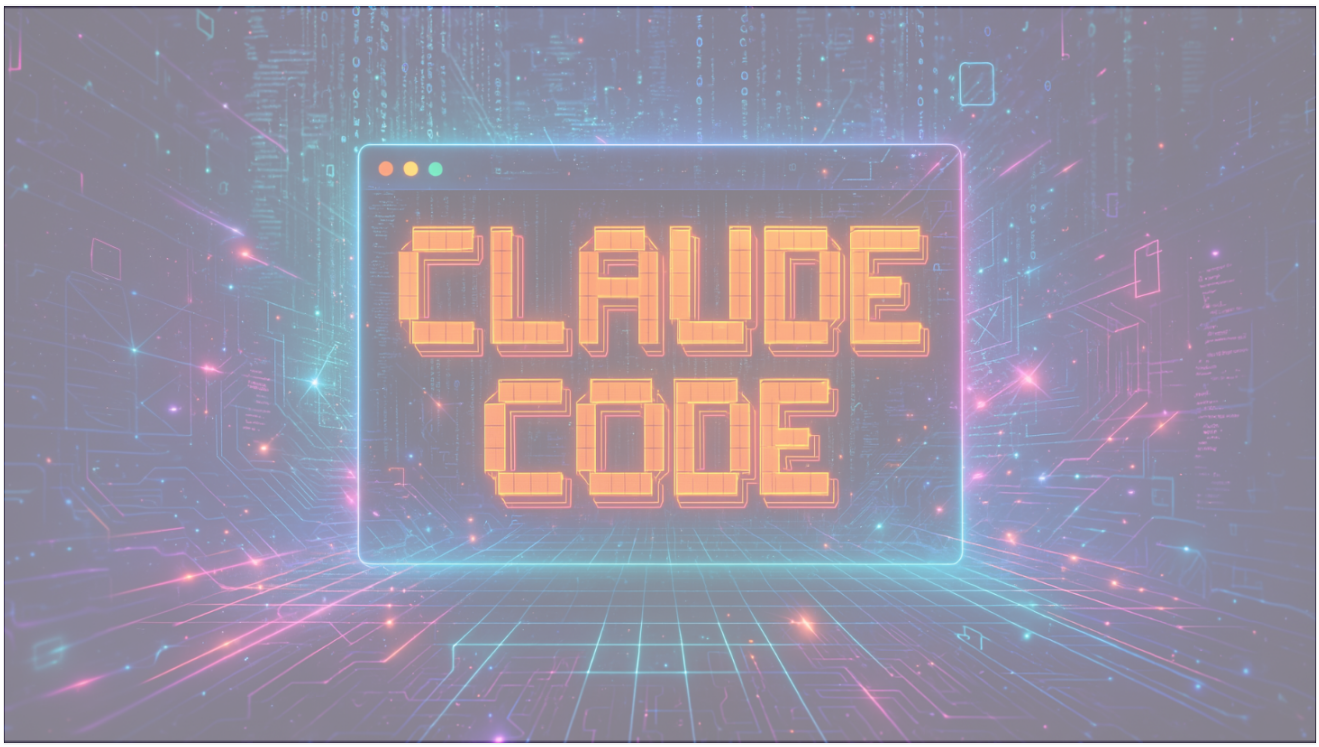


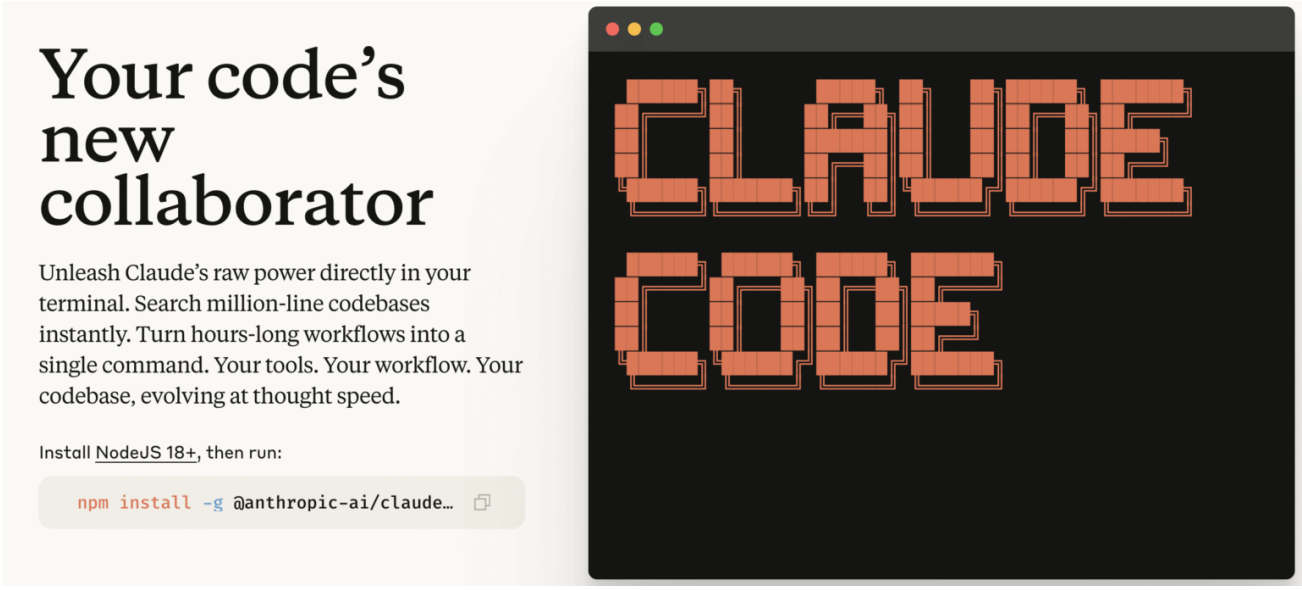
Les meilleures pratiques pour tirer le meilleur de CLAUDE CODE !

— Article initialement publié sur mon Patreon le 03/07/2025 —

Bon, comme vous le savez déjà sur Patreon, j'ai testé Claude Code depuis sa sortie début 2025 et franchement, pour moi c'est un game changer. Malheureusement, ce que j'ai remarqué, c'est que 90% des gens l'utilisaient mal. Ils lancent des commandes au pif, ça marche bien mais comme c'est pas parfait, ils se découragent et parfois finissent même par revenir à ChatGPT alors que c'est carrément moins bien !! Grave erreur !

C'est pourquoi aujourd'hui, on va monter un projet complet ensemble. J'ai pensé à un truc un peu simple mais assez complet, du genre un outil pour résumer automatiquement des articles de presse. Ça va nous permettre de voir toutes les bonnes pratiques recommandées par Anthropic et par moi, car oui, j'ai aussi pas mal testé Claude Code et j'ai quelques petits conseils à vous donner.

Avant de commencer, assurez-vous donc d'avoir **Claude Code** installé. Claude Code est inclus dans les abonnements Claude Pro (20$/mois), Max et Ultra. Une fois que vous avez accès, on peut y aller.



D'abord, on crée notre structure de projet. Ouvrez votre terminal et tapez :

```
mkdir article-summarizer
cd article-summarizer
```

Le truc le plus important avec Claude Code, c'est le fichier **CLAUDE.md**. C'est lui qui va dire à Claude comment bosser sur votre projet. C'est un peu comme un README mais spécialement pour l'IA. Claude le lit automatiquement au début de chaque session et vous pouvez l'initialiser dans Claude Code avec la commande `/init` ou le créer directement à la racine de votre projet :

```
touch CLAUDE.md
```

Et ensuite, mettez ceci dedans :

```
# Project: Article Summarizer

## Description
Un outil CLI pour résumer automatiquement des articles de
presse en français.
Utilise l'API OpenAI pour générer des résumés concis et
pertinents.

## Bash commands
- npm run dev: Lance l'outil en mode développement
- npm run build: Compile le TypeScript
- npm run test: Execute les tests
- npm run summarize: Lance l'outil de résumé

## Code style
- TypeScript strict mode
- Import/export ES modules
- Pas de any, tout doit être typé
- Tests unitaires pour chaque fonction

## Architecture
- src/index.ts: Point d'entrée CLI
- src/fetcher.ts: Récupération du contenu des articles
- src/summarizer.ts: Logique de résumé avec OpenAI
- src/config.ts: Configuration et variables d'environnement

## Testing
- Jest pour les tests unitaires
- Tests d'intégration pour l'API
- Mock des appels externes
```

Comme vous pouvez le voir, ce fichier CLAUDE.md contient une description de mon projet, une liste des commandes que je veux, le style du code, la présence de tests

unitaires, l'architecture que j'ai en tête, le langage à utiliser également...etc. Bref, je décris l'essentiel ici. Ce sont les règles immuables que devra respecter Claude Code à chaque fois qu'il travaillera sur votre projet.

D'ailleurs, petite astuce que j'ai trouvée, vous pouvez placer des fichiers CLAUDE.md à plusieurs endroits stratégiques ! En plus de la racine du projet, vous pouvez en mettre dans `~/.claude/CLAUDE.md` pour des instructions globales qui s'appliquent à tous vos projets. Super pratique si vous avez des conventions de code communes ! Et pour les monorepos, placez-en dans les sous-dossiers comme ça Claude chargera automatiquement le CLAUDE.md du dossier parent ET celui du sous-dossier où vous travaillez. Vous pouvez même utiliser `CLAUDE.local.md` pour des instructions locales que vous ne voulez pas commiter.

Voilà et maintenant, roulements de tambour, le moment magique !! On va demander à Claude de créer tout le projet d'un coup.

Lancez Claude Code et tapez le prompt suivant :

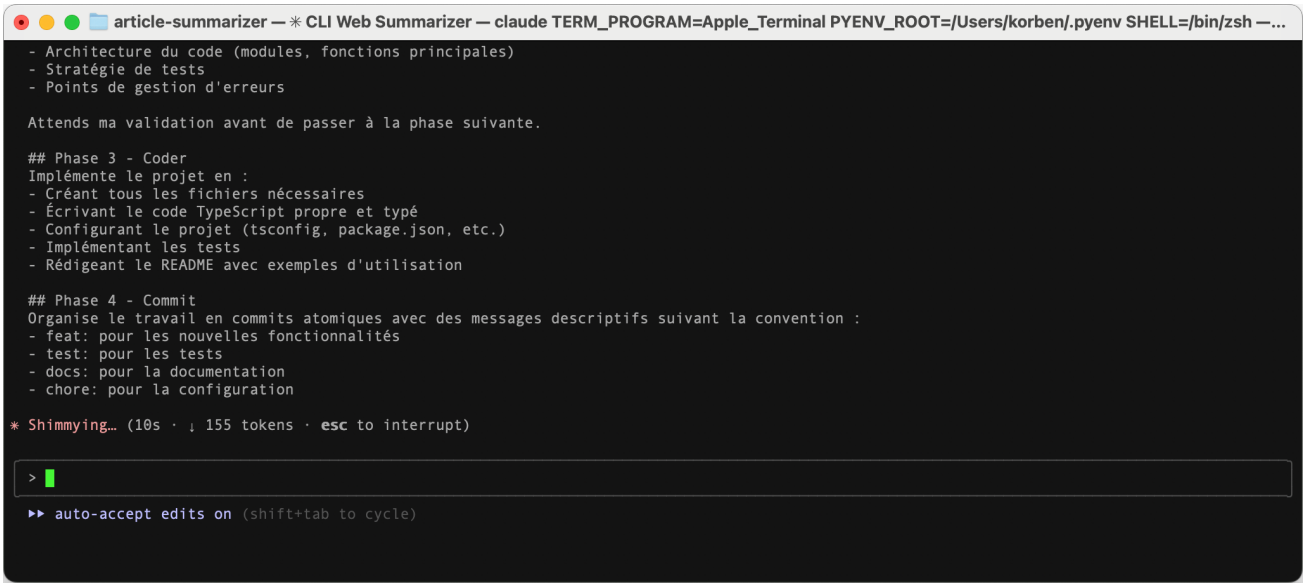
```
Crée un outil CLI en TypeScript pour résumer des articles web.

Suis ce workflow en 4 phases :

## Phase 1 - Explorer
Analyse d'abord les besoins :
- Un CLI qui prend une URL d'article en paramètre
- Récupération du contenu de l'article web
- Génération d'un résumé en français via l'API OpenAI
- Affichage du résumé dans le terminal
- Gestion d'erreurs robuste
- Tests unitaires
- Documentation README

Pose-moi des questions si tu as besoin de clarifications sur :
- Le format de sortie souhaité
- Les limites de taille pour les résumés
```

```
- Les types d'erreurs à gérer
- L'architecture préférée

## Phase 2 - Planifier
Crée un plan détaillé incluant :
- Structure des dossiers et fichiers
- Dépendances npm nécessaires
- Architecture du code (modules, fonctions principales)
- Stratégie de tests
- Points de gestion d'erreurs

Attends ma validation avant de passer à la phase suivante.

## Phase 3 - Coder
Implémente le projet en :
- Créant tous les fichiers nécessaires
- Écrivant le code TypeScript propre et typé
- Configurant le projet (tsconfig, package.json, etc.)
- Implémentant les tests
- Rédigeant le README avec exemples d'utilisation

## Phase 4 - Commit
Organise le travail en commits atomiques avec des messages
descriptifs suivant la convention :
- feat: pour les nouvelles fonctionnalités
- test: pour les tests
- docs: pour la documentation
- chore: pour la configuration
```

Claude va alors analyser votre CLAUDE.md et créer toute la structure du projet. Mais attention, c'est pas de la magie non plus. Il faudra vérifier ce qu'il fait et itérer s'il y a des problèmes.

Petite astuce que j'ai apprise en lisant la doc d'Anthropic, si vous avez un problème vraiment complexe à résoudre, ajoutez les mots magiques « **think**« , « **think hard**« ,

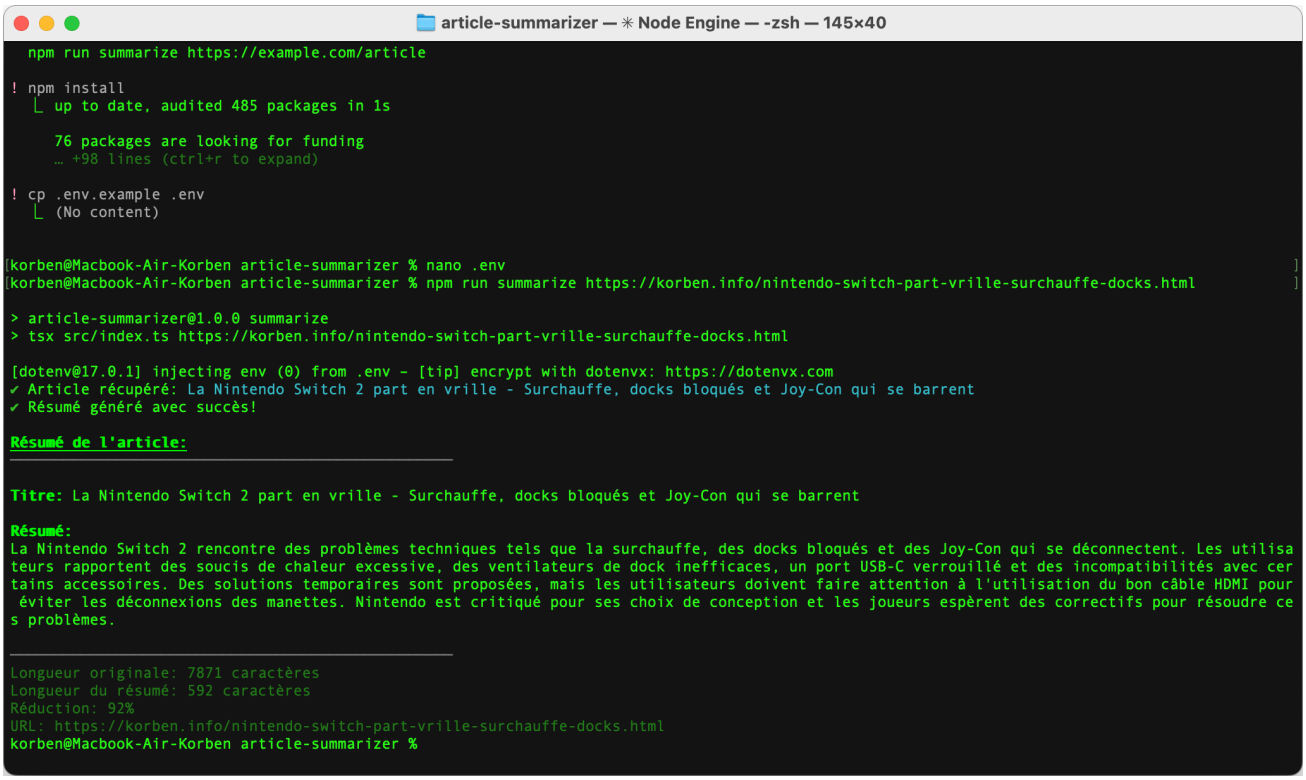
« **think harder** » ou même « **ultrathink** » dans votre prompt. Chaque niveau donne plus de temps de réflexion à Claude pour analyser le problème en profondeur. Pour notre projet c'est overkill, mais sur de l'architecture complexe ou du debug hardcore, ça peut vraiment faire la différence !



Le workflow, c'est celui en 4 phases recommandé par Anthropic :

- **Phase 1 – Explorer** : Claude analyse le projet et comprend ce qu'on veut faire. Il va poser des questions si besoin.
- **Phase 2 – Planifier** : Il crée un plan détaillé. C'est là que vous pouvez ajuster avant qu'il code.
- **Phase 3 – Coder** : L'implémentation proprement dite. Claude va créer les fichiers, écrire le code, configurer le projet.
- **Phase 4 – Commit** : Il fait des commits propres avec des messages descriptifs.

D'ailleurs, j'ai testé le code généré et je n'ai eu aucun bug, il a fonctionné du premier coup ! C'est assez exceptionnel et cela est dû à cette méthodologie.



Vous pouvez également personnaliser les permissions de Claude ! Par défaut, il demande la permission pour chaque action qui pourrait modifier votre système (écriture de fichiers, commandes bash, etc.). C'est chiant à la longue alors utilisez la commande `/permissions` pour ajouter des outils à la liste blanche.

Par exemple, ajoutez `Edit` pour toujours autoriser l'édition de fichiers, ou `Bash(git commit:*)` pour les commits git. Vous pouvez aussi lancer Claude avec le paramètre `--dangerously-skip-permissions` pour le mode « YOLO » où il fait tout sans demander, mais attention, utilisez ça uniquement dans un container Docker isolé !

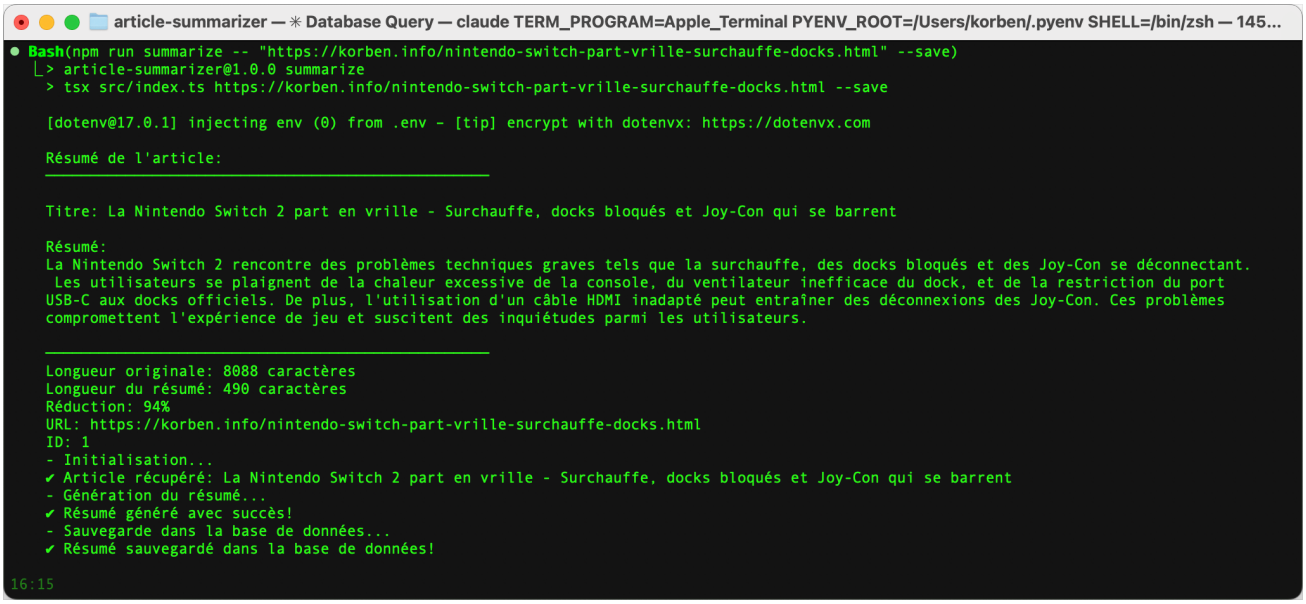
Un truc cool avec Claude Code ce sont aussi les serveurs MCP. En gros, ça permet à Claude d'interagir avec des services externes. Pour notre projet, on va donc ajouter un server pour PostgreSQL (pour stocker les résumés) et Puppeteer (pour scraper les sites récalcitrants). Vous devez bien sûr installer ces outils au préalable, créer les bases, les configurer...etc, et au pire, si vous ne savez pas faire, demandez à Claude Code qui le fera pour vous. De toute façon, je fourni sur le Patreon, le zip avec tout le code de ce projet.

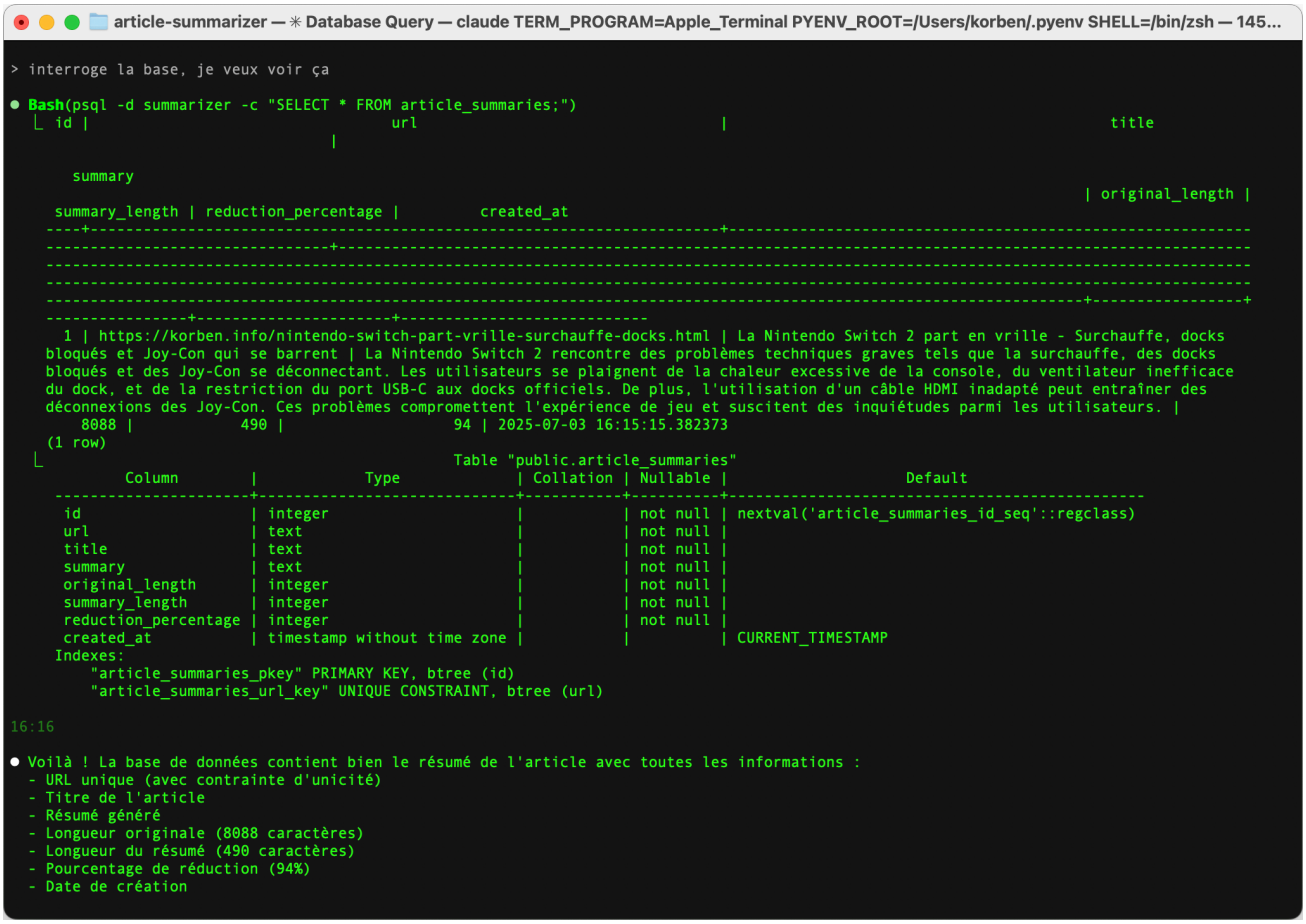
Ensuite, créez un fichier ici : `.claude/claude_config.json` :

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres",
"postgresql://localhost/summarizer"]
    },
    "puppeteer": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-puppeteer"]
    }
  }
}
```

Maintenant Claude peut directement interagir avec votre base de données et scraper des sites et surtout pas besoin d'écrire le code vous-même.

Vous pouvez aussi créer un fichier `.mcp.json` à la racine de votre projet et le commiter comme ça, tous les devs qui bossent sur le projet auront automatiquement accès aux mêmes serveurs MCP. C'est ce que fait Anthropic en interne et franchement c'est super pratique pour standardiser les outils dans une équipe !



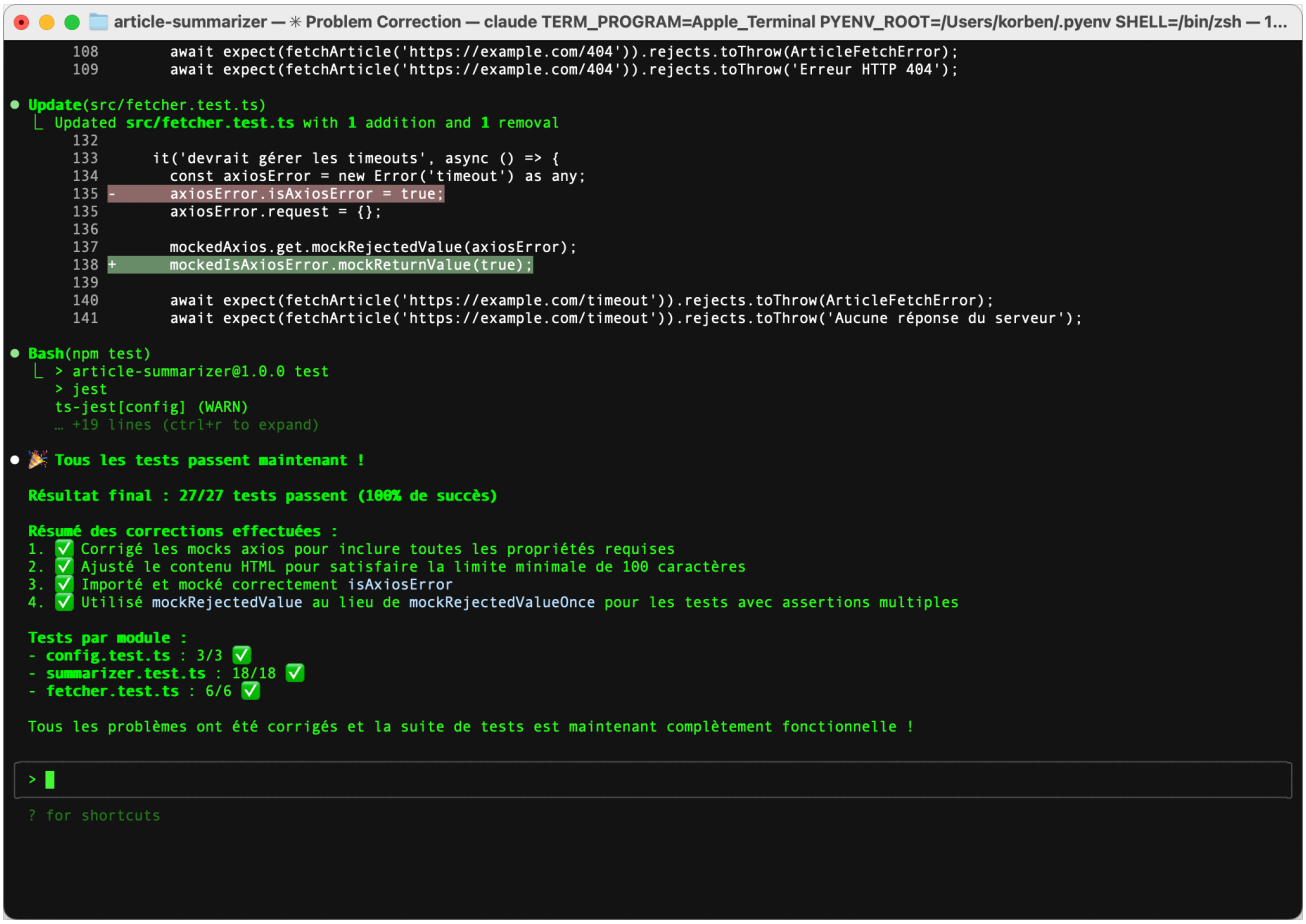


Autre conseil pratique pour obtenir de bons résultats avec Claude Code, c'est de pratiquer le test-driven development. C'est vraiment là où Claude Code brille. Demandez-lui d'abord d'écrire les tests comme ceci :

```
Écris des tests complets pour le module de résumé.
Les tests doivent couvrir :
- Résumé d'un article valide
- Gestion des erreurs réseau
- Articles trop courts/longs
- Formats de sortie
```

Une fois les tests écrits, demandez l'implémentation. Claude va alors coder jusqu'à ce que tous les tests passent. C'est magique aussi !

Ça force Claude à vraiment comprendre ce qu'on attend et évite qu'il parte dans tous les sens. Et le bonus, c'est que vous pouvez facilement revenir en arrière si l'implémentation ne vous plaît pas puisque les tests sont déjà commités !



Et pour les projets avec interface, prenez des screenshots ou faites des maquettes car Claude peut les analyser et itérer sur le design. Vous pouvez ainsi glisser-déposer des images ou donner le chemin d'un fichier image directement dans la zone de chat de Claude Code car il est multimodal ! Autant en profiter !

Genre vous faites une UI moche, vous prenez un screen, et vous dites « *améliore le design, inspire-toi de tel ou tel service*« . Et boom, il refait tout mais en mieux.

Ah, et encore un conseil d'ami, utilisez des commandes custom ! Moi ça m'a beaucoup aidé car ça permet de cadrer très fortement Claude Code pour qu'il arrête d'improviser. Comme ça vous pouvez mettre au point de véritables workflows au sein de Claude Code qui, s'ils sont bien faits, seront aussi efficaces qu'un code custom avec des appels API direct.

Créez un dossiers comme ceci `.claude/commands/`, puis par exemple à l'intérieur un fichier `optimize.md`.

Voici son contenu :

```
Analyse le code du projet et :
1. Identifie les problèmes de performance
2. Trouve les duplications de code
3. Suggère des optimisations
4. Vérifie la couverture de tests
5. Implémente les améliorations
```

Vous pouvez aussi utiliser le mot-clé spécial `$ARGUMENTS` dans vos commandes custom pour les rendre paramétrables. Par exemple, créez `.claude/commands/fix-issue.md` et mettez ça dedans :

```
Analyse et corrige l'issue GitHub numéro : $ARGUMENTS

1. Utilise `gh issue view` pour récupérer les détails
2. Comprends le problème décrit
3. Cherche les fichiers concernés dans le codebase
4. Implémente la correction
5. Écris et lance les tests
6. Vérifie que tout passe (lint, typecheck)
7. Commit avec un message descriptif
8. Crée une PR qui référence l'issue
```

Et maintenant vous pouvez juste taper `/project:fix-issue` **1234** et Claude s'occupera de tout ! C'est ça la puissance des commandes custom paramétrables.

Vous pouvez aussi (et c'est ce que j'ai fait dans certains de mes projets) créer des templates en YAML dans `.claude/shared/` qui contiendront vos instructions et les appeler dans votre fichier optimize.md comme ceci :

```
# Optimisation du Projet

## Phase 1 - Analyse Complète
{{.claude/shared/analysis.yml}}
```

```
Effectue une analyse approfondie en suivant ces étapes :
- Performance : identifie les goulots d'étranglement
- Code smell : trouve les duplications et anti-patterns
- Architecture : évalue la structure actuelle
- Tests : vérifie la couverture et la qualité

## Phase 2 - Planification des Optimisations
{{.claude/shared/optimization-plan.yml}}

Crée un plan détaillé avec :
1. Liste priorisée des problèmes trouvés
2. Solutions proposées pour chaque problème
3. Impact estimé de chaque optimisation
4. Ordre d'implémentation recommandé

**Attends ma validation avant de continuer.**

## Phase 3 - Refactoring
{{.claude/shared/refactoring-rules.yml}}

Applique les optimisations en respectant :
- Les principes SOLID
- Les patterns de clean code
- La rétrocompatibilité
- Les conventions du projet

## Phase 4 - Tests et Validation
{{.claude/shared/testing-strategy.yml}}

Assure-toi que :
- Tous les tests existants passent
- La couverture de tests est maintenue ou améliorée
- Les nouvelles optimisations sont testées
- Les benchmarks montrent une amélioration
```

```
## Phase 5 - Documentation et Commit
{{.claude/shared/commit-conventions.yml}}

Finalise avec :
- Mise à jour de la documentation
- Commits atomiques bien structurés
- Changelog des optimisations
- Métriques avant/après
```

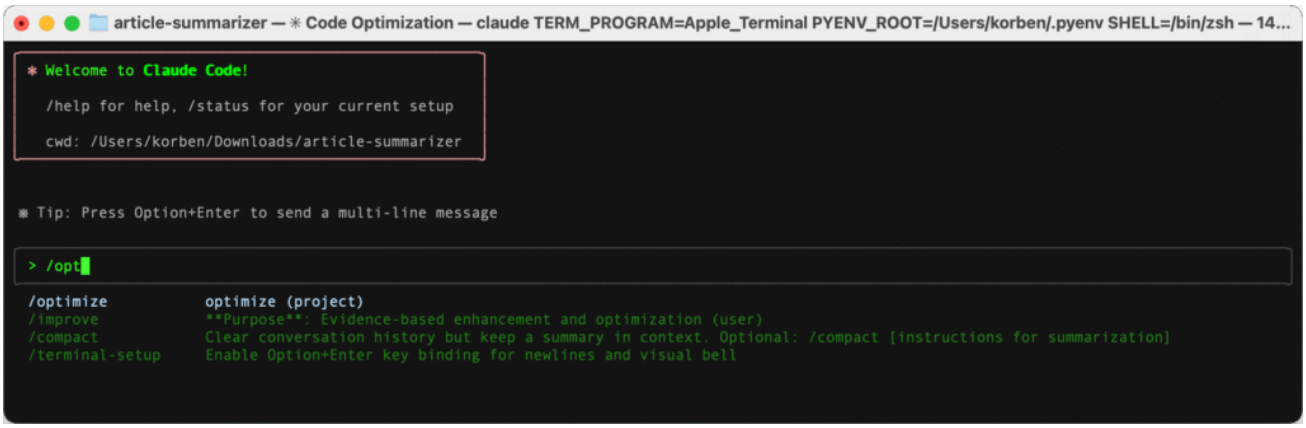
Et par exemple, dans un des YAML, comme `analysis.yml`, vous écrivez vos instructions comme ceci :

```
analyse:
  performance:
    - profiler_temps_execution
    - identifier_requetes_n_plus_un
    - verifier_utilisation_memoire
    - analyser_taille_bundle

  qualite_code:
    - detecter_duplications_seuil: 20_lignes
    - trouver_dependances_inutilisees
    - verifier_complexite_cyclomatique
    - identifier_methodes_longues

  architecture:
    - verifier_dependances_circulaires
    - valider_frontieres_modules
    - evaluer_couplage_cohesion
```

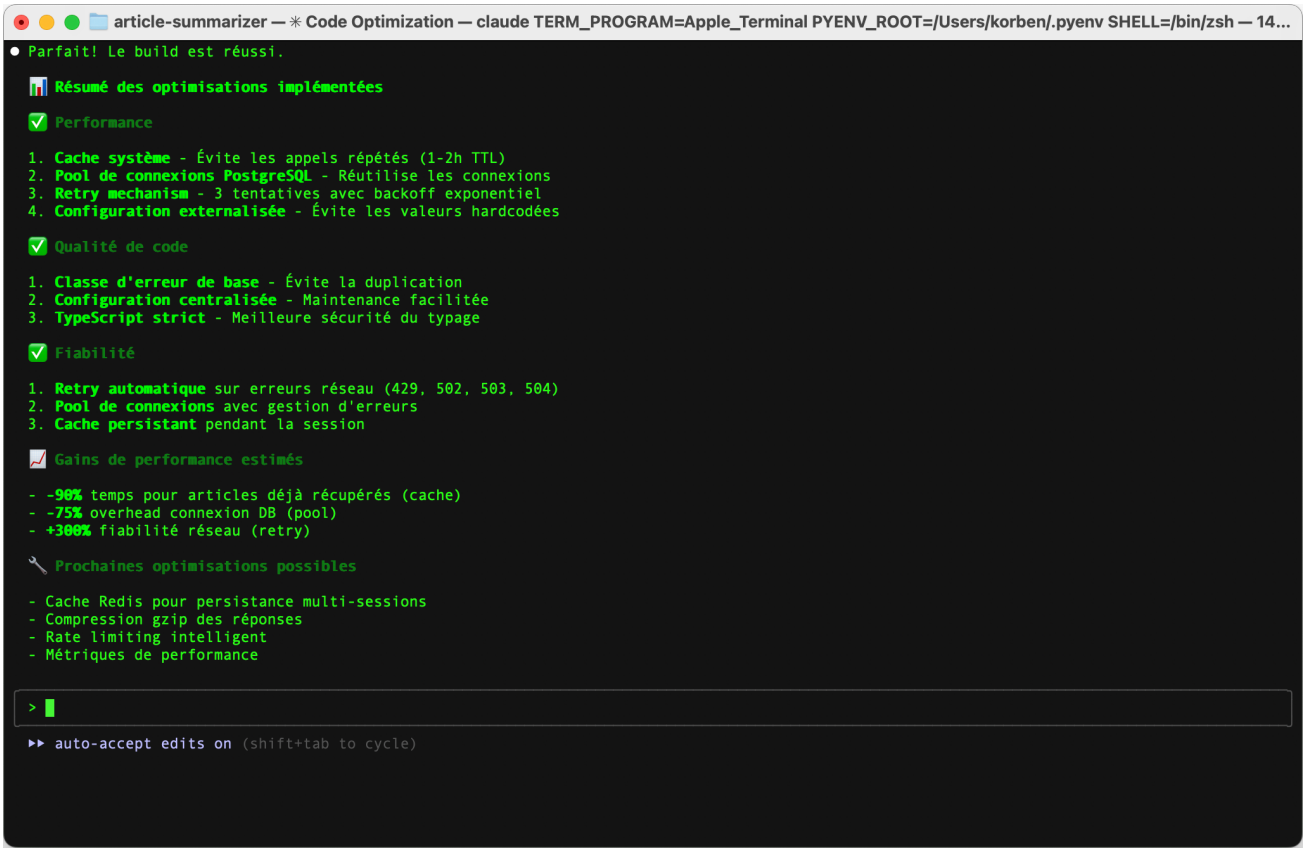
Pour choper cette structure, je me suis vraiment inspiré de ce qui avait été fait sur **SuperClaude** que j'ai adapté pour mes propres projets.

Procéder comme ceci va vous permettre de réutiliser des configurations standard

mais également de maintenir une certaine cohérence entre vos projets et bien sûr de tout personnaliser et de versionner facilement vos différentes versions sans tout casser. Et comme vous pouvez le voir, j'ai tout écrit en français et vous restez assez libre dans la rédaction et la structure (du moment que vous faites du vrai YAML). C'est le gros avantage des LLM.



Relancez ensuite Claude Code et vous pourrez maintenant juste taper `/optimize` et Claude fera tout le boulot.



Un truc que j'utilise tout le temps aussi, c'est la touche # pour ajouter des

instructions directement dans CLAUDE.md pendant que je bosse. Par exemple, si je veux qu'il respecte une nouvelle règle ou que je découvre une commande utile, je tape # et je dis à Claude « ajoute npm run format dans les commandes bash ». Et voilà, c'est ajouté et disponible pour toutes mes futures sessions ! C'est vraiment sa mémoire !

Maintenant, quelques erreurs courantes à éviter :

- **Corriger manuellement le code généré** : Non ! Si le code est pourri, améliorez votre prompt ou votre CLAUDE.md
- **Contexte surchargé** : Utilisez `/clear` régulièrement pour garder Claude focus
- **Pas de tests** : Sérieux, les tests c'est 50% de la puissance de Claude Code
- **Ignorer les erreurs** : Claude peut debugger, laissez-le faire
- **Ne pas être spécifique** : Plus vous êtes précis dans vos instructions, meilleur sera le résultat. Au lieu de « ajoute des tests », dites « écris des tests unitaires pour la fonction de parsing HTML, en mockant les appels réseau »

Et pour corriger Claude en cours de route, utilisez ces touches :

- **Echap** pour interrompre Claude à tout moment sans perdre le contexte
- **Double Echap** pour revenir en arrière dans l'historique et éditer un prompt précédent
- **MAJ+Tab** pour activer/désactiver le mode auto-accept si vous voulez que Claude bosse en autonomie
- Et vous pouvez aussi demander explicitement à Claude de faire un plan AVANT de coder avec cette simple demande : « *ne code rien pour l'instant*«

Et pour des workflows plus complexes, j'utilise pour ma part, la technique du « **multi-agent factory**« . En gros j'ai :

1. Un agent Claude pour planifier avec réflexion étendue
2. Un agent pour implémenter le code
3. Un agent de vérification qui valide tout

Pour la réalisation de ça, c'est assez similaire à ce que je viens de vous montrer. Il suffit de créer des rôles (planner, developer, validator), de leur donner chacun des

instructions précises, éventuellement un peu de config ou de contexte dans un .yml et de rédiger un workflow global qui appelera les agents à tour de rôle.

```
.claude/
├── agents/
│   ├── planner/
│   │   ├── instructions.md
│   │   └── context.yml
│   ├── developer/
│   │   ├── instructions.md
│   │   └── context.yml
│   └── validator/
│       ├── instructions.md
│       └── context.yml
├── workflows/
│   └── multi-agent-flow.md
└── shared/
    └── project-context.yml
```

C'est overkill pour notre petit projet, mais pour du gros dev, c'est imbattable.

Une technique de ouf que vous pouvez aussi pour les gros projets, ce sont les git worktrees ! Au lieu de cloner plusieurs fois votre repo, créez des worktrees pour bosser sur plusieurs features en parallèle. Voici comment faire :

```
# Créer un worktree pour une nouvelle feature
git worktree add ../projet-feature-auth feature/auth

# Lancer Claude dans chaque worktree
cd ../projet-feature-auth && claude

# Dans un autre terminal
git worktree add ../projet-feature-api feature/api
cd ../projet-feature-api && claude
```

Comme ça vous pouvez avoir 3-4 Claude qui bossent en parallèle sur des features

différentes sans conflits ! Quand c'est fini, nettoyez avec `git worktree remove ../projet-feature-auth`.

Et pour l'automatisation, Claude Code dispose aussi d'un mode headless ultra puissant avec ce paramètre : `claude -p "votre prompt"` . Cela lancera Claude sans interface, ce qui est parfait pour :

- Les hooks pre-commit pour vérifier le code
- Le triage automatique des issues GitHub
- Les migrations de masse sur des centaines de fichiers
- L'intégration dans vos pipelines CI/CD

Par exemple, pour migrer 2000 fichiers de React vers Vue :

```
#!/bin/bash
for file in $(find . -name "*.jsx"); do
  claude -p "Migre $file de React vers Vue. Retourne OK si succès, FAIL sinon." \
    --allowedTools Edit "Bash(git commit:*)" \
    --output-format json
done
```

Toutefois, j'ai l'impression que l'utiliser comme ça utilise l'API de Claude directement et sera donc facturé à la requête et pas inclus dans votre forfait Max.

Vous pouvez aussi utiliser Claude pour en apprendre un peu plus sur une nouvelle base de code ! C'est devenu ma technique préférée pour me plonger dans un projet existant (que je récupère sur Github par exemple… C'est comme ça que j'ai analysé SuperClaude d'ailleurs). En gros, je lance Claude, je fais un petit `/init` et je lui pose les mêmes questions que je poserais au dev du projet :

- « Comment fonctionne le système de logging ? »
- « Où est implémentée l'authentification ? »
- « Pourquoi on utilise cette librairie plutôt qu'une autre ? »
- « Quels sont les edge cases gérés par cette fonction ? »

Claude va alors explorer le code et répondre avec le contexte, ce qui va vous faire gagner un temps de dingue. C'est super cool pour se replonger dans du vieux coder « legacy » sans doc et dont le dev est parti en retraite y'a 15 ans ^^. Oui, ça sent le vécu, j'avoue !

Et si votre projet est sur Git, il faut absolument que vous installiez la commande `gh` comme ça Claude l'utilisera pour interagir avec votre repo. D'abord ça permet de ne plus se prendre la tête avec la syntaxe des commandes Git (j'aime pas, j'avoue) et surtout de faire :

- Des recherches dans l'historique git (« *Quels changements sont passés dans la v1.2.3 ?*«)
- D'écrire des messages de commit en analysant automatiquement vos changements
- De gérer les rebases complexes et résoudre les conflits
- Et aussi de créer des PR avec des descriptions détaillées
- ...etc etc

C'est vraiment super, je ne fonctionne plus que comme ça !

Et pour les chercheurs et data scientists parmi vous, Claude Code gère aussi les Jupyter notebooks ! Vous ouvrez votre notebook côte à côte avec Claude Code dans VS Code, vous le lancez en mode IDE avec la commande `/ide` et il pourra alors lire les outputs, y compris les graphiques. Vraiment pratique pour explorer des données ou nettoyer un notebook avant de le montrer aux collègues.

Pensez aussi à utiliser des checklists pour les tâches complexes ! Par exemple, si vous avez 50 erreurs de lint à corriger, vous pouvez demander ceci à Claude :

```
1. Lance le linter et écris toutes les erreurs dans
checklist.md
2. Corrige chaque erreur une par une
3. Vérifie après chaque correction
4. Coche l'erreur dans la checklist
5. Continue jusqu'à ce que tout soit propre
```

Claude va alors méthodiquement tout corriger (normalement) sans rien oublier, ce qui est parfait pour les migrations ou les refactorings massifs.

Et pour finir, quelques astuces que j'ai grappillé par ci, par là :

- **Utilisez la complétion par Tab** : Tapez le début d'un nom de fichier et Tab pour que Claude complète. Super pratique pour référencer des fichiers précis
- **Donnez des URLs à Claude** : Il peut lire des pages web avec sa commande Fetch ! Collez une URL de doc et demandez-lui de l'implémenter
- « **Pipez » des données** comme ceci : `cat logs.txt | claude` pour analyser des logs volumineux
- **Sans oublier le mode JSON pour l'automatisation** : Ajoutez `--output-format stream-json` en mode headless ce qui vous permettra de parser facilement les résultats

Ce projet qu'on vient de créer, c'est évidemment juste un exemple. Mais les principes restent les mêmes pour n'importe quoi : Assistant email, générateur de tweets, outil de veille techno… L'important c'est de bien structurer avec CLAUDE.md, d'utiliser les bonnes phases de workflow, et de laisser Claude faire le gros du boulot.

D'ailleurs, si vous voulez aller plus loin, checkez les Desktop Extensions sorties récemment. Ça permet d'installer des MCP servers en un clic, sans toucher au JSON ce qui est pratique pour les allergiques du terminal.

Et voilà, maintenant vous savez utiliser Claude Code comme un chef. Plus d'excuses pour coder comme en 2010 ! L'IA est là pour nous aider, alors autant en profiter intelligemment. Et si vous découvrez de nouvelles astuces et façons de faire, hésitez pas à me les envoyer, je suis toujours curieux !

Sources : Claude Code – Anthropic, Documentation officielle Claude Code, Claude Code Best Practices – Engineering at Anthropic, GitHub – Claude Code Repository, Desktop Extensions – Anthropic Engineering

---

Publié 3 juillet 2025 dans Intelligence artificielle
par Korben ✨

Étiquettes :
ai, automation, Claude, développement, tutoriel

Commentaires

Laisser un commentaire

Connexion en tant que Korben ✨. Modifier votre profil. Se déconnecter ? Les champs obligatoires sont indiqués avec *

Commentaire *

Laisser un commentaire