Permissions and sandboxing: boundary and approval

Key takeaways

1. Every permission system answers two questions: what it can touch, when to ask.
2. Claude permissions ask and block at the tool layer; sandbox bounds subprocesses.
3. Codex splits cleanly: sandbox_mode is the boundary, approval_policy is the ask.

Common misunderstandings of agent permissions

Which claim would you defend right now? Choose before the evidence.

"A deny rule like Bash(rm -rf*) means the agent can never delete my files."

"Permission prompts are the safety mechanism, so the more often the agent asks, the safer I am."

"The sandbox is on by default — an agent I install is already confined to my project."

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Two questions every permission system answers

Configure each on purpose, once, instead of clicking through prompts

What can it touch?

The boundary. Filesystem and network limits that hold even when the agent is wrong about what it is doing. Enforced by the OS, not by good intentions.

Codex: sandbox_mode. Claude: sandboxed Bash (/sandbox).

When must it ask?

The approval. Which actions pause for a human decision and which proceed. This is where prompt fatigue is created — and where it is fixed.

Codex: approval_policy. Claude: permissions rules.

The map is layered, not one mechanism per question: Claude permissions answer both at the tool layer; sandboxes extend the boundary below it.

Claude Code: the permissions block

allow, ask, and deny rules over tools, plus a default mode

claude/settings.json — rule syntax is Tool or Tool(specifier)

```
"permissions": {
  "allow": ["Read", "Grep", "Bash(git status*)"],   # run without asking
  "ask":   ["Bash(git push*)"],                     # always pause here
  "deny":  ["Read(~/.ssh/**)", "Bash(sudo*)"],      # blocked outright
  "defaultMode": "acceptEdits",     # what happens with no matching rule
  "additionalDirectories": ["~/notes"]      # widen the reachable tree
}
```

- allow and ask decide when Claude asks; deny rules and Read/Edit path rules genuinely block the tool call.
- defaultMode covers everything no rule matches: default asks for writes, acceptEdits auto-approves file edits, plan only plans.
- None of these rules constrain what a Bash subprocess does once it is running — that is the next slide.

What a rule can and cannot stop

Tool-layer rules govern tool calls, not the processes they start

Rule	Genuinely stops	Cannot stop
deny “Read(~/.ssh/**)”	Claude’s Read tool on your keys	cat ~/.ssh/id_ed25519 inside Bash
deny “Bash(rm -rf*)”	commands that start with rm -rf	bash -c ‘rm -rf …’ or a script that deletes
allow “Bash(git status*)”	prompt fatigue for safe reads	nothing — allow rules only skip the ask

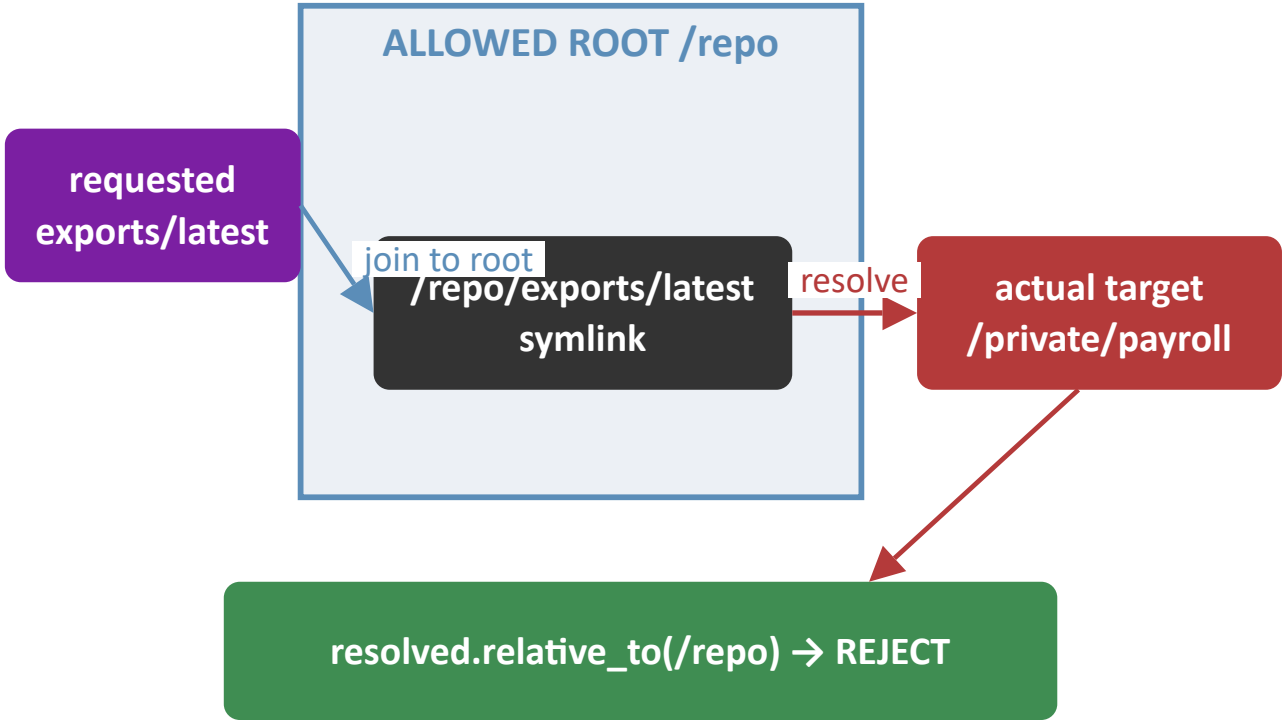
Command-string deny patterns are guardrails against accidents, not security against a determined subprocess.

A path string is not a filesystem boundary

Resolve .. and symlinks before proving a host tool stays beneath its allowed root

Resolve the object before checking the boundary

```
def confined(root: Path, requested: str) ->
Path:
    root = root.resolve()
    path = (root / requested).resolve()
    path.relative_to(root)  # raises on escape
    if not path.is_file():
        raise PolicyError("not_a_file")
    return path
```



- A schema validates a string, not where the string resolves.
- Lexical checks miss both ../ and in-root symlinks.
- The OS sandbox remains the lower boundary if host validation fails.

The name is inside /repo; the object is outside it.

Claude Code sandbox: closing the subprocess bypass

An OS-enforced boundary under the tool layer, off by default

- The sandbox confines Bash and everything Bash starts: filesystem writes and network access are limited by the OS (Seatbelt on macOS, bubblewrap on Linux), not by pattern matching.
- It is disabled by default; /sandbox in a session shows and changes its state.
- With the sandbox on, permission rules become the fine control on top of a real fence; without it, they are the only layer you have.

Permissions decide when Claude asks. The sandbox decides what a subprocess can touch. You want both on purpose.

Where rules live

Student view: three files; rules merge, deny wins anywhere

File	Scope	Typical use
~/.claude/settings.json	you, every project	your personal allow list and guardrails
.claude/settings.json	everyone in this repo	project rules, committed and reviewed
.claude/settings.local.json	you, this repo	personal overrides, gitignored

- Permission arrays merge across scopes — they do not override like scalar settings — and a matching deny at any scope wins.
- Managed (organization) settings and CLI flags sit above all three; this table is the subset a student edits.

Worked example: my actual settings.json

Calibrated risk, annotated honestly — not a starting point

the instructor's laptop — read the annotations before copying anything

```
"allow": ["Bash", "WebSearch", "Read", "Glob", "Grep", …],
         # bare Bash auto-approves EVERY command
"deny":  ["Bash(rm -rf /)", "Bash(rm -rf ~)", "Bash(sudo rm -rf*)",
          "Bash(git push --force*)", "Bash(dd if=* of=/dev/*)",
          "Bash(mkfs*)", …],          # ~20 accident guardrails
"defaultMode": "acceptEdits",
"additionalDirectories": ["~/git-papers", "~/dotfiles", "~/git", …]
```

- Why I run this loose: everything I care about is committed and pushed, the deny list catches the classic disasters, and I review diffs — the compensating controls are habits, not the config.
- Why you should not start here: bare Bash in allow removes the ask for every command, and the deny list cannot block what it does not match.

Your first permissions block

Allow what you can defend, deny the disasters, keep the ask

a defensible start: reads are free, writes still ask outside edits

```
"permissions": {
  "allow": ["Read", "Glob", "Grep",
            "Bash(git status*)", "Bash(git diff*)", "Bash(git log*)",
            "Bash(ls*)", "Bash(rg*)"],
  "deny":  ["Bash(sudo*)", "Bash(git push --force*)",
            "Read(~/.ssh/**)", "Read(**/secrets/**)"],
  "defaultMode": "acceptEdits"
}
```

Add allow rules one at a time, when a prompt annoys you and you can say why skipping it is safe.

Codex: two orthogonal knobs

sandbox_mode is the boundary; approval_policy is the ask

Knob	Values	Question it answers
sandbox_mode	read-only · workspace-write · danger-full-access	what can it touch
approval_policy	untrusted · on-request · never	when must it ask

~/.codex/config.toml — danger-full-access is never your default

```
approval_policy = "on-request"      # the course seed — also the
sandbox_mode    = "workspace-write" # fresh-install default

[sandbox_workspace_write]
writable_roots = ["~/git-class"]  # widen deliberately, one root at a time
network_access = true             # default false — installs fail without it
```

Permission profiles: the successor (beta)

Know it exists; do not mix it with the sandbox keys

- default_permissions and [permissions.*] tables are the beta successor to sandbox_mode and [sandbox_workspace_write] — named, reusable boundary profiles.
  - The API is not stable and is explicitly labeled beta — which is why this course does not use it.
- approval_policy survives unchanged and composes with profiles; only the sandbox keys are replaced.
- Never configure both systems at once: if the old sandbox keys are present, Codex uses them and ignores your profiles.

Same config, different fences

One policy model on every surface; enforcement is platform-native

Platform	Boundary mechanism	What you notice
macOS — CLI, IDE, desktop app	Seatbelt (sandbox-exec)	nothing: it is invisible, which is why the app feels simple
Linux and WSL2	bubblewrap	may need one package install before the sandbox works
Windows native, elevated	dedicated low-privilege sandbox users	admin setup creates local accounts — that is not malware
Windows native, unelevated	restricted token from your own account	weaker fallback for locked-down machines

On Windows, WSL2 is usually the calmer path: Linux semantics, bubblewrap, no local sandbox accounts.

HW0: make your permissions tolerable

You and your agent author the policy; self-check verifies it

- Author a permissions block in claude/settings.json — portable JSON in your dotfiles repo; no Claude subscription required to write and commit it.
- Set your Codex policy in codex/config.seed.toml (the tracked source of truth) and mirror it into ~/.codex/config.toml.
- Run the probe from guides/permissions.md: ask Codex to write one file outside your writable roots, deny the prompt, confirm nothing appeared.
- Review every rule line by line; reject anything you would not defend. Record the rule you debated and the probe outcome in SUBMISSION.md.

./self-check prints a TODO line for anything still missing.

Key Takeaways

1. Boundary and approval are different controls: configure what the agent can touch and when it must ask, separately and on purpose.
2. Claude rules ask and block at the tool layer and only a sandbox bounds subprocesses; Codex splits the two questions into sandbox_mode and approval_policy.
3. A permission prompt you click without reading is configuration debt: fix the rule, not the habit.