Retour au Dossier Organisationnel	
Sujets connexes	Cartographie fonctionnelle de LoGeAs Proc#145 : Procédure de Gestion des Versions Informations sur la version en cours

Guide pratique — Cycle de release, étape par étape

Document complémentaire à Procédure de Gestion des Versions — celui-ci répond à la question « qu'est-ce que je fais, concrètement, à chaque étape, qui le fait, et où ? ». Destiné au développeur et au chef de projet, pas à un usage certification.

Où se lancent ces commandes

`/fusion-dev`, `/release-beta`, `/release-client` et `/rapport-tests-qualite` sont des **commandes Claude Code** (l'assistant IA installé sur le poste de développement), pas des commandes shell classiques. Elles se tapent directement dans l'interface Claude Code (CLI ou extension), pas dans un terminal Git Bash / PowerShell nu.

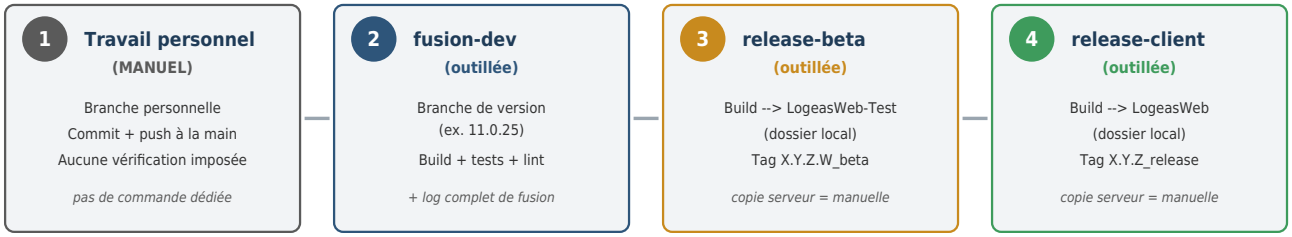
- **Poste concerné :** celui qui a un checkout local de `logeas-web` **et** accès au dossier `D:\Developpements\VersionPublic\ExecutablesWindows\` (pour les versions des composants Galaxie LoGeAs dans le rapport de release).
- **Dossier de travail :** toujours la racine du dépôt `logeas-web` (ou `logeas-cartographie` pour la dernière étape de `/release-client`).
- **Branche active au moment de lancer la commande :**
  - `/fusion-dev` : peu importe la branche de départ, la commande fait elle-même le `checkout` vers la branche de version.
  - `/release-beta` et `/release-client` : doivent être lancées **sur la branche de version** (ex. `11.0.25`), pas sur une branche personnelle.

Qui fait quoi

Étape	Qui décide de lancer	Qui exécute
1. Travail personnel	Développeur (son propre rythme)	Développeur
2. `/fusion-dev`	Développeur (son travail est prêt)	Développeur
3. `/release-beta`	Chef de projet ou développeur senior (juge la branche stable)	Développeur ou chef de projet — l'un ou l'autre peut techniquement lancer la commande
4. `/release-client`	**Chef de projet** (validation métier que la beta est bonne à livrer)	Développeur ou chef de projet
Copie manuelle vers le serveur (test ou client)	—	Personne ayant accès au bureau à distance du serveur concerné

En résumé : le développeur porte les étapes 1 et 2 seul. À partir de l'étape 3, c'est une décision d'équipe/de chef de projet de passer à l'étape suivante — l'exécution technique de la commande peut rester déléguée au développeur, mais le feu vert est côté chef de projet, en particulier pour l'étape 4 (on livre au client).

Vue d'ensemble



Aucune étape ne crée de nouvelle branche Git pour la beta ou la release — un simple tag marque le code correspondant.

Étape	Décision	Où	Commande
1. Travail personnel	Développeur	Poste dev, branche perso	(aucune — voir détail plus bas)
2. Intégration	Développeur	Claude Code, `logeas-web`	`/fusion-dev`
3. Beta	Chef de projet	Claude Code, `logeas-web`, branche de version	`/release-beta`
4. Release client	Chef de projet	Claude Code, `logeas-web` puis `logeas-cartographie`	`/release-client`

Autres commandes "utiles"

Contexte	Commande	Explication
logeas-Web.fr “D:\Developpements\Sources\NewInterfaces\logeas-web”	`ng build --configuration "production,beta"`	Génère une version de test de l'interface dans “D:\Developpements\VersionPublic\LogeasWeb-Test\browser” sans passer par les test, la publication, les tags... sauvage quoi !!
logeas-Web.fr “D:\Developpements\Sources\NewInterfaces\logeas-web”	`ng build --configuration production`	Génère une version de l'interface release dans “D:\Developpements\VersionPublic\LogeasWeb\browser” Ne bump pas la version (package.json reste à la version courante) Ne régénère pas version.ts (donc l'écran “info version” affichera des infos obsolètes/incohérentes) Ne tague pas la branche git (donc aucune traçabilité de ce qui a été déployé) Écrase directement ce que voient les vrais utilisateurs de production

—-

Étape 1 — Travail personnel (manuel)

Déclencheur : en continu, dès qu'on commence à développer quelque chose.

Qui : le développeur, seul.

Où : son poste, terminal ou client Git, dans `logeas-web`, sur sa branche personnelle.

Ce qu'on fait, à la main :

1. `git checkout -b <nom-de-branche>` depuis la branche de version en cours (ex. `11.0.25`) — pas de convention de nom imposée.
2. On développe, on teste localement si besoin.
3. `git add <fichiers>` — **jamais** `git add -A` à l'aveugle, on relit ce qu'on ajoute.
4. `git commit -m "…"` — message en français, court, orienté fonctionnel.
5. `git push -u origin <branche>` — sauvegarde régulière, au fil de l'eau.

Volontairement aucune vérification ici (pas de build, pas de tests) : c'est une sauvegarde de travail potentiellement inachevé, pas encore un contrôle qualité.

À vérifier soi-même avant de committer :

- Rien de sensible n'est ajouté (secrets, fichiers de config locale, `dist/`, `coverage/`).
- Si le changement touche en réalité un comportement partagé (`DGService`, composants génériques), le code doit vivre dans `logeas-lib`, pas ici.

Étape 2 — Intégration (''/fusion-dev'')

Déclencheur : le travail sur la branche personnelle est prêt à rejoindre le tronc commun.

Qui : le développeur qui a fait le travail (ou toute personne de l'équipe).

Où : Claude Code, dans `logeas-web`.

Ce que la commande fait, dans l'ordre :

1. `git fetch`
2. `git checkout <branche-de-version>` puis `git pull`
3. `git log <branche-de-version>..<branche-perso> –pretty=…` — récupère la liste complète des commits qui vont être intégrés
4. `git merge –no-ff <branche-perso>` — le message de fusion embarque cette liste de commits, pas juste un titre générique
5. `ng build` — vérifie que ça compile
6. `npm test` — tests unitaires (Jest)
7. `npm run lint` — seulement les **nouvelles** erreurs introduites par le diff comptent, pas la dette préexistante
8. Si tout est vert : `git push` de la branche de version

À vérifier après coup :

- Le commit de fusion contient bien la liste des commits intégrés (pas juste « Merge branch X into Y »).
- Aucun conflit n'a été résolu à l'aveugle sur un point métier — en cas de doute, la procédure doit s'arrêter et demander plutôt que trancher seule.

- Build/tests/lint sont bien passés **après** la fusion, pas seulement avant (l'intégration de plusieurs travaux peut casser des choses invisibles avant).

Si ça échoue : la procédure corrige sur place si c'est trivial, sinon elle s'arrête et explique — jamais de `push --force`, jamais de fusion poussée avec des tests rouges.

Étape 3 — Beta (''/release-beta'')

Déclencheur : la branche de version est jugée stable, prête pour un tour de validation interne.

Qui décide : chef de projet ou développeur senior. **Qui exécute :** développeur ou chef de projet.

Où : Claude Code, dans `logeas-web`, **sur la branche de version** (pas sur une branche perso).

Ce que la commande fait, dans l'ordre :

1. `git status` — vérifie que la branche est propre et à jour avec `origin`
2. `npm run build:beta`, qui enchaîne :
   - `node scripts/version.js beta` — calcule le numéro `X.Y.Z.W` (voir règle de numérotation dans la procédure de référence)
   - régénération de la licence DevExtreme
   - `ng build --configuration production,beta` — sortie vers `LogeasWeb-Test`
3. `node scripts/release-report.js beta`, qui :
   - calcule le diff Git depuis le **dernier tag `_release`** (jamais depuis le dernier beta — le rapport cumule tout le cycle)
   - relance `npx jest --coverage --json` pour un résumé de tests à jour (pass/fail, couverture)
   - lit la version des 4 composants de la Galaxie LoGeAs directement sur les exécutables compilés (`D:\Developpements\VersionPublic\ExecutablesWindows\`)
   - écrit `src/assets/releases/X.Y.Z.W.html` + met à jour `manifest.json`
4. Commit dédié « Preparation version X.Y.Z.W » (`package.json` + `version.ts` + le rapport généré)
5. `git tag -a X.Y.Z.W_beta`
6. `git push` + `git push origin <tag>`

À vérifier après coup :

- Le numéro de version généré est bien celui attendu (cohérent avec le nom de la branche).
- Le rapport de release ne signale pas d'échec de test bloquant — s'il y en a, décider **avec l'équipe** si on tague quand même.
- Le build est bien présent dans `D:\Developpements\VersionPublic\LogeasWeb-Test\browser\`.
- Le rapport est consultable dans l'appli via l'écran « Rapports de release » (Administration, ou directement `/RapportsRelease`, accessible sans connexion).

Ce qui reste manuel : copier `LogeasWeb-Test\browser\` vers le serveur de test — accessible uniquement en bureau à distance, aucune procédure ne le fait à votre place.

Étape 4 — Release client (''/release-client'')

Déclencheur : la beta a été validée (ou directement si aucune beta n'est jugée nécessaire pour ce cycle).

Qui décide : chef de projet — c'est la décision de livrer au client. **Qui exécute :** développeur ou chef de projet.

Où : Claude Code, dans `logeas-web` (étapes 1 à 6), puis dans `logeas-cartographie` pour la relecture/le commit du rapport qualité (étape 7).

Ce que la commande fait, dans l'ordre :

1. `git status` — branche propre et à jour
2. Si cette release fait suite à une beta : vérifie que rien d'autre que des correctifs ciblés n'a été ajouté depuis le tag beta (`git log <tag_beta>..HEAD`) — sinon ça mérite peut-être un nouveau tour de beta.
3. `npm run build`, qui enchaîne :
   - `node scripts/version.js release` — repasse en 3 chiffres, **reprend le même X.Y.Z que les betas du cycle** (jamais de numéro « orphelin »)
   - `ng build –configuration production` — sortie vers `LogeasWeb`
4. `node scripts/release-report.js release` — même mécanique qu'en beta
5. Commit dédié « Preparation version X.Y.Z » (`package.json` + `version.ts` + le rapport)
6. `git tag -a X.Y.Z_release`
7. `git push` + push du tag
8. **`/rapport-tests-qualite`** — uniquement à cette étape, jamais en beta : régénère le rapport qualité NF dans `logeas-cartographie` (dépôt frère, pas `logeas-web`), à partir d'une exécution réelle des tests de `logeas-web` **et** `logeas-lib`

À vérifier après coup :

- Le build est bien présent dans `D:\Developpements\VersionPublic\LogeasWeb\browser\`.
- Le tag ne rentre pas en collision avec un tag `_release` déjà existant.
- Le rapport qualité NF (`logeas-cartographie`) a bien été régénéré, relu, et **proposé au commit** — jamais commité automatiquement, dans aucun des deux dépôts.

Ce qui reste manuel : copie vers le serveur client (bureau à distance).

Cas particulier — Basculer vers une version mineure en cours de cycle

Quand : la branche de version en cours (ex. `11.0.25`) a accumulé plus que de simples correctifs — nouvelles fonctionnalités, évolution de structure de base, ou tout ce qui correspond à la définition du chiffre **Minor** dans la procédure de référence. On veut alors que la prochaine version publiée porte un

numéro mineur (`X.Y+1.0`) plutôt qu'un simple correctif (`X.Y.Z+1`).

Aucun script à modifier : `scripts/version.js` détermine le numéro cible en comparant le **nom de la branche** au `X.Y.Z` actuel de `package.json` — peu importe si c'est le 2e ou le 3e chiffre qui change, dès que le nom de branche est « supérieur », c'est traité comme un nouveau cycle et le saut est automatique.

Procédure :

1. Se mettre à jour sur la branche de version en cours :

```
git checkout 11.0.25 && git pull
```

1. Créer la nouvelle branche mineure **depuis sa pointe** (conserve tout le travail déjà accumulé, betas comprises) :

```
git checkout -b 11.1.0
git push -u origin 11.1.0
```

1. **Ne pas supprimer ni réécrire l'ancienne branche** (`11.0.25`) : elle garde son historique et ses tags beta déjà publiés (ex. `11.0.25.1_beta`). C'est une exception volontaire à la règle « pas de `X.Y.Z` orphelin » de la procédure de référence — cette branche ne débouchera jamais sur un tag `_release`, ce qui vaut la peine d'être noté quelque part pour ne pas s'interroger plus tard.
2. **Prévenir l'équipe** : à partir de maintenant, `/fusion-dev` cible `11.1.0`, pas `11.0.25`. Quiconque a encore du travail en cours vers l'ancienne branche doit basculer dessus.
3. Rien à changer côté `package.json` : au prochain `/release-beta` sur `11.1.0`, le script constate que `package.json` est encore à l'ancien numéro (ex. `11.0.25.1`) et que la branche `11.1.0` lui est supérieure → nouveau cycle détecté automatiquement → saut direct à `11.1.0.1`.

Cas particulier — Travailler en parallèle sur deux versions (correctifs release en ligne / évolutions à venir)

Quand : en permanence, deux pistes de travail coexistent — corriger des bugs sur ce qui tourne déjà chez les clients, et développer les évolutions de la prochaine version. Ce cas concerne l'étape 1 (travail personnel), avant même d'arriver à `/fusion-dev`.

Piste	Usage	Limite	Branche `logeas-web`	Branche `logeas-lib`
Release en ligne	Corriger des bugs sur ce qui tourne actuellement chez les clients.	Correctifs uniquement — pas de nouvelle fonctionnalité, pas de changement de structure de base.	11.1.0	11.0.8
Prochaine version	Développer les évolutions de la prochaine release.	Pas encore en production — rien d'urgent à y corriger pour un client.	11.2.0	11.2.0

```
   PROD (clients)
        |
   [ web 11.1.0 / lib 11.0.8 ]  <- correctifs urgents
        |
        |  a reporter manuellement (cherry-pick / fusion)
        v
   [ web 11.2.0 / lib 11.2.0 ]  <- evolutions en cours
        |
        v
   future release
```

Piège fréquent : un correctif commité uniquement sur la branche « release en ligne » n'atterrit jamais tout seul sur la branche « prochaine version ». S'il n'est pas reporté à la main, il est perdu dès la sortie de la version suivante — le principe d'immutabilité d'une version publiée (voir la procédure de référence) empêche de le rattraper après coup.

Pourquoi ce n'est pas juste un `git checkout` : `logeas-web` ne dépend pas du code source de `logeas-lib` mais d'un paquet **déjà compilé et empaqueté** (fichier `.tgz`, référencé dans `package.json` avec le numéro de version en dur dans le chemin — ex. `file:../logeas-lib/dist/ngx-logeasweb-ui/ngx-logeasweb-ui-11.0.8.tgz`), pour être sûr de tester exactement ce qui sera publié. `dist/` (les paquets construits côté `logeas-lib`) et `package-lock.json` ne sont pas versionnés dans Git : un simple `git checkout` d'une autre branche ne restaure donc jamais cette chaîne — elle doit être rejouée intégralement (rebuild des 3 librairies, `npm pack` de chacune, déplacement du `.tgz`, 3 lignes à éditer dans `package.json`, `npm install`).

Outil : `scripts/switch-lib-version.ps1` (dans `logeas-web`) — automatise cette chaîne. Piloté par des profils dans `scripts/switch-lib-profiles.json` :

```
{
  "hotfix": { "webBranch": "11.1.0", "libBranch": "11.0.8" },
  "next":   { "webBranch": "11.2.0", "libBranch": "11.2.0" }
}
```

```
# Basculer en mode correctifs (release en ligne)
.\scripts\switch-lib-version.ps1 -Profile hotfix

# Basculer en mode évolutions (prochaine version)
.\scripts\switch-lib-version.ps1 -Profile next
```

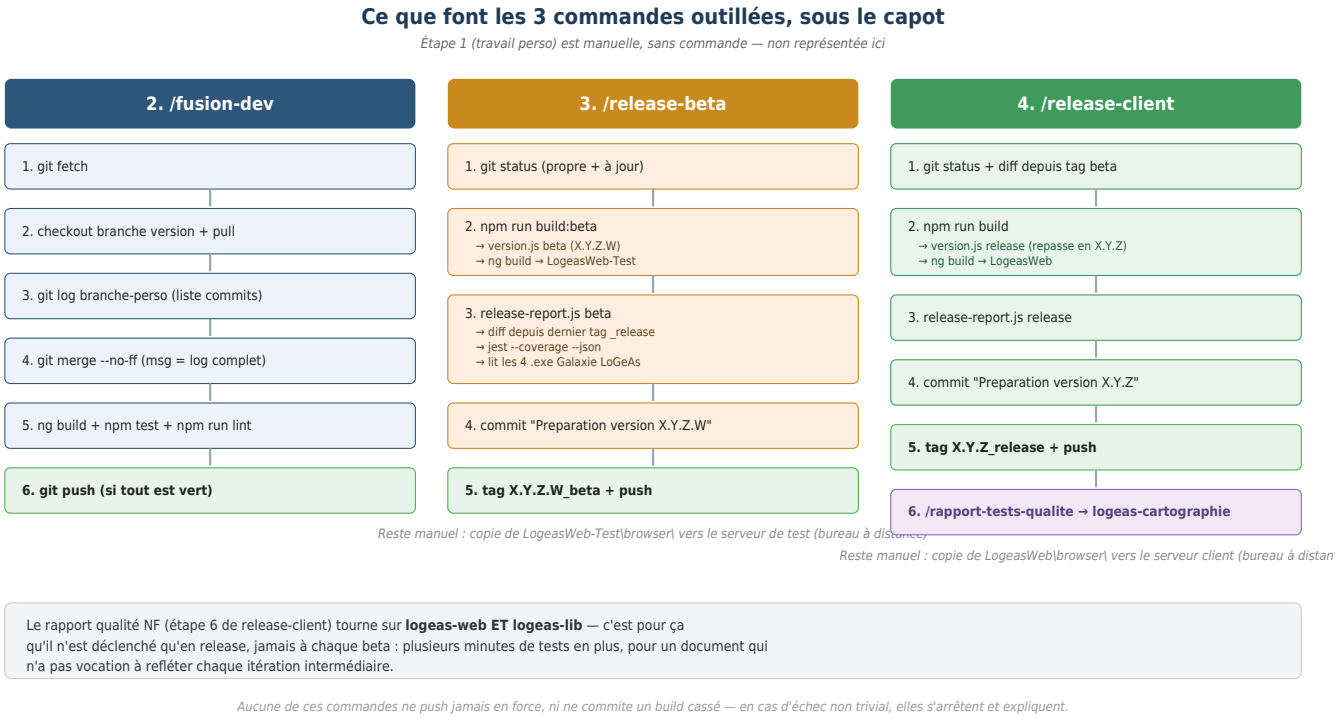
Le script :

- refuse de démarrer si l'un des deux dépôts (`logeas-web` ou `logeas-lib`) a des modifications non commitées — rien n'est écrasé silencieusement ;
- **compare l'historique des deux pistes et demande confirmation** si des commits existent sur une version antérieure et ne sont pas encore présents sur la cible — c'est le garde-fou contre le piège décrit plus haut ;
- checkout les branches, rebuild et repackage les 3 librairies `logeas-lib` (`ngx-logeascom-ui`, `ngx-logeasweb-ui`, `ngx-interface-ui`, dans cet ordre de dépendance), met à jour `package.json` de `logeas-web`, relance `npm install`.

À faire à chaque nouvelle release : quand une nouvelle branche mineure est créée (voir le cas

particulier précédent), mettre à jour `scripts/switch-lib-profiles.json` avec les nouveaux noms de branche — aucune modification du script lui-même n'est nécessaire.

Aide-mémoire : qui fait quoi, sous le capot



Équivalences SourceTree des commandes Git

Toutes les commandes Git listées ci-dessus sont exécutées automatiquement par les commandes Claude Code — vous n'avez normalement rien à taper à la main. Ce tableau sert pour les cas où on doit reproduire une étape manuellement (dépannage, vérification, ou si on préfère l'interface graphique) :

Commande Git	**Équivalent dans SourceTree**
`git fetch`	Bouton **Fetch** (barre d'outils du haut)
`git checkout <branche>`	Onglet **Branches** (panneau de gauche) → double-clic sur la branche
`git pull`	Bouton **Pull**

Commande Git	Équivalent dans SourceTree
`git status`	Onglet **File status** — visible en permanence, pas besoin de le lancer
`git add <fichiers>`	Onglet **File status** → cocher les fichiers à inclure dans le prochain commit
`git commit -m "…"`	Bouton **Commit** → saisir le message dans la zone de texte
`git merge –no-ff <branche>`	Clic droit sur la branche à fusionner (dans le graphe ou l'onglet Branches) → **Merge [branche] into [branche courante]** → cocher « Create a commit even if merge resolved via fast-forward »
`git tag -a X.Y.Z_release -m "…"`	Clic droit sur le commit dans le graphe → **Tag…** → cocher **Annotated tag**, saisir le nom et le message
`git push`	Bouton **Push** → dans la boîte de dialogue, cocher aussi les **tags** à pousser (case « Push all tags » ou sélection du tag spécifique) — sinon le tag reste local
`git log <a>..<b>`	Sélectionner la branche dans le graphe et lire l'historique visuellement ; le format exact utilisé par `/fusion-dev` (un séparateur technique par commit) est interne au script, pas à reproduire à la main

Point d'attention SourceTree spécifique aux tags : SourceTree ne pousse pas les tags automatiquement avec un `push` classique — il faut explicitement cocher le tag dans la fenêtre de push, ou faire un push dédié sur le tag après coup. C'est une source d'erreur fréquente (tag créé localement mais jamais visible sur `origin`).

Erreurs fréquentes à connaître

- **Confondre « beta » et « alpha »** : il n'y a que deux étapes outillées de génération, beta et release. Pas de troisième étape intermédiaire.
- **Lancer `npm run build` à la main pour un simple test de compilation** : ça déclenche le bump de version (`prebuild`). Pour juste vérifier que ça compile sans effet de bord, utiliser `ng build` directement (c'est ce que fait `/fusion-dev`).
- **Oublier que le rapport qualité NF ne se met à jour qu'en release**, pas en beta — si besoin d'un état des lieux ponctuel entre deux releases, relancer `/rapport-tests-qualite` manuellement.
- **Croire que le déploiement est automatique** : à aucune étape le contenu n'est copié sur un vrai serveur — c'est toujours une action manuelle, via bureau à distance.
- **Pousser un commit sans pousser le tag associé (SourceTree)** : voir l'encadré ci-dessus — un tag local qui n'existe pas sur `origin` ne sert à rien pour l'équipe.
- **Corriger un bug sur la branche « release en ligne » sans le reporter sur la branche d'évolutions** : le correctif est perdu dès que la version suivante sort, sans avertissement. Utiliser `scripts/switch-lib-version.ps1` pour basculer entre les deux pistes — il alerte si des commits d'une version antérieure ne sont pas encore reportés sur la cible.

Document généré le 11 août 2026, mis à jour le 12 août 2026 (ajout du cas particulier de bascule vers une version mineure), mis à jour le 18 août 2026 (ajout du cas particulier de travail en parallèle sur deux versions et de l'outil `switch-lib-version.ps1`) — à mettre à jour si le contenu des commandes `/fusion-dev`, `/release-beta`, `/release-client` ou `/rapport-tests-qualite`

évolue.

From:
https://wiki-logeas.fr/certif/ - **dokuwiki-certif**

Permanent link:
https://wiki-logeas.fr/certif/doku.php?id=certif:procedure:develop:gestionversionpratique&rev=1787046364

Last update: **2026/08/18 11:46**