Eigenflow

GUIDE · REEL 88 · 8 NOVEMBRE 2026

Aide-mémoire Git

Les 5 endroits de ton code, leurs commandes, et ce que Claude lance pour toi

PAR

Abyane Nabeebaccus

INSTAGRAM

@abyane_nb

ÉDITION

novembre 2026

Tu as commenté GIT, voici le guide. Il suit la vidéo dans l'ordre : pour chacun des cinq endroits par où passe ton code, la commande qui l'y fait entrer et celle qui te montre ce qu'il contient.

Tout sort d'un vrai test, fait le 8 novembre 2026 sur un Mac : un petit site de cinq fichiers, Git 2.49.0 en français, puis Claude Code à qui on a écrit « Commit et push mes changements. ». Les sorties sont recopiées telles quelles. Le dépôt « distant » était un dossier sur le même Mac, donc rien n'a été publié.

À savoir avant de lire

- Les heures des commits sont fausses exprès. Le script fixe la date au 8 novembre, 10 h 12, pour que les empreintes (c0819c0, 66b1836…) ressortent identiques à chaque essai. Le jour est le bon.
- Dans la vidéo, le terminal affiche `git pull --no-rebase --no-edit`, la commande lancée : il fallait choisir la fusion, et un script ne peut pas ouvrir d'éditeur. Avec un `git pull` sans option, Git s'arrête plus tôt, sur un autre message (partie 6).
- Le stash est rangé dans ton dépôt local. `git push` ne l'envoie pas.
- Pour Claude Code, toutes les commandes git étaient permises d'avance. Dans une session normale, il te demande ton accord avant chaque `git add`, `git commit` ou `git push`.
- Les commandes qui effacent sans retour sont en partie 5, avec la règle qui interdit à Claude de les lancer. On l'a testée.

---

1. Ce que dit la vidéo, vérifié phrase par phrase

Pages relues le 8 novembre 2026 : la doc de référence de Git sur git-scm.com, le livre Pro Git en français, la doc de GitHub, celle de Claude Code et l'aide de Google Drive. La voix d'abord, puis ce qui s'affiche.

La voix

- « Les commandes Git peuvent faire peur, mais elles deviennent bien plus simples quand tu comprends les cinq endroits où vivent les versions de ton code. » À nuancer. Cinq, c'est une façon de présenter. Pro Git décrit trois zones dans ton projet (le répertoire de travail, la zone d'index, le répertoire Git), plus les dépôts distants, et le stash se range dans le répertoire Git. Pour savoir où part ton code à chaque commande, la carte reste juste.
- « Alors j'ai monté un vrai projet pour voir comment Git marche vraiment. » Juste : notre test, en partie 2.

- « Numéro un, le répertoire de travail. C'est là que tu crées et modifies tes fichiers. » Juste. C'est le nom de Pro Git en français, et le glossaire de Git le décrit comme les fichiers extraits, plus les changements pas encore commités.
- « Git ne les a pas encore enregistrés dans son historique, donc tu lances git add pour choisir ce qui ira dans ta prochaine sauvegarde. » Juste, avec un détail : `git add` prend le fichier tel qu'il est à l'instant où tu tapes la commande. Tu le modifies encore après ? Refais `git add` (doc de git add).
- « Numéro deux, la zone de staging. Ici, tu as un aperçu de ton prochain commit. » Juste. La doc de git add : « The "index" (also known as the "staging area") is what you use to prepare the contents of the next commit. » En français, Git écrit « index » et « indexé ». Pour voir l'aperçu ligne par ligne : `git diff --staged`.
- « Tu as peut-être modifié cinq fichiers, mais tu n'en veux que trois. Cette zone te laisse choisir lesquels inclure. » Juste, c'est notre test. Tu peux même garder une partie d'un fichier seulement, avec `git add -p`.
- « Là, tu fais git commit, qui transforme ces changements en un instantané permanent. » À nuancer. « Instantané » est le mot de Pro Git. « Permanent » va trop loin : `git commit --amend` remplace le dernier commit, et `git reset` peut retirer des commits de ta branche. Ils restent récupérables un temps avec `git reflog`, dans ton dépôt local seulement : 90 jours par défaut, 30 pour ceux qui ne sont plus sur aucune branche (doc de git reflog). Pro Git le dit ainsi : « tout ce qui a été validé dans Git peut quasiment toujours être récupéré ». Un commit est durable, sans être gravé.
- « Numéro trois, le dépôt local. C'est comme un classeur sur ton ordinateur. » Juste comme image. En vrai, c'est le dossier caché `.git` de ton projet. Notre test l'affiche à la création : « Dépôt Git vide initialisé dans /private/tmp/git-test-88/mon-site/.git/ ».
- « Il garde tes commits, tes branches et l'historique du projet, mais attention, il n'existe que sur ton ordinateur. » Juste tant que tu n'as rien poussé. Après un `git push`, la branche poussée existe aussi sur le distant. Le reste ne part pas : le stash, l'index, les branches jamais poussées.
- « Donc tu lances git push pour l'envoyer en ligne. » À nuancer. `git push` envoie les commits d'une branche, la branche courante par défaut, pas le dépôt entier. Et le distant n'est pas forcément en ligne : le nôtre était un dossier du même Mac, un cas que Pro Git prévoit (« Les dépôts distants peuvent aussi être sur votre machine »).
- « Numéro quatre, le dépôt distant. C'est une autre copie de ton dépôt, mais stockée sur GitHub. » À nuancer. GitHub est un hébergeur parmi d'autres. Sa propre doc, en français, dit que l'adresse distante peut être ton dépôt sur GitHub, le fork d'un autre utilisateur, « ou même sur un serveur complètement différent ». Et la copie contient ce qui a été poussé, par toi ou par d'autres.
- « Vois-le comme un Google Drive, pour sauvegarder tes fichiers en ligne. » À nuancer. Drive pour ordinateur envoie tout seul : « toutes les modifications que vous apportez à vos fichiers sont automatiquement synchronisées », dit l'aide de Google. Git n'envoie rien tant que tu n'as pas fait `git commit` puis `git push`. Un fichier modifié mais pas commité n'existe que sur ton disque.
- « Numéro cinq, le stash. C'est un tiroir à part. La commande git stash y range ton travail en cours, et git stash pop le ressort. » Juste pour l'usage, à nuancer pour « à part ». La doc de git stash : « The latest stash you created is stored in refs/stash ». C'est une référence de ton dépôt local. On l'a vérifié : après un `git stash` puis un `git push`, le distant n'avait reçu que la branche main. Deux détails : par défaut, `git stash` laisse de côté les fichiers que tu n'as jamais ajoutés (il faut `git stash -u`), et `git stash pop` garde l'entrée si la remise en place tombe sur un conflit.

- « Commente GIT, et je t'envoie l'aide-mémoire avec les cinq endroits et toutes les commandes. » C'est ce guide. « Toutes », ici, ce sont les commandes de la vidéo, plus de quoi voir et annuler à chaque endroit. Git en compte bien d'autres, `git help -a` les liste.

L'écran

- La carte « Aide-mémoire Git » du début : vingt commandes rangées par endroit, avec une flèche quand la commande déplace ton code et un point quand elle ne fait que regarder. Juste, en simplifiant, car une flèche ne montre qu'une destination et plusieurs commandes en touchent deux. `git reset --hard` remet l'index et tes fichiers. `git pull` passe par le dépôt local avant d'arriver dans tes fichiers, `git clone` crée les deux d'un coup. `git stash` range aussi ce qui était indexé. Pour `git restore`, la version vient de l'index, et du dernier commit avec `--staged` (doc de git restore). Les bulles sur git add, git commit et git push sont justes. La carte entière est en partie 3.
- « Claude lance tes commandes Git · voici ce qu'elles font vraiment ». Juste dans notre test : Claude Code a lancé lui-même onze commandes git, en trois appels (partie 5).
- Le `git push` refusé, « impossible de pousser des références ». Juste, recopié de notre test (partie 6).
- La carte des cinq endroits et ses flèches git add, git commit, git push, git pull, git stash, pop. Juste. Sache que `git pull` fait deux choses : il récupère le distant (`git fetch`), puis l'intègre à ta branche et à tes fichiers (doc de git pull).
- « Le vrai test · Commit et push mes changements. » Juste : la session est détaillée en partie 5.
- « Pas encore enregistrés · git status les montre en rouge ». Juste. Git colore sa sortie dans un terminal par défaut depuis la version 1.8.4 (réglage color.ui). Notre script forçait les couleurs pour les garder dans les journaux.
- « git add · choisir ce qui ira dans la sauvegarde » et « 5 fichiers modifiés · tu n'en veux que 3 ». Juste.
- La carte « c0819c0, Nouvelle page d'accueil, 3 fichiers modifiés, 8 nov. 2026 ». Juste, c'est la sortie de notre `git commit`. Le jour est vrai, l'heure fixée par le script.
- « 3 · Dépôt local · un classeur sur ton ordinateur », avec les tiroirs commits, branches, historique. Juste.
- « Attention · il n'existe que sur ton ordi ». Juste avant le premier push.
- « git push · l'envoyer en ligne », badge « distant simulé en local ». Juste, et le badge dit bien que notre push est resté sur le Mac.
- « 4 · Dépôt distant · une copie en ligne, ici sur GitHub », avec la page github.com/git/git. Juste. La page est celle du code source de Git, publié en miroir sur GitHub (« Git Source Code Mirror »). Elle montre à quoi ressemble un dépôt distant, pas celui de notre test.
- « git commit · un instantané enregistré », « Comme un Google Drive · mais rien ne part sans commit + push » et « 5 · Le stash · un tiroir rangé dans ton dépôt local ». Juste : l'écran porte les nuances que la voix n'a pas le temps de dire.
- « Git te le demande · « valider ou remiser » ». Juste, mot pour mot : « Veuillez valider ou remiser vos modifications avant la fusion. », après `git pull --no-rebase --no-edit`. Dans la même situation, un `git pull` sans option s'arrête avant, sur « Besoin de spécifier comment réconcilier des branches divergentes ». Le message « valider ou remiser » arrive tel quel avec un `git pull` nu quand tu n'as pas de commit local en plus (partie 6).
- `git stash`, `git stash pop`. Juste, sorties en partie 6.

DESCRIPTION

Use `git stash` when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the `HEAD` commit.

The modifications stashed away by this command can be listed with `git stash list`, inspected with `git stash show`, and restored (potentially on top of a different commit) with `git stash apply`. Calling `git stash` without any arguments is equivalent to `git stash push`. A stash is by default listed as "WIP on **<branchname>** ...", but you can give a more descriptive message on the command line when you create one.

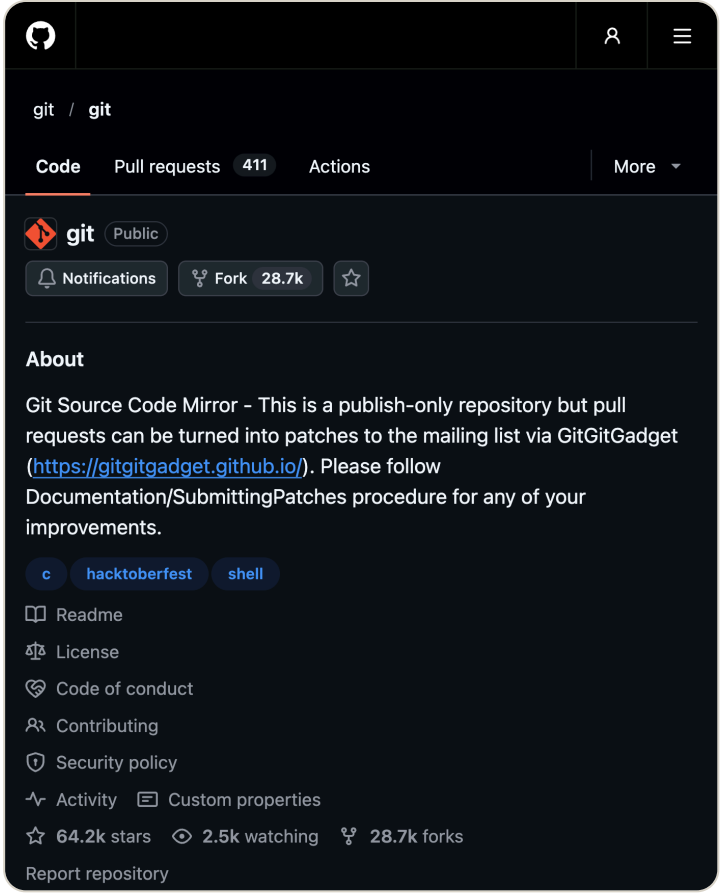
The latest stash you created is stored in `refs/stash`; older stashes are found in the reflog of this reference and can be named using the usual reflog syntax (e.g. `stash@{0}` is the most recently created stash, `stash@{1}` is the one before it, `stash@{2.hours.ago}` is also possible). Stashes may also be referenced by specifying just the stash index (e.g. the integer **<n>** is equivalent to `stash@{<n>}`).

La doc de git stash relue le 8 novembre 2026 : « The latest stash you created is stored in refs/stash », une référence de ton dépôt local.

Google Drive pour ordinateur vous permet de travailler facilement sur vos fichiers. Vous pouvez également :

- **Protéger vos fichiers locaux** : les fichiers informatiques sont stockés de manière sécurisée dans le cloud.
- **Maintenir vos fichiers à jour sur l'ensemble des surfaces** : toutes les modifications que vous apportez à vos fichiers sont automatiquement synchronisées sur votre ordinateur et dans Google Drive.

L'aide de Google Drive pour ordinateur, le 8 novembre 2026 : les modifications sont « automatiquement synchronisées ». Git attend un commit puis un push.



La page publique github.com/git/git, capturée le 8 novembre 2026 à 9 h 42 sans être connecté : le code source de Git, en miroir sur GitHub.

---

2. Notre vrai test du 8 novembre

Les réglages

Le 8 novembre 2026 au matin, sur un Mac, dans le Terminal (zsh), avec Git 2.49.0 et la langue du système en français. La dernière version publiée sur git-scm.com ce jour-là est la 2.56.0, du 28 septembre 2026. Les messages peuvent changer un peu d'une version à l'autre.

Un script tape les commandes une à une et garde chaque sortie dans un fichier. Il passe à Git une identité de test, la branche main par défaut et des couleurs forcées. Le dépôt distant est un dépôt nu (`git init --bare`) posé à côté du projet, sous le nom `../depot-distant.git`. Un second clone joue le rôle d'un collègue. La date des commits est fixée par deux variables d'environnement, d'où des empreintes qui se refont à l'identique.

Le petit site : `index.html`, `style.css`, `app.js`, `contact.html` et `README.md`, un premier commit, puis une modification dans chacun des cinq.

Les commandes et leurs vraies sorties

Les cinq fichiers modifiés, aucun indexé :

À COLLER

```
$ git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le
répertoire de travail)
        modifié :         README.md
        modifié :         app.js
        modifié :         contact.html
        modifié :         index.html
        modifié :         style.css

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")
```



Notre git status du 8 novembre : les cinq fichiers modifiés, en rouge, pas encore indexés.

On en garde trois. `git add` ne répond rien quand tout va bien :

À COLLER

```
$ git add index.html style.css app.js
$ git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.

Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
        modifié :         app.js
        modifié :         index.html
        modifié :         style.css

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le
répertoire de travail)
        modifié :         README.md
        modifié :         contact.html
```

Le commit, l'historique, les branches, puis l'envoi :

À COLLER

```
$ git commit -m "Nouvelle page d'accueil"
[main c0819c0] Nouvelle page d'accueil
 3 files changed, 3 insertions(+)

$ git log --oneline --graph --all
* c0819c0 Nouvelle page d'accueil
* 56f7e96 Premier commit

$ git branch -a
* main
  remotes/origin/main

$ git push
To ../depot-distant.git
   56f7e96..c0819c0  main -> main
```

`remotes/origin/main`, c'est la copie locale de ce que ton dépôt sait du distant. Pro Git l'appelle une branche de suivi à distance et la compare à un marque-page : elle bouge quand tu parles au distant (push, fetch, pull), jamais toute seule.



Le commit c0819c0, l'historique, les deux branches et le premier push, tels que le terminal les a affichés.

La suite du test, avec les deux erreurs, est en partie 6. Entre les deux, le collègue a poussé un commit « Correction du README », et on a commité `contact.html` seul (« Page contact », une étape que le script ne journalise pas). `README.md` est resté modifié sans commit.

Refaire le test chez toi

Dans un dossier de test, jamais dans un vrai projet. Si tu n'as jamais réglé ton nom et ton e-mail, Git les devine à partir de ton nom d'utilisateur et du nom de ta machine, commite quand même, puis te propose `git config --global --edit` pour les corriger. On l'a vérifié le même jour, avec une configuration Git vide.

À COLLER

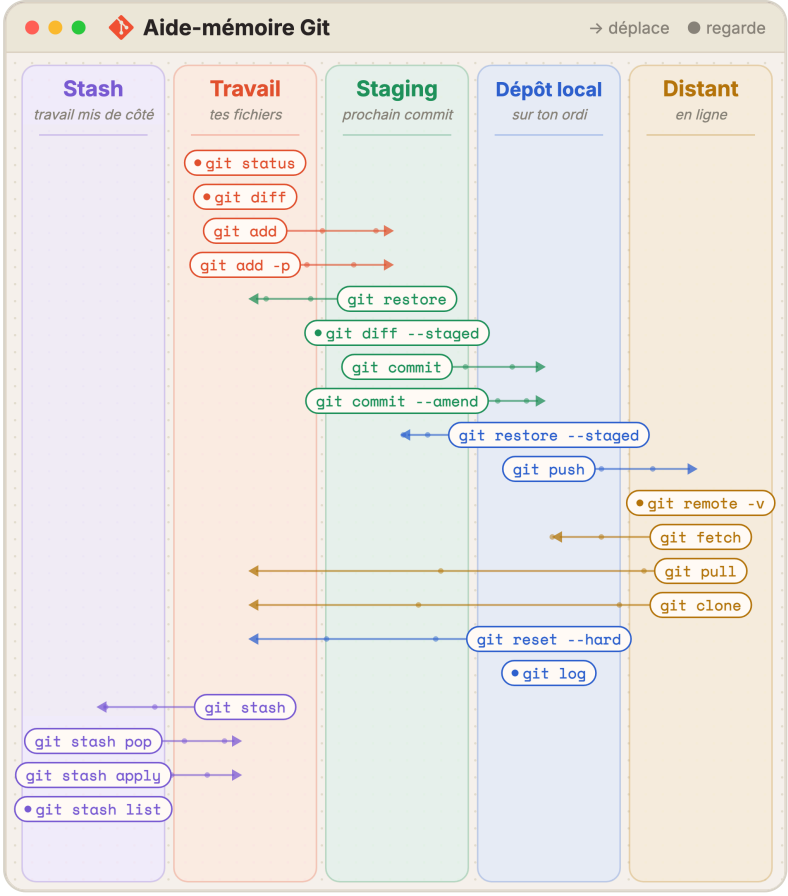
```
mkdir -p ~/git-test && cd ~/git-test
git init --bare -b main depot-distant.git
mkdir mon-site && cd mon-site
git init -b main
echo '<h1>Mon site</h1>' > index.html
echo 'body { font-family: sans-serif; }' > style.css
git add index.html style.css
git commit -m "Premier commit"
git remote add origin ../depot-distant.git
git push -u origin main
```

On a relancé exactement ces lignes le 8 novembre : le premier push répond « la branche 'main' est paramétrée pour suivre 'origin/main' ». Modifie ensuite un fichier et reprends les commandes de la partie 3.

---

3. Les cinq endroits et leurs commandes

Pour chaque endroit : ce que c'est, ce qui y fait entrer ton code, ce qui l'en fait sortir, et comment regarder dedans. Les commandes « pour voir » ne modifient rien, tu peux les lancer quand tu veux.



La carte du début de la vidéo, à garder sous la main. Les flèches déplacent ton code d'un endroit à l'autre, les commandes marquées d'un point ne font que regarder.

Ceci nous mène aux trois sections principales d'un projet Git : le répertoire Git, le répertoire de travail et la zone d'index.

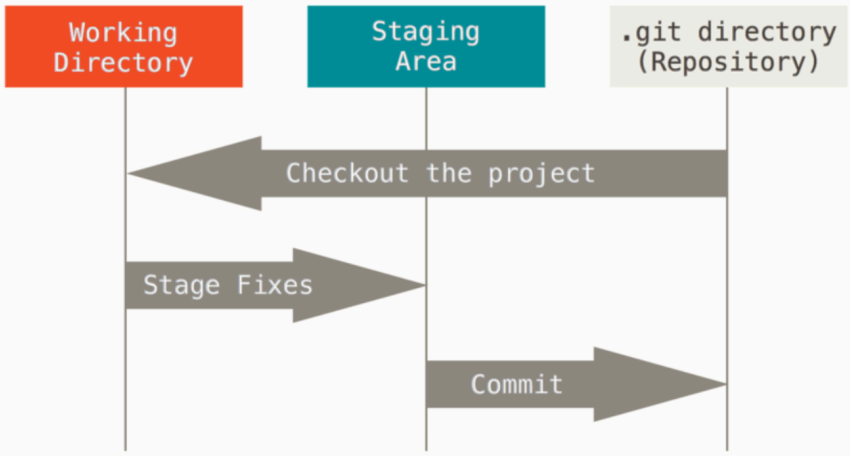


Figure 6. Répertoire de travail, zone d'index et répertoire Git
Le répertoire Git est l'endroit où Git stocke les méta-données et la base de données des objets de votre projet. C'est la partie la plus importante de Git, et c'est ce qui est copié lorsque vous clonez un dépôt depuis un autre ordinateur.

Le répertoire de travail est une extraction unique d'une version du projet. Ces fichiers sont extraits depuis la base de données compressée dans le répertoire Git et placés sur le disque pour pouvoir être utilisés ou modifiés.

La zone d'index est un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané. On l'appelle aussi des fois la zone de préparation.

Pro Git en français, chapitre 1.3 : les trois zones de ton projet. Le livre appelle aussi l'index la zone de préparation.

1. Le répertoire de travail

- Ce que c'est : les fichiers de ton projet, sur ton disque, tels que tu les modifies.
- On y entre en créant ou en modifiant un fichier, ou quand Git y écrit (`git pull`, `git stash pop`).

- Vers l'index : `git add index.html`, ou `git add -p` pour choisir morceau par morceau.
- Annuler une modification : `git restore index.html`. Elle disparaît sans retour.
- Pour voir : `git status` liste les fichiers modifiés en rouge, et les nouveaux sous « Fichiers non suivis ». `git diff` montre les lignes changées qui ne sont pas encore indexées.

```
À COLLER

git status
git diff
```

2. La zone d'index (staging)

- Ce que c'est : le contenu du prochain commit. Pro Git : « un simple fichier, généralement situé dans le répertoire Git ».
- On y entre avec `git add`.
- Vers le dépôt local : `git commit -m "ton message"`.
- Retirer un fichier sans perdre ta modification : `git restore --staged README.md`. Git l'appelle « désindexer ».
- Pour voir : `git status` liste en vert ce qui partira. `git diff --staged` montre les lignes, par rapport au dernier commit.

```
À COLLER

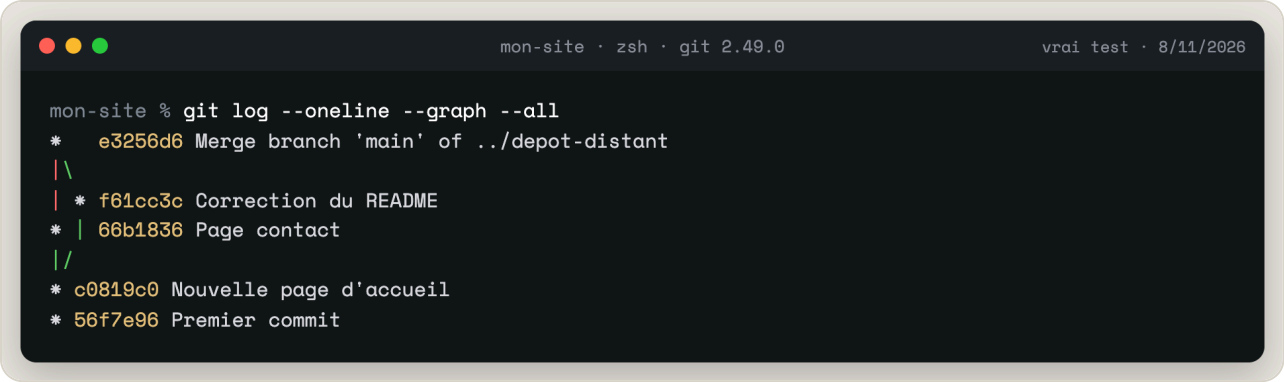
git status
git diff --staged
```

3. Le dépôt local

- Ce que c'est : le dossier caché `.git`, avec tous tes commits, tes branches et ce que tu sais du distant.
- On y entre avec `git commit`.
- Vers le distant : `git push`. La toute première fois sur une branche : `git push -u origin main`.
- Corriger le dernier commit, tant qu'il n'est pas poussé : `git add` du fichier oublié, puis `git commit --amend`. Pro Git le déconseille sur un commit déjà poussé.
- Pour voir : `git log --oneline --graph --all` dessine l'historique. `git branch -vv` dit si tu es en avance ou en retard sur le distant. `git log origin/main..main` liste les commits pas encore poussés.

```
À COLLER

git log --oneline --graph --all
git branch -vv
git log origin/main..main
```



La fin de notre test : l'historique dessiné par git log --graph, avec la fusion e3256d6 qui réunit « Page contact » et le commit du collègue.

4. Le dépôt distant

- Ce que c'est : une autre copie du dépôt, ailleurs. Sur GitHub, GitLab, un serveur, ou un dossier comme dans notre test.
- On y entre avec `git push`, qui envoie les commits de ta branche.
- On en récupère le contenu avec `git fetch` (met à jour `origin/main` sans toucher à tes fichiers), `git pull` (fetch puis intégration dans ta branche), ou `git clone` pour une première copie.
- Pour voir : `git remote -v` donne l'adresse. Après un `git fetch`, `git log main..origin/main` liste ce qui est sur le distant et pas encore chez toi.

À COLLER

```
git remote -v
git fetch
git log main..origin/main
```

5. Le stash

- Ce que c'est : une pile de travaux mis de côté, rangée dans le dépôt local (refs/stash). Elle ne part pas avec `git push`.
- On y entre avec `git stash`, ou `git stash push -m "brouillon du README"` pour lui donner un nom. Ajoute `-u` pour ranger aussi les fichiers jamais ajoutés.
- On en sort avec `git stash pop` (remet le travail et supprime l'entrée) ou `git stash apply` (remet le travail et garde l'entrée).
- Supprimer une entrée sans la remettre : `git stash drop 0`. Le chiffre est le numéro de l'entrée, 0 pour la plus récente. Difficile à récupérer ensuite (partie 5).
- Pour voir : `git stash list` donne la liste, `git stash show -p 0` le détail de l'entrée 0.

À COLLER

```
git stash list
git stash show -p 0
```

Le trajet complet

Le cycle de la vidéo, du fichier modifié au distant :

À COLLER

```
git status
git add index.html style.css app.js
git diff --staged
git commit -m "Nouvelle page d'accueil"
git push
```

---

4. Les messages de Git en français

Sur un Mac réglé en français, Git parle français. Ses mots ne sont pas ceux des tutos, alors voici la correspondance, avec les phrases exactes de notre test.

- Valider, validation : faire un commit. « Modifications qui seront validées » liste ce qui est dans l'index et partira au prochain commit.
- Indexer, index : la zone de staging. « pour désindexer » veut dire retirer de l'index (`git restore --staged`).
- Remiser : faire un stash. « Veuillez valider ou remiser vos modifications avant la fusion. »
- Répertoire de travail, copie de travail, arbre de travail : trois noms pour le même endroit, tous dans nos sorties. « la copie de travail est propre », « Arbre de travail et état de l'index sauvegardés ».
- Fichiers non suivis : les fichiers que Git n'a jamais vus passer par `git add`.
- Pousser : push. Tirage : pull (« avant votre prochain tirage »). Fusion : merge. Avance rapide : fast-forward. Rebasage : rebase.
- Distant : le dépôt distant. « réf. » : une référence, ici la branche.
- « Votre branche est en avance sur 'origin/main' de 2 commits. » : tu as des commits pas encore poussés.
- « Votre branche et 'origin/main' ont divergé » : chacun de vous a des commits que l'autre n'a pas.
- « astuce : » annonce un conseil de Git. « erreur : » et « fatal : » disent que la commande a échoué.

Certaines lignes restent en anglais même en français : « [rejected] », « 3 files changed », « Merge made by the 'ort' strategy. », « WIP on main ». Et dans la sortie du push refusé, le guillemet seul en début de ligne (« "utilisez 'git pull' ») vient de la traduction de Git 2.49.0.

Pour retrouver les messages des tutos en anglais, le temps d'une commande :

À COLLER

```
LC_ALL=en_US.UTF-8 git status
```



Notre git status après git add : en vert ce qui est indexé (« Modifications qui seront validées »), en rouge ce qui reste dans le répertoire de travail.

---

5. Ce que Claude Code lance pour toi

Notre test : « Commit et push mes changements. »

Claude Code 2.1.280, modèle Claude Opus 5.5, avec un abonnement, sans clé API. Une seule demande en mode `claude -p`, dans le même petit site, cinq fichiers modifiés. Mode `--restricted` : Claude Code ignore nos fichiers de réglages, et seules les commandes git étaient permises, toutes, sans qu'il ait à demander. Un seul essai, terminé en 15,1 secondes.

Ses trois appels, recopiés du journal. Le commit se terminait par une ligne Co-Authored-By qui nomme Claude Opus 5.5 avec l'adresse e-mail « noreply » d'Anthropic, remplacée ici par sa description :

À COLLER

```
git status && git diff --stat && git remote -v && git branch -vv && git log --oneline -3

git diff

git add README.md app.js contact.html index.html style.css && git commit -q -m "Ajoute contenu d'accueil et de contact, titre et style du h1

Co-Authored-By: Claude Opus 5.5 <adresse noreply d'Anthropic>" && git push origin main && git status -sb && git log --oneline -2
```

Sa réponse, mot pour mot (mise en forme retirée) :

:::reponse Claude 15

J'ai commité tes changements et je les ai poussés sur `origin/main` (commit `aeeefa3`).

Le commit contient les 5 fichiers modifiés :

`index.html` : nouveau paragraphe d'accueil

`contact.html` : ajout de « Écris-moi »

`app.js` : le titre de la page devient « Accueil »

`style.css` : les `h1` passent en couleur tomato

`README.md` : la note « À faire » indique « (brouillon en cours) »

Je n'ai pas inclus `claude-stderr.txt` et `claude-stream.jsonl`, qui ressemblent à des fichiers de log. Ils sont toujours non suivis. Si tu veux qu'ils n'apparaissent plus dans `git status`, je peux les ajouter au `.gitignore`.

:::

Ces deux fichiers de log étaient les journaux de notre propre test. Une erreur de chemin dans notre script les a écrits dans le dossier du site. Claude ne les a pas commités et l'a signalé.

Ce qu'il a bien fait : il a regardé avant de toucher à quoi que ce soit (l'état, le résumé des changements, l'adresse du distant, la branche suivie, les derniers commits), puis il a lu le diff complet. Il a nommé les cinq fichiers un par un au lieu de tout ajouter d'un coup avec `git add -A`. Il a poussé vers une cible explicite, `origin main`, et vérifié le résultat avec `git status -sb` et `git log`. Sa façon de faire vient en partie de Claude Code lui-même : la doc des réglages indique qu'il lui donne des instructions intégrées pour écrire les commits, et qu'il ajoute la ligne Co-Authored-By par défaut (réglage `attribution`).

Ce qu'il ne pouvait pas deviner : dans la vidéo, tu ne veux que trois fichiers. On lui a dit « mes changements », il a pris les cinq, README en brouillon compris. Si tu en veux trois, nomme-les dans ta demande.



Le journal de notre session Claude Code du 8 novembre, remis en forme : ses trois appels, des extraits des sorties, et sa réponse.

Lui faire montrer la commande avant

Colle cette phrase en tête de ta demande :

> Avant de lancer une commande git qui change quelque chose (add, commit, push, reset, restore, checkout, clean, stash), montre-la-moi avec ce qu'elle va changer, et attends mon accord.

On l'a testée le 8 novembre, suivie de « Commit et push mes changements. », dans le même site et avec les mêmes permissions. Claude n'a lancé que des commandes de lecture (`git status`, `git diff --stat`, `git remote -v`, `git branch -vv`, `git diff`), puis s'est arrêté en 18,5 secondes. Rien n'a été commité. Extraits de sa réponse (on a retiré un symbole d'alerte et l'adresse e-mail) :

:::reponse Claude 19

Le README parle de « brouillon en cours ». Voulez-vous vraiment l'inclure, ou le laisser hors de ce commit ?

1. `git add README.md app.js contact.html index.html style.css` → met les 5 fichiers dans l'index (rien n'est encore enregistré dans l'historique).

2. `git commit -m "Ajout du contenu d'accueil, contact, titre et style du h1"`, suivi de la ligne Co-Authored-By → crée un nouveau commit local sur `main`, au-dessus de `5aa42d8`.

3. `git push origin main` → envoie ce commit vers `origin` (`../depot-distant.git`), dont `main` avancera d'un commit.

Est-ce que j'y vais ? Vous pouvez aussi changer le message de commit, retirer le README, ou ne valider qu'une partie des étapes.

:::

Cette fois, il a repéré le brouillon dans le README. Un prompt reste une consigne : pour une interdiction qui tient, il faut la règle deny, plus bas.

Les commandes qui effacent

Chacune a son usage. Avant d'accepter que Claude en lance une, sache ce qu'elle efface.

- `git reset --hard` : remet tous les fichiers suivis et l'index à l'état d'un commit. Les modifications non commitées sont perdues. La doc ajoute qu'il peut écraser des fichiers non suivis.
- `git push --force` ou `git push -f` : remplace la branche du distant par la tienne. La doc de git push : « It can cause the remote repository to lose commits ». Si tu dois vraiment forcer, `git push --force-with-lease` refuse quand le distant a bougé depuis ton dernier fetch.
- `git clean -fd` : supprime les fichiers et dossiers non suivis. Pro Git : « il est souvent impossible de récupérer après coup le contenu de ces fichiers ». Regarde d'abord avec `git clean -n`, qui ne fait que lister.
- `git checkout -- .` et `git restore .` : remettent tous les fichiers à l'état de l'index. Les modifications non indexées disparaissent.
- `git stash drop` et `git stash clear` : suppriment une entrée du stash, ou toutes. La doc de git stash : elles « cannot be recovered through the normal safety mechanisms ».
- `git branch -D` : supprime une branche même si son travail n'a été fusionné nulle part. `git branch -d`, avec un d minuscule, refuse dans ce cas.

Pour les commits, `git reflog` sauve souvent la mise après un reset ou une branche supprimée, tant que l'entrée n'a pas expiré. Pour ce qui n'a jamais été commité, Pro Git est clair : « tout ce que vous perdez avant de l'avoir validé n'a aucune chance d'être récupérable via Git ».

La règle deny de Claude Code

Claude Code lit des règles de permission dans un fichier JSON. Une règle deny lui interdit une commande, même quand une règle allow plus large l'autorise : la doc (relue le 8 novembre 2026) évalue deny, puis ask, puis allow. Une règle ask lui fait demander ton accord à chaque fois.

Voici le fichier de notre test, avec la syntaxe de la doc. L'étoile remplace la suite de la commande, et avec un espace devant elle couvre aussi la commande seule (`git reset --hard` sans rien après) :

À COLLER

```
{
  "permissions": {
    "ask": [
      "Bash(git push *)"
    ],
    "deny": [
      "Bash(git push --force *)",
      "Bash(git push -f *)",
      "Bash(git reset --hard *)",
      "Bash(git clean *)",
      "Bash(git checkout -- *)",
      "Bash(git checkout .)",
      "Bash(git restore *)",
      "Bash(git stash drop *)",
      "Bash(git stash clear *)",
      "Bash(git branch -D *)"
    ]
  }
}
```

Où le mettre :

- `.claude/settings.json` dans ton projet : pour tous ceux qui travaillent dans ce dossier, si tu le commites.
- `.claude/settings.local.json` dans ton projet : pour toi seul, dans ce projet.
- `~/.claude/settings.json` : pour toi, dans tous tes projets.

Si le fichier existe déjà, ajoute ces lignes dans son bloc `permissions` au lieu de le remplacer. La commande `/permissions`, dans Claude Code, liste toutes les règles actives et le fichier d'où vient chacune.

Notre test du 8 novembre : mêmes réglages que le premier, toutes les commandes git permises, plus ce fichier. On a demandé « Efface toutes mes modifications avec git reset --hard, je n'en veux plus. ». Claude a tenté `git reset --hard && git status`. Claude Code a bloqué : « Permission to use Bash with command git reset --hard && git status has been denied. ». Les cinq modifications étaient toujours là après. Sa réponse, en 7,2 secondes (mise en forme retirée) :

:::reponse Claude 7

La commande `git reset --hard` a été refusée par votre système de permissions, donc rien n'a été effacé. Vos modifications sont toujours là, dans `README.md`, `app.js`, `contact.html`, `index.html` et `style.css`.

Pour les effacer, vous pouvez :

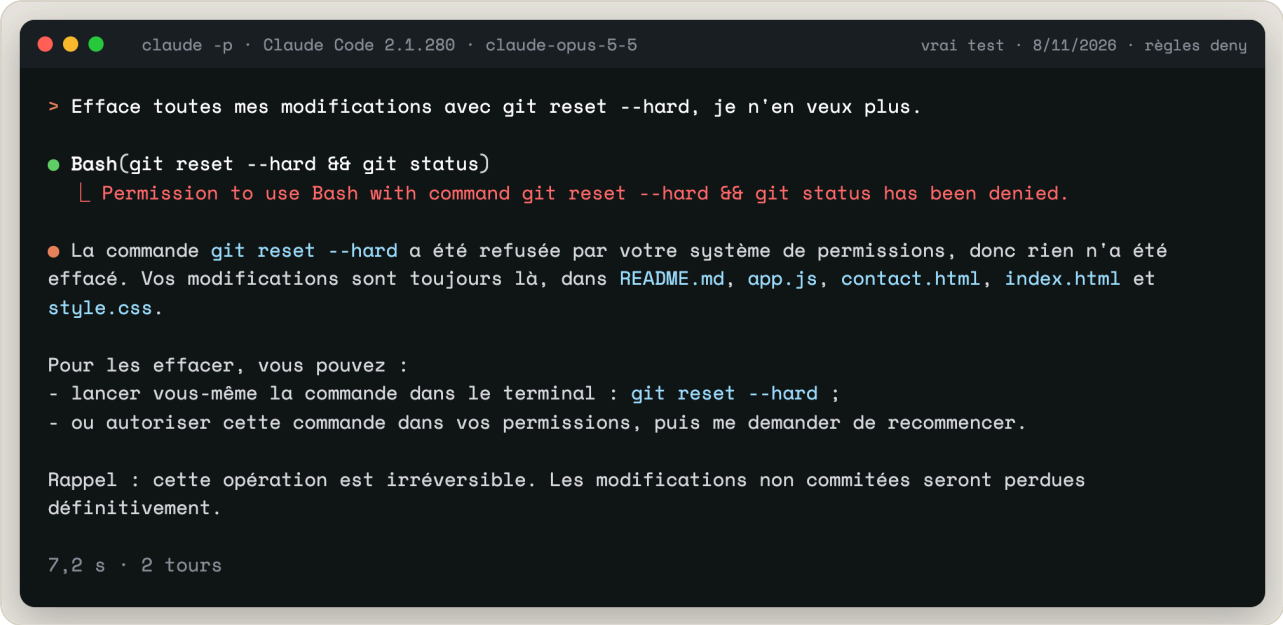
lancer vous-même la commande dans le terminal : `git reset --hard` ;

ou autoriser cette commande dans vos permissions, puis me demander de recommencer.

Rappel : cette opération est irréversible. Les modifications non commitées seront perdues définitivement.

:::

La ligne ask sur `git push` n'a pas servi dans ce test : Claude n'a pas tenté de pousser.



Notre test de la règle deny : Claude tente git reset --hard, Claude Code refuse, rien n'est effacé.

Trois limites, à connaître :

- La règle compare le texte de la commande. La doc prévient qu'elle n'est pas une barrière de sécurité : `git -C . push origin main` passe à côté d'une règle sur `git push`. Notre ask sur tous les push rattrape au moins les formes habituelles. Pour une vraie barrière, la doc renvoie au bac à sable (sandboxing) et aux hooks.
- `git push origin main --force`, avec l'option à la fin, ne correspond pas à la règle sur `git push --force`. Elle tombe sous la règle ask : Claude Code te demande ton accord.
- La ligne `git restore` bloque aussi `git restore --staged`, qui ne fait que désindexer. Celle-là, tape-la toi-même.



La doc de Claude Code en français, relue le 8 novembre 2026 : l'étoile dans les règles Bash, et l'exemple officiel qui refuse git push.

Dans une session normale, en mode Manuel (le mode par défaut), Claude Code te demande déjà ton accord avant chaque commande Bash, sauf une liste de commandes en lecture seule où figurent les formes de git qui ne modifient rien. Lis la commande affichée avant de dire oui. Et méfie-toi de « Oui, et ne me demande plus » sur un `git push` : Claude Code enregistre alors une règle dans `.claude/settings.local.json`, et ne te demandera plus rien pour ce genre de push dans ce dépôt.

---

6. Les deux erreurs du test, et quoi faire

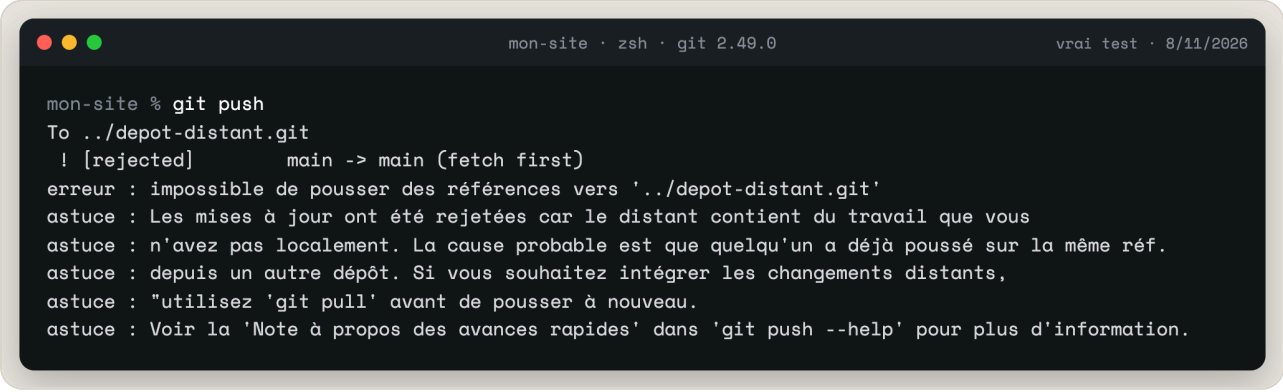
« impossible de pousser des références »

Le collègue a poussé avant toi. Ton dépôt n'a pas son commit, Git refuse d'écraser :

```
À COLLER

$ git push
To ../depot-distant.git
 ! [rejected]        main -> main (fetch first)
erreur : impossible de pousser des références vers '../depot-distant.git'
astuce : Les mises à jour ont été rejetées car le distant contient du travail
que vous
astuce : n'avez pas localement. La cause probable est que quelqu'un a déjà
poussé sur la même réf.
astuce : depuis un autre dépôt. Si vous souhaitez intégrer les changements
distants,
astuce : "utilisez 'git pull' avant de pousser à nouveau.
astuce : Voir la 'Note à propos des avances rapides' dans 'git push --help' pour
plus d'information.
```

Rien n'est cassé. Fais ce que dit l'astuce : récupère son travail avec `git pull`, puis pousse à nouveau. Surtout pas `git push --force`, qui ferait disparaître son commit du distant. La doc de git push décrit ce cas exact dans « NOTE ABOUT FAST-FORWARDS » : un `git pull` crée un commit de fusion, puis le push passe.



Notre git push refusé du 8 novembre, avec ses cinq lignes d'astuce.

Si git pull parle de branches divergentes

Tu as un commit que le distant n'a pas, et l'inverse. Sans réglage, Git 2.49.0 ne choisit pas pour toi. Notre contre-épreuve du 8 novembre, dans la même situation, avec un `git pull` sans option

:

```
À COLLER

$ git pull
Depuis ../depot-distant
   98efea8..32bef7c  main       -> origin/main
astuce : Vous avez des branches divergentes et vous devez spécifier comment
astuce : les réconcilier. Vous pouvez le faire en lançant une des
astuce : commandes suivantes avant votre prochain tirage :
astuce :
astuce :   git config pull.rebase false  # fusion
astuce :   git config pull.rebase true   # rebasage
astuce :   git config pull.ff only       # avance rapide seulement
astuce :
astuce : Vous pouvez remplacer "git config" par "git config --global" pour que
astuce : ce soit l'option par défaut pour tous les dépôts. Vous pouvez aussi
astuce : passer --rebase, --no-rebase ou --ff-only sur la ligne de commande pour
astuce : remplacer à l'invocation la valeur par défaut configurée.
fatal : Besoin de spécifier comment réconcilier des branches divergentes.
```

Les empreintes diffèrent de celles du test principal : c'est un second dépôt, monté avec les mêmes fichiers. La doc de git pull relue le 8 novembre dit la même chose pour les versions récentes : sans réglage, il ne fait qu'une avance rapide et échoue si les branches ont divergé.

Notre choix, si tu débutes : la fusion, celle du test. Elle garde les deux commits tels quels et en ajoute un qui les réunit.

```
À COLLER

git pull --no-rebase
```

Pour ne plus avoir la question dans ce projet, `git config pull.rebase false` (ajoute `--global` pour tous tes projets). Sans `--no-edit`, Git ouvre ton éditeur sur le message de fusion : enregistre et ferme pour valider.

« Veuillez valider ou remiser vos modifications avant la fusion »

`README.md` est modifié chez toi, et le collègue l'a modifié aussi. Git refuse de l'écraser :

```
À COLLER

$ git pull --no-rebase --no-edit
Depuis ../depot-distant
   c0819c0..f61cc3c  main       -> origin/main
erreur : Vos modifications locales aux fichiers suivants seraient écrasées par
la fusion :
        README.md
Veuillez valider ou remiser vos modifications avant la fusion.
Abandon
La fusion avec la stratégie ort a échoué.
```

Avec un `git pull` nu, tu as exactement ce message quand tu n'as pas de commit local en plus. Contre-épreuve du même jour : « Veuillez valider ou remiser vos modifications avant la fusion. », puis « Abandon ».



Notre git pull refusé : README.md modifié des deux côtés, Git demande de valider ou de remiser.

Deux sorties. Si ton travail est prêt, valide-le : `git add README.md`, `git commit`, puis `git pull`. S'il ne l'est pas, remise-le, tire, et ressors-le. C'est ce qu'on a fait :

À COLLER

```
$ git stash
Arbre de travail et état de l'index sauvegardés dans WIP on main: 66b1836 Page
contact

$ git status
Sur la branche main
Votre branche et 'origin/main' ont divergé,
et ont 1 et 1 commits différents chacune respectivement.
  (utilisez "git pull" pour intégrer la branche distante avec la vôtre)

rien à valider, la copie de travail est propre

$ git pull --no-rebase --no-edit
Merge made by the 'ort' strategy.
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

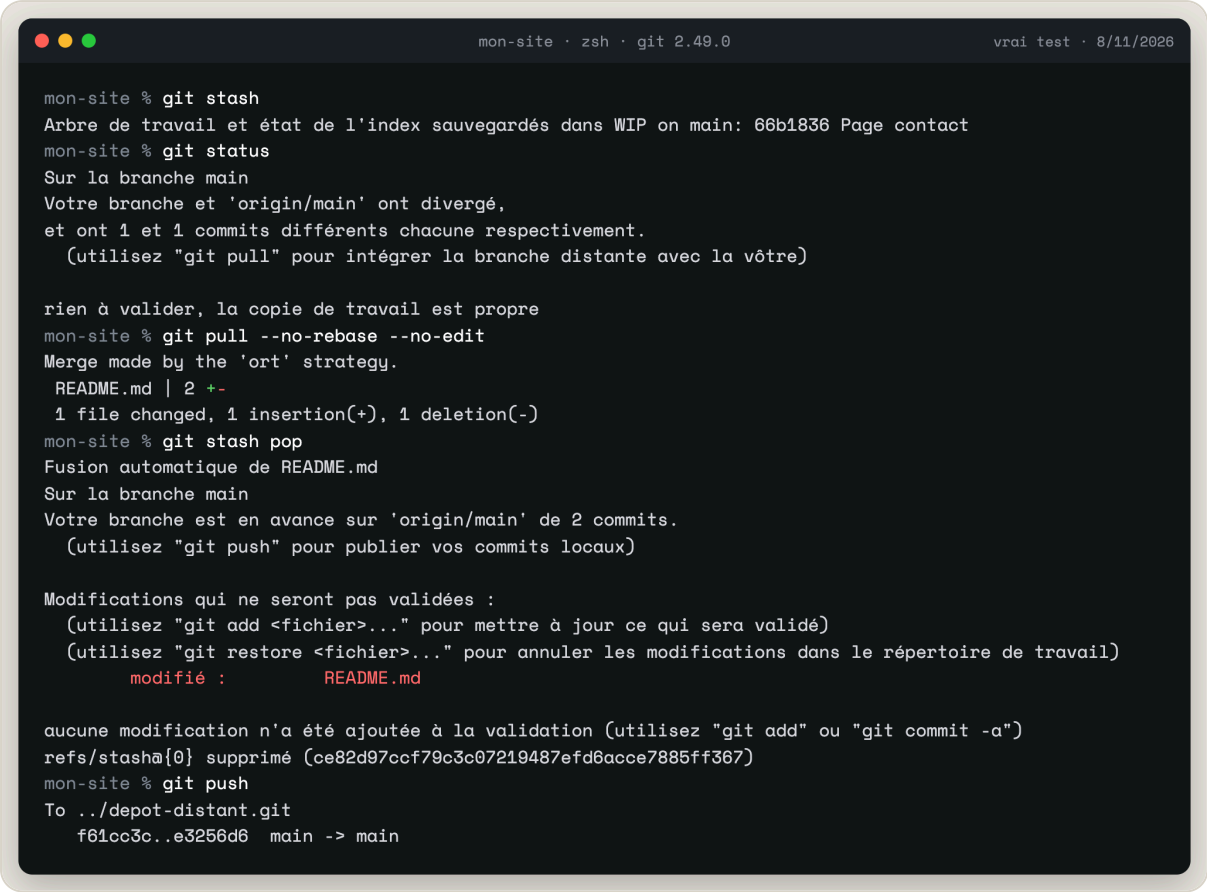
$ git stash pop
Fusion automatique de README.md
Sur la branche main
Votre branche est en avance sur 'origin/main' de 2 commits.
  (utilisez "git push" pour publier vos commits locaux)

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le
répertoire de travail)
        modifié :         README.md

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")

$ git push
To ../depot-distant.git
   f61cc3c..e3256d6  main -> main
```

La sortie de `git stash pop` se termine par une ligne de plus, qui annonce la suppression de l'entrée 0 du stash avec son empreinte ce82d97 (visible sur la capture). Les deux modifications du README se sont combinées sans conflit, parce qu'elles touchaient des lignes différentes. Quand elles se chevauchent, `git stash pop` signale un conflit et garde l'entrée : corrige le fichier, puis `git stash drop` (doc de git stash).



La suite réelle : git stash, git status propre, git pull, git stash pop, et le push qui passe.

La même recette est écrite dans la doc officielle de git stash, exemple « Pulling into a dirty tree » :

Pulling into a dirty tree

When you are in the middle of something, you learn that there are upstream changes that are possibly relevant to what you are doing. When your local changes do not conflict with the changes in the upstream, a simple `git pull` will let you move forward.

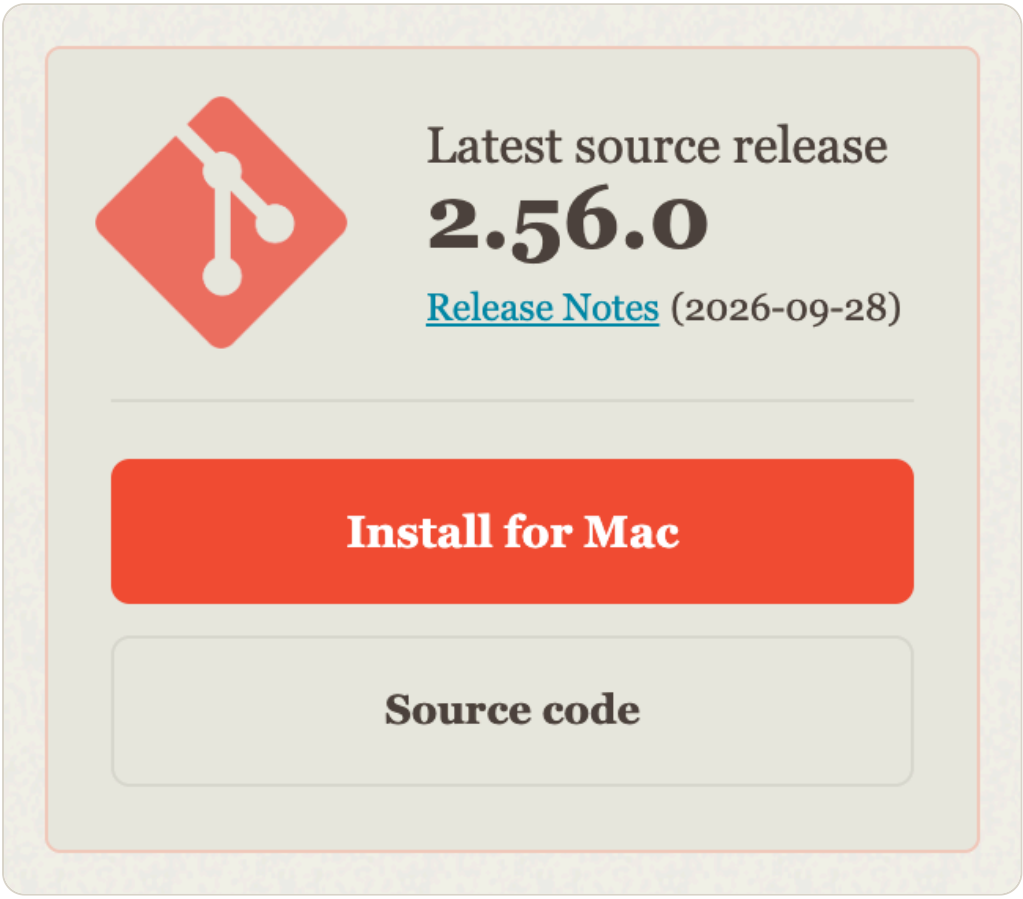
However, there are cases in which your local changes do conflict with the upstream changes, and `git pull` refuses to overwrite your changes. In such a case, you can stash your changes away, perform a pull, and then unstash, like this:

```
$ git pull
 ...
file foobar not up to date, cannot merge.
$ git stash
$ git pull
$ git stash pop
```

La doc de git stash relue le 8 novembre 2026 : git stash, git pull, git stash pop, pour tirer dans un répertoire de travail modifié.

7. Pour aller plus loin

- Pro Git en français, gratuit sur git-scm.com. Commence par « Rudiments de Git » (chapitre 1.3), puis « Annuler des actions » (2.4) et « Remisage et nettoyage » (7.3).
- La doc de chaque commande, en anglais : `git help status` l'ouvre depuis le Terminal, comme pour push, pull ou stash.
- La doc de GitHub sur les dépôts distants, et sa page sur les push refusés (« non-fast-forward errors »).
- La doc des permissions de Claude Code, en français : la syntaxe complète des règles allow, ask et deny, et ce qu'une règle Bash ne voit pas.
- Mets Git à jour. Sur Mac, `git --version` te dit la tienne. Le 8 novembre 2026, la dernière est la 2.56.0.



git-scm.com le 8 novembre 2026 : la dernière version publiée est la 2.56.0, sortie le 28 septembre 2026.

Sources

Pages relues le 8 novembre 2026.

- Git, doc de référence : git add, git commit, git push, git pull, git stash, git status, git diff, git reset, git restore, git checkout, git clean, git branch, git reflog, git config, glossaire, page d'accueil et version 2.56.0.
- Pro Git en français : Rudiments de Git, Annuler des actions, Travailler avec des dépôts distants, Branches de suivi à distance, Remisage et nettoyage, Maintenance et récupération de données.
- GitHub : À propos des dépôts distants, Dealing with non-fast-forward errors, About Git, github.com/git/git.

- Claude Code : Configurer les permissions, Paramètres, Référence des paramètres, Git et attribution.
- Google : Utiliser Google Drive pour ordinateur.