

Module 5 — Hooks et Automatisation

Jour 2 — 13 hooks, commands custom

The Bearded Bear

Février 2026



Table des matières

Module 5 : Hooks et Automatisation (1h30)	3
Objectifs	3
1. Qu'est-ce qu'un Hook?	3
Definition	3
Pourquoi les hooks?	3
Architecture	3
2. Les 13 Evenements Hooks	4
Liste complete	4
Cycle de vie d'un outil	5
Cycle de vie d'une session	5
3. Configuration dans settings.json	6
Emplacement	6
Structure d'un hook	6
Variables d'environnement disponibles	7
4. Matchers	7
Matchers par outil	7
Matchers specialises	7
5. Regles de Fonctionnement	9
Timeout	9
Sorties stdout et stderr	10
Exit codes	10
6. Hooks Prompt-Based	11
Concept	11
Configuration	11
Fonctionnement	11
Avantages vs hooks shell	12
7. Cas d'Usage Concrets	12
Lint automatique apres ecriture	12
Securite PreToolUse :Bash	13
Notifications equipe	13
Re-injection contexte apres compaction	14
Backup pre-compaction	14
8. Slash Commands Personnalisees	15
Concept	15
Structure	15
Creation d'une commande	15
Variables disponibles	16
Commandes d'equipe	16
9. Combinaison Hooks + Commands	16
Workflow integre	16
Exemple complet de settings.json	17

Exercice Pratique (20min)	18
Partie 1 : Hook de securite (10min)	18
Partie 2 : Commande custom (10min)	18
Verification	18
Points Cles a Retenir	19

Module 5 : Hooks et Automatisation (1h30)

Objectifs

A la fin de ce module, vous serez capable de : - Configurer les 13 evenements hooks disponibles dans Claude Code - Utiliser les matchers pour cibler des outils specifiques - Creer des hooks de securite, de qualite et de notification - Maitriser les hooks prompt-based avec Haiku 4.5 - Creer des slash commands personnalisees pour votre equipe

1. Qu'est-ce qu'un Hook ?

Definition

Les **Hooks** sont des commandes shell qui s'executent automatiquement lors d'evenements specifiques dans Claude Code. Ils transforment Claude Code d'un simple assistant interactif en une plateforme d'automatisation puissante.

Pourquoi les hooks ?

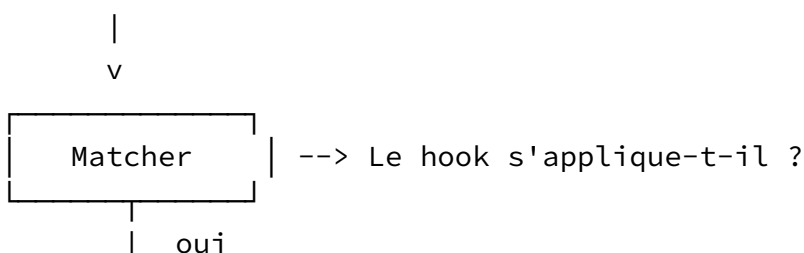
Il existe une distinction fondamentale entre les instructions textuelles et les hooks :

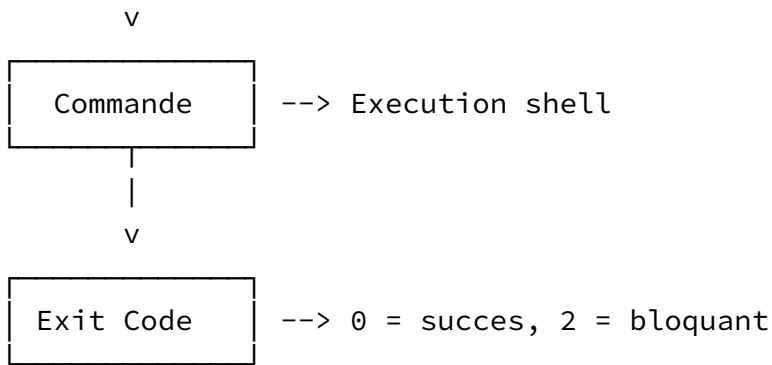
Mecanisme	Force	Comportement
CLAUDE.md	Suggestion	Claude peut ignorer si le contexte change
Rules	Suggestion forte	Priorise mais pas garanti
Hooks	Enforcement	Execution automatique, bloquant si configure

Regle d'or : CLAUDE.md = suggestions. Hooks = requirements. Pour toute contrainte critique (securite, qualite, conventions), utilisez des hooks.

Architecture

Evenement Claude Code





2. Les 13 Evenements Hooks

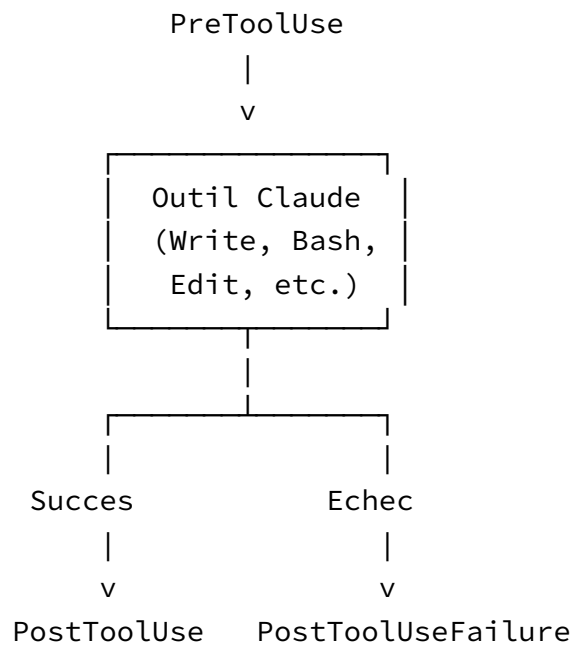
Liste complete

Attention : Certaines documentations non officielles mentionnent des evenements fictifs comme `PreWrite`, `PostWrite`, `PreBash`, `PostBash`, `Error` ou `TokenThreshold`. Voici la liste **exacte et complete** des 13 evenements supports :

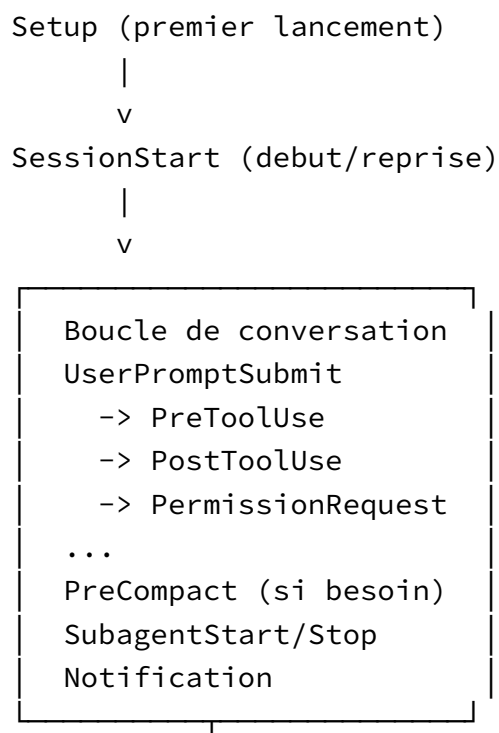
#	Evenement	Declencheur	Usage typique
1	<code>PreToolUse</code>	Avant execution d'un outil	Validation, securite, backup
2	<code>PostToolUse</code>	Apres succes d'un outil	Lint, format, notification
3	<code>PostToolUseFailure</code>	Apres echec d'un outil	Auto-recovery, logging erreurs
4	<code>PermissionRequest</code>	Demande de permission	Controle d'accès automatisé
5	<code>UserPromptSubmit</code>	Soumission prompt utilisateur	Transformation, validation input
6	<code>Stop</code>	Fin de reponse Claude	Cleanup, rapport de session
7	<code>SubagentStop</code>	Fin d'un sous-agent	Agregation resultats
8	<code>SubagentStart</code>	Lancement d'un sous-agent	Logging, configuration contexte
9	<code>Notification</code>	Conditions d'alerte	Alertes Slack/Teams
10	<code>PreCompact</code>	Avant compaction contexte	Sauvegarde etat, checkpoint
11	<code>SessionStart</code>	Debut/reprise de session	Initialisation environnement
12	<code>SessionEnd</code>	Fin de session	Cleanup, statistiques
13	<code>Setup</code>	Premier lancement projet	Installation dependances
14	<code>TeammateIdle</code>	Teammate inactif (Agent Teams)	Reassignment, alerting

#	Evenement	Declencheur	Usage typique
15	TaskCompleted	Tache terminee (Agent Teams)	Chainage, notification

Cycle de vie d'un outil



Cycle de vie d'une session



```

      |
      v
    Stop (fin reponse)
      |
      v
  SessionEnd (fin session)

```

3. Configuration dans settings.json

Emplacement

Les hooks se configurent dans `.claude/settings.json` a la racine du projet :

```

{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Write",
        "command": "echo 'Un fichier va etre modifie'"
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Bash",
        "command": "echo 'Commande executee avec succes'"
      }
    ]
  }
}

```

Structure d'un hook

Chaque hook est un objet avec les proprietes suivantes :

```

{
  "matcher": "NomDeLOutil",
  "command": "commande_shell_a_executer"
}

```

Propriete	Description	Obligatoire
matcher	Pattern pour filtrer les evenements	Non (sans matcher = tous les evenements)
command	Commande shell a executer	Oui

Propriete	Description	Obligatoire
-----------	-------------	-------------

Variables d'environnement disponibles

Chaque hook recoit des variables d'environnement contextuelles :

```
$TOOL_NAME      # Nom de l'outil (Write, Bash, Edit, etc.)
$TOOL_INPUT     # Input JSON de l'outil (parametres)
$TOOL_OUTPUT    # Output de l'outil (PostToolUse uniquement)
$SESSION_ID     # Identifiant de la session
$PROJECT_DIR    # Repertoire du projet
```

4. Matchers

Matchers par outil

Les matchers les plus courants ciblent les outils Claude Code :

Matcher	Outil cible	Exemple d'usage
Write	Ecriture de fichier	Lint apres creation
Edit	Modification de fichier	Format apres edition
Bash	Execution de commande	Securite, validation
Read	Lecture de fichier	Logging d'accès
Glob	Recherche de fichiers	Audit
Grep	Recherche dans le contenu	Audit
WebFetch	Requete HTTP	Filtrage URLs

Matchers specialises

Certains evenements disposent de matchers specifiques qui permettent un ciblage plus fin :

Notification

Matcher	Declencheur
permission_prompt	Claude demande une permission a l'utilisateur

Matcher	Declencheur
idle_prompt	Claude attend une reponse de l'utilisateur

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "permission_prompt",
        "command": "curl -s -X POST https://hooks.slack.com/services/xxx -d
↪ '{"text\\": \"Claude attend votre permission\\\"}'"
      },
      {
        "matcher": "idle_prompt",
        "command": "notify-send 'Claude Code' 'Claude attend votre reponse'"
      }
    ]
  }
}
```

PreCompact

Matcher	Declencheur
manual	Compaction declenchee par /compact
auto	Compaction automatique (limite contexte)

```
{
  "hooks": {
    "PreCompact": [
      {
        "matcher": "auto",
        "command": "cp .claude/context-essentials.md
↪ /tmp/backup-context-$(date +%s).md"
      }
    ]
  }
}
```

SessionStart

Matcher	Declencheur
startup	Nouvelle session (premier lancement)
resume	Reprise de session existante (--resume)

Matcher	Declencheur
clear	Après /clear
compact	Après compaction du contexte

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "compact",
        "command": "cat .claude/context-essentials.md"
      },
      {
        "matcher": "startup",
        "command": "echo 'Bienvenue ! Projet: $(basename $PROJECT_DIR)'"
      }
    ]
  }
}
```

Setup

Matcher	Declencheur
init	Premier lancement dans le projet
maintenance	Lancement avec --init ou --maintenance

```
{
  "hooks": {
    "Setup": [
      {
        "matcher": "init",
        "command": "npm install && echo 'Dependances installees'"
      }
    ]
  }
}
```

5. Regles de Fonctionnement

Timeout

Chaque hook dispose d'un **timeout de 10 minutes**. Au-dela, le hook est automatiquement interrompu. Cela évite les hooks qui bloquent indéfiniment la session.

Sorties stdout et stderr

Sortie	Qui la voit	Usage
stdout	Claude (le modele)	Informations que Claude doit prendre en compte
stderr	L'utilisateur (terminal)	Messages de feedback pour le developpeur

Exemple pratique :

```
#!/bin/bash
# Hook PostToolUse:Write - Lint automatique

FILE=$(echo "$TOOL_INPUT" | jq -r '.file_path')

# stdout -> Claude voit le resultat
echo "Lint execute sur $FILE"

# stderr -> L'utilisateur voit dans le terminal
echo "Verification qualite en cours..." >&2

# Executer le linter
npx eslint "$FILE" 2>&1
```

Exit codes

Exit code	Signification	Effet
0	Succes	Le workflow continue normalement
1	Echec non bloquant	Claude est informe mais continue
2	Echec bloquant	L'operation est annulee

Important : L'exit code 2 est le mecanisme de blocage. C'est ce qui rend les hooks plus puissants que les instructions textuelles.

Exemple de hook bloquant :

```
#!/bin/bash
# Hook PreToolUse:Bash - Bloquer les commandes dangereuses

COMMAND=$(echo "$TOOL_INPUT" | jq -r '.command')
```

```
# Verifier les commandes dangereuses
if echo "$COMMAND" | grep -qE 'rm\s+-rf\s+/\|dd\s+if=|mkfs|:(){ :|:& };: ';
  then
    echo "BLOQUE: Commande dangereuse detectee: $COMMAND" >&2
    exit 2 # Bloquant : l'execution est annulee
  fi

exit 0 # OK : laisser passer
```

6. Hooks Prompt-Based

Concept

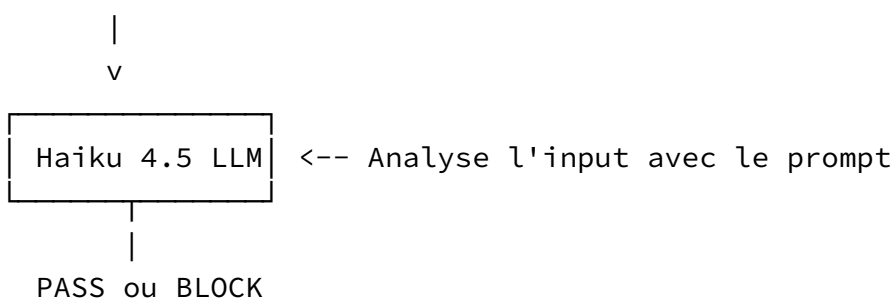
Au lieu d'exécuter une commande shell, un hook prompt-based envoie un prompt à un modèle rapide (Haiku 4.5) pour analyser la situation. Cela permet des validations plus intelligentes qu'un simple grep.

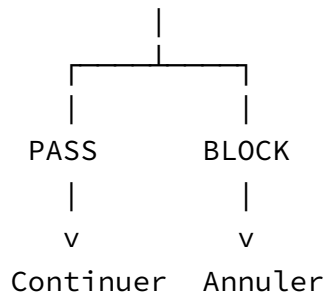
Configuration

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Write",
        "type": "prompt",
        "prompt": "Verifie que le fichier qui va etre ecrit ne contient pas
          ↳ de secrets (API keys, mots de passe en dur, tokens). Si un
          ↳ secret est detecte, reponds BLOCK sinon reponds PASS."
      }
    ]
  }
}
```

Fonctionnement

Evenement PreToolUse:Write





Avantages vs hooks shell

Aspect	Hook Shell	Hook Prompt
Intelligence	Pattern matching basique (regex)	Comprehension semantique
Faux positifs	Frequents	Rares
Complexite	Script a maintenir	Prompt en langage naturel
Latence	Rapide (~100ms)	Plus lent (~1-2s, appel LLM)
Cout	Gratuit	Tokens Haiku 4.5 (tres faible)

7. Cas d'Usage Concrets

Lint automatique apres ecriture

```

{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write",
        "command": "FILE=$(echo $TOOL_INPUT | jq -r '.file_path'); if [[
          ↪ \"$FILE\" == *.js || \"$FILE\" == *.ts ]]; then npx eslint --fix
          ↪ \"$FILE\" 2>/dev/null; fi"
      },
      {
        "matcher": "Write",
        "command": "FILE=$(echo $TOOL_INPUT | jq -r '.file_path'); if [[
          ↪ \"$FILE\" == *.py ]]; then python -m black \"$FILE\"
          ↪ 2>/dev/null; fi"
      }
    ]
  }
}

```

Securite PreToolUse :Bash

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "command": "COMMAND=$(echo $TOOL_INPUT | jq -r '.command'); if echo
        ↪ \"$COMMAND\" | grep -qE
        ↪ '(curl|wget).*\\.(sh|py|rb)\\s*\\|\\s*(bash|sh|python)'; then
        ↪ echo 'BLOQUE: Telechargement et execution de script detecte'
        ↪ >&2; exit 2; fi; exit 0"
      },
      {
        "matcher": "Bash",
        "command": "COMMAND=$(echo $TOOL_INPUT | jq -r '.command'); if echo
        ↪ \"$COMMAND\" | grep -qE 'rm\\s+-rf\\s+(\\/|~|\\$HOME)'; then
        ↪ echo 'BLOQUE: Suppression recursive dangereuse' >&2; exit 2; fi;
        ↪ exit 0"
      }
    ]
  }
}
```

Notifications equipe

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "permission_prompt",
        "command": "curl -s -X POST
        ↪ 'https://hooks.slack.com/services/T00/B00/xxx' -H 'Content-Type:
        ↪ application/json' -d '{"text": "Claude Code attend une
        ↪ permission sur le projet '$(basename $PROJECT_DIR)\\\"}'"
      }
    ],
    "Stop": [
      {
        "command": "echo 'Session terminee a $(date)' >>
        ↪ .claude/session-log.txt"
      }
    ]
  }
}
```

Re-injection contexte apres compaction

Ce hook est particulierement important pour les sessions longues. Quand le contexte est compacte automatiquement, des informations critiques peuvent etre perdues. Le hook `SessionStart:compact` permet de les re-injecter :

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "compact",
        "command": "cat .claude/context-essentials.md"
      }
    ]
  }
}
```

Le fichier `.claude/context-essentials.md` contient :

```
# Contexte Essentiel

## Decisions architecturales en cours
- Pattern CQRS adopte pour le module Orders
- Migration vers PostgreSQL 16 en cours

## Tache en cours
- Feature: Systeme de notifications
- Branche: feature/notifications
- Fichiers principaux: src/notification/

## Contraintes
- Ne pas modifier l'API publique v1
- Tests obligatoires pour chaque endpoint
```

Backup pre-compaction

```
{
  "hooks": {
    "PreCompact": [
      {
        "matcher": "auto",
        "command": "echo '## Backup contexte - '$(date) >>
          ↪ .claude/compaction-log.md"
      }
    ]
  }
}
```

8. Slash Commands Personnalisées

Concept

Les slash commands sont des fichiers Markdown dans `.claude/commands/` qui deviennent des commandes disponibles dans Claude Code. Elles permettent de créer des workflows réutilisables pour votre équipe.

Structure

```
.claude/
  commands/
    audit.md          -> /project:audit
    review.md         -> /project:review
    deploy-check.md   -> /project:deploy-check
```

Création d'une commande

Chaque fichier `.md` dans `.claude/commands/` devient une commande accessible via `/project:<nom-du-fichier>`.

Exemple : `/project:audit` Fichier `.claude/commands/audit.md`:

Realise un audit complet du projet en suivant ces etapes :

1. **Architecture** : Vérifie la structure des répertoires et la séparation des couches
2. **Qualité** : Lance les linters et analyseurs statiques
3. **Tests** : Vérifie la couverture et la qualité des tests
4. **Sécurité** : Cherche les vulnérabilités connues et les secrets en dur
5. **Documentation** : Vérifie que la documentation est à jour

Pour chaque étape, donne un score sur 10 et liste les problèmes trouvés.

Termine par un rapport consolidé avec un plan d'action priorisé.

`$ARGUMENTS`

Exemple : `/project:review` Fichier `.claude/commands/review.md`:

Effectue une code review du fichier ou du diff fourni.

Critères de review :

- Respect des principes SOLID
- Couverture de tests

- Securite (OWASP Top 10)
- Performance
- Lisibilite et maintenabilite
- Conventions de nommage

Format de sortie :

- Liste des problemes par severite (critique, majeur, mineur)
- Suggestions d'amelioration
- Points positifs a souligner

\$ARGUMENTS

Variables disponibles

Variable	Description	Exemple
\$ARGUMENTS	Texte passe apres la commande	/project:audit src/services/ -> src/services/
\$SELECTION	Code selectionne dans l'IDE (VS Code)	Le bloc de code surligne

Commandes d'équipe

Les commandes dans `.claude/commands/` sont versionnees avec le projet. Toute l'équipe y a acces en faisant un `git pull`.

Exemples de commandes utiles pour une equipe :

```
.claude/commands/
onboard.md          # Guide d'onboarding nouveau membre
debug-db.md         # Workflow debug base de donnees
deploy-check.md     # Checklist pre-deploiement
feature-scaffold.md # Scaffolding nouvelle feature
incident.md         # Procedure incident production
migrate.md          # Guide migration de schema
```

9. Combinaison Hooks + Commands

Workflow integre

Les hooks et les commandes se combinent pour creer des workflows puissants :

1. Le developpeur lance `/project:feature-scaffold "notifications"`
-> Claude genere la structure
2. Hook `PostToolUse:Write` se declenche
-> Lint automatique sur chaque fichier cree
3. Hook `PreToolUse:Bash` se declenche
-> Verification securite avant chaque commande
4. Le developpeur lance `/project:review`
-> Claude effectue la review
5. Hook `Stop` se declenche
-> Log de la session

Exemple complet de settings.json

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "command": "COMMAND=$(echo $TOOL_INPUT | jq -r '.command'); if echo
          ↪ \"$COMMAND\" | grep -qE 'rm\\s+-rf'; then echo 'BLOQUE: rm -rf
          ↪ interdit' >&2; exit 2; fi; exit 0"
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Write",
        "command": "FILE=$(echo $TOOL_INPUT | jq -r '.file_path');
          ↪ EXT=${FILE##*.}; case $EXT in js|ts) npx eslint --fix \"$FILE\"
          ↪ 2>/dev/null;; py) python -m black \"$FILE\" 2>/dev/null;; esac"
      }
    ],
    "SessionStart": [
      {
        "matcher": "compact",
        "command": "cat .claude/context-essentials.md"
      },
      {
        "matcher": "startup",
        "command": "echo 'Projet: '$(basename $PROJECT_DIR)' | Branche:
          ↪ '$(git branch --show-current 2>/dev/null || echo 'N/A')'"
      }
    ],
    "Notification": [
```

```

    {
      "matcher": "idle_prompt",
      "command": "notify-send 'Claude Code' 'En attente de votre reponse'
        ↪ 2>/dev/null || true"
    }
  ],
  "Stop": [
    {
      "command": "echo \"$(date): session stop\" >> .claude/activity.log"
    }
  ]
}

```

Exercice Pratique (20min)

Partie 1 : Hook de securite (10min)

Creez un hook `PreToolUse: Bash` qui bloque les commandes suivantes : `- rm -rf /` (et variantes avec `~`, `$HOME`) - `chmod 777` sur n'importe quel fichier - Telechargement et execution de scripts (`curl ... | bash`)

Testez en demandant a Claude d'executer ces commandes.

Fichier a creer/modifier : `.claude/settings.json`

Partie 2 : Commande custom (10min)

Creez une commande `/project:audit` qui demande a Claude de : 1. Lister la structure du projet 2. Identifier les fichiers sans tests correspondants 3. Chercher les TODO/FIXME dans le code 4. Donner un rapport avec un score global

Fichier a creer : `.claude/commands/audit.md`

Verification

- ☐ Le hook bloque `rm -rf /` avec exit code 2
- ☐ Le hook bloque `chmod 777`
- ☐ Le hook bloque `curl ... | bash`
- ☐ La commande `/project:audit` est accessible
- ☐ La commande produit un rapport structure

Points Cles a Retenir

1. **13 evenements hooks** couvrent tout le cycle de vie d'une session Claude Code
 2. **stdout -> Claude, stderr -> utilisateur** : separation claire des sorties
 3. **Exit code 2 = bloquant** : mecanisme d'enforcement le plus puissant
 4. **Hooks > CLAUDE.md** pour les contraintes critiques (securite, qualite)
 5. **Matchers specialises** (Notification, PreCompact, SessionStart, Setup) permettent un ciblage fin
 6. **Hooks prompt-based** utilisent Haiku 4.5 pour des validations intelligentes
 7. **Slash commands** dans `.claude/commands/` sont partagees avec toute l'equipe via Git
 8. **La combinaison hooks + commands** cree des workflows d'automatisation complets
-

Duree : 1h30 **Prochain module :** Module 6 : MCP et Integrations