ChatGPT

# Claude Code CLI Hook System: Comprehensive Reference and Examples

Claude Code CLI supports a **hook system** that lets you inject custom behavior at various points in a coding session's lifecycle. Hooks are configured in your Claude settings (e.g. `~/.claude/settings.json`) and are executed as external scripts or commands when specific events occur [1] [2]. The CLI passes a JSON payload to each hook via STDIN and uses the hook's exit code and output to decide how to proceed [3]. This ensures certain tasks (like linting, testing, etc.) run deterministically rather than relying on the AI to remember them [4] [5]. Below, we detail each available hook event, including its purpose, input/output structure, effects on the session, and a Node.js example demonstrating its use.

## PreToolUse Hook

**Purpose & Behavior:** The **PreToolUse** hook runs *after Claude formulates a tool's parameters but before the tool is actually executed* [6]. This allows you to intercept tool usage requests (e.g. file edits, shell commands, etc.) and programmatically decide whether to allow them, deny them, or alter their inputs. It's commonly used to enforce policies or auto-approve certain tool actions. For example, you might use PreToolUse to require approval for dangerous shell commands or to auto-adjust parameters (like filtering a command) before execution.

**Input:** The hook receives a JSON object (via STDIN) with common fields and tool-specific info. Key fields include:

- `session_id` (string) – The unique session identifier.
- `transcript_path` (string) – Path to the conversation log file (JSONL).
- `cwd` (string) – Current working directory when the hook is invoked.
- `permission_mode` (string) – Current permission mode (e.g. `"default"`, `"plan"`, etc.).
- `hook_event_name` (string) – The event name (`"PreToolUse"` in this case).
- `tool_name` (string) – Name of the tool Claude intends to use (e.g. `"Write"`, `"Bash"`, `"Edit"`, etc.).
- `tool_input` (object) – The parameters for the tool call (structure varies by tool; e.g. for `Write`, it might contain a `file_path` and `content` [7]).
- `tool_use_id` (string) – Unique ID for this tool invocation.

**Output & Effects:** A PreToolUse hook can control the tool execution in several ways:

- **Allow or Deny the tool:** By returning a JSON with `hookSpecificOutput.permissionDecision`, the hook can **auto-approve or deny** the tool call. For example, `"permissionDecision": "allow"` will bypass the usual permission prompt (if any) and let the tool run, whereas `"deny"` will prevent the tool from executing [8]. In a denial, you can provide a `permissionDecisionReason` that will be shown to Claude as the reason for the denial [9]. There is also an `"ask"` option to explicitly trigger a user confirmation dialog [10].

- **Modify tool input:** The hook can alter the tool's parameters by returning an `updatedInput` object in the JSON output [11]. This allows you to sanitize or tweak the tool arguments before execution (often combined with an `"allow"` decision to auto-approve the modified call).
- **No output (exit 0):** If the hook exits with code 0 and produces no JSON output, the tool call proceeds normally under the standard permission system (i.e. the user may be prompted for approval if required).
- **Blocking with error (exit code 2):** If the hook exits with code 2, Claude Code will *block the tool call entirely*. The `stderr` from the hook is treated as an error message and fed back to Claude as the reason for blocking [12]. This is a way to enforce a hard refusal (e.g. if a policy is violated) – the model will receive the error and not execute the tool.

**Example – Auto-Replace and Block Commands:** The Node.js script below demonstrates a PreToolUse hook that (1) blocks any attempt to run a `rm` command in the Bash tool (to prevent accidental file deletion), and (2) automatically replaces the `grep` command with a more efficient `rg` (ripgrep) command, auto-allowing that change. It uses all provided inputs (checking the `tool_name` and `tool_input.command`) and shows how to output a JSON decision with `updatedInput` for the modification:

```
#!/usr/bin/env node
const fs = require('fs');  // for reading hook input from stdin

/**
 * Handle PreToolUse hook: enforce safe commands and adjust inputs.
 * @param {Object} input - JSON payload for the PreToolUse event.
 * @param {string} input.session_id - Claude session ID.
 * @param {string} input.transcript_path - Path to the conversation log file.
 * @param {string} input.cwd - Current working directory of Claude Code.
 * @param {string} input.permission_mode - Current permission mode (e.g.
"default").
 * @param {string} input.hook_event_name - Name of the event ("PreToolUse").
 * @param {string} input.tool_name - Name of the tool about to be used (e.g.
"Bash").
 * @param {Object} input.tool_input - Parameters for the tool (varies by
tool).
 * @param {string} input.tool_use_id - Unique identifier for this tool
invocation.
 * @returns {number} Exit code (0 = proceed, 2 = block with error).
 */
function handlePreToolUse(input) {
  // Only intervene for Bash commands in this example
  if (input.tool_name === 'Bash' && typeof input.tool_input.command ===
'string') {
    const cmd = input.tool_input.command;
    // 1. Block any 'rm' command for safety
    if (/\\brm\\s/.test(cmd)) {
      console.error('Blocked: unsafe "rm" command is not allowed.');
      return 2;  // Exit code 2 blocks the tool execution
    }
    // 2. If using 'grep', replace it with 'rg' (ripgrep) for better
performance
```

```
    if (cmd.includes('grep')) {
      const newCmd = cmd.replace(/\\bgrep\\b/, 'rg');
      input.tool_input.command = newCmd;
      // Output JSON instructing Claude to allow the modified command
      const output = {
        hookSpecificOutput: {
          hookEventName: 'PreToolUse',
          permissionDecision: 'allow',
          permissionDecisionReason: 'Replaced grep with rg for efficiency',
          updatedInput: input.tool_input   // use the modified command
        }
      };
      console.log(JSON.stringify(output));
      return 0;  // Exit 0 with JSON means tool will run with updated input
    }
  }
  // Default: no intervention for other tools/commands
  return 0;
}

// Parse input from stdin and execute the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handlePreToolUse(input));
```

*Explanation:* In this PreToolUse hook, any Bash command containing `rm` is blocked by returning exit code 2 with an error on **stderr** (which Claude will receive, causing it not to run the command). If the Bash command uses `grep`, the script modifies the command to use `rg` instead, outputs a JSON object with `permissionDecision: "allow"` and the `updatedInput`, and exits 0. This causes Claude to proceed with the revised command **without asking the user for permission** (the hook auto-approved it) [8] [11]. All other tool usages pass through unaffected.

## PermissionRequest Hook

**Purpose & Behavior:** The **PermissionRequest** hook runs whenever Claude is about to present a permission dialog to the user [13]. In Claude Code's default mode, certain tool uses (especially those that could modify files, run shell commands, etc.) trigger a confirmation prompt. This hook gives you a chance to automatically handle those prompts. It's typically used to auto-allow trusted actions (to streamline workflow) or auto-deny disallowed actions, without user intervention. The PermissionRequest hook sees the same tool invocation details as PreToolUse, but at the moment a permission is being requested.

**Input:** The JSON input for PermissionRequest includes the common fields (`session_id`, `cwd`, etc.) and details of the pending permission prompt. Important fields:

- `tool_name` (string) – The name of the tool for which permission is being requested (e.g. `"Bash"`, `"Write"`, etc.).
- `tool_input` (object) – The parameters of the tool call (same as in PreToolUse). For instance, if Claude wants to run a Bash command, this will include the `command` string.
- *(The payload may also include the* `tool_use_id` *and a human-readable permission prompt message, but the key info for logic is the tool and its inputs.)*

**Output & Effects:** A PermissionRequest hook can programmatically decide the outcome of the permission dialog:

- **Auto-allow the action:** To grant permission without user input, output a JSON with `hookSpecificOutput.decision.behavior: "allow"`. You can also modify the tool's parameters by including an `updatedInput` under `decision` if needed [14]. Claude will then proceed to run the tool immediately as if the user clicked "Allow".
- **Auto-deny the action:** To refuse permission, output JSON with `behavior: "deny"`. You may provide a `message` explaining why it's denied (this message is given to the model as context for the refusal) and an `interrupt` flag [15]. If `interrupt: true`, Claude will stop execution entirely after the denial; if false, Claude will simply not run that tool and continue the session.
- **Do nothing (no output):** If the hook exits 0 without producing any output, Claude Code will proceed to show the permission prompt to the user as normal. The user can then manually allow or deny.
- **Exit code 2:** If the hook exits with code 2, it is treated similar to a denial – the permission is not granted, and the `stderr` from the hook is shown as an error message to Claude (the model) [12]. Essentially, this auto-denies the request with an error context.

**Example – Auto-Allow Safe Commands, Deny Dangerous Ones:** The Node.js example below implements a PermissionRequest hook that automatically allows benign shell commands and denies any command that attempts file deletion. It examines the incoming `tool_name` and `tool_input` (here focusing on Bash commands) and uses the appropriate JSON outputs to respond to Claude's permission query:

```javascript
#!/usr/bin/env node
const fs = require('fs');

/**
 * Handle PermissionRequest hook: auto-approve or deny tool permissions.
 * @param {Object} input - JSON payload for the PermissionRequest event.
 * @param {string} input.session_id - Claude session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Permission mode (e.g. "default").
 * @param {string} input.hook_event_name - Event name ("PermissionRequest").
 * @param {string} input.tool_name - Tool requesting permission (e.g.
"Bash").
 * @param {Object} input.tool_input - Parameters of the tool call.
 * @param {string} input.tool_use_id - Unique tool invocation ID.
 * @returns {number} Exit code (0 to continue, typically).
 */
function handlePermissionRequest(input) {
  if (input.tool_name === 'Bash') {
    const cmd = input.tool_input.command || '';
    // Deny any Bash command attempting to remove files
    if (/\\brm\\s/.test(cmd)) {
      const output = {
        hookSpecificOutput: {
          hookEventName: 'PermissionRequest',
          decision: {
            behavior: 'deny',
            message: 'Deleting files is not permitted by policy.',
            interrupt:
false  // don't stop the whole session, just deny this action
          }
        }
      };
      console.log(JSON.stringify(output));
      return 0;  // exit 0 with JSON to deny the permission request
    } else {
      // Auto-allow other Bash commands (e.g. simple read-only operations)
      const output = {
        hookSpecificOutput: {
          hookEventName: 'PermissionRequest',
          decision: {
            behavior: 'allow',
            updatedInput:
input.tool_input  // proceed with the original tool input
          }
        }
      };
      console.log(JSON.stringify(output));
      return 0;  // exit 0 with JSON to allow the action
    }
  }
  // For other tools, don't auto-decide – let the user handle it
  return 0;
}

// Run the handler with input from stdin
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handlePermissionRequest(input));
```

*Explanation:* In this PermissionRequest hook example, when Claude asks for permission to run a Bash command, the script inspects the command. If the command contains `rm` (which could delete files), the hook responds with a JSON decision to **deny** the request, supplying a reason for the denial [15]. This refusal is passed to Claude so it understands the action is disallowed. For other Bash commands (deemed safe), the hook auto-**allows** them by returning a JSON decision with `behavior: "allow"` [14]. The `updatedInput` here just echoes the same input (no changes), but is included to illustrate the field. In both cases, the hook exits with code 0 after printing the JSON, indicating a successful handling of the permission dialog. If the tool was something other than Bash, the hook does nothing, so the user would see the prompt and decide manually.

## PostToolUse Hook

**Purpose & Behavior:** The **PostToolUse** hook runs *immediately after a tool has finished executing (successfully)* [16]. It gives you a chance to react to the outcome of tool usage. This is useful for validating or processing the results of a tool and providing feedback or additional instructions to Claude. For example, after a "Write" tool modifies a file, a PostToolUse hook could run a linter or tests; after a "Bash" command runs, it could analyze the command's output. PostToolUse hooks can be filtered by tool name (using matchers) just like PreToolUse/PermissionRequest [17].

**Input:** The JSON input to PostToolUse includes both the tool's input and its result. Key fields:

- `tool_name` (string) – Which tool was used (e.g. `"Write"`, `"Edit"`, `"Bash"`, etc.).
- `tool_input` (object) – The parameters that were given to the tool (same format as PreToolUse). For instance, for a `Write` tool this would include the `file_path` and new content that was written [18].
- `tool_response` (object) – The result returned by the tool. The structure depends on the tool: for a file operation it might include a success flag or new file path; for a Bash command it might contain the exit code and output text. (In the docs example for `Write`, `tool_response` has a `success: true` flag and the file path [19].)
- `tool_use_id` (string) – The unique ID of that tool execution (same ID as seen in PreToolUse/ PermissionRequest for this call).

**Output & Effects:** A PostToolUse hook can influence Claude's next steps or provide info:

- **Suggest corrections (block Claude's conclusion):** The hook can determine something went wrong or needs attention and request Claude to continue working on it. To do this, output a JSON with `"decision": "block"` and a `reason` explaining the issue [20]. Marking a post-tool step as "blocked" means *Claude will not consider the task finished*. Instead, the model will receive a system prompt with the given reason, prompting it to address the issue. For example, if tests run by a Bash command failed, you might block and give a reason like "Tests failed, please fix the errors," causing Claude to continue coding.
- **Provide additional context:** The hook can also supply extra context to inform Claude's next response. Include `hookSpecificOutput.additionalContext` in a JSON output (with exit 0) to inject a piece of text for Claude to consider [21]. Unlike a block reason, this doesn't force Claude to continue, but augments its information (for example, providing logs or metrics from the tool output).
- **No output:** If the hook exits normally (0) without printing JSON, Claude will carry on as usual with whatever it was doing. Essentially, the tool's result is accepted and the conversation proceeds.
- **Error exit (code 2):** If the hook exits with 2, the tool had already run, so you can't undo it, but Claude will be shown the hook's stderr as an error message [22]. This could be used to inform the model of an issue post hoc. (Typically, using the JSON "block" approach is preferred for instructing Claude to handle errors.)

**Example – Enforce Test Success:** In this Node.js PostToolUse hook example, we assume Claude sometimes runs a test suite via a Bash command (e.g. `npm test`). The hook checks the outcome: if tests failed, it blocks Claude from considering the task complete and provides a reason so that Claude will attempt to fix the issues. Otherwise, it does nothing. This demonstrates reading both `tool_input` and `tool_response` to decide on further action:

```
#!/usr/bin/env node
const fs = require('fs');

/**
 * Handle PostToolUse hook: react to tool results (e.g. test outcomes).
 * @param {Object} input - JSON payload for the PostToolUse event.
 * @param {string} input.session_id - Session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Permission mode (e.g. "default").
 * @param {string} input.hook_event_name - Event name ("PostToolUse").
 * @param {string} input.tool_name - Tool that was just used.
 * @param {Object} input.tool_input - Parameters that were passed to the
tool.
 * @param {Object} input.tool_response - The tool's output/response data.
 * @param {string} input.tool_use_id - Unique tool invocation ID.
 * @returns {number} Exit code.
 */
function handlePostToolUse(input) {
  if (input.tool_name === 'Bash') {
    const cmd = input.tool_input.command || '';
    // If the Bash command was a test run (e.g. "npm test"):
    if (cmd.includes('npm test') || cmd.includes('pytest') ||
cmd.includes('go test')) {
      // Check if tests passed or failed (assuming tool_response contains
exitCode or output text)
      const exitCode = input.tool_response.exitCode ?? 0;
      const outputText = input.tool_response.stdout ||
input.tool_response.output || '';
      const failed = exitCode !== 0 || /FAIL|Error/i.test(outputText);
      if (failed) {
        // Tests failed: block Claude from finishing and provide feedback
        const result = {
          decision: 'block',
          reason:
'Test suite failed. Please address the failing tests before proceeding.'
        };
        console.log(JSON.stringify(result));
        return 0;  // Claude will receive the reason and continue working on
it
      }
    }
  }
  // Default: no intervention if tool result is fine or not relevant
  return 0;
}

// Execute the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handlePostToolUse(input));
```

*Explanation:* In this hook, after any Bash command runs, we check if the command was likely a test run (`npm test` or similar). If so, we inspect the `tool_response` – here we assume it provides an `exitCode` or output text. If the tests failed (non-zero exit or output containing “FAIL”), the hook prints a JSON object with `"decision": "block"` and a reason [20], then exits 0. As a result, Claude will interpret that the coding work is not done; it will get the reason “Test suite failed...” and likely attempt to fix the failing tests instead of stopping. If tests passed (or if the tool was something else), the hook returns 0 with no output, so Claude proceeds normally. This way, the PostToolUse hook enforces that Claude only stops when tests succeed, implementing a basic continuous integration check.

## Notification Hook

**Purpose & Behavior:** The **Notification** hook runs whenever Claude Code emits a notification event in the UI [23]. Notifications are system messages like permission prompts, idle timeouts, authentication notices, etc., that aren't directly part of the user/Claude conversation but indicate some status. The Notification hook is useful for logging or reacting to these events externally (for example, sending desktop notifications or logging permission prompts for audit). You can filter notifications by type using match patterns. Common notification types include: `"permission_prompt"` (when Claude is asking for tool permission), `"idle_prompt"` (Claude has been idle waiting for input), `"auth_success"`, `"elicitation_dialog"`, etc. [24].

**Input:** The input JSON for a Notification hook includes the usual common fields and specifically:

- `message` (string) – The notification text/message. For example: `"Claude needs your permission to use Bash"` or `"Claude is waiting for your input."` [25].
- `notification_type` (string) – A code for the type of notification, e.g. `"permission_prompt"`, `"idle_prompt"`, etc. [25].

(There is no tool-specific data since this is not tied to a tool invocation, except that a permission_prompt indirectly relates to a tool request.)

**Output & Effects:** The Notification hook is generally used for side effects or user feedback, not for controlling Claude's behavior (there's no direct JSON schema to approve/deny here, aside from possibly using common fields like `systemMessage`). Key points:

- **Logging or alerts:** The hook can print output or perform actions like sending an alert. Any text you print to **stdout** will by default appear only in the user's verbose log (if opened), so for critical alerts you might want to handle them differently (e.g. write to a file or use other means to notify the user).
- **No decision control:** There is no `decision` field specific to Notification events. The hook cannot block a notification (notifications are informational). Using exit code 2 doesn't affect Claude – it will simply show the error text to the *user* in the interface (and not to Claude) [26]. Thus, exit codes are mostly useful for indicating hook failures in this context.
- **Side effects:** This hook is ideal for triggering external actions. For example, you might integrate with an OS notifier or log system. The hook runs asynchronously alongside the normal Claude flow (it doesn't alter what Claude does next).

**Example – User Alerts for Notifications:** The following Node.js script demonstrates a Notification hook that logs or alerts on specific notification types. It prints a message to stdout for permission prompts and idle prompts, which the user could see in verbose mode or later review. This shows how to utilize `notification_type` and `message` from the input:

```
#!/usr/bin/env node
const fs = require('fs');

/**
 * Handle Notification hook: log or alert on certain events.
 * @param {Object} input - JSON payload for the Notification event.
 * @param {string} input.session_id - Session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Current permission mode.
 * @param {string} input.hook_event_name - Event name ("Notification").
 * @param {string} input.message - Notification message text.
 * @param {string} input.notification_type - Type of notification (e.g.
"idle_prompt").
 * @returns {number} Exit code.
 */
function handleNotification(input) {
  const type = input.notification_type;
  if (type === 'permission_prompt') {
    // Claude is asking for permission to use a tool
    console.log(`[Claude Notification] Permission requested: ${input.message}
`);
    // (In a real scenario, you might integrate with a system notifier here)
  } else if (type === 'idle_prompt') {
    // Claude has been idle for a while
    console.log(`[Claude Notification] Idle: Waiting for user input...`);
  }
  // Other notification types could be handled or ignored
  return 0;
}

// Run the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handleNotification(input));
```

*Explanation:* This Notification hook checks the `notification_type`. If Claude triggers a permission dialog (`"permission_prompt"`), the hook logs a message (which could be picked up by a monitoring system or seen by the user in verbose output) indicating Claude needs approval. If Claude is idle (`"idle_prompt"`), it logs a different message. In practice, one might replace the `console.log` with an OS-level notification (for example, using a Node notifier library or writing to a GUI) so that the user immediately knows Claude is waiting. The hook doesn't attempt to alter Claude's behavior – it purely provides informative output. Since no critical error occurs, it exits with 0. (If this hook failed to run, an exit code 2 could be used to surface an error message to the user, but that's an edge case [26].)

## UserPromptSubmit Hook

**Purpose & Behavior:** The **UserPromptSubmit** hook triggers *when the user submits a prompt*, just before Claude receives it [27]. This is a powerful hook point because it allows preprocessing or filtering of user input. You can use it to validate the prompt (e.g. disallow certain content or ensure required info is present) or to enrich the prompt with additional context automatically (like injecting relevant code or data from your environment). Essentially, it's your chance to programmatically modify or augment what the user just asked, before Claude acts on it.

**Input:** The JSON for UserPromptSubmit includes:

- `prompt` (string) – The full text of the user's prompt that was just submitted [28]. This is the primary field of interest.
- *(Plus the standard fields like session_id, transcript_path, etc. There are no tool-specific fields since this event is about the user's message.)*

**Output & Effects:** The UserPromptSubmit hook can either inject additional context into the conversation or block the prompt from proceeding:

- **Add context to the prompt:** If the hook prints any **stdout** while exiting with code 0, that output will be **added to Claude's context** for this prompt. The simplest way is to print plain text which will appear as a system message that Claude sees. For more structured control, you can output JSON with `hookSpecificOutput.additionalContext` containing the text to inject [29]. Either method (plain text or JSON with `additionalContext`) will provide extra information for Claude before it generates a response [30]. This is useful for automatically supplying relevant code snippets, environment details, or clarifications. Multiple hooks can add context and it will all be included.
- **Block or alter the prompt:** If you decide the user's prompt should not be processed (e.g. it's against policy or simply a mistaken input), the hook can prevent it. Return a JSON with `"decision": "block"` and a `reason` explaining why [29], then exit 0. Claude will not see the prompt at all, and the `reason` will be shown to the user (but not to Claude) indicating the prompt was blocked. Additionally, exiting with code 2 will also block the prompt; in that case, the hook's stderr is shown to the user as the error [31].
- **Modify the prompt text:** There is no direct field to change the prompt in the JSON schema (other than adding context or blocking it). If needed, you could block the prompt and then possibly re-submit a modified version via a system message, but typically adding context achieves most customization.

**Example – Prompt Validation and Auto-Context:** The Node.js example below implements a UserPromptSubmit hook that does two things: it blocks overly short prompts (asking the user to be more specific), and it automatically adds context if the prompt references a file. Specifically, if the user's prompt contains a pattern like `"file: <path>"`, the hook will attempt to read that file from the project directory and include its content as additional context for Claude. This uses `prompt` from input and shows both blocking and context injection:

```
#!/usr/bin/env node
const fs = require('fs');
const path = require('path');

/**
 * Handle UserPromptSubmit hook: validate and enrich user prompts.
 * @param {Object} input - JSON payload for the UserPromptSubmit event.
 * @param {string} input.session_id - Session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Permission mode.
 * @param {string} input.hook_event_name - Event name ("UserPromptSubmit").
 * @param {string} input.prompt - The user's prompt text.
 * @returns {number} Exit code.
 */
```

```
function handleUserPromptSubmit(input) {
  const userPrompt = input.prompt || "";
  // 1. Simple validation: block empty or very short prompts
  if (userPrompt.trim().length < 5) {
    // Too little content in prompt - block it
    console.error('Prompt is too short. Please provide more details.');
    return 2;  // Exit code 2: block prompt, show stderr to user as error
  }
  // 2. Auto-context: if prompt references a file, inject that file's content
  const filePattern = /file:\\s*([\\w\\.\\/\\-]+)/i;
  const match = userPrompt.match(filePattern);
  if (match) {
    let filePath = match[1];
    // Resolve relative paths against Claude's current directory
    if (!path.isAbsolute(filePath)) {
      filePath = path.join(input.cwd, filePath);
    }
    if (fs.existsSync(filePath) && fs.statSync(filePath).isFile()) {
      const fileContent = fs.readFileSync(filePath, 'utf8');
      const contextText = `Content of ${filePath}:\n${fileContent}`;
      const output = {
        hookSpecificOutput: {
          hookEventName: 'UserPromptSubmit',
          additionalContext: contextText
        }
      };
      console.log(JSON.stringify(output));
      // Exit 0: Claude will receive the additional context along with the
original prompt
      return 0;
    }
  }
  // No modifications; allow the prompt as-is
  return 0;
}

// Execute the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handleUserPromptSubmit(input));
```

*Explanation:* This hook first checks the length of the user's prompt. If it's shorter than 5 characters (after trimming whitespace), it's likely too vague or possibly an accidental submit, so the hook blocks it. By calling `console.error` with a message and returning exit code 2, the CLI will erase the prompt and show the error to the user, prompting them to try again [31]. Next, the hook searches the prompt for a pattern like "file: filename". If such a reference is found and the file exists in the current project (`input.cwd` gives the base directory), the hook reads the file content and outputs it as `additionalContext` in JSON [29]. This text will be injected into Claude's context before Claude responds, effectively as if the user had provided that file content. The hook then exits 0, allowing the prompt to proceed (now with the extra context attached). If the prompt passes both checks (not too short, no file reference), the hook does nothing and exits 0, letting Claude handle the prompt normally.

This example illustrates both validation (to improve prompt quality) and automatic context provisioning, enhancing Claude's ability to answer the question.

## Stop Hook

**Purpose & Behavior:** The **Stop** hook runs when the *main Claude agent has finished its response* and is about to stop [32]. In a normal session, Claude stops when it believes it has completed the user's request or when you issue a `/stop`. The Stop hook gives you a last chance to decide if Claude should really stop or if it should continue working on the task. This is particularly useful in "agentic" workflows where you want Claude to autonomously continue until certain criteria are met (for example, all tests pass, or all subtasks are done). Essentially, the Stop hook can turn a one-shot interaction into an iterative loop by blocking the stop and feeding Claude further instructions.

**Input:** The Stop hook input JSON includes:

- *(Common fields like session_id, etc.)*
- `stop_hook_active` (boolean) – This field is `true` if this Stop event was itself triggered after a previous Stop hook already instructed Claude to continue [33]. It serves as a safeguard to prevent infinite loops: if your Stop hook keeps telling Claude to continue, this flag will be true on subsequent Stop triggers so you can decide to finally let it stop.

There are no tool-specific fields here, since this event pertains to the end of Claude's response.

**Output & Effects:** The Stop hook can control whether the session should conclude or not:

- **Continue the session (block the stop):** To make Claude continue instead of stopping, output a JSON with `"decision": "block"` and a `reason` [34]. The `reason` should instruct Claude on what to do next or why it shouldn't stop yet. When you block a stop, Claude treats it as if it got a new prompt – it will see the reason (but not the word "block") and will keep going. For example, you might supply a reason like "Tests are failing, continue fixing the code" or "There are remaining tasks, do not stop yet." **Important:** If you do this, be careful to eventually allow the session to stop – otherwise Claude could loop indefinitely. Use the `stop_hook_active` flag to detect if you already continued once and avoid repeating infinitely.
- **Allow the stop:** If you want the session to end normally, do not output any blocking decision. Simply exit with code 0 and no JSON (or provide any output that doesn't include a block decision). In practice, most Stop hooks will decide dynamically — e.g., "if condition X is not met, block, otherwise let stop."
- **No special effect on user interface:** Unlike tool hooks, a Stop hook's primary use is internal control. If you exit with code 2 (error), Claude's stop will be blocked as well, and the error text (stderr) will be shown to Claude as if it were a system message [35]. However, using the JSON `"decision": "block"` is the clearer way to extend the session.

**Example – Continuous Development Until Success:** In this Node.js Stop hook example, we simulate an environment where we only want Claude to stop when a test suite passes. The hook automatically runs the project's tests when Claude tries to stop. If the tests fail, it blocks the stop and instructs Claude to continue fixing issues; if they pass (or if we've already continued once to avoid infinite loops), it allows the session to end. This demonstrates using external commands (`execSync` to run tests) and the `stop_hook_active` flag:

```
#!/usr/bin/env node
const fs = require('fs');
const { execSync } = require('child_process');

/**
 * Handle Stop hook: decide whether Claude should stop or continue.
 * @param {Object} input - JSON payload for the Stop event.
 * @param {string} input.session_id - Session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Permission mode.
 * @param {string} input.hook_event_name - Event name ("Stop").
 * @param {boolean} input.stop_hook_active - True if this stop event is after
a previous continuation.
 * @returns {number} Exit code.
 */
function handleStop(input) {
  // If we've already continued once before (to avoid infinite loop), allow
stop
  if (input.stop_hook_active) {
    return 0;
  }
  try {
    // Attempt to run the test suite (synchronously)
    execSync('npm test', { stdio: 'ignore' });
    // If tests pass (exit code 0), do nothing and let Claude stop
    return 0;
  } catch (err) {
    // Tests failed (non-zero exit): instruct Claude to keep working
    const output = {
      decision: 'block',
      reason: 'Some tests are still failing. Continue debugging and fixing
the issues before stopping.'
    };
    console.log(JSON.stringify(output));
    return 0;
  }
}

// Run the handler with input from stdin
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handleStop(input));
```

*Explanation:* This Stop hook first checks `stop_hook_active`. If `true`, it means the hook already prevented one stop earlier (Claude continued at least once), so to avoid an endless loop, the hook simply returns 0, allowing the session to end. If `stop_hook_active` is false (first time stopping), the hook runs `npm test` in the project directory. If the tests throw an error (non-zero exit code), the catch block constructs a JSON output with `"decision": "block"` and a reason message [36]. This reason explicitly tells Claude that tests are failing and it should continue working. Claude will receive this and proceed to fix the problems instead of stopping. The hook then exits 0, as the JSON output is enough to influence Claude. If the tests pass, no JSON is output and the hook returns 0 immediately – meaning Claude is allowed to stop normally since the goal is achieved. This pattern ensures Claude keeps iterating on the task until success conditions are met (tests passing), at which point the Stop hook no longer blocks further.

## SubagentStop Hook

**Purpose & Behavior:** The **SubagentStop** hook triggers when a *Claude subagent (spawned via the* `Task` *tool) finishes its task* [37]. Claude Code can create subagents to work on subtasks in parallel; each subagent is essentially a Claude instance handling a "Task". The SubagentStop hook is analogous to the Stop hook but for these subagents. It allows you to determine if a subagent has truly completed its assignment or if it should be made to continue/refine its result.

**Input:** The input is similar to the Stop hook:

- `stop_hook_active` (boolean) – Indicates if this stop event is after a previous continuation for the same subagent (to avoid infinite loops, just like the main Stop hook).
- Additionally, you can identify which subagent/task this is by context (though the input doesn't explicitly name the subagent, the `tool_name` would be `"Task"` in the initiating PreToolUse, and the subagent's conversation can be found in the transcript if needed).

Common fields like session_id and cwd are present as usual.

**Output & Effects:** The SubagentStop hook uses the same mechanism as Stop:

- **Continue subagent (block stop):** Output JSON with `"decision": "block"` and a `reason` to have the subagent continue working instead of concluding [34]. The reason will be given to the subagent Claude instance as guidance on what to do next. This is used if the subagent's result is not satisfactory or incomplete.
- **Allow subagent to stop:** If no JSON decision is output, the subagent will stop normally and return its result to the main session.
- Like before, check `stop_hook_active` to avoid endless retries. If true, you likely should allow the subagent to finish to prevent it looping forever.
- In practice, you might use SubagentStop hooks to ensure a subtask meets certain criteria. For example, if a subagent was tasked with retrieving data and the data is not found, you could block and tell it to try a different approach.

**Example – Ensure Subtask Completion:** The Node.js example below imagines a scenario where each subagent is expected to produce a specific output file (say, as part of its task). The SubagentStop hook checks for that output file's existence. If the file is missing when the subagent tries to stop, the hook blocks the stop and asks the subagent to keep working. If the file exists or if we've already continued once, it lets the subagent finish:

```
#!/usr/bin/env node
const fs = require('fs');
const path = require('path');

/**
 * Handle SubagentStop hook: ensure subagent tasks are fully complete.
 * @param {Object} input - JSON payload for the SubagentStop event.
 * @param {string} input.session_id - Session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Permission mode.
 * @param {string} input.hook_event_name - Event name ("SubagentStop").
 * @param {boolean} input.stop_hook_active - True if this subagent already
continued once.
 * @returns {number} Exit code.
 */
function handleSubagentStop(input) {
  if (input.stop_hook_active) {
    // Already forced a continuation once; allow stop now to avoid infinite
loop
    return 0;
  }
  // Suppose each subagent should output a file named "subagent_output.txt"
in cwd
  const expectedFile = path.join(input.cwd, 'subagent_output.txt');
  if (!fs.existsSync(expectedFile)) {
    // The expected output is missing – tell subagent to keep working
    const output = {
      decision: 'block',
      reason: 'The subtask is not complete. Please continue until
"subagent_output.txt" is created.'
    };
    console.log(JSON.stringify(output));
    return 0;
  }
  // File exists, meaning subagent accomplished its goal – allow it to stop
  return 0;
}

// Execute the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handleSubagentStop(input));
```

*Explanation:* This SubagentStop hook anticipates that the subagent's task involves creating a file `subagent_output.txt`. When the subagent signals it's done, the hook checks for that file. If `subagent_output.txt` **does not exist**, clearly the subagent hasn't finished its required work – so the hook outputs a JSON with `decision: "block"` and a reason instructing the subagent to continue working until the file is produced. Claude (the subagent instance) will receive this and continue its task instead of terminating. If the file does exist, or if we already blocked once (`stop_hook_active` is true in a subsequent stop), the hook returns 0 with no output, allowing the subagent to conclude. This ensures the subagent completes its specific objective before stopping, similar to how the Stop hook ensures the main task is fully done [34]. Once the subagent stops (either normally or after being told to continue), its result is returned to the main Claude conversation.

## PreCompact Hook

**Purpose & Behavior:** The **PreCompact** hook runs *right before Claude performs a context compaction* [38]. Claude Code uses compaction (summarizing or truncating parts of the conversation) when the context window is full or when the user manually triggers a `/compact`. The PreCompact hook allows you to take action prior to this compaction. Typical uses include saving a copy of the conversation (so no information is lost), cleaning up or annotating content before compaction, or integrating version control (e.g. making a checkpoint commit before the context is condensed).

**Input:** The PreCompact input contains:

- `trigger` (string) – What caused compaction. It can be `"manual"` if the user invoked `/compact` intentionally, or `"auto"` if Claude is auto-compacting because the context is at capacity [39].
- `custom_instructions` (string) – If the compaction was manual and the user provided custom instructions for how to compact, those instructions are included here [40]. It will be an empty string for auto compactions.
- *(Plus standard fields like session_id, etc.)*

**Output & Effects:** The PreCompact hook does not have a specialized decision schema to control compaction itself – compaction will happen regardless (you generally cannot block it via a hook). The purpose is mainly for side effects or preparation:

- **Backup or logging:** You might use this hook to save the state before compaction. For example, you could archive the full transcript to a file or git commit, ensuring you have a record before Claude summarizes it away. This is a common pattern to “never lose work during long sessions.”
- **Custom actions based on trigger:** Because you know whether it’s an auto or manual compaction and any instructions, you can script different behavior. For instance, for manual compaction with special instructions, you might log those instructions or handle them differently.
- **No decision control:** There is no `decision` field for PreCompact. If you return exit code 2, it won’t cancel the compaction; it will just surface an error message to the user (non-blocking) [41]. The compaction will proceed after the hook runs. So generally you’ll use exit code 0 (possibly with output for the user’s reference or just silent).
- **Persisting state:** One interesting use could be to flush any in-memory state or to instruct Claude to remember something important by printing a system message (but since compaction usually means Claude will summarize anyway, that may not be reliable).

**Example – Backup Conversation Log:** The Node.js example below implements a PreCompact hook that creates a backup of the conversation transcript file just before compaction. It also logs a message if the compaction was manual and had custom instructions. This demonstrates using the `trigger` and `custom_instructions` fields and performing file operations with `transcript_path`:

```
#!/usr/bin/env node
const fs = require('fs');

/**
 * Handle PreCompact hook: backup context and log compaction triggers.
 * @param {Object} input - JSON payload for the PreCompact event.
 * @param {string} input.session_id - Session ID.
 * @param {string} input.transcript_path - Path to the conversation log.
 * @param {string} input.cwd - Current working directory.
 * @param {string} input.permission_mode - Permission mode.
 * @param {string} input.hook_event_name - Event name ("PreCompact").
 * @param {string} input.trigger - What triggered compaction ("manual" or
"auto").
 * @param {string} input.custom_instructions - User-provided compact
instructions (if any).
 * @returns {number} Exit code.
 */
function handlePreCompact(input) {
  if (input.trigger === 'manual' && input.custom_instructions) {
    console.log(`Compaction triggered manually with instructions: "$
{input.custom_instructions}"`);
  }
  // Backup the transcript file before compaction
  const backupPath = input.transcript_path + '.backup_before_compact';
  try {
    fs.copyFileSync(input.transcript_path, backupPath);
    console.log(`Transcript backed up to ${backupPath}`);
  } catch (err) {
    console.error(`Failed to backup transcript: ${err.message}`);
    // (Non-blocking: compaction will proceed even if backup fails)
  }
  return 0;
}

// Run the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handlePreCompact(input));
```

*Explanation:* This PreCompact hook first checks if `trigger` is `"manual"` and if any `custom_instructions` were provided by the user. If so, it logs those instructions to stdout (so the user can see in verbose mode what compaction instructions were used). Then it creates a backup of the entire transcript file by copying it to a new file with a `.backup_before_compact` extension. This ensures that even if Claude compacts and removes some context, the original conversation is preserved externally. The hook catches any file system errors during backup and logs them to stderr. Regardless of success or failure, it exits 0 – compaction continues either way. The messages printed are informational (since exit 0, they won't interrupt Claude; the backup success message would be visible in verbose output). This hook illustrates using PreCompact to add safety and logging around the compaction process without altering Claude's behavior.

## SessionStart Hook

**Purpose & Behavior:** The **SessionStart** hook runs whenever a new Claude Code session begins (or an existing session is resumed, which under the hood starts a new session) [42]. This is triggered on initial startup, on explicit resume (`--resume` or `/resume` commands), and even on a "clear" (which ends the old session and starts fresh), as well as after an automatic compaction that essentially restarts the session with a summary [43]. The SessionStart hook is ideal for doing any setup required at the beginning of a session. Common uses include loading project-specific context (like summarizing recent code changes, loading issue/task lists), initializing environment variables or dependencies, and configuring the development environment for Claude.

**Input:** The SessionStart input JSON provides:

- `source` (string) – Indicates what caused the session start. It can be `"startup"` (a brand new session), `"resume"` (continuation of a previous session or state), `"clear"` (user cleared the session), or `"compact"` (session restarted due to compaction) [44] [45]. Knowing the source can inform what context to load.
- *(Standard fields like session_id, transcript_path, etc., are included as well. Note: SessionStart hooks uniquely have access to a special environment variable, not in JSON, called* `CLAUDE_ENV_FILE` *which is explained below.)*

**Output & Effects:** The SessionStart hook cannot block the session from starting (there's no concept of denying a session start), but it can augment the starting state:

- **Inject initial context:** The hook can output text that will be inserted into the conversation as system or developer-provided context that Claude sees from the get-go. To do this, output JSON with `hookSpecificOutput.additionalContext` containing your context string [46]. Claude will take this into account as if it was part of the system setup. Multiple SessionStart hooks can add context, which will be concatenated. For example, you might add a summary of the project, a list of current tickets, or instructions like "This project uses framework X, keep responses in that style."
- **Set environment variables:** When Claude Code launches the SessionStart hook, it provides an environment variable `CLAUDE_ENV_FILE` pointing to a file. The hook can write `export KEY=VALUE` lines to this file, and those env vars will be loaded into Claude's environment for all subsequent tool executions in this session [47] [48]. This is very useful for setting up things like PATH adjustments, API keys, or other environment config that your tools (e.g. compilers, linters, etc.) might need. It persists only for the session's lifetime.
- **Install or initialize tools:** You could run commands (via the hook script itself or by launching subprocesses) to, say, install dependencies or start services. (Be mindful of timeouts – hooks should finish reasonably fast, or you might want to spawn background processes.)
- **No blocking:** As mentioned, there's no mechanism to stop a session from starting via the hook. If the hook exits with code 2, it will just show an error message to the user (in verbose mode) but the session will still start [49]. So usually you aim for exit 0.

**Example – Load Project Context and Set Environment:** The Node.js SessionStart hook example below does two things when a session starts: it reads a `CONTEXT.md` file from the project and injects its content as additional context for Claude, and it sets a couple of environment variables (like `NODE_ENV` and an API token) by appending to `CLAUDE_ENV_FILE`. This demonstrates using `source` to decide what to do, reading files for context, and writing to the env file:

```
#!/usr/bin/env node
const fs = require('fs');
const path = require('path');

/**
 * Handle SessionStart hook: load initial context and configure environment.
 * @param {Object} input - JSON payload for the SessionStart event.
 * @param {string} input.session_id - New session ID.
 * @param {string} input.transcript_path - Path to conversation log.
 * @param {string} input.cwd - Project working directory (root of Claude
session).
 * @param {string} input.permission_mode - Permission mode.
 * @param {string} input.hook_event_name - Event name ("SessionStart").
 * @param {string} input.source - How the session was started ("startup",
"resume", "clear", "compact").
 * @returns {number} Exit code.
 */
function handleSessionStart(input) {
  // 1. Inject project context at startup/resume
  let contextToAdd = "";
  const contextFile = path.join(input.cwd, 'CONTEXT.md');
  if (fs.existsSync(contextFile)) {
    contextToAdd = fs.readFileSync(contextFile, 'utf8');
  }
  if (contextToAdd) {
    const output = {
      hookSpecificOutput: {
        hookEventName: 'SessionStart',
        additionalContext: contextToAdd
      }
    };
    console.log(JSON.stringify(output));
  }

  // 2. Set up environment variables for the session
  const envFilePath = process.env.CLAUDE_ENV_FILE;
  if (envFilePath) {
    // Persist some environment variables for tools
    fs.appendFileSync(envFilePath, 'export NODE_ENV=development\n');
    fs.appendFileSync(envFilePath, 'export API_TOKEN=abcdef12345\n');
  }

  return 0;
}

// Execute the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handleSessionStart(input));
```

*Explanation:* When Claude Code starts a session, this hook runs. First, it looks for a file `CONTEXT.md` in the project directory (given by `input.cwd`). If found, it reads the file content (which might contain an overview of the project or recent notes) and prepares a JSON output with that content as `additionalContext` [50]. By printing this JSON and exiting 0, the hook ensures Claude's conversation context begins with that content already included. This can give Claude important knowledge before the user even asks anything. Next, the hook checks for the special environment file path in `process.env.CLAUDE_ENV_FILE` [47]. If present, it appends a couple of environment variable definitions to that file. Here we set `NODE_ENV=development` and an example `API_TOKEN`. All subsequent bash or tool commands that Claude runs will have these variables available in their environment (because Claude Code will source that file for each tool execution) [48]. The hook then exits successfully. This way, at session start, the AI is primed with project context and the environment is configured (e.g., the `API_TOKEN` could be used by any script or tool without exposing it directly in prompts). By checking `input.source`, you could tailor behavior (for example, only load `CONTEXT.md` on a fresh `startup` vs. a resume), but in this simple example we treat them the same.

## SessionEnd Hook

**Purpose & Behavior:** The **SessionEnd** hook runs when a Claude Code session ends [51]. This could be triggered by the user exiting the CLI, logging out, clearing the session, or other termination events. The SessionEnd hook is your opportunity to perform any cleanup or final logging once the session is over. For instance, you might save a summary of the session, upload artifacts, stop background services started during SessionStart, or simply record how the session ended.

**Input:** The input JSON for SessionEnd includes:

- `reason` (string) – The reason the session ended [52]. Possible values include `"clear"` (user cleared the session), `"logout"` (user logged out), `"prompt_input_exit"` (the user exited while the prompt input was open), or `"other"` for miscellaneous reasons. This helps distinguish why the session ended.
- *(Standard fields like session_id, cwd, etc., are present as well. Notably,* `cwd` *might still be set to the project directory, which can be useful for cleanup tasks like removing temp files.)*

**Output & Effects:** The SessionEnd hook is mainly for side effects. There is no way to prevent a session from ending at this point (the session is ending regardless):

- **Cleanup resources:** If your SessionStart hook launched background processes (like a dev server) or allocated resources, SessionEnd is where you'd stop those processes or free resources. You might kill any remaining child processes, or delete temporary files, etc.
- **Logging and analytics:** This is a good place to log the session transcript or any stats (duration, number of prompts, tools used, etc.) to an external file or system for later review. For example, you could append to a log file information like when the session ended and why.
- **No blocking:** You cannot block the session from ending. Any exit code from the hook (even 2) will not stop termination; at most, an error message could be shown to the user in verbose mode if something failed. The hook runs and then the session is closed – it's a fire-and-forget scenario [53]. So ensure your hook executes quickly or asynchronously handle longer tasks (since the user might already be gone).

**Example – Session Logging and Cleanup:** The Node.js example below implements a SessionEnd hook that appends an entry to a log file noting that the session ended and why, and also cleans up a temporary directory that might have been used during the session. This illustrates using the `reason` field and performing file operations at session teardown:

```javascript
#!/usr/bin/env node
const fs = require('fs');
const path = require('path');

/**
 * Handle SessionEnd hook: log session end and perform cleanup.
 * @param {Object} input - JSON payload for the SessionEnd event.
 * @param {string} input.session_id - The ID of the session that is ending.
 * @param {string} input.transcript_path - Path to the conversation log file.
 * @param {string} input.cwd - Working directory of the session.
 * @param {string} input.permission_mode - Permission mode.
 * @param {string} input.hook_event_name - Event name ("SessionEnd").
 * @param {string} input.reason - Reason for session termination (e.g.
"logout", "clear").
 * @returns {number} Exit code.
 */
function handleSessionEnd(input) {
  // Append session end information to a log file
  const logMessage = `[${new Date().toISOString()}] Session $
{input.session_id} ended (reason: ${input.reason}).\n`;
  fs.appendFileSync(path.join(input.cwd, 'session_log.txt'), logMessage);

  // Cleanup: remove a temporary directory if it exists
  const tempDir = path.join(input.cwd, '.tmp_claude');
  if (fs.existsSync(tempDir)) {
    fs.rmSync(tempDir, { recursive: true, force: true });
  }

  return 0;
}

// Execute the handler
const input = JSON.parse(fs.readFileSync(0, 'utf8'));
process.exit(handleSessionEnd(input));
```

*Explanation:* When the session is ending, this hook writes a line to `session_log.txt` in the project directory, noting the timestamp, session ID, and the reason for termination. For example, it might log something like: `[2025-11-20T01:30:00.123Z] Session abc123 ended (reason: logout).` This creates a simple audit trail of sessions. Then it checks if a directory named `.tmp_claude` exists (perhaps created during the session for temporary files) and removes it, cleaning up any leftover data. Finally, it exits with code 0. The session will terminate regardless, but this ensures that we leave no clutter and have a record of what happened.

---

Each of the above examples reflects real-world use cases for Claude Code CLI hooks, leveraging all available hook inputs to make decisions. By combining these hooks, you can greatly customize Claude Code's behavior – for instance, automatically formatting code before edits, committing changes after file writes, running tests and only stopping when they pass, and preserving context across sessions [5]. The hook system, as documented by Anthropic [4], provides deterministic control points in the otherwise free-form AI coding process, enabling integration with your development workflow and ensuring important actions happen every time.

---

[1] [3] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] Hooks reference - Claude Code Docs
https://code.claude.com/docs/en/hooks

2 How I use Claude Code (+ my best tips)
https://www.builder.io/blog/claude-code

4 Claude Code Hooks: What is and How to Use It - CometAPI
https://www.cometapi.com/claude-code-hooks-what-is-and-how-to-use-it/

5 Enabling Claude Code to work more autonomously \ Anthropic
https://www.anthropic.com/news/enabling-claude-code-to-work-more-autonomously