# Rules

Use rules to control which commands Codex can run outside the sandbox.

Rules are experimental and may change.

## Create a rules file

1.  Create a `.rules` file under `~/.codex/rules` (for example, `~/.codex/rules/default.rules`).

2.  Add a rule. This example prompts before allowing `gh pr view` to run outside the sandbox.

```
# Prompt before running commands with the prefix `gh pr view`
        outside the sandbox.
prefix_rule(
    # The prefix to match.
    pattern = ["gh", "pr", "view"],

    # The action to take when Codex requests to run a matching
        command.
    decision = "prompt",

    # Optional rationale for why this rule exists.
    justification = "Viewing PRs is allowed with approval",

    # `match` and `not_match` are optional "inline unit tests"
        where you can
    # provide examples of commands that should (or should not)
        match this rule.
    match = [
        "gh pr view 7888",
        "gh pr view --repo openai/codex",
        "gh pr view 7888 --json title,body,comments",
    ],
    not_match = [
        # Does not match because the `pattern` must be an exact
        prefix.
```

```
        "gh pr ——repo openai/codex view 7888",
    ],
)
```

3. Restart Codex.

Codex loads every `*.rules` file under `~/.codex/rules` at startup. When you add a command to the allow list in the TUI, Codex appends a rule to `~/.codex/rules/default.rules` so future runs can skip the prompt.

# Understand rule fields

`prefix_rule()` supports these fields:

- `pattern` **(required)**: A non-empty list that defines the command prefix to match. Each element is either:
    - A literal string (for example, `"pr"`).
    - A union of literals (for example, `["view", "list"]`) to match alternatives at that argument position.
- `decision` **(defaults to `"allow"`)**: The action to take when the rule matches. Codex applies the most restrictive decision when more than one rule matches (`forbidden` > `prompt` > `allow`).
    - `allow`: Run the command outside the sandbox without prompting.
    - `prompt`: Prompt before each matching invocation.
    - `forbidden`: Block the request without prompting.
- `justification` **(optional)**: A non-empty, human-readable reason for the rule. Codex may surface it in approval prompts or rejection messages. When you use `forbidden`, include a recommended alternative in the justification when appropriate (for example, `"Use \rg` instead of `grep`.”`).
- `match` and `not_match` **(defaults to `[]`)**: Examples that Codex validates when it loads your rules. Use these to catch mistakes before a rule takes effect.

When Codex considers a command to run, it compares the command's argument list to `pattern`. Internally, Codex treats the command as a list of arguments (like what `execvp(3)` receives).

# Shell wrappers and compound commands

Some tools wrap multiple shell commands into a single invocation, for example:

```
["bash", "—lc", "git add . && rm —rf /"]
```

Because this kind of command can hide multiple actions inside one string, Codex treats `bash -lc`, `bash -c`, and their `zsh` / `sh` equivalents specially.

## When Codex can safely split the script

If the shell script is a simple, linear chain of commands made only of:

- plain words (no variable expansion, no `VAR=...`, `$FOO`, `*`, etc.)
- joined by safe operators (`&&`, `||`, `;`, or `|`)

then Codex parses it (using tree-sitter) and splits it into individual commands before applying your rules.

So the script above is treated as two separate commands:

- `["git", "add", "."]`
- `["rm", "-rf", "/"]`

Codex then evaluates each command against your rules, and the most restrictive result wins.

So even if you allow `pattern=["git", "add"]`, Codex will not auto-allow `git add . && rm -rf /`, as the `rm -rf /` portion is evaluated separately and prevents the whole invocation from being auto-allowed.

This prevents dangerous commands from being smuggled in alongside safe ones.

## When Codex does not split the script

If the script uses more advanced shell features, such as:

- redirection (`>`, `>>`, `<`)
- substitutions (`$(...)`, `` `...` ``)
- environment variables (`FOO=bar`)
- globbing (`*`, `?`)
- control flow (`if`, `for`, `&&` with assignments, etc.)

then Codex does not try to interpret or split it.

In those cases, the entire invocation is treated as:

`["bash", "-lc", "<full script>"]`

and your rules are applied to that **single** invocation.

With this special handling, you get the security of per-command evaluation when it's safe to do so, and conservative behavior when it isn't.

## Test a rule file

Use `codex execpolicy check` to test how your rules apply to a command:

```
codex execpolicy check --pretty \
  --rules ~/.codex/rules/default.rules \
  -- gh pr view 7888 --json title,body,comments
```

The command emits JSON showing the strictest decision and any matching rules, including any `justification` values from matched rules. Use more than one `--rules` flag to combine files, and add `--pretty` to format the output.

## Understand the rules language

The `.rules` file format uses `Starlark` (see the language spec). Its syntax is like Python, but it's designed to be safe to run: the rules engine can run it without side effects (for example, touching the filesystem).