Claude Code
un assistant de codage agentique en CLI

Christophe Guyeux

IUT NFC – UMLP

12 mai 2026

Sommaire

Introduction

Objectifs du cours

- Comprendre ce qu'est un assistant de codage agentique en CLI.
- Maîtriser les briques de Claude Code : outils, permissions, mémoire, hooks, MCP, plugins.
- Construire des configurations professionnelles : sécurité, automatisation, intégration CI.
- Situer Claude Code par rapport à Codex CLI, Aider, Cursor, Continue, Cline.

Qu'est-ce que Claude Code ?

Claude Code est un assistant de codage agentique en ligne de commande, développé par Anthropic.

- « Agentique » : il décide quels outils appeler (Read, Edit, Bash, Grep...) pour accomplir une tâche.
- « En CLI » : il s'exécute dans un terminal, avec accès direct au système de fichiers et au shell.
- Pilotable depuis VS Code, JetBrains, le web, et en CI (GitHub Actions, GitLab CI).

Différence clé avec un chatbot

Un chatbot *discute* de votre code. Claude Code *le modifie* : il lit, édite, exécute des commandes, fait des commits Git.

Anatomie d'une session

1. Vous tapez `claude` dans un terminal, dans un répertoire de projet.
2. Vous demandez « ajoute une fonction qui valide les emails ».
3. Claude explore : `Grep` pour trouver le bon fichier, `Read` pour le lire.
4. Claude propose une modification : il vous montre le diff.
5. Vous validez ; il applique le `Edit`, lance les tests via `Bash`, commit.

Boucle agentique

`Penser` → `Choisir un outil` → `L'exécuter` → `Lire le résultat` → ...Cette boucle est exactement celle qu'on a programmée « à la main » dans le cours Langchain au chapitre 4.

L'écosystème Claude Code

Claude Code se présente sous plusieurs formes :

- CLI terminal (au cœur du cours) : `claude` dans un shell.
- Application desktop (Mac, Windows, Linux).
- Web : `https://claude.ai/code`.
- Extensions IDE : VS Code, JetBrains.
- Slack : `@Claude Code` dans un canal.
- CI/CD : GitHub Actions, GitLab CI.

Toutes partagent le même moteur (les outils, les permissions, le CLAUDE.md). On apprend une fois, on déploie partout.

Pourquoi un cours dédié ?

- Les outils intégrés (Read, Edit, Bash, Grep, WebFetch, Agent...) sont riches mais demandent à être compris pour bien prompter.
- Les permissions sont une vraie discipline : un agent qui exécute du Bash sans garde-fou est dangereux.
- Les hooks et les MCP servers ouvrent à l'automatisation et à l'intégration de systèmes externes.
- Bien utilisé, Claude Code accélère le développement de 2-5$\times$; mal utilisé, il introduit de la dette technique invisible.

Plan du cours

1. Introduction (ce chapitre).
2. Détour par Codex CLI (OpenAI), pour montrer que les concepts sont génériques.
3. Premiers pas : installation, login, première session.
4. Mémoire et contexte : `CLAUDE.md`, `/init`, `/memory`.
5. Outils et permissions : modes, règles, sécurité.
6. Sous-agents et slash commands.
7. Hooks (validation, journalisation, blocage).
8. MCP servers (Postgres, GitHub, Slack...).
9. Plugins, IDE, CI, et alternatives (Aider, Cursor...).

Et chez OpenAI ? Codex CLI

Claude Code n'est pas seul

Anthropic a lancé Claude Code, mais le concept d'assistant de codage agentique en CLI s'est généralisé.

Du côté d'OpenAI, l'équivalent direct s'appelle Codex CLI :

- Même philosophie : un agent dans le terminal, qui lit, écrit, exécute, commit.
- Mêmes briques : outils, permissions, mémoire projet, sous-agents, MCP.
- Code source ouvert (`github.com/openai/codex`).
- Accès gratuit limité pour tout détenteur d'un compte ChatGPT (un quota journalier suffit pour découvrir l'outil ; les comptes Plus/Pro lèvent les limites).

Bonne nouvelle

Ce que vous apprendrez avec Claude Code se transpose presque mot pour mot à Codex. Les concepts priment sur les commandes.

Installer et lancer Codex

```
# Installation officielle (npm, recommandée)
npm install -g @openai/codex

# Alternatives
brew install codex                          # macOS
# Binaires pré-compilés : releases GitHub openai/codex

# Vérifier la version
codex --version

# Première connexion (compte ChatGPT)
# -> ouvre une page web pour autoriser l'accès
codex login

# Démarrer une session dans un projet
cd ~/projets/mon-app
codex
```

- Prérequis : Node.js $\geq$ 18 (pour `npm`).
- `codex login` ouvre une page web ; on s'authentifie avec son compte ChatGPT.
- Lancer `codex` dans un répertoire de projet ouvre la REPL agentique.

Claude Code et Codex en parallèle

Concept	Claude Code	Codex CLI
Binaire CLI	`claude`	`codex`
Config projet (répertoire)	`.claude/`	`.codex/`
Mémoire projet (fichier)	`CLAUDE.md`	`AGENTS.md`
Skills	`.claude/skills/`	`.codex/skills/`
Sous-agents, MCP, hooks	oui	oui (équivalents)
Authentification	`/login`	`codex login`
Accès gratuit	non (Pro/Max)	oui, quota limité

Pour la suite du cours

On utilisera Claude Code comme support de référence. Les configurations, hooks et skills présentés se traduisent en Codex en renommant les répertoires et les fichiers de mémoire projet.

Premiers pas

Installation

```
# macOS, Linux, WSL : installation native (auto-update intégré)
curl -fsSL https://claude.ai/install.sh | bash

# Alternative macOS via Homebrew
brew install --cask claude-code

# Alternative Windows
# PowerShell : irm https://claude.ai/install.ps1 | iex
# winget     : winget install Anthropic.ClaudeCode
```

- Installation native : auto-update en arrière-plan, pas de Node.js requis.
- Homebrew, WinGet, apt/dnf/apk : disponibles aussi.

Connexion

Au premier lancement de `claude`, on est invité à se connecter :

- Claude Pro / Max / Team / Enterprise (recommandé).
- Compte Claude Console avec crédits prépayés (un workspace « Claude Code » est créé automatiquement pour suivre les coûts).
- Fournisseurs cloud entreprise : Amazon Bedrock, Google Vertex AI, Microsoft Foundry.

Identifiants stockés localement après le premier login. `/login` pour changer de compte.

Démarrer une session

Ouvrir le terminal dans un répertoire de projet, puis :

```
$ cd ~/projets/mon-app
$ claude
```

L'écran d'accueil affiche :

- Vos conversations récentes pour ce répertoire.
- Un rappel des commandes (`/help`, `/resume`).
- Les dernières mises à jour produit.

Pas besoin d'« ajouter du contexte » manuellement : Claude lit les fichiers *quand il en a besoin*.

Commandes CLI essentielles

```
# Démarrer une session interactive dans le projet courant
claude

# Reprendre la dernière conversation de ce répertoire
claude -c

# Choisir une conversation à reprendre dans une liste
claude -r

# Exécuter une seule requête, puis quitter (utile en script)
claude -p "explique la fonction handle_payment"

# Format JSON pour pipeline CI ou logging
claude -p "review du dernier commit" --output-format json

# Forcer un modèle particulier (Opus, Sonnet, Haiku)
claude --model haiku
```

Tableau des modes d'invocation

`claude`	Mode interactif (REPL)
`claude "task"`	Tâche unique, mode interactif
`claude -p "query"`	Requête unique, sortie texte, puis quitte
`claude -c`	Reprend la dernière conversation du répertoire
`claude -r`	Choisit une conversation antérieure
`-output-format json`	Sortie machine-readable (CI, logs)
`-model haiku`	Force un modèle moins cher / plus rapide

Exemple de session interactive

```
## Session interactive typique (transcript résumé)

```
$ cd ~/projets/api-paiements
$ claude

  Bienvenue dans Claude Code.
  Conversation récente : "fix race condition in payment handler"
  /help pour les commandes, /resume pour reprendre.

> que fait ce projet ?

  • Read(README.md)
  • Read(pyproject.toml)
  • Read(src/api/__init__.py)

  Ce projet est une API FastAPI pour traiter des paiements Stripe...
  [3 paragraphes de synthèse]

> ajoute un endpoint /refund qui annule un paiement

  • Grep("@app.post", src/)
  • Read(src/api/payments.py)
  • Edit(src/api/payments.py) -- propose un diff de 18 lignes

  J'ai ajouté `POST /refund` qui appelle `stripe.Refund.create`.
  Veux-tu que j'écrive aussi le test ?

> oui, et lance les tests

  • Read(tests/test_payments.py)
  • Edit(tests/test_payments.py) -- diff de 22 lignes
  • Bash(pytest tests/test_payments.py -v)

  3 nouveaux tests passent (durée 1.4 s). RAS.
```
```

Lecture de la session

- Claude annonce chaque appel d'outil : `Read`, `Grep`, `Edit`, `Bash`.
- Avant chaque modification, il propose un diff et attend l'approbation (mode `default`).
- `Bash(pytest...)` permet de vérifier que ça marche avant de passer à la suite.
- Le ton est conversationnel : « veux-tu que j'écrive aussi le test ? » est typique.

Sept conseils pour bien démarrer

1. Commencez par laisser Claude explorer avant de demander des modifications.
2. Soyez spécifique : « corriger le bug de connexion qui montre un écran vide » mieux que « corriger le bug ».
3. Découpez les grosses tâches en étapes numérotées.
4. Lisez toujours les diffs avant d'approuver.
5. Tapez `?` pour voir les raccourcis clavier.
6. `/clear` pour repartir d'un contexte vide ; `/compact` pour résumer l'historique.
7. Utilisez `Tab` pour la complétion, ↑ pour l'historique.

Récapitulatif

- Installation en une commande (curl, Homebrew, WinGet).
- Login OAuth avec un compte Claude (ou via fournisseur cloud entreprise).
- `claude` démarre une REPL ; `claude -p` pour un one-shot scriptable.
- Toujours valider les diffs avant d'approuver une modification.

Suite

Au prochain cours : la mémoire projet via `CLAUDE.md` pour que Claude comprenne d'emblée vos conventions.

Mémoire et contexte

Pourquoi un `CLAUDE.md` ?

Sans contexte projet, Claude doit tout redécouvrir à chaque session : architecture, conventions, commandes pour lancer les tests.

- Avec un `CLAUDE.md`, Claude lit ce fichier au démarrage et comprend le projet d'emblée.
- Le fichier persiste : il survit aux `/clear` et aux nouvelles sessions.
- Il est versionné dans Git, donc partagé avec l'équipe.

Règle de base : tout ce que vous expliqueriez à un nouveau collègue le premier jour devrait être dans `CLAUDE.md`.

Deux niveaux de `CLAUDE.md`

- Global : `~/.claude/CLAUDE.md`. Vos préférences personnelles, valables pour *tous* vos projets.
- Projet : `<racine>/CLAUDE.md`. Spécifique à un dépôt, versionné avec le code.

Les deux sont chargés simultanément : le global pose vos habitudes, le projet pose le contexte technique.

Bonne pratique

Global : préférences langagières, conventions cross-projet. *Projet* : architecture, commandes, état courant. Ne pas mélanger.

Exemple de CLAUDE.md global

```
# Instructions globales (extrait de ~/.claude/CLAUDE.md)

## Rédaction
- Toujours répondre en français.
- Pas de tirets cadratins (em dash) dans les textes destinés à autrui.
- Pas d'emojis dans les commits.

## Code
- Préférer ‘pathlib‘ à ‘os.path‘ en Python.
- Suivre la convention de nommage du projet courant (cf. CLAUDE.md projet).

## Suppression de fichiers
- Jamais de ‘rm‘ direct : utiliser ‘gio trash <fichier>‘ à la place.
```

Exemple de CLAUDE.md projet

```
# CLAUDE.md (projet api-paiements)

## Nature du projet
API REST FastAPI qui orchestre les paiements Stripe pour la marketplace.

## Architecture
- `src/api/` : routes FastAPI (un fichier par ressource).
- `src/services/` : logique métier (Stripe, base de données).
- `src/models/` : schémas Pydantic.
- `tests/` : pytest (fixtures dans `conftest.py`).

## Commandes essentielles
- `uv run pytest` : tests unitaires.
- `uv run uvicorn src.main:app --reload` : serveur local.
- `make lint` : ruff + mypy.

## Conventions
- Routes en anglais (`/payments`), code et tests en anglais, commentaires en français.
- Toujours wrapper les appels Stripe dans un `try/except StripeError`.
- Logger via `structlog`, jamais `print`.

## État courant
- Refactor en cours : passage de webhooks synchrones à une queue Redis (branche `feature/queue`).
- Le rate limiting Stripe nous a piégés en novembre, ne pas le réintroduire.
```

Structure recommandée

Un bon `CLAUDE.md` projet a typiquement quatre sections :

1. Nature du projet : à quoi ça sert, en deux phrases.
2. Architecture : la carte des dossiers et leur rôle.
3. Commandes essentielles : tests, dev server, lint, build.
4. Conventions et état courant : règles métier, écueils connus, refactors en cours.

Mettre à jour le fichier après chaque session significative : nouvelle convention adoptée, refactor terminé, écueil rencontré.

Commandes liées à la mémoire et au contexte

```
# Initialiser un CLAUDE.md projet à partir du contenu existant
/init

# Visualiser ce que Claude voit comme mémoire
/memory

# Effacer la conversation, garder la mémoire
/clear

# Compresser l'historique en un résumé pour gagner des tokens
/compact

# Reprendre une conversation antérieure
/resume

# Voir le contexte courant (tokens consommés, fichiers lus)
/context
```

Anatomie de `/init`

`/init` demande à Claude de lire le projet et de proposer un `CLAUDE.md` adapté :

1. Il scanne les fichiers de configuration (`package.json`, `pyproject.toml`, `Cargo.toml`...).
2. Il lit le README s'il existe.
3. Il infère l'architecture en parcourant les principaux dossiers.
4. Il propose un `CLAUDE.md` structuré ; vous éditez et committez.

Excellent point de départ, toujours à relire et éditer : Claude propose une base, l'humain la complète avec les non-dits.

Gestion du contexte

Le contexte est l'ensemble des messages, fichiers lus, et résultats d'outils que Claude voit pour décider.

- `/context` affiche les tokens consommés et les fichiers présents.
- `/clear` repart d'un contexte vide (la mémoire `CLAUDE.md` reste).
- `/compact` compresse l'historique : Claude résume ce qui a été fait pour libérer des tokens.

Bonne pratique

`/clear` avant de changer de sujet ; `/compact` si on dépasse 80 % du contexte sans avoir fini.

Récapitulatif

- `CLAUDE.md` global pour vos préférences, projet pour le contexte technique.
- `/init` génère une première version, à éditer.
- `/clear`, `/compact`, `/context` pour gérer le contexte d'une session.
- Mettre à jour le fichier après chaque session significative.

Suite

Au prochain cours : les outils qu'utilise Claude et les permissions pour les encadrer.

Outils et permissions

Les outils intégrés

Claude dispose nativement d'un ensemble d'outils pour interagir avec votre projet. Tableau résumé :

Outil	Rôle	Permission
`Read`	Lire un fichier	libre par défaut
`Edit`	Modifier un fichier (diff)	demande
`Write`	Créer / écraser un fichier	demande
`Bash`	Exécuter une commande shell	demande
`Grep`	Recherche par regex	libre
`Glob`	Lister des fichiers par pattern	libre
`WebFetch`	Lire une page web	demande
`Agent`	Déléguer à un sous-agent	demande

Anatomie d'un appel d'outil

Quand Claude veut lire un fichier, il annonce son intention :

- • `Read(src/api/payments.py)` apparaît dans la session.
- En mode `default`, vous voyez un prompt « autoriser ? » avec trois choix : oui, oui-toujours-pour-cet-outil, non.
- `Read` et `Grep` sont libres par défaut (pas de prompt) : lire ne fait pas de mal.
- `Edit`, `Write`, `Bash` demandent une approbation à chaque fois (sauf règle qui dit le contraire).

Les modes de permission

Cinq modes contrôlent le comportement par défaut.

Mode	Comportement
`default`	Read libre, le reste demande
`plan`	Read + commandes shell read-only seulement
`acceptEdits`	Auto-approuve Edit/Write dans le `cwd`
`auto`	Continu sans interruption (expérimental)
`bypassPermissions`	Tout autorisé (sauf garde-fous root)

Bonne pratique

`plan` pour explorer un projet inconnu en sécurité. `acceptEdits` en « flow » sur du code qu'on connaît bien. `bypassPermissions` seulement dans une VM ou un container jetable.

Bascule entre modes

```
# Mode par défaut : demande confirmation pour Edit/Write/Bash
claude

# Mode plan : Read et grep autorisés, aucune modification possible
claude --permission-mode plan

# Mode acceptEdits : auto-approuve les Edit/Write dans le répertoire courant
claude --permission-mode acceptEdits

# Mode bypassPermissions : tout est autorisé sauf garde-fous root (DANGEREUX)
claude --dangerously-skip-permissions

# Bascule en cours de session via slash command
# > /permissions plan
# > /permissions acceptEdits
```

Règles fines : `.claude/settings.json`

Les modes sont du *macroscopique*. Pour aller plus loin, on écrit des règles allow / ask / deny :

- allow : autorisation automatique sans prompt.
- ask : prompt à chaque fois.
- deny : refus automatique, l'outil ne s'exécute pas.

Ordre de priorité : deny > ask > allow. Le premier match gagne.

Exemple de règles

```
{
  "permissions": {
    "allow": [
      "Bash(npm run *)",
      "Bash(git status)",
      "Bash(git diff*)",
      "Bash(git log*)",
      "Bash(uv run pytest*)",
      "Read",
      "Edit",
      "Grep"
    ],
    "ask": [
      "Bash(git push*)",
      "Bash(git commit*)"
    ],
    "deny": [
      "Bash(rm -rf*)",
      "Bash(curl*|*sh)",
      "Read(.env*)",
      "Read(**/*.pem)"
    ]
  },
  "defaultMode": "default"
}
```

Décortiquer l'exemple

- allow : on autorise les commandes `npm run *`, `git` en lecture, `pytest`, et tous les `Read/Edit/Grep` libres.
- ask : on demande pour `git commit` et `git push` (actions Git destructives ou publiques).
- deny : on bloque `rm -rf`, les pipes `curl | sh` (vecteur d'attaque classique), et la lecture de fichiers sensibles (`.env`, certificats).

Patterns

`Bash(npm run *)` accepte `npm run test`, `npm run build`, etc. `Read(.env*)` bloque `.env`, `.env.local`, `.env.production`.

Localisation des règles

Les règles peuvent vivre à trois endroits :

1. `~/.claude/settings.json` : règles personnelles, valent pour tous vos projets.
2. `<racine>/.claude/settings.json` : règles d'équipe, versionnées dans Git.
3. `<racine>/.claude/settings.local.json` : règles du développeur, ignorées par Git.

Les trois sont fusionnés ; en cas de conflit, le plus restrictif gagne (deny l'emporte).

Récapitulatif

- Outils intégrés : `Read`, `Edit`, `Write`, `Bash`, `Grep`, `WebFetch`, `Agent`.
- Cinq modes de permission : `default`, `plan`, `acceptEdits`, `auto`, `bypassPermissions`.
- Règles fines allow / ask / deny dans `.claude/settings.json`.
- Trois localisations : global / projet (versionné) / local (non versionné).

Suite

Au prochain cours : les skills, la fonctionnalité signature de Claude Code pour étendre ses capacités.

Skills

Pourquoi les skills sont la fonctionnalité-clé

Un skill est une capacité réutilisable packagée que Claude charge *uniquement quand pertinent*.

Trois propriétés essentielles :

- Auto-déclenché : Claude voit la *description* et choisit le skill quand la question correspond.
- Progressive disclosure : tant que le skill n'est pas activé, il coûte zéro contexte. Activé, il charge ses instructions ; ses ressources annexes ne se chargent qu'à la demande.
- Réutilisable et partageable : un skill est un dossier. On le partage par Git, plugin, ou copie.

Skill, sous-agent, slash command : trois rôles distincts

	Skill	Sous-agent	Slash command
Forme	Capacité (instructions + scripts)	Worker isolé avec ses propres outils	Prompt préformaté
Contexte	Inline, dans la session principale	Fork, isolé, retourne un résumé	Inline
Invocation	Auto par description ou explicite	Explicite ou auto par description	Toujours explicite
Pour	Étendre les capacités de Claude	Déléguer une recherche / audit	Raccourcir un workflow répétitif

Les trois cohabitent sans friction. Le skill est la brique la plus puissante grâce à l'auto-invocation et au progressive disclosure.

Anatomie d'un SKILL.md

Un skill est un dossier <nom>/ contenant a minima un fichier SKILL.md :

- Un frontmatter YAML : name, description, allowed-tools, etc.
- Un corps Markdown : les instructions que Claude suivra une fois le skill activé.

Champ critique

Le description est ce qui déclenche l'activation automatique. Tout se joue ici.

Exemple : skill code-review

```
---
name: code-review
description: Review code changes for security, performance, and correctness. Trigger with a PR URL or diff, "review this before I merge", "is this
    code safe?", or when checking a change for N+1 queries, injection risks, missing edge cases, or error handling gaps.
allowed-tools: Read, Bash(git diff*), Bash(git log*), Grep
---

# Code Review

Tu es un senior engineer qui fait une revue de code rigoureuse.

## Méthode
1. Récupère le diff : `git diff HEAD~1` ou la diff fournie en argument.
2. Lis chaque fichier modifié pour comprendre le contexte autour.
3. Évalue dans cet ordre :
   - **Sécurité** : injection SQL, XSS, secrets en clair, auth cassée.
   - **Correction** : edge cases, gestion d'erreur, race conditions.
   - **Performance** : requêtes N+1, boucles non bornées, index manquants.
   - **Lisibilité** : noms, complexité, duplications.

## Sortie attendue
- **Bloquants** : issues à corriger avant merge.
- **Suggestions** : améliorations souhaitables.
- **RAS** si tout est propre.

Sois bref. Pas d'emojis, pas de félicitations.
```

Lecture

- `description` : énonce explicitement les phrases-types qui doivent déclencher (« review this before I merge », « is this code safe ? »). Pas de description vague comme « aide à la revue ».
- `allowed-tools` : restreint le skill à `Read`, certains `Bash`, et `Grep`. Pas de `Write` : la revue ne modifie rien.
- Le corps Markdown structure la méthode : sécurité d'abord, puis correction, puis performance.
- Format de sortie imposé pour rester lisible.

Progressive disclosure : économiser le contexte

Un skill se charge en trois couches :

1. Frontmatter (`name` + `description`) : toujours visible, taille minime. C'est la *table des matières* des skills disponibles.
2. Corps de `SKILL.md` : chargé *uniquement* quand le skill est activé.
3. Ressources annexes (autres fichiers du dossier) : chargées *seulement* si le corps les référence.

Conséquence

On peut avoir 50 skills installés, chacun avec 10 KB de doc, sans surcharger le contexte. Seuls ceux qu'on utilise vraiment paient leur coût.

Localisation des skills

- Personnel : `~/.claude/skills/<nom>/SKILL.md` (vos skills, valent pour tous vos projets).
- Projet : `<racine>/.claude/skills/<nom>/SKILL.md` (versionnés avec le code).
- Plugin : `<plugin>/skills/<nom>/SKILL.md`, namespacé en `<plugin>:<skill>`.
- Natifs : fournis avec Claude Code (`pdf`, `docx`, `xlsx`, `code-review`, `debug`, `search`, `recall`, `skill-creator`, etc.).

Priorité en cas de conflit de nom : enterprise > personnel > projet.

Skill avec script : structure typique

Un skill peut embarquer ses propres scripts. La variable `${CLAUDE_SKILL_DIR}` résout son chemin.

- `SKILL.md` décrit quand et comment appeler le script.
- Le script Python/Bash/quelconque vit à côté de `SKILL.md`.
- Des références (`references/options.md`, etc.) chargées à la demande pour la doc avancée.

Exemple : skill pdf-extract

````
---
name: pdf-extract
description: Extract text and tables from PDF files. Trigger when the user asks to
    "lire ce PDF", "extraire d'un PDF", "convertir PDF en markdown", or mentions a
    .pdf file path.
allowed-tools: Bash(python3 *), Read
---

# PDF extraction

Pour chaque chemin PDF passé en argument :

```bash
python3 ${CLAUDE_SKILL_DIR}/extract.py "$ARGUMENTS"
```

Le script écrit le texte page par page sur stdout. Pour les tableaux complexes,
    reformate-les en markdown (`|---|`).

Pour les options avancées (OCR, images, en-têtes répétés), voir `references/options
    .md` qui sera chargé à la demande.
````

Le script associé

```python
#!/usr/bin/env python3
"""Extraction texte d'un PDF, page par page (extract.py du skill pdf-extract)."""
import sys
from pypdf import PdfReader

if len(sys.argv) != 2:
    sys.exit("Usage : extract.py <chemin.pdf>")

pdf = PdfReader(sys.argv[1])
for i, page in enumerate(pdf.pages, 1):
    print(f"--- page {i} ---")
    print(page.extract_text())
```

L'arborescence et les skills natifs

````
## Arborescence type d'un skill avec scripts

```
~/.claude/skills/pdf-extract/
+-- SKILL.md            <- frontmatter + instructions, toujours visible (court)
+-- extract.py          <- script invoqué par SKILL.md
`-- references/
    `-- options.md      <- doc avancée, chargée à la demande
```

## Skills natifs disponibles

```
build-dashboard      cv                 design-doc-mermaid
debug                docx               explore-data
markdown-converter   pdf                pptx
recall               reflect            researcher
review               search             security-review
skill-creator        sql-queries        task-management
view-pdf             web-artifacts      xlsx
```

Lister à tout moment : `/help`, ou demander à Claude « quels skills as-tu disponibles ? ».
````

Bonnes pratiques pour la `description`

- Énoncer les triggers : « trigger when user asks to ... », « use for ... ».
- Spécifique, jamais générique : « Review code for security risks » oui ; « Help with code » non.
- Inclure les mots-clés naturels de l'utilisateur (« PR », « diff », « merge », « deploy »).
- Mettre l'essentiel en premier : la description peut être tronquée si elle dépasse le budget.

Erreur classique

« Skill utilitaire pour aider » : Claude ne le déclenchera jamais car rien n'y fait écho dans une question d'utilisateur.

Workflow de création

1. `mkdir -p ~/.claude/skills/mon-skill`
2. Écrire `SKILL.md` avec un frontmatter minimal et des instructions précises.
3. Tester en posant une question qui devrait matcher la description ; si le skill ne se déclenche pas, ajuster la description.
4. Itérer ; ajouter des scripts ou des références au besoin.

Aide officielle

Le skill natif `/skill-creator` guide la création, l'évaluation et l'optimisation d'un skill custom. À utiliser dès la première création.

Récapitulatif

- Un skill = un dossier `<nom>/` avec `SKILL.md` + ressources optionnelles.
- Auto-invocation par matching de `description` : la qualité de cette description fait tout.
- Progressive disclosure : zéro coût contexte tant qu'inactif, charge à la demande.
- Skills personnels, projet, plugin, natifs : couches multiples, priorité *enterprise* > *personnel* > *projet*.
- Peut embarquer des scripts via `${CLAUDE_SKILL_DIR}`.

Suite

Au prochain cours : sous-agents et slash commands, les deux autres mécanismes d'extension.

Sous-agents et slash commands

Pourquoi déléguer à un sous-agent ?

Le contexte principal est précieux : tout fichier lu, tout résultat d'outil le remplit. Une fois plein, Claude perd en qualité.

Un sous-agent a son propre contexte isolé :

- Il fait sa recherche, lit beaucoup de fichiers, accumule des résultats.
- Il rend une synthèse courte au contexte principal.
- Le contexte principal reste léger, focalisé sur la tâche utilisateur.

Analogie

Un sous-agent est à Claude ce qu'un stagiaire qui rapporte une synthèse de réunion est au manager : il a vu tous les détails, mais ne lui rapporte que l'essentiel.

Les sous-agents natifs

Quelques sous-agents fournis par Claude Code :

- Explore : recherche et exploration de codebase à grande échelle.
- Plan : planification d'une implémentation, sans édition.
- code-reviewer : revue d'un changement avant merge.
- statusline-setup : configure votre status line.

Liste à jour avec `/agents`.

Créer un sous-agent custom

Un fichier markdown sous `.claude/agents/<nom>.md` avec frontmatter YAML suffit.

```
---
name: code-reviewer
description: Review code changes for security, performance, and correctness. Trigger with a PR URL or diff, "review this before I merge", "is this
    code safe?".
tools: Read, Grep, Bash, WebFetch
model: sonnet
---

You are an experienced senior engineer doing a code review.

Your priorities, in order :
1. Security : SQL injection, XSS, secrets in code, broken auth.
2. Correctness : edge cases, error handling, race conditions.
3. Performance : N+1 queries, unbounded loops, missing indexes.
4. Style and maintainability.

Output format :
- "Bloquants" : issues that must be fixed before merging.
- "Suggestions" : nice-to-have improvements.
- "RAS" : if everything looks fine.

Be terse. No emojis. No congratulations.
```

Anatomie du frontmatter

- name : identifiant unique, utilisé pour invoquer.
- description : c'est ce qui déclenche l'invocation automatique. À soigner : phrases types qui font matcher « review this code », etc.
- tools : liste blanche des outils accessibles au sous-agent. Restreindre = sécuriser.
- model : `opus` (cher, profond), `sonnet` (équilibre), `haiku` (rapide, simple).

Les slash commands

Une slash command est un raccourci tapé en `/<nom>` qui :

- exécute du texte préformaté (un prompt long, des instructions étape par étape) ;
- peut accepter des arguments (`$ARGUMENTS`) ;
- est partagée via Git (équipe) ou perso (`~/.claude/commands/`).

Slash commands natives

```
# Slash commands natives utiles
/help                # liste les commandes et skills disponibles
/agents              # liste les sous-agents (natifs et custom)
/init                # génère un CLAUDE.md projet
/memory              # édite la mémoire en cours
/clear               # repart d'un contexte vide
/compact             # résume l'historique
/resume              # reprend une conversation
/review              # revue de code automatique
/permissions         # voir et éditer les règles
/context             # affiche tokens consommés et fichiers chargés
/login               # change de compte
/exit                # quitter
```

Créer une commande custom

Fichier `.claude/commands/deploy-staging.md` :

```
---
description: Déploie la branche courante en staging via le script make deploy-staging.
argument-hint: [version-tag]
---

Tu vas déployer en staging la branche `$ARGUMENTS` (par défaut : la branche courante).

Étapes :
1. Vérifie que `git status` est propre.
2. Lance `make test` ; bloque le déploiement si un test échoue.
3. Lance `make deploy-staging`.
4. Affiche l'URL de l'environnement et l'état des healthchecks.

Si l'une des étapes échoue, arrête-toi et explique le problème.
```

Comment l'utiliser

- En session : taper `/deploy-staging v1.4.2`.
- `$ARGUMENTS` sera remplacé par `v1.4.2`.
- Claude exécute les étapes décrites dans le markdown.

Bonne pratique

Une commande custom est un prompt versionné. Versionner les bons prompts dans Git évite que chacun écrive le sien à la main, avec des oublis.

Sous-agent ou slash command ?

Sous-agent	Slash command
Contexte isolé.	Reste dans le contexte principal.
Tools whitelistés.	Toutes les tools de la session.
Invocation auto par description.	Invocation explicite /<nom>.
Pour : recherche, audit, longue tâche.	Pour : workflows répétitifs, prompts longs.

Récapitulatif

- Sous-agent : worker à contexte isolé, idéal pour la recherche et l'audit.
- Slash command : prompt versionné, idéal pour les workflows répétitifs.
- Frontmatter YAML pour les deux ; `description` bien soignée pour le routage automatique.
- `$ARGUMENTS` pour passer des paramètres à une slash command.

Suite

Au prochain cours : les hooks, qui permettent de réagir aux événements de Claude (validation, journalisation, blocage).

Hooks

Pourquoi des hooks ?

Un hook est un script déclenché par un événement de Claude Code : avant un outil, après un outil, à la fin d'une session, etc.

Cas d'usage courants :

- Validation : refuser une commande dangereuse avant exécution.
- Auto-formatage : lancer `ruff` ou `prettier` après chaque `Edit`.
- Journalisation : enregistrer chaque outil dans un log d'audit.
- Blocage en fin de session : empêcher la sortie tant qu'un commit n'a pas été fait.

Les événements disponibles

Événement	Quand il se déclenche
`SessionStart`	Au démarrage d'une session
`UserPromptSubmit`	Quand l'utilisateur envoie un message
`PreToolUse`	Avant l'exécution d'un outil
`PostToolUse`	Après l'exécution d'un outil
`Stop`	Quand Claude finit sa réponse
`SessionEnd`	À la fermeture de la session
`PreCompact`	Avant un compactage de contexte
`Notification`	Notification système (idle, etc.)

Anatomie d'un hook

Un hook est :

1. un script (Bash, Python, n'importe quoi d'exécutable) ;
2. qui reçoit un JSON sur stdin décrivant l'événement ;
3. et retourne un code de sortie :
   - 0 : autorisation, silencieux.
   - 2 : blocage, le `stderr` est montré à l'utilisateur.
   - autre : non bloquant, juste un avertissement.

Configuration : `.claude/settings.json`

Trois hooks ici : un `PreToolUse` sur `Bash`, un `PostToolUse` sur `Edit|Write`, un `Stop`.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "tools/check_dangerous_bash.py"
          }
        ]
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "ruff format $CLAUDE_FILE_PATHS && ruff check --fix $CLAUDE_FILE_PATHS"
          }
        ]
      }
    ],
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "tools/check_cahier_updated.py"
          }
        ]
      }
    ]
  }
}
```

Décortiquer

- `PreToolUse / matcher: "Bash"` : intercepte chaque commande Bash. Notre script Python décide si elle est dangereuse.
- `PostToolUse / matcher: "Edit|Write"` : après une modification, on lance `ruff format` sur les fichiers touchés via `$CLAUDE_FILE_PATHS`.
- `Stop` : à la fin de la réponse, on vérifie qu'un cahier de labo a été mis à jour si du travail substantiel a eu lieu.

Exemple de hook : bloquer les bash dangereux

```
"""Hook PreToolUse : bloque les commandes Bash dangereuses.

Reçoit un JSON sur stdin avec la commande proposée. Sortie :
  - exit 0 : autoriser (silencieux).
  - exit 2 : bloquer (le message stderr est montré à l'utilisateur).
"""
import json
import re
import sys

DANGEREUX = [
    r"\brm\s+-rf\s+/",
    r"\bcurl\b.*\|\s*(ba)?sh\b",
    r"\bdd\s+if=",
    r":\(\)\s*\{",
]

evt = json.load(sys.stdin)
cmd = evt.get("tool_input", {}).get("command", "")

for pattern in DANGEREUX:
    if re.search(pattern, cmd):
        print(f"Hook : commande dangereuse bloquée ({pattern}).", file=sys.stderr)
        sys.exit(2)

sys.exit(0)
```

Lecture

- Le script lit le JSON d'événement sur `stdin`.
- Il extrait `tool_input.command` : la commande shell proposée par Claude.
- Il teste contre une liste de patterns dangereux (`rm -rf /`, `curl | sh`, `dd if=`, fork bomb).
- Match : `exit 2` avec un message $\rightarrow$ Claude refuse d'exécuter et explique à l'utilisateur.

Bonne pratique

Les hooks sont exécutés sur votre machine, avec vos droits. Ne pas y faire confiance à un script venu d'ailleurs sans audit.

Patterns courants de hooks

- Auto-format : `PostToolUse Edit|Write` → `prettier`, `ruff`, `gofmt`.
- Audit log : `PreToolUse` → append au log de tous les outils utilisés.
- Notification : `Stop` → envoie un message Slack pour les sessions longues.
- Garde-fou : `PreToolUse Bash` → refuser les patterns dangereux.
- Forcer commit : `SessionEnd` → refuse la sortie si `git status` montre des changements.

Récapitulatif

- Un hook = un script + un événement + un matcher.
- Codes de sortie : 0 (autoriser), 2 (bloquer), autre (avertir).
- Configuration centrale dans `.claude/settings.json`.
- Scripts dans la machine locale, attention à la confiance.

Suite

Au prochain cours : les MCP servers pour donner à Claude un accès à des systèmes externes (DB, Slack, GitHub).

MCP servers

Le problème

Claude est puissant pour le code, mais il ne sait rien de :

- Votre base de données de production.
- Vos PRs en cours sur GitHub.
- Vos messages Slack, vos tickets Jira/Linear.
- Vos fichiers Drive, vos calendriers, vos mails.

Comment lui donner accès à *tout cela* sans réinventer l'intégration à chaque fois ?

Le Model Context Protocol (MCP)

MCP est un protocole ouvert (Anthropic + communauté) qui standardise la communication entre :

- un LLM client (Claude Code, mais aussi d'autres),
- et des serveurs MCP qui exposent des outils et des ressources d'un système particulier.

Analogie

MCP est à un assistant IA ce que LSP est à un IDE : un protocole standard qui rend les intégrations *combinables*. Un serveur MCP écrit une fois fonctionne avec tout client compatible.

L'écosystème MCP

Quelques serveurs disponibles « prêts à l'emploi » :

- Postgres / SQLite / MySQL : exécution de requêtes, introspection schéma.
- GitHub / GitLab : PRs, issues, commits, recherche.
- Slack : poster, lire les canaux, fil d'une conversation.
- Linear / Jira / Notion : tickets, pages, projets.
- Google Drive / Gmail / Calendar : recherche, lecture, création.
- Filesystem : accès contrôlé à un dossier hors du projet courant.

Plus de 300 serveurs dans le registre officiel + écosystème custom.

Configurer des MCP : `.mcp.json`

Au niveau projet, on liste les serveurs MCP avec leur type (`stdio`, `http`, `sse`) :

```
{
  "mcpServers": {
    "postgres": {
      "type": "stdio",
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres"],
      "env": {
        "DATABASE_URL": "postgres://user:pass@localhost:5432/mydb"
      }
    },
    "github": {
      "type": "stdio",
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN": "ghp_xxxxxxxxxxxxxxxxxxx"
      }
    },
    "filesystem-data": {
      "type": "stdio",
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "/srv/data/datasets"
      ]
    },
    "slack": {
      "type": "http",
      "url": "https://mcp.slack.com/v1",
      "headers": {
        "Authorization": "Bearer xoxb-..."
      }
    }
  }
}
```

Lecture

- `postgres` : démarré comme un sous-processus (`stdio`) avec une URL de connexion.
- `github` : pareil, avec un PAT (token d'accès personnel).
- `filesystem-data` : expose le dossier `/srv/data/datasets` en lecture/écriture contrôlée.
- `slack` : serveur HTTP distant avec un Bearer token.

Sécurité

Le `.mcp.json` contient des secrets : il ne doit jamais être committé tel quel. Versionner un `.mcp.json.example` et lire les vrais secrets via variables d'environnement (`${ENV_VAR}`).

Exemple de session avec un MCP postgres

```
## Avec un MCP postgres branché

‘‘‘
> combien de commandes sont restées en pending depuis hier ?

  • mcp__postgres__query(
      "SELECT COUNT(*) FROM orders
       WHERE status = 'pending'
         AND created_at >= now() - interval '1 day'"
    )

  Il y a 47 commandes en attente depuis hier.

> montre les 5 plus anciennes avec le client et le total

  • mcp__postgres__query(
      "SELECT id, customer_email, total_cents/100.0 as eur, created_at
       FROM orders
       WHERE status = 'pending'
       ORDER BY created_at ASC
       LIMIT 5"
    )

  | id   | client                 | total | depuis     |
  |------|------------------------|-------|------------|
  | 1042 | a.dupont@exemple.fr    | 89.00 | 2j 4h      |
  | 1057 | b.martin@exemple.fr    |152.50 | 1j 18h     |
  | ...
‘‘‘
```

Ce qui change pour Claude

- Au démarrage de session, Claude découvre les outils exposés par chaque MCP : `mcp__postgres__query`, `mcp__github__create_pr`, etc.
- Le routage est automatique : sur une question SQL, Claude appelle `mcp__postgres__query`.
- Permissions fines : on peut autoriser `mcp__github__*` mais refuser `mcp__github__delete_repo`.

Créer son propre MCP server

Un MCP server est un petit programme qui :

1. Implémente le protocole MCP (en JSON sur `stdio` ou en HTTP).
2. Expose une liste de tools (avec schéma JSON Schema) et de resources.
3. Reçoit des appels et retourne des résultats.

SDK officiels en Python (`mcp[cli]`) et TypeScript (`@modelcontextprotocol/sdk`).

Exemple typique

« API interne maison » : 30 lignes de Python autour de votre client SDK et le LLM peut piloter votre back-office.

Récapitulatif

- MCP : protocole standard pour brancher Claude sur des systèmes externes.
- Configuration via `.mcp.json` : type (`stdio`/`http`/`sse`), commande ou URL, secrets.
- Outils exposés sous le préfixe `mcp__<server>__<tool>`, soumis aux mêmes règles de permission que les outils natifs.
- Écosystème riche (300+ serveurs), SDK pour créer les vôtres.

Suite

Au dernier cours : plugins, intégrations IDE, CI, et comparaison avec les outils concurrents.

Plugins, IDE, CI, et alternatives

Plugins : packager une configuration Claude Code

Un plugin regroupe et distribue, dans un seul artefact :

- des slash commands (`commands/`) ;
- des sous-agents (`agents/`) ;
- des hooks (`hooks/`) ;
- des MCP servers et leur config (`.mcp.json`).

Pour qui ?

Une équipe qui veut une configuration Claude Code cohérente (mêmes commandes, mêmes hooks de sécurité, mêmes MCP). Plutôt qu'un `.claude/` versionné ad-hoc, on packagee comme un module installable.

Anatomie d'un plugin

Un manifest `.claude-plugin/plugin.json` décrit le contenu :

```
{
  "name": "review-toolkit",
  "version": "1.2.0",
  "description": "Sous-agents et commandes pour revue de code et audit sécurité",
  "author": "Equipe Plateforme",
  "components": {
    "agents": ["agents/security-review.md", "agents/perf-audit.md"],
    "commands": ["commands/check-pr.md", "commands/check-deps.md"],
    "hooks": ["hooks/block-secrets.json"]
  }
}
```

Distribution

- Marketplace public : registres officiels, partage communautaire.
- Repo Git privé : `/plugin install <git-url>`, idéal pour l'entreprise.
- Local : `claude -plugin-dir ./mon-plugin` pour développement.

Les composants d'un plugin sont namespacés : `/review-toolkit:check-pr` pour ne pas collisionner avec d'autres plugins.

Intégrations IDE

Claude Code dispose de plugins natifs pour les principaux IDE :

- VS Code (extension officielle) : sidebar avec chat, sélection → « review this », navigation des diffs.
- JetBrains (IntelliJ, PyCharm, WebStorm...) : sidebar équivalente, context menu, intégration LSP.
- Le moteur reste le CLI : l'IDE est une UI, les outils, permissions, hooks et MCP sont identiques.

Choix pratique : CLI pour les workflows programmables (Bash + outils), IDE pour l'inline suggest et le contexte d'éditeur.

Modes scriptables et CI

Plusieurs flags rendent Claude Code utilisable dans un pipeline :

- `-p "query"` : un seul prompt, sortie texte, pas d'interactif.
- `-output-format json` : sortie JSON ligne-par-ligne (parsable avec `jq`).
- `-dangerously-skip-permissions` : autoriser tous les outils (à n'utiliser qu'en sandbox / container).
- `-add-dir <chemin>` : étendre l'accès aux fichiers.

GitHub Actions : Claude reviewer automatique

```yaml
name: Claude Code review
on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    permissions:
      pull-requests: write
      contents: read
    steps:
      - uses: actions/checkout@v4
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
          mode: review
          comment_on_pr: true
```

Cas d'usage CI typiques

- Review automatique de PR : commentaire généré sur chaque pull request.
- Auto-fix de lint : Claude lit le rapport, propose un commit avec les corrections.
- Migration assistée : Claude refactore en arrière-plan une partie de la codebase, ouvre une PR.
- Génération de docs : à chaque release, met à jour le changelog et la doc API.

Paysage des assistants de codage agentiques

Claude Code n'est pas seul. Cinq alternatives majeures à connaître :

- Codex CLI (OpenAI)
- Aider
- Cursor
- Continue
- Cline

Codex CLI (OpenAI)

- Concurrent direct le plus comparable à Claude Code : CLI agentique, outils, mode interactif.
- Modèles GPT-5.5 et descendants. Compute use (clic/clavier), browser intégré, multi-agents en parallèle.
- Open source côté CLI (repo `openai/codex`), gratuit avec un compte ChatGPT (Plus / Pro / Team).
- Forte intégration avec l'IDE web Codex et l'app desktop.

Quand le préférer ?

Si vous êtes déjà dans l'écosystème OpenAI et que GPT-5.5 + Computer Use vous parlent. Le CLI est plus jeune que Claude Code, l'écosystème de plugins/MCP moins mature.

Aider

- Pair programmer en TUI open source, agnostique du fournisseur LLM (Claude, GPT, locaux via Ollama).
- Forte intégration Git : chaque modification = un commit auto.
- Léger, rapide, philosophie « Unix way » : un outil qui fait une chose bien.
- Pas de hooks ni de MCP : moins extensible que Claude Code.

Quand le préférer ?

Pour rester multi-LLM, ou pour piloter un modèle local. Pour des changements ciblés où la traçabilité Git compte.

Cursor

- Un IDE (fork de VS Code) avec un assistant agent intégré.
- Excellente UX inline : `Cmd+K` pour générer, `Cmd+L` pour chatter, complétion contextuelle puissante.
- Modèles propriétaires de Cursor + Claude / GPT en backend (selon plan).
- Pas vraiment scriptable en CLI : c'est un IDE, pas un outil composable.

Quand le préférer ?

Pour l'écriture de code interactive dans un éditeur. Mauvais choix pour la CI ou les workflows scriptés.

Continue

- Extension open source pour VS Code et JetBrains.
- Multi-LLM (Claude, GPT, Ollama, Llama via vLLM...).
- Slash commands et context providers configurables, similaire à Claude Code mais limité à l'IDE.
- Bonne option pour les équipes qui veulent un assistant IDE auto-hébergé.

Cline

- Extension VS Code agentique (« Claude Dev », désormais « Cline »).
- Très proche de Claude Code en philosophie (outils, plan/act modes), mais intégrée à l'éditeur.
- Open source ; consomme votre clé API Anthropic / OpenAI directement.
- Idéal si vous préférez ne pas quitter VS Code.

Tableau de positionnement

	Forme	LLM	Open source	Niche
Claude Code	CLI + IDE + CI	Claude	Non	Workflows complets, équipes, sécurité
Codex CLI	CLI + app + IDE	GPT	CLI seul	Écosystème OpenAI, computer use
Aider	TUI	Multi	Oui	Multi-LLM, traçabilité Git
Cursor	IDE (fork VS Code)	Multi	Non	UX inline IDE
Continue	Extension IDE	Multi	Oui	IDE auto-hébergé
Cline	Extension VS Code	Multi	Oui	Agent dans l'éditeur

Comment choisir ?

- Workflows complexes, équipe, CI, MCP, sécurité forte → Claude Code.
- Multi-LLM ou modèles locaux, écriture conversationnelle → Aider.
- Écriture de code en IDE en mode flow → Cursor (propriétaire) ou Continue (open).
- Agent dans VS Code avec sa propre clé API → Cline.
- Écosystème OpenAI déjà en place → Codex CLI.

Combiner plutôt que choisir

Beaucoup d'équipes en utilisent deux : un IDE-natif (Cursor / Cline) pour l'écriture, un CLI agentique (Claude Code / Codex) pour les workflows.

Récapitulatif final

Au cours de ces huit chapitres, on a vu :

1. L'écosystème Claude Code et sa boucle agentique.
2. Installation, login, première session.
3. Mémoire (`CLAUDE.md` global et projet).
4. Outils et permissions (modes, règles fines).
5. Sous-agents et slash commands.
6. Hooks (validation, journalisation, blocage).
7. MCP servers (intégrations externes via protocole standard).
8. Plugins, IDE, CI, et le paysage concurrent.

À vous

Configurez votre propre `CLAUDE.md`, ajoutez un MCP, écrivez un hook. La pratique est la seule manière d'intérioriser ces outils.

Merci.

Questions ?