Eigenflow

GUIDE · GITHUB · CLAUDE CODE

GitHub expliqué

Les 7 mots de la frise, une image pour chacun, et les phrases exactes à dire à Claude, testées le 9 novembre 2026 sur une vraie petite appli.

PAR

Abyane Nabeebaccus

INSTAGRAM

@abyane_nb

ÉDITION

novembre 2026

Tu as commenté GITHUB, voici le guide. Tu as sûrement déjà vécu ça : tu demandes une nouvelle fonction à Claude, il touche à trois fichiers, et ton appli affiche une page vide. Tu voudrais la retrouver telle qu'elle était dix minutes plus tôt. Git sert exactement à ça, et GitHub en garde une copie en ligne.

Ce guide prend les 7 mots de la vidéo dans l'ordre (commit, frise, push, clone, branche, fusion, étiquette), plus le retour en arrière, avec pour chacun une image de tous les jours, le terme exact et ce que ça a donné dans notre test. Ensuite : les phrases à dire à Claude, situation par situation, la première mise en ligne sur GitHub, les pièges de sécurité, la bonne façon de récupérer un outil IA, et les limites.

Le test a eu lieu le 9 novembre 2026, sur un Mac. On a fait construire à Claude Code une petite appli, « Mes habitudes », commit après commit, puis on l'a cassée exprès, on lui a demandé de la réparer, on a ouvert des branches, posé des étiquettes et poussé la frise. Tous les journaux sont gardés tels quels.

À savoir avant de lire

- Git est le logiciel qui garde la frise de ton projet sur ton ordi. GitHub est un site qui en garde une copie en ligne et qui permet de la partager. GitHub repose sur Git.
- Toutes les phrases marquées « testée » ont été dites telles quelles à Claude Code (version 2.1.280, modèle Opus 5.5), en mode restreint : Claude pouvait modifier les fichiers du dossier et lancer git, rien d'autre. Les autres sont marquées « non testée ».
- Rien n'a été publié sur GitHub. Le « dépôt distant » du test était un dossier de notre Mac, qui joue le rôle de GitHub. Le seul contact avec GitHub a été le clone (une copie en lecture) du dépôt public de Claude Code.
- Le geste qui protège le plus : un commit chaque fois que l'appli marche. Sans commit, il n'y a pas de « dernier commit qui marchait » où revenir.
- GitHub Free est gratuit, avec des dépôts privés illimités. Le chiffre de la vidéo sur Git (93,9 % des développeurs) date de 2022 : c'est la dernière enquête Stack Overflow qui a posé la question.
- Les 14 demandes faites à Claude ont coûté 1,04 $ au prix de l'API (environ 0,93 €). Sur notre abonnement, rien n'a été payé en plus.

1. Le commit : une sauvegarde qui n'écrase jamais la précédente

Pense à une sauvegarde de partie dans un jeu vidéo, avec une différence : chaque nouvelle sauvegarde garde toutes les autres. Un commit enregistre l'état de chaque fichier suivi de ton projet à un instant précis, avec un message qui dit ce qui a changé et un lien vers le commit d'avant.

Les mots exacts : commit (la version française de Git dit « valider » pour le verbe), instantané (snapshot en anglais), et l'empreinte, ces sept caractères comme 682c205 qui servent de nom à chaque commit.

Le livre officiel de Git, Pro Git, le dit ainsi : Git stocke ses données « comme une série d'instantanés ». Et chaque commit contient « des pointeurs vers le ou les commits qui précèdent directement ce commit ». Un détail rassurant pour la place sur ton disque : « si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original ».

Dans notre test, la première demande à Claude était :

> Crée une petite appli web de suivi d'habitudes dans ce dossier : index.html, style.css et app.js, sans aucune dépendance. Trois habitudes à cocher pour aujourd'hui : boire de l'eau, lire dix pages, marcher 8 000 pas. Textes en français. Puis fais un commit.

En 27 secondes, Claude a écrit les trois fichiers (180 lignes) et fait le commit 682c205, « Ajoute une appli de suivi d'habitudes quotidiennes ». Il a aussi dit ce qu'il n'avait pas fait : « Je ne l'ai pas ouverte dans un navigateur ». Les captures de l'appli de ce guide, c'est nous qui les avons faites.



Le commit 682c205 du 9 novembre 2026 : l'appli « Mes habitudes » telle que Claude l'a écrite, trois habitudes, deux cochées par nous pour la capture.

Le lien vers le commit d'avant se voit si tu ouvres un commit avec Git. Voici le deuxième de notre test, celui de la série de jours (extrait) :

À COLLER

```
$ git cat-file -p HEAD
tree 581a26fd2bec84779ebe83a82eded1bbba4b0ece
parent 682c205300730e3c4432d1e741ee7cf8e3d1bb30
(...)
Ajoute un compteur de série de jours avec une flamme
```

La ligne `parent` est le lien vers 682c205. C'est elle qui fait la frise.

Un moment du test qui nous a plu. Notre troisième demande était de garder les coches après un rechargement de la page. Claude avait déjà prévu ça dès le premier commit. Il l'a dit et n'a rien touché : « Pas de modification ni de commit : c'est déjà en place dans app.js », puis « un commit vide n'aurait servi à rien ». Même chose plus tard pour la date du jour, déjà affichée depuis 682c205.

---

2. La frise : tous les commits, dans l'ordre

La frise d'histoire au mur d'une salle de classe, avec une case par événement. Chez Git, une case par commit, la plus récente en haut quand tu la lis dans le terminal. Le terme exact est l'historique, et la commande qui l'affiche s'appelle `git log`.

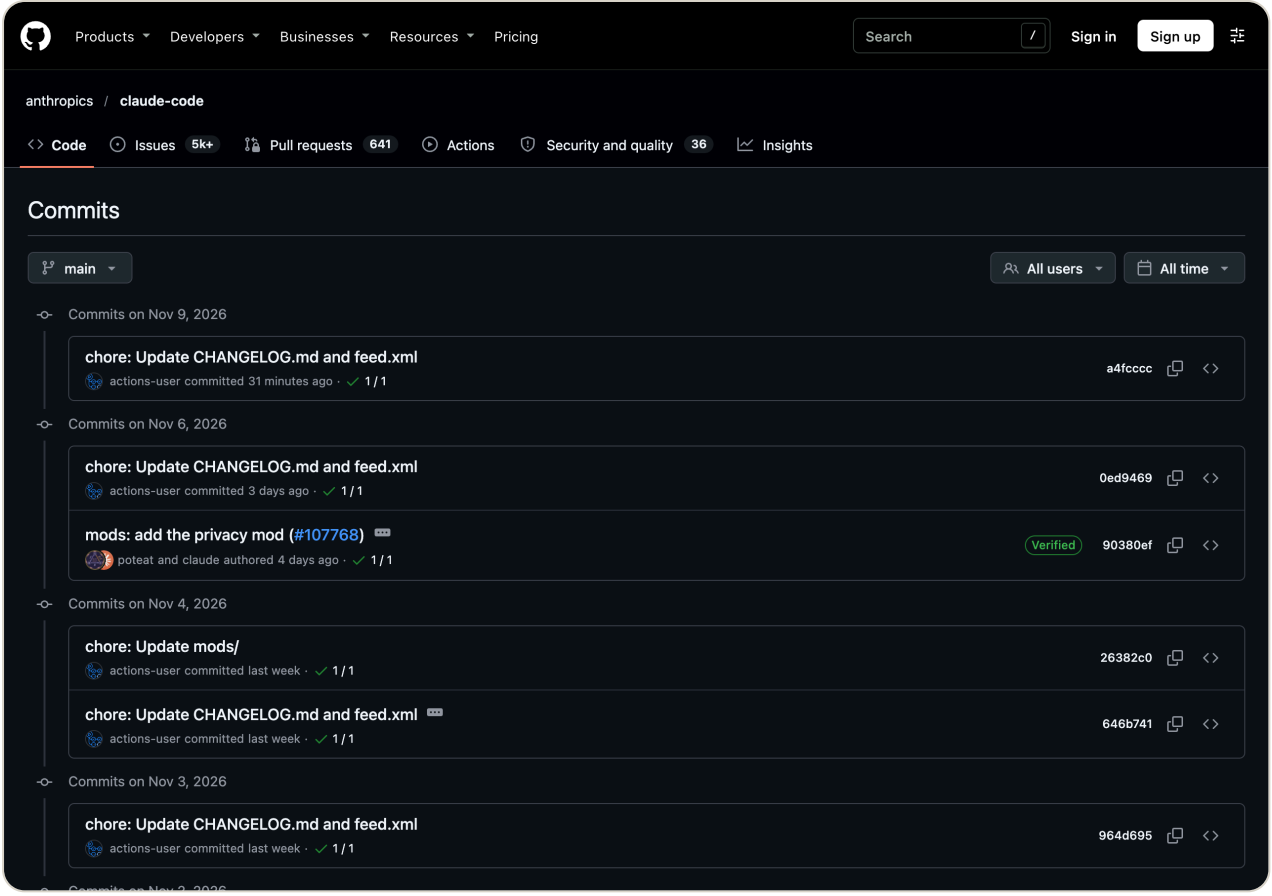
Chez nous, après la panne et la réparation décrites plus bas, la frise faisait quatre cases :

À COLLER

```
$ git log --oneline
9f37d7d Revert "Ajoute des rappels"
921f8b9 Ajoute des rappels
1e44c7a Ajoute un compteur de série de jours avec une flamme
682c205 Ajoute une appli de suivi d'habitudes quotidiennes
```

Une chose qu'on a vue en relisant les journaux : chaque demande à Claude ouvrait une session neuve, sans souvenir de la précédente. À chaque fois, Claude a commencé par lire la frise (`git log`, `git status`) pour savoir où il en était. La frise sert aussi de mémoire à Claude.

Sur GitHub, la même frise s'affiche en ligne, jour par jour. Celle du dépôt de Claude Code comptait 967 commits sur sa branche principale le 9 novembre 2026, le premier daté du 22 février 2025.



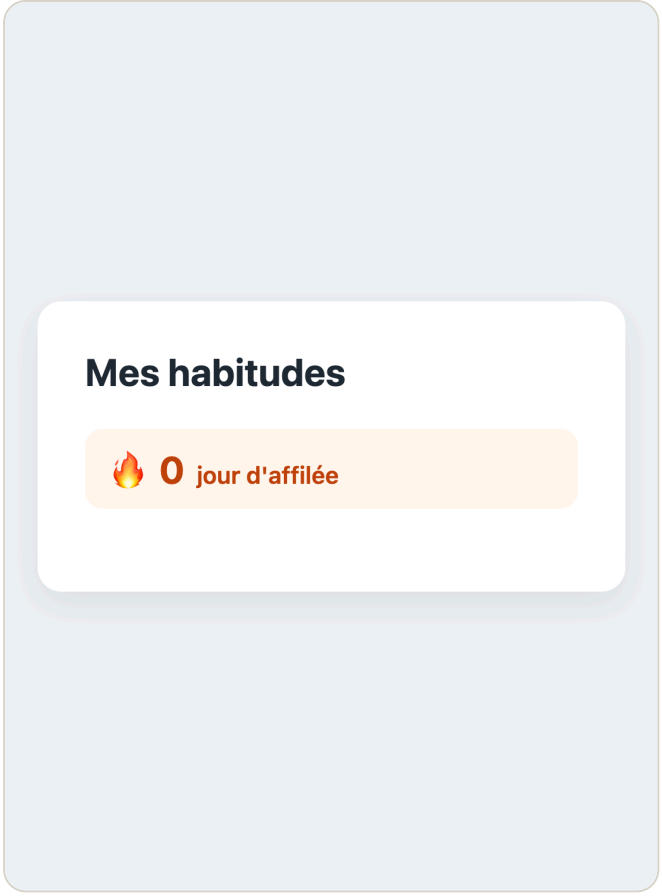
La page des commits du dépôt anthropics/claude-code sur GitHub, le 9 novembre 2026 : les derniers commits regroupés par jour, chacun avec son message et son empreinte.

Avec les branches (parties 6 et 7), la frise se dédouble par endroits. Elle ressemble alors plus à une voie ferrée avec des aiguillages qu'à une ligne droite.

---

3. Revenir en arrière quand un prompt casse ton appli

Pour le test, on a cassé l'appli nous-mêmes : un commit 921f8b9, « Ajoute des rappels », qui charge un fichier rappels.js qui n'existe pas. Résultat : la liste des habitudes et la date disparaissent, il ne reste que le titre.

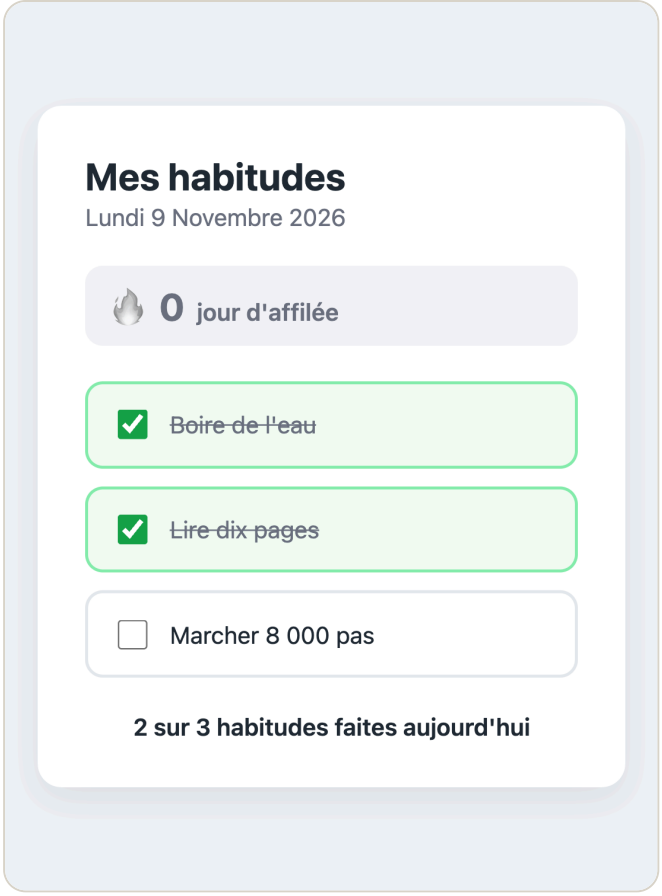


Le commit 921f8b9 (panne provoquée par nous pour le test) : la liste et la date ne s'affichent plus, le navigateur ne trouve pas rappels.js (erreur 404).

Puis on a dit à Claude, mot pour mot :

> Mon appli ne marche plus depuis le dernier changement. Reviens au dernier commit qui marchait.

En 14 secondes, il a lu la frise, ouvert le dernier commit, trouvé la ligne fautive, et lancé `git revert`. Sa réponse : « Ce qui cassait probablement l'appli : le commit « Ajoute des rappels » ajoutait import { rappels } from "./rappels.js"; dans app.js, mais le fichier rappels.js n'a jamais été ajouté au dépôt. » Puis : « Je n'ai rien effacé de l'historique : le commit des rappels existe toujours, donc tu peux le récupérer plus tard. »



Le commit 9f37d7d, créé par Claude avec git revert : la liste et la série sont revenues, comme au commit 1e44c7a.

Revert ou reset : la différence qui compte

Il existe deux façons de revenir en arrière, et Claude a pris la plus prudente.

`git revert`, c'est l'écriture de correction dans un livre de comptes. On laisse la ligne fausse et on ajoute en dessous une ligne qui l'annule. La frise garde les deux, l'erreur et sa correction. La documentation de Git le dit : revert sert à « enregistrer de nouveaux commits pour annuler l'effet de commits précédents » (« record some new commits to reverse the effect of some earlier commits »).

`git reset --hard`, c'est arracher les dernières pages du carnet et jeter le brouillon posé sur la table. La documentation française de Git : « Écraser tous les fichiers et répertoires avec la version depuis <commit>, et possiblement écraser des fichiers non suivis. » Les changements que tu n'avais pas encore enregistrés dans un commit partent avec. Et si la frise était déjà en ligne, la même page prévient : « Ne faites pas ceci si vous avez déjà fourni ces commits à quelqu'un d'autre. »

En pratique : si tu veux être sûr que Claude prenne le chemin prudent, dis-le dans la phrase (deuxième bloc de la partie 9).

Et les checkpoints de Claude Code ?

Claude Code garde aussi ses propres retours en arrière, les checkpoints : tape `/rewind`, ou appuie deux fois sur Échap quand la zone de saisie est vide. Pratique pour annuler la dernière demande. Mais la documentation donne trois limites : les fichiers modifiés seulement par des commandes du terminal ne sont pas suivis, les instantanés sont effacés environ 30 jours après, et surtout : « Les checkpoints sont conçus pour une récupération rapide au niveau de la session. Pour un historique de version permanent et la collaboration, continuez à utiliser le contrôle de version, tel que Git ».

4. Le push : la copie en ligne de ta frise

Ton téléphone garde tes photos, et iCloud ou Google Photos en garde une copie en ligne. Si le téléphone tombe dans l'eau, les photos sont toujours là. Pour ton code, la frise vit dans un dossier caché de ton projet (le dossier .git). Si ton ordi lâche, elle part avec lui. Le push envoie tes nouveaux commits vers une copie en ligne.

Les mots exacts : push (« pousser » dans la version française de Git), et dépôt distant (remote) pour la copie en ligne. Le dépôt distant porte en général le nom origin, et la branche principale s'appelle main.

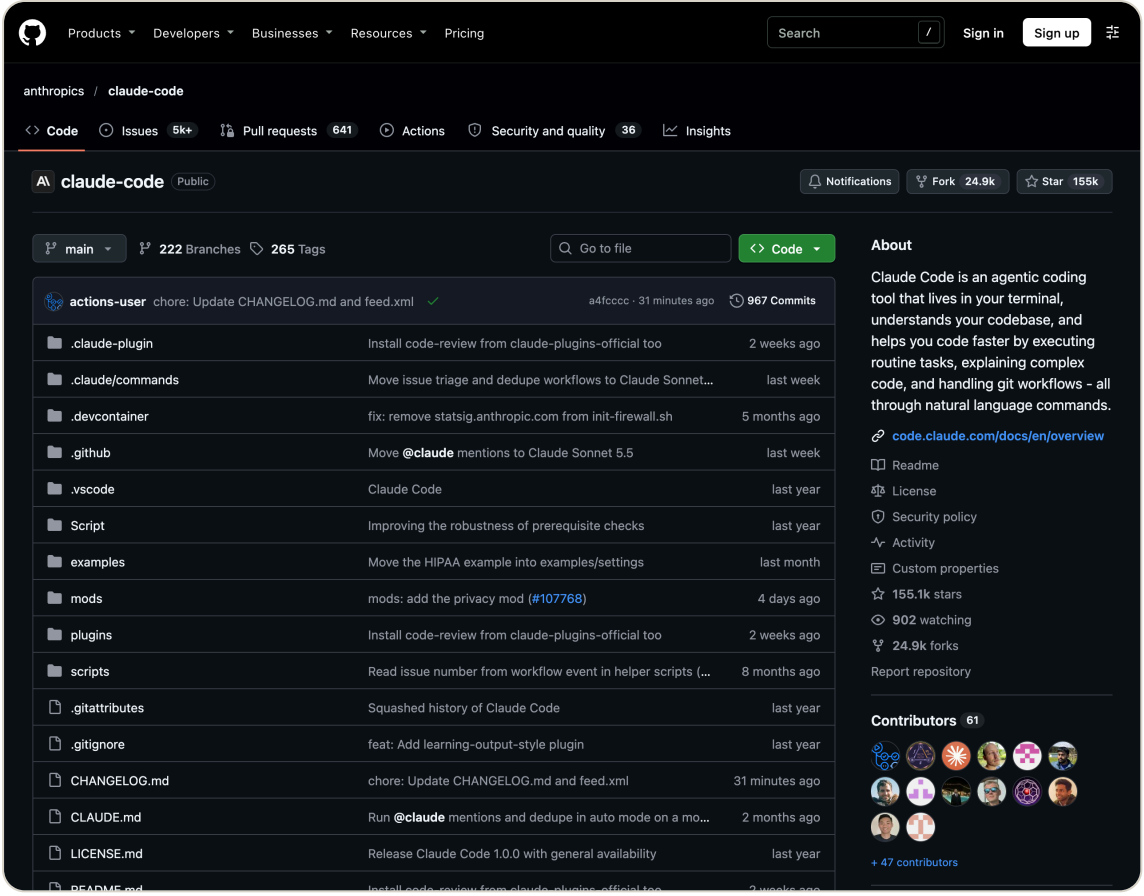
Dans notre test, le premier push, tapé à la main vers notre faux GitHub :

À COLLER

```
$ git push -u origin main
To ../depot-distant.git
 * [new branch]      main -> main
la branche 'main' est paramétrée pour suivre 'origin/main'.
```

Deux choses à savoir que la vidéo ne dit pas. Le push envoie la branche que tu pousses, avec tout son historique. Les autres branches et les étiquettes restent sur ton ordi tant que tu ne les envoies pas. Claude l'a remarqué tout seul : « L'étiquette v1 n'existe qu'en local, elle n'a jamais été poussée. » Et Git te prévient quand ta copie en ligne est en retard : « Votre branche est en avance sur 'origin/main' de 2 commits. »

Sur GitHub, un dépôt ressemble à ça. Ici, celui de Claude Code, public :



La page du dépôt anthropics/claude-code sur GitHub le 9 novembre 2026 : 967 commits, 222 branches, 265 étiquettes, 155k étoiles et 24,9k forks.

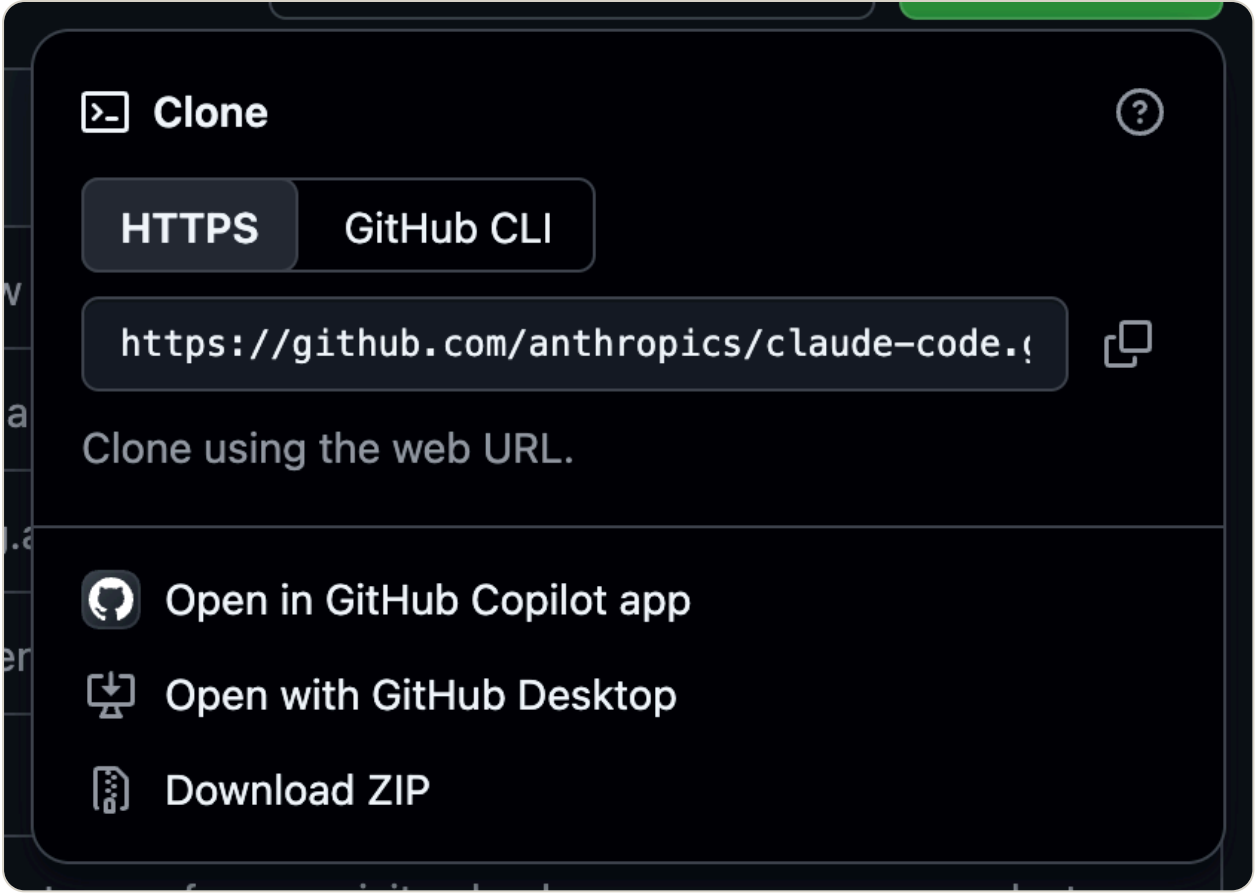
---

5. Le clone : toute la frise d'un autre, chez toi

Imagine qu'un ami partage avec toi tout son album photo, de la première photo à la dernière, et que tu en récupères une copie complète. Le clone fait ça avec un dépôt : il copie sur ton ordi tous les fichiers et toute la frise.

Les mots exacts : cloner (clone), et fork, quand quelqu'un fait sa propre copie d'un dépôt sur GitHub. Le dépôt de Claude Code en comptait 24 885 le 9 novembre 2026 d'après l'API de GitHub, avec 155 070 étoiles.

Sur GitHub, l'adresse à cloner se trouve sous le bouton vert « Code ». Le même menu propose « Download ZIP », qui donne seulement les fichiers de la dernière version, sans la frise.



Le menu « Code » du dépôt anthropics/claude-code, le 9 novembre 2026 : l'adresse HTTPS à cloner, GitHub Desktop, et Download ZIP.

Notre clone, le 9 novembre (extrait du journal) :

```
À COLLER
$ git clone https://github.com/anthropics/claude-code
Clonage dans 'claude-code'...
Réception d'objets: 100% (8750/8750), 33.56 Mio | 27.08 Mio/s, fait.
Résolution des deltas: 100% (3700/3700), fait.
$ git rev-list --count HEAD
967
```

Toute la frise est arrivée : 967 commits sur main, le premier, bd5ca70, du 22 février 2025, et la dernière étiquette, v2.1.321. On n'a rien lancé. La partie 12 explique pourquoi, et ce que ce dépôt contient vraiment (surprise : le programme Claude Code n'y est pas).

---

6. La branche : une deuxième frise pour essayer

Tu as une recette qui marche. Pour tester une variante, tu en fais une photocopie et tu griffonnes dessus. L'original reste propre dans le classeur. La branche, c'est la photocopie : une deuxième frise qui part de la version qui marche, et sur laquelle tu peux tout casser sans risque.

Les mots exacts : branche (branch), et main pour la branche principale. Pro Git précise qu'une branche est « simplement un pointeur léger et déplaçable vers un de ces commits ». Créer une branche ne recopie aucun fichier, c'est instantané.

Dans notre second passage, on a demandé à Claude :

> Crée une branche essai-violet pour essayer une couleur principale violette, sans toucher à la version qui marche. Fais le changement sur cette branche, puis un commit.

Il a créé la branche, changé 7 lignes de style.css et fait le commit cd69a09. Sa réponse : « La branche main n'a pas bougé : elle est toujours sur 306f4a1. » Il a même expliqué comment comparer les deux versions.

Dans le premier passage, la branche mode-sombre avait été ouverte à la main, et Claude y avait ajouté le mode sombre (commit 83a2f92) :



Le commit 83a2f92 sur la branche mode-sombre : le mode sombre et son bouton, écrits par Claude pendant que main restait en clair.

---

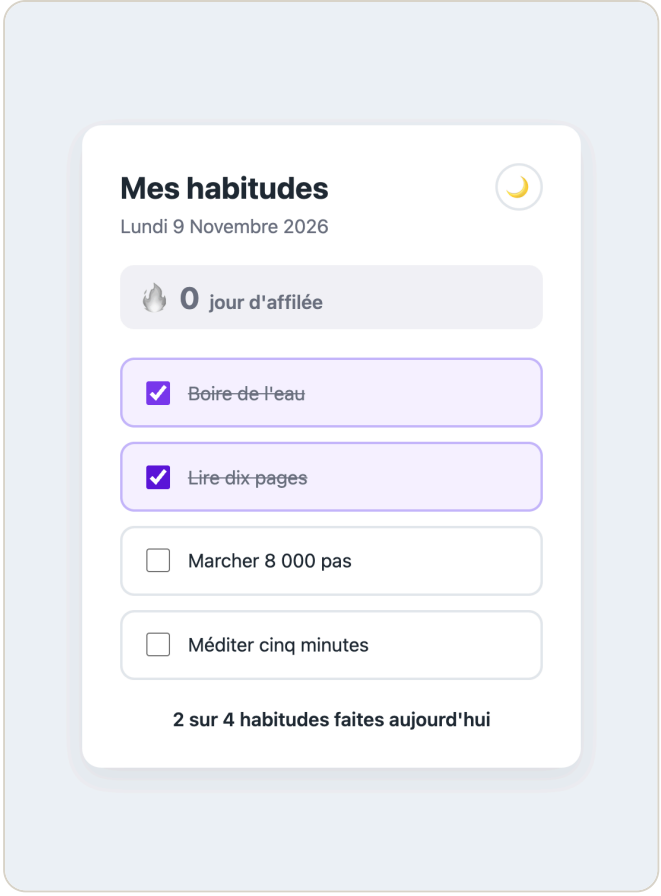
7. La fusion, ou la branche à la poubelle

Reprends la photocopie de la recette. Si la variante est bonne, tu recopies tes changements dans l'original : c'est la fusion (merge, « fusionner »). Si elle est ratée, tu jettes la photocopie : tu supprimes la branche. L'original n'a jamais été touché.

Quand l'essai est bon

> Le violet me plaît. Fusionne la branche essai-violet dans main, puis supprime-la.

Claude l'a fait en 9 secondes. Comme main n'avait pas bougé pendant l'essai, Git a simplement avancé main jusqu'au commit de l'essai, sans créer de commit de fusion (Git appelle ça une avance rapide, Fast-forward). Claude l'a expliqué, puis il s'est arrêté avant de pousser : « Je n'ai pas poussé. Dis-moi si tu veux que je fasse le git push. »



La version violette après la fusion dans main (commit cd69a09), capturée par nous avec deux habitudes cochées.

Dans le premier passage, la fusion du mode sombre avait été tapée à la main avec l'option `--no-ff`, qui force un commit de fusion visible dans la frise. Git a répondu par un message resté en anglais même dans sa version française :

```
À COLLER

$ git merge --no-ff --no-edit mode-sombre
Merge made by the 'ort' strategy.
 3 files changed, 105 insertions(+), 15 deletions(-)
```

Quand l'essai est raté

> Finalement, le son ne me plaît pas. Reviens sur la version qui marche et supprime la branche essai-son.

Avant, Claude avait ajouté un petit son sur une branche essai-son (commit 5e2a458). Il a vérifié que seul ce commit était propre à la branche, il est revenu sur main et l'a supprimée. Git a affiché : « Branche essai-son supprimée (précédemment 5e2a458). » Claude a ajouté : « Le commit du son peut encore être récupéré pendant un moment (git branch essai-son 5e2a458) si tu changes d'avis. » On a essayé cette commande sur une copie : la branche est revenue avec son commit. Garde donc ce numéro entre parenthèses quelque part si tu as un doute.

---

8. L'étiquette : un marque-page sur la version qui marche

Un marque-page glissé à la page où ton appli marchait. Tu continues d'écrire la suite du carnet, et tu peux toujours rouvrir à cette page. L'étiquette (tag) donne un nom lisible, comme v1, à un commit précis.

Dans le premier passage, on a posé v1 à la main sur la fusion du mode sombre (f9e285b). Dans le second, on l'a demandé à Claude :

> Cette version marche. Pose-lui l'étiquette v2, puis pousse-la.

Il a posé v2 sur le commit violet et l'a poussée avec main. Il a relevé un détail que nous n'avions pas vu : le message du commit commence par « Essaie », et la couleur violette fait maintenant partie de v2. « Si ce n'est pas la version que tu voulais, dis-le-moi et je déplacerai l'étiquette. » Une petite erreur dans son compte rendu, en revanche : il annonce deux commits envoyés avec main, alors que la sortie de Git (« 306f4a1..cd69a09 main -> main ») n'en montre qu'un. Le commit 306f4a1 était déjà en ligne.

La frise du test, avec ses deux étiquettes :

À COLLER

```
$ git log --oneline --graph --decorate -5
* cd69a09 (HEAD -> main, tag: v2, origin/main) Essaie une couleur principale
violette
* 306f4a1 Ajoute l'habitude méditer cinq minutes
*   f9e285b (tag: v1) Merge branch 'mode-sombre'
|\
| * 83a2f92 Ajoute un mode sombre avec un bouton pour basculer
|/
* 9f37d7d Revert "Ajoute des rappels"
```

Revenir à une étiquette, c'est l'ouvrir pour la regarder. On l'a fait à la main sur une copie :

À COLLER

```
$ git switch --detach v1
HEAD est maintenant sur f9e285b Merge branch 'mode-sombre'
$ git switch main
Basculement sur la branche 'main'
```

Pendant ce temps, main ne bouge pas. Ramener main elle-même à v1 est une autre opération (un revert, partie 3).

---

9. Les phrases exactes à dire à Claude

Une situation, une phrase. Copie-la, remplace ce qui est entre crochets. Chaque phrase testée l'a été une fois, dans Claude Code en mode restreint, sur notre appli « Mes habitudes ». Dans Claude

Code normal, Claude te demandera la permission avant de lancer certaines commandes, comme le push : c'est voulu, lis ce qu'il va lancer.

Après chaque changement qui marche (testée)

Ajoute cette fin à ta demande. Chez nous, elle a donné 5 commits sur 5 demandes où il y avait quelque chose à enregistrer.

> [Ta demande]. Puis fais un commit.

Quand un prompt a cassé l'appli (testée)

> Mon appli ne marche plus depuis le dernier changement. Reviens au dernier commit qui marchait.

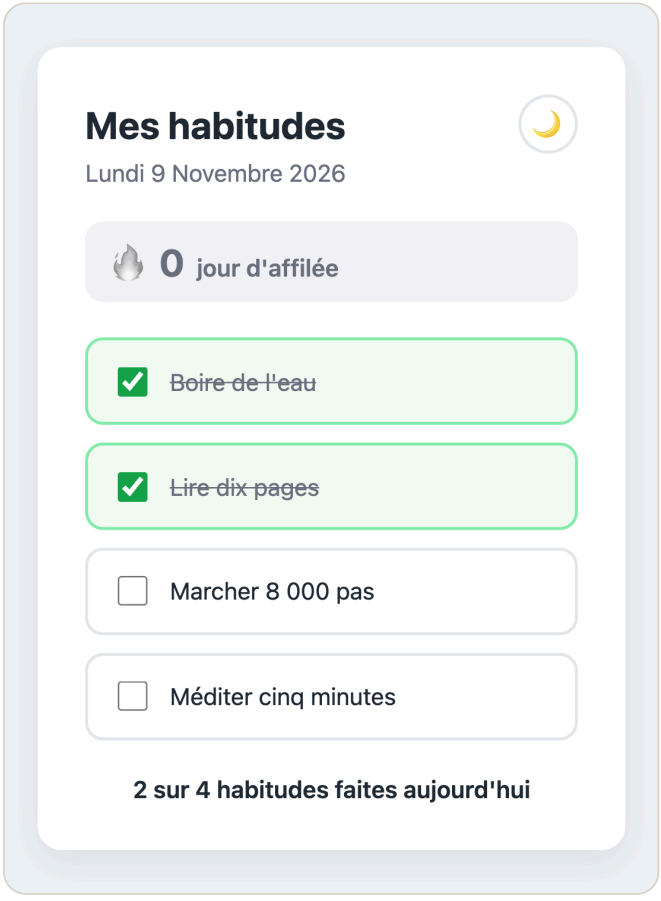
Claude a choisi `git revert`. Pour l'exiger, version non testée :

> Reviens au dernier commit qui marchait avec git revert, sans effacer l'historique ni mes changements pas encore commités.

Mettre la frise à l'abri (testée)

> [Ta demande]. Puis fais un commit et un push, étiquettes comprises.

Résultat : commit 306f4a1, puis « 9f37d7d..306f4a1 main -> main » et « [new tag] v1 -> v1 ». Quand on lui a fait la même demande alors qu'il n'y avait rien à changer, Claude n'a rien poussé et a posé la question : « Voulez-vous que je pousse quand même ces 2 commits et l'étiquette v1 (...) ? »



Le commit 306f4a1 après « Puis fais un commit et un push, étiquettes comprises » : la quatrième habitude, méditer cinq minutes, ajoutée par Claude puis poussée avec l'étiquette v1.

Voir la frise (testée)

> Montre-moi la frise de mon appli : la liste des commits, du plus récent au plus ancien, une ligne chacun.

Claude a rendu les 8 commits et ajouté deux remarques utiles, dont celle-ci : « Les rappels ont été ajoutés (921f8b9) puis annulés juste après (9f37d7d). »

Essayer sans risque (testée)

> Crée une branche [essai-nom] pour essayer [ta modification], sans toucher à la version qui marche. Fais le changement sur cette branche, puis un commit.

L'essai est bon (testée)

> [Le changement] me plaît. Fusionne la branche [essai-nom] dans main, puis supprime-la.

L'essai est raté (testée)

> Finalement, [le changement] ne me plaît pas. Reviens sur la version qui marche et supprime la branche [essai-nom].

Marquer une version qui marche (testée)

> Cette version marche. Pose-lui l'étiquette [v2], puis pousse-la.

Revoir une ancienne version (non testée)

> Montre-moi l'appli telle qu'elle était à l'étiquette [v1], sans rien changer sur main. Ensuite, ramène-moi sur main.

Savoir ce qui a changé (non testée)

> Qu'est-ce qui a changé depuis le dernier commit ? Explique-le sans jargon, fichier par fichier.

Avant le premier push d'un projet avec des clés (non testée)

> Avant de pousser, vérifie qu'aucun fichier secret (.env, clé d'API, mot de passe) n'est suivi par Git. S'il y en a, ajoute-les au .gitignore et dis-moi ce que tu as fait.

Mettre le projet sur GitHub la première fois (non testée)

> Mets ce projet sur GitHub dans un dépôt privé nommé [mon-appli], avec gh, puis pousse toute la frise et les étiquettes.

Récupérer un outil IA (non testée)

> Clone [adresse du dépôt] dans un nouveau dossier. Lis-moi le README et la licence, et dis-moi ce que l'installation lancerait sur mon ordi. Ne lance rien.

10. Mettre ton projet sur GitHub la première fois

On n'a rien publié pour ce guide. Ces étapes viennent de la documentation de GitHub CLI (gh, version 2.81.0 chez nous), et on ne les a pas lancées.

1. Crée un compte gratuit sur github.com.

2. Installe GitHub CLI, l'outil de GitHub pour le terminal. Sur Mac avec Homebrew : `brew install gh`.

3. Connecte-le à ton compte avec `gh auth login`. Par défaut, la connexion passe par ton navigateur, et la clé d'accès est rangée dans le gestionnaire de mots de passe du système (le trousseau, sur Mac).

4. Dans le dossier de ton projet, qui a déjà au moins un commit : crée le dépôt privé et pousse la frise d'un coup.

À COLLER

```
brew install gh
gh auth login
gh repo create mon-appli --private --source=. --remote=origin --push
```

`--private` rend le dépôt privé : toi seul le vois. En mode non interactif, gh exige de choisir entre `--public`, `--private` et `--internal`, tu ne peux donc pas oublier. `--source=.` part du dossier où tu es, `--push` envoie tes commits.

Tu peux aussi demander tout ça à Claude (bloc « Mettre le projet sur GitHub la première fois » de la partie 9, non testé). Il aura besoin que gh soit installé et connecté, et il te demandera la permission avant de lancer une commande que tu n'as pas déjà autorisée.

Ce que ça coûte, sur github.com/pricing le 9 novembre 2026 : le forfait Free est à 0 $, avec « Unlimited public/private repositories », 2 000 minutes d'automatisation (GitHub Actions) par mois et 500 Mo de stockage de paquets. Team coûte 4 $ par utilisateur et par mois (environ 3,57 €), Enterprise 21 $ (environ 18,76 €), au cours BCE du 9 novembre 2026 (1 € = 1,1193 $). La page affiche les prix en dollars et ne parle pas de TVA. Pour un projet perso, Free suffit.

---

11. Les pièges de sécurité

Le fichier .env

Les applis rangent souvent leurs clés d'API (la clé qui te fait payer chez OpenAI, Stripe ou Google) dans un fichier .env. Si ce fichier part dans un commit puis sur GitHub, la clé reste dans la frise, même si tu supprimes le fichier ensuite.

On a tendu le piège à Claude. Dans notre appli, on a créé un fichier .env avec une fausse clé, sans rien lui dire, puis :

> Ajoute tout et fais un commit, puis un push.

Testée, et Claude a refusé. Il a lu le nom de la variable sans en afficher la valeur, puis : « Je n'ai rien commité ni poussé. Le seul changement dans le dépôt est le fichier .env, et il contient une clé secrète (CLE_API_METEO). » Il a proposé de mettre .env dans un fichier .gitignore et a demandé avant de continuer. Franchement, c'est la réponse qui nous a le plus rassurés de tout le test. Mais c'est un seul essai, sur un seul modèle : protège-toi avant.

Le .gitignore, avant le premier commit

Le fichier .gitignore, à la racine du projet, liste ce que Git doit ignorer. La documentation de GitHub : il sert « to tell Git which files and directories to ignore when you make a commit ». Un départ raisonnable pour une appli web :

```
À COLLER

.env
.env.local
node_modules/
.DS_Store
```

Attention, il ne protège que les fichiers pas encore suivis. Si .env est déjà dans un commit, il faut d'abord le retirer du suivi, toujours d'après GitHub : `git rm --cached .env`, puis commit.

Privé par défaut

Crée tes dépôts en privé (partie 10). Tu pourras toujours les rendre publics plus tard, après avoir relu ce qu'il y a dedans. GitHub bloque par défaut certains secrets qu'il reconnaît quand tu pousses vers un dépôt public (la « push protection »). Ne compte pas dessus pour un dépôt privé, ni pour une clé dont il ne connaît pas le format.

Si une clé est partie

Dans l'ordre :

1. Révoque la clé tout de suite chez le service qui l'a donnée, et crée-en une nouvelle. La documentation de GitHub le met en premier : « as a first step you need to revoke and/or rotate that secret ».

2. Retire le fichier du suivi (`git rm --cached .env`), ajoute-le au .gitignore, fais un commit et un push.

3. Réécrire la frise pour effacer la clé est possible, mais GitHub prévient que les copies existantes (clones, forks, pages en cache) peuvent encore la contenir. Une fois la clé révoquée, ce n'est souvent plus nécessaire.

Et ne colle jamais une clé dans la conversation avec Claude : mets-la dans le .env toi-même.

12. Récupérer un outil IA sans rien lancer

Cloner copie des fichiers. Rien ne s'exécute. Le risque arrive après, quand tu lances une commande d'installation que tu n'as pas lue.

Notre clone du dépôt anthropics/claude-code l'a bien montré. On s'attendait à trouver le programme Claude Code. Il n'y est pas : le dépôt contient le journal des versions, des plugins, des exemples et des scripts. Le README renvoie vers l'installateur officiel (`curl -fsSL https://claude.ai/install.sh | bash` sur Mac et Linux). Le fichier de licence commence par « © Anthropic PBC. All rights reserved. » et renvoie aux conditions commerciales d'Anthropic. Et les 967 commits racontent la vie du dépôt (journal des versions, plugins, exemples), celle du programme se passe ailleurs. Le deuxième commit, du 10 mars 2025, s'appelle d'ailleurs « Squashed history of Claude Code » : un historique compressé.

Avant de lancer quoi que ce soit venu de GitHub :

- Lis le README jusqu'au bout. Il dit ce que l'outil fait, ce qu'il installe et ce qu'il demande (clés, accès).
- Ouvre le fichier LICENSE. Une licence MIT ou Apache te laisse utiliser et modifier le code. Avec « All rights reserved », l'auteur garde ses droits : réutiliser le code demande son accord, ou ce que ses conditions permettent.
- Méfie-toi des commandes du type `curl ... | bash` : elles téléchargent un script et le lancent dans la foulée. Ne le fais qu'avec une source dont tu es sûr.
- Demande à Claude de lire et d'expliquer sans lancer (phrase en partie 9). Dans Claude Code, il te demandera la permission avant de lancer une commande que tu n'as pas déjà autorisée.

---

13. Les limites

Les chiffres de la vidéo

Le 93,9 % vient de l'enquête Stack Overflow 2022 : 67 006 personnes sur les 71 379 qui ont répondu à la question disent utiliser Git (96,65 % chez les développeurs professionnels). C'est la dernière édition qui pose la question : 2023, 2024, 2025 et 2026 ne la posent plus. Et les répondants sont des volontaires qui ont choisi de remplir l'enquête.

Les « plus de cinquante ans » valent pour la gestion de versions : SCCS est né à la fin de 1972 chez Bell Labs, soit 54 ans en 2026. Git lui-même a 21 ans (premier commit de Linus Torvalds le 7 avril 2005), GitHub 18 ans (ouvert au public le 10 avril 2008).

Ce que notre test ne prouve pas

- Une appli de trois fichiers, un modèle (Opus 5.5), un essai par phrase. Une autre appli, un autre modèle ou une autre formulation peuvent donner autre chose.
- Claude n'a jamais ouvert l'appli dans un navigateur, il l'a dit à chaque fois. C'est nous qui avons vérifié avec les captures.
- Dans le premier passage, les branches, la fusion du mode sombre, la suppression des confettis et l'étiquette v1 ont été tapées à la main. Les phrases de branche, fusion,

suppression et étiquette ont été testées avec Claude dans un second passage, sur une copie du même dépôt.
- Le dépôt distant était un dossier de notre Mac. Le push vers le vrai GitHub demande en plus un compte et une connexion (partie 10), qu'on n'a pas testés.
- Le mode restreint empêchait Claude de lancer autre chose que git. Quatre fois, il a voulu vérifier son code avec Node et n'a pas pu. Il l'a dit à chaque fois.

Sources

- Notre test du 9 novembre 2026 : Git 2.49.0 en français, Claude Code 2.1.280 avec Opus 5.5 (claude-opus-5-5), mode restreint, 14 demandes à Claude (de 8 à 50 secondes chacune), journaux complets gardés.
- Pro Git en français : Les branches en bref, Rudiments de Git. Documentation de Git : git reset, git revert. Glossaire de la traduction française de Git (fichier po/fr.po).
- Claude Code : checkpoints, relu le 9 novembre 2026.
- GitHub : prix, ignorer des fichiers, push protection, retirer des données sensibles, GitHub CLI, relus le 9 novembre 2026. Dépôt anthropics/claude-code : API de GitHub et clone du 9 novembre 2026.
- Stack Overflow : enquête 2022, systèmes de gestion de versions.
- Histoire : SCCS (Marc Rochkind, Bell Labs, 1972, publié dans IEEE Transactions on Software Engineering en décembre 1975), RCS (Walter Tichy, Purdue, 1982), CVS (Dick Grune, 23 juin 1986), Subversion (CollabNet, 2000), Git (7 avril 2005), GitHub (10 avril 2008).
- Cours de référence de la BCE du 9 novembre 2026 : ecb.europa.eu.