Permessi, sandbox e sicurezza ◆

Regole allow/ask/deny, permission modes, il Bash sandboxato con isolamento a livello di OS, e come lasciare Claude autonomo senza dargli le chiavi di casa.

Aggiornato: luglio 2026 · Fonte: documentazione ufficiale Anthropic (code.claude.com/docs) · Vol. 5 della serie bignami Claude Code.

1. Il sistema dei permessi

chi decide cosa può fare Claude

Tre tipi di regole, gestibili da `/permissions` o nei file settings (committabili nel repo per tutto il team): **allow** = il tool gira senza approvazione; **ask** = chiede conferma ogni volta; **deny** = vietato. Valutazione nell'ordine **deny → ask → allow**: vince la prima che matcha, e la specificità *non* conta — un deny ampio come `Bash(aws *)` blocca tutto, anche ciò che un allow più stretto permetterebbe. Un deny **a nome nudo** (`Bash`) rimuove il tool dal contesto di Claude; uno scoped (`Bash(rm *)`) lascia il tool e blocca le chiamate corrispondenti.

Chi applica cosa (gerarchia dell'enforcement)

Le regole di permesso sono applicate **da Claude Code**, non dal modello: istruzioni nel prompt o in CLAUDE.md orientano ciò che Claude *prova* a fare, ma non cambiano ciò che gli è *consentito*. Dal più debole al più forte: CLAUDE.md (persuasione) → regole permessi / hook PreToolUse (enforcement del client) → sandbox (enforcement dell'OS, vale anche per i processi figli).

◆ Sintassi delle regole

Regola	Matcha
`Bash` / `WebFetch` / `Read`	Ogni uso del tool (equivale a `Bash(*)`).
`Bash(npm run build)`	Esattamente quel comando.
`Bash(npm run test *)`	Prefisso + wildcard. **Lo spazio prima di * è cruciale**: `Bash(ls *)` matcha `ls -la` ma non `lsof`; `Bash(ls*)` matcha entrambi. Un singolo * copre anche gli spazi: `Bash(git * main)` matcha `git push origin main`.
`Read(./.env)` / `Edit(/src/**/*.ts)`	Pattern in stile gitignore. Ancore: `//path` = assoluto dalla radice, `~/path` = home, `/path` = **relativo alla root del progetto** (trappola classica!), `path` = relativo alla cwd. Un nome nudo come `Read(.env)` matcha a qualsiasi profondità.
`WebFetch(domain:example.com)`	Fetch verso quel dominio.
`Agent(model:opus)` / `Bash(run_in_background:true)`	Match su un parametro top-level dell'input (solo per deny e ask). Un parametro per regola; * come wildcard nel valore.
`mcp__github` / `mcp__*`	Tutti i tool di quel server MCP / tutti i tool MCP (deny/ask). Gli allow accettano glob solo dopo un prefisso letterale `mcp__<server>__`.

Comandi composti e wrapper

- Claude Code riconosce `&&`, `||`, `;`, `|`, `&` e newline: una regola deve matchare **ogni** sottocomando. `Bash(safe-cmd *)` non autorizza `safe-cmd && altro`.
- Wrapper strippati prima del match: `timeout`, `time`, `nice`, `nohup`, `stdbuf`, xargs senza flag. **Non** strippati: `npx`, `docker exec`, `devbox run`... — `Bash(devbox run *)` matcherebbe anche `devbox run rm -rf .`: scrivi la regola con runner + comando interno.
- Set built-in di comandi **read-only** (ls, cat, grep, find, wc, diff, stat, du, cd, git in sola lettura...) che girano sempre senza prompt in ogni modalità.

Il caso curl (perché i pattern sugli argomenti sono fragili)

`Bash(curl http://github.com/ *)` non copre opzioni prima dell'URL, https, redirect, variabili, doppi spazi. Più solido: **deny** su curl/wget + **WebFetch(domain:github.com)** per i domini consentiti, oppure un hook PreToolUse che valida gli URL. E ricorda: se Bash è permesso, la rete è raggiungibile comunque — il confine vero lo mette la sandbox (cap. 3).

Le deny di Read/Edit coprono i tool file di Claude e i comandi file riconosciuti in Bash (cat, head, sed...), **non** uno script Python che apre i file da sé: per quello serve la sandbox.

2. Permission modes

Shift+Tab per ciclare

Mode	Comportamento

`default`	Chiede al primo uso di ogni tool.
`acceptEdits`	Accetta automaticamente le modifiche ai file e i comandi filesystem comuni (mkdir, touch, mv, cp) nelle directory di lavoro.
`plan`	**Plan mode**: Claude esplora in sola lettura e presenta un piano prima di toccare qualsiasi cosa. Il modo più sicuro per le modifiche grosse.
`auto`	Approva le chiamate con verifiche di sicurezza in background che controllano l'allineamento con la richiesta (research preview).
`dontAsk`	Nega tutto ciò che non è pre-approvato da regole allow: la baseline per run non presidiati e CI blindate.
`bypassPermissions`	Salta i prompt (le regole **ask** esplicite forzano comunque il prompt, e `rm -rf /` o `~` restano un salvavita che chiede). **Solo in ambienti isolati**: container o VM. Gli admin lo disattivano con `permissions.disableBypassPermissionsMode: "disable"` nei managed settings.

3. La sandbox Bash

enforcement a livello di sistema operativo

La sandbox ribalta l'approccio: invece di approvare ogni comando, definisci **quali file e quali domini** i comandi possono toccare, e l'OS applica il confine a ogni comando Bash **e ai suoi processi figli**. Su macOS usa Seatbelt (nulla da installare); su Linux e WSL2 servono `bubblewrap` e `socat` (`sudo apt-get install bubblewrap socat`). Si attiva e gestisce con `/sandbox`.

Due modalità

- **Auto-allow**: i comandi sandboxati girano **senza prompt**; ciò che non può girare sandboxato (es. rete verso host non consentiti) ricade nel normale flusso di permessi. Le deny esplicite e le ask scoped valgono comunque; rm su / o home chiede sempre.
- **Regular permissions**: stesse restrizioni, ma i prompt restano.

Default: scrittura solo su working directory e temp di sessione; la prima volta che serve un dominio di rete nuovo, Claude Code chiede. Escape hatch: se un comando fallisce per la sandbox, Claude può riprovare fuori (con tua approvazione); disattivabile con `"allowUnsandboxedCommands": false` (strict mode).

Ubuntu 24.04+: il gotcha AppArmor

Il profilo AppArmor di default impedisce a bubblewrap di creare user namespace. Se `sysctl kernel.apparmor_restrict_unprivileged_userns` ritorna 1, aggiungi un profilo per 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
```

Configurazione (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"}
      ]
    }
  }
}
```

Nota i prefissi: qui `/tmp/build` è un normale path assoluto (sintassi diversa dalle regole Read/Edit!). `mode: mask` per le variabili è elegante: il comando sandboxato vede un valore sentinella, e il proxy inietta quello vero solo verso gli host dichiarati — gh e npm continuano a funzionare senza mai esporre il token.

◆ Scegliere il livello di isolamento

Opzione	Cosa isola	Quando
Bash sandbox integrata	Filesystem e rete dei comandi Bash, a livello OS	Uso quotidiano con più autonomia e meno prompt
Dev container / Docker	L'intero ambiente di esecuzione	Coerenza di team, o per usare bypassPermissions in sicurezza
VM	Tutto, kernel incluso	Massima diffidenza: codice non fidato, agenti totalmente autonomi

Prompt injection: la minaccia da tenere a mente

Contenuti esterni (pagine web fetchate, output di MCP di terzi, issue, README di repo clonati) possono contenere istruzioni ostili che provano a dirottare Claude. Difese pratiche: verifica di fidarti di ogni server MCP prima di connetterlo; tieni deny su file sensibili (`Read(.env)`, `Read(~/.ssh/**)`); usa la sandbox con credenziali negate/mascherate per limitare il danno potenziale; e per l'autonomia spinta preferisci ambienti isolati. Le protezioni git integrate bloccano comunque di default operazioni distruttive come `git reset --hard` non richiesto.

Fonti: code.claude.com/docs — permissions, permission-modes, sandboxing, sandbox-environments, security · /permissions e /sandbox sono i pannelli di controllo.