Chapitre F — Ship, Deploy & Lifecycle

12 skills regroupés sous ce thème.

- freeze
- setup-deploy
- context-save
- context-restore
- sync-gbrain
- ship
- setup-gbrain
- landing-report
- document-release
- unfreeze
- land-and-deploy
- retro

freeze

freeze

> *Catalogue généré le 2026-05-11*

En une phrase

Verrouille les modifications de fichiers dans un seul dossier choisi, pour éviter que Claude touche par erreur du code en dehors de ta zone de travail.

Quand l'utiliser

- Tu débugues un module précis et tu veux empêcher Claude de "corriger" autre chose pendant qu'il y est.
- Tu lances un refactoring sur un dossier (`src/components/`) et tu veux garantir que rien d'autre ne bouge.
- Tu travailles sur une feature isolée et tu veux protéger le reste du projet d'effets de bord.
- Tu fais réviser ton code par Claude et tu veux qu'il reste confiné à un sous-dossier.

Comment l'invoquer

- **Slash command** : `/freeze`
- **Phrases déclencheurs (texte)** : "freeze", "restrict edits", "only edit this folder", "lock down edits"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `freeze` met en place une barrière technique : tu lui donnes un chemin de dossier, et à partir de ce moment-là, toute tentative de Claude d'éditer ou d'écrire un fichier situé en dehors de ce dossier est bloquée. Pas seulement signalée — réellement bloquée par un hook (un mécanisme de gstack qui intercepte les appels aux outils Edit et Write).

Concrètement, Claude te demande quel dossier verrouiller. Tu réponds par exemple `src/auth/`, et il enregistre ce périmètre pour toute la session. Si Claude essaie d'éditer un fichier hors de ce dossier (par exemple `src/billing/utils.ts`), l'opération est refusée automatiquement. Les autres actions (lire des fichiers, lancer du bash, faire des recherches) restent autorisées.

C'est une protection contre les dérapages, pas une sécurité absolue : un script bash mal écrit pourrait quand même modifier des fichiers hors zone. Mais pour le cas le plus courant (Claude qui se met à "améliorer" tout ce qu'il croise pendant un debug), ça suffit largement. Pour lever le verrouillage, tu utilises `/unfreeze` ou tu termines simplement la conversation.

Source

- **Plugin** : `gstack`
- **Nom interne** : `freeze`
- **Fichier** : `/home/thymon/.claude/skills/gstack/freeze/SKILL.md`

setup-deploy

setup-deploy

> *Catalogue généré le 2026-05-11*

En une phrase

Configure une fois pour toutes comment ton projet se déploie, pour que la commande `/land-and-deploy` sache automatiquement où envoyer ton code en production.

Quand l'utiliser

- Tu démarres un nouveau projet et tu veux préparer le déploiement automatisé.
- Tu changes d'hébergeur (Fly.io, Render, Vercel, Netlify, Heroku, Railway) et tu dois réenregistrer la config.
- Tu ajoutes un health check (page qui vérifie que le site répond) pour que les déploiements soient vérifiés.
- Tu travailles avec gstack pour la première fois sur un projet existant et tu veux que `/land-and-deploy` fonctionne.

Comment l'invoquer

- **Slash command** : `/setup-deploy`
- **Phrases déclencheurs (texte)** : "setup deploy", "configure deployment", "set up land-and-deploy", "how do I deploy with gstack", "add deploy config"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `setup-deploy` est un assistant de configuration qui détecte automatiquement comment ton projet se déploie, puis enregistre cette information dans le fichier `CLAUDE.md` du projet. Une fois la config posée, les autres skills (notamment `/land-and-deploy`) savent où envoyer ton code, comment vérifier que le déploiement a réussi, et quelle URL surveiller.

Claude commence par fouiller ton projet pour repérer les fichiers révélateurs : `fly.toml` pour Fly.io, `vercel.json` pour Vercel, `netlify.toml` pour Netlify, `Procfile` pour Heroku, ou un workflow GitHub Actions de déploiement. Il devine l'URL de production, vérifie si le CLI de la plateforme est installé, et te propose une config. Si rien n'est détecté (cas custom), il te pose les questions essentielles via un mini-questionnaire : comment se déclenche le déploiement, quelle URL surveiller, comment vérifier qu'il a réussi.

Tout est sauvegardé dans une section `## Deploy Configuration` de ton `CLAUDE.md`. Tu peux relancer `/setup-deploy` à tout moment pour modifier la config — c'est idempotent (relancer ne crée pas de doublons), donc pas de risque. Le skill ne déploie rien lui-même ; il sert uniquement à préparer le terrain pour `/land-and-deploy`.

Source

- **Plugin** : `gstack`
- **Nom interne** : `setup-deploy`
- **Fichier** : `/home/thymon/.claude/skills/gstack/setup-deploy/SKILL.md`

context-save

context-save

> *Catalogue généré le 2026-05-11*

En une phrase

Sauvegarde l'état complet de ton travail en cours (décisions, fichiers modifiés, prochaines étapes) pour pouvoir reprendre plus tard sans rien perdre.

Quand l'utiliser

- Tu dois interrompre une session pour la nuit, le week-end, ou parce qu'on t'appelle ailleurs.
- Tu veux passer à un autre sujet sans perdre où tu en étais sur le précédent.
- Tu veux transférer le contexte d'un workspace Conductor à un autre (ou d'une machine à une autre).
- Tu sens que la conversation devient longue et tu veux figer un point de reprise propre.
- Tu vas tester quelque chose de risqué et tu veux pouvoir retomber sur tes pieds.

Comment l'invoquer

- **Slash command** : `/context-save` (optionnellement avec un titre : `/context-save refonte-auth`)
- **Phrases déclencheurs (texte)** : "save progress", "save state", "save my work", "context save"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `context-save` joue le rôle d'un ingénieur senior qui prend des notes méticuleuses avant de quitter le bureau. Il capture l'état de ton travail en deux temps : d'abord l'état technique (la branche git, les fichiers modifiés, les commits récents, le diff en cours), ensuite l'état mental (sur quoi tu travailles, quelles décisions ont été prises, ce qu'il reste à faire, les pistes essayées qui n'ont pas marché).

Le tout est enregistré dans un fichier markdown dans `~/.gstack/projects/<ton-projet>/checkpoints/` avec un nom horodaté. Tu peux donner un titre à ta sauvegarde (`/context-save refonte-auth`) ou laisser Claude en deviner un. Les fichiers sont en mode append-only : chaque sauvegarde crée un nouveau fichier, rien n'est jamais écrasé. Tu peux donc empiler autant de points de reprise que tu veux.

Important : ce skill ne modifie jamais ton code. Il se contente de lire l'état actuel et d'écrire le fichier de sauvegarde. Pour reprendre plus tard, tu utilises son skill jumeau `/context-restore`. Tu peux aussi lister toutes tes sauvegardes avec `/context-save list` (par défaut pour la branche actuelle, ou `--all` pour toutes branches confondues). Ancien nom du skill : `/checkpoint`, renommé parce que Claude Code utilise maintenant `/checkpoint` pour autre chose.

Source

- **Plugin** : `gstack`
- **Nom interne** : `context-save`
- **Fichier** : `/home/thymon/.claude/skills/gstack/context-save/SKILL.md`

context-restore

context-restore

> *Catalogue généré le 2026-05-11*

En une phrase

Recharge la dernière sauvegarde de travail créée par `/context-save` pour reprendre exactement là où tu t'étais arrêté, même sur une autre branche ou un autre workspace.

Quand l'utiliser

- Tu reprends ton travail le lendemain matin et tu veux retrouver le fil.
- Tu changes de workspace Conductor et tu veux récupérer le contexte d'un autre.
- Tu ne te souviens plus où tu en étais sur un projet et tu veux que Claude te le rappelle.
- Tu veux relire ce qui était décidé avant de continuer.
- Tu reviens sur un projet après plusieurs jours d'absence.

Comment l'invoquer

- **Slash command** : `/context-restore` (ou `/context-restore <fragment-du-titre>` pour cibler une sauvegarde précise)
- **Phrases déclencheurs (texte)** : "resume where i left off", "restore context", "where was i", "pick up where i left off", "context restore"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `context-restore` est le pendant de `/context-save`. Il joue le rôle d'un ingénieur senior qui lit les notes méticuleuses laissées par un collègue (toi, hier) pour reprendre le travail sans perdre une minute. Il cherche les sauvegardes dans `~/.gstack/projects/<ton-projet>/checkpoints/`, charge la plus récente, et te la présente proprement.

Par défaut, il prend la sauvegarde la plus récente toutes branches confondues. C'est volontaire : si tu travailles sur Conductor et que tu changes de workspace, tu veux pouvoir retomber sur ce que tu faisais ailleurs. Si tu lui passes un fragment de titre (`/context-restore auth-refactor`), il trouve la sauvegarde correspondante. Il te montre : ce sur quoi tu travaillais, les décisions prises, le travail restant, les notes (pièges, idées essayées qui n'ont pas marché).

Si la sauvegarde a été créée sur une autre branche que celle où tu te trouves actuellement, Claude te le signale et te suggère de changer de branche avant de reprendre. À la fin, il te propose trois options : continuer sur le premier point restant, afficher le fichier complet, ou simplement noter que tu as récupéré le contexte. Comme `/context-save`, il ne modifie jamais de code — il lit et présente, c'est tout.

Source

- **Plugin** : `gstack`
- **Nom interne** : `context-restore`
- **Fichier** : `/home/thymon/.claude/skills/gstack/context-restore/SKILL.md`

sync-gbrain

sync-gbrain

> *Catalogue généré le 2026-05-11*

En une phrase

Met à jour la mémoire de ton projet dans gbrain (un index sémantique de ton code) pour que Claude puisse faire des recherches intelligentes au lieu de simplement chercher du texte.

Quand l'utiliser

- Tu viens d'ajouter beaucoup de code et tu veux que Claude le retrouve par concept (pas juste par mot-clé).
- Tu as installé gbrain via `/setup-gbrain` et tu lances l'indexation pour la première fois.
- Tu remarques que `gbrain search` ne trouve plus tes nouveaux fichiers.
- Tu veux forcer une réindexation complète (option `--full`) après un gros refactor.
- Tu changes de branche ou de workspace et tu veux que la recherche reste à jour.

Comment l'invoquer

- **Slash command** : `/sync-gbrain` (options : `--full`, `--code-only`, `--dry-run`, `--no-memory`, `--no-brain-sync`, `--quiet`)
- **Phrases déclencheurs (texte)** : "sync gbrain", "refresh gbrain", "re-index this repo", "gbrain search isn't finding things"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `sync-gbrain` est le pendant rafraîchissement de `/setup-gbrain`. Pendant que `/setup-gbrain` installe et configure gbrain une fois pour toutes, `/sync-gbrain` se relance à chaque fois que tu veux que la mémoire reflète l'état actuel de ton code. Concrètement, gbrain est un cerveau local (une petite base de données) qui transforme ton code en représentations sémantiques. Une fois indexé, Claude peut chercher "où est la logique d'authentification" au lieu de chercher littéralement le mot "auth".

Le sync se fait en trois étapes : le code (ré-indexation des fichiers source de ton dépôt), la mémoire (les artefacts gstack comme les plans CEO, les designs), et la synchro avec une éventuelle copie cloud. Par défaut, c'est rapide (environ 50 ms) car seul ce qui a changé depuis la dernière fois est traité. Une réindexation complète (`--full`) prend 25 à 35 minutes sur un gros projet — à réserver aux cas où tu sens que la recherche n'est plus fiable.

Le skill commence par vérifier que `/setup-gbrain` a déjà été lancé. Si non, il s'arrête net : impossible de synchroniser quelque chose qui n'existe pas. Il met aussi à jour automatiquement la section `## GBrain Search Guidance` de ton `CLAUDE.md` pour que Claude sache préférer `gbrain search` à `Grep` dans tes prochaines sessions. Idempotent : tu peux le relancer autant de fois que tu veux sans casser quoi que ce soit.

Source

- **Plugin** : `gstack`
- **Nom interne** : `sync-gbrain`
- **Fichier** : `/home/thymon/.claude/skills/gstack/sync-gbrain/SKILL.md`

ship

ship

> *Catalogue généré le 2026-05-11*

En une phrase

Lance le workflow complet d'expédition de ton code : tests, revue de diff, bump de version, mise à jour du CHANGELOG, commit, push, et création de la Pull Request, tout automatiquement.

Quand l'utiliser

- Tu as fini une feature et tu veux la pousser sur GitHub en tant que PR prête à merger.
- Tu veux un workflow d'expédition strict (tests + revue + doc) avant chaque déploiement.
- Tu déploies plusieurs fois par jour et tu veux que tout soit automatisé et cohérent.
- Tu veux que la version (VERSION) et le CHANGELOG soient mis à jour proprement à chaque sortie.

Comment l'invoquer

- **Slash command** : `/ship`
- **Phrases déclencheurs (texte)** : "ship it", "create a pr", "push to main", "deploy this", "ship", "merge and push", "get it deployed"
- **Auto-invocation** : □ Oui — quand tu dis "le code est prêt", "on peut pousser", "crée une PR", Claude propose ce skill.

Description détaillée

Le skill `ship` est le workflow le plus ambitieux de gstack. Il enchaîne automatiquement, sans demander confirmation à chaque étape, toutes les opérations nécessaires pour transformer ton travail en cours en Pull Request prête à merger. La règle d'or : dire "/ship", c'est dire "vas-y, fais tout". Le skill ne s'arrête que pour les vraies décisions humaines (bump de version majeur, conflits de merge, faille de revue critique).

Le workflow se décompose en une vingtaine d'étapes : préflight (vérifier que tu n'es pas sur la branche principale, lire `git status`), détection de la branche de base, exécution des tests, audit de couverture, vérification que les éléments du plan sont marqués DONE, revue avant atterrissage (pre-landing review qui détecte les bugs subtils), revue adversariale automatique (subagent Claude + challenge Codex), bump du VERSION avec choix automatique du niveau (PATCH/MICRO en auto, MINOR/MAJOR sur demande), génération de l'entrée CHANGELOG dans la voix produit, commit propre, push, et création de la PR avec le rapport complet en description.

C'est aussi un workflow idempotent : tu peux relancer `/ship` autant de fois que tu veux. Les vérifications tournent à chaque fois, mais les actions déjà faites (push, création de PR) ne sont pas dupliquées — la PR existante est mise à jour. Le skill produit à la fin l'URL de ta Pull Request. Si tu veux ensuite la merger et la déployer, tu enchaînes avec `/land-and-deploy`.

Source

- **Plugin** : `gstack`
- **Nom interne** : `ship`
- **Fichier** : `/home/thymon/.claude/skills/gstack/ship/SKILL.md`

setup-gbrain

setup-gbrain

> *Catalogue généré le 2026-05-11*

En une phrase

Installe et configure gbrain (la mémoire intelligente de tes projets) sur ton Mac, puis le branche à Claude Code pour qu'il puisse l'interroger comme un outil.

Quand l'utiliser

- Tu démarres avec gstack et tu veux activer la recherche sémantique sur ton code.
- Tu veux une mémoire persistante qui se souvient des décisions et patterns au-delà d'une session.
- Tu changes de machine et tu veux migrer ta mémoire vers une base partagée (Supabase).
- Tu veux passer d'une base locale (PGLite) à une base partagée entre plusieurs machines.
- Tu veux te connecter à un gbrain distant déjà hébergé par toi ou un coéquipier.

Comment l'invoquer

- **Slash command** : `/setup-gbrain` (options : `--repo`, `--switch`, `--resume-provision`, `--cleanup-orphans`)
- **Phrases déclencheurs (texte)** : "setup gbrain", "install gbrain", "connect gbrain", "start gbrain", "configure gbrain"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `setup-gbrain` est l'assistant d'installation pour gbrain, un système de mémoire persistante qui indexe ton code et tes artefacts (plans, designs, rapports). Une fois installé, Claude peut l'utiliser via deux canaux : en ligne de commande (`gbrain search`) ou comme outil MCP (un protocole qui permet à Claude d'appeler des services externes).

Il propose quatre chemins d'installation, présentés sous forme de questionnaire. Path 1 : tu as déjà une URL Supabase (cas où ta mémoire est hébergée dans le cloud), tu la colles. Path 2a/2b : créer un nouveau projet Supabase, soit automatiquement (en passant ton token Supabase), soit manuellement (tu signes sur supabase.com). Path 3 : PGLite local — pas de compte, environ 30 secondes, ta mémoire reste sur ton Mac uniquement. Path 4 : te connecter à un gbrain distant déjà en service (par exemple si un coéquipier ou une autre machine à toi héberge déjà un serveur `gbrain serve`).

Le skill détecte l'état actuel avant de proposer quoi que ce soit (gbrain déjà installé, déjà configuré, etc.) et saute les étapes inutiles. Il gère aussi la politique de confiance par dépôt (`/setup-gbrain --repo`), la migration d'engine (PGLite ↔ Supabase via `--switch`), et nettoyer les projets Supabase orphelins (`--cleanup-orphans`). Une fois fini, tu utilises `/sync-gbrain` pour indexer ton code et garder la mémoire à jour.

Source

- **Plugin** : `gstack`
- **Nom interne** : `setup-gbrain`
- **Fichier** : `/home/thymon/.claude/skills/gstack/setup-gbrain/SKILL.md`

landing-report

landing-report

> *Catalogue généré le 2026-05-11*

En une phrase

Affiche un tableau de bord en lecture seule qui montre quelles versions sont déjà réservées par des PR ouvertes et quelle version `/ship` réserverait la prochaine fois.

Quand l'utiliser

- Tu travailles en parallèle sur plusieurs branches (Conductor) et tu veux savoir quelle version chaque branche va revendiquer.
- Tu veux voir d'un coup d'œil toutes les Pull Requests ouvertes et leur état d'avancement.
- Tu te demandes quel numéro de version `/ship` va attribuer à ta prochaine livraison.
- Tu coordonnes plusieurs développeurs (ou plusieurs sessions Claude) sur le même dépôt.

Comment l'invoquer

- **Slash command** : `/landing-report`
- **Phrases déclencheurs (texte)** : "landing report", "version queue", "ship queue", "what version comes next", "show open PR versions", "what's in the queue"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `landing-report` est un mini tableau de bord pour le système d'expédition "workspace-aware ship" de gstack. Quand tu lances `/ship`, gstack réserve un slot de version (par exemple v1.7.0.0) pour ta branche. Si tu travailles en même temps sur plusieurs branches dans des workspaces Conductor différents, il faut éviter que deux branches réservent le même numéro de version — d'où ce rapport.

Le skill ne modifie rien : il se contente de lire l'état actuel et de te le présenter. Il liste les PR ouvertes avec leur version revendiquée, les workspaces Conductor voisins qui ont du travail en cours et qui vont probablement expédier bientôt, et te dit quelle version `/ship` choisirait pour toi maintenant. C'est utile pour la coordination, surtout en équipe ou quand tu jongles entre plusieurs branches d'évolution.

C'est volontairement un skill très léger : pas de questions, pas d'actions, juste une photographie de l'état du queue. Tu lances la commande, tu regardes le rapport, tu décides. Pour effectivement réserver une version et livrer du code, tu utilises ensuite `/ship`.

Source

- **Plugin** : `gstack`
- **Nom interne** : `landing-report`
- **Fichier** : `/home/thymon/.claude/skills/gstack/landing-report/SKILL.md`

document-release

document-release

> *Catalogue généré le 2026-05-11*

En une phrase

Met à jour automatiquement toute la documentation de ton projet (README, ARCHITECTURE, CONTRIBUTING, CLAUDE.md, CHANGELOG) après une livraison, pour que rien ne devienne obsolète sans qu'on s'en rende compte.

Quand l'utiliser

- Tu viens de faire `/ship` et tu veux synchroniser la doc avec ce qui vient de partir.
- Une PR est sur le point d'être mergée et tu veux que la doc reflète les changements avant le merge.
- Tu sens que ton README ou ton ARCHITECTURE.md a pris du retard sur le code.
- Tu veux polir la voix du CHANGELOG (le rendre orienté utilisateur plutôt que technique).
- Tu veux marquer comme terminés les éléments de ton TODOS.md qui sont effectivement faits.

Comment l'invoquer

- **Slash command** : `/document-release`
- **Phrases déclencheurs (texte)** : "update docs after ship", "document what changed", "post-ship docs", "update the docs", "sync documentation"
- **Auto-invocation** : □ Oui — Claude propose ce skill après un `/ship` réussi ou après le merge d'une PR.

Description détaillée

Le skill `document-release` tourne après que ton code soit commité (et idéalement après que la PR soit créée) mais avant qu'elle ne soit mergée. Sa mission : faire le tour de tous les fichiers de documentation du projet, les comparer au diff de la branche, et appliquer les corrections évidentes automatiquement. Les changements risqués ou subjectifs te sont posés en question.

Il fonctionne en plusieurs passes. D'abord, il analyse les commits de ta branche et classifie les changements (nouvelle feature, comportement modifié, suppression, infra). Ensuite, il fait l'audit fichier par fichier : pour chaque `.md` (README, ARCHITECTURE, CONTRIBUTING, CLAUDE.md, etc.), il vérifie que les exemples sont toujours valables, que les commandes listées existent encore, que la structure du projet décrite correspond bien à ce qu'il y a sur disque. Pour `CONTRIBUTING.md`, il fait même un "test de premier contributeur" : chaque commande d'installation est passée en revue comme si tu débarquais sur le projet.

Les corrections factuelles évidentes (paths, comptes, version numbers, items à ajouter dans un tableau) sont appliquées sans demander. Les changements narratifs (philosophie, sécurité, suppression de sections) ou les gros rewrites te sont soumis. Il polit aussi le CHANGELOG (sans jamais réécrire les entrées, juste polir la voix), marque les TODOS terminés, et te propose de bumper VERSION si nécessaire. Le but : qu'à chaque release, ta doc reste vivante et alignée sur ton code.

Source

- **Plugin** : `gstack`
- **Nom interne** : `document-release`
- **Fichier** : `/home/thymon/.claude/skills/gstack/document-release/SKILL.md`

unfreeze

unfreeze

> *Catalogue généré le 2026-05-11*

En une phrase

Lève le verrouillage de dossier posé par `/freeze` pour permettre à nouveau les modifications dans tout le projet, sans avoir à fermer la session.

Quand l'utiliser

- Tu avais lancé `/freeze` pour te concentrer sur un dossier, et tu as fini — tu veux rouvrir le champ.
- Tu veux élargir la zone éditable sans terminer la conversation et en redémarrer une.
- Tu réalises que tu dois aussi toucher un fichier hors du périmètre verrouillé.
- Tu changes de sujet de travail et tu n'as plus besoin de la restriction.

Comment l'invoquer

- **Slash command** : `/unfreeze`
- **Phrases déclencheurs (texte)** : "unfreeze", "unlock edits", "remove freeze", "allow all edits", "unlock all directories", "remove edit restrictions"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `unfreeze` est le pendant minimaliste de `/freeze`. Sa seule mission : effacer le fichier d'état créé par `/freeze` pour que les hooks (les intercepteurs qui bloquaient les éditions hors zone) laissent à nouveau tout passer. Concrètement, il supprime un petit fichier dans `~/.gstack/` qui contenait le chemin du dossier verrouillé.

Une fois `/unfreeze` exécuté, tu peux à nouveau éditer n'importe quel fichier du projet, exactement comme avant d'avoir lancé `/freeze`. Le skill te confirme l'opération en te rappelant quel dossier était verrouillé, au cas où tu veuilles vérifier que c'était bien celui que tu pensais.

Note utile : les hooks restent enregistrés pour la session (ils ne disparaissent vraiment qu'à la fermeture de la conversation), mais ils n'ont plus d'effet puisque le fichier d'état est absent. Si tu veux re-verrouiller plus tard sur un autre dossier, tu relances simplement `/freeze` — pas besoin de redémarrer la session.

Source

- **Plugin** : `gstack`
- **Nom interne** : `unfreeze`
- **Fichier** : `/home/thymon/.claude/skills/gstack/unfreeze/SKILL.md`

land-and-deploy

land-and-deploy

> *Catalogue généré le 2026-05-11*

En une phrase

Merge ta Pull Request, attend que le déploiement passe, puis vérifie automatiquement que la production tourne bien — l'étape qui suit `/ship`.

Quand l'utiliser

- Tu as fini `/ship` (ta PR est créée) et tu veux que tout le monde la merge et déploie sans toi.
- Tu veux que Claude surveille le déploiement et te dise si la production est saine après.
- Tu veux que la vérification post-déploiement (canary) tourne automatiquement.
- Tu déploies plusieurs fois par jour et tu veux que tout l'enchaînement soit géré pour toi.

Comment l'invoquer

- **Slash command** : `/land-and-deploy` (avec arguments : `/land-and-deploy #123`, `/land-and-deploy <url>`, `/land-and-deploy #123 <url>`)
- **Phrases déclencheurs (texte)** : "merge", "land", "deploy", "merge and verify", "land it", "ship it to production", "merge and deploy"
- **Auto-invocation** : □ Oui — Claude propose ce skill quand tu dis que c'est prêt à merger ou à déployer.

Description détaillée

Le skill `land-and-deploy` joue le rôle d'un ingénieur release qui a déployé des milliers de fois en production. Il sait que les deux pires moments en software sont les merges qui cassent la prod et les merges qui restent en file d'attente 45 minutes pendant que tu fixes l'écran. Sa mission : gérer ces deux moments gracieusement, en gardant ta confiance via des vérifications claires.

Le workflow démarre là où `/ship` s'arrête. `/ship` a créé la PR ; `/land-and-deploy` la merge, attend que la CI et le déploiement tournent, puis vérifie la santé de la production via des checks canary (un canari, des vérifications légères qui passent après chaque déploiement pour détecter les régressions). Sur ta première utilisation pour un projet, il fait un dry run : il détecte ton infrastructure de déploiement (Fly.io, Vercel, Netlify, Heroku, Railway), teste que ses commandes fonctionnent, et te montre exactement ce qui va se passer étape par étape avant de toucher quoi que ce soit. Une fois validé, il enregistre la config pour ne plus jamais reposer la question.

Sur les runs suivants, c'est en mode efficace : statut bref, peu d'explications. Le workflow est majoritairement automatisé. Il ne s'arrête que pour les vrais blocages : authentification GitHub manquante, pas de PR pour la branche, échecs de CI, conflits, échec de déploiement (avec proposition de revert), problèmes de santé détectés par le canary (avec proposition de revert). Le merge method (squash/merge/rebase) est auto-détecté depuis les settings du dépôt — pas de question pour ça non plus.

Source

- **Plugin** : `gstack`
- **Nom interne** : `land-and-deploy`
- **Fichier** : `/home/thymon/.claude/skills/gstack/land-and-deploy/SKILL.md`

retro

retro

> *Catalogue généré le 2026-05-11*

En une phrase

Génère une rétrospective d'ingénierie sur les 7 derniers jours (ou la fenêtre que tu choisis) en analysant tes commits, ton rythme de travail, et la qualité de ton code, avec encouragements et axes de progression.

Quand l'utiliser

- Vendredi soir ou en fin de sprint, tu veux un récap de ce qui a été livré cette semaine.
- Tu veux comparer ta semaine actuelle à la précédente (rythme, volume, types de commits).
- Tu veux une vue d'équipe : qui a fait quoi, sur quels fichiers, avec quels patterns.
- Tu veux un rapport cross-projets sur tous tes outils d'IA (Claude Code, Cursor, etc.) avec `/retro global`.
- Tu veux un suivi de tendances dans le temps : as-tu progressé sur la couverture de tests, le nombre de PR, le sang-froid ?

Comment l'invoquer

- **Slash command** : `/retro` (variantes : `/retro 24h`, `/retro 14d`, `/retro 30d`, `/retro compare`, `/retro compare 14d`, `/retro global`, `/retro global 14d`)
- **Phrases déclencheurs (texte)** : "weekly retro", "what did we ship", "engineering retrospective"
- **Auto-invocation** : □ Oui — Claude propose ce skill en fin de semaine ou de sprint.

Description détaillée

Le skill `retro` produit une rétrospective d'ingénierie complète à partir de l'historique git de ton dépôt. Par défaut, il analyse les 7 derniers jours, mais tu peux choisir 24h, 14j ou 30j. Il identifie automatiquement qui est l'utilisateur (toi, via `git config user.name`) et oriente le récit autour de toi : "ce que TU as livré" vs "ce que tes coéquipiers ont livré".

Le skill collecte beaucoup de signaux : tous les commits avec leur auteur et leurs stats, la répartition entre code de tests et code de production, les timestamps pour détecter les sessions de travail et la distribution horaire, les fichiers les plus touchés (hotspots), les numéros de PR mentionnés, qui touche quels fichiers, le nombre de commits par auteur, l'historique des triages Greptile, le contenu de ton `TODOS.md`, le nombre de tests, et même les données de télémétrie d'usage des skills gstack. Il croise tout ça pour produire un récit utile.

Le mode `compare` met côte à côte deux périodes pour voir les tendances (tu as livré 30 % de plus cette semaine, tes tests ont augmenté de 12 %, etc.). Le mode `global` ne nécessite même pas d'être dans un dépôt git — il fait une rétro transverse à tous tes projets et tous tes outils d'IA. Il intègre aussi les apprentissages passés (`gstack-learnings-search`) pour rappeler quand un pattern déjà observé revient. Tu peux ajouter du contexte non-git via un fichier `~/.gstack/retro-context.md` (réunions, décisions hors code) pour enrichir le récit. C'est aussi un outil pour les seniors et les CTO qui veulent voir où passe le temps de leur équipe.

Source

- **Plugin** : `gstack`
- **Nom interne** : `retro`
- **Fichier** : `/home/thymon/.claude/skills/gstack/retro/SKILL.md`