Hooks in Claude Code

3 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=IKAPHIMDAZM
https://www.youtube.com/watch?v=IkaPHiMDazM

SUMMARY

Hooks provide a reliable way to run commands at specific points in the cloud code lifecycle, ensuring that actions are executed every time without exception. They can be configured in the settings JSON file and are essential for tasks like auto-formatting, logging, and blocking risky operations.

- *Hooks are deterministic and always run, unlike regular commands that may fail occasionally.*
- *Common use cases include auto-formatting after file edits, logging executed commands, blocking dangerous operations, and sending notifications.*
- *Hooks are set up in the settings JSON file, where you select events and commands to execute.*
- *Pre tool use hooks can block operations based on specific conditions, enforcing strict rules.*
- *Post tool use hooks are ideal for tasks like auto-formatting and logging after tool execution.*
- *Hooks are project-level configurations that can be shared across teams by checking them into the repository.*
- *The `claude projectdir` environment variable allows commands to reference project scripts, ensuring compatibility regardless of the working directory.*
- *For guaranteed execution of tasks, use hooks instead of prompts.*

Speaker 1: Hooks let you run commands at different points in cloud code's lifecycle. The key difference between hooks and everything else we covered is that hooks are deterministic. They always run. So put it this way, you can tell Claude in your claude MD file to run prettier after every every file edit, and most of the time it will do that. But sometimes it won't. It's not perfect, but a hook makes it happen every single time with no exceptions.

Speaker 1: Common use cases could include auto formatting after file edits, logging all executed commands for compliance, blocking dangerous operations like modifying production files, and sending yourself notifications. When CLAUDE finishes a task, hooks are configured in your settings JSON file. You pick an event, optionally set a matcher for which tools it applies to, and provide a command to run user prompt Submit runs When you submit a prompt before Claude processes it. Pre tool use, which runs before a tool call, Post tool use runs after a tool call, completes notification, runs when claude sends a notification, and stop runs when Claude finishes responding.

Speaker 1: The most common hook Auto formatting After edit, you set a post tool use hook with a matcher of edit or multi edit right so it fires whenever claude modifies a file, the command checks the file extension and runs the appropriate formatter. This could be prettier for typescript go format for go rough for python whatever your project uses pre tool use Hooks can block tool calls before they execute, so your hook receives a tool name and input as JSON on stdin.

Speaker 1: If it exits with code 2, the action is blocked and the stderror message gets fed back to Claude as feedback, so claude knows why it was blocked and can adjust. Exit code zero means proceed. Exit code two means block. This is how you enforce hard rules. Block writes to a production config directory. Block bash commands that contain RM rf Block commits to main whatever your team needs to be guaranteed, not suggested. Hooks configured in claude settings JSON are project level and can be checked into your repo.

Speaker 1: This means that your entire team gets the same. Hooks automatically use the claude projectdir environment variable and your commands to reference scripts stored in your project so they work regardless of claude's current working directory. Hooks gives you deterministic control over cloud code's behavior. Use post tool use for auto formatting and logging. Use pre tool use to block dangerous operations. Configure them in the slash hooks or in settings JSON and check them into your repository so your team gets them too.

Speaker 1: If something needs to happen every time without fail, don't put it in a prompt, put it in a hook.