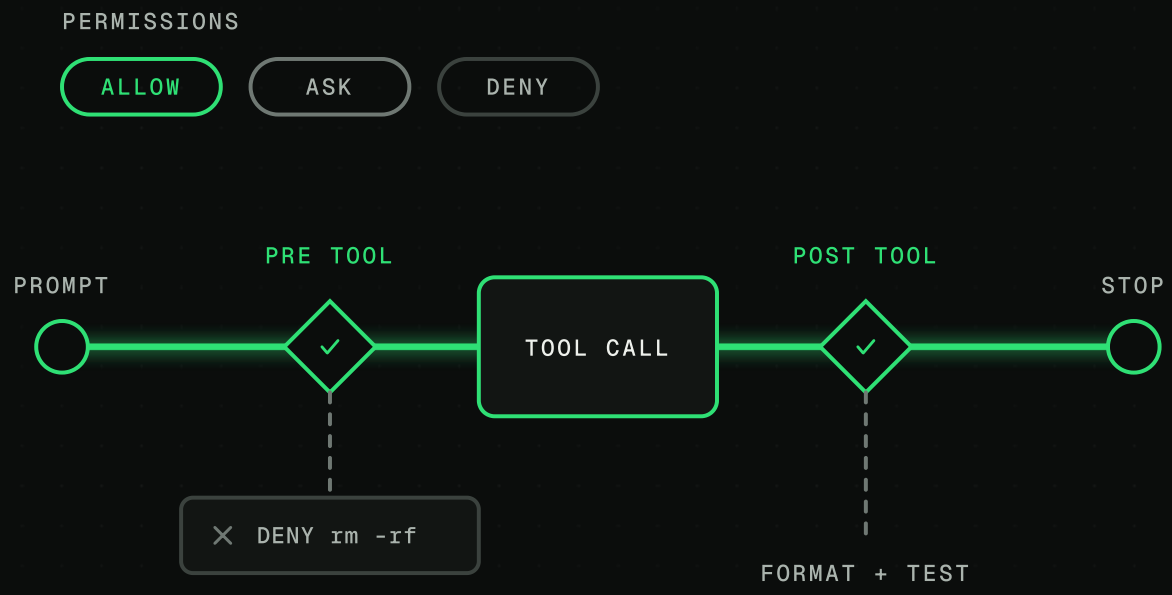


TEMPLATE · FREE

Claude Code Hooks and Permissions Starter

Starter permission rules and four hook templates that make Claude Code enforce a repository's hard rules.

HOW TO USE THIS TEMPLATE

Move hard rules out of CLAUDE.md prose and into permission rules and hooks that the harness enforces.

A copy-ready .claude/settings.json, four hook templates with their scripts, two reference tables and a rollout checklist, so the harness enforces your hard rules instead of CLAUDE.md requesting them.

WORKING METHOD

01 Copy the starter to .claude/settings.json at the repository root, replace every bracketed value, and delete rules that do not apply.

02 Add one hook at a time: merge its snippet into the single hooks object, save the script under .claude/hooks/, and make it executable.

03 Test each script by piping sample JSON into it, then confirm it is registered by running /hooks inside Claude Code.

04 Try variations in .claude/settings.local.json before committing them.

05 Finish with the rollout checklist and commit settings and hooks in one reviewed pull request.

INSIDE

- ✓ Starter settings.json with permission rules
- ✓ 4 hook templates
- ✓ Hook event reference table
- ✓ Rollout checklist

BEFORE YOU START

Command hooks run with your full user permissions. Review hook scripts and changes to .claude/ like production code.

Bash rules match the command text Claude writes, and the documentation says they are not a security boundary: a deny for curl does not stop /usr/bin/curl or sh -c 'curl …'. Use sandboxing where that matters.

Read and Edit deny rules cover the built-in file tools, not a script that opens files itself.

This area changes between releases. Every construct here comes from the official documentation on the date shown under Sources; confirm it against your installed version before rollout.

CONTENTS

01	Starter settings.json	TEMPLATE
02	Hook template 1: block edits to protected paths	TEMPLATE
03	Hook template 2: block dangerous shell commands	TEMPLATE
04	Hook template 3: format and check after edits	TEMPLATE
05	Hook template 4: completion check before Claude stops	TEMPLATE
06	Hook event reference	TABLE
07	Settings files and precedence	TABLE
08	Rollout checklist	22 CHECKS

Starter settings.json

Permission rules for a shared project file: what runs without asking, what always prompts, and what is blocked.

.CLAUDE/SETTINGS.JSON · COPY AND ADAPT

```
{
  "$schema": "https://json.schemastore.org/claude-code-settings.json",
  "permissions": {
    "allow": [
      "Bash([LINT COMMAND])",
      "Bash([TEST COMMAND] *)",
      "Bash([BUILD COMMAND])",
      "Bash(git commit *)"
    ],
    "ask": [
      "Bash(git push *)",
      "Bash([DEPLOY COMMAND] *)",
      "Bash([MIGRATION COMMAND] *)",
      "Edit(/[CI CONFIG DIRECTORY]/**)"
    ],
    "deny": [
      "Read(./.env)",
      "Read(./.env.*)",
      "Read(./[SECRETS DIRECTORY]/**)",
      "Edit(/[GENERATED OR VENDORED DIRECTORY]/**)",
      "Bash(curl *)",
      "Bash(git push --force *)",
      "Bash(git reset --hard *)",
      "Bash([DESTRUCTIVE COMMAND] *)"
    ],
    "disableBypassPermissionsMode": "disable"
  }
}
```

01 Rule format is Tool or Tool(specifier). Rules are evaluated deny first, then ask, then allow; the first match decides and specificity does not change the order, so an allow rule cannot carve an exception out of a deny rule.

02 In a Bash rule, * matches any text including spaces. Put it after the subcommand: Bash(git commit *) allows only commits, while Bash(git *) allows every git command. A rule with no * matches one exact command.

03 Read and Edit paths use gitignore syntax. /path anchors at the primary working directory when the rule is in project settings, ./path or path is relative to the current directory, ~/path is the home directory, and //path is an absolute filesystem path.

04 Use Edit(…) for anything that writes and Read(…) for anything that reads or searches. The documentation says Write(path), Glob(path) and Grep(path) rules with a path are accepted but not matched.

05 A bare filename matches at any depth, so in a monorepo Read(.env) or Read(**/.env) blocks every nested .env under the current directory. Read(./.env.*) also blocks .env.example; rename the example file or drop that rule if Claude needs to read it.

06 The allow list in a committed project file applies only after each person accepts the workspace trust dialog for the folder. Deny and ask rules apply immediately.

07 disableBypassPermissionsMode set to "disable" makes Claude Code reject --dangerously-skip-permissions. Remove the line if your team runs bypass mode on purpose inside isolated containers.

08 Settings files are strict JSON. A // comment or a trailing comma is a syntax error, and the $schema line is optional editor validation that can lag behind new releases.

02 / 08 TEMPLATE

Hook template 1: block edits to protected paths

A PreToolUse hook on Edit and Write that exits with code 2 when the target file matches a protected path, so the edit never runs and Claude is told why.

.CLAUDE/SETTINGS.JSON + .CLAUDE/HOOKS/PROTECT-PATHS.SH · COPY AND ADAPT

```
# 1. Merge into .claude/settings.json
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/protect-paths.sh"
          }
        ]
      }
    ]
  }
}

# 2. Save as .claude/hooks/protect-paths.sh, then: chmod +x .claude/hooks/protect-paths.sh
#!/bin/sh
# PreToolUse hook for Edit|Write. Exit 2 blocks the edit; stderr goes to Claude.
command -v jq >/dev/null 2>&1 || { echo "protect-paths.sh needs jq" >&2; exit 2; }

INPUT=$(cat)
FILE_PATH=$(printf '%s' "$INPUT" | jq -r '.tool_input.file_path // empty')
# file_path arrives absolute. On Windows it uses backslashes, so normalize.
FILE_PATH=$(printf '%s' "$FILE_PATH" | tr '\\' '/')

# One path fragment per line. Keep the slashes: "/.git/" must not match ".github".
while IFS= read -r pattern; do
  case "$FILE_PATH" in
    *"$pattern"*)
```

```
      echo "Blocked: $FILE_PATH matches protected pattern '$pattern'. [WHAT TO DO INSTEAD]" >&2
      exit 2
      ;;
  esac
done <<'EOF'
/.env
/.git/
/[LOCKFILE NAME]
/[GENERATED OR VENDORED DIRECTORY]/
/[APPLIED MIGRATIONS DIRECTORY]/
EOF

exit 0
```

01 Exit code 2 is the blocking code. Exit code 1 without valid JSON output is a non-blocking error and the edit goes ahead, even though 1 is the usual Unix failure code.

02 The matcher Edit|Write is an exact list of tool names and is case-sensitive. A hook only sees calls to the tools it matches, so Claude can still change a protected file through a shell command.

03 Pair this hook with an Edit deny rule for the same paths in the starter settings. The documentation says Edit rules apply to all built-in tools that edit files and to file commands Claude Code recognizes in Bash.

04 Match a path fragment such as /migrations/ instead of anchoring at the start, because tool_input.file_path is always absolute by the time the hook runs.

05 By default a hook that cannot start (wrong path, missing execute bit) is a non-blocking error. In Claude Code v2.1.295 or later you can add "onFailure": "block" to the handler so a broken policy hook blocks instead of letting everything through.

---

03 / 08 TEMPLATE

Hook template 2: block dangerous shell commands

A PreToolUse hook on Bash that inspects the full command text and returns a JSON decision: deny for commands that must never run, ask for commands a person must confirm.

.CLAUDE/SETTINGS.JSON + .CLAUDE/HOOKS/GUARD-BASH.SH · COPY AND ADAPT

```
# 1. Merge into .claude/settings.json
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/guard-bash.sh"
          }
```

```
        ]
      }
    ]
  }
}

# 2. Save as .claude/hooks/guard-bash.sh, then: chmod +x .claude/hooks/guard-bash.sh
#!/bin/sh
# PreToolUse hook for Bash. Prints one JSON decision on stdout and exits 0.
command -v jq >/dev/null 2>&1 || { echo "guard-bash.sh needs jq" >&2; exit 2; }

INPUT=$(cat)
COMMAND=$(printf '%s' "$INPUT" | jq -r '.tool_input.command // empty')

decide() {
  jq -n --arg decision "$1" --arg reason "$2" '{
    hookSpecificOutput: {
      hookEventName: "PreToolUse",
      permissionDecision: $decision,
      permissionDecisionReason: $reason
    }
  }'
  exit 0
}

case "$COMMAND" in
  *"rm -rf /"*|*"rm -rf ~"*|*'rm -rf $HOME'*)
    decide deny "Recursive delete of an absolute or home path. Use a relative path inside the
project." ;;
  *"git push"*"--force"*|*"git push"*" -f"*)
    decide deny "Force push is blocked. Push a new commit or ask [BRANCH OWNER]." ;;
  *"git reset --hard"*|*"git clean -f"*)
    decide ask "This discards uncommitted work." ;;
  *"[PRODUCTION DEPLOY COMMAND]"*)
    decide ask "Production deploy. Confirm [APPROVAL REQUIREMENT] first." ;;
  *"[DESTRUCTIVE DATABASE COMMAND]"*)
    decide deny "[WHY THIS IS BLOCKED AND THE SAFE ALTERNATIVE]" ;;
esac

exit 0
```

01 permissionDecision accepts allow, deny, ask and defer. This template uses only deny and ask: deny cancels the call and shows the reason to Claude, ask shows the normal permission prompt with the reason.

02 The decision must sit inside hookSpecificOutput with hookEventName set to PreToolUse. A permissionDecision at the top level of the JSON is ignored without an error.

03 Print nothing but the JSON object on stdout. Output that does not start with { is read as plain text, which is why the script builds the object with jq instead of string concatenation.

04 Exiting 0 with no output is not an approval. The call continues through the normal permission flow, so your allow, ask and deny rules still decide it.

05 A hook deny holds in every permission mode, including bypassPermissions. A hook allow never overrides a deny or ask rule from settings.

06 On Windows, Claude can route shell commands through the PowerShell tool, and a hook that matches only Bash never fires there. Use the matcher Bash|PowerShell and add PowerShell command patterns if your team works on Windows.

07 The force-push pattern also matches --force-with-lease. Narrow the pattern if your workflow allows that flag.

Hook template 3: format and check after edits

A PostToolUse hook on Edit and Write that formats the edited file, runs a fast check on it, and exits 2 with the failure output so Claude sees the problem right after the edit.

.CLAUDE/SETTINGS.JSON + .CLAUDE/HOOKS/FORMAT-AND-CHECK.SH · COPY AND ADAPT

```
# 1. Merge into .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/format-and-check.sh",
            "timeout": 120
          }
        ]
      }
    ]
  }
}

# 2. Save as .claude/hooks/format-and-check.sh, then: chmod +x .claude/hooks/format-and-check.sh
#!/bin/sh
# PostToolUse hook for Edit|Write. The edit already happened; exit 2 only reports.
command -v jq >/dev/null 2>&1 || exit 0

INPUT=$(cat)
FILE_PATH=$(printf '%s' "$INPUT" | jq -r '.tool_input.file_path // empty')
[ -n "$FILE_PATH" ] && [ -f "$FILE_PATH" ] || exit 0

# Only handle source files. Replace the extensions.
case "$FILE_PATH" in
  *".[EXTENSION 1]"|*".[EXTENSION 2]") ;;
  *) exit 0 ;;
esac

cd "$CLAUDE_PROJECT_DIR" || exit 0

[FORMAT COMMAND] "$FILE_PATH" >/dev/null 2>&1

if ! OUTPUT=$([LINT OR TYPECHECK COMMAND] "$FILE_PATH" 2>&1); then
```

```
  echo "Check failed after editing $FILE_PATH. Fix these before continuing:" >&2
  printf '%s\n' "$OUTPUT" | head -n 40 >&2
  exit 2
fi

exit 0
```

01 PostToolUse cannot block or undo: the tool has already run. Exit code 2 on this event shows your stderr to Claude, which is how the check result reaches it.

02 Claude never sees stderr from a hook that exits 0, so a warning printed before a successful exit is invisible to it. Exit 2 when you want Claude to act on the output.

03 timeout is in seconds. The documented default for command hooks is 600; a short value keeps a hung formatter from stalling the session.

04 Keep this hook fast and scoped to the edited file. Put the full test suite in the completion check, not here, because this runs after every matching edit.

05 This hook does not run when a Bash command or a process outside Claude Code rewrites a file. The documentation points to the FileChanged event for reacting to a specific file changing on disk, whatever wrote it.

06 CLAUDE_PROJECT_DIR is exported to the hook process and stays at the project root where the session started, even if Claude enters a worktree. Read cwd from the input JSON if the hook must know where Claude is working.

---

05 / 08 TEMPLATE

Hook template 4: completion check before Claude stops

A Stop hook that runs the test command when the working tree has changes and exits 2 if it fails, which sends Claude back to keep working with the failure output.

.CLAUDE/SETTINGS.JSON + .CLAUDE/HOOKS/COMPLETION-CHECK.SH · COPY AND ADAPT

```
# 1. Merge into .claude/settings.json
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/completion-check.sh",
            "timeout": 300
          }
        ]
      }
    ]
  }
```

```
}

# 2. Save as .claude/hooks/completion-check.sh, then: chmod +x .claude/hooks/completion-check.sh
#!/bin/sh
# Stop hook. Exit 2 prevents Claude from stopping; stderr tells it why.
command -v jq >/dev/null 2>&1 || exit 0

INPUT=$(cat)
# Claude is already continuing because of a stop hook: let it stop this time.
if [ "$(printf '%s' "$INPUT" | jq -r '.stop_hook_active')" = "true" ]; then
  exit 0
fi

cd "$CLAUDE_PROJECT_DIR" || exit 0

# Nothing modified or untracked: nothing to verify.
if [ -z "$(git status --porcelain 2>/dev/null)" ]; then
  exit 0
fi

if ! OUTPUT=$([TEST COMMAND] 2>&1); then
  echo "The check [TEST COMMAND] is failing. Fix it before finishing, or report the failure and why it
cannot be fixed." >&2
  printf '%s\n' "$OUTPUT" | tail -n 40 >&2
  exit 2
fi

exit 0
```

01 Stop fires whenever Claude finishes responding, not only at the end of a task, and it does not fire on a user interrupt. The git status guard keeps the check from running after answers that changed nothing.

02 Stop has no matcher support, so the matcher group holds only the hooks array. A matcher added to an event without matcher support is silently ignored.

03 stop_hook_active is true when Claude is already continuing because of a stop hook. With the early exit, this hook pushes Claude back once per stop and then lets it finish, so a check that cannot pass does not loop.

04 Claude Code also overrides a Stop hook after eight consecutive blocks with no tool call in between. Do not design a check that depends on blocking more often than that.

05 On Stop, an exit 2 with nothing on stdout whose stderr says a file is missing, such as No such file or directory, is treated as a non-blocking error. Make sure the test command itself exists before trusting this gate.

06 The onFailure field has no effect on Stop hooks. If the script cannot run, Claude stops normally, so keep CI as the final gate for a passing suite.

Hook event reference

The events most useful for enforcement and automation, what each can and cannot do, and where to use it. Event names are case-sensitive.

Tool events (PreToolUse, PostToolUse, PostToolUseFailure, PermissionRequest, PermissionDenied) match on the tool name: a matcher of only letters, digits, underscores, hyphens, spaces, commas and pipes is an exact name or list such as Edit|Write, and anything else is an unanchored JavaScript regular expression. An empty, omitted or "*" matcher matches everything. The documentation lists more events than this table, including Setup, UserPromptExpansion, PostToolBatch, PermissionDenied, StopFailure, TaskCreated, TaskCompleted, TeammateIdle, InstructionsLoaded, CwdChanged, DirectoryAdded, WorktreeCreate, WorktreeRemove, PostCompact, PreModelSwitch, PostModelSwitch, MessageDisplay, SubagentStart, Elicitation and ElicitationResult. Check the hooks reference for the events your installed version supports.

EVENT	WHEN IT FIRES	WHAT IT CAN DO	TYPICAL USE
PreToolUse	After Claude creates tool parameters, before the tool call runs	Allow, deny, ask or defer the call; rewrite its input; exit 2 blocks it	Protect paths, block dangerous commands
PermissionRequest	When a tool call needs a permission decision	Allow or deny through hookSpecificOutput.decision.behavior; exit 2 is not honored	Auto-approve one narrow, known-safe prompt
PostToolUse	After a tool call succeeds	Cannot undo the call; exit 2 or decision block sends feedback to Claude	Format, lint, fast checks on the edited file
PostToolUseFailure	After a tool call fails	Exit 2 shows stderr to Claude; the tool already failed	Log failures, add a hint about the cause
UserPromptSubmit	When a prompt is submitted, before Claude processes it	Exit 2 blocks the prompt; can add context for Claude	Inject branch or freeze status with each prompt
Stop	When Claude finishes responding; not on a user interrupt	Exit 2 or decision block keeps Claude working	Completion check: tests, lint, uncommitted work
SubagentStop	When a subagent finishes	Exit 2 prevents the subagent from stopping	Apply the completion check to delegated work
SessionStart	When a session begins or resumes	Cannot block; can add context for Claude	Load environment notes, re-inject context after compaction
ConfigChange	When a configuration file changes during a session	Exit 2 blocks the change, except for policy settings	Audit or block settings edits made mid-session
PreCompact	Before context compaction	Exit 2 blocks compaction; stdout becomes instructions for the summary	Tell the summary what it must keep

EVENT	WHEN IT FIRES	WHAT IT CAN DO	TYPICAL USE
FileChanged	When a watched file changes on disk; the matcher lists filenames	No decision control; runs after the change	Reload environment when .envrc or .env changes
Notification	When Claude Code sends a notification	No decision control; exit code and stderr are ignored	Desktop alert when Claude needs input
SessionEnd	When a session terminates	Cannot block; hooks share a 1.5-second default budget	Cleanup and logging

07 / 08 REFERENCE

Settings files and precedence

Where a rule or hook lives decides who gets it and what can override it. Rows are ordered from highest precedence to lowest.

Precedence applies per key: a key at a higher level overrides the same key lower down. List keys such as permissions.allow and permissions.deny are merged across files instead of replaced. Permission rules add one stricter rule on top: if a tool use is denied at any level, no other level can allow it, so a user-level deny blocks a project-level allow and the reverse. The same hook handler defined in more than one settings file runs once. Run /status to see which settings files loaded, /permissions to see every rule with the file it comes from, and /hooks to see every hook with its source. If Claude Code creates .claude/settings.local.json it keeps the file out of git; if you create it by hand, add it to .gitignore yourself.

LEVEL	LOCATION	WHO IT AFFECTS	PUT HERE
1. Managed	managed-settings.json, an MDM policy, or server-managed settings	Everyone the organization deploys it to	Rules and hooks nobody may override
2. Command line	JSON or a file passed with --settings at launch	That one session only	Trial values; disableAllHooks for a single run
3. Project local	.claude/settings.local.json	You, in this one project	Personal overrides; testing a rule or hook before sharing it
4. Shared project	.claude/settings.json, committed to the repository	Everyone working in the folder that contains it	Team permission rules and hooks: this starter
5. User	~/.claude/settings.json	You, in every project on this machine	Personal rules and notification hooks

Rollout checklist

Verify the rules and hooks on one machine, then ship them to the team as reviewed code.

Before enabling

4 CHECKS

- [ ] 01 Check the installed Claude Code version against the version notes in this document; expect v2.1.295 or later before you add the onFailure field to any hook.
- [ ] 02 Run jq --version in the shell your hooks will use; expect a version string, because every script in this starter parses its input with jq.
- [ ] 03 Validate the settings file by running jq . .claude/settings.json; expect the formatted file back, since comments and trailing commas are syntax errors.
- [ ] 04 Search for leftover placeholders with grep -rn '\[[A-Z]' .claude; expect no matches in the settings file or in any hook script.

Permission rules

5 CHECKS

- [ ] 05 Open /permissions in a session started at the repository root; expect every starter rule listed and attributed to the project settings file.
- [ ] 06 Ask Claude to read the .env file through its file tools; expect the read to be refused without a permission prompt appearing.
- [ ] 07 Ask Claude to run the lint command after accepting the workspace trust dialog; expect it to run with no permission prompt.
- [ ] 08 On a scratch branch, ask Claude to run git push; expect a permission prompt from the ask rule, then answer No to cancel it.
- [ ] 09 Start a new session and read the startup output; expect no warning about a deny or ask rule that matches no known tool.

Hooks

7 CHECKS

- [ ] 10 Run /hooks inside Claude Code; expect the two PreToolUse hooks, the PostToolUse hook and the Stop hook listed with the project settings file as their source.
- [ ] 11 Run ls -l .claude/hooks; expect the execute bit on all four scripts, because a script that cannot start is a non-blocking error.
- [ ] 12 Pipe a harmless Bash input into guard-bash.sh from a terminal and run echo $?; expect exit code 0 and no output on stdout.

- [ ] 13 Pipe an input whose tool_input.file_path ends in /.env into protect-paths.sh; expect exit code 2 and a Blocked message on stderr.
- [ ] 14 Ask Claude to add a comment to the .env file; expect the edit to be blocked before it runs and the Blocked message returned to Claude.
- [ ] 15 Watch the transcript on each hook's first real run; expect no hook error notice containing No such file or directory, which means the path is wrong.
- [ ] 16 Break one test on purpose and let Claude finish a change; expect the Stop hook to send Claude back once with test output, then let it stop.

Team rollout

6 CHECKS

- [ ] 17 Commit .claude/settings.json and .claude/hooks/ in one pull request; expect review from whoever owns deployment and security before it merges.
- [ ] 18 Run git ls-files -s .claude/hooks after staging; expect mode 100755 on every script so the execute bit reaches teammates.
- [ ] 19 Run git check-ignore .claude/settings.local.json; expect the path printed, confirming personal overrides stay out of commits.
- [ ] 20 Have one teammate pull, start a session, accept the trust dialog and run /status; expect the project settings file in the Setting sources line.
- [ ] 21 Open CLAUDE.md and find each hard rule now enforced here; expect each to name its hook or deny rule, with duplicated prose removed.
- [ ] 22 List every place the repository runs claude -p unattended; expect a named reviewer for .claude/ changes, because those runs execute repository hooks without a trust dialog.

FIELD NOTES

Common mistakes, useful habits, hard limits.

01 **Exiting 1 to block**

A hook that exits with code 1 and prints no valid JSON is a non-blocking error: Claude Code shows a notice and the action goes ahead. Policy hooks must exit 2, or exit 0 and print a JSON decision. The same applies to a hook whose script path is wrong, which fails to start and blocks nothing unless onFailure is set to block.

02 **Using a hook allow to override a rule**

A PreToolUse hook that returns allow only skips the interactive prompt. Deny and ask rules from settings are still evaluated, so the call is still refused or still prompts. Fix the rule itself, and keep hooks for tightening behavior or for decisions that need to inspect the full input.

03 **Treating a Bash deny rule as containment**

Bash rules match the command as Claude writes it. A deny for rm does not stop /bin/rm or bash -c with the same command inside, and a deny for git push does not stop git -C . push. Use the hook to inspect the full command text, and use sandboxing or restricted credentials when the limit must hold regardless of how a command is spelled.

PRACTICAL TIPS

Use the if field on a hook handler to run it only for some calls, for example "if": "Bash(git *)". It takes one permission rule, works only on tool events, and is documented as best-effort, so it is a filter for cost and noise, not the enforcement itself.

When several PreToolUse hooks answer the same call, the most restrictive decision wins in the order deny, defer, ask, allow. Every matching hook still runs to completion, so do not rely on one hook's deny to stop another hook's side effects.

To see what a hook did, press Ctrl+O for the transcript view. For execution details, start Claude Code with claude --debug-file and a path, then read that file after triggering the hook.

A blocking hook is evaluated before allow rules. To run all Bash commands without prompts except a few, the documentation suggests allowing Bash and rejecting the specific commands in a PreToolUse hook.

To turn hooks off for one run regardless of project settings, start with claude --settings '{"disableAllHooks": true}'. Hooks delivered through managed settings keep running unless the managed level disables them.

WHEN NOT TO USE THIS

A throwaway prototype or a personal script folder with no shared branch, no deployment and no secrets, where the default permission prompts are enough.

As the only control over production access. If an agent must not reach production, restrict the credentials and network it has; rules and hooks limit Claude Code, not the account it runs as.

For guidance that needs judgment, such as style preferences or how to structure a change. Keep that in CLAUDE.md, and reserve hooks and deny rules for rules with zero exceptions.

SOURCES · CHECKED 2026-11-22

Claude Code docs: Hooks reference · Hook configuration shape (hooks object, event keys, matcher groups with matcher and hooks arrays); handler fields type, command, args, if, timeout (seconds; default 600 for command hooks), async, onFailure (v2.1.295 or later; no effect on Stop, SubagentStop, TaskCompleted, TeammateIdle); matcher evaluation (exact names or lists with | or , versus unanchored JavaScript regular expression; empty, omitted or * matches all; ignored on events without matcher support such as Stop); common input fields (session_id, transcript_path, cwd, permission_mode, hook_event_name); PreToolUse input (tool_name, tool_input, tool_use_id; tool_input.command for Bash; tool_input.file_path always absolute for Write, Edit, Read; backslash separators on Windows); exit code semantics (0 success, 2 blocking error, any other code a non-blocking error); the per-event exit code 2 table; JSON output rules (single JSON object on stdout, hookSpecificOutput with hookEventName); PreToolUse permissionDecision values allow, deny, ask, defer with permissionDecisionReason, and precedence deny, defer, ask, allow; PostToolUse cannot undo and exit 2 shows stderr to Claude; Stop input stop_hook_active and the eight consecutive continuation cap; Stop treats exit 2 with a missing-file stderr as non-blocking; CLAUDE_PROJECT_DIR placeholder and environment variable, unchanged when Claude enters a worktree; Bash|PowerShell matcher guidance for Windows; disableAllHooks behavior; command hooks run with full user permissions; workspace trust rules for hooks; the debug log flags --debug and --debug-file.
https://code.claude.com/docs/en/hooks

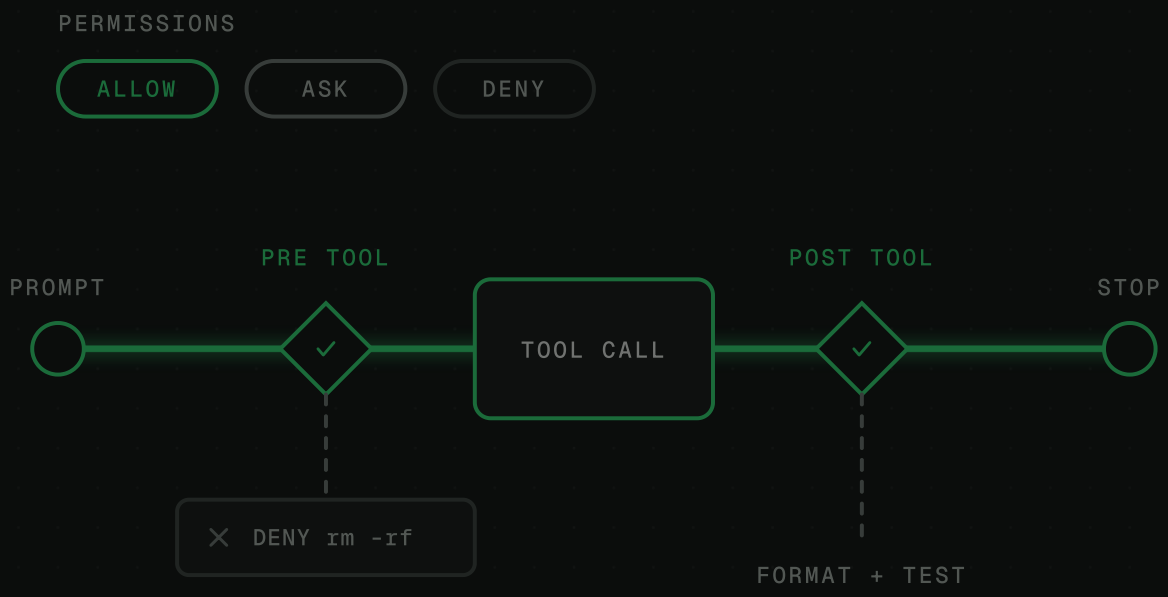
Claude Code docs: Automate actions with hooks (hooks guide) · The protect-files example this starter adapts (INPUT=$(cat), jq -r '.tool_input.file_path // empty', exit 2, the shell-form command "$CLAUDE_PROJECT_DIR"/.claude/hooks/protect-files.sh, chmod +x); the event list with when each fires; the matcher table per event; exit 0 on PreToolUse is not an approval; a hook allow does not bypass deny or ask rules and a hook deny holds in every permission mode including bypassPermissions; the if field uses permission rule syntax, takes one rule, works only on tool events and is best-effort; hook locations and scope; /hooks browser; Stop fires whenever Claude finishes responding and not on user interrupts; the stop_hook_active early-exit pattern; testing a script by piping sample JSON and checking echo $?; misplaced permissionDecision at the top level is ignored; output not starting with { is read as plain text; Ctrl+O transcript view; Edit|Write hooks do not see changes made by shell commands and git status --porcelain lists modified and untracked files.
https://code.claude.com/docs/en/hooks-guide

Claude Code docs: Settings files and precedence · The four settings files and their scope (~/.claude/settings.json, .claude/settings.json, .claude/settings.local.json, managed settings); precedence order managed, command line (--settings), project local, shared project, user; list keys such as permissions.allow merge across files; settings files are strict JSON; the $schema URL and that the schema can lag releases; edits to permissions and hooks reload in a running session; /status shows the Setting sources line; project allow rules wait for workspace trust while deny and ask rules apply right away; Claude Code keeps a settings.local.json it creates out of git, and a hand-created one must be added to .gitignore.
https://code.claude.com/docs/en/settings

Claude Code docs: Configure permissions · Rule format Tool or Tool(specifier); evaluation order deny, then ask, then allow, first match decides regardless of specificity; Bash wildcard behavior (* matches any text, placement after the subcommand, exact match without *); compound commands are split and deny or ask rules apply to any subcommand; Bash rules do not match other invocation forms and are not a security boundary; Read and Edit path pattern types (//path, ~/path, /path, path or ./path) and where /path anchors per settings source; bare filenames match at any depth; Write(path), Glob(path) and Grep(path) rules do not match; what Read and Edit deny rules cover and do not cover; a deny at any level cannot be allowed by another level; PreToolUse hooks run before the permission prompt, deny and ask rules are evaluated regardless of the hook result, and a blocking hook takes precedence over allow rules; /permissions lists rules with their source file; startup warning for deny or ask rules matching no known tool; -p and SDK sessions run repository hooks without a trust dialog.
https://code.claude.com/docs/en/permissions

Claude Code docs: Settings reference · The permissions object keys allow, ask, deny, defaultMode and disableBypassPermissionsMode; permissions.ask prompts even in a permission mode that would otherwise approve; the documented deny examples Read(./.env), Read(./.env.*), Read(./secrets/**) and Bash(curl *); disableBypassPermissionsMode takes the string "disable", works from any file and makes Claude Code reject --dangerously-skip-permissions.

https://code.claude.com/docs/en/settings-reference



FAISAL KARKOH / RESOURCES

More working files for building with AI.

Prompt packs, checklists, templates and playbooks for shipping reliable AI products. Free to download and reuse in your own work.

LIBRARY
faisalkarkoh.com/resources

WRITING
faisalkarkoh.com/blog

WORK TOGETHER
faisalkarkoh.com/contact