

Permissions, sandbox and security

Allow/ask/deny rules, permission modes, the sandboxed Bash with OS-level isolation, and how to let Claude run autonomously without handing over the keys to the house.

Updated: July 2026 · Source: official Anthropic documentation (code.claude.com/docs) · Vol. 5 of the Claude Code cheatsheet series.

1. The permission system

who decides what Claude may do

Three kinds of rules, managed via `/permissions` or in the settings files (committable to the repo for the whole team): **allow** = the tool runs without approval; **ask** = confirmation required every time; **deny** = forbidden. Evaluation order is **deny** → **ask** → **allow**: the first match wins, and specificity does *not* matter — a broad deny like `Bash(aws *)` blocks everything, even what a narrower allow would permit. A **bare-name** deny (`Bash`) removes the tool from Claude's context; a scoped one (`Bash(rm *)`) keeps the tool and blocks matching calls.

Who enforces what (the enforcement hierarchy)

Permission rules are enforced **by Claude Code**, not by the model: instructions in the prompt or in CLAUDE.md shape what Claude *tries* to do, but don't change what it is *allowed* to do. From weakest to strongest: CLAUDE.md (persuasion) → permission rules / PreToolUse hooks (client-side enforcement) → sandbox (OS-level enforcement, which also covers child processes).

Rule syntax

Rule	Matches
<code>Bash / WebFetch / Read</code>	Every use of the tool (equivalent to <code>Bash(*)</code>).
<code>Bash(npm run build)</code>	Exactly that command.
<code>Bash(npm run test *)</code>	Prefix + wildcard. The space before * is crucial: <code>Bash(ls *)</code> matches <code>ls -la</code> but not <code>ls -la</code> ; <code>Bash(ls*)</code> matches both. A single * also spans spaces: <code>Bash(git * main)</code> matches <code>git push origin main</code> .
<code>Read(./env) / Edit(/src/**/*.ts)</code>	Gitignore-style patterns. Anchors: <code>//path</code> = absolute from the root, <code>~/path</code> = home, <code>/path</code> = relative to the project root (classic trap!), <code>path</code> = relative to the cwd. A bare name like <code>Read(.env)</code> matches at any depth.
<code>WebFetch(domain:example.com)</code>	Fetches to that domain.
<code>Agent(model:opus) / Bash(run_in_background:true)</code>	Match on a top-level input parameter (deny and ask only). One parameter per rule; * as a wildcard in the value.
<code>mcp_github / mcp_*</code>	All tools of that MCP server / all MCP tools (deny/ask). Allows accept globs only after a literal <code>mcp_<server>_</code> prefix.

Compound commands and wrappers

- Claude Code recognizes `&&`, `||`, `;`, `|`, `&` and newlines: a rule must match **every** subcommand. `Bash(safe-cmd *)` does not authorize `safe-cmd && other`.
- Wrappers stripped before matching: `timeout`, `time`, `nice`, `nohup`, `stdbuf`, xargs without flags. **Not** stripped: `npm`, `docker exec`, `devbox run` ... — `Bash(devbox run *)` would also match `devbox run rm -rf .`: write the rule with runner + inner command.
- Built-in set of **read-only** commands (`ls`, `cat`, `grep`, `find`, `wc`, `diff`, `stat`, `du`, `cd`, `read-only git`...) that always run without prompting in every mode.

The curl case (why argument patterns are fragile)

`Bash(curl http://github.com/ *)` doesn't cover options before the URL, https, redirects, variables, double spaces. More robust: **deny** `curl/wget` + **WebFetch(domain:github.com)** for the allowed domains, or a PreToolUse hook that validates URLs. And remember: if Bash is allowed, the network is reachable anyway — the real boundary comes from the sandbox (ch. 3).
Read/Edit denies cover Claude's file tools and the file commands recognized in Bash (`cat`, `head`, `sed`...), **not** a Python script that opens files on its own: for that you need the sandbox.

2. Permission modes

Shift+Tab to cycle

Mode	Behavior
------	----------

default	Asks on the first use of each tool.
acceptEdits	Auto-accepts file edits and common filesystem commands (mkdir, touch, mv, cp) in the working directories.
plan	Plan mode: Claude explores read-only and presents a plan before touching anything. The safest mode for large changes.
auto	Approves calls with background safety checks that verify alignment with the request (research preview).
dontAsk	Denies everything not pre-approved by allow rules: the baseline for unattended runs and locked-down CI.
bypassPermissions	Skips the prompts (explicit ask rules still force a prompt, and <code>rm -rf /</code> or <code>~</code> remain a lifesaver that asks). Only in isolated environments: containers or VMs. Admins disable it with <code>permissions.disableBypassPermissionsMode: "disable"</code> in managed settings.

3. The Bash sandbox

operating-system-level
enforcement

The sandbox flips the approach: instead of approving each command, you define **which files and which domains** commands may touch, and the OS enforces the boundary on every Bash command **and its child processes**. On macOS it uses Seatbelt (nothing to install); on Linux and WSL2 you need `bubblewrap` and `socat` (`sudo apt-get install bubblewrap socat`). Enable and manage it with `/sandbox`.

Two modes

- **Auto-allow:** sandboxed commands run **without prompting**; whatever can't run sandboxed (e.g. network to non-allowed hosts) falls back to the normal permission flow. Explicit denies and scoped asks still apply; `rm` on `/` or `home` always asks.
- **Regular permissions:** same restrictions, but the prompts remain.

Default: writes only to the working directory and session temp; the first time a new network domain is needed, Claude Code asks. Escape hatch: if a command fails because of the sandbox, Claude can retry outside it (with your approval); disable with `"allowUnsandboxedCommands": false` (strict mode).

Ubuntu 24.04+: the AppArmor gotcha

The default AppArmor profile prevents bubblewrap from creating user namespaces. If `sysctl kernel.apparmor_restrict_unprivileged_userns` returns 1, add a profile for `bwrap`:

```
sudo tee /etc/apparmor.d/bwrap <<'EOF'
abi <abi/4.0>,
include <tunables/global>
profile bwrap /usr/bin/bwrap flags=(unconfined) {
    userns,
    include if exists <local/bwrap>
}
EOF
sudo systemctl reload apparmor
```

Configuration (settings.json)

```
{
  "sandbox": {
    "enabled": true,
    "filesystem": {
      "allowWrite": ["~/.kube", "/tmp/build"],
      "denyRead": ["~/"],
      "allowRead": ["."]
    },
    "credentials": {
      "files": [
        {"path": "~/.aws/credentials", "mode": "deny"},
        {"path": "~/.ssh", "mode": "deny"}
      ],
      "envVars": [
        {"name": "GITHUB_TOKEN", "mode": "mask"}
      ]
    }
  }
}
```

Note the prefixes: here `/tmp/build` is a plain absolute path (different syntax from the Read/Edit rules!). `mode: mask` for variables is elegant: the sandboxed command sees a sentinel value, and the proxy injects the real one only toward the declared hosts — `gh` and `npm` keep working without ever exposing the token.

◆ Choosing the isolation level

Option	What it isolates	When
Built-in Bash sandbox	Filesystem and network of Bash commands, at the OS level	Daily use with more autonomy and fewer prompts
Dev container / Docker	The entire execution environment	Team consistency, or to use <code>bypassPermissions</code> safely
VM	Everything, kernel included	

Prompt injection: the threat to keep in mind

External content (fetched web pages, third-party MCP output, issues, READMEs of cloned repos) can contain hostile instructions that try to hijack Claude. Practical defenses: verify you trust each MCP server before connecting it; keep denies on sensitive files (`Read(.env)` , `Read(~/.ssh/**)`); use the sandbox with denied/masked credentials to limit the potential damage; and for heavy autonomy prefer isolated environments. The built-in git protections still block destructive operations like an unrequested `git reset --hard` by default.

Sources: code.claude.com/docs — [permissions](#), [permission-modes](#), [sandboxing](#), [sandbox-environments](#), [security](#) · [/permissions](#) and [/sandbox](#) are the control panels.