01 — gstack (plugin natif)

46 skills du plugin gstack, organisés par chapitre thématique.

    - make-pdf
    - learn
    - skillify
    - gstack-upgrade

- □□ Index — 01 — gstack (plugin natif)

Chapitre A — Design & UI

5 skills regroupés sous ce thème.

plan-design-review

plan-design-review

> *Catalogue généré le 2026-05-11*

En une phrase

Relit ton plan de fonctionnalité avec un œil de designer, note chaque dimension visuelle sur 10 et te dit ce qu'il faudrait changer pour qu'elle devienne excellente.

Quand l'utiliser

- Quand tu as rédigé un plan ou un design doc et tu veux le passer au crible avant de coder.
- Pour repérer les erreurs UX, de hiérarchie ou d'accessibilité avant qu'elles soient dans le code.
- Quand tu hésites entre plusieurs approches visuelles et tu veux un avis structuré.
- Pendant une session en plan mode, pour valider qu'une feature UI est bien pensée.

Comment l'invoquer

- **Slash command** : `/plan-design-review`
- **Phrases déclencheurs (texte)** : "review the design plan" / "design critique" / "review ux plan" / "check design decisions"
- **Auto-invocation** : Oui — Claude suggère ce skill de lui-même quand tu as un plan avec des composants UI/UX qui mérite une relecture avant l'implémentation.

Description détaillée

Ce skill fonctionne comme une revue de plan menée par un designer senior. Il lit ton document de plan, identifie les dimensions visuelles et UX importantes (hiérarchie, typographie, espacement, motion, accessibilité, états vides, etc.), et donne une note 0-10 sur chacune. Pour chaque note inférieure à 10, il explique précisément ce qui manque et ce qu'il faudrait modifier pour atteindre la note maximale.

Contrairement à `/design-review` qui audite un site déjà en ligne, `plan-design-review` travaille uniquement sur du texte — sur la phase amont, avant que le code soit écrit. Il fonctionne en mode interactif : il pose des questions ciblées si quelque chose manque dans ton plan, propose des options, et modifie directement le plan pour intégrer les corrections.

C'est l'équivalent design des skills `/plan-ceo-review` (stratégie) et `/plan-eng-review` (architecture). Les trois peuvent être enchaînés via `/autoplan`. Il est conçu pour fonctionner même en plan mode (mode où Claude ne touche pas au code), ce qui est exactement le moment où tu veux ce regard critique.

Source

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

design-review

design-review

> *Catalogue généré le 2026-05-11*

En une phrase

Audite ton site déjà en ligne avec un œil de designer, repère les problèmes visuels et UX, puis les corrige directement dans le code en commit après commit.

Quand l'utiliser

- Quand ton site marche mais "fait un peu cheap" et tu veux le polir.
- Pour repérer les soucis d'alignement, d'espacement, de hiérarchie, ou de typographie qu'on ne voit plus à force de regarder son propre site.
- Pour traquer les "patterns AI slop" (les tics visuels qui trahissent un design généré).
- Avant de montrer ton site à quelqu'un et tu veux qu'il fasse "pro".

Comment l'invoquer

- **Slash command** : `/design-review`
- **Phrases déclencheurs (texte)** : "audit the design" / "visual QA" / "check if it looks good" / "design polish" / "visual design audit" / "fix design issues"
- **Auto-invocation** : Oui — Claude suggère ce skill quand tu mentionnes une incohérence visuelle ou que tu veux peaufiner l'aspect d'un site déjà en ligne.

Description détaillée

Ce skill ouvre le navigateur invisible sur ton site, prend des captures d'écran, et les analyse avec un regard de designer. Il cherche des problèmes concrets : espacements incohérents, alignements ratés, contraste insuffisant, hiérarchie typographique floue, interactions trop lentes, ou patterns trop "génériques AI".

Une fois qu'il a sa liste de problèmes, il ne se contente pas de te la donner — il les corrige un par un directement dans le code source. Chaque correction est faite atomiquement (un commit = un fix), avec une capture avant/après pour prouver que le souci est résolu. C'est une vraie boucle : repérer, corriger, vérifier, recommencer.

À ne pas confondre avec `/plan-design-review`, qui fait la même chose mais sur un PLAN (du texte), avant que le code existe. `design-review` travaille sur le site vivant après le déploiement.

C'est l'outil parfait pour la dernière passe avant de montrer quelque chose. Tu peux le lancer plusieurs fois — chaque passe attrape ce que la précédente a manqué.

Source

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

design-html

design-html

> *Catalogue généré le 2026-05-11*

En une phrase

Transforme un design approuvé (maquette, plan, ou simple description) en HTML/CSS de qualité production, prêt à être servi en ligne, sans dépendances.

Quand l'utiliser

- Quand tu as choisi une maquette via `/design-shotgun` et tu veux en faire une vraie page web.
- Quand un plan CEO ou design est validé et il faut maintenant le matérialiser en code.
- Quand tu décris une page à Claude et tu veux qu'il te ponde le HTML d'un coup, sans framework lourd.
- Pour créer une landing page, une page produit, ou un mini-site léger qui doit charger vite.

Comment l'invoquer

- **Slash command** : `/design-html`
- **Voice triggers** : « build the design » · « code the mockup » · « make it real »
- **Phrases déclencheurs (texte)** : "finalize this design" / "turn this into HTML" / "build me a page" / "implement this design"
- **Auto-invocation** : □ Oui — Claude suggère ce skill quand tu viens d'approuver un design ou que tu as un plan prêt à coder.

Description détaillée

Ce skill génère du HTML/CSS "Pretext-native" — une approche où le texte coule naturellement, où les hauteurs sont calculées dynamiquement, et où les layouts s'adaptent au contenu. Pas de JavaScript lourd, pas de framework, juste environ 30 Ko de code total pour rendre la page autonome.

Il sait s'enchaîner avec d'autres skills : il peut partir d'une maquette validée par `/design-shotgun`, d'un plan stratégique de `/plan-ceo-review`, du contexte d'une revue design `/plan-design-review`, ou simplement d'une description écrite par toi.

Il a un routage intelligent : selon le type de design demandé (landing, dashboard, formulaire, etc.), il choisit les bons patterns Pretext à appliquer. Le résultat est censé être directement publiable, pas un brouillon à retoucher.

L'intérêt par rapport à un "Claude qui code une page" classique : le résultat est pensé pour être responsive, accessible et léger dès le départ, plutôt que de faire un Tailwind générique qu'il faudra ensuite nettoyer.

Source

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

design-shotgun

design-shotgun

> *Catalogue généré le 2026-05-11*

En une phrase

Génère plusieurs variantes de design en parallèle, les affiche côte à côte sur un tableau de comparaison, et te laisse choisir celle qui te plaît avant d'aller plus loin.

Quand l'utiliser

- Quand tu commences une page et tu n'as pas d'idée précise de ce que tu veux visuellement.
- Quand tu n'aimes pas comment ça rend mais tu ne sais pas comment l'expliquer — laisse Claude proposer plusieurs pistes.
- Pour explorer rapidement plusieurs directions artistiques avant de t'engager.
- Quand tu veux un "brainstorm visuel" comme on en ferait avec un designer humain.

Comment l'invoquer

- **Slash command** : `/design-shotgun`
- **Phrases déclencheurs (texte)** : "explore designs" / "show me options" / "design variants" / "visual brainstorm" / "I don't like how this looks"
- **Auto-invocation** : □ Oui — Claude suggère ce skill quand tu décris une feature UI sans avoir vu à quoi elle pourrait ressembler.

Description détaillée

Ce skill fonctionne comme un "shotgun" : il tire plein de cartouches en même temps. Il génère plusieurs maquettes différentes pour la même demande, avec des directions visuelles distinctes (par exemple : une version minimaliste, une version colorée, une version dense, une version dans un style éditorial, etc.).

Une fois les variantes générées, il ouvre un tableau de comparaison où elles s'affichent côte à côte. Tu peux les regarder, donner ton feedback structuré (j'aime ceci, je n'aime pas cela), et le skill itère en générant une nouvelle vague basée sur ton retour.

C'est un skill autonome — tu peux le lancer n'importe quand, même sans avoir d'autre skill en cours. Il se branche bien sur `/design-html` derrière (une fois ta maquette préférée choisie, ce skill la transforme en code) ou sur `/plan-design-review` (pour critiquer le plan associé).

L'idée derrière, c'est que les humains choisissent mieux quand ils comparent que quand ils notent dans le vide. Au lieu de te demander "tu veux quoi ?" sans contexte, le skill te dit "voilà 6 directions possibles, laquelle te parle ?".

Source

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

design-consultation

design-consultation

> *Catalogue généré le 2026-05-11*

En une phrase

Pose les bases du design d'un nouveau projet — palette, typographie, espacement, motion — et te génère un fichier DESIGN.md qui devient la référence visuelle de tout le projet.

Quand l'utiliser

- Quand tu démarres un projet de zéro et tu n'as pas encore de direction artistique.
- Quand tu veux poser une "bible visuelle" claire avant de commencer à coder les écrans.
- Pour avoir une palette de couleurs et un système typographique cohérents partout sur le site.
- Quand tu veux une page de prévisualisation qui montre ta palette + tes polices côte à côte.

Comment l'invoquer

- **Slash command** : `/design-consultation`
- **Phrases déclencheurs (texte)** : "design system" / "brand guidelines" / "create DESIGN.md" / "create a brand" / "design from scratch"
- **Auto-invocation** : Oui — Claude suggère ce skill quand tu attaques l'UI d'un nouveau projet qui n'a encore ni système de design ni DESIGN.md.

Description détaillée

Ce skill fonctionne comme une consultation avec un designer. Il commence par poser des questions sur ton produit — à qui il s'adresse, quelle ambiance tu veux donner, quels concurrents tu aimes, lesquels tu détestes. Puis il fait sa recherche : il regarde l'état de l'art dans ton domaine et identifie les directions visuelles qui pourraient coller.

Ensuite, il te propose un système de design complet : l'esthétique générale (minimaliste, éditorial, ludique…), une famille typographique avec les tailles, une palette de couleurs (primaires, secondaires, accents, états d'erreur/succès), un système d'espacement, des règles de layout, et même les motion tokens si tu prévois des animations.

Le résultat final, c'est un fichier `DESIGN.md` à la racine de ton projet, qui devient la source unique de vérité pour toutes les décisions visuelles à venir. En bonus, il génère des pages de prévisualisation HTML qui affichent ta palette et tes polices en condition réelle, pour valider que ça rend bien.

Si ton site existe déjà et que tu veux DÉDUIRE le système de design à partir de l'existant (au lieu d'en créer un de zéro), utilise plutôt `/plan-design-review` ou `/design-review`.

Source

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

Chapitre B — Browser, QA & Dogfooding

9 skills regroupés sous ce thème.

setup-browser-cookies

setup-browser-cookies

> *Catalogue généré le 2026-05-11*

En une phrase

Importe les cookies de connexion de ton vrai navigateur Chrome vers le navigateur invisible utilisé par les skills de test, pour qu'il soit déjà connecté à tes comptes.

Quand l'utiliser

- Avant de tester une page qui demande d'être connecté (admin, dashboard, espace membre).
- Quand un skill de QA bute sur une page de login et ne sait pas comment y entrer.
- Pour faire dogfooder ton site comme si tu étais un utilisateur déjà authentifié.
- Quand tu veux éviter de re-taper un mot de passe à chaque session de test.

Comment l'invoquer

- **Slash command** : `/setup-browser-cookies`
- **Phrases déclencheurs (texte)** : "import cookies" / "login to the site" / "authenticate the browser" / "setup authenticated session"
- **Auto-invocation** : Sur demande explicite, ou suggéré par un autre skill quand un test bute sur un mur de connexion.

Description détaillée

Ce skill ouvre une petite interface dans ton navigateur où tu choisis quels domaines (par exemple github.com, ton dashboard, etc.) tu veux importer. Tu cliques sur le "+" à côté du domaine voulu et les cookies de session sont copiés vers le navigateur invisible que gstack utilise pour les tests.

Il détecte tout seul les navigateurs Chromium installés sur ta machine (Chrome, Comet, Brave, etc.). Tu peux aussi lui passer directement un domaine en argument si tu sais ce que tu veux importer, sans passer par l'interface graphique.

Une fois les cookies importés, ils restent disponibles pour toutes les commandes suivantes du navigateur de test. Si tu utilises déjà le mode CDP (le navigateur invisible est branché sur ton vrai navigateur), ce skill détecte le cas et te dit que ce n'est pas nécessaire — tes sessions sont déjà partagées.

Côté sécurité, l'interface n'affiche que les noms de domaines et le nombre de cookies, jamais leur contenu. Sur macOS, la première importation peut déclencher une demande du Trousseau (Keychain) — il faut cliquer sur "Toujours autoriser".

Source

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

scrape

scrape

> *Catalogue généré le 2026-05-11*

En une phrase

Récupère les données affichées sur une page web (texte, prix, liste, etc.) et te les rend en JSON propre, sans avoir à écrire de scraper toi-même.

Quand l'utiliser

- Quand tu veux extraire le contenu d'une page (article, liste de produits, prix, tableau) pour le réutiliser ailleurs.
- Pour récupérer les infos d'une page que tu n'as pas le droit ou le temps de coder une API contre.
- Pour comparer le contenu de deux URLs (par exemple ton site avant/après un déploiement).
- Quand tu poses la question "qu'est-ce qu'il y a sur cette page ?" en cherchant des données précises.

Comment l'invoquer

- **Slash command** : `/scrape`
- **Phrases déclencheurs (texte)** : "scrape" / "get data from" / "pull from" / "extract from" / "what's on" a page
- **Auto-invocation** : Oui — Claude propose ce skill quand tu décris une tâche d'extraction de données depuis le web.

Description détaillée

Ce skill est en lecture seule : il navigue vers une page, regarde le contenu, et retourne un JSON structuré. Il ne clique pas sur des boutons, ne remplit pas de formulaires, ne soumet rien — pour ces flux qui modifient l'état d'un site, tu utiliserais `/automate` à la place.

La première fois que tu lances un scraping avec une nouvelle intention (par exemple "récupère tous les titres d'articles du blog"), le skill prototype le flux en utilisant les primitives du navigateur invisible de gstack. Il analyse la page, repère les éléments à extraire, et retourne le JSON.

Les appels suivants sur la même intention sont automatiquement routés vers une "browser-skill" codifiée (un mini-programme spécialisé qu'il a généré lui-même) qui répond en environ 200 ms. Autrement dit : la première fois c'est lent (le temps de réfléchir à la structure), les fois d'après c'est très rapide.

Ce mécanisme est utile pour des tâches répétitives : tu peux par exemple scraper le statut d'une page de production toutes les heures sans payer le coût d'analyse à chaque fois.

Source

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

pair-agent

pair-agent

> *Catalogue généré le 2026-05-11*

En une phrase

Permet à un autre agent IA (Codex, Cursor, OpenClaw, etc.) de contrôler ton navigateur, dans son propre onglet, avec des permissions limitées et sécurisées.

Quand l'utiliser

- Quand tu veux faire travailler deux agents IA différents sur le même site, sans qu'ils se marchent dessus.
- Pour donner un accès lecture/écriture limité à un agent distant qui doit naviguer (par exemple pour faire du test ou du scraping).
- Quand tu collabores avec un autre agent (OpenClaw, Hermes, etc.) qui a besoin d'un vrai navigateur.
- Pour partager temporairement ta session de navigateur avec quelqu'un d'autre, sans donner tes identifiants.

Comment l'invoquer

- **Slash command** : `/pair-agent`
- **Voice triggers** : « pair agent » · « connect agent » · « share my browser » · « remote browser access »
- **Phrases déclencheurs (texte)** : "pair agent" / "connect agent" / "share browser" / "remote browser" / "let another agent use my browser" / "give browser access"

- **Auto-invocation** : Sur demande explicite uniquement.

Description détaillée

Ce skill génère une clé d'appairage et imprime les instructions exactes que l'autre agent doit suivre pour se connecter à ton navigateur. L'autre agent peut être absolument n'importe quoi qui sait faire un appel HTTP : OpenClaw, Hermes, Codex CLI, Cursor, ou même un script maison.

L'agent distant reçoit son propre onglet dans ton navigateur, avec un accès "scoped" — par défaut, lecture et écriture sur les pages, mais pas les commandes admin du navigateur. Si l'autre agent a besoin de plus, il doit le demander explicitement.

Côté sécurité, c'est costaud : le démon démarre deux écouteurs HTTP. Un local (127.0.0.1) garde toutes les commandes, et un tunnel ngrok ne laisse passer qu'une allowlist très restreinte (`/connect`, `/command` avec un token limité à 26 commandes de pilotage, et `/sidebar-chat`). Les tentatives bloquées sont loggées. Bref : tu peux partager sans flipper.

C'est utile principalement quand tu construis des pipelines multi-agents ou que tu veux qu'un agent externe vienne dépanner sur ton setup sans avoir à dupliquer toute la config de navigateur.

Source

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

canary

canary

> *Catalogue généré le 2026-05-11*

En une phrase

Surveille ton site en production après un déploiement, prend des captures d'écran à intervalles réguliers, et te prévient si quelque chose se casse.

Quand l'utiliser

- Juste après un déploiement, quand tu veux être sûr que rien n'est cassé sans rester collé à ton écran.
- Pour détecter automatiquement les régressions de performance (le site qui devient lent).
- Pour repérer les erreurs JavaScript dans la console qui sont apparues après ta mise en ligne.
- Quand tu veux surveiller une page sensible pendant quelques heures sans intervention humaine.

Comment l'invoquer

- **Slash command** : `/canary`
- **Phrases déclencheurs (texte)** : "monitor deploy" / "canary" / "post-deploy check" / "watch production" / "verify deploy" / "monitor after deploy" / "canary check"
- **Auto-invocation** : Sur demande explicite, ou suggéré juste après un déploiement effectué via `/ship` ou `/land-and-deploy`.

Description détaillée

Le terme "canary" vient des canaris qu'on emmenait dans les mines de charbon : si l'oiseau tombait, on savait qu'il y avait un problème dans l'air. Ici, le canary, c'est ce skill qui veille sur ton site en production.

Concrètement, il ouvre le démon du navigateur invisible de gstack et lance des sondages réguliers sur ton site live. À chaque passage, il prend une capture d'écran, regarde la console pour repérer les erreurs JS, mesure le temps de chargement, et compare tout ça à des références prises avant le déploiement (les "baselines").

Si quelque chose s'écarte trop de la baseline — une erreur en console qui n'était pas là, une page qui met soudainement 3 secondes au lieu de 500 ms, un élément clé qui a disparu — il t'alerte.

Tu peux le lancer juste après `/ship` ou `/land-and-deploy` pour avoir l'esprit tranquille pendant que tu fais autre chose. C'est une sorte de "monitoring léger maison" qui ne remplace pas un vrai outil de monitoring pro, mais qui rattrape déjà 80% des régressions visibles en post-deploy.

Source

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

connect-chrome

connect-chrome

> *Catalogue généré le 2026-05-11*

En une phrase

Lance une fenêtre de Chromium pilotée par l'IA, visible à l'écran, avec une extension de sidebar qui affiche en temps réel tout ce que Claude fait dans le navigateur.

Quand l'utiliser

- Quand tu veux VOIR Claude utiliser un navigateur (au lieu du mode invisible classique).
- Pour tester un site en regardant chaque clic, chaque saisie, chaque navigation.
- Quand tu veux discuter avec Claude pendant qu'il navigue, via la chat intégrée dans la sidebar.
- Pour contourner les protections anti-bot de certains sites (le mode stealth est intégré).
- Voice : quand tu dis "montre-moi le navigateur" pendant que tu lui parles.

Comment l'invoquer

- **Slash command** : `/connect-chrome`
- **Voice triggers** : « show me the browser »
- **Phrases déclencheurs (texte)** : "open gstack browser" / "launch browser" / "connect chrome" / "open chrome" / "real browser" / "launch chrome" / "side panel" / "control my browser"
- **Auto-invocation** : Sur demande explicite uniquement.

Description détaillée

Ce skill est un alias de `/open-gstack-browser` (le nom historique). Il ouvre une fenêtre Chromium classique, à l'écran, que Claude peut piloter exactement comme s'il s'agissait du navigateur invisible — sauf que là, tu vois tout en direct.

L'extension de sidebar est pré-installée à l'intérieur. Elle affiche un fil d'activité (chaque action de Claude apparaît en temps réel), un terminal PTY connecté à ta session Claude pour échanger avec lui pendant qu'il navigue, et un inspecteur CSS pour examiner les éléments visuellement.

Sous le capot, c'est le même navigateur que `/browse` utilise quand il est en mode invisible — mêmes commandes, même comportement, mais visible. Le mode "stealth" anti-bot est intégré : les sites qui détectent les navigateurs automatisés (Cloudflare, certains shops) ont beaucoup plus de mal à te repérer.

C'est l'outil idéal quand tu fais du debug visuel ou que tu veux apprendre comment Claude raisonne en regardant ses actions se dérouler à l'écran.

Source

- **Plugin** : `gstack`
- **Nom interne** : `open-gstack-browser` (le dossier `connect-chrome` est un alias)
- **Fichier** : `/home/thymon/.claude/skills/gstack/connect-chrome/SKILL.md`

browse

browse

> *Catalogue généré le 2026-05-11*

En une phrase

Donne à Claude un navigateur invisible et rapide pour ouvrir une page, cliquer, remplir des champs, prendre des captures, et tester ton site comme un utilisateur.

Quand l'utiliser

- Quand tu veux que Claude teste lui-même une fonctionnalité après l'avoir codée.
- Pour vérifier qu'un déploiement n'a rien cassé (smoke test rapide).
- Pour prendre une capture d'écran d'un bug et la joindre à un rapport.
- Pour tester comment ton site rend sur différentes tailles d'écran (responsive).
- Pour faire dogfooder un parcours utilisateur de bout en bout (login, formulaire, paiement…).

Comment l'invoquer

- **Slash command** : `/browse`
- **Phrases déclencheurs (texte)** : "open in browser" / "test the site" / "take a screenshot" / "dogfood this" / "browse a page" / "headless browser"
- **Auto-invocation** : Sur demande explicite, et utilisé en interne par presque tous les autres skills QA et design.

Description détaillée

C'est la brique de base sur laquelle s'appuient tous les autres skills de QA et de design de gstack. C'est un navigateur Chromium "headless" (sans interface graphique visible) basé sur Playwright, qui répond en environ 100 ms par commande.

Il sait tout faire : naviguer vers une URL, cliquer sur un élément, remplir un formulaire, gérer les dialogues (alert/confirm), faire des assertions ("ce bouton existe", "ce texte est présent"), comparer l'état avant/après une action, prendre des captures d'écran annotées, tester en mode mobile ou desktop, et gérer les uploads de fichiers.

Pour Thymon, l'intérêt n'est pas tant de le piloter à la main que de comprendre qu'il existe : quand un skill comme `/qa`, `/design-review` ou `/canary` parle de "tester ton site", c'est `browse` qui fait le boulot en coulisse.

Tu peux aussi le lancer directement quand tu as besoin d'une action ponctuelle — par exemple "prends une capture de ma page d'accueil sur iPhone 12". Pour la version VISIBLE (où tu vois la fenêtre), utilise `/open-gstack-browser` ou `/connect-chrome`.

Source

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

open-gstack-browser

open-gstack-browser

> *Catalogue généré le 2026-05-11*

En une phrase

Lance la fenêtre visible du navigateur Chromium piloté par Claude, avec l'extension sidebar baked-in pour voir tout ce qu'il fait en direct.

Quand l'utiliser

- Quand tu veux voir Claude utiliser un navigateur à l'écran (au lieu du mode invisible).
- Pour suivre en direct un parcours de test, clic par clic.
- Quand tu veux discuter avec Claude pendant qu'il navigue, via la chat de la sidebar.
- Pour contourner des protections anti-bot — le mode stealth est intégré dès l'ouverture.
- Voice : quand tu dis "montre-moi le navigateur" pendant une session vocale.

Comment l'invoquer

- **Slash command** : `/open-gstack-browser`
- **Voice triggers** : « show me the browser »
- **Phrases déclencheurs (texte)** : "open gstack browser" / "launch chromium" / "open chrome" / "real browser" / "launch browser" / "side panel" / "control my browser"
- **Auto-invocation** : Sur demande explicite uniquement.

Description détaillée

C'est le skill canonique pour ouvrir le navigateur de gstack en mode visible. `/connect-chrome` est juste un alias plus court qui pointe vers ce même skill — le nom officiel et le dossier de code, c'est `open-gstack-browser`.

Quand tu le lances, une vraie fenêtre Chromium s'ouvre. Claude la pilote exactement comme il pilote le mode invisible (`/browse`), mais cette fois tu vois tout. À côté de la zone web, une sidebar te montre : un fil d'activité avec chaque action exécutée, un terminal PTY interactif qui te connecte à ta session Claude pour lui parler sans quitter le navigateur, et un inspecteur CSS pour examiner les éléments en cours.

Le mode "anti-bot stealth" est activé par défaut. Concrètement, ça veut dire que les sites qui détectent les navigateurs automatisés (Cloudflare, certains marketplaces, certains paywalls) ont beaucoup plus de mal à te bloquer.

C'est l'outil parfait pour le debug visuel, l'apprentissage (regarder comment Claude raisonne), et les sites qui résistent au mode headless classique.

Source

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

qa-only

qa-only

> *Catalogue généré le 2026-05-11*

En une phrase

Teste ton site et te rend un rapport de bugs détaillé avec captures et étapes pour reproduire — mais ne touche jamais au code, jamais.

Quand l'utiliser

- Quand tu veux UN AVIS sur ce qui ne va pas, sans que Claude commence à "réparer" des trucs tout seul.
- Pour générer un rapport de bugs propre à transmettre à un collègue ou à reprendre toi-même plus tard.
- Avant un point de revue où tu veux savoir l'état réel du site, sans pollution par des modifs.
- Quand tu veux une note de "santé" du site à un moment T pour comparer plus tard.

Comment l'invoquer

- **Slash command** : `/qa-only`
- **Voice triggers** : « bug report » · « just check for bugs »
- **Phrases déclencheurs (texte)** : "just report bugs" / "qa report only" / "test but don't fix"
- **Auto-invocation** : Oui — Claude suggère ce skill quand tu demandes un rapport de bugs sans toucher au code.

Description détaillée

C'est la version "audit pur" de `/qa`. Elle exécute exactement la même batterie de tests systématiques sur ton application web — navigation, formulaires, états vides, responsive, console errors, performance — mais avec une règle absolue : interdiction de modifier le code.

Le résultat, c'est un rapport structuré qui contient un score de santé global, la liste des problèmes classés par criticité (critical / high / medium / cosmetic), des captures d'écran pour chaque bug, et les étapes exactes pour le reproduire. Tu peux ensuite décider toi-même ce que tu veux corriger, ou refiler le rapport à quelqu'un d'autre.

Le contraste avec `/qa` est important : `/qa` lance la boucle complète "tester → corriger → vérifier" et ferme les bugs au fur et à mesure. `qa-only` s'arrête au "tester" et ne fait rien d'autre. Tu choisis selon que tu veux l'autonomie complète ou la simple constatation.

C'est aussi un bon outil pour figer l'état d'un site dans le temps : tu lances `qa-only` aujourd'hui, tu fais des modifs pendant deux semaines, tu le relances, tu compares les deux rapports.

Source

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

qa

qa

> *Catalogue généré le 2026-05-11*

En une phrase

Teste systématiquement ton site, trouve les bugs, et les corrige un par un dans le code en commit propre, jusqu'à ce que tout fonctionne.

Quand l'utiliser

- Quand tu viens de finir une feature et tu veux savoir si "ça marche vraiment".
- Pour un check complet avant de mettre en ligne — formulaires, navigation, responsive, edge cases.
- Quand tu poses la question "est-ce que ça marche ?" et tu veux une vraie réponse, pas une intuition.
- Quand tu acceptes que Claude corrige les bugs trouvés lui-même (sinon préfère `/qa-only`).

Comment l'invoquer

- **Slash command** : `/qa`
- **Voice triggers** : « quality check » · « test the app » · « run QA »
- **Phrases déclencheurs (texte)** : "qa" / "QA" / "test this site" / "find bugs" / "test and fix" / "fix what's broken"
- **Auto-invocation** : Oui — Claude suggère ce skill quand tu dis qu'une feature est prête à tester ou quand tu poses la question "est-ce que ça marche ?".

Description détaillée

C'est le couteau suisse du test : il fait à la fois l'audit ET la réparation. La boucle complète : tester systématiquement, trouver les bugs, corriger dans le code source, commiter atomiquement, et re-tester pour confirmer que c'est bien réglé. Tout ça en autonomie.

Tu choisis le niveau d'exhaustivité :

- **Quick** : seulement les bugs critiques et high (ce qui empêche le site de marcher).
- **Standard** : critical + high + medium (la couverture habituelle).
- **Exhaustive** : tout, jusqu'aux soucis cosmétiques (alignement, micro-typographie).

À la fin, tu reçois un résumé avec deux scores de santé — celui d'avant la passe et celui d'après — la preuve des corrections (captures avant/après pour chaque fix), et un verdict "prêt à shipper" ou "il reste des soucis bloquants".

À ne pas confondre avec `/qa-only`, qui fait le même audit mais s'arrête à la simple constatation sans toucher au code. Utilise `/qa` quand tu fais confiance à Claude pour corriger, et `/qa-only` quand tu veux garder le contrôle sur les modifs.

Source

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

Chapitre C — Sécurité & Audit

2 skills regroupés sous ce thème.

cso

cso

> *Catalogue généré le 2026-05-11*

En une phrase

Mode « Chief Security Officer » : Claude audite ton projet à la recherche de failles de sécurité, secrets exposés, dépendances vulnérables et risques d'attaque.

Quand l'utiliser

- Avant de déployer une application sensible (qui touche à de l'argent, des données personnelles, de l'auth)
- Quand tu veux vérifier qu'aucun mot de passe ou clé API ne traîne dans ton historique git
- Quand tu veux un audit OWASP Top 10 (les dix grandes catégories de vulnérabilités web)
- Après avoir ajouté de nouvelles dépendances, pour vérifier qu'elles ne contiennent pas de failles connues
- Périodiquement (par exemple chaque mois) pour suivre l'évolution de la sécurité de ton projet

Comment l'invoquer

- **Slash command** : `/cso` (à taper dans Claude Code)
- **Voice triggers** : « see-so » · « see so » · « security review » · « security check » · « vulnerability scan » · « run security »
- **Phrases déclencheurs (texte)** : "security audit" / "threat model" / "pentest review" / "owasp review" / "CSO review"

- **Auto-invocation** : Sur demande explicite

Description détaillée

`/cso` lance un audit de sécurité à la fois sur ton code et sur ton infrastructure. Le skill détecte d'abord ton stack technique (Node, Python, Ruby, Go, etc.) puis cartographie la surface d'attaque : endpoints publics, routes authentifiées, points d'upload de fichiers, intégrations externes, webhooks, jobs en arrière-plan.

Il enchaîne plusieurs phases. **Archéologie des secrets** : il fouille l'historique git pour repérer les clés AWS (`AKIA…`), clés OpenAI (`sk-…`), tokens GitHub, etc. qui auraient pu être commit par erreur. **Chaîne de dépendances** : il lance `npm audit` ou équivalent, repère les scripts d'install suspects dans tes dépendances de production. **Pipeline CI/CD** : il vérifie tes workflows GitHub Actions (actions non pinnées, `pull_request_target` dangereux, injection de scripts). **OWASP Top 10 + STRIDE** : injections SQL, XSS, CSRF, contrôles d'accès cassés, etc.

Deux modes existent. **Daily** est silencieux (signale seulement à 8/10 de confiance, zéro bruit). **Comprehensive** est l'audit mensuel approfondi (à 2/10, tout sort). Tu peux aussi cibler des sous-périmètres : `--infra`, `--code`, `--skills`, `--diff` (limité aux changements de ta branche). À chaque exécution, les résultats sont stockés pour suivre les tendances dans le temps.

Source

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

guard

guard

> *Catalogue généré le 2026-05-11*

En une phrase

Mode « sécurité maximale » : Claude t'avertit avant chaque commande destructrice ET t'empêche de modifier des fichiers hors d'un dossier précis.

Quand l'utiliser

- Quand tu travailles sur de la production en direct et tu veux zéro accident
- Quand tu débugges un système critique et tu veux limiter le périmètre des modifications
- Quand tu donnes la main à Claude sur un projet sensible et tu veux des garde-fous serrés
- Quand tu veux la combinaison `/careful` + `/freeze` activée en une seule commande
- Avant de toucher à n'importe quoi qui pourrait coûter cher si ça casse

Comment l'invoquer

- **Slash command** : `/guard` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "guard mode" / "full safety" / "lock it down" / "maximum safety"
- **Auto-invocation** : Sur demande explicite

Description détaillée

`/guard` active deux protections d'un coup. La première vient de `/careful` : avant d'exécuter une commande potentiellement destructrice (`rm -rf`, `DROP TABLE`, `git push --force`, `git reset --hard`, `kubectl delete`, `docker system prune`, etc.), Claude s'arrête et te demande confirmation. Tu peux toujours autoriser, mais tu ne peux pas le faire sans le voir. Certains cas évidents (`rm -rf node_modules`, `rm -rf .next`, etc.) restent autorisés sans alerte.

La seconde vient de `/freeze` : tu choisis un dossier précis (par exemple `src/auth/`) et toutes les modifications de fichiers en dehors de ce dossier sont bloquées. Quand le skill démarre, il te demande où placer cette frontière, puis enregistre le dossier dans son fichier d'état.

Les deux protections fonctionnent via des « hooks » Claude Code qui interceptent chaque appel aux outils Bash, Edit et Write avant exécution. Tu vois passer un statut « Checking for destructive commands… » ou « Checking freeze boundary… » à chaque action sensible. Pour relâcher la contrainte sur le dossier, tu lances `/unfreeze`. Pour tout désactiver, tu termines la session — les hooks sont session-scopés.

C'est le mode à activer quand tu veux que Claude soit utile mais qu'il ne puisse pas tout casser par excès de zèle.

Source

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

Chapitre D — Planning & Stratégie

6 skills regroupés sous ce thème.

plan-tune

plan-tune

> *Catalogue généré le 2026-05-11*

En une phrase

Règle la fréquence des questions que Claude te pose et inspecte ton profil de développeur (ce que tu as déclaré aimer vs ce que ton comportement révèle).

Quand l'utiliser

- Quand Claude te pose toujours la même question agaçante et que tu veux qu'il arrête
- Quand tu veux voir ton profil : tes préférences (rapide vs minutieux, scope large vs petit, etc.)
- Quand tu veux changer ta posture : « je veux être consulté à chaque décision » ou au contraire « délègue, fais au mieux »
- Quand tu veux voir l'historique des questions que Claude t'a posées dans ce projet
- Quand tu veux activer ou désactiver complètement le système de réglage de questions

Comment l'invoquer

- **Slash command** : `/plan-tune` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "tune questions" / "stop asking me that" / "too many questions" / "show my profile" / "show my vibe" / "developer profile" / "turn off question tuning"
- **Auto-invocation** : □ Oui — Claude propose ce skill quand il remarque que la même question revient ou que tu surcharges souvent ses recommandations.

Description détaillée

`/plan-tune` est un panneau de réglages conversationnel : tu parles en langage naturel, pas besoin de mémoriser des sous-commandes. Tu peux dire « montre-moi mon profil », « arrête de me demander ça », « passe-moi en mode délègue » et Claude comprend.

Le skill gère deux choses. D'un côté, un registre de toutes les questions que les autres skills posent (validation, choix de scope, routage), avec trois préférences possibles : ne jamais demander, demander à chaque fois, demander seulement pour les décisions irréversibles. De l'autre, un profil développeur en cinq dimensions : appétit de scope (petit vs « boil the ocean »), tolérance au risque, préférence de détail (réponses courtes vs explications), autonomie (consulter vs déléguer), soin de l'architecture.

Ton profil a deux pistes : celui que tu as déclaré toi-même et celui que ton comportement réel suggère. Quand il y a un écart, le skill te montre la différence (« tu te dis prudent mais tes choix sont rapides »). Cette version 1 est purement observationnelle : aucun autre skill ne modifie son comportement à partir du profil — c'est juste un miroir pour t'aider à mieux te connaître et désamorcer les questions répétitives.

Source

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

plan-ceo-review

plan-ceo-review

> *Catalogue généré le 2026-05-11*

En une phrase

Une relecture de plan en mode « PDG/fondateur » : Claude challenge ton ambition, repense le problème, et te pousse vers le produit dix étoiles.

Quand l'utiliser

- Quand tu as un plan ou un document de design et tu te demandes s'il est assez ambitieux
- Quand tu veux qu'on remette en cause les prémisses (« et si on faisait carrément l'inverse ? »)
- Quand tu hésites entre « tout faire d'un coup » et « livrer le minimum »
- Quand tu veux quelqu'un qui pense à 5-10 ans, pas seulement au prochain sprint
- Avant de coder, pour t'assurer que tu construis la bonne chose, pas juste « la bonne chose qu'on t'a demandée »

Comment l'invoquer

- **Slash command** : `/plan-ceo-review` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "think bigger" / "expand scope" / "strategy review" / "rethink this plan" / "is this ambitious enough"
- **Auto-invocation** : □ Oui — Claude propose ce skill quand tu questionnes l'ambition d'un plan ou que ton projet sent qu'il pourrait viser plus haut.

Description détaillée

`/plan-ceo-review` lit ton plan ou ton document de design et l'analyse comme le ferait un fondateur expérimenté (style YC, Stripe, Apple). Tu choisis d'abord l'une des quatre postures : EXPANSION DE SCOPE (rêver grand, ajouter ce qui rendrait le produit dix fois meilleur), EXPANSION SÉLECTIVE (tenir le scope mais piocher les meilleures idées), TENIR LE SCOPE (rigueur maximale sans bouger l'objectif), RÉDUCTION (couper jusqu'à l'essentiel).

Le skill applique seize « instincts cognitifs » de grands dirigeants : portes réversibles vs irréversibles (Bezos), focus comme soustraction (Jobs), inversion (« qu'est-ce qui nous ferait échouer ? »), paranoïa stratégique (Grove), volonté comme stratégie (Altman). Il fait un audit système de ton repo avant de commencer (historique git, TODOs, design docs existants), puis te pose des questions une par une via un format de décision structuré : titre, explication simple, recommandation, pour/contre, conséquence si tu te trompes.

Le résultat est un plan affiné avec un rapport « GSTACK REVIEW REPORT » ajouté à la fin. Aucun code n'est touché — la sortie c'est un meilleur plan, prêt à passer ensuite à `/plan-eng-review` pour la partie technique.

Source

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

autoplan

autoplan

> *Catalogue généré le 2026-05-11*

En une phrase

Lance d'un coup toute la chaîne de relectures (PDG, design, ingénierie, expérience développeur) en laissant Claude décider lui-même des choix intermédiaires.

Quand l'utiliser

- Quand tu as un plan brouillon et tu veux qu'il passe par toutes les relectures sans répondre à 15-30 questions intermédiaires
- Quand tu fais confiance à Claude pour les décisions de routine et tu veux juste arbitrer les vraies questions de goût à la fin
- Quand tu n'as pas le temps de faire `/plan-ceo-review` puis `/plan-eng-review` puis `/plan-devex-review` à la main
- Pour un plan déjà mûr où tu veux maximum de rigueur sans intervention constante
- Quand tu veux la « rolls » du planning : sortie = plan entièrement validé

Comment l'invoquer

- **Slash command** : `/autoplan` (à taper dans Claude Code)
- **Voice triggers** : « auto plan » · « automatic review »
- **Phrases déclencheurs (texte)** : "auto review" / "autoplan" / "run all reviews" / "review this plan automatically" / "make the decisions for me"

- **Auto-invocation** : □ Oui — Claude propose ce skill quand tu as un plan prêt et veux passer toutes les relectures sans 15-30 questions intermédiaires.

Description détaillée

`/autoplan` lit les fichiers de skill de `/plan-ceo-review`, `/plan-eng-review` et `/plan-devex-review` et les exécute à la suite, dans l'ordre obligatoire CEO → Design → Eng → DX. Chaque phase finit complètement avant que la suivante commence. La qualité de l'analyse reste la même — toutes les sections sont exécutées à fond.

La différence : les questions intermédiaires sont auto-décidées selon six principes. **Choisir la complétude** (livrer la version complète). **Boil the lakes** (corriger tout dans le rayon d'impact si c'est moins d'une journée). **Pragmatisme** (entre deux solutions équivalentes, la plus propre). **DRY** (refuser les doublons). **Explicite plutôt que malin** (10 lignes évidentes plutôt que 200 lignes abstraites). **Biais vers l'action** (avancer plutôt que tergiverser).

Les décisions de goût (« deux approches sont également valables ») sont mises de côté et présentées tout à la fin dans un récap d'approbation. Si Claude et Codex sont tous les deux d'accord pour challenger ta direction (par exemple fusionner deux features que tu voulais séparées), c'est traité spécialement : ta direction reste par défaut, les deux IA doivent défendre le changement.

Une seule commande, plan complètement reviewé en sortie. C'est l'équivalent d'une grosse réunion de plan compressée en une session.

Source

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

plan-eng-review

plan-eng-review

> *Catalogue généré le 2026-05-11*

En une phrase

Une relecture de plan en mode « manager d'ingénierie » : Claude verrouille ton architecture, tes flux de données, tes cas limites et ta couverture de tests avant que tu codes.

Quand l'utiliser

- Quand tu as un plan ou un design doc et tu t'apprêtes à commencer à coder
- Quand tu veux t'assurer que les diagrammes, l'architecture et les flux sont solides
- Quand tu veux qu'on traque tous les cas limites avant qu'ils explosent en production
- Après `/plan-ceo-review` (qui s'occupe du « pourquoi »), pour s'occuper du « comment »
- Avant d'attaquer un gros refacto ou un nouveau système

Comment l'invoquer

- **Slash command** : `/plan-eng-review` (à taper dans Claude Code)
- **Voice triggers** : « tech review » · « technical review » · « plan engineering review »
- **Phrases déclencheurs (texte)** : "review the architecture" / "engineering review" / "lock in the plan" / "review architecture"
- **Auto-invocation** : □ Oui — Claude propose ce skill quand tu as un plan ou design doc et tu t'apprêtes à coder, pour repérer les soucis d'architecture avant l'implémentation.

Description détaillée

`/plan-eng-review` joue le rôle d'un manager d'ingénierie expérimenté. Avant toute relecture, il fait un « scope challenge » : est-ce que du code existant résout déjà ce problème ? Quelle est la version minimale ? Si le plan touche plus de 8 fichiers ou crée plus de 2 nouveaux services, c'est un signal d'alarme qui déclenche une remise en question.

Le skill applique des principes solides : un seul changement à la fois (refacto puis comportement, jamais les deux), DRY, observabilité (logs, métriques, traces), sécurité par défaut, code « bien ingénieré » (ni fragile ni sur-conçu). Il valorise beaucoup les diagrammes ASCII pour les flux de données, machines à états, pipelines, et exige qu'ils restent à jour.

La relecture passe par quatre sections en mode interactif : Architecture, Qualité du code, Tests, Performance. Pour chaque section, jusqu'à huit problèmes principaux sont présentés un par un avec recommandation, pour/contre, et tu décides. Une règle anti-shortcut empêche Claude d'écrire tous les retours dans un fichier de plan sans te les présenter — chaque trouvaille passe par toi.

À la fin tu obtiens un plan blindé prêt à coder, avec les bonnes briques techniques, les bons tests, et les bons garde-fous. Aucun code n'est écrit pendant la session.

Source

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

plan-devex-review

plan-devex-review

> *Catalogue généré le 2026-05-11*

En une phrase

Une relecture de plan en mode « expérience développeur » : Claude vérifie que ton API, ta CLI ou ton SDK seront agréables à utiliser pour d'autres devs.

Quand l'utiliser

- Quand tu construis quelque chose que d'autres développeurs vont utiliser (API, CLI, SDK, librairie, framework, plateforme)
- Quand tu veux que ton « time to hello world » (TTHW) soit court : moins de 2 minutes pour un premier essai réussi
- Quand tu veux benchmarker ton approche contre les standards (Stripe, Vercel, Next.js, etc.)
- Quand tu écris la doc d'onboarding et tu veux qu'elle soit irréprochable
- Avant de publier publiquement un outil pour développeurs

Comment l'invoquer

- **Slash command** : `/plan-devex-review` (à taper dans Claude Code)
- **Voice triggers** : « dx review » · « developer experience review » · « devex review » · « devex audit » · « API design review » · « onboarding review »
- **Phrases déclencheurs (texte)** : "DX review" / "developer experience audit" / "devex review" / "API design review"

- **Auto-invocation** : Oui — Claude propose ce skill quand tu as un plan pour un produit destiné aux développeurs (APIs, CLIs, SDKs, librairies, plateformes, docs).

Description détaillée

`/plan-devex-review` adopte le point de vue d'un developer advocate qui a déjà onboardé sur cent outils différents. La DX (Developer Experience), c'est l'UX pour développeurs — mais avec une barre plus haute, parce que tes utilisateurs cuisinent eux-mêmes pour vivre.

Le skill propose trois postures : **DX EXPANSION** (chercher l'avantage compétitif, viser le tier Stripe/Vercel), **DX POLISH** (blinder chaque point de contact), **DX TRIAGE** (corriger seulement les manques critiques). Il évalue ton plan selon huit principes : zéro friction au démarrage, étapes incrémentales, apprendre en faisant, défauts opinionnés avec échappatoires, lutter contre l'incertitude (chaque erreur = problème + cause + correctif), montrer du code en contexte, vitesse comme fonctionnalité, créer des moments magiques.

Sept dimensions sont notées de 0 à 10 : Utilisable, Crédible, Trouvable, Utile, Précieux, Accessible, Désirable. Pour chaque note, le skill explique ce qu'il faudrait pour atteindre 10. Il chronométre aussi le TTHW (Time to Hello World) avec des tiers clairs : champion (<2min), compétitif (2-5min), à améliorer (5-10min), drapeau rouge (>10min).

Le skill explore d'abord les personas de développeurs cibles, écrit un récit d'empathie (« voilà ce que vit Sarah en arrivant »), benchmarke contre les leaders du domaine, puis te présente les améliorations à apporter. La sortie est un plan blindé pour produire une expérience que les développeurs ont envie de partager.

Source

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

office-hours

office-hours

> *Catalogue généré le 2026-05-11*

En une phrase

Une séance type « bureau YC » : Claude joue le rôle d'un partenaire qui te pose les bonnes questions avant que tu construises quoi que ce soit.

Quand l'utiliser

- Quand tu as une idée et tu te demandes si ça vaut le coup de la construire
- Quand tu veux brainstormer un projet, un side-project, une feature, un hackathon
- Avant de lancer `/plan-ceo-review` ou `/plan-eng-review`, pour avoir un design doc solide en entrée
- Quand tu n'arrives pas à articuler clairement le problème que tu veux résoudre
- Quand tu te lances dans quelque chose de nouveau et tu veux poser les bases proprement

Comment l'invoquer

- **Slash command** : `/office-hours` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "brainstorm this" / "I have an idea" / "help me think through this" / "office hours" / "is this worth building"
- **Auto-invocation** : □ Oui — Claude invoque ce skill automatiquement quand tu décris une nouvelle idée de produit, demandes si ça vaut le coup de la construire, ou explores un concept avant d'écrire le moindre code.

Description détaillée

`/office-hours` est une session de brainstorming structurée. Avant tout, le skill te demande ton objectif : startup, intrapreneuriat, hackathon, open source, apprentissage, ou juste pour le fun. Selon ta réponse, il bascule dans l'un des deux modes.

Mode startup applique six questions de diagnostic YC : la demande est-elle réelle (pas juste de l'intérêt poli) ? Quel est le concurrent réel (souvent un tableur + Slack bricolés) ? Quelle spécificité désespérante peux-tu nommer (un vrai client, un vrai cas) ? Quel est le wedge le plus étroit qui peut générer du revenu cette semaine ? As-tu observé directement des utilisateurs ? Le produit tient-il dans 5-10 ans ? Le ton est exigeant : la spécificité est la seule monnaie, l'intérêt n'est pas la demande, les mots de l'utilisateur battent le pitch du fondateur.

Mode builder est plus chaleureux : design thinking, exploration, alternatives, brainstorming créatif pour les projets perso ou d'apprentissage.

Une règle stricte : aucune ligne de code n'est écrite, aucun autre skill d'implémentation n'est invoqué. La sortie est uniquement un design doc structuré (problème, contraintes, approche choisie, alternatives explorées) sauvegardé dans `~/.gstack/projects/<projet>/`. Ce document devient ensuite l'entrée des skills suivants (`/plan-ceo-review`, `/plan-eng-review`).

Source

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

Chapitre E — Debug, Investigation & Code Review

6 skills regroupés sous ce thème.

review

review

> *Catalogue généré le 2026-05-11*

En une phrase

Une relecture de code automatique avant de fusionner ta branche : Claude lit ta « diff » (les changements) et te signale tout ce qui pourrait poser problème.

Quand l'utiliser

- Quand tu viens de finir une fonctionnalité et que tu veux un avis avant de la pousser en production
- Quand tu veux vérifier qu'aucun changement parasite ne s'est glissé dans ta branche (« scope drift »)
- Quand tu veux savoir si ta branche couvre bien ce qui était prévu dans ton plan ou ton ticket
- Avant de demander à Claude de lancer `/ship`, pour repérer les soucis tôt
- Quand un PR est ouvert et que tu veux une seconde lecture critique

Comment l'invoquer

- **Slash command** : `/review` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "review this pr" / "code review" / "check my diff" / "pre-landing review"
- **Auto-invocation** : □ Oui — Claude propose ce skill proactivement quand tu t'apprêtes à fusionner ou livrer du code.

Description détaillée

`/review` est une relecture sérieuse de tout ce que ta branche a changé par rapport à la branche principale (`main` ou `master`). Claude détecte d'abord la branche cible (GitHub, GitLab ou en git pur), puis lance plusieurs vérifications. Il compare le contenu de ta « diff » à ce qui était demandé dans `TODOS.md`, dans la description du PR ou dans tes messages de commit — pour voir si tu as fait trop, trop peu, ou exactement ce qu'il fallait.

Le skill cherche les problèmes structurels que les tests automatiques ne voient pas : risques de sécurité SQL, frontières de confiance avec les LLM, effets de bord conditionnels, code dupliqué, manques de tests, etc. Quand un fichier de plan existe (par exemple un design issu de `/office-hours`), Claude vérifie aussi si chaque élément du plan a bien été livré, ou s'il manque quelque chose.

À la fin tu reçois un rapport clair listant les points qui doivent être corrigés avant la fusion, ceux qui peuvent attendre, et ceux qui sont déjà bons. C'est l'étape « passage à l'inspection » avant de livrer.

Source

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

investigate

investigate

> *Catalogue généré le 2026-05-11*

En une phrase

Une méthode rigoureuse pour débugger en quatre phases : Claude refuse de réparer un bug sans avoir d'abord trouvé sa cause profonde.

Quand l'utiliser

- Quand un truc « marchait hier et ne marche plus » et que tu ne comprends pas pourquoi
- Quand tu vois une erreur 500, un message d'erreur ou un comportement inattendu
- Quand tu as déjà essayé deux ou trois réparations rapides et que ça ne tient pas
- Quand le bug semble revenir à chaque fois, comme un jeu de taupes
- Avant de fermer un ticket de bug, pour t'assurer que la vraie cause a été traitée

Comment l'invoquer

- **Slash command** : `/investigate` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "debug this" / "fix this bug" / "why is this broken" / "investigate this error" / "root cause analysis"
- **Auto-invocation** : □ Oui — Claude invoque ce skill automatiquement quand tu signales une erreur, une stack trace, un comportement bizarre, ou un dysfonctionnement.

Description détaillée

`/investigate` applique une règle stricte appelée « Iron Law » : pas de correctif sans cause profonde identifiée. C'est la différence entre éteindre la flamme et trouver la fuite de gaz. Claude suit quatre phases : enquête (lire les symptômes, les logs, les changements récents), analyse (matcher avec des patterns connus comme race condition, cache obsolète, null propagé), hypothèse (vérifier l'idée avant d'écrire du code), implémentation (le correctif minimal + un test de régression).

Le skill verrouille aussi automatiquement le périmètre de modification au dossier concerné par le bug, pour t'éviter de changer du code non lié. Il vérifie les enseignements (« learnings ») accumulés par Claude sur tes projets précédents — si tu as déjà rencontré un bug similaire, il s'en sert.

Règle des trois échecs : si trois hypothèses successives sont fausses, Claude s'arrête et te demande s'il faut continuer, escalader, ou ajouter du logging et observer. À la fin tu obtiens un rapport structuré (symptôme, cause, correctif, preuve que ça marche, test de non-régression). C'est plus lent que « répare-moi ça vite » mais ça évite les bugs qui reviennent.

Source

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

codex

codex

> *Catalogue généré le 2026-05-11*

En une phrase

Demande un second avis à une IA concurrente (OpenAI Codex) — version « développeur 200 QI direct et sans filtre » qui challenge ton code ou tes idées.

Quand l'utiliser

- Quand Claude a fini un truc important et tu veux qu'une autre IA le critique
- Quand tu veux qu'une seconde IA essaie volontairement de casser ton code (mode adversarial)
- Quand tu hésites sur une décision technique et tu veux entendre un autre point de vue
- Quand tu suspectes que Claude est trop d'accord avec toi et tu veux quelqu'un de plus tranchant
- Pour les décisions importantes où l'accord entre deux IA renforce ta confiance

Comment l'invoquer

- **Slash command** : `/codex` (à taper dans Claude Code)
- **Voice triggers** : « code x » · « code ex » · « get another opinion »
- **Phrases déclencheurs (texte)** : "codex review" / "codex challenge" / "ask codex" / "second opinion" / "consult codex"
- **Auto-invocation** : Sur demande explicite (tu dois invoquer toi-même)

Description détaillée

`/codex` est un pont vers la CLI d'OpenAI Codex, présenté comme « le développeur 200 QI un peu autiste » : direct, terse, techniquement précis, qui challenge les hypothèses. Le skill fonctionne en trois modes. **Review** lance une relecture de code indépendante sur ta « diff » actuelle, avec un verdict pass/fail. **Challenge** met Codex en mode adversarial : il essaie activement de casser ton code, de trouver les cas limites, les bugs subtils. **Consult** te laisse poser n'importe quelle question avec continuité de session pour les suivis.

Si tu tapes juste `/codex` sans argument, Claude détecte ce qui est pertinent : s'il y a une diff il propose review ou challenge, s'il y a un fichier de plan il propose de le relire, sinon il te demande ce que tu veux explorer. Tu peux aussi forcer un niveau de raisonnement plus profond avec `--xhigh`.

La sortie est présentée fidèlement, pas résumée — c'est l'avis brut de Codex que tu vois. C'est utile quand l'accord entre Claude et Codex donne plus de confiance, et quand leur désaccord signale une décision où ton jugement humain est important. Tu dois avoir installé `codex` (`npm install -g @openai/codex`) et être authentifié.

Source

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

health

health

> *Catalogue généré le 2026-05-11*

En une phrase

Un tableau de bord de la santé de ton code : Claude lance tous les outils de qualité du projet et te donne une note globale sur 10.

Quand l'utiliser

- Quand tu veux savoir en deux minutes si ton projet est en bon état
- Avant une grosse session de refacto, pour avoir un point de référence
- Après une grosse session de refacto, pour vérifier que rien ne s'est dégradé
- Quand tu reprends un projet après plusieurs semaines et tu veux faire le point
- Pour suivre dans le temps l'évolution de la qualité (l'historique est gardé)

Comment l'invoquer

- **Slash command** : `/health` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "health check" / "code quality" / "how healthy is the codebase" / "run all checks" / "quality score"
- **Auto-invocation** : Sur demande explicite

Description détaillée

`/health` détecte automatiquement les outils de qualité installés dans ton projet : vérificateur de types (TypeScript, ruff, pylint), linter (Biome, ESLint, RuboCop), runner de tests (vitest, jest, pytest, cargo test, go test), détecteur de code mort (knip), linter shell (shellcheck), et état de gbrain si tu l'utilises. La première fois, il te propose de sauvegarder cette configuration dans `CLAUDE.md` pour ne pas avoir à redétecter ensuite.

Le skill lance chaque outil indépendamment, capture la sortie, le code de retour et la durée. Chaque catégorie reçoit une note de 0 à 10 selon un barème clair (typecheck propre = 10, plus de 50 erreurs = 0, etc.). Une formule pondérée calcule ensuite une note composite : les tests pèsent 28%, le typecheck 22%, le lint 18%, le code mort 13%, le lint shell 9%, gbrain 10%.

Tu obtiens un tableau de bord lisible avec le score, le statut (CLEAN / WARNING / NEEDS WORK / CRITICAL), la durée et les détails des problèmes principaux. Chaque exécution est archivée dans `~/.gstack/projects/<projet>/health-history.jsonl` pour que Claude puisse te montrer les tendances et recommander où concentrer tes efforts.

Source

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

careful

careful

> *Catalogue généré le 2026-05-11*

En une phrase

Mode « prudence » : Claude t'avertit avant chaque commande dangereuse (`rm -rf`, `DROP TABLE`, `git push --force`, etc.) pour que tu puisses confirmer ou annuler.

Quand l'utiliser

- Quand tu travailles sur de la production ou un serveur partagé
- Quand tu débugges un système en direct et tu ne veux pas effacer un fichier important par mégarde
- Quand tu vas exécuter des migrations de base de données et tu veux un filet de sécurité
- Quand tu n'es pas sûr de toi et tu préfères que Claude te demande deux fois plutôt qu'une
- Pour activer juste les avertissements (sans le verrouillage de dossier de `/guard`)

Comment l'invoquer

- **Slash command** : `/careful` (à taper dans Claude Code)
- **Phrases déclencheurs (texte)** : "be careful" / "safety mode" / "prod mode" / "careful mode" / "warn before destructive"
- **Auto-invocation** : Sur demande explicite

Description détaillée

`/careful` installe un garde-fou qui intercepte chaque commande Bash que Claude veut exécuter, avant qu'elle ne tourne. Si la commande correspond à un motif dangereux, Claude s'arrête et te demande confirmation avec un message explicite. Tu peux toujours autoriser, mais tu ne peux pas le faire à ton insu.

Voici les motifs qui déclenchent un avertissement : suppression récursive (`rm -rf`, `rm -r`, `rm --recursive`), suppression de table ou de base SQL (`DROP TABLE`, `DROP DATABASE`, `TRUNCATE`), réécriture d'historique git (`git push --force`, `git push -f`, `git reset --hard`), perte de modifications non commitées (`git checkout .`, `git restore .`), suppressions Kubernetes (`kubectl delete`), nettoyage Docker (`docker rm -f`, `docker system prune`).

Quelques cas évidents et fréquents sont autorisés sans alerte parce qu'ils sont sûrs : `rm -rf node_modules`, `.next`, `dist`, `__pycache__`, `.cache`, `build`, `.turbo`, `coverage`. Pas la peine de t'embêter pour vider un dossier de build.

Techniquement, le skill fonctionne via un « hook » PreToolUse de Claude Code qui lit la commande, la compare aux motifs ci-dessus, et renvoie une demande de permission si nécessaire. Le hook est actif tant que la session dure — pour le désactiver, tu termines la conversation ou tu en commences une nouvelle.

Source

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

devex-review

devex-review

> *Catalogue généré le 2026-05-11*

En une phrase

Audit en direct de l'expérience développeur : Claude teste vraiment ton produit (CLI, API, docs) au lieu de juste lire le plan — il chronomètre, clique, et fait des captures.

Quand l'utiliser

- Après avoir livré une feature destinée aux développeurs, pour mesurer la réalité plutôt que la théorie
- Quand tu veux savoir combien de temps prend vraiment ton « hello world »
- Quand tu veux vérifier si les messages d'erreur sont compréhensibles, avec captures à l'appui
- Pour comparer ce que `/plan-devex-review` avait prédit (« 3 minutes pour démarrer ») avec la réalité (« 8 minutes en vrai »)
- Avant de publier ou promouvoir un outil pour développeurs

Comment l'invoquer

- **Slash command** : `/devex-review` (à taper dans Claude Code)
- **Voice triggers** : « dx audit » · « test the developer experience » · « try the onboarding » · « developer experience test »
- **Phrases déclencheurs (texte)** : "test the DX" / "DX audit" / "developer experience test" / "try the onboarding" / "live dx audit"

- **Auto-invocation** : □ Oui — Claude propose ce skill après avoir livré une feature destinée aux développeurs.

Description détaillée

`/devex-review` est la version « test sur le terrain » de `/plan-devex-review`. Au lieu de relire un plan, Claude utilise l'outil `browse` (un navigateur sans interface) pour ouvrir vraiment ta documentation, suivre le tutoriel d'installation, essayer le premier exemple, déclencher volontairement des erreurs pour voir ce qu'il se passe. Il prend des captures d'écran et chronomètre chaque étape.

Le skill applique les mêmes huit principes DX que la version planning (zéro friction au démarrage, étapes incrémentales, défauts opinionnés avec échappatoires, etc.) et les mêmes sept dimensions de scoring (Utilisable, Crédible, Trouvable, Utile, Précieux, Accessible, Désirable). Mais cette fois les notes sont basées sur l'observation directe, pas sur l'analyse du plan.

`browse` peut tester les surfaces accessibles via le web : pages de docs, playgrounds d'API, dashboards web, flux d'inscription, tutoriels interactifs, pages d'erreur. Il ne peut pas tester ce qui demande un terminal local, une vraie adresse mail, ou des credentials réels — pour ces parties, Claude te dit clairement ce qui reste à vérifier à la main.

Le rapport final compare aussi tes scores réels avec ceux prédits par `/plan-devex-review` si tu l'avais lancé avant. C'est le « boomerang » qui ramène les promesses du plan face à la réalité du produit livré.

Source

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

Chapitre F — Ship, Deploy & Lifecycle

12 skills regroupés sous ce thème.

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`

Chapitre G — Performance & Benchmark

2 skills regroupés sous ce thème.

benchmark-models

benchmark-models

> *Catalogue généré le 2026-05-11*

En une phrase

Compare plusieurs modèles d'IA (Claude, GPT, Gemini) sur la même tâche pour voir lequel est le plus rapide, le moins cher et le plus performant, avec des chiffres au lieu d'impressions.

Quand l'utiliser

- Tu veux savoir quel modèle utiliser pour un type de tâche précis (Claude vs GPT vs Gemini).
- Tu suspectes qu'un de tes skills gstack tournerait mieux avec un autre modèle.
- Tu veux mesurer l'impact d'une nouvelle version de modèle sur tes coûts et tes temps de réponse.
- Tu hésites entre plusieurs modèles pour une nouvelle automatisation et tu veux des données.
- Tu veux établir une référence pour détecter les régressions de qualité quand les modèles évoluent.

Comment l'invoquer

- **Slash command** : `/benchmark-models`
- **Voice triggers** : « compare models » · « model shootout » · « which model is best »
- **Phrases déclencheurs (texte)** : "benchmark models", "compare models", "which model is best for X", "cross-model comparison", "model shootout"

- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `benchmark-models` est un comparatif côte à côte. Tu lui donnes un prompt (ou un skill gstack), il l'envoie au même moment à Claude (via Claude Code), GPT (via le CLI Codex d'OpenAI), et Gemini, puis te ressort un tableau comparatif : temps de réponse, nombre de tokens consommés, coût, et optionnellement une note de qualité décidée par un juge LLM (un autre modèle qui note les sorties sur 10).

À ne pas confondre avec `/benchmark` qui mesure la performance d'une page web (Core Web Vitals). Celui-ci compare des modèles d'IA. Le workflow est interactif : Claude te demande d'abord quel prompt utiliser (un skill gstack, un prompt en ligne, ou un fichier sur disque), puis quels providers inclure. Une commande "dry-run" préalable te montre lesquels sont authentifiés sur ta machine — pas de surprise au moment de payer les appels API.

Le juge qualité ajoute environ 0,05 $ par run. Tu peux le désactiver si tu veux juste comparer vitesse et coût. À la fin du test, Claude résume : le plus rapide, le moins cher, le plus qualitatif (si le juge a tourné), et te propose de sauvegarder les résultats en JSON dans `~/.gstack/benchmarks/` pour pouvoir comparer plus tard. Chaque ligne de tableau montre le coût réel — tu sais ce que tu dépenses avant le prochain run.

Source

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

benchmark

benchmark

> *Catalogue généré le 2026-05-11*

En une phrase

Mesure les performances d'une page web (temps de chargement, Core Web Vitals, taille des fichiers) pour détecter quand ta nouvelle PR fait ramer le site sans que personne s'en aperçoive.

Quand l'utiliser

- Tu veux établir une mesure de référence (baseline) avant de modifier quelque chose qui peut impacter la perf.
- Tu veux comparer la perf actuelle d'une page à sa baseline et voir les régressions.
- Tu veux savoir quelles sont les ressources les plus lentes à charger sur une page.
- Tu veux vérifier que ton bundle JavaScript ne dépasse pas un budget perf (par exemple 500 KB).
- Tu veux suivre les tendances perf sur plusieurs semaines pour repérer les dérives lentes.

Comment l'invoquer

- **Slash command** : `/benchmark <url>` (options : `--baseline`, `--quick`, `--pages <liste>`, `--diff`, `--trend`)
- **Voice triggers** : « speed test » · « check performance »
- **Phrases déclencheurs (texte)** : "performance", "benchmark", "page speed", "lighthouse", "web vitals", "bundle size", "load time"
- **Auto-invocation** : Sur demande explicite.

Description détaillée

Le skill `benchmark` joue le rôle d'un ingénieur perf qui sait que les sites ne deviennent pas lents d'un seul coup : ils meurent à petit feu, 50 ms ici, 20 KB là, et un jour la page met 8 secondes à charger et personne ne sait quand ça a commencé. Sa mission : mesurer, enregistrer une baseline, comparer, alerter dès qu'une régression apparaît.

Il s'appuie sur le démon de navigateur de gstack (browse) et utilise les APIs Performance du navigateur pour collecter des mesures précises : TTFB (temps avant le premier octet), FCP (First Contentful Paint), LCP (Largest Contentful Paint), DOM Interactive, DOM Complete, Full Load, plus l'analyse de chaque ressource (taille, durée, type). Il distingue les bundles JS et CSS, compte les requêtes, identifie les ressources tierces lentes (analytics, fonts), et applique des seuils standards : plus de 50 % d'augmentation ou plus de 500 ms en absolu = REGRESSION, plus de 20 % = WARNING.

Quatre modes principaux. `--baseline` capture la mesure de référence (à lancer avant tes changements). Mode comparaison (par défaut) compare la perf actuelle à la baseline. `--quick` fait juste un check rapide sans baseline. `--trend` affiche l'historique des mesures pour repérer les tendances (par exemple : "LCP a doublé en 8 jours"). `--diff` ne benchmark que les pages affectées par ta branche actuelle. À la fin, il te donne aussi un check de budget perf (FCP < 1,8 s, LCP < 2,5 s, total JS < 500 KB, total CSS < 100 KB, etc.) avec une note A/B/C, et recommande des optimisations concrètes (code splitting, lazy loading, etc.).

Source

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

Chapitre H — Outils & Méta

4 skills regroupés sous ce thème.

make-pdf

make-pdf

> *Catalogue généré le 2026-05-11*

En une phrase

Transforme n'importe quel fichier markdown en PDF de qualité publication, avec marges propres, numéros de page, table des matières cliquable et même un filigrane DRAFT si besoin.

Quand l'utiliser

- Tu veux convertir une note, un essai, une lettre ou un rapport en PDF présentable.
- Tu veux partager un document avec quelqu'un qui ne lit pas de markdown.
- Tu prépares un brouillon et tu veux un filigrane "DRAFT" en diagonale pour bien signaler que ce n'est pas final.
- Tu produis une vraie publication (essai, mémoire) avec page de garde, table des matières et coupures de chapitre automatiques.
- Tu veux itérer rapidement sur le rendu visuel d'un document long.

Comment l'invoquer

- **Slash command** : `/make-pdf`
- **Voice triggers** : « make this a pdf » · « make it a pdf » · « export to pdf » · « turn this into a pdf » · « turn this markdown into a pdf » · « generate a pdf » · « make a pdf from » · « pdf this markdown »
- **Phrases déclencheurs (texte)** : "make a PDF", "export to PDF", "turn this markdown into a PDF", "generate a document"

- **Auto-invocation** : □ Oui — dès que tu as un fichier `.md` ouvert et que tu demandes un PDF, Claude propose ce skill.

Description détaillée

Le skill `make-pdf` repose sur un petit binaire compilé qui produit des PDF avec une mise en page soignée : marges de 1 pouce, Helvetica partout, guillemets typographiques, tirets cadratins corrects, et un texte qui se copie-colle proprement (pas de "S a i l i n g" à la lettre près comme avec certains outils). C'est l'équivalent d'avoir un mini-Faber & Faber pour tes notes markdown.

Trois modes principaux. Le mode rapide (`generate fichier.md`) sort un PDF propre avec en-tête, numéro de page et mention CONFIDENTIAL en pied. Le mode publication (`--cover --toc`) ajoute une page de garde avec ton nom et la date, une table des matières cliquable, et coupe automatiquement chaque chapitre H1 sur une nouvelle page. Le mode brouillon (`--watermark DRAFT`) ajoute un filigrane diagonal sur chaque page pour bien signaler que ce n'est pas final.

Il y a aussi un mode prévisualisation (`preview fichier.md`) qui ouvre le rendu HTML dans ton navigateur, parfait pour itérer sans repasser par le PDF à chaque modification. Les options principales couvrent la taille de page (Letter, A4, Legal), les marges (en pouces, points, centimètres ou millimètres), le titre, l'auteur, et la possibilité de cacher le pied "CONFIDENTIAL". Sur Linux, il faut installer `fonts-liberation` pour que les polices s'affichent correctement.

Source

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

learn

learn

> *Catalogue généré le 2026-05-11*

En une phrase

Consulte, recherche et nettoie les "apprentissages" que gstack a accumulés sur ton projet au fil des sessions (patterns, pièges, préférences, choix d'archi).

Quand l'utiliser

- Tu te demandes "tiens, on n'avait pas déjà résolu ce bug ?" et tu veux fouiller la mémoire du projet.
- Tu veux voir ce que gstack a appris sur ton projet jusqu'ici.
- Tu veux nettoyer des notes obsolètes (fichiers supprimés, informations contradictoires).
- Tu veux exporter les apprentissages dans `CLAUDE.md` pour que ce soit toujours pris en compte.
- Tu veux ajouter manuellement une règle ou une préférence à mémoriser.

Comment l'invoquer

- **Slash command** : `/learn` (sous-commandes : `search`, `prune`, `export`, `stats`, `add`)
- **Phrases déclencheurs (texte)** : "show learnings", "what have we learned", "manage project learnings", "didn't we fix this before?"
- **Auto-invocation** : □ Oui — quand tu demandes "est-ce qu'on n'a pas déjà vu ça ?" ou que tu cherches un pattern passé.

Description détaillée

Le skill `learn` est ton wiki personnel pour chaque projet. Pendant que tu utilises gstack (avec `/review`, `/ship`, `/investigate`, etc.), Claude enregistre automatiquement les choses qu'il observe : un pattern qui revient, un piège à éviter, une préférence que tu exprimes, un choix d'architecture. Tout ça s'accumule dans un fichier `learnings.jsonl` propre à ton projet.

Le skill propose six commandes pour gérer cette mémoire. `/learn` tout court te montre les 20 dernières entrées groupées par type. `/learn search <terme>` cherche dans la mémoire. `/learn prune` détecte les entrées obsolètes (fichiers référencés qui n'existent plus, contradictions entre deux notes) et te propose de les supprimer ou les corriger. `/learn export` formate les apprentissages en markdown propre que tu peux coller dans `CLAUDE.md` ou un fichier de doc. `/learn stats` te donne un récap par type et par source. `/learn add` te permet d'ajouter manuellement une règle.

Chaque entrée a un type (`pattern`, `pitfall`, `preference`, `architecture`, `tool`), une clé courte, une description en une phrase, et un score de confiance de 1 à 10. C'est la couche de mémoire long terme de gstack : ce qui survit aux sessions et aux compactions de conversation. Important : ce skill ne modifie jamais ton code, il gère uniquement les apprentissages.

Source

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

skillify

skillify

> *Catalogue généré le 2026-05-11*

En une phrase

Transforme le dernier scraping réussi (via `/scrape`) en un skill permanent et automatisé, pour que la même extraction reparte en 200 millisecondes au lieu de tout réapprendre.

Quand l'utiliser

- Tu viens d'utiliser `/scrape` pour extraire des données d'un site, ça marche, et tu veux le réutiliser plus tard.
- Tu veux automatiser une extraction récurrente (front page d'un site, listing, stats).
- Tu veux passer d'un scraping piloté pas-à-pas par un agent à un script déterministe rapide.
- Tu veux partager ton extraction avec d'autres projets (mode global) ou la garder pour un projet précis.

Comment l'invoquer

- **Slash command** : `/skillify`
- **Phrases déclencheurs (texte)** : "skillify", "codify this scrape", "save this scrape", "make this permanent", "codify"
- **Auto-invocation** : Sur demande explicite (typiquement après un `/scrape` réussi).

Description détaillée

Le skill `skillify` est le multiplicateur de productivité du système de scraping de gstack. La logique : `/scrape` est lent parce que Claude pilote le navigateur en direct, page par page, sélecteur par sélecteur, en essayant des options jusqu'à trouver. Une fois que ça marche, c'est dommage de tout refaire à chaque fois. `/skillify` prend le dernier scraping réussi de la conversation et le transforme en un script Playwright propre, testé, et réutilisable.

Le workflow est strict pour ne jamais livrer un skill cassé. Il commence par retrouver le dernier `/scrape` valide dans la conversation (max 10 tours en arrière). Il propose un nom de skill court, des phrases déclencheurs (3 à 5), et te demande où le ranger : niveau global (`~/.gstack/browser-skills/`, accessible à tous tes projets) ou niveau projet (uniquement ce dépôt). Il synthétise ensuite un `script.ts` (la logique d'extraction sous forme de fonction pure), un `script.test.ts` (un test qui rejoue contre une capture HTML enregistrée comme fixture), et capture cette fixture (HTML figé du site cible). Tout est écrit dans un dossier temporaire d'abord.

Le test tourne dans ce dossier temporaire. S'il passe ET que tu approuves explicitement, le skill est promu dans son emplacement définitif. Sinon, le dossier temporaire est supprimé entièrement : pas d'état "à moitié livré". Une fois en place, les prochaines fois que tu lances `/scrape` avec une intention similaire, ce script déterministe tourne en environ 200 millisecondes — sans agent, sans navigation manuelle. Limitations à savoir : un seul URL par skill, les fixtures peuvent devenir obsolètes si le site change, et ce skill ne gère pas les flows qui modifient le site (formulaires, clics destructifs) — c'est le job de `/automate`.

Source

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

gstack-upgrade

gstack-upgrade

> *Catalogue généré le 2026-05-11*

En une phrase

Met à jour gstack vers sa dernière version, détecte automatiquement comment il est installé (global, local, vendored) et te montre ce qui change.

Quand l'utiliser

- gstack t'a affiché un message `UPGRADE_AVAILABLE` au début d'une session et tu veux mettre à jour.
- Tu veux passer manuellement à la dernière version de gstack sans attendre un prompt.
- Tu as activé l'auto-upgrade et tu veux comprendre ce qui s'est passé après une mise à jour automatique.
- Tu veux configurer le comportement des mises à jour (auto ou pas, snoozer une version, désactiver les checks).
- Tu veux voir le CHANGELOG des versions sorties depuis ta dernière mise à jour.

Comment l'invoquer

- **Slash command** : `/gstack-upgrade`
- **Voice triggers** : « upgrade the tools » · « update the tools » · « gee stack upgrade » · « g stack upgrade »
- **Phrases déclencheurs (texte)** : "upgrade gstack", "update gstack", "get latest version", "update gstack version", "get latest gstack"

- **Auto-invocation** : □ Oui — quand le préambule d'un skill détecte `UPGRADE_AVAILABLE`, Claude propose ce skill.

Description détaillée

Le skill `gstack-upgrade` gère le cycle de mise à jour de gstack. Au début de chaque session, gstack vérifie en arrière-plan s'il existe une version plus récente. Quand c'est le cas, il te le signale dans le préambule d'un skill. À ce moment, tu as quatre choix : oui mettre à jour maintenant, oui et active l'auto-upgrade pour ne plus jamais y penser, pas maintenant (avec un snooze qui s'allonge à chaque report : 24h, puis 48h, puis 1 semaine), ou ne plus me demander.

Le skill détecte automatiquement comment gstack est installé sur ta machine. Cinq cas possibles : install global git (le plus courant, dans `~/.claude/skills/gstack/`), install local git (cloné dans le projet courant), vendored local (copié sans git dans `.claude/skills/gstack/`), vendored global, ou pas trouvé du tout. Pour les installs git, il fait un `git fetch + reset --hard origin/main + ./setup`. Pour les vendored, il clone une nouvelle copie, l'échange avec l'ancienne (gardée en `.bak` au cas où), puis relance le setup.

Si tu as des modifications locales non commitées dans le dossier gstack (rare mais possible pour les contributeurs), le skill les stash avant l'upgrade et te prévient pour que tu puisses les récupérer après. Auto-upgrade s'active via la config (`gstack-config set auto_upgrade true`) ou la variable d'environnement `GSTACK_AUTO_UPGRADE=1`. Pour désactiver complètement les vérifications de mises à jour, tu réponds "Never ask again" — ça met `update_check: false` dans la config. À la fin de l'upgrade, le skill te montre ce qui a changé depuis ta version précédente.

Source

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

□□ Index — 01 — gstack (plugin natif)

01 — gstack (plugin natif)

> Index des skills regroupés dans ce livre. Clique sur un skill pour ouvrir sa fiche.

46 skills documentés.

Chapitre A — Design & UI

Skill	En une phrase
design-consultation	Pose les bases du design d'un nouveau projet — palette, typographie, espacement, motion — et te génère un fichier DESIGN.md qui…
design-html	Transforme un design approuvé (maquette, plan, ou simple description) en HTML/CSS de qualité production, prêt à être servi en…
design-review	Audite ton site déjà en ligne avec un œil de designer, repère les problèmes visuels et UX, puis les corrige directement dans le…
design-shotgun	Génère plusieurs variantes de design en parallèle, les affiche côte à côte sur un tableau de comparaison, et te laisse choisir…
plan-design-review	Relit ton plan de fonctionnalité avec un œil de designer, note chaque dimension visuelle sur 10 et te dit ce qu'il faudrait…

Chapitre B — Browser, QA & Dogfooding

Skill	En une phrase
browse	Donne à Claude un navigateur invisible et rapide pour ouvrir une page, cliquer, remplir des champs, prendre des captures, et…
canary	Surveille ton site en production après un déploiement, prend des captures d'écran à intervalles réguliers, et te prévient si…
connect-chrome	Lance une fenêtre de Chromium pilotée par l'IA, visible à l'écran, avec une extension de sidebar qui affiche en temps réel tout…
open-gstack-browser	Lance la fenêtre visible du navigateur Chromium piloté par Claude, avec l'extension sidebar baked-in pour voir tout ce qu'il fait…
pair-agent	Permet à un autre agent IA (Codex, Cursor, OpenClaw, etc.) de contrôler ton navigateur, dans son propre onglet, avec des…
qa	Teste systématiquement ton site, trouve les bugs, et les corrige un par un dans le code en commit propre, jusqu'à ce que tout…
qa-only	Teste ton site et te rend un rapport de bugs détaillé avec captures et étapes pour reproduire — mais ne touche jamais au code,…
scrape	Récupère les données affichées sur une page web (texte, prix, liste, etc.) et te les rend en JSON propre, sans avoir à écrire de…
setup-browser-cookies	Importe les cookies de connexion de ton vrai navigateur Chrome vers le navigateur invisible utilisé par les skills de test, pour…

Chapitre C — Sécurité & Audit

Skill	En une phrase
cso	Mode « Chief Security Officer » : Claude audite ton projet à la recherche de failles de sécurité, secrets exposés, dépendances…

Skill	En une phrase
guard	Mode « sécurité maximale » : Claude t'avertit avant chaque commande destructrice ET t'empêche de modifier des fichiers hors d'un…

Chapitre D — Planning & Stratégie

Skill	En une phrase
autoplan	Lance d'un coup toute la chaîne de relectures (PDG, design, ingénierie, expérience développeur) en laissant Claude décider…
office-hours	Une séance type « bureau YC » : Claude joue le rôle d'un partenaire qui te pose les bonnes questions avant que tu construises…
plan-ceo-review	Une relecture de plan en mode « PDG/fondateur » : Claude challenge ton ambition, repense le problème, et te pousse vers le…
plan-devex-review	Une relecture de plan en mode « expérience développeur » : Claude vérifie que ton API, ta CLI ou ton SDK seront agréables à…
plan-eng-review	Une relecture de plan en mode « manager d'ingénierie » : Claude verrouille ton architecture, tes flux de données, tes cas limites…
plan-tune	Règle la fréquence des questions que Claude te pose et inspecte ton profil de développeur (ce que tu as déclaré aimer vs ce que…

Chapitre E — Debug, Investigation & Code Review

Skill	En une phrase
careful	Mode « prudence » : Claude t'avertit avant chaque commande dangereuse (`rm -rf`, `DROP TABLE`, `git push --force`, etc.) pour que…
codex	Demande un second avis à une IA concurrente (OpenAI Codex) — version « développeur 200 QI direct et sans filtre » qui challenge…

Skill	En une phrase
devex-review	Audit en direct de l'expérience développeur : Claude teste vraiment ton produit (CLI, API, docs) au lieu de juste lire le plan —…
health	Un tableau de bord de la santé de ton code : Claude lance tous les outils de qualité du projet et te donne une note globale sur…
investigate	Une méthode rigoureuse pour débugger en quatre phases : Claude refuse de réparer un bug sans avoir d'abord trouvé sa cause…
review	Une relecture de code automatique avant de fusionner ta branche : Claude lit ta « diff » (les changements) et te signale tout ce…

Chapitre F — Ship, Deploy & Lifecycle

Skill	En une phrase
context-restore	Recharge la dernière sauvegarde de travail créée par `/context-save` pour reprendre exactement là où tu t'étais arrêté, même sur…
context-save	Sauvegarde l'état complet de ton travail en cours (décisions, fichiers modifiés, prochaines étapes) pour pouvoir reprendre plus…
document-release	Met à jour automatiquement toute la documentation de ton projet (README, ARCHITECTURE, CONTRIBUTING, CLAUDE.md, CHANGELOG) après…
freeze	Verrouille les modifications de fichiers dans un seul dossier choisi, pour éviter que Claude touche par erreur du code en dehors…
land-and-deploy	Merge ta Pull Request, attend que le déploiement passe, puis vérifie automatiquement que la production tourne bien — l'étape qui…
landing-report	Affiche un tableau de bord en lecture seule qui montre quelles versions sont déjà réservées par des PR ouvertes et quelle version…
retro	Génère une rétrospective d'ingénierie sur les 7 derniers jours (ou la fenêtre que tu choisis) en analysant tes commits, ton…

Skill	En une phrase
setup-deploy	Configure une fois pour toutes comment ton projet se déploie, pour que la commande `/land-and-deploy` sache automatiquement où…
setup-gbrain	Installe et configure gbrain (la mémoire intelligente de tes projets) sur ton Mac, puis le branche à Claude Code pour qu'il…
ship	Lance le workflow complet d'expédition de ton code : tests, revue de diff, bump de version, mise à jour du CHANGELOG, commit,…
sync-gbrain	Met à jour la mémoire de ton projet dans gbrain (un index sémantique de ton code) pour que Claude puisse faire des recherches…
unfreeze	Lève le verrouillage de dossier posé par `/freeze` pour permettre à nouveau les modifications dans tout le projet, sans avoir à…

Chapitre G — Performance & Benchmark

Skill	En une phrase
benchmark	Mesure les performances d'une page web (temps de chargement, Core Web Vitals, taille des fichiers) pour détecter quand ta…
benchmark-models	Compare plusieurs modèles d'IA (Claude, GPT, Gemini) sur la même tâche pour voir lequel est le plus rapide, le moins cher et le…

Chapitre H — Outils & Méta

Skill	En une phrase
gstack-upgrade	Met à jour gstack vers sa dernière version, détecte automatiquement comment il est installé (global, local, vendored) et te…
learn	Consulte, recherche et nettoie les "apprentissages" que gstack a accumulés sur ton projet au fil des sessions (patterns, pièges,…

Skill	En une phrase
make-pdf	Transforme n'importe quel fichier markdown en PDF de qualité publication, avec marges propres, numéros de page, table des...
skillify	Transforme le dernier scraping réussi (via `/scrape`) en un skill permanent et automatisé, pour que la même extraction reparte en...