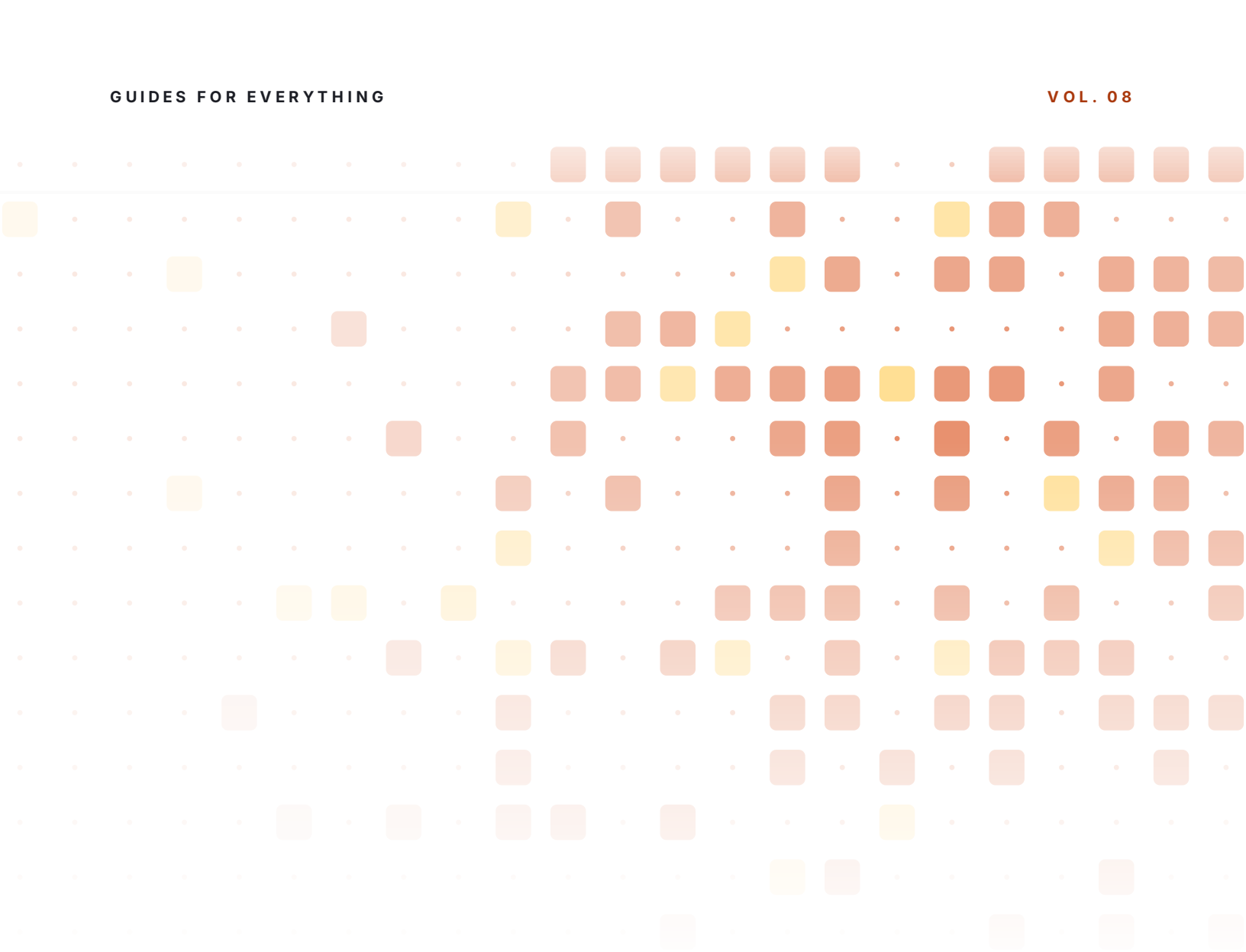




Maximum Autonomy

Every setting, flag, config file, and token choice for
hands-off AI agents, tool by tool

A POWER-USER'S GUIDE

CLAUDE EDITION

Maximum Autonomy

Every setting, flag, config file, and token choice for hands-off AI agents, tool by tool

CLAUDE EDITION Expanded edition written by Claude (Anthropic's AI) in Claude Code and published in GuidesForEverything PR #4.

This guide is for one goal: letting your AI agents work **hands-off**, with as few approval prompts as possible, across Claude Code, the Claude apps, Cursor, VS Code and Copilot, Codex, Gemini CLI, GitHub, MCP servers, n8n, and Notion.

You've decided you accept the risk, so this guide doesn't argue with you. It gives you the exact switches. It also shows the one habit that makes maximum autonomy sustainable: **contain the agent instead of restricting it**. Give it a box where mistakes are cheap, then remove the guardrails inside that box. Each chapter ends with what *still* prompts, no matter what you set.

HOW TO READ THIS MANUAL

Tip — a shortcut or good habit.

Warning — easy to get wrong.

\$ **command** — text you type, exactly as shown.

Note — useful background.

Danger — can lose data, money, or access.

Settings > **Privacy** — a menu path to click.

New words appear in **bold** the first time and are collected in the Glossary at the back.

Settings, flags, and menu paths were checked against each vendor's official documentation in October 2026. These products change monthly, and safety prompts in particular are added and renamed often. If a setting name doesn't match what you see, check the vendor's current docs. You are responsible for what your agents do with the access you grant.

Contents

01 The Autonomy Playbook	8
1.1 A few words before we start	8
1.2 Four levels of autonomy	9
1.3 Contain, don't restrict	10
1.4 Choose a level for each project	10
1.5 Quick-start: maximum autonomy at a glance	11
02 Build a Safe Playground	13
2.1 Pick your box	13
2.2 Option 1: a dev container (recommended for coding agents)	14
2.3 Option 2: Claude Code's built-in Bash sandbox	16
2.4 Option 3: the sandbox runtime	17
2.5 Option 4: a plain Docker container	17
2.6 Option 5: a virtual machine or spare computer	18
2.7 Option 6: cloud sessions	18
2.8 Network allowlists: who can the agent talk to?	19
2.9 Option 7: a separate user account on your computer	19
2.10 Troubleshooting the box	19
03 Accounts, Tokens, and Spend Caps	20
3.1 Give agents their own identity	20
3.2 Tokens: narrow but complete	21
3.3 Where each agent looks for credentials	21
3.4 Subscription tokens vs API keys	22
3.5 Spend caps: the seat belt for your wallet	22
3.6 Keep secrets out of reach	23
04 Claude Code (Terminal)	24
4.1 The settings files	24
4.2 The six permission modes	25
4.3 Level 3: auto mode by default	25
4.4 Level 4: bypass permissions	27
4.5 Protected paths	28
4.6 Critical paths: the deletion circuit breaker	29
4.7 Permission rules	29
4.8 Hooks: answer prompts with a script	30
4.9 A complete maximum-autonomy settings file	31
4.10 Reduce prompts without bypass	32
4.11 What still prompts in Claude Code, no matter what	32

05 Claude Code Without You: Headless, CI, SDK, and Cloud	33
5.1 Print mode: one command, no chat window	33
5.2 Background agents	34
5.3 Git worktrees: parallel agents without collisions	35
5.4 The Claude Code GitHub Action	35
5.5 The Agent SDK	37
5.6 Cloud sessions at claude.ai/code	38
5.7 Routines and scheduled tasks	39
06 Claude Desktop, Cowork, and claude.ai	40
6.1 The Code tab in Claude Desktop	40
6.2 Cowork	41
6.3 Connectors in claude.ai and the Claude apps	41
6.4 Claude in Chrome	42
6.5 Claude Code in VS Code	42
6.6 Desktop scheduled tasks	43
6.7 Troubleshooting the Claude apps	43
07 OpenAI Codex	44
7.1 Two controls that work together	44
7.2 The permission modes	45
7.3 Level 3 from the command line	45
7.4 Level 4: no prompts, no sandbox	46
7.5 Every flag that matters	46
7.6 Profiles: one file per autonomy level	47
7.7 Trust your projects	47
7.8 MCP servers without prompts	47
7.9 Protected paths	48
7.10 Scripts and CI: <code>codex exec</code>	48
7.11 The Codex GitHub Action	49
7.12 Codex Cloud	49
7.13 Notifications	50
7.14 What still asks or pauses in Codex	50
08 Gemini CLI	51
8.1 Approval modes	51
8.2 Pre-approve specific tools	52
8.3 MCP servers you trust	53
8.4 Folder trust	53
8.5 The sandbox	53
8.6 Headless mode	54
8.7 The GitHub Action	55
8.8 What still asks in Gemini CLI	55
09 VS Code and GitHub Copilot	56

9.1	Copilot agent in VS Code	56
9.2	Copilot CLI	57
9.3	Copilot cloud agent (on GitHub)	59
9.4	What still asks in VS Code and Copilot	59
10	Cursor	60
10.1	The editor's agent	60
10.2	The Cursor CLI	62
10.3	Cloud Agents	63
10.4	What still asks in Cursor	63
11	More Coding Agents	64
11.1	Devin Desktop (formerly Windsurf)	64
11.2	Devin CLI	65
11.3	Devin (cloud)	66
11.4	Cline	66
11.5	Aider	66
11.6	OpenHands	67
11.7	Goose	67
11.8	Browser app builders	68
11.9	Gone, or check for yourself	68
12	GitHub: Tokens and Repository Settings	69
12.1	Authenticate agents broadly	69
12.2	Repository settings that remove friction	71
12.3	Branch protection that agents can live with	71
12.4	What still prompts on GitHub	72
13	MCP Servers and Connectors	73
13.1	Give servers enough power	74
13.2	What MCP is, in one paragraph	74
13.3	Approve servers, then tools	74
14	n8n	76
14.1	Let AI agents build and run your workflows	76
14.2	Self-hosted: unlock everything	76
14.3	n8n Agents	77
14.4	Run n8n itself safely	77
15	Notion	78
15.1	Notion MCP (simplest, full personal access)	78
15.2	An API connection with every capability	78
15.3	Notion AI agents	78
15.4	Which option to use	79
16	Browsing Agents	80

16.1	What always stops for you	80
16.2	Reduce the prompts you can	81
16.3	Build an agent browser profile	81
17	Running Unattended	82
17.1	Keep a session alive with tmux	82
17.2	Schedules on your own machine: cron	83
17.3	A wrapper script with seat belts	83
17.4	Schedules in CI	84
17.5	Vendor clouds	84
17.6	Long-running sessions	84
18	Safety Nets	86
18.1	Git is your undo button	86
18.2	Checkpoints built into the tools	87
18.3	Backups for everything else	87
18.4	Kill switches	87
18.5	Budgets as the last net	88
19	Monitoring Agents	89
19.1	Get told when an agent needs you	89
19.2	Logs and transcripts	91
19.3	Dashboards with OpenTelemetry	91
19.4	A five-minute daily review	92
20	Multi-Agent Setups	93
20.1	One worktree per agent	93
20.2	A tmux layout for three agents	94
20.3	Subagents and agent teams	94
20.4	Cloud fleets	94
20.5	Rules that prevent collisions	94
21	The Master Comparison	96
21.1	Where the settings live	96
21.2	Levels, side by side	96
21.3	Headless commands	97
21.4	Built-in sandboxes and clouds	97
21.5	What can't be turned off	98
22	Copy-Paste Config Files	99
22.1	Claude Code	99
22.2	Codex	99
22.3	Gemini CLI	100
22.4	VS Code and Copilot	100
22.5	Cursor CLI	100
22.6	Devin CLI	101

22.7 Aider	101
22.8 Goose	101
22.9 A dev container for any agent	102
23 Troubleshooting	103
23.1 Claude Code	103
23.2 Codex	104
23.3 Gemini CLI, Copilot, Cursor, and others	104
23.4 General	105
24 What Always Asks: The Master List	106
Your Autonomy Setup Checklist	108
Glossary	109

The Autonomy Playbook

An **AI agent** is an AI model that doesn't just answer questions: it takes actions. It reads files, edits code, runs commands in a terminal, browses websites, and calls other services on your behalf. Every agent tool ships with a safety habit built in: before it does something that could cause harm, it stops and asks you. That pause is called a **permission prompt** (or an approval prompt).

Prompts are sensible defaults, but they get in the way of the thing that makes agents valuable: working while you do something else. This guide shows you, tool by tool, how to remove those pauses, and how to set up your computer and accounts so that removing them is a reasonable bet rather than a reckless one.

IN THIS CHAPTER

- The words you need: agent, prompt, mode, sandbox, token
- The four levels of agent autonomy
- “Contain, don't restrict”: the principle behind this guide
- How to choose a level for each project
- The quick-start setup for each tool

1.1 A few words before we start

You'll meet these terms on almost every page. Each one is in the glossary at the back too.

TERM	PLAIN-ENGLISH MEANING
Permission mode	A preset that decides which actions ask first
Allowlist	A list of actions pre-approved to run without asking
Deny rule	A rule that blocks an action in every mode
Sandbox	A walled-off area that limits what files and websites a command can reach
Token	A long secret string that lets a program act as you on a service
Headless	Running an agent from a script, with no chat window
Config file	A text file (often JSON or TOML) where a tool keeps its settings

TIP

JSON and **TOML** are just two formats for settings files. JSON uses curly braces and quotes (`{"key": "value"}`); TOML uses `key = "value"` lines. You can edit both in any text editor. If a tool refuses to start after an edit, a missing comma or quote is the usual culprit.

1.2 Four levels of autonomy

Every agent tool offers some version of these four levels:

LEVEL	WHAT HAPPENS	EXAMPLES
1. Ask	The agent asks before most actions	Claude Code <i>Manual</i> , Cursor <i>Allowlist</i> with an empty list
2. Allowlist	Pre-approved actions run; everything else asks	Allow rules, terminal allowlists
3. Classifier	A second AI reviews each action and blocks only risky ones	Claude Code <i>Auto</i> , Cursor <i>Auto-review</i> , Codex <i>Approve for me</i>
4. Bypass	Everything runs. No checks except a few hard-coded ones	<code>--dangerously-skip-permissions</code> , Cursor <i>Run Everything</i> , Codex <code>--yolo</code>

Level 3 gets you about 95% of the hands-off feel while still catching disasters. Level 4 is true maximum autonomy. This guide gives you both, so you can choose per tool and per project.

What a classifier actually does

A **classifier** is a second, smaller AI model that looks at each action the main agent wants to take and answers one question: “Is this safe, and is it what the person asked for?” If yes, the action runs with no prompt. If no, it’s blocked, and the agent is told to find another way. Classifiers are good at catching obvious disasters (deleting a database, sending secrets to a strange website) and at spotting **prompt injection**. They are not perfect, and every vendor says so.

1.3 Contain, don't restrict

Prompts exist to stop an agent from damaging things that matter. You can get the same protection a better way: put the agent somewhere where **nothing it can reach matters too much**. Then you can safely turn the prompts off.

- **A disposable machine:** a container, a virtual machine, a cloud dev box, or a spare computer.
- **Separate accounts:** a dedicated GitHub account or bot for the agent, a separate Notion workspace or teamspace, a separate n8n instance.
- **Narrow-but-complete tokens:** full power *inside* the agent's projects, nothing outside them.
- **Budgets:** spending caps on every API and cloud account.
- **Backups and Git:** everything committed, everything restorable.

Chapter 2 shows how to set this up in an afternoon, and Chapter 3 covers tokens and money.

☆ IMPORTANT

The single biggest real-world risk of hands-off agents is **prompt injection**: instructions hidden in a web page, email, issue, or document that the agent reads and obeys. Containment limits the damage it can do. Bypass settings remove the last line of defense against it.

1.4 Choose a level for each project

You don't need one setting for your whole life. Most people end up with a mix.

SITUATION	SENSIBLE LEVEL	WHY
Throwaway prototype in a container	4. Bypass	Nothing to lose
Your main project on your laptop	3. Classifier	Speed, with a safety net
Scheduled job in CI	2. Allowlist, or 4 inside the runner	Predictable, no human present
Anything touching production or money	1. Ask, or 3 with deny rules	Mistakes are expensive
Agent that reads email or the web	3. Classifier, in a box	Prompt injection is likely

DECIDE IN TWO MINUTES

- 1 Ask what the agent can reach.**
List the folders, accounts, and tokens available where it runs.
- 2 Ask what the worst mistake would cost.**
A deleted test folder is cheap; a leaked cloud key is not.

3

Shrink the reach first.

Move the work into a container or give it a narrower token (Chapters 2 and 3).

4

Then raise the level.

Once the worst case is cheap, pick Level 3 or 4.

1.5 Quick-start: maximum autonomy at a glance

TOOL	HANDS-OFF WITH SAFETY REVIEW	FULL BYPASS
Claude Code (CLI)	<code>"defaultMode": "auto" in ~/.claude/settings.json</code>	<code>claude --dangerously-skip-permissions</code>
Claude Desktop, Code tab	Mode selector, then Auto	Settings > Claude Code > Allow bypass permissions mode , then Bypass permissions
Claude Cowork	Mode selector, then Automatically approve	(no separate bypass)
claude.ai connectors	Each tool set to Always allow	(same)
Cursor	Approvals & Execution: Auto-review	Approvals & Execution: Run Everything
Cursor CLI	Allow rules in <code>cli-config.json</code>	<code>agent --force</code> (alias <code>--yolo</code>)
VS Code + Copilot	Assisted permissions (experimental)	Allow all / Autopilot , <code>chat.tools.global.autoApprove</code>
Copilot CLI	<code>/permissions assisted</code> or allow rules	<code>copilot --yolo</code> (alias of <code>--allow-all</code>)
OpenAI Codex	<code>codex --sandbox workspace-write</code> or <code>codex --approve-for-me</code>	<code>codex --yolo</code> , or <code>approval_policy = "never" + sandbox_mode = "danger-full-access"</code>
Gemini CLI	<code>--approval-mode=auto_edit</code>	<code>gemini --approval-mode=yolo</code>
Devin Desktop (Windsurf)	Auto-execution Auto	Auto-execution Turbo
Devin CLI	<code>--permission-mode smart</code>	<code>--permission-mode bypass</code>
Cline	Auto Approve toggles	Settings > Features > YOLO Mode
Aider	(asks only to confirm)	<code>aider --yes-always</code>
OpenHands CLI	<code>--llm-approve</code>	<code>--always-approve</code> , or <code>--headless</code>
Goose	<code>/mode smart_approve</code>	<code>GOOSE_MODE=auto</code> (the default)
GitHub	Bot account + fine-grained token	Broad token + Actions write permissions + auto-merge
n8n	Instance-level MCP + API key	Plus <code>NODES_EXCLUDE=""</code> on self-hosted

Notion	Notion MCP (OAuth)	Internal connection with all capabilities, shared at the top level
---------------	--------------------	--

- ☐ I know which projects get Level 3 and which get Level 4
- ☐ I know what my agent can reach where it runs
- ☐ I've read the prompt-injection warning and plan to contain agents that read the web or email

Build a Safe Playground

Fifteen minutes of setup here lets you run everything else in this guide at full throttle. The idea is simple: if the agent works inside a box, the worst it can do is wreck the box. You throw the box away and make a new one.

IN THIS CHAPTER

- The six kinds of “box” and how they compare
- A dev container step by step, with a working `devcontainer.json`
- Claude Code’s built-in Bash sandbox and the sandbox runtime
- Plain Docker, virtual machines, and cloud sandboxes
- Network allowlists: deciding which websites the agent can reach

2.1 Pick your box

BOX	WHAT IT ISOLATES	NEEDS DOCKER	EFFORT
Built-in Bash sandbox	Shell commands only	No	Minimal
Sandbox runtime	The whole agent process	No	Low
Dev container	A full dev environment	Yes	Medium
Custom Docker container	A full dev environment	Yes	Medium
Virtual machine	A whole operating system	No	High
Cloud session	A whole OS, hosted for you	No	None

The first two run on your own computer with operating-system guardrails. The rest put the agent inside something separate. For bypass mode, Anthropic’s guidance is clear: always use a container, a VM, or the sandbox runtime, so that file tools, plugins, and MCP servers are inside the boundary too.

⚠ WARNING

No box is perfect. If the agent can reach the internet, it can still leak whatever it can read. If your project folder is mounted writable, the agent can still change that code. Isolation shrinks the damage; it doesn't erase it.

2.2 Option 1: a dev container (recommended for coding agents)

A **dev container** is a Docker container configured as a complete development environment. **Docker** is a free program that runs lightweight, disposable “mini-computers” called **containers**. Anthropic publishes a reference dev container for Claude Code that runs as a **non-root user** (an ordinary account, not the all-powerful administrator) with a network firewall, which is exactly the setup bypass mode is designed for.

RUN AGENTS IN A DEV CONTAINER

- 1 **Install Docker Desktop** (docker.com) and **VS Code** with the **Dev Containers** extension.
- 2 **Add a `.devcontainer/` folder** to your project with the file below.
- 3 **Open the project in the container.**
Command Palette (`Cmd` + `Shift` + `P` or `Ctrl` + `Shift` + `P`), then **Dev Containers: Rebuild Container**.
- 4 **Sign in.**
Open a terminal in the container and run `claude`.
- 5 **Run your agent in bypass mode.**
If it goes wrong, rebuild the container; your host machine is untouched.

The smallest working file uses Anthropic's **Dev Container Feature**, an add-on that installs Claude Code into any image:

`.DEVCONTAINER/DEVCONTAINER.JSON`

```
{
  "image": "mcr.microsoft.com/devcontainers/base:ubuntu",
  "features": {
    "ghcr.io/anthropics/devcontainer-features/claude-code:1.0": {}
  }
}
```

The `:1.0` pins the installer script, not Claude Code itself; the feature installs the latest release.

Keep your sign-in across rebuilds

By default, a rebuild wipes the container's home folder, so you'd sign in every time. Store Claude Code's config folder in a Docker **volume** (a storage area that survives rebuilds). For a container whose user is `node`:

DEVCONTAINER.JSON (EXCERPT)

```
"mounts": [
  "source=claude-code-config,target=/home/node/.claude,type=volume"
],
"containerEnv": {
  "CLAUDE_CONFIG_DIR": "/home/node/.claude"
}
```

Want one volume per project? Put `${devcontainerId}` in the source name, as Anthropic's reference configuration does: `source=claude-code-config-${devcontainerId}`.

Lock the network

The reference container (in the `anthropics/claude-code` repository, folder `.devcontainer`) includes `init-firewall.sh`, a script that limits outbound traffic to the destinations it allows. A firewall inside a container needs extra permissions, so the reference adds the `NET_ADMIN` and `NET_RAW` capabilities through `runArgs`.

Stop the container changing under you

DEVCONTAINER.JSON (EXCERPT)

```
"containerEnv": {
  "CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC": "1",
  "DISABLE_AUTOUPDATER": "1"
}
```

The first line turns off telemetry and error reporting (and features like Remote Control that need it); the second stops Claude Code updating itself. To pin a version, install with `npm install -g @anthropic-ai/claude-code@X.Y.Z` in a Dockerfile instead of using the feature.

NOTE

On Linux and macOS, Claude Code refuses `--dangerously-skip-permissions` when run as **root** or with `sudo`, except inside a recognized sandbox. Dev containers and a normal user account both satisfy this. Check that `remoteUser` in your dev container is a non-root account.

Codespaces works too

GitHub Codespaces runs dev containers in the cloud. Store `ANTHROPIC_API_KEY`, or a `CLAUDE_CODE_OAUTH_TOKEN` made with `claude setup-token`, as a Codespaces secret so sign-in carries across codespaces.

2.3 Option 2: Claude Code's built-in Bash sandbox

You don't always need Docker. Claude Code can wrap every shell command in an operating-system sandbox: **Seatbelt** on macOS and **bubblewrap** on Linux and WSL2 (Windows' Linux layer). On Linux, install `bubblewrap` and `socat` first.

Run `/sandbox` inside Claude Code and choose **auto-allow**, or set it in a file:

```
~/.CLAUDE/SETTINGS.JSON

{
  "sandbox": {
    "enabled": true,
    "autoAllowBashIfSandboxed": true,
    "failIfUnavailable": true,
    "filesystem": {
      "allowWrite": ["/tmp/build"],
      "denyRead": ["~/aws/credentials"]
    },
    "network": {
      "allowedDomains": ["github.com", "*.npmjs.org"]
    }
  }
}
```

KEY	WHAT IT DOES
<code>enabled</code>	Sandboxes Bash commands
<code>autoAllowBashIfSandboxed</code>	Sandboxed commands run with no prompt (default <code>true</code>)
<code>failIfUnavailable</code>	Refuse to start if the sandbox can't load, instead of running unsandboxed
<code>filesystem.allowWrite</code> / <code>denyRead</code>	Extra writable paths; paths hidden from commands
<code>network.allowedDomains</code>	Websites sandboxed commands may reach
<code>excludedCommands</code>	Commands that run outside, such as <code>"docker *"</code>

💡 TIP

Without `failIfUnavailable`, a missing dependency silently means “no sandbox.” Turn it on whenever you rely on the sandbox for safety.

2.4 Option 3: the sandbox runtime

The Bash sandbox only covers shell commands; file edits and MCP servers still run on your host. Anthropic's **sandbox runtime** (the npm package `@anthropic-ai/sandbox-runtime`) wraps the *whole* Claude Code process in the same isolation.

LAUNCH CLAUDE CODE INSIDE THE RUNTIME

1 **Install the Linux packages** `bubblewrap`, `socat`, and `ripgrep` (macOS needs nothing extra).

2 **Write** `~/.srt-settings.json`.

Allow writes to your project, `~/.claude`, `~/.claude.json`, and `/tmp` (`/private/tmp` on macOS). Allow the domains `api.anthropic.com`, `claude.ai`, and `platform.claude.com`. The package README documents the exact format.

3 **Create the config paths on Linux:**

```
mkdir -p ~/.claude && { [ -f ~/.claude.json ] || echo '{}' > ~/.claude.json; }
```

4 **Launch it:**

```
npx @anthropic-ai/sandbox-runtime claude
```

The runtime denies network access by default and refuses to start if your settings file is invalid. It also blocks writes to `.git/hooks`, `.git/config`, `.mcp.json`, and shell startup files on its own.

⚠ DANGER

If `~/.srt-settings.json` doesn't exist, the runtime still starts, with everything locked down. A clean start is not proof your settings loaded. Check that the agent can reach what it needs before you walk away.

2.5 Option 4: a plain Docker container

If you'd rather not use VS Code, run any agent in a throwaway container from the terminal. This mounts only the current project:

TERMINAL

```
docker run --rm -it \
  --user 1000:1000 \
  -v "$PWD":/work -w /work \
  -e ANTHROPIC_API_KEY \
  node:22 bash
```

FLAG	MEANING
<code>--rm</code>	Delete the container when you exit
<code>-it</code>	Interactive terminal
<code>--user 1000:1000</code>	Run as a normal user, not root
<code>-v "\$PWD":/work</code>	Share only this folder with the container
<code>-e ANTHROPIC_API_KEY</code>	Pass your key in from the host environment

Inside, install your agent with npm and run it in bypass mode. Add `--network none` to cut the internet entirely (the agent then can't reach its own model API, so this suits offline tools only), or use a firewall script like Anthropic's for an allowlist.

2.6 Option 5: a virtual machine or spare computer

A **virtual machine (VM)** is a whole simulated computer with its own operating system. It's the strongest separation you can get at home. Options include a small cloud server, a local hypervisor (UTM, Parallels, VirtualBox, Hyper-V), or a spare laptop.

Docker Sandboxes (docs.docker.com/ai/sandboxes) is a free Docker product that gives each agent a **microVM**, a tiny fast-starting VM, with its own Docker daemon and workspace sync. Anthropic's docs list it as a way to run Claude Code isolated on any host.

2.7 Option 6: cloud sessions

Several vendors run the agent on *their* machines:

SERVICE	WHERE IT RUNS	NETWORK CONTROL
Claude Code cloud sessions (claude.ai/code)	Anthropic-managed VM	None / Trusted / Full / Custom
Codex Cloud	OpenAI-managed workspace	Off, Package managers, Custom, All
Copilot cloud agent	GitHub Actions runner	Firewall with allowlist
Cursor Cloud Agents	Cursor-managed VM	Allow all / Default + allowlist / Allowlist only
Devin	Cognition-managed VM	Security profiles

These are the easiest “box” of all: the agent can't touch your laptop. The trade-off is that cloud sessions often cap autonomy (Claude Code's cloud sessions don't offer bypass, for example) and they can only see what you push to GitHub.

2.8 Network allowlists: who can the agent talk to?

A **network allowlist** is a list of websites (domains) the agent's commands may reach; everything else is blocked. It's the best single defense against data being smuggled out after a prompt injection.

A good starting list for a coding agent:

- Your model API (e.g. `api.anthropic.com`)
- `github.com` and `codehost.github.com`
- `registry.npmjs.org`
- `pypi.org`, `files.pythonhosted.org`
- `crates.io`, `proxy.golang.org`
- Your own company's API domains

Add domains only when a task fails because one is blocked. Each tool's chapter shows where its allowlist lives.

2.9 Option 7: a separate user account on your computer

Create a second user account on your Mac or PC just for agents. Your personal files, browser logins, password manager, and SSH keys stay out of reach. It's not as strong as a VM, but it costs nothing and takes five minutes.

2.10 Troubleshooting the box

PROBLEM	FIX
"cannot be used with root/sudo privileges"	Set a non-root <code>remoteUser</code> , or run Docker with <code>--user</code>
Sign-in lost on every rebuild	Mount a volume at <code>~/.claude</code> and set <code>CLAUDE_CONFIG_DIR</code>
Bash sandbox silently off on Linux	Install <code>bubblewrap</code> and <code>socat</code> ; set <code>failIfUnavailable</code>
Bubblewrap fails inside Docker	Set <code>sandbox.enableWeakerNestedSandbox</code> to <code>true</code>
<code>npm install</code> fails in the box	Add the registry to the network allowlist

- ☐ I've picked a box for each project that will run in bypass mode
- ☐ The agent runs as a non-root user, and the network is limited to an allowlist
- ☐ I can throw the box away and rebuild it in minutes, sign-in included

Accounts, Tokens, and Spend Caps

A box limits what an agent can touch on *your computer*. Accounts and tokens limit what it can touch on *the internet*. Spend caps limit what it can cost you. Together they turn “the agent went wrong” from a disaster into an annoyance.

IN THIS CHAPTER

- Why agents should have their own accounts
- How tokens work, and the “narrow but complete” rule
- Where each agent reads its credentials from
- Spend caps for Claude, plus the same idea for other vendors
- Keeping secrets out of an agent’s reach

3.1 Give agents their own identity

When an agent uses *your* GitHub login or *your* cloud account, it can do anything you can do, everywhere. A separate identity fences it in.

SERVICE	AGENT IDENTITY	WHAT IT BUYS YOU
GitHub	A second “machine user” account, or a GitHub App	Added only to the repos it works on
Cloud (AWS, GCP, Azure)	A separate project or account with its own billing	Can’t touch production
Notion	A separate teamspace, or an internal connection	Only pages you share
n8n	A separate instance or project	Its workflows only
Email / chat	A dedicated bot or alias	Can’t read your personal inbox

 TIP

GitHub's terms allow one free **machine user** account per person for automation. Invite it as a collaborator on just the repositories your agents use. Its commits and pull requests are then clearly labeled as the agent's work.

3.2 Tokens: narrow but complete

A **token** (also called an access token, API key, or personal access token) is a long secret string. Whoever holds it can act as the account it belongs to, within its **scopes** (the permissions it was given) and until it **expires**.

The rule for maximum autonomy is **narrow but complete**:

- **Complete:** inside the projects the agent works on, grant everything it might need (read, write, open pull requests, run workflows). Missing permissions cause failures and, worse, prompts.
- **Narrow:** grant nothing outside those projects. No personal repos, no billing, no admin unless it truly needs it.
- **Expiring:** set an expiry date you'll remember, such as 90 days, and put a reminder in your calendar.

Chapter 12 has the exact GitHub token settings.

3.3 Where each agent looks for credentials

Most agents can sign in through a browser, but unattended runs need a token in an **environment variable** (a named value your shell passes to programs).

AGENT	ENVIRONMENT VARIABLE(S)	NOTES
Claude Code	<code>ANTHROPIC_API_KEY</code> or <code>CLAUDE_CODE_OAUTH_TOKEN</code>	Make the OAuth token with <code>claude setup-token</code> (Pro, Max, Team, Enterprise)
Codex	<code>CODEX_API_KEY</code>	Set it only for the <code>codex exec</code> call that needs it
Copilot CLI	<code>COPILOT_GITHUB_TOKEN</code> , then <code>GH_TOKEN</code> , then <code>GITHUB_TOKEN</code>	Checked in that order
Cursor CLI	<code>CURSOR_API_KEY</code>	Or <code>--api-key</code>
Gemini CLI	<code>GEMINI_API_KEY</code>	Or Google Cloud credentials
GitHub CLI	<code>GH_TOKEN</code>	Used by many agents for <code>gh</code>

⚠ WARNING

OpenAI's Codex docs warn against setting an API key as a job-level variable in CI workflows that also run repository code: build scripts, tests, and dependency hooks in the same job can read it. Give the key only to the step that runs the agent.

3.4 Subscription tokens vs API keys

KIND	BILLED TO	BEST FOR
Subscription token (e.g. <code>claude-setup-token</code>)	Your Pro/Max/Team plan's usage limits	Personal scripts and your own CI
API key (e.g. from the Claude Console)	Pay-as-you-go API billing	Shared CI, team automation, anything reused across people

Anthropic recommends an API key rather than an OAuth token for secrets shared across repositories, because an OAuth token is tied to the subscription of the person who created it.

3.5 Spend caps: the seat belt for your wallet

An agent stuck in a loop at 3 a.m. can burn through money. Every account an agent uses should have a hard limit.

SET SPEND LIMITS FOR CLAUDE

- 1 Claude Console (API keys).**
Open `platform.claude.com`, then your workspace's limits, and set a **workspace spend limit**. Give agents their own workspace and key so the limit applies only to them.
- 2 Pro or Max (usage credits).**
Open `Settings > Usage` on `claude.ai`. In **Usage credits**, turn credits on or off and set a **monthly spend limit**. In Claude Code, `/usage` shows spend against it.
- 3 Team or Enterprise.**
Owners set spend limits in the admin settings and see the spend report in org analytics.
- 4 Per run.**
In Claude Code print mode, `--max-budget-usd 10` stops a single run at about \$10 (a client-side estimate, so leave headroom).

Other vendors offer the same idea under different names: project budgets and usage limits for OpenAI API keys, budgets and alerts in Google Cloud Billing, and spending limits in GitHub billing settings for Copilot premium requests and Actions minutes. Set a hard cap wherever one exists and an email alert wherever one doesn't.

☆ **IMPORTANT**

A budget **alert** only tells you; a budget **limit** stops spending. Where a vendor offers only alerts, keep the agent's key in its own project so you can revoke it fast.

3.6 Keep secrets out of reach

Agents read files. If a secret sits in a file the agent can read, assume the agent (or a prompt injection) can read it.

- **Don't mount secrets into the box.** Pass only the one key the agent needs, as an environment variable.
- **Hide credential files.** In Claude Code's sandbox, `sandbox.filesystem.denyRead` can hide `~/.aws/credentials`; in Cursor, `.cursorignore` blocks files from the agent.
- **Redact environment variables.** Copilot CLI's `--secret-env-vars=VAR` hides a variable from shell and MCP server environments.
- **Use separate `.env` files.** Keep production secrets on machines agents never run on.
- **Turn on push protection.** GitHub blocks pushes that contain detected secrets (Chapter 12).

- ☐ Agents have their own GitHub account or App
- ☐ Every agent token is scoped to its projects only and has an expiry date
- ☐ Unattended runs read tokens from environment variables, not files in the repo
- ☐ Hard spend limits are set on every AI and cloud account the agent uses
- ☐ Production secrets are nowhere the agent can read

Claude Code (Terminal)

Claude Code has the most complete autonomy controls of any tool. It runs in your terminal, in VS Code and JetBrains, in the Claude Desktop app, and in the cloud, and every one of those reads the same settings files.

IN THIS CHAPTER

- The six permission modes and what each lets through
- Where settings live and which files can set which modes
- Auto mode: the classifier, its default blocks, and how to tune it
- Bypass mode, protected paths, and critical paths
- Permission rules, hooks, and the full maximum-autonomy settings file
- What still prompts, no matter what

4.1 The settings files

- `~/.claude/settings.json` : **user** settings. Your machine-wide defaults.
- `.claude/settings.json` (project, shared with your team) and `.claude/settings.local.json` (project, just you).
- **Managed settings**, such as `/etc/claude-code/managed-settings.json` on Linux, set by an organization. These win over everything.
- `--settings <file or JSON>` on the command line, for one session.

⚠ WARNING

`"defaultMode": "auto"` and `"defaultMode": "bypassPermissions"` are **ignored** in project files (`.claude/settings.json` and `.claude/settings.local.json`); this stops a cloned repository from switching your agent to bypass. Put them in `~/.claude/settings.json`, in managed settings, in a `--settings` file, or pass the flag.

4.2 The six permission modes

MODE	RUNS WITHOUT ASKING	BEST FOR
<code>default</code> (Manual)	Reads only	Sensitive work
<code>acceptEdits</code>	Reads, edits, and <code>mkdir</code> , <code>touch</code> , <code>rm</code> , <code>mv</code> , <code>cp</code> , <code>sed</code> in your folders	Reviewing with <code>git diff</code> afterwards
<code>plan</code>	Reads (plus classifier-approved commands when auto is available)	Exploring before changing
<code>auto</code>	Everything, with background safety checks	Long tasks, hands-off work
<code>dontAsk</code>	Pre-approved tools only; anything else is denied	Locked-down CI
<code>bypassPermissions</code>	Everything	Containers and VMs only

Press `Shift` + `Tab` during a session to cycle modes; the status bar shows the active one (for example `auto mode on` or `bypass permissions on`). Start in a mode with `claude --permission-mode <mode>`. `manual` is accepted as an alias for `default`.

NOTE

Deny rules block in every mode, including bypass. **Allow rules have no effect in bypass**, which already allows everything. That makes deny rules your one hard guardrail inside a fully open session.

4.3 Level 3: auto mode by default

With Claude Code v2.1.283 or later, interactive terminal and VS Code sessions start in **auto mode** on every plan and provider, where a classifier model reviews risky actions instead of asking you. To make it explicit and permanent:

~/.CLAUDE/SETTINGS.JSON

```
{
  "permissions": {
    "defaultMode": "auto"
  }
}
```

Auto mode needs a supported model. On the Anthropic API that means Claude Opus 4.6 or later, Sonnet 4.6 or later, Haiku 5.5, or a Fable model. Team and Enterprise admins can switch it off with `permissions.disableAutoMode`.

What the classifier blocks by default

The classifier trusts your working directory and the git remotes configured when the session started. Out of the box it blocks, among others:

BLOCKED	EXAMPLE
Downloading and running code	<code>`curl ...`</code>
Sending sensitive data out	Posting secrets to an unknown site
Production deploys and migrations	<code>kubectl apply</code> on prod
Discarding uncommitted work	<code>git reset --hard</code> , <code>git clean -fd</code>
Force push	<code>git push --force</code>
Destroying infrastructure	<code>terraform destroy</code>
Merging an unapproved PR	<code>gh pr merge</code> with no human review
Granting permissions	IAM or repo-admin changes
Mass deletion on cloud storage	Emptying a bucket

Reduce false alarms

- **Teach it your infrastructure.** Run `/auto-mode-setup` to draft `autoMode.environment` entries (trusted repos, buckets, domains, services) from your project and recent sessions, and save them.
- **Approve standing patterns.** Tell it in plain English *with specifics*: “you may force-push to branches starting with `agent/`” works; “you can force-push” clears nothing.
- **Write rules in settings.** `autoMode` takes plain-English arrays. Include `"$defaults"` to keep the built-in rules:

~/CLAUDE/SETTINGS.JSON

```
{
  "autoMode": {
    "environment": [
      "$defaults",
      "github.com/my-org/* repositories are ours",
      "staging.example.com is a test environment"
    ],
    "allow": ["$defaults", "Force-push to branches named agent/*"],
    "soft_deny": ["$defaults", "Never run terraform apply"]
  }
}
```

- **Check the rules:** `claude auto-mode defaults` prints the built-in lists; `claude auto-mode config` shows your effective config.
- **See what was blocked:** `/permissions`, then the **Recently denied** tab. Press `r` to retry an action with a manual approval.

NOTE

If the classifier blocks 3 actions in a row or 20 in total, auto mode pauses and asks you. These thresholds can't be changed. In `claude -p` runs with no prompt available, the blocked action is skipped and Claude keeps working.

WARNING

Things you say (“don’t push yet”) are treated as boundaries, but they live only in the conversation. If context compaction removes that message, the boundary can be lost. For a hard guarantee, write a deny rule.

4.4 Level 4: bypass permissions

One session:**TERMINAL**

```
claude --dangerously-skip-permissions
```

(Equivalent: `claude --permission-mode bypassPermissions`.) The first time, you accept a warning; Claude Code then records `"skipDangerousModePermissionPrompt": true` in `~/.claude/settings.json` so it doesn't ask again.

Every session, by default:**~/.CLAUDE/SETTINGS.JSON**

```
{
  "permissions": {
    "defaultMode": "bypassPermissions"
  },
  "skipDangerousModePermissionPrompt": true
}
```

Start in another mode but keep bypass one keypress away: `claude --allow-dangerously-skip-permissions` adds bypass to the `Shift + Tab` cycle without starting in it. You can't switch *into* bypass from a session started without it enabled.

A shell alias for convenience:**~/.ZSHRC OR ~/.BASHRC**

```
alias yolo='claude --dangerously-skip-permissions'
```

⚠ DANGER

Bypass mode offers no protection against prompt injection. Anthropic's own warning: use it only in containers, VMs, or dev containers, ideally without open internet access.

4.5 Protected paths

Some files are guarded because changing them could break your repository or quietly change Claude's own settings. Writes to them are never auto-approved except in bypass mode.

- `.git`, `.config/git`
- `.vscode`, `.idea`
- `.husky`, `.cargo`, `.yarn`, `.mvn`
- `.devcontainer`, `.devcontainer.json`
- `.claude` (mostly)
- `.gitconfig`, `.gitmodules`
- Shell files: `.bashrc`, `.zshrc`, `.profile` ...
- `.npmrc`, `.yarnrc`, `bunfig.toml`
- `.pre-commit-config.yaml`
- `.mcp.json`, `.claude.json`

MODE	PROTECTED-PATH WRITES
<code>default</code> , <code>acceptEdits</code>	Prompted
<code>auto</code>	Sent to the classifier
<code>dontAsk</code>	Denied
<code>bypassPermissions</code>	Allowed

Allow rules can't pre-approve these; the check runs first. When prompted for `.claude/`, the option **Yes, and allow Claude to edit files in this project's .claude folder for this session** saves you repeat prompts.

4.6 Critical paths: the deletion circuit breaker

Critical paths are the filesystem root, your home folder, top-level system folders, and your working directory and its parents. No allow rule and no hook can approve `rm` or `rmdir` aimed at them.

MODE	CRITICAL-PATH <code>RM</code>
<code>default</code> , <code>acceptEdits</code> , <code>plan</code>	Asks you
<code>auto</code>	Asks in the terminal with a 2-minute countdown; denied elsewhere
<code>dontAsk</code>	Denied
<code>bypassPermissions</code>	Asks, with a 2-minute countdown in the terminal

If the countdown runs out, the command is denied and Claude is told what to do instead, so an unattended session keeps working. After three unanswered countdowns, further critical-path removals are denied immediately. Setting `CLAUDE_CODE_DISABLE_DANGEROUS_RM_TIMEOUT=1` changes this: auto mode sends them to the classifier instead, and bypass shows the prompt with no time limit.

TIP

Avoid the prompt entirely by deleting literal subfolders (`rm -rf ./build`) rather than the working directory itself, and ask agents in your `CLAUDE.md` to do the same.

4.7 Permission rules

Rules live under `permissions` in any settings file, as `allow`, `ask`, and `deny` arrays. Each rule names a tool, optionally with a pattern in brackets:

RULE	MATCHES
<code>Bash</code>	Every shell command
<code>Bash(npm test)</code>	Exactly <code>npm test</code>
<code>Bash(git push *)</code>	<code>git push</code> with any arguments
<code>Edit(src/**)</code>	Edits anywhere under <code>src</code>
<code>Read(/etc/**)</code>	Reads under the absolute path <code>/etc</code>
<code>WebFetch</code>	Every web fetch
<code>mcp__github__*</code>	Every tool from the <code>github</code> MCP server

Precedence is deny, then ask, then allow. An MCP allow glob must name its server: `mcp_*` on its own is ignored in allow rules (it works in deny rules).

4.8 Hooks: answer prompts with a script

A **hook** is a command Claude Code runs automatically at a set moment. Two events matter for autonomy:

- **PreToolUse** runs before every tool call and can return `"permissionDecision": "allow"`, `"deny"`, or `"ask"`.
- **PermissionRequest** runs only when Claude Code is about to ask you, and can answer on your behalf.

~/CLAUDE/SETTINGS.JSON (EXCERPT)

```
{
  "hooks": {
    "PermissionRequest": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "~/claude/approve.sh" }
        ]
      }
    ]
  }
}
```

The script reads the request as JSON on standard input and prints a decision:

JSON

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": { "behavior": "allow" }
  }
}
```

Deny and ask rules are still checked after a hook says allow, and a hook can never approve a critical-path deletion. In non-interactive sessions, a **PermissionRequest** that no hook answers is denied.

4.9 A complete maximum-autonomy settings file

~/CLAUDE/SETTINGS.JSON

```
{
  "permissions": {
    "defaultMode": "bypassPermissions",
    "allow": [
      "Bash", "Edit", "Write", "WebFetch", "WebSearch",
      "mcp__github__*", "mcp__notion__*"
    ],
    "deny": ["Bash(git push --force *)", "Read(~/.ssh/**)"],
    "additionalDirectories": ["~/projects"]
  },
  "skipDangerousModePermissionPrompt": true,
  "enableAllProjectMcpServers": true,
  "env": {
    "CLAUDE_CODE_DISABLE_DANGEROUS_RM_TIMEOUT": "1"
  }
}
```

What each line buys you:

SETTING	EFFECT
<code>defaultMode: bypassPermissions</code>	Every session starts with no prompts
<code>allow</code> rules	Useful when you drop into auto or manual mode: these tools stop asking. (No effect <i>in</i> bypass; broad rules like <code>Bash</code> are suspended while in auto mode.)
<code>deny</code> rules	Your hard guardrails, enforced even in bypass
<code>mcp__server__*</code>	Approves every tool from that MCP server
<code>additionalDirectories</code>	Lets Claude read and edit outside the folder you started in
<code>enableAllProjectMcpServers</code>	Auto-approves servers listed in projects' <code>.mcp.json</code> files
<code>CLAUDE_CODE_DISABLE_DANGEROUS_RM_TIMEOUT</code>	Removes the 2-minute countdown on critical deletions (the prompt itself stays)

TIP

Workspace trust: the first time you open a folder, Claude Code asks whether you trust it. Answer once per repository and it remembers. For automation, `-p` runs don't show the dialog, though project allow rules aren't applied there until a folder is trusted.

4.10 Reduce prompts without bypass

If you'd rather stay in Manual or Accept-edits mode for some projects:

- `/fewer-permission-prompts` scans your history and proposes an allowlist.
- `/sandbox`, then **auto-allow**, runs shell commands inside an OS-level sandbox *without prompting* (macOS, Linux, WSL2).
- Answer prompts with **Yes, and don't ask again** to save rules automatically.
- In Manual or Accept-edits mode, a Bash prompt can offer **Yes, and switch to auto mode**.

PROMPT

Look through my last week of Claude Code sessions in this project. List the commands I approved by hand most often, then add safe ones to the allow list in `.claude/settings.local.json`. Don't add anything that deletes, pushes, or deploys. Show me the diff before saving.

4.11 What still prompts in Claude Code, no matter what

STILL ASKS, EVEN IN BYPASS	WHY / WORKAROUND
Tools matching an explicit ask rule	Remove the ask rule
<code>rm</code> / <code>rmdir</code> targeting a critical path	A deliberate circuit breaker; use literal subfolder paths
<code>AskUserQuestion</code> and MCP tools marked <code>requiresUserInteraction</code>	By design; they need an answer from a person
Connector tools your organization set to "ask"	Admin policy
Reads outside working directories if <code>blockReadsOutsideWorkingDirectories</code> is on	Leave it off, or add directories
Some cross-session messaging approvals	Only matters with several sessions messaging each other
<code>Remove-Item</code> on system paths or bare wildcards (PowerShell)	Denied outright in every mode
Sessions started with <code>--restricted</code>	Bypass is refused
Cloud sessions (claude.ai/code)	Bypass isn't offered; Auto is the maximum

- ☐ `~/.claude/settings.json` sets `defaultMode` to `auto` or `bypassPermissions`
- ☐ Deny rules cover the few things you never want, even in bypass
- ☐ `/auto-mode-setup` has taught the classifier; bypass runs only in a box

CHAPTER 5

Claude Code Without You: Headless, CI, SDK, and Cloud

Everything in the last chapter assumed you were sitting at a terminal. This chapter is about the times you aren't: scripts, scheduled jobs, GitHub workflows, your own programs, and Anthropic's cloud.

IN THIS CHAPTER

- Print mode (`claude -p`) and the flags that keep it safe
- Background agents that outlive your terminal
- The Claude Code GitHub Action, including scheduled runs
- The Agent SDK's permission options
- Cloud sessions, routines, and auto-fix for pull requests

5.1 Print mode: one command, no chat window

Print mode runs a single task and exits. It's how you call Claude Code from scripts, cron jobs, and CI.

TERMINAL

```
claude -p "Fix the failing tests and commit the result" \  
  --dangerously-skip-permissions \  
  --permission-prompts none \  
  --max-turns 60 \  
  --max-budget-usd 10 \  
  --output-format json
```

FLAG	WHAT IT DOES
<code>-p</code> , <code>--print</code>	Run once, print the answer, exit
<code>--permission-prompts none</code>	Deny anything that would still need a human, instead of waiting forever
<code>--max-turns</code>	Stop after this many agent turns (exits with an error)
<code>--max-budget-usd</code>	Stop once estimated spend reaches this amount
<code>--output-format</code>	<code>text</code> , <code>json</code> , or <code>stream-json</code>
<code>--json-schema</code>	Validate the final answer against a JSON Schema
<code>--bare</code>	Skip hooks, skills, plugins, MCP servers, and CLAUDE.md for a fast, predictable start
<code>--no-session-persistence</code>	Don't save the session to disk
<code>--fallback-model</code>	Try another model if the first is overloaded

Without bypass, use an exact allowlist. Anything not listed is denied, and the run never waits:

TERMINAL

```
claude -p "run the test suite and summarize failures" \
  --permission-mode dontAsk \
  --allowedTools "Bash(npm test)" "Read"
```

TIP

Chain runs with `--resume <session-id>` or `--continue` (most recent in this folder). Name sessions with `-n nightly-fix` so they're easy to find.

Let a program answer prompts

`--permission-prompt-tool <mcp_tool>` hands every prompt to an MCP tool you wrote, which replies allow or deny. It's the scripted version of a human approver, useful when you want custom logic such as “allow anything inside `/work`”.

5.2 Background agents

TERMINAL

```
claude --bg "Upgrade all dependencies and fix what breaks"
```

`--bg` starts a **background agent** and returns immediately, printing its session ID and management commands. It keeps running after you close the terminal. Run `claude agents` to see and manage them. `claude --bg --exec 'pytest -x'` runs a plain shell command as a background job instead.

☆ IMPORTANT

Background sessions honor bypass only when your acceptance is recorded in `~/.claude/settings.json` (or managed settings). With no acceptance recorded, `claude --bg --permission-mode bypassPermissions` is refused until you accept the warning once in an interactive session.

5.3 Git worktrees: parallel agents without collisions

`claude -w feature-auth` starts Claude in its own **git worktree** (a second working copy of the repo on its own branch) at `.claude/worktrees/feature-auth`. Add `--tmux` to open it in a tmux session. Chapter 20 covers running several agents this way.

5.4 The Claude Code GitHub Action

The **GitHub Action** runs Claude Code inside GitHub's own servers whenever something happens in your repository: a comment mentioning `@claude`, a new pull request, or a schedule.

SET IT UP IN ONE COMMAND

- 1 **Open Claude Code** in the repository.
- 2 **Run** `/install-github-app`.
It installs the Claude GitHub App, adds an authentication secret, and opens a pull request with the workflow file.
- 3 **Merge the pull request.**
Now `@claude` works in any issue or PR comment.

Prefer doing it by hand? Install the app from github.com/apps/claude, add a repository secret `ANTHROPIC_API_KEY` (or `CLAUDE_CODE_OAUTH_TOKEN` from `claude setup-token`), and add this workflow:

```
.GITHUB/WORKFLOWS/CLAUDE.YML
```

```
name: Claude Code
on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
jobs:
  claude:
    if: contains(github.event.comment.body, '@claude')
    runs-on: ubuntu-latest
    permissions:
      contents: write
      pull-requests: write
      issues: write
      id-token: write
      actions: read
    steps:
      - uses: actions/checkout@v6
        with:
          fetch-depth: 1
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
```

`id-token: write` is required for the action's GitHub App authentication; `actions: read` lets Claude read CI results.

Run on a schedule

With a `prompt` input, the action runs on any event, including a **cron schedule** (a timetable written as five fields: minute, hour, day, month, weekday). Give it tools explicitly with `claude_args`, because a plain prompt has no shell or GitHub access until you grant it:

.GITHUB/WORKFLOWS/NIGHTLY.YML

```
name: Nightly Report
on:
  schedule:
    - cron: "0 9 * * *"
jobs:
  report:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      issues: read
      id-token: write
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
          prompt: "Summarize yesterday's commits and open issues"
          claude_args: |
            --allowedTools "mcp__github__list_commits,mcp__github__list_issues"
```

For maximum autonomy inside the runner, which is already a throwaway machine, pass `--dangerously-skip-permissions` or broad `--allowedTools` in `claude_args`.

Still limited in the Action: the person triggering it must have write access (set `allowed_non_write_users` to widen this), and bot accounts are rejected unless listed in `allowed_bots`, which stops bots triggering Claude in a loop. GitHub runs schedules only from the default branch, and in public repositories disables them after 60 days without activity.

5.5 The Agent SDK

The **Agent SDK** lets you build your own programs on top of Claude Code's engine, in Python or TypeScript. Permissions are options on the query:

AGENT.PY

```
from claude_agent_sdk import query, ClaudeAgentOptions

options = ClaudeAgentOptions(
    permission_mode="bypassPermissions",
    disallowed_tools=["Bash(git push --force *)"],
)
```

OPTION	EFFECT
<code>permission_mode</code> / <code>permissionMode</code>	Any of the six modes
<code>allowed_tools</code> / <code>allowedTools</code>	Pre-approves listed tools
<code>disallowed_tools</code> / <code>disallowedTools</code>	Deny rules; enforced even in bypass
<code>canUseTool</code> callback	Your code decides each prompt
<code>set_permission_mode()</code>	Change mode mid-session

 WARNING

`allowed_tools` does **not** restrict bypass mode: unlisted tools fall through to the mode, which approves them. To lock an agent down, pair `allowedTools` with `permissionMode: "dontAsk"`. Subagents inherit bypass, too.

5.6 Cloud sessions at claude.ai/code

Cloud sessions run in an Anthropic-managed VM, so your laptop can be closed. Start one from the web, the mobile app, or the terminal with `claude --cloud "Fix the login bug"`. Pull one back to your terminal with `claude --teleport`.

SETTING	MAXIMUM-AUTONOMY CHOICE
Permission mode	Auto (bypass isn't available; file edits are pre-approved in every mode)
Network access	Full for any domain, or Custom with your own allowlist
Setup script	Install everything the agent needs up front

Network levels are **None**, **Trusted** (package registries, GitHub, cloud SDKs), **Full**, and **Custom**. GitHub and enabled connectors stay reachable whatever you choose.

Auto-fix pull requests

Claude can watch a pull request and push fixes when CI fails or a reviewer comments. In a cloud session, open the CI status bar and select **Auto-fix**; from your terminal, run `/autofix-pr` on the PR's branch. It needs the Claude GitHub App installed.

5.7 Routines and scheduled tasks

OPTION	RUNS ON	MACHINE ON?	PROMPTS
Routines (<code>/schedule</code> , <code>claude.ai/code/routines</code>)	Anthropic's cloud	No	None (autonomous)
Desktop scheduled tasks	Your computer	Yes	Configurable per task
<code>/loop</code> in a session	Your computer	Yes, session open	Inherits the session

Routines (research preview, Pro, Max, Team, and Enterprise) package a prompt, repositories, and connectors, and run on a schedule (minimum hourly), from an API call with a bearer token, or on GitHub events such as a new pull request. `/loop 5m /babysit-prs` reruns a prompt every five minutes inside an open session.

- ☐ Unattended runs use `-p` with `--max-turns` and `--max-budget-usd`
- ☐ CI either runs in bypass inside the runner or uses `dontAsk` with an exact allowlist
- ☐ The GitHub Action's secret is an API key if several people share it
- ☐ Background agents work because bypass acceptance is in `~/.claude/settings.json`
- ☐ Cloud sessions have the network level the task needs

CHAPTER 6

Claude Desktop, Cowork, and claude.ai

The Claude apps (Desktop, the web at claude.ai, and mobile) put agents in friendlier windows. Under the hood, the desktop **Code** tab is the same Claude Code you met in the last two chapters, while **Cowork** and **connectors** have their own approval switches.

IN THIS CHAPTER

- Bypass and auto mode in the desktop Code tab
- Claude Code in VS Code and JetBrains
- Cowork’s “Automatically approve” mode
- Setting every connector tool to “Always allow”
- Claude in Chrome’s per-site approvals

6.1 The Code tab in Claude Desktop

The Code tab uses the same engine and settings files as the CLI.

ENABLE BYPASS IN THE DESKTOP APP

- 1 **Open Settings** in Claude Desktop and go to **Claude Code**.
- 2 **Turn on “Allow bypass permissions mode.”** On Pro and Max plans this is a personal toggle; on Team and Enterprise, your organization’s policy controls it.
- 3 **Choose the mode.**
In a session, open the mode selector next to the send button (or press **Cmd** + **Shift** + **M**) and pick **Bypass permissions**, or **Auto** for classifier-reviewed autonomy.

4 It sticks per folder.

The desktop app remembers the mode you pick for each project folder (except Plan). A `defaultMode` in `~/ .claude/settings.json` sets the starting point for new folders.

Still asks in the desktop app: the Claude Code list from Chapter 3, plus **safety classifiers on actions Claude takes on external websites** in the built-in browser (any mode), and destructive desktop actions such as deleting a session. On websites, Claude won't purchase things, create accounts, or solve CAPTCHAs without you. Answer a site's first action card with **Always allow** to stop repeat prompts for that site.

6.2 Cowork

Cowork (a tab in Claude Desktop) does longer agentic work on your files, in your browser, and through connectors.

- In the Cowork mode selector, choose **Automatically approve**. On Team and Enterprise, an admin must leave **Allow “Automatically approve” mode** on (it's on by default) in `Organization settings` > `Cowork` > `Permissions`.
- For connector **write** tools to stop asking every task, your admin must also turn on **Allow “Always allow” for connector tools** (off by default on Team and Enterprise). Then choose **Allow for all tasks** in the approval dialog.

6.3 Connectors in claude.ai and the Claude apps

Each connector's tools can be set to **Always allow**, **Needs approval**, or **Blocked**.

MAKE CONNECTORS HANDS-OFF

- 1 Open `Settings` > `Connectors` (or `Customize`, then `Connectors`, in the desktop app).
- 2 Click a connector, then **Tool permissions**.
- 3 Set each category or tool to **Always allow**.
Do this for read and write tools you want the agent to use freely.
- 4 Repeat for each connector.

NOTE

Custom connectors' tools usually lack “read-only” labels, so they default to asking. Set them to **Always allow** explicitly. Organization owners can cap what members may allow, and that cap can't be overridden by individuals.

6.4 Claude in Chrome

The Chrome extension asks per site the first time Claude acts there. Choose **Always allow** for sites you want it to operate on freely. It will still stop for purchases, account creation, and CAPTCHAs, and its safety classifier can still raise a prompt on risky page actions.

6.5 Claude Code in VS Code

The VS Code extension starts conversations in auto mode by default. To pin a different starting mode, set `claudeCode.initialPermissionMode` in your VS Code user settings to `default`, `acceptEdits`, `plan`, or `bypassPermissions`. It doesn't accept `auto`; to use Auto, leave the setting unset and pick **Auto** from the mode indicator once.

To make **Bypass permissions** appear at all, turn on the extension's **Allow dangerously skip permissions** setting first.

VS CODE SETTINGS.JSON

```
{
  "claudeCode.initialPermissionMode": "bypassPermissions"
}
```

NOTE

Plan mode keeps its blocks in the VS Code chat panel even when bypass is available, and in auto mode a critical-path deletion is denied there immediately, because the panel can't show the terminal's countdown prompt.

6.6 Desktop scheduled tasks

The desktop app can run a Claude Code task on a timetable (minimum interval one minute) while your computer is on, with access to local files. Each task has its own permission setting, so choose a hands-off mode for tasks you won't be watching.

6.7 Troubleshooting the Claude apps

PROBLEM	FIX
No Bypass permissions in the mode list	Turn on Allow bypass permissions mode in Settings > Claude Code; on Team/Enterprise ask an admin
Mode resets each time	The app remembers per folder; set <code>defaultMode</code> in <code>~/.claude/settings.json</code> for new folders
Cowork still asks for connector writes	Admin must enable Allow "Always allow" for connector tools
A custom connector asks for every tool	Set each tool to Always allow explicitly
Chrome asks again on every site	Choose Always allow on the site's first action card

- ☐ Desktop: **Allow bypass permissions mode** on, mode chosen per folder
- ☐ VS Code: bypass setting on, or Auto picked once
- ☐ Cowork: **Automatically approve** selected
- ☐ Every connector tool you use set to **Always allow**
- ☐ Chrome: sites you trust set to **Always allow**

OpenAI Codex

Codex is OpenAI's coding agent. It comes as a command-line tool (the Codex CLI), an IDE extension for VS Code and similar editors, part of the ChatGPT desktop app, and **Codex Cloud**, which runs tasks on OpenAI's machines. They share one settings file: `~/.codex/config.toml`.

IN THIS CHAPTER

- Codex's two controls: the sandbox and the approval policy
- The four permission modes in the app, IDE, and CLI
- Every flag and `config.toml` key for hands-off work
- `codex exec` for scripts and CI, and the Codex GitHub Action
- Codex Cloud environments and internet access
- What still prompts or pauses

7.1 Two controls that work together

Codex separates *what it can reach* from *when it asks*:

CONTROL	QUESTION IT ANSWERS	VALUES
Sandbox mode (<code>sandbox_mode</code> , <code>--sandbox</code>)	What can commands touch?	<code>read-only</code> , <code>workspace-write</code> , <code>danger-full-access</code>
Approval policy (<code>approval_policy</code> , <code>--ask-for-approval</code>)	When does Codex pause?	<code>on-request</code> , <code>never</code> , or a <code>granular table</code>
Reviewer (<code>approvals_reviewer</code>)	Who answers a pause?	<code>user</code> or <code>auto_review</code>

Changing who reviews a request never widens the sandbox. That's the key idea: you can make Codex prompt-free *inside* a tight sandbox, or remove the sandbox entirely.

7.2 The permission modes

In the ChatGPT desktop app and the IDE extension, open the permissions control below the message box. In the CLI, type `/permissions`.

MODE	SANDBOX	APPROVALS	ASKS YOU
Ask for approval	<code>workspace-write</code>	<code>on-request</code> , you review	To edit outside the workspace or use the internet
Approve for me	<code>workspace-write</code>	<code>on-request</code> , <code>auto_review</code>	Only for actions flagged as potentially unsafe
Full access	<code>danger-full-access</code>	<code>never</code>	Never
Custom	From <code>config.toml</code>	From <code>config.toml</code>	Depends

Approve for me is Codex's Level 3: a reviewer agent checks each action that would otherwise need approval, looking for data exfiltration, credential probing, persistent security weakening, and destructive actions. Low- and medium-risk actions go ahead; critical-risk actions are denied. It uses extra model calls, so it adds to your usage.

7.3 Level 3 from the command line

TERMINAL

```
codex --approve-for-me
```

`--approve-for-me` (alias `--not-so-yolo`) means `workspace-write` plus `on-request` plus the automatic reviewer. The same in `config.toml`:

~/.CODEX/CONFIG.TOML

```
sandbox_mode = "workspace-write"
approval_policy = "on-request"
approvals_reviewer = "auto_review"

[sandbox_workspace_write]
network_access = true
```

By default `workspace-write` has **no network** for commands. `network_access = true` lets `npm install` and friends work without asking.

7.4 Level 4: no prompts, no sandbox

One session:

TERMINAL

```
codex --yolo
```

`--yolo` is the alias of `--dangerously-bypass-approvals-and-sandbox`: every command runs with no approvals and no sandbox. **Make it the default:**

~/CODEX/CONFIG.TOML

```
approval_policy = "never"
sandbox_mode = "danger-full-access"
```

In the Codex app and IDE extension, choose **Full access** from the permissions menu.

A middle ground: `approval_policy = "never"` with `sandbox_mode = "workspace-write"` never asks, and simply refuses anything the sandbox doesn't allow. That's ideal when you want zero prompts but still want Codex walled into the project.

⚠ WARNING

`--full-auto` is no longer supported, and neither is `approval_policy = "untrusted"`. Old scripts and blog posts still use them. Replace `--full-auto` with `--sandbox workspace-write` (plus `-a never` if you want no prompts) or with `--approve-for-me`.

7.5 Every flag that matters

FLAG	MEANING
<code>-s</code> , <code>--sandbox <mode></code>	<code>read-only</code> , <code>workspace-write</code> , or <code>danger-full-access</code>
<code>-a</code> , <code>--ask-for-approval <policy></code>	<code>on-request</code> or <code>never</code>
<code>--approve-for-me</code>	Auto-review preset
<code>--yolo</code>	No approvals, no sandbox
<code>--add-dir <path></code>	Another writable folder (repeatable)
<code>-C</code> , <code>--cd <path></code>	Working folder
<code>-c key=value</code>	Override any config key for this run
<code>-p</code> , <code>--profile <name></code>	Layer <code>\$CODEX_HOME/<name>.config.toml</code> on top
<code>--search</code>	Live web search instead of the cached index

7.6 Profiles: one file per autonomy level

A **profile** is a small extra config file layered on top of your main one. Keep one for each level you use:

```
~/.CODEX/YOLO.CONFIG.TOML
```

```
approval_policy = "never"
sandbox_mode = "danger-full-access"
```

```
~/.CODEX/WALLED.CONFIG.TOML
```

```
approval_policy = "never"
sandbox_mode = "workspace-write"

[sandbox_workspace_write]
network_access = true
writable_roots = ["/tmp/build"]
```

Then run `codex -p yolo` inside a container, or `codex -p walled` on your laptop.

7.7 Trust your projects

Codex starts in `read-only` until you trust a folder, and skips a project's own `.codex/` settings in untrusted folders. Trust a project once:

```
~/.CODEX/CONFIG.TOML
```

```
[projects."/home/me/code/my-app"]
trust_level = "trusted"
```

7.8 MCP servers without prompts

Each MCP server (and each of its tools) can have its own approval mode: `auto`, `prompt`, `writes` (prompt only for tools not marked read-only), or `approve`.

```
~/.CODEX/CONFIG.TOML
```

```
[mcp_servers.github]
url = "https://api.githubcopilot.com/mcp/"
bearer_token_env_var = "GITHUB_PAT"
default_tools_approval_mode = "approve"
```

NOTE

With an interactive approval policy, Codex still asks before app or MCP tools that advertise a **destructive** annotation. `approval_policy = "never"` removes those prompts too.

7.9 Protected paths

Even in `workspace-write`, these stay read-only inside each writable root: `.git` (and the real git folder it points to), `.agents`, `.codex`, and `.aws`. Only `danger-full-access` lifts this.

7.10 Scripts and CI: `codex exec`

`codex exec` (short form `codex e`) runs one task and exits. It streams progress to the error stream and prints only the final answer, so you can pipe it.

TERMINAL

```
codex exec --sandbox workspace-write \
-a never \
--json \
-o result.md \
"Fix the failing tests"
```

FLAG	MEANING
<code>--json</code>	One JSON event per line (<code>thread.started</code> , <code>turn.completed</code> , ...)
<code>-o</code> , <code>--output-last-message <file></code>	Save the final answer
<code>--output-schema <file></code>	Force the final answer to match a JSON Schema
<code>--ephemeral</code>	Don't save session files
<code>--skip-git-repo-check</code>	Run outside a git repository
<code>--ignore-user-config</code>	Don't load <code>~/.codex/config.toml</code>
<code>codex exec resume --last "..."</code>	Continue the previous run

By default `codex exec` runs **read-only**. Give it `--sandbox workspace-write` to edit, or `--sandbox danger-full-access` (or `--yolo`) inside an isolated runner. In CI, pass the key only to that command: `CODEX_API_KEY=... codex exec ...`.

7.11 The Codex GitHub Action

.GITHUB/WORKFLOWS/CODEX.YML (EXCERPT)

```
- uses: openai/codex-action@v1
  with:
    openai-api-key: ${ secrets.OPENAI_API_KEY }
    prompt-file: .github/codex/prompts/review.md
    sandbox: workspace-write
```

The action starts a proxy so your key isn't exposed to repository code. Its `safety-strategy` defaults to `drop-sudo`, which removes `sudo` before Codex runs; `unprivileged-user` runs Codex as a named account; `unsafe` is required on Windows runners. Keep `drop-sudo` on shared runners.

7.12 Codex Cloud

Codex Cloud runs tasks in isolated cloud workspaces, so work continues while your computer sleeps. Start on the web or in the desktop app with **Work in > Cloud** and a published **environment** (the saved setup: repositories, tools, and access).

GIVE A CLOUD ENVIRONMENT INTERNET ACCESS

- 1 Open the environment (or **Settings > Codex Cloud > Environments > Create environment**).
- 2 Turn on "Allow Codex to access internet."
- 3 Choose Allow domains:
Package managers (npm, PyPI, crates, Go, Maven, apt, GitHub, and more), **Custom domains only**, or **All (unrestricted)**.
- 4 Add hosts under **Additional allowed domains** if needed, then save, test, and **Publish**.

Store keys as **network secrets**: saving one adds its domains to the allowlist automatically. Your organization's **Agent Security** settings apply on top and can't be overridden from an environment.

7.13 Notifications

Point `notify` at a program that runs when a turn completes, for chat webhooks or desktop alerts:

```
~/.CODEX/CONFIG.TOML
```

```
notify = ["python3", "/home/me/bin/codex-notify.py"]
```

```
[tui]
```

```
notifications = true
```

7.14 What still asks or pauses in Codex

STILL HAPPENS	WHY
Prompts to leave the sandbox, in <code>on-request</code>	Use <code>-a never</code> or Full access
Destructive MCP/app tools ask, in interactive policies	Use <code>approval_policy = "never"</code>
Safety monitoring can pause a task	Runs separately from approvals; in the CLI a paused task ends
Writes to <code>.git</code> , <code>.codex</code> , <code>.agents</code> , <code>.aws</code>	Protected unless <code>danger-full-access</code>
Untrusted folders start read-only	Trust the project
Admin managed requirements	Can disable modes or restrict reviewers
Cloud tasks can't reach a domain	Environment allowlist and Agent Security rules

- ☐ `~/.codex/config.toml` sets the default level you want
- ☐ Profiles exist for “walled” and “yolo” runs
- ☐ Scripts no longer use `--full-auto`
- ☐ Projects you use are marked `trusted`
- ☐ CI passes the API key only to the `codex exec` step
- ☐ Cloud environments have the domains they need

Gemini CLI

Gemini CLI is Google's open-source terminal agent. It's free to start with a Google account and keeps its settings in `~/.gemini/settings.json` (user) and `.gemini/settings.json` (project).

IN THIS CHAPTER

- The four approval modes and how to start in each
- `settings.json` keys for autonomy and safety
- The policy engine: allow and deny rules in TOML files
- Folder trust, the sandbox, and headless mode
- The Gemini GitHub Action

8.1 Approval modes

MODE	WHAT RUNS WITHOUT ASKING
<code>default</code>	Reads; edits and commands ask
<code>auto_edit</code>	File edits too
<code>plan</code>	Read-only research and planning
<code>yolo</code>	Everything

TERMINAL

```
gemini --approval-mode=yolo
```

NOTE

The short `--yolo` (`-y`) flag still works but is **deprecated**; use `--approval-mode=yolo`. YOLO can only be switched on from the command line, never from `settings.json`, so put it in an alias:

```
~/.ZSHRC OR ~/.BASHRC
```

```
alias gem='gemini --approval-mode=yolo --sandbox'
```

For a hands-off default that stops short of YOLO, set the starting mode in settings:

```
~/.GEMINI/SETTINGS.JSON
```

```
{
  "general": {
    "defaultApprovalMode": "auto_edit",
    "checkpointing": { "enabled": true }
  },
  "security": {
    "enablePermanentToolApproval": true
  }
}
```

`defaultApprovalMode` accepts `default`, `auto_edit`, or `plan`. **Checkpointing** saves a snapshot before changes so you can roll back. `enablePermanentToolApproval` adds **Allow for all future sessions** to approval dialogs, so one click stops a prompt forever.

8.2 Pre-approve specific tools

The quick way is `tools.allowed`, a list of tools that skip the confirmation dialog:

```
~/.GEMINI/SETTINGS.JSON (EXCERPT)
```

```
{
  "tools": {
    "allowed": [
      "run_shell_command(git)",
      "run_shell_command(npm test)"
    ]
  }
}
```

The more powerful way is the **policy engine**: TOML rule files in `~/.gemini/policies/` (any name ending in `.toml`). Each rule names a tool, optional conditions, a decision (`allow`, `deny`, or `ask_user`), and a priority from 0 to 999; the highest-priority match wins.

```
~/GEMINI/POLICIES/AUTONOMY.TOML
```

```
[[rule]]
toolName = "run_shell_command"
commandPrefix = "git"
decision = "allow"
priority = 100
```

```
[[rule]]
toolName = "run_shell_command"
commandPrefix = "rm -rf"
decision = "deny"
priority = 200
```

A deny rule keeps a guardrail even in YOLO mode. Rules can be limited to modes (`yolo`, `autoEdit`, `plan`); a rule with no modes is always active. Admin policies sit in a higher tier than user policies, which sit above workspace policies.

8.3 MCP servers you trust

Add `"trust": true` to a server in `mcpServers` to skip confirmation for all its tools:

```
~/GEMINI/SETTINGS.JSON (EXCERPT)
```

```
{
  "mcpServers": {
    "github": {
      "httpUrl": "https://api.githubcopilot.com/mcp/",
      "trust": true
    }
  }
}
```

8.4 Folder trust

The first time you open a folder, Gemini CLI asks whether to trust it. Untrusted folders restrict what runs, and project settings and `.env` files may be ignored. Your answers are saved in `~/gemini/trustedFolders.json`. For one non-interactive session, `--skip-trust` trusts the current folder. **Trust parent folder** trusts everything under a directory such as `~/code`.

8.5 The sandbox

`--sandbox` (`-s`) runs tools in an isolated environment. Set it permanently with `tools.sandbox` or the `GEMINI_SANDBOX` environment variable:

VALUE	MEANING
<code>true</code>	Use the default sandbox for your platform
<code>docker</code> or <code>podman</code>	Run in a container (default image <code>ghcr.io/google/gemini-cli</code>)
A profile or command	Custom sandbox

On macOS the default Seatbelt profile, `permissive-open`, restricts writes but allows the network. The sandbox only sees the current project unless you mount more. Pairing `--approval-mode=yolo` with `--sandbox` is the sensible maximum-autonomy setup on your own machine.

8.6 Headless mode

Gemini CLI runs **headless** (no interactive screen) when you pass `-p` or when it isn't attached to a terminal:

TERMINAL

```
gemini -p "Update the changelog from the last 20 commits" \  
  --approval-mode=yolo \  
  --output-format json
```

`--output-format` accepts `text`, `json` (one object with `response` and `stats`), or `stream-json`. `--include-directories` adds folders to the workspace; `--resume latest` continues the last session.

8.7 The GitHub Action

Google's `google-github-actions/run-gemini-cli` action runs Gemini CLI in a workflow. Pass `gemini_api_key` (or Google Cloud workload identity inputs), a `prompt`, and optionally `settings`, a JSON string written to the project's `.gemini/settings.json`. Since the runner is disposable, YOLO-style autonomy there is reasonable.

8.8 What still asks in Gemini CLI

STILL ASKS OR REFUSES	WHY
YOLO refused	<code>security.disableYoloMode</code> or <code>admin.secureModeEnabled</code> is true
No "Always allow" options	<code>security.disableAlwaysAllow</code> or admin secure mode
Tools restricted in a new folder	Folder trust; trust it or use <code>--skip-trust</code>
A specific command asks	A policy rule with <code>ask_user</code>
A command is blocked	A policy <code>deny</code> rule, which also applies in YOLO

- ☐ An alias starts Gemini with `--approval-mode=yolo --sandbox`
- ☐ `defaultApprovalMode` is `auto_edit` for interactive use
- ☐ Policy files deny the few things you never want
- ☐ Trusted MCP servers have `"trust": true`
- ☐ Your code folder is trusted (parent-folder trust)

VS Code and GitHub Copilot

GitHub Copilot is GitHub's AI assistant. For autonomy it comes in three forms: **agent mode** in VS Code's chat, the **Copilot CLI** in your terminal, and the **Copilot cloud agent** (formerly the coding agent), which works on GitHub's servers and opens pull requests.

IN THIS CHAPTER

- Allow all, Autopilot, and the global auto-approve setting in VS Code
- Copilot CLI flags, permission modes, and programmatic runs
- Tool rules: allow some commands, deny others
- The cloud agent's firewall and MCP settings
- What still asks

9.1 Copilot agent in VS Code

LEVEL	HOW
Allow all	Choose Allow all in the chat session's permission picker: "runs all tool calls without confirmation"
Autopilot	Choose Autopilot in the mode picker: auto-approves tools, retries errors, and answers blocking questions itself
Advanced Autopilot (preview)	<code>"chat.autopilot.advanced.enabled": true</code> , where a separate model decides when the task is complete
Every workspace	<code>"chat.tools.global.autoApprove": true</code>
This session only	Type <code>/yolo</code> (or <code>/autoApprove</code>); undo with <code>/disableYolo</code>

VS CODE SETTINGS.JSON

```
{
  "chat.tools.global.autoApprove": true,
  "chat.autopilot.advanced.enabled": true
}
```

Finer-grained alternatives: `chat.tools.terminal.autoApprove` (terminal command rules) and `chat.tools.urls.autoApprove` (web addresses). Terminal writes detected *outside* the workspace still ask by default; that's governed by `chat.tools.terminal.blockDetectedFileWrites`.

Running the **Claude agent harness inside VS Code**? Its equivalent of bypass is

`"github.copilot.chat.claudeAgent.allowDangerouslySkipPermissions": true`. The **Claude Code extension** has its own **Allow dangerously skip permissions** toggle in its settings, after which **Bypass permissions** appears in its mode indicator.

Still asks in VS Code: tools or terminal rules you've set to `false`; organization (device-management) policies; and, unless you change `blockDetectedFileWrites`, terminal writes it detects outside the workspace. An enabled sandbox still limits file and network access even in Allow all.

Finer-grained terminal rules

If you'd rather not approve everything, `chat.tools.terminal.autoApprove` takes a map of commands to `true` (run freely) or `false` (always ask). Patterns between slashes are treated as regular expressions:

VS CODE SETTINGS.JSON

```
{
  "chat.tools.terminal.autoApprove": {
    "npm": true,
    "git": true,
    "/^git push --force/": false,
    "rm": false
  }
}
```

9.2 Copilot CLI

TERMINAL

```
copilot --yolo
```

`--yolo` is an alias for `--allow-all`, which combines `--allow-all-tools`, `--allow-all-paths`, and `--allow-all-urls`. Inside a session, `/yolo` or `/allow-all` does the same; `/permissions` switches between `default`, `assisted` (an AI reviews requests, Level 3), and `allow-all`, and `/permissions reset` clears your approvals.

FLAG	MEANING
<code>--allow-all-tools</code>	Tools run without confirmation (env <code>COPILOT_ALLOW_ALL</code>); required for programmatic use
<code>--allow-tool='shell(git:*)'</code>	Pre-approve a tool or command family
<code>--deny-tool='shell(git push)'</code>	Block it; deny beats allow
<code>--allow-url=...</code> , <code>--deny-url=...</code>	Web access rules
<code>--add-dir=PATH</code>	Extra folder the agent may use
<code>--autopilot</code>	Keep working until the task is complete
<code>--max-autopilot-continues=N</code>	Cap autopilot's continuations
<code>--no-ask-user</code>	Disable the question tool so it never stops to ask
<code>--secret-env-vars=VAR</code>	Hide a variable from shells and MCP servers

Tool rules use `shell(...)` for commands, `write` for file writes, and the server name for MCP tools, such as `MyMCP(create_issue)` or all of `MyMCP`. `shell(git:*)` matches `git push` and `git pull` but not `gitea`.

Programmatic runs

TERMINAL

```
copilot -p "Fix the lint errors and commit" \
  --allow-all-tools --allow-all-paths \
  --no-ask-user -s \
  --share=./copilot-run.md
```

`-p` runs once and exits; `-s` prints only the answer; `--share` saves a Markdown transcript. `--plan --mode autopilot` plans first and then carries on into autopilot without waiting for a human. In CI, put a fine-grained token with the **Copilot Requests** permission in `COPILOT_GITHUB_TOKEN`.

NOTE

Admins can block every allow-all option with `permissions.disableBypassPermissionsMode` set to `"disable"`, or allow only assisted approval with `"allow-auto-only"`.

9.3 Copilot cloud agent (on GitHub)

Assign an issue to Copilot and it works in its own GitHub Actions environment, then opens a pull request. It already runs without prompts. Widen what it can reach in the repository's

Settings > **Copilot** > **Internet access** :

- **Enable firewall: Disabled** gives it unrestricted internet (maximum reach, maximum exfiltration risk).
- **Recommended allowlist**: on by default, covering common package registries.
- **Custom allowlist**: add the domains your build needs.

Add MCP servers in **Settings** > **Copilot** > **MCP servers**, then click **Save MCP configuration**. Secrets the servers need go in Agents secrets whose names start with `COPILOT_MCP_`. Organization owners can set firewall rules for every repository, including **Let repositories decide**.

9.4 What still asks in VS Code and Copilot

STILL ASKS OR BLOCKS	WHY / WORKAROUND
Tools or terminal rules set to <code>false</code>	Your own rules; change them
Terminal writes detected outside the workspace	<code>chat.tools.terminal.blockDetectedFileWrites</code>
Organization (device-management) policies	Admin control
Sandbox limits in Allow all	An enabled sandbox still limits files and network
CLI allow-all flags ignored	<code>disableBypassPermissionsMode</code> policy
Cloud agent can't reach a site	Firewall allowlist

- ☐ VS Code: `chat.tools.global.autoApprove` on, or Autopilot chosen
- ☐ Copilot CLI: alias with `--yolo`, or `--allow-tool` rules for daily use
- ☐ Programmatic runs use `--allow-all-tools --no-ask-user`
- ☐ Cloud agent firewall and MCP servers set for the repository

Cursor

Cursor is an AI-first code editor built on VS Code. It has three agents: the one in the editor, a command-line agent called `agent`, and **Cloud Agents** that run on Cursor's machines.

IN THIS CHAPTER

- The three Run Modes: Auto-review, Allowlist, Run Everything
- `permissions.json` and `sandbox.json`
- The Cursor CLI's permission tokens and `--force`
- Cloud Agents and their network modes
- What still asks

10.1 The editor's agent

TURN ON RUN EVERYTHING

- 1 **Open settings:**
`Settings` > `Agents` > `Approvals & Execution`.
- 2 **Set the Run Mode to Run Everything.**
Every tool call, including shell commands, MCP tools, and web fetches, runs automatically, with no sandbox and no classifier.
- 3 **Read Access** is hidden in Run Everything (nothing is checked). In other modes, set it to **System** to allow reads outside the workspace without asking.

Prefer a safety net? **Auto-review** runs allowlisted calls immediately, sandboxes shell commands where it can, and sends the rest to a classifier. Tune it in plain English in

```
~/.cursor/permissions.json:
```

```
~/CURSOR/PERMISSIONS.JSON
```

```
{
  "autoRun": {
    "allow_instructions": [
      "Any git command in this project, including push to branches starting with agent/.",
      "Installing packages with npm, pnpm, or pip."
    ],
    "block_instructions": []
  }
}
```

The sandbox's network access and extra writable paths live in `~/cursor/sandbox.json`.

RUN MODE	WHAT RUNS WITHOUT ASKING	CLASSIFIER
Auto-review	Allowlisted calls; other shell commands in the sandbox; the rest reviewed	Yes
Allowlist	Only your allowlist (sandboxed shell commands optional)	No
Run Everything	Every tool call	No

Auto-review covers shell, MCP, and Fetch calls. Its classifier runs on small Cursor-managed models; on a team plan, if it's grayed out, an admin needs to allow the fallback model in **Team Settings > Models**. Cursor states plainly that Auto-review is not a security boundary. `permissions.json` can live in `~/cursor/` (all projects) or `<project>/cursor/` (one project); a team configuration in the dashboard overrides both.

TIP

The easiest way to edit `permissions.json` is to ask the agent: "I want every AWS CLI command to go through approval first." It writes the instruction for you.

10.2 The Cursor CLI

Install it with `curl https://cursor.com/install -fsS | bash` (Windows PowerShell: `irm 'https://cursor.com/install?win32=true' | iex`). The command is `agent`. It reads permissions from `~/.cursor/cli-config.json` (global) or `.cursor/cli.json` (project):

~/.CURSOR/CLI-CONFIG.JSON

```
{
  "permissions": {
    "allow": [
      "Shell(*)", "Read(**)", "Write(**)",
      "WebFetch(*)", "Mcp(*:*)"
    ],
    "deny": ["Shell(rm)", "Write(**/.env*)"]
  }
}
```

TOKEN	MATCHES
<code>Shell(git)</code>	Any <code>git</code> subcommand
<code>Shell(curl:*)</code>	<code>curl</code> with any arguments
<code>Read(src/**/*.*ts)</code> / <code>Write(src/**)</code>	Files by glob
<code>WebFetch(*.example.com)</code>	Fetches from that domain
<code>Mcp(server:tool)</code>	One MCP tool; <code>*</code> for wildcards

Hands-off CLI runs

TERMINAL

```
agent -p --force --trust --approve-mcps \
  --output-format json \
  "Refactor this module to modern syntax"
```

FLAG	MEANING
<code>-p</code> , <code>--print</code>	Non-interactive; can use write and shell tools
<code>-f</code> , <code>--force</code> (alias <code>--yolo</code>)	Allow commands unless explicitly denied
<code>--trust</code>	Trust the workspace without asking (headless only)
<code>--approve-mcps</code>	Approve all MCP servers automatically
<code>--sandbox enabled</code> / <code>disabled</code>	Turn the sandbox on or off
<code>-w</code> , <code>--worktree [name]</code>	Work in a new git worktree under <code>~/.cursor/worktrees/</code>

Without `--force`, print mode only *proposes* file changes. For scripts, set `CURSOR_API_KEY` instead of signing in.

10.3 Cloud Agents

Cloud Agents run in isolated VMs with full development environments, as many in parallel as you like. Start them from the editor, the web, Slack (`@cursor`), or a GitHub or Bitbucket comment (`@cursor`). They clone your repo, work on a separate branch, and push for handoff. Add secrets at cursor.com/dashboard/cloud-agents.

Cloud Agents have internet access by default. Network modes:

MODE	REACH
Allow all network access	Any host
Default + allowlist	Cursor's defaults plus your domains
Allowlist only	Only your domains (plus Cursor's own services and your git host)

10.4 What still asks in Cursor

STILL ASKS	WHY
Edits to configuration files (like workspace settings)	Default guardrail outside Run Everything
Writes to <code>.git/config</code> , <code>.git/hooks</code> , <code>.vscode</code> , <code>.cursorignore</code>	Protected paths in Auto-review and Allowlist
Commands that must leave the sandbox	Auto-review and Allowlist
Every new MCP connection	Approve once, or <code>--approve-mcps</code> in the CLI
Team admin policies and built-in security rules	Can't be weakened by local files

- ☐ Editor: Run Everything (in a box) or Auto-review with allow instructions
- ☐ `~/.cursor/permissions.json` has your standing instructions
- ☐ CLI: `cli-config.json` allows what you use; scripts use `-p --force --trust`
- ☐ Cloud Agents: network mode and secrets set

More Coding Agents

The big names aren't the only agents worth running hands-off. This chapter covers the rest of the current field, from Cognition's Devin family to open-source tools you can run anywhere.

IN THIS CHAPTER

- Devin Desktop (formerly Windsurf), Devin CLI, and Devin in the cloud
- Cline's Auto Approve and YOLO mode, in the editor and the CLI
- Aider, OpenHands, and Goose: open-source terminal agents
- Browser app builders
- Tools that are gone or that we couldn't verify

11.1 Devin Desktop (formerly Windsurf)

Cognition's editor, long known as **Windsurf**, now appears in its docs as **Devin Desktop**, with the **Cascade** agent inside it. Cascade's terminal commands follow one of four **auto-execution levels**, chosen in the **Devin Settings** panel in the bottom-right corner of the editor:

LEVEL	BEHAVIOR
Disabled	Every command needs approval
Allowlist Only	Only allow-listed commands run
Auto	The agent judges safety; risky commands ask (premium models only)
Turbo	Everything runs, except your deny list

Turbo is the maximum. Build your lists from the command cards: when Cascade waits for approval, open the dropdown next to **Run** and choose **Prefix** (auto-run every command starting the same way, like `git *`), **Allow** (that exact command), or **Deny**.

NOTE

On Teams and Enterprise, admins can cap the maximum level (for example at Auto, which hides Turbo) and add team-wide lists in **Admin Portal > Team Settings > Terminal Commands > Manage Lists**. A team deny list always wins.

The **Devin Local** agent inside the editor uses the Devin CLI permission model below instead.

11.2 Devin CLI

The **Devin CLI** is Cognition's terminal agent. It has five permission modes:

MODE	SHELL AND FETCH	FILE EDITS
Normal	Ask	Ask
Accept Edits	Ask	Auto in the workspace
Smart	Auto when a fast model judges them safe	Auto in the workspace
Bypass	Auto	Auto
Autonomous (with <code>--sandbox</code>)	Auto, contained by the OS sandbox	Ask

TERMINAL

```
devin --permission-mode bypass
```

Inside a session, `/bypass` (aliases `/yolo`, `/dangerous`) or `/mode bypass` switches over; `DEVIN_PERMISSION_MODE=smart devin` sets the mode from the environment. `devin -p "prompt"` runs once and prints the answer.

Rules live in `~/.config/devin/config.json` (you), `.devin/config.json` (the project, committed), and `.devin/config.local.json` (you, in this project, git-ignored):

~/.CONFIG/DEVIN/CONFIG.JSON

```
{
  "permissions": {
    "allow": ["Read(**)", "Exec(git)", "Exec(npm run)"],
    "deny": ["Exec(sudo)"]
  }
}
```

Organization deny and ask rules stay active in Smart, Bypass, and Autonomous modes.

11.3 Devin (cloud)

Cloud **Devin** sessions run in Cognition's VMs and already work without step-by-step approvals. Your levers are what it can reach and what it can spend: **Secrets** (organization, personal, repo, or session), **Security Profiles** that restrict network, MCP, git, and GitHub CLI access, and **ACU limits** (Devin's usage units) per organization and per user on Enterprise.

11.4 Cline

Cline is an open-source agent for VS Code and JetBrains, with a CLI.

AUTO APPROVE SETTING	ALLOWS
Read project files / Read all files	Reads inside / outside the workspace
Edit project files / Edit all files	Edits inside / outside the workspace
Execute safe commands / Execute all commands	Commands the model marks safe / all commands
Use the browser	Web fetching and searching
Use MCP servers	MCP tools and resources

The “all” options only extend their base toggle, so turn both on. For everything at once, open **Cline Settings**, then **Features**, and tick **YOLO Mode**: file changes anywhere, all commands, browser actions, MCP tools, and Plan-to-Act switches are approved.

The **Cline CLI** starts in act mode with auto-approve on by default (`--auto-approve` defaults to `true`). `cline "your task" --json` gives machine-readable output, and `cline` supports cron scheduling for recurring jobs.

11.5 Aider

Aider is a long-standing open-source terminal pair programmer. It edits files directly and commits each change to git automatically (`--auto-commits` is on by default), which makes rollback easy.

TERMINAL

```
aider --yes-always --auto-test --test-cmd "npm test" \  
  --message "Add input validation to the signup form"
```

OPTION	MEANING
<code>--yes-always</code> (env <code>AIDER_YES_ALWAYS</code>)	Say yes to every confirmation
<code>--message "..."</code>	Send one message, apply the reply, exit
<code>--auto-test</code> + <code>--test-cmd</code>	Run tests after each change
<code>--no-suggest-shell-commands</code>	Stop it proposing shell commands at all

Put defaults in `.aider.conf.yml` in your repo root or home folder.

11.6 OpenHands

OpenHands is an open-source agent platform with a CLI, a web GUI, and a cloud service. It runs code in a sandbox (Docker by default).

COMMAND	BEHAVIOR
<code>openhands</code>	Asks before actions (default)
<code>openhands --llm-approve</code>	An AI reviews actions
<code>openhands --always-approve</code>	Runs everything
<code>openhands --headless -t "task"</code>	No UI; always runs in always-approve mode

Add `--json` in headless mode for one JSON event per line, or `-f task.txt` to read the task from a file.

11.7 Goose

Goose is an open-source agent (desktop app and CLI) whose default mode is already **Completely Autonomous**.

MODE	COMMAND	BEHAVIOR
Completely Autonomous	<code>/mode auto</code>	No approvals (default)
Smart Approval	<code>/mode smart_approve</code>	Low-risk actions run; others ask
Manual Approval	<code>/mode approve</code>	Asks before tools
Chat Only	<code>/mode chat</code>	No tools

Set the default with `GOOSE_MODE=auto` or in **Settings > Chat > Default Mode** in the desktop app. `goose run -t "instructions"` runs a task and exits; `GOOSE_MAX_TURNS` caps how long it can go without input (default 1000).

11.8 Browser app builders

TOOL	MAX-AUTONOMY SETTING
Replit Agent	In the Agent's tools panel, set Autonomy Level to Max for long, unsupervised runs (labeled beta)
Lovable, Bolt, v0, Base44	They act without per-step approvals already. Give them a connected GitHub repo and backend, and let them run; review via version history

11.9 Gone, or check for yourself

- **Roo Code** was shut down on May 15, according to the notice in its repository README. Its README points users to Cline and to ZooCode, a community fork. Remove it from unattended setups.
- **Amp** (Sourcegraph's agent) and **Jules** (Google's asynchronous coding agent) are still offered, but we couldn't confirm their current approval settings for this edition. Both run largely without step-by-step prompts; check their docs for the exact switches before relying on them.

- ☐ Devin Desktop at Turbo, with a deny list, only inside a box
- ☐ Devin CLI started with `--permission-mode bypass` or `smart`
- ☐ Cline: base toggles plus “all” options, or YOLO Mode
- ☐ Aider aliased with `--yes-always`
- ☐ Any Roo Code setups migrated

GitHub: Tokens and Repository Settings

GitHub is where agents push code, open pull requests, and trigger automation. Here's how to give them full power with minimal friction.

12.1 Authenticate agents broadly

The GitHub CLI (used by Claude Code, Codex, and others):

TERMINAL

```
gh auth login
gh auth refresh -h github.com -s repo,workflow,read:org,gist,delete_repo
```

The `workflow` scope lets agents edit `.github/workflows/` files; without it, pushes that touch workflows are rejected. `delete_repo` is only needed if you want agents to delete repositories.

A fine-grained personal access token (for MCP servers, n8n, and scripts): go to [Settings](#) > [Developer settings](#) > [Personal access tokens](#) > [Fine-grained tokens](#), then **Generate new token**:

OPTION	MAXIMUM-AUTONOMY CHOICE
Resource owner	Your account (or the organization the agent works in)
Expiration	As long as you're comfortable with, or custom
Repository access	All repositories (or every repo the agent should touch)
Contents	Read and write
Pull requests	Read and write
Issues	Read and write
Workflows	Read and write
Actions	Read and write
Administration	Read and write (only if agents create repos or change settings)
Metadata	Read (required)

Classic tokens are simpler but blunter: the `repo` and `workflow` scopes cover most agent work.

For teams: install the **Claude GitHub App** (run `/install-github-app` in Claude Code) on **All repositories** so `@cLaude` works everywhere.

TIP

GitHub Apps vs tokens. A **GitHub App** is an integration installed on chosen repositories. It gets short-lived tokens automatically, acts under its own name, and doesn't depend on a person's account. For a team, an App (like the Claude GitHub App, or your own) beats a personal token. For one person, a machine-user account with a fine-grained token is simpler.

CREATE A MACHINE-USER SETUP IN TEN MINUTES

- 1 **Create a second GitHub account** (for example `yourname-bot`) with its own email and 2FA.
- 2 **Invite it** as a collaborator with **Write** access on the repositories agents use.
- 3 **Sign in as the bot** and create a fine-grained token with the permissions above, scoped to those repositories.
- 4 **Store the token** in your agent's environment (`GH_TOKEN`) or your CI secrets.
- 5 **Tell git who it is:**

```
git config user.name "yourname-bot"
```

and the bot's email, inside the agent's box.

12.2 Repository settings that remove friction

SETTING	WHERE	CHOOSE
Workflow permissions	<code>Settings</code> > <code>Actions</code> > <code>General</code>	Read and write permissions
Let Actions open and approve PRs	Same page	Tick Allow GitHub Actions to create and approve pull requests
Auto-merge	<code>Settings</code> > <code>General</code> , under Pull Requests	Tick Allow auto-merge ; agents can then run <code>gh pr merge --auto --squash</code>
Branch rules	<code>Settings</code> > <code>Rules</code> > <code>Rulesets</code>	Remove required reviews, or add the agent's account or app to the ruleset's Bypass list
Delete merged branches	<code>Settings</code> > <code>General</code>	Tick Automatically delete head branches

TIP

A powerful, still-sane setup: leave `main` protected by status checks only (tests must pass), put your agent's account on nothing else, and turn on auto-merge. Agents ship on their own, but only green code lands.

12.3 Branch protection that agents can live with

A **ruleset** is a set of rules GitHub enforces on branches. The trick for autonomy is to make rules that check *results* rather than requiring a person:

RULE	KEEP IT?	WHY
Require status checks to pass	Yes	Only green code lands
Block force pushes to <code>main</code>	Yes	History can't be rewritten
Restrict deletions of <code>main</code>	Yes	The branch can't vanish
Require pull request reviews	Optional	Removes autonomy unless the agent is on the bypass list
Require signed commits	Optional	Agents need a signing key

Agents then work on branches like `agent/fix-login`, open pull requests, and enable auto-merge. When tests pass, GitHub merges on its own.

PROMPT

Work on a new branch named `agent/<short-description>`. When the change is complete and tests pass locally, push, open a pull request with a clear summary, and run: `gh pr merge --auto --squash`

12.4 What still prompts on GitHub

- **Sudo mode:** sensitive actions in the web interface (deleting repos, changing security settings) re-ask for your password or 2FA. Tokens and the API aren't affected.
 - **Push protection:** pushes containing detected secrets are blocked until bypassed for that push.
 - **Claude's auto mode** blocks merging pull requests no human approved, and force pushes, by default. Use bypass mode, or grant a specific standing approval, if you want agents to do that.
 - **Organization policies** (SSO token authorization, required reviews, Copilot policies) override your personal choices.
 - **Copilot cloud agent and Actions approvals:** workflows triggered by first-time contributors or bots may need a maintainer to click **Approve and run**, depending on [Settings](#) > [Actions](#) > [General](#) .
- ☐ Agents use a machine user or GitHub App, not your main account
 - ☐ Fine-grained token scoped to the agent's repos, with [workflow](#) access if it edits CI
 - ☐ Workflow permissions set to read and write; Actions may open PRs
 - ☐ [main](#) protected by status checks and no force-push; auto-merge allowed
 - ☐ Push protection left on

MCP Servers and Connectors

Every MCP tool call can trigger an approval prompt. Here's how to approve them up front in each host.

HOST	AUTO-APPROVE MCP TOOLS
Claude Code	Allow rules: <code>"mcp__github__*"</code> , <code>"mcp__notion__*"</code> , one per server (a bare <code>mcp__*</code> is ignored). Or bypass mode. <code>"enableAllProjectMcpServers": true</code> skips approving <code>.mcp.json</code> servers
Claude apps	Connector, then Tool permissions , then Always allow
Cursor	Run Everything covers MCP calls; Auto-review sends them to the classifier
VS Code	Allow all , Autopilot, or <code>chat.tools.global.autoApprove</code>
Codex	<code>approval_policy = "never"</code> , or <code>default_tools_approval_mode = "approve"</code> per server
Gemini CLI	<code>--approval-mode=yolo</code> , or per server <code>"trust": true</code> in its <code>mcpServers</code> entry
n8n AI Agent	Don't add human-review steps to its tools
Cursor CLI	<code>--approve-mcps</code> , plus <code>Mcp(server:tool)</code> allow tokens
Copilot CLI	<code>--allow-tool='ServerName'</code>

13.1 Give servers enough power

- **Remote OAuth servers** (GitHub, Notion, Linear...) act with *your* account's permissions. Approve every scope they request during sign-in.
- **Token-based servers:** create the token with the broad scopes from Chapter 12 (GitHub) or full capabilities (Notion, Chapter 15).
- **Local servers:** they run with your user's file and network permissions. Point filesystem servers at the folders you want writable.

Still asks: tools a server marks as needing user interaction (`requiresUserInteraction`) always prompt in Claude Code, and so do connector tools your organization set to “ask.” ChatGPT's developer-mode connectors may still ask you to confirm write actions.

13.2 What MCP is, in one paragraph

The **Model Context Protocol (MCP)** is a standard way for agents to use outside tools. An **MCP server** is a small program (on your computer, or a web address) that offers tools such as “create issue” or “search pages”. The **host** is the agent app that connects to it. Each tool call is a separate action the host may ask you to approve, which is why MCP is a common source of prompts.

13.3 Approve servers, then tools

Most hosts ask twice: once to *connect* a server, and again for each *tool call*. Remove both:

HOST	APPROVE CONNECTING	APPROVE CALLS
Claude Code	<code>enableAllProjectMcpServers: true</code>	<code>mcp__server__*</code> allow rules
Cursor CLI	<code>--approve-mcps</code>	<code>Mcp(*:*)</code>
Gemini CLI	Trusted folder	<code>"trust": true</code>
Codex	Configured in <code>config.toml</code>	<code>approval_mode = "approve"</code>

⚠ WARNING

An MCP server that reads email, web pages, or tickets is a prompt-injection doorway. Auto-approve its *read* tools freely, but think twice before auto-approving *write* tools on the same agent that reads untrusted content.

- ☐ Every server you rely on is approved to connect
- ☐ Its tools are pre-approved in each host you use
- ☐ Tokens behind those servers follow the “narrow but complete” rule

n8n

14.1 Let AI agents build and run your workflows

OPEN N8N TO AI CLIENTS

1 Create an API key:

Settings > **n8n API** > **Create an API key**. Copy it into scripts, Claude Code, or other tools that call n8n's REST API.

2 Enable instance-level MCP:

Settings > **Instance-level MCP**, then **Enable MCP access** (owner or admin). Use **Connect** to get the server URL (ending in `/mcp-server/http`) and connect Claude, Cursor, Codex, or ChatGPT with OAuth or an API key.

3 Expose workflows:

under **Workflows enabled**, click **Enable workflows** and add the ones agents may run, or in each workflow's **Settings** toggle **Available in MCP**. Clients can build and edit workflows (n8n 2.13+) as well as run them.

4 Agents in workflows:

in AI Agent nodes, attach the tools you want and *don't* add human-in-the-loop approval to them.

14.2 Self-hosted: unlock everything

TERMINAL

```
# Re-enable nodes n8n 2.0+ blocks by default (Execute Command, etc.)  
NODES_EXCLUDE="[]"
```

Set it in your Docker or Docker Compose environment and restart. **Execute Command** runs shell commands on the n8n host. It's powerful and dangerous, so keep self-hosted n8n in a container.

Still asks / limits: n8n Cloud doesn't offer Execute Command at all. The MCP server only lets clients run or see the full contents of workflows you enable for it (`search_workflows` previews everything you can access). Admins can disable MCP or Agents instance-wide.

14.3 n8n Agents

Beyond AI Agent *nodes* inside workflows, n8n now has first-class **Agents**: saved agents with their own tools, memory, and chat channels. Two points matter for autonomy:

- An Agent is built as a **draft**. It only runs on its channels after you **publish** it.
- Tools that need **human approval** pause the agent until someone answers. Leave approval off the tools you want it to use freely, and keep it on anything that sends messages or spends money.

14.4 Run n8n itself safely

A CONTAINED SELF-HOSTED N8N

- 1 **Run it in Docker** on a small VM, not on your laptop.
 - 2 **Give it its own credentials**
an agent GitHub token, a separate Slack bot, its own API keys with spend caps.
 - 3 **Back up the database volume** nightly.
 - 4 **Only then** set `NODES_EXCLUDE="[]"` if you need Execute Command.
- ☐ API key created; instance-level MCP enabled
 - ☐ Agent-usable workflows marked **Available in MCP**
 - ☐ AI Agent tools have no human-approval step (where safe)
 - ☐ Self-hosted n8n runs in a container with backups

Notion

15.1 Notion MCP (simplest, full personal access)

Connect `https://mcp.notion.com/mcp` from Claude (**Settings** > **Connectors** > **Notion**), Claude Code, Cursor, or ChatGPT. Signing in with OAuth gives the AI exactly **your** access: every page you can see and edit. Then set its tools to **Always allow** in the host app (Chapter 13).

15.2 An API connection with every capability

For n8n, Zapier, and scripts:

CREATE AN ALL-ACCESS CONNECTION

- 1 **Open** `notion.com/profile/integrations` (or **Settings** > **Connections** > **Develop or manage integrations**) and create a new **internal connection** for your workspace. You must be a workspace owner.
- 2 **Enable every capability:**
read content, update content, insert content, read comments, insert comments, and read user information including email addresses.
- 3 **Grant content access broadly.**
In the connection's **Content access** tab, add every top-level page in each teamspace you want covered. Child pages inherit access.
- 4 **Copy the API token** into your tools.

15.3 Notion AI agents

Custom Agents run on schedules or triggers. Give each agent the sources and tools it needs in the Agent Builder, and review its run log rather than approving each action. On Business and Enterprise plans, admins control who can create agents, and Custom Agents use paid Notion credits.

Still limited: a connection can never see pages you haven't shared with it. Workspace owners can restrict or revoke MCP clients in [Settings](#) > [Connections](#) . Agents run inside Notion's own permission system and can't exceed the access of the content shared with them.

15.4 Which option to use

YOU WANT	USE
Claude, Cursor, or ChatGPT to work in your whole workspace	Notion MCP with OAuth
A script or n8n to write to specific pages	Internal connection with an API token
Work that runs inside Notion on a schedule	Notion Custom Agents
Agents kept away from private pages	A separate teamspace shared only with the connection

TIP

Notion keeps page history. If an agent rewrites a page badly, open the page's  menu, then **Version history**, and restore an earlier version (how far back depends on your plan).

- ☐ Notion MCP connected and its tools set to **Always allow**
- ☐ API connection has every capability and top-level pages shared
- ☐ Agent-writable content lives in a teamspace you can afford to lose

Browsing Agents

A **browsing agent** drives a real web browser: it clicks, types, fills forms, and reads pages for you. ChatGPT’s agent mode (which absorbed the earlier Operator), Claude in Chrome, and Claude Desktop’s built-in browser are the main examples. They are the hardest agents to make fully hands-off, on purpose: the web is where prompt injection lives, and where money and accounts are.

IN THIS CHAPTER

- What browsing agents can and can’t do without you
- How to cut down their prompts
- A safe “agent browser profile”

16.1 What always stops for you

TOOL	STOPS FOR
ChatGPT agent	Confirmations before high-impact actions such as purchases or sending; “watch mode” on sensitive sites, where you must keep the tab in view
Claude in Chrome	First action on each site; purchases, account creation, CAPTCHAs; classifier-flagged risky page actions
Claude Desktop browser	Safety classifiers on actions on external websites, in every permission mode

These aren’t settings you can remove. Plan around them: let the agent research, compare, and fill in drafts, and do the final “buy” or “send” click yourself.

16.2 Reduce the prompts you can

- **Approve sites once.** In Claude in Chrome, answer a site's first action card with **Always allow**.
- **Sign in ahead of time.** Agents stop at login screens. Log in to the sites the agent will use before you hand over.
- **Write complete instructions.** Include the details the agent would otherwise ask about (sizes, dates, budgets, which account).

PROMPT

Find three flights from Boston to Denver on 12 November, leaving after 9 a.m., under \$350, carry-on only. Compare them in a table. Don't book anything; stop on the final payment page for the cheapest one so I can review it.

16.3 Build an agent browser profile

GIVE THE AGENT ITS OWN BROWSER

- 1 **Create a new browser profile** (Chrome: profile icon, then **Add**) named "Agent".
- 2 **Install only the agent's extension** in it, nothing else.
- 3 **Sign in only to the accounts the agent needs**, ideally its own accounts, never your bank or main email.
- 4 **Don't save payment cards** in this profile.
- 5 **Use it only for agent tasks.**
Your personal profile stays out of reach.

⚠ DANGER

A browsing agent signed in to your main email can be tricked by a single malicious web page into forwarding messages. Separate profiles and separate accounts are the containment for browsers.

- ☐ Browsing agents run in a separate browser profile
- ☐ Sites you use often are set to **Always allow**
- ☐ Payment and send steps are left for you to click

Running Unattended

“Unattended” means the agent runs while you’re asleep, in a meeting, or away from your desk. There are four common ways to do it: a terminal session that keeps running, a schedule on your own machine, a schedule in CI, and a vendor’s cloud.

IN THIS CHAPTER

- `tmux`: keep a session alive after you disconnect
- `cron` and `launchd`: schedules on your own machine
- CI schedules on GitHub Actions
- Long-running sessions and how to keep them on track
- A wrapper script with time and money limits

17.1 Keep a session alive with `tmux`

tmux is a free terminal program that keeps sessions running after you close the window or lose your connection. Install it with `brew install tmux` (Mac) or `sudo apt install tmux` (Ubuntu).

Start a named session: `tmux new -s agent`

Detach (leave it running): `Ctrl` + `B`, then `D`

List sessions: `tmux ls`

Reattach: `tmux attach -t agent`

Kill it: `tmux kill-session -t agent`

Start your agent in bypass inside the tmux session, detach, and check back later. On a cloud VM, this survives your laptop going to sleep. Claude Code's `--bg` flag and `claude -w <name> --tmux` do similar jobs for you.

17.2 Schedules on your own machine: cron

cron runs commands on a timetable on Linux and macOS. Edit your schedule with `crontab -e`. Each line has five time fields (minute, hour, day of month, month, day of week) and a command:

CRONTAB -E

```
# Every weekday at 07:30, run the nightly agent script
30 7 * * 1-5 /home/me/bin/agent-nightly.sh >> /home/me/logs/agent.log 2>&1
```

TIP

cron runs with a nearly empty environment. Use full paths to programs, and load your API key inside the script (for example `source ~/.agent-env`) rather than relying on your shell profile.

On a Mac, **launchd** is Apple's native scheduler, and cron also works. On Windows, use **Task Scheduler**.

17.3 A wrapper script with seat belts

Put limits around every unattended run: a time limit, a turn limit, a budget, a log, and a git safety branch.

~/BIN/AGENT-NIGHTLY.SH

```
#!/usr/bin/env bash
set -euo pipefail
source ~/.agent-env          # exports ANTHROPIC_API_KEY
cd ~/code/my-app
git fetch origin && git switch -C agent/nightly origin/main

timeout 45m claude -p "$(cat ~/prompts/nightly.md)" \
  --dangerously-skip-permissions \
  --permission-prompts none \
  --max-turns 80 \
  --max-budget-usd 8 \
  --output-format json > ~/logs/nightly-$(date +%F).json

git push -u origin agent/nightly
```

LINE	SEAT BELT
<code>set -euo pipefail</code>	Stop on the first error
<code>git switch -C agent/nightly</code>	Work on a fresh branch, never <code>main</code>
<code>timeout 45m</code>	Kill the run after 45 minutes
<code>--max-turns</code> , <code>--max-budget-usd</code>	Cap effort and spend
JSON log per day	A record of what happened

Swap the `claude` line for `codex exec --yolo`, `gemini -p ... --approval-mode=yolo`, `copilot -p ... --allow-all-tools`, or `agent -p --force` to use another tool.

17.4 Schedules in CI

GitHub Actions can run on a cron schedule (`on: schedule`), in UTC. Chapter 5 has a full Claude Code example and Chapter 7 the Codex Action. Rules worth remembering:

- Schedules run only from the default branch.
- In public repositories, GitHub disables schedules after 60 days without activity.
- Runners are disposable, so bypass modes are reasonable there. Keep secrets limited to the agent step.

17.5 Vendor clouds

The simplest unattended option is a vendor's cloud: Claude Code **routes** and cloud sessions, **Codex Cloud** scheduled tasks (created from ChatGPT web or the desktop app), **Copilot cloud agent**, **Cursor Cloud Agents**, and **Devin**. Your laptop can be off. Chapter 2 covers their network settings.

17.6 Long-running sessions

Long sessions drift. Keep them on track:

- **Write the goal down.** Put the task, the definition of done, and the boundaries in a file (`TASK.md` or your `CLAUDE.md` / `AGENTS.md`) the agent re-reads.
- **Ask for checkpoints.** “Commit after each working step” gives you restore points.
- **Prefer several short runs** over one marathon; chain them with `--resume`.
- **Let it continue after usage limits.** Claude Code can resume automatically when a `claude.ai` usage limit resets (the `autoContinueAtUsageLimit` setting).

TASK.MD

Goal: migrate all API routes from Express to Fastify.

Done when: npm test passes and no file imports "express".

Rules: work on branch agent/fastify; commit after each route; never touch the deploy/ folder; if blocked for more than 3 attempts, write BLOCKED.md explaining why and stop.

- ☐ Long runs happen in tmux, `--bg`, CI, or a vendor cloud
- ☐ Scheduled runs use a wrapper with timeout, turn, and budget limits
- ☐ Each run works on its own branch and writes a log
- ☐ The task file defines “done” and when to stop

Safety Nets

Containment limits what can go wrong. Safety nets make what *does* go wrong easy to undo. With good nets you can say “yes” to more autonomy, because every mistake is a few clicks from reversed.

IN THIS CHAPTER

- Git as an undo button: commits, branches, and checkpoints
- Built-in checkpoints in each tool
- Backups for everything outside Git
- Kill switches: how to stop an agent fast
- Budget limits as a last line

18.1 Git is your undo button

Git records every saved version (a **commit**) of your project. If an agent breaks something, you return to a good commit.

See what changed: `git status` and `git diff`

Undo uncommitted changes to one file: `git restore path/to/file`

Save a restore point before a run: `git commit -am "before agent run"`

Find an old good state: `git log --oneline`

Go back on a new branch: `git switch -c rescue <commit-id>`

Recover “lost” commits: `git reflog`

Habits that make Git a reliable net:

- **Agents never work on `main`**. Give each run its own branch (`agent/...`).
- **Commit often**. Ask agents to commit after each working step.
- **Push often**. A pushed branch survives the box being destroyed.
- **Protect `main`** with status checks and no force pushes (Chapter 12).

18.2 Checkpoints built into the tools

TOOL	CHECKPOINT FEATURE
Claude Code	Rewind to an earlier point in the session (press <code>Esc</code> twice, or <code>/rewind</code>)
Gemini CLI	<code>general.checkpointing.enabled</code> saves a snapshot before changes
Cline	Checkpoints in the task history
Aider	A git commit for every change (<code>/undo</code> reverts the last one)
Cursor, Devin Desktop	Restore points in the chat timeline

NOTE

Session checkpoints usually cover file edits made by the agent's edit tools, not everything a shell command did (a database change, a deleted folder outside the project). Git and backups cover the rest.

18.3 Backups for everything else

- ☐ Your computer: Time Machine (Mac) or File History (Windows) on
- ☐ Cloud drives: version history enabled (Google Drive, Dropbox, OneDrive keep old versions)
- ☐ Databases the agent can write: automatic daily backups or snapshots
- ☐ Notion: page version history; export the workspace now and then
- ☐ n8n: back up the database volume; export workflows
- ☐ VM or container: a snapshot before big runs

18.4 Kill switches

Know how to stop each agent *before* you need to.

TO STOP	DO THIS
An interactive terminal agent	<code>Esc</code> to interrupt the turn, <code>Ctrl + C</code> to quit
A Claude Code background agent	<code>claude agents</code> , then stop it
A tmux session	<code>tmux kill-session -t agent</code>
A runaway process	<code>pkill -f claude</code> (or <code>codex</code> , <code>gemini</code> , <code>agent</code>)
A container	<code>docker stop <name></code>
A GitHub Actions run	Actions tab, the run, then Cancel workflow
A Copilot cloud agent task	Stop it from the pull request or the agents page
A leaked or abused token	Revoke it on the service's token page, immediately
Spending	Lower the spend limit to zero, or disable the API key

☆ IMPORTANT

The fastest universal kill switch is **revoking the agent's tokens**. An agent without its API key can't think, and one without its GitHub token can't push. That's another reason agents should have their own keys: you can cut them off without locking yourself out.

18.5 Budgets as the last net

Spend limits (Chapter 3) stop a runaway loop from becoming a big bill. Per-run caps like `--max-budget-usd` and `--max-turns` stop a single run; account-level limits stop everything.

- ☐ Agents always work on branches; `main` is protected
- ☐ Commits and pushes happen often
- ☐ Backups cover files, drives, and databases the agent can reach
- ☐ I know the kill switch for every agent I run
- ☐ Spend limits are set and I've tested that revoking a key stops the agent

Monitoring Agents

Hands-off doesn't mean eyes-off. A little monitoring tells you when an agent finished, when it got stuck on one of the prompts that never goes away, and what it actually did.

IN THIS CHAPTER

- Notifications when an agent needs you or finishes
- Logs and transcripts for each tool
- OpenTelemetry for dashboards
- A daily review routine

19.1 Get told when an agent needs you

Claude Code: Notification and Stop hooks

The **Notification** hook fires on events such as `permission_prompt` (a prompt has waited about six seconds) and `idle_prompt` (Claude finished about 60 seconds ago and you haven't typed). The **Stop** hook fires when Claude finishes responding.

`~/CLAUDE/SETTINGS.JSON (EXCERPT)`

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "permission_prompt",
        "hooks": [
          { "type": "command", "command": "~/bin/ping-phone.sh 'Needs you'" }
        ]
      }
    ],
    "Stop": [
      {
        "hooks": [
          { "type": "command", "command": "~/bin/ping-phone.sh 'Done'" }
        ]
      }
    ]
  }
}
```

The script can send anything you like. A simple version posts to a free push service such as ntfy:

`~/BIN/PING-PHONE.SH`

```
#!/usr/bin/env bash
curl -s -d "$1" https://ntfy.sh/my-secret-topic-name > /dev/null
```

TIP

To run a hook the instant Claude asks for permission (rather than after six seconds), use the **PermissionRequest** hook instead.

Other tools

TOOL	NOTIFICATION OPTION
Codex CLI	<code>notify = ["program", "args"]</code> runs on each completed turn; <code>tui.notifications</code> for terminal alerts
ChatGPT desktop (Codex)	Settings: turn-completion alerts never, in background, or always
Cline	"Enable notifications" in Auto Approve
Aider	<code>--notifications</code> (and <code>--notifications-command</code>)
Devin Desktop	Agent Command Center shows local and cloud agents in one view
Cloud agents	Pull requests, Slack, and email from the vendor

19.2 Logs and transcripts

TOOL	WHERE THE RECORD IS
Claude Code	Session transcripts (<code>.jsonl</code>) under <code>~/.claude/projects/</code> ; <code>--output-format json</code> in scripts
Codex	Session rollout files under <code>~/.codex/</code> ; <code>codex exec --json</code> events
Copilot CLI	<code>--share=FILE</code> or <code>--share-gist</code> after a programmatic run
OpenHands	<code>--headless --json > run.jsonl</code>
GitHub Actions	The workflow run log

Keep the JSON output of every unattended run. It's your audit trail, and it lets you ask an agent later: "read last night's log and tell me what went wrong."

19.3 Dashboards with OpenTelemetry

OpenTelemetry (OTel) is a standard format for sending usage metrics and event logs to monitoring tools such as Grafana, Datadog, or Honeycomb. Claude Code supports it:

TERMINAL

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_LOGS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
```

Set `OTEL_METRICS_EXPORTER=console` to see the data in your terminal first. Team and Enterprise admins also get usage analytics in the admin console, and API users see spend on the Claude Console usage page.

19.4 A five-minute daily review

REVIEW LAST NIGHT'S AGENTS

- 1 **Check notifications** for anything that got stuck.
- 2 **Open the pull requests** agents created; read the summaries and diffs.
- 3 **Skim the logs** for denied actions and repeated failures.
- 4 **Check spend** against your limits.
- 5 **Adjust**
add an allow rule for a repeated harmless prompt, or a deny rule for anything that worried you.

- ☐ I get a phone or desktop alert when an agent needs me or finishes
- ☐ Every unattended run keeps a JSON log
- ☐ I review agent pull requests and spend daily

Multi-Agent Setups

Once one agent runs hands-off, the obvious next step is several at once: one fixing bugs, one writing tests, one upgrading dependencies. The challenge is keeping them from tripping over each other.

IN THIS CHAPTER

- Git worktrees: one folder per agent
- Subagents and agent teams inside one tool
- Cloud fleets
- Rules that keep parallel agents from colliding

20.1 One worktree per agent

A **git worktree** is an extra working folder for the same repository, checked out on its own branch. Agents in different worktrees can edit freely without overwriting each other.

TERMINAL

```
git worktree add ../my-app-tests -b agent/tests
git worktree add ../my-app-deps -b agent/deps
git worktree list
git worktree remove ../my-app-tests # when done
```

Several tools create worktrees for you:

TOOL	COMMAND
Claude Code	<code>claude -w <name></code> (folder <code>.claude/worktrees/<name></code>), add <code>--tmux</code>
Codex	<code>codex exec --worktree "..."</code>
Cursor CLI	<code>agent -w <name></code> (under <code>~/.cursor/worktrees/</code>)

20.2 A tmux layout for three agents

TERMINAL

```
tmux new -s fleet -d
tmux send-keys -t fleet "cd ../my-app-tests" C-m
tmux send-keys -t fleet "claude --bg 'Raise test coverage'" C-m
tmux split-window -t fleet
tmux send-keys -t fleet "cd ../my-app-deps && codex --yoLo" C-m
tmux attach -t fleet
```

Each pane is a separate agent in its own worktree, each in its own box if you're in bypass.

20.3 Subagents and agent teams

A **subagent** is a helper agent that the main agent starts for one part of a job, with its own instructions and tools. In Claude Code, subagents inherit the parent's permission mode, so a parent in bypass gives its subagents full access too. Claude Code's **agent teams** go further: several sessions coordinate and message each other. When teammates message each other, the auto-mode classifier reviews each message in auto mode, and cross-session messaging safeguards still apply in bypass.

Devin CLI and Cline also offer subagents and agent teams; Copilot CLI has a `--fleet` option for parallel work.

20.4 Cloud fleets

The easiest parallelism is in the cloud, where each task gets its own VM: Claude Code cloud sessions and routines, Codex Cloud, Copilot cloud agent (assign several issues), Cursor Cloud Agents ("as many as you want in parallel"), and Devin. Each produces a pull request, and your branch protection decides what lands.

20.5 Rules that prevent collisions

- ☐ One agent, one branch, one worktree
- ☐ Split work by folder or feature, not by "whoever gets there first"
- ☐ Each agent writes its plan to a file the others can read
- ☐ Only humans (or auto-merge with green checks) merge to `main`
- ☐ Shared resources (databases, ports, cloud accounts) are separate per agent
- ☐ A total spend limit covers the whole fleet

 WARNING

Costs multiply with agents. Three agents in parallel use roughly three times the tokens. Set the account-level spend limit with the whole fleet in mind.

The Master Comparison

One place to compare every tool in this guide. Use it to pick tools, or as a reminder of where a setting lives.

21.1 Where the settings live

TOOL	USER CONFIG	PROJECT CONFIG
Claude Code	<code>~/.claude/settings.json</code>	<code>.claude/settings.json</code> , <code>.claude/settings.local.json</code>
Codex	<code>~/.codex/config.toml</code>	<code>.codex/</code> (trusted projects only)
Gemini CLI	<code>~/.gemini/settings.json</code> , <code>~/.gemini/policies/</code>	<code>.gemini/settings.json</code>
Copilot CLI	<code>~/.copilot/settings.json</code>	Repository settings
VS Code	User <code>settings.json</code>	<code>.vscode/settings.json</code>
Cursor editor	<code>~/.cursor/permissions.json</code> , <code>sandbox.json</code>	<code>.cursor/permissions.json</code>
Cursor CLI	<code>~/.cursor/cli-config.json</code>	<code>.cursor/cli.json</code>
Devin CLI	<code>~/.config/devin/config.json</code>	<code>.devin/config.json</code>
Aider	<code>~/.aider.conf.yml</code>	<code>.aider.conf.yml</code>
Goose	<code>goose configure</code>	—

21.2 Levels, side by side

TOOL	LEVEL 3 (REVIEWED)	LEVEL 4 (BYPASS)
Claude Code	<code>--permission-mode auto</code>	<code>--dangerously-skip-permissions</code>
Codex	<code>--approve-for-me</code>	<code>--yolo</code>
Gemini CLI	<code>--approval-mode=auto_edit</code> (edits only)	<code>--approval-mode=yolo</code>

Copilot CLI	<code>/permissions assisted</code>	<code>--yolo</code>
VS Code Copilot	Assisted permissions	Allow all / Autopilot
Cursor editor	Auto-review	Run Everything
Cursor CLI	Allow tokens	<code>--force</code>
Devin Desktop	Auto	Turbo
Devin CLI	Smart	Bypass
Cline	Auto Approve toggles	YOLO Mode
OpenHands	<code>--llm-approve</code>	<code>--always-approve</code>
Goose	<code>smart_approve</code>	<code>auto</code> (default)
Aider	—	<code>--yes-always</code>

21.3 Headless commands

TOOL	ONE-SHOT COMMAND	JSON OUTPUT
Claude Code	<code>claude -p "..."</code>	<code>--output-format json</code>
Codex	<code>codex exec "..."</code>	<code>--json</code>
Gemini CLI	<code>gemini -p "..."</code>	<code>--output-format json</code>
Copilot CLI	<code>copilot -p "..."</code>	<code>-s</code> (answer only)
Cursor CLI	<code>agent -p "..."</code>	<code>--output-format json</code>
Devin CLI	<code>devin -p "..."</code>	—
Cline CLI	<code>cline "..."</code>	<code>--json</code>
OpenHands	<code>openhands --headless -t "..."</code>	<code>--json</code>
Goose	<code>goose run -t "..."</code>	—
Aider	<code>aider --message "..."</code>	—

21.4 Built-in sandboxes and clouds

TOOL	LOCAL SANDBOX	CLOUD OPTION
Claude Code	Bash sandbox, sandbox runtime	Cloud sessions, routines
Codex	<code>workspace-write</code> (Seatbelt / Linux)	Codex Cloud
Gemini CLI	<code>--sandbox</code> (Seatbelt, Docker, Podman)	GitHub Action
Copilot	VS Code sandbox	Cloud agent
Cursor	Auto-review sandbox	Cloud Agents
Devin	<code>--sandbox</code> (Autonomous mode)	Devin cloud
OpenHands	Docker sandbox	OpenHands cloud

21.5 What can't be turned off

TOOL	SURVIVES EVERY SETTING
Claude Code	Critical-path deletes, ask rules, AskUserQuestion , org policies
Codex	Safety monitoring pauses, managed requirements
Gemini CLI	Admin secure mode, policy denies
Copilot	Org policies, disableBypassPermissionsMode
Cursor	Team policies, built-in security rules
Devin	Org deny/ask rules, admin max level
Browsing agents	Purchases, sends, CAPTCHAs

Copy-Paste Config Files

Every file in this chapter is a complete starting point for maximum autonomy. Copy it, adjust paths and names, and save. Each assumes the agent runs inside a box from Chapter 2.

22.1 Claude Code

~/CLAUDE/SETTINGS.JSON

```
{
  "permissions": {
    "defaultMode": "bypassPermissions",
    "deny": ["Bash(git push --force *)", "Read(~/.ssh/**)"]
  },
  "skipDangerousModePermissionPrompt": true,
  "enableAllProjectMcpServers": true,
  "sandbox": { "enabled": false },
  "env": { "CLAUDE_CODE_DISABLE_DANGEROUS_RM_TIMEOUT": "1" }
}
```

For your laptop, swap `bypassPermissions` for `auto` and set `"sandbox": { "enabled": true, "failIfUnavailable": true }`.

22.2 Codex

~/CODEX/CONFIG.TOML

```
approval_policy = "never"
sandbox_mode = "danger-full-access"
web_search = "live"
notify = ["/home/me/bin/ping-phone.sh", "Codex done"]

[projects."/work"]
trust_level = "trusted"
```

22.3 Gemini CLI

~/.GEMINI/SETTINGS.JSON

```
{
  "general": {
    "defaultApprovalMode": "auto_edit",
    "checkpointing": { "enabled": true }
  },
  "security": { "enablePermanentToolApproval": true },
  "tools": { "sandbox": "docker" }
}
```

~/.ZSHRC

```
alias gem='gemini --approval-mode=yolo --sandbox'
```

22.4 VS Code and Copilot

VS CODE SETTINGS.JSON

```
{
  "chat.tools.global.autoApprove": true,
  "chat.autopilot.advanced.enabled": true,
  "claudeCode.initialPermissionMode": "bypassPermissions"
}
```

~/.ZSHRC

```
alias cop='copilot --yolo --no-ask-user'
```

22.5 Cursor CLI

~/.CURSOR/CLI-CONFIG.JSON

```
{
  "permissions": {
    "allow": [
      "Shell(*)", "Read(**)", "Write(**)",
      "WebFetch(*)", "Mcp(*:*)"
    ],
    "deny": ["Write(**/.env*)"]
  }
}
```

22.6 Devin CLI

~/CONFIG/DEVIN/CONFIG.JSON

```
{
  "permissions": {
    "allow": ["Read(**)", "Exec(git)", "Exec(npm)", "Exec(pnpm)"],
    "deny": ["Exec(sudo)"]
  }
}
```

~/ZSHRC

```
alias dv='devin --permission-mode bypass'
```

22.7 Aider

~/AIDER.CONF.YML

```
yes-always: true
auto-commits: true
auto-test: true
test-cmd: npm test
```

22.8 Goose

~/ZSHRC

```
export GOOSE_MODE=auto
export GOOSE_MAX_TURNS=200
```

22.9 A dev container for any agent

.DEVCONTAINER/DEVCONTAINER.JSON

```
{
  "image": "mcr.microsoft.com/devcontainers/javascript-node:22",
  "remoteUser": "node",
  "features": {
    "ghcr.io/anthropics/devcontainer-features/claude-code:1.0": {}
  },
  "mounts": [
    "source=agent-home-${devcontainerId},target=/home/node/.claude,type=volume"
  ],
  "containerEnv": {
    "CLAUDE_CONFIG_DIR": "/home/node/.claude",
    "DISABLE_AUTOUPDATER": "1"
  },
  "postCreateCommand": "npm install -g @openai/codex @google/gemini-cli"
}
```

TIP

Commit the dev container to your repository, but never commit keys. Pass them in from your host environment or as Codespaces secrets.

Troubleshooting

When an agent that should be hands-off keeps stopping, find the symptom here.

23.1 Claude Code

SYMPTOM	LIKELY CAUSE	FIX
<code>defaultMode: auto</code> ignored, starts in Manual	Set in a project file	Move it to <code>~/.claude/settings.json</code>
"cannot be used with root/sudo privileges"	Running as root	Use a non-root user or dev container
Auto mode "unavailable"	Unsupported model, or <code>disableAutoMode</code>	Switch model; check managed settings
Auto mode suddenly asks every time	3-in-a-row or 20-total blocks	Approve once; add <code>autoMode</code> rules
<code>rm -rf</code> of a folder asks even in bypass	Critical path	Delete a literal subfolder instead
Edit to <code>.git</code> or <code>.claude</code> asks	Protected path	Bypass mode, or approve for the session
<code>--bg</code> bypass ignored	Acceptance not in user settings	Add <code>skipDangerousModePermissionPrompt</code> to <code>~/.claude/settings.json</code>
<code>mcp_*</code> allow rule does nothing	Allow globs must name a server	Use <code>mcp__github__*</code>
Project allow rules ignored in <code>-p</code>	Folder not trusted	Open it once interactively and trust it
CI run hangs	A prompt with nobody to answer	<code>--permission-prompts none</code> or <code>dontAsk</code>
Cloud session ignores bypass	Not offered in the cloud	Use Auto

23.2 Codex

SYMPTOM	LIKELY CAUSE	FIX
"unknown flag --full-auto"	Removed	<code>--sandbox workspace-write</code> or <code>--approve-for-me</code>
<code>npm install</code> fails	No network in <code>workspace-write</code>	<code>network_access = true</code>
Asks before every MCP tool	Server approval mode	<code>default_tools_approval_mode = "approve"</code>
Starts read-only	Folder not trusted	<code>trust_level = "trusted"</code>
Can't write <code>.git</code>	Protected path	<code>danger-full-access</code>
<code>codex exec</code> won't run	Not a git repo	<code>--skip-git-repo-check</code>
Task paused	Safety monitoring	Review findings; resume if offered
Cloud task can't download	Domain not allowed	Add it under Additional allowed domains

23.3 Gemini CLI, Copilot, Cursor, and others

SYMPTOM	LIKELY CAUSE	FIX
Gemini YOLO refused	<code>disableYoloMode</code> or admin secure mode	Change setting or ask admin
YOLO in <code>settings.json</code> ignored	Only allowed on the command line	Use an alias
Gemini ignores project settings	Folder untrusted	Trust it, or <code>--skip-trust</code>
Copilot CLI <code>-p</code> asks for tools	Programmatic mode needs it	<code>--allow-all-tools</code>
Copilot <code>--yolo</code> ignored	Admin policy	<code>disableBypassPermissionsMode</code> is set
Copilot cloud agent blocked from a site	Firewall	Add to the custom allowlist
Cursor CLI proposes but doesn't edit	No <code>--force</code>	Add <code>--force</code>
Cursor Auto-review grayed out	Classifier model not allowed	Admin enables it in Team Settings > Models
Devin Desktop has no Turbo	Admin capped the level	Ask admin
Cline still asks for "all" commands	Base toggle off	Enable both toggles, or YOLO Mode
Roo Code stopped working	Shut down	Move to Cline or another tool

23.4 General

SYMPTOM	LIKELY CAUSE	FIX
Push to workflow files rejected	Token lacks workflow scope	<code>gh auth refresh -s workflow</code>
Push blocked "secret detected"	Push protection	Remove the secret; rotate it
Agent can't open PRs in Actions	Workflow permissions	Read and write; allow PR creation
Cron job does nothing	Empty environment	Full paths; load keys in the script
Agent stops at a login page	Not signed in	Sign in first in the agent's profile
Costs spike overnight	Loop or fleet	Per-run caps and account spend limit

What Always Asks: The Master List

These are the prompts and blocks that survive every setting in this guide. They're worth knowing so you're not surprised when a "fully autonomous" agent pauses.

TOOL	ALWAYS ASKS OR BLOCKS
Claude Code	Critical-path <code>rm</code> / <code>rmdir</code> (root, home, top-level dirs, working dir); explicit ask rules; <code>AskUserQuestion</code> ; <code>requiresUserInteraction</code> MCP tools; org "ask" connector policies; PowerShell <code>Remove-Item</code> on system paths; running bypass as root outside a sandbox
Claude Desktop / Chrome	Purchases, account creation, CAPTCHAs; classifier-flagged actions on external sites; deleting sessions
Cowork	Write connector tools when the admin hasn't allowed "Always allow"
Claude cloud sessions	No bypass; Auto is the ceiling
Cursor	Team admin policies; hard-coded security rules; (outside Run Everything) protected paths and commands leaving the sandbox
VS Code / Copilot	Tools set to <code>false</code> ; organization policies; enabled sandbox limits
Codex	Safety-monitoring pauses; managed requirements; protected <code>.git</code> , <code>.codex</code> , <code>.agents</code> , <code>.aws</code> outside full access
Gemini CLI	Admin secure mode or <code>disableYoLoMode</code> ; policy deny rules; untrusted folders
GitHub	Sudo mode in the web UI; push protection for secrets; org SSO and policies
n8n	Workflows not enabled for MCP; Execute Command unavailable on Cloud
Notion	Pages not shared with a connection; owner-level MCP restrictions
ChatGPT agent	Confirmations before high-impact actions (purchases, sending); "watch mode" on sensitive sites

Devin Desktop / CLI	Admin maximum level and team deny lists; org deny and ask rules
Cline	Nothing beyond what you leave unticked, once YOLO Mode is on
OpenHands headless	Nothing: always-approve can't be turned off there

 TIP

When a “fully autonomous” agent pauses, check this table first. If the pause is on the list, plan around it (leave that step for yourself, or rewrite the command). If it isn’t, it’s a setting you can change; the tool’s chapter shows where.

Your Autonomy Setup Checklist

- ☐ Agents run in a dev container, VM, cloud box, or separate user account
- ☐ Separate GitHub account or app for agents; fine-grained token with full repo permissions
- ☐ Hard spend limits set on every AI and cloud account
- ☐ Everything in Git; `main` protected by status checks; auto-merge allowed
- ☐ `~/.claude/settings.json`: `defaultMode` set to `auto` or `bypassPermissions`, plus `skipDangerousModePermissionPrompt`
- ☐ Claude Desktop: **Allow bypass permissions mode** on; Cowork: **Automatically approve**
- ☐ Every connector tool set to **Always allow**
- ☐ Cursor: **Run Everything** (or Auto-review with allow instructions)
- ☐ VS Code: `chat.tools.global.autoApprove` on, or Autopilot
- ☐ Codex `config.toml`: `approval_policy = "never"`, `sandbox_mode = "danger-full-access"`; Gemini alias with `--approval-mode=yolo`
- ☐ n8n: API key, instance-level MCP on, workflows enabled for MCP; `NODES_EXCLUDE="[]"` if self-hosted
- ☐ Notion: MCP connected; API connection with all capabilities and top-level pages shared
- ☐ Unattended runs have `--max-turns` and `--max-budget-usd` limits
- ☐ Cursor CLI, Copilot CLI, Devin CLI, and Aider aliases set for hands-off runs
- ☐ Notifications reach my phone when an agent needs me or finishes
- ☐ I know every kill switch, and revoking a key stops the agent

Glossary

ACU	Agent Compute Unit, the usage unit Devin bills in.
Agent	An AI model that takes actions, such as editing files and running commands, not just answering.
Agent SDK	Anthropic's library for building your own programs on Claude Code's engine.
Allowlist	A list of pre-approved actions an agent may take without asking.
API key	A secret string that lets a program use a service's API and bills to your account.
Approval policy	Codex's setting for when it pauses to ask (<code>on-request</code> or <code>never</code>).
Approve for me	Codex's mode in which an automatic reviewer answers approval requests.
Auto mode	Claude Code's permission mode in which a classifier reviews actions instead of a person.
Auto-review	Cursor's Run Mode in which a classifier checks calls that aren't allowlisted.
Autopilot	A mode in VS Code and Copilot CLI that keeps working until the task is complete.
Background agent	An agent session that keeps running after you close the terminal (<code>claude --bg</code>).
Bypass permissions	A mode that skips approval prompts entirely (<code>--dangerously-skip-permissions</code> in Claude Code).
Checkpoint	A saved snapshot you can roll back to after an agent's changes.
Classifier	A second AI model that judges whether an agent's action is safe and matches the request.
Cloud agent	An agent that runs on the vendor's machines instead of your computer.

Containment	Limiting what an agent can reach (a container, separate accounts, budgets) instead of limiting what it may do.
Critical path	A folder, such as the root, your home folder, or the working directory, that Claude Code protects from deletion in every mode.
Cron	The Linux and macOS scheduler that runs commands on a timetable.
Deny rule	A rule that blocks a specific action, even in bypass mode.
Dev container	A Docker container configured as a development environment, used to isolate agents.
Docker	Software that runs disposable, isolated containers.
Environment variable	A named value your shell passes to programs, often used for tokens.
Fine-grained token	A GitHub access token limited to chosen repositories and permissions.
Folder trust	A one-time question about whether a tool may fully run in a folder.
GitHub Action	A workflow that runs automatically on GitHub's servers when something happens in a repository.
GitHub App	An integration installed on chosen repositories that acts under its own name with short-lived tokens.
Headless / print mode	Running an agent non-interactively from a script, such as <code>claude -p</code> .
Hook	A command an agent tool runs automatically at a set moment, such as before each tool call.
Kill switch	A fast way to stop an agent, such as revoking its token.
Machine user	A separate GitHub account used only for automation.
MCP	Model Context Protocol, a standard way for agents to use outside tools.
MicroVM	A small, fast-starting virtual machine.
Network allowlist	A list of websites an agent's commands may reach; all others are blocked.
Non-root user	An ordinary account without administrator powers.
OpenTelemetry	A standard format for sending usage metrics and logs to monitoring tools.
Permission mode	A preset that decides which actions an agent may take without asking.
Permission prompt	A pause in which the agent asks you to approve an action.
Policy engine	Gemini CLI's system of TOML rule files that allow, deny, or ask about tools.

Profile (Codex)	An extra config file layered on top of <code>config.toml</code> with <code>--profile</code> .
Prompt injection	Instructions hidden in content an agent reads, meant to hijack its behavior.
Protected path	A file or folder, such as <code>.git</code> , that tools guard from automatic writes.
Routine	A saved Claude Code task that runs in Anthropic's cloud on a schedule, an API call, or a GitHub event.
Ruleset	GitHub settings that protect branches, with an optional bypass list.
Sandbox	An isolated environment limiting the files and network a command can reach.
Sandbox runtime	Anthropic's package that wraps a whole process, such as Claude Code, in OS isolation.
Scope	A permission attached to a token, such as "write to repositories".
Spend limit	A hard cap on how much an account can spend.
Subagent	A helper agent started by the main agent for part of a task.
Sudo mode	GitHub's re-authentication step for sensitive actions in the web interface.
tmux	A terminal program that keeps sessions running after you disconnect.
Token	A secret string that lets a program act as an account.
Turbo mode	Devin Desktop's auto-execution level that runs every command except deny-listed ones.
Virtual machine	A whole simulated computer with its own operating system.
Worktree	An extra working folder for the same git repository, on its own branch.
YOLO mode	Informal name for fully automatic approval in tools like Gemini CLI, Codex, and Copilot CLI.