SEED
echo "Les contributions passent par le fork de chacun." >> README.md
git commit -am "docs: préciser le circuit des contributions"
git push origin main
```

```
[main 821c09b] docs: préciser le circuit des contributions
 1 file changed, 1 insertion(+)
Enumerating objects: 5, done.
Counting objects:  20% (1/5)
Counting objects:  40% (2/5)
Counting objects:  60% (3/5)
Counting objects:  80% (4/5)
Counting objects: 100% (5/5)
Counting objects: 100% (5/5), done.
Delta compression using up to 32 threads
Compressing objects:  50% (1/2)
Compressing objects: 100% (2/2)
Compressing objects: 100% (2/2), done.
Writing objects:  33% (1/3)
Writing objects:  66% (2/3)
Writing objects: 100% (3/3)
Writing objects: 100% (3/3), 338 bytes | 338.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To /tmp/git-wf/upstream.git
   178dcb3..821c09b  main → main
```

Le contributeur synchronise son `main` avec `fetch` puis `rebase` — les deux gestes érigés en norme dans Les bases de Git :

```
cd $CONTRIB
git fetch upstream
git rebase upstream/main
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects:  20% (1/5)
remote: Counting objects:  40% (2/5)
remote: Counting objects:  60% (3/5)
remote: Counting objects:  80% (4/5)
remote: Counting objects: 100% (5/5)
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects:  50% (1/2)
remote: Compressing objects: 100% (2/2)
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects:  33% (1/3)
Unpacking objects:  66% (2/3)
Unpacking objects: 100% (3/3)
Unpacking objects: 100% (3/3), 318 bytes | 318.00 KiB/s, done.
From /tmp/git-wf/upstream
 * [new branch]      main       -> upstream/main


Successfully rebased and updated refs/heads/main.
```

Puis il travaille sur une branche dédiée et la pousse vers **son** fork, jamais vers l'amont où il n'a pas de droits :

```
git switch -c contribution/atelier
mkdir -p contributions
echo "Ce que l'atelier m'a appris : distinguer origin et upstream." >
contributions/atelier.md
git add contributions/atelier.md
git commit -m "docs: ajouter la contribution de l'atelier"
git push -u origin contribution/atelier
```

```
Switched to a new branch 'contribution/atelier'
[contribution/atelier 5abfeba] docs: ajouter la contribution de l'atelier
 1 file changed, 1 insertion(+)
 create mode 100644 contributions/atelier.md
Enumerating objects: 9, done.
Counting objects:  11% (1/9)
```

```
Counting objects:  22% (2/9)
Counting objects:  33% (3/9)
Counting objects:  44% (4/9)
Counting objects:  55% (5/9)
Counting objects:  66% (6/9)
Counting objects:  77% (7/9)
Counting objects:  88% (8/9)
Counting objects: 100% (9/9)
Counting objects: 100% (9/9), done.
Delta compression using up to 32 threads
Compressing objects:  20% (1/5)
Compressing objects:  40% (2/5)
Compressing objects:  60% (3/5)
Compressing objects:  80% (4/5)
Compressing objects: 100% (5/5)
Compressing objects: 100% (5/5), done.
Writing objects:  14% (1/7)
Writing objects:  28% (2/7)
Writing objects:  42% (3/7)
Writing objects:  57% (4/7)
Writing objects:  71% (5/7)
Writing objects:  85% (6/7)
Writing objects: 100% (7/7)
Writing objects: 100% (7/7), 724 bytes | 724.00 KiB/s, done.
Total 7 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To /tmp/git-wf/fork.git
 * [new branch]      contribution/atelier → contribution/atelier
branch 'contribution/atelier' set up to track 'origin/contribution/atelier'.
```

La pull request est l'interface de la forge ; son effet côté mainteneur se réduit à deux gestes connus : récupérer la branche du fork, la fusionner sous une trace explicite.

```
cd $SEED
git fetch $FORK contribution/atelier
git merge --no-ff FETCH_HEAD -m "Fusion de la contribution atelier depuis le fork"
git push origin main
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects:  20% (1/5)
remote: Counting objects:  40% (2/5)
remote: Counting objects:  60% (3/5)
remote: Counting objects:  80% (4/5)
remote: Counting objects: 100% (5/5)
```

```
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects:  33% (1/3)
remote: Compressing objects:  66% (2/3)
remote: Compressing objects: 100% (3/3)
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects:  25% (1/4)
Unpacking objects:  50% (2/4)
Unpacking objects:  75% (3/4)
Unpacking objects: 100% (4/4)
Unpacking objects: 100% (4/4), 398 bytes | 398.00 KiB/s, done.
From /tmp/git-wf/fork
 * branch            contribution/atelier → FETCH_HEAD
Merge made by the 'ort' strategy.
 contributions/atelier.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 contributions/atelier.md
Enumerating objects: 6, done.
Counting objects:  16% (1/6)
Counting objects:  33% (2/6)
Counting objects:  50% (3/6)
Counting objects:  66% (4/6)
Counting objects:  83% (5/6)
Counting objects: 100% (6/6)
Counting objects: 100% (6/6), done.
Delta compression using up to 32 threads
Compressing objects:  25% (1/4)
Compressing objects:  50% (2/4)
Compressing objects:  75% (3/4)
Compressing objects: 100% (4/4)
Compressing objects: 100% (4/4), done.
Writing objects:  20% (1/5)
Writing objects:  40% (2/5)
Writing objects:  60% (3/5)
Writing objects:  80% (4/5)
Writing objects: 100% (5/5)
Writing objects: 100% (5/5), 511 bytes | 511.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
To /tmp/git-wf/upstream.git
   821c09b..a64e981  main → main
```

Critères de validation. L’exercice est reçu si trois observations tiennent sur l’amont.

```
git -C $SEED log --oneline --graph -6
```

```
*   a64e981 (HEAD → main, origin/main) Fusion de la contribution atelier depuis
le fork
|\
| * 5abfeba docs: ajouter la contribution de l'atelier
|/
* 821c09b docs: préciser le circuit des contributions
* 178dcb3 docs: initialiser le dépôt amont
```

Le dernier commit de `main` porte deux parents — la fusion de la contribution est tracée ; le fichier `contributions/atelier.md` existe dans l'amont alors que son auteur n'y a jamais écrit ; et la synchronisation du contributeur n'a produit aucun commit de fusion, le rebase ayant suffi. Ces trois faits sont exactement ce qu'une forge montre sur la page d'une pull request acceptée.

8 Exercice final : fusionner à deux et résoudre un conflit de fusion

⚠️ Exercice : le binôme et la même ligne

Dans le dépôt que l'enseignement fournit — vous y avez un accès en écriture à deux —, indépendamment de son contenu. Ce dépôt est indiqué dans l'annonce de séance ; à défaut, prenez le dépôt que vous avez poussé à l'exercice de la leçon précédente et invitez votre binôme en écriture :

0. Le fichier partagé `contributions/README.md` doit exister sur `main` avec une ligne `Dernière mise à jour : …`. Si le dépôt fourni ne le contient pas, l'un des deux l'y ajoute et le pousse ; l'autre synchronise.
1. Chacun crée **sa propre branche** depuis `main` : `git switch -c <pseudo>-contrib`.
2. Chacun ajoute un fichier `contributions/<pseudo>.md` décrivant une chose apprise de la séquence Git, puis **modifie de son côté** la ligne `Dernière mise à jour : …` du README pour y porter son pseudo. Validation, poussée, pull request ouverte avec le gabarit de la section Gabarit de pull request et checklist de revue.
3. Chaque pull request est **relue par l'autre** avant toute fusion : un commentaire, une demande ou une approbation, consignés sur la demande.
4. Fusionnez la première pull request, puis tentez la seconde : le conflit apparaît. Reproduisez-le en local depuis **votre** branche : `git pull origin main` (après avoir déclaré `git config --global pull.rebase false`, cf. §8) — les marqueurs étudiés dans Les bases de Git (section « Conflits de fusion ») apparaissent dans votre copie de travail.
5. Résolution **à deux, en personne** : identifier les deux régions du conflit et trancher, éditer, `git add` pour marquer la résolution, commiter la fusion.
6. Accessoire : une modification non commitée avant un changement de branche s'annule par `git restore <fichier>`, puis `git switch` remet en place — la grammaire moderne établie par la leçon précédente.

Corrigé de référence. La simulation ci-dessous produit les mêmes objets, les mêmes refus et les mêmes marqueurs que votre binôme constatera sur la forge : un serveur nu, deux clones, un fichier partagé `journal.md` dont la ligne d'en-tête est modifiée des deux côtés. La première cellule définit `SERVEUR2` et les deux clones ; en réexécution isolée, repartir d'elle.

```
SERVEUR2=/tmp/git-wf-serveur.git
rm -rf $SERVEUR2 /tmp/git-wf-a /tmp/git-wf-b
git config --global init.defaultBranch main
```

```
git config --global user.name "Atelier Git"
git config --global user.email "atelier.git@example.invalid"
git config --global pull.rebase false
git init --bare $SERVEUR2
git clone $SERVEUR2 /tmp/git-wf-a
cd /tmp/git-wf-a
printf 'Journal du binôme.\n\nDernière mise à jour : atelier\n' > journal.md
git add journal.md
git commit -m "docs: initialiser le journal partagé"
git push -u origin main
git clone $SERVEUR2 /tmp/git-wf-b
```

La simulation suppose un `pull` à fusion explicite ; depuis Git 2.27, ce comportement doit être déclaré (`pull.rebase false`), sans quoi Git refuse d'arbitrer entre fusion et rebassage sur des branches divergentes.

Premier étudiant : il modifie la ligne et pousse.

```
cd /tmp/git-wf-a
printf 'Journal du binôme.\n\nDernière mise à jour : étudiant A\n' > journal.md
git commit -am "docs: journal mis à jour par A"
git push origin main
```

```
[main badd308] docs: journal mis à jour par A
 1 file changed, 1 insertion(+), 1 deletion(-)
Enumerating objects: 5, done.
Counting objects:  20% (1/5)
Counting objects:  40% (2/5)
Counting objects:  60% (3/5)
Counting objects:  80% (4/5)
Counting objects: 100% (5/5)
Counting objects: 100% (5/5), done.
Delta compression using up to 32 threads
Compressing objects:  50% (1/2)
Compressing objects: 100% (2/2)
Compressing objects: 100% (2/2), done.
Writing objects:  33% (1/3)
Writing objects:  66% (2/3)
Writing objects: 100% (3/3)
Writing objects: 100% (3/3), 328 bytes | 328.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To /tmp/git-wf-serveur.git
   06c997d..badd308  main → main
```

Second étudiant, parti de l'état antérieur, fait de même dans son clone :

```
cd /tmp/git-wf-b
printf 'Journal du binôme.\n\nDernière mise à jour : étudiant B\n' > journal.md
git commit -am "docs: journal mis à jour par B"
```

```
[main 838ce95] docs: journal mis à jour par B
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Sa poussée est refusée : le clone `b` ignore le commit de `a` et sa proposition l'effacerait. La cellule est marquée `error: true` pour que le refus reste visible au rendu :

```
cd /tmp/git-wf-b
git push origin main
```

```
To /tmp/git-wf-serveur.git
 ! [rejected]        main → main (fetch first)
error: failed to push some refs to '/tmp/git-wf-serveur.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

```
: 1
```

L'étudiant qui pull récupère alors le commit de `a` et tente la fusion implicite du `pull` : impossible de décider automatiquement, la même ligne a bougé des deux côtés. La cellule bute sur le conflit, et le rendu conserve le diagnostic :

```
cd /tmp/git-wf-b
git pull origin main
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects:  20% (1/5)
remote: Counting objects:  40% (2/5)
remote: Counting objects:  60% (3/5)
remote: Counting objects:  80% (4/5)
remote: Counting objects: 100% (5/5)
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects:  50% (1/2)
```

```
remote: Compressing objects: 100% (2/2)
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects:  33% (1/3)
Unpacking objects:  66% (2/3)
Unpacking objects: 100% (3/3)
Unpacking objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.
From /tmp/git-wf-serveur
 * branch            main       → FETCH_HEAD
   06c997d..badd308  main       → origin/main
Auto-merging journal.md
CONFLICT (content): Merge conflict in journal.md
Automatic merge failed; fix conflicts and then commit the result.
```

```
: 1
```

Le fichier en conflit contient désormais les deux régions étudiées dans Les bases de Git — la version du commit courant et la version entrante — séparées par `=======` :

```
cd /tmp/git-wf-b
cat journal.md
```

```
Journal du binôme.

<<<<<<< HEAD
Dernière mise à jour : étudiant B
=======
Dernière mise à jour : étudiant A
>>>>>>> badd3086a03f3ad321183e418245bc3444be1300
```

Remarque. Le style par défaut de Git montre deux candidats ; `git config merge.conflictstyle diff3` (documenté dans `git-config(1)`) ajoute la base commune entre `|||||||` et `=======`.

Résoudre, ce n'est pas choisir un camp : c'est produire un contenu que les deux auteurs acceptent. Le journal garde les deux mentions, ordonnées dans le temps :

```
cd /tmp/git-wf-b
cat > journal.md <<'EOF'
Journal du binôme.

Dernière mise à jour : étudiant A
```

```
Dernière mise à jour : étudiant B
EOF
git add journal.md
git commit --no-edit
git push origin main
```

```
[main a846b73] Merge branch 'main' of /tmp/git-wf-serveur
Enumerating objects: 10, done.
Counting objects:  10% (1/10)
Counting objects:  20% (2/10)
Counting objects:  30% (3/10)
Counting objects:  40% (4/10)
Counting objects:  50% (5/10)
Counting objects:  60% (6/10)
Counting objects:  70% (7/10)
Counting objects:  80% (8/10)
Counting objects:  90% (9/10)
Counting objects: 100% (10/10)
Counting objects: 100% (10/10), done.
Delta compression using up to 32 threads
Compressing objects:  25% (1/4)
Compressing objects:  50% (2/4)
Compressing objects:  75% (3/4)
Compressing objects: 100% (4/4)
Compressing objects: 100% (4/4), done.
Writing objects:  16% (1/6)
Writing objects:  33% (2/6)
Writing objects:  50% (3/6)
Writing objects:  66% (4/6)
Writing objects:  83% (5/6)
Writing objects: 100% (6/6)
Writing objects: 100% (6/6), 633 bytes | 633.00 KiB/s, done.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
To /tmp/git-wf-serveur.git
   badd308..a846b73  main → main
```

La fusion est achevée ; son tracé se vérifie dans le graphe :

```
cd /tmp/git-wf-b
git log --oneline --graph -6
```

```
*   a846b73 (HEAD → main, origin/main, origin/HEAD) Merge branch 'main' of /
tmp/git-wf-serveur
|\
| * badd308 docs: journal mis à jour par A
* | 838ce95 docs: journal mis à jour par B
|/
* 06c997d docs: initialiser le journal partagé
```

Barème. La grille d'évaluation combine les six critères de revue (§4) et les étapes 2–5 de la consigne ; chaque ligne du barème se décide à la lecture de la pull request ou du dépôt, sans interprétation.

Critère	Preuve attendue	Points
Titre de la PR conforme à Conventional Commits	Page de la PR	1
Gabarit complété : les quatre rubriques	Page de la PR	2
Un fichier touché, une ligne de justification	Rubrique Modifications	1
Vérification énoncée et réexécutable	Rubrique Vérification	1
Pipeline de CI verte	Affichage de la CI sur la PR	1
Relecture par l'autre : commentaire puis approbation	Fil de discussion de la PR	2
Conflit reproduit puis résolu, marqueur absent du fichier final	Contenu du README après fusion	2
Fusion tracée sur `main`	`git log --graph`	2

Total : douze points. Le barème accorde délibérément plus de poids à la trace de relecture et à la résolution du conflit qu'à la dextérité sur la ligne de commande : c'est le comportement d'équipe qui est évalué, non la vitesse d'exécution.

9 Section bonus : simuler un flux Gitflow

Pour éprouver physiquement la chorégraphie de Gitflow, sans forge. Appliquez chaque mouvement dans le dépôt de travail cloné du dépôt nu et vérifiez, après

chaque phase, ce que montre `git log --graph --oneline`. Les trois mouvements à nommer : une branche de release naît de `develop` et retourne aux deux branches permanentes ; un hotfix naît de `main` et retourne aux deux ; chaque retour laisse un commit de fusion et, sur `main`, un tag.

```
# Bloc non exécuté par le rendu — patron de commandes.
# installation en deux temps : (1) dépôt nu central
git config --global init.defaultBranch main
git config --global user.name "Atelier Git"
git config --global user.email "atelier.git@example.invalid"
git init --bare /tmp/git-wf-bonus.git
# (2) clone de travail — origin est donc définie
git clone /tmp/git-wf-bonus.git /tmp/git-wf-bonus
cd /tmp/git-wf-bonus

# deux branches permanentes
git switch -c develop
git push -u origin develop

# une fonctionnalité complète
git switch -c feature/connexion develop
# ... validations sur la branche ...
git switch develop
git merge --no-ff feature/connexion

# une release
git switch -c release/1.0 develop
echo "Notes de version 1.0" > RELEASE-NOTES.md; git add RELEASE-NOTES.md; git
commit -m "release: préparer la 1.0"
git switch main
git merge --no-ff release/1.0
git tag -a v1.0 -m "Version 1.0"
git switch develop
git merge --no-ff release/1.0

# un correctif urgent, né de main
git switch main
git switch -c hotfix/1.0.1
echo "Correctif 1.0.1 : régression d'authentification corrigée" >> RELEASE-
NOTES.md; git add RELEASE-NOTES.md; git commit -m "fix: corriger la régression
d'authentification"
git switch main
git merge --no-ff hotfix/1.0.1
git tag -a v1.0.1 -m "Correctif 1.0.1"
```

```
git switch develop
git merge --no-ff hotfix/1.0.1
```

Critère de réussite : les deux tags `v1.0` et `v1.0.1` portent sur `main` ; `develop` contient les fusions de la release et du hotfix ; aucun commit direct de fonctionnalité n'apparaît sur `main` en dehors de ces fusions.

10 Conclusion et ressources

Un workflow produit ses effets à trois conditions, chacune vérifiable : formalisation dans le README du projet — les quatre objectifs de ce début de leçon, transcrits en règles d'équipe ; protection de la branche principale côté serveur — push direct interdit, relecture et pipeline exigées ; vérification automatisée à chaque pull request et à chaque fusion, avec la commande qui convient au projet, dans une liste close : `make test`, `mvn verify`, `pytest`, ou l'équivalent de votre outillage. Un workflow ni écrit ni appliqué cesse d'être un workflow : il devient une habitude que l'on croit collective.

Parmi les six modèles, la réponse par défaut du cours tient en une phrase : le rythme de GitHub Flow avec la discipline de relecture du feature branch workflow, c'est-à-dire des branches courtes et des pull requests relues. Le forking s'impose dès que les contributeurs n'ont pas de droits ; Gitflow répond à un rythme de versions numérotées ; le trunk-based suppose une maturité de CI que l'on annonce rarement acquis d'avance.

Ressources. Six lectures, toutes citées par le corpus du cours ; la dernière est un reflexe de terminal, non un site.

- Pro Git, 2ᵉ édition, version en ligne, licence CC BY-NC-SA — https://git-scm.com/book/fr/v2
- Guide du GitHub flow — https://guides.github.com/introduction/flow/
- Vincent Driessen, A Successful Git Branching Model (2010) — http://nvie.com/posts/a-successful-git-branching-model/
- Comparaison des workflows par Atlassian — https://www.atlassian.com/fr/git/tutorials/comparing-workflows/gitflow-workflow
- Trunk-based development — https://trunkbaseddevelopment.com/
- Documentation officielle de GitHub CLI — https://cli.github.com/manual/ (Source : manuel GitHub CLI, cli.github.com/manual.) — la version de `gh` installée dans l'image du cours se relève par `gh --version`.
- Le glossaire officiel, en terminal : `gitglossary(7)`, et sa porte d'entrée `git(1)`.