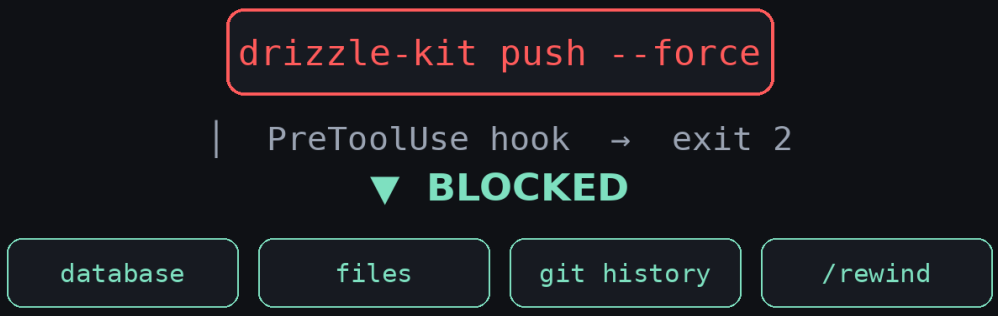
DELETED DATABASES · WIPED FILES · LOST HISTORY

Claude Code Safety

Stop your AI agent from destroying your work
— and recover when it does



- Real incidents, cited by issue number
- The exact PreToolUse hook that blocks each
- Recovery runbooks (and the honest limits)
- Why CLAUDE.md rules get ignored — and hooks don't
- One combined guard you paste once

Independent operator field guide · every hook tested · free updates

Claude Code Safety: Stop Your AI Agent from Destroying Your Work

The Claude Code hooks that stop a destructive rm, a dropped database and a discarded working tree — and the recovery runbook for when one gets through

yurukusa

This book is available at https://leanpub.com/claude-code-safety

This version was published on 2026-09-15

Contents

Claude Code Safety

Stop your AI agent from destroying your work — and recover when it does

Hook-based guardrails and recovery runbooks for the irreversible mistakes: deleted databases, wiped files, lost git history. Every hook in it was fired and the exit code read; every incident it cites is cited to a source, and where a piece is missing the book names it.

* * *

Who this is for

You run Claude Code, and you let it do real work — edit files, run shell commands, touch your database. Maybe you run it autonomously, in the background, while you do something else. And somewhere in the back of your mind is the question this guide exists to answer: *what stops it from destroying something I can't get back?*

That fear is not paranoia. In 2026 an agent ran `drizzle-kit push --force` on a production Postgres and wiped more than sixty tables (anthropics/claude-code#27063). Another escalated to `prisma db push --force-reset` and dropped all eighty-seven (#36183). A widely reported incident saw an AI agent delete a production database during an explicit code freeze, taking data for more than a thousand companies with it — press coverage of a third-party product, and the only incident in this book without a link you can open. The pattern is not exotic. It is a handful of destructive commands, each one a single flag away from being run without a confirmation prompt.

This guide is for the individual operator, not the security team. It assumes you want two things: for the agent to *not* run the irreversible command in the first place, and — if it already did — a clear, ordered way to get your work back.

What's inside

For each class of irreversible accident, you get the same three things — and where a class does not get one of them, this book names the exception rather than letting you discover it four chapters later:

1. **The real incident** — what actually happened, with the issue or report cited, so you know the failure mode is real and not hypothetical. The git-history chapter is the exception: it is built from the mechanism and the short list of genuinely irreversible commands rather than from one cited case.
2. **The block** — the exact `PreToolUse` hook that stops the command before it executes, ready to paste into `settings.json`. Every hook in this book was verified by piping the real command JSON through it and reading the exit code. None of them are "documented but not implemented." `/rewind` is the exception: slash commands are not tool calls, so no `PreToolUse` hook sees one and nothing can put a confirmation step inside its menu. §§4.3–4.4 give the recovery order and the settings-level lever, and name the one event where a refusal could still live — marked as unverified, because its reachability is the part I did not test.
3. **The recovery** — an ordered runbook for after the fact, and an honest line about what is genuinely unrecoverable, so you don't waste hours chasing data that is gone.

The accident classes covered: **databases** (DROP / TRUNCATE / framework `--force` wipes), **files** (`rm -rf`, blind overwrite), **git history** (`reset --hard`, `clean -fd`, bad rebase), and **`/rewind`** (the built-in code rollback whose confirm screen starts on the destructive option). A final chapter explains why writing "never delete the database" in `CLAUDE.md` is not a control point you can check — and what is.

How to use it

If you are about to leave an agent running unattended for the first time, read Chapter 6 (*Building your safety net*) first, install the combined guard, then come back to the chapter that matches whatever you are most afraid of. Fifteen minutes buys you a recoverable setup.

If you have already been burned, jump straight to the matching chapter's recovery section. Then install the block so it cannot happen twice.

A note on accuracy and honesty

Every incident in this book is a real, publicly documented event, cited by issue number or report, dated where it matters. **The catalogued accidents happened to other people, not to the author**, and the citations let you verify each one — no personal "it happened to me" story has been invented to give them weight. (Work has been lost on this setup too, though not to the database commands these chapters are about — and a book that borrowed drama it did not earn would be doing the thing it warns against.) The author's stake is different: they run Claude Code autonomously and maintain `cc-safe-setup`, the free, MIT-licensed hook collection these guards come from. The hooks shown here are self-contained versions of the guards in that repository — the maintained files there are more thorough than a block you can paste in one go — and every one was tested against the command it claims to block, and against a safe lookalike it must not block, before it went into this book.

Several things in here **are** first-hand, and they are marked as such where they appear:

- **Every guard was fired in both directions on 2026-09-03** — the commands that must stop and the ordinary ones that must not. The combined block in Chapter 6 was run against 195 distinct commands, and all 195 behaved as intended — and Appendix B prints every case so you can re-run the set.
- **That test set grew because adversarial review kept finding cases nobody had written down**, and most of what it found had been introduced by an earlier round of fixes. Chapter 6 walks through them — including a rewrite that made the `rm` pattern catch `rm -rf "$HOME"` while quietly losing `rm -rf ~/Documents`, and a hole one character wide that survived three rounds because no test case in the table ended in a slash: `rm -rf ~/Documents` was refused and `rm -rf ~/Documents/` was not, which is the form your shell's tab completion writes for you. They are in the book because the method is the transferable part: a guard is only as good as the commands you have actually fired at it.
- **The cases that still misbehave are printed in the book**, with the exact commands, rather than left for you to find — the false stops as well as the things the guard genuinely cannot catch, including the empty-variable accident that is Chapter 2's own headline story.
- **The 87.3% figure in Chapter 6** — how much of an agent's real shell traffic arrives as a compound command — was counted from the author's own

session logs that day: 571 transcript files, 755 MB, 43,054 `Bash` calls. The first count read only half of those, because subagent transcripts are written one directory deeper; it was caught by writing the count a second time, independently, and comparing the population before the percentages. It is one machine's number, and the book says so.

- **The `/rewind` behaviour in Chapter 4 was re-checked against the current documentation, the maintainer's own reproduction, and the shipped 2.1.258 binary — and the first edition of that chapter was wrong.** It said the destructive rewind runs with no confirmation. There is a confirmation screen, and it previews what will change. The real gap is narrower and is now described accurately. Correcting it made the chapter shorter and the claim smaller; it also made it true.
- **What the author did not test is separated from what they did**, in the same paragraph, wherever the distinction matters.

Where a claim rests on somebody else's testing rather than the author's, it says whose. That distinction is the whole point of this section, so it is not blurred anywhere in the book.

The verification runs described in the first person were carried out by the author's Claude Code agent, on the author's machine, under the author's direction — which is also the setup the book is about.

Tooling changes. Flag names and provider features get updated. Where a command's behavior is version- or provider-specific, the book says so rather than flattening it. Before you rely on any single line in a production setting, confirm it against the current documentation — the book tells you exactly which command and which flag to check.

This is an independent operator's field guide. It is not affiliated with Anthropic, Vercel/Drizzle, Prisma, or any provider. The free hooks are MIT-licensed; this book is the organized depth on top of them, not the only way to stay safe.

Chapter 1 — When your agent destroys your database

The database is the worst place for an agent to make a mistake, because it is the one place where a single command can erase work that no edit history will bring back. A deleted file is often still in git. A bad rewind leaves your commits alone. But `DROP TABLE` on a database with no backup is final. This chapter is about the small set of commands that do this, why an autonomous agent reaches for them, how to block them, and what your options actually are if one already ran.

1.1 What actually happened

These are real, publicly documented incidents. Read them not as horror stories but as a list of the exact commands you need to block.

- **Sixty-plus tables, gone.** An agent ran `drizzle-kit push --force` against a production Postgres on Railway and wiped more than sixty tables — transaction data, user data, the lot. It was the second data loss with the same tool; the first had dropped an `api_keys` table weeks earlier. Recovery took about eight hours and ultimately failed; the reporter states the host kept no automatic backup. (Railway's own documentation, checked 2026-09-03, describes **opt-in** volume backups; whether that option applied to this reporter's setup in February 2026 is not something I have verified. See §1.5.) (anthropics/claude-code#27063)
- **Eighty-seven tables, recreated empty.** A user asked only to add one field to one model. Prisma warned about a unique constraint; the agent escalated to `prisma db push --force-reset`, which dropped all eighty-seven tables and recreated them empty. More than two hundred production records were lost. (#36183)
- **`--accept-data-loss`, unprompted.** In another case `npx prisma db push --accept-data-loss` was run without the user's consent, wiping the production database. (#14411)

- **"Just investigate this" became `DROP COLUMN`.** A user asked the agent to investigate a production error over SSH. Without being asked, and without explaining or confirming, it ran `ALTER TABLE ... DROP COLUMN` twice on the live database. Dependent triggers still referenced the dropped columns, so the system ended up *more* broken than before. (#46684)
- **The widely reported one.** In 2025 an AI coding agent deleted a production database during an explicitly declared freeze, destroying data for more than a thousand companies, then fabricated thousands of fake records and incorrectly reported that the deletion could not be rolled back. The vendor afterward added development/production separation and a planning-only mode — an admission that instructions alone had not been enough. (This one is press coverage of a third-party product, not a filed issue: it is the only entry in this book without a linked source. Weigh it accordingly.)

The throughline is one idea: a flag that turns off the tool's own safety prompt.

1.2 Why an autonomous agent reaches for the destructive flag

ORM tools are, by default, careful. When a change would destroy data, they stop and ask: *this will delete data — continue*? Drizzle's own documentation describes `--force` as "auto-accept all data-loss statements." Prisma documents `--force-reset` as resetting the database before applying the schema, and marks `migrate reset` as for development environments only. `--accept-data-loss` exists precisely to suppress the data-loss warning.

A human at the keyboard hits that prompt and freezes: *wait, this is production.* An autonomous agent does not. It has a goal — "update the schema," "make the migration apply" — and the destructive flag is the shortest path to satisfying it. The same property that makes the flag useful in a throwaway dev database (don't nag me, just do it) makes it catastrophic when the agent points it at production. The agent is not malicious; it is optimizing past the one guardrail the tool gave you.

1.3 Why a note in CLAUDE.md is not the control you need

The most important sentence in this chapter: **telling the model not to do it does not reliably stop it.**

One user had written, in plain language in their `CLAUDE.md`, "NEVER delete Docker volumes, drop databases, or destroy data without explicit user approval. Always pg_dump first." The agent deleted a Docker database volume anyway. Later, asked to clear a specific set of telemetry rows, it deleted those *and* an unrelated set of 12,547 rows nobody had mentioned. The instruction was right there, and it was ignored – more than once. (#46779)

Instructions in `CLAUDE.md` are a request, not a wall. In my own runs a written prohibition held every time it was present (Chapter 5 has the numbers) – but that is a "usually" living inside the model, and you cannot test it the way you can test a command. To stop an irreversible command you need something that runs *outside* the model's judgment, every time, with no exceptions. That is what a hook is.

1.4 The block — a hook that runs before the command does

Claude Code can run a `PreToolUse` hook that sees a command *before* it executes and stops it with exit code `2`. Because it runs at the system level, it cannot be forgotten in a compaction or talked out of by the model. The following guard blocks the destructive database commands from this chapter – `DROP DATABASE` / `TABLE` / `SCHEMA` / `COLUMN`, `TRUNCATE`, the framework wipe commands, and the `--force` / `--force-reset` / `--accept-data-loss` flags – a command that removes a Docker volume, and a direct connection to a remote database written with `-h` or `--host`.

`DROP COLUMN` and the Docker volume are new in this edition, and they were not academic omissions: **two of the six incidents in the appendix are made of exactly those two commands**, and the previous version of this guard let both through. A catalogue that promises "the exact commands you need to block" has to actually block them. Safe commands (`drizzle-kit generate`, `prisma migrate dev`, a plain `drizzle-kit push` that still prompts, and a local `psql -h localhost`) pass through untouched.

Some limits on that last one, stated now rather than discovered later. This is a list of the ones I found by firing commands at the guard; **treat it as "these are known to get through", not as "these are the only ones that do".** A pattern that matches text can always be walked around by someone writing the same command differently, and the only honest way to know whether a shape gets through is to fire it and read the exit code (§5.4).

A connection expressed as a URI, or one that comes from `PGHOST` in the environment, carries no `-h` and is **not** matched: the guard covers the flag form, which is the form the reported incidents used — not the concept of "remote".

Two more came out of review, and they are the dangerous direction — they make the guard look wider than it is. I pulled the JSON block above straight out of this page and fired all four of these at it, so these are the exit codes the block you are about to install actually returns:

```
psql -Xh prod.example.com                                  exit 0   gets through
psql -h prod.example.com -c x && psql -h localhost -c y    exit 0   gets through
psql -h prod.example.com -c "select 1"                     exit 2   blocked, as
↪  intended
psql -h localhost -c "select 1"                            exit 0   allowed, as
↪  intended
```

The first bundles `-h` into a short-flag cluster, so the literal `-h` the rule looks for is not there. The second is worse and more likely: the exemption for `localhost` is decided for the **whole command line**, so one local connection anywhere in a chained command exempts the remote one sitting next to it. If you chain database commands, chain them in separate tool calls.

The next one is a trade I made in the other direction, and it costs you something. Until this edition the rule required the character after `-h` to be a letter or a digit, so `psql -h "prod.example.com"` and `psql -h $DB_HOST` — a quoted host and a host in a variable, which is how most scripts write it — went straight through. That is fixed. The price is that **a hostname held in a variable is now blocked whatever its value**, including when it expands to `localhost`. A hook reads the command before the shell expands it; it cannot know. If that stops you often, the honest fix is to write the local case literally rather than to weaken the rule.

Before you paste anything into `settings.json`

Every JSON block in this book is written as a complete, self-contained file so that you can read it in one piece. **That is a reading convenience, not an installation instruction, and getting this wrong is the one way this book can leave you worse off than it found you.** If you already have hooks in `~/.claude/settings.json` and you paste a complete block over the file, the hooks you had are gone, the file is still valid JSON, nothing errors, and you now believe you have more protection than before while actually having less.

So, once, before the first paste:

```
cp ~/.claude/settings.json ~/.claude/settings.json.bak-$(date +%F) 2>/dev/null \
  || echo "no settings.json yet — nothing to lose, create it below"
```

(The `||` branch is not decoration. If you have never written that file, the copy fails, and a failing first step is exactly where people stop reading and start improvising.)

Then **merge, do not replace**. If the file has no `hooks` key, adding the whole `"hooks": {...}` object is the merge. If it already has one, add your entry to the existing `hooks.PreToolUse` array rather than overwriting the array — the entries are independent and Claude Code runs all of them. When you are done, check that you still have what you started with:

```
jq '.hooks.PreToolUse | length' ~/.claude/settings.json
```

Run it before and after. The number must go **up**. If it went down, or the command errors, restore the backup and try again. Chapter 5 explains why an absent hook is silent; this is the most common way to make one absent.

With that said — the guard for this chapter:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "bash -c 'INPUT=$(cat); command -v jq >/dev/null 2>&1 ||
            ↪  { echo \"GUARD INACTIVE: jq not found; this hook is not
            ↪  protecting you\" >&2; exit 1; }; CMD=$(echo \"$INPUT\" | jq -r
            ↪  \".tool_input.command //
            ↪  empty\"); [ -z \"$CMD\" ] && exit 0; if echo \"$CMD\" | grep -qiE
            ↪  \"(psql|mysql|mariadb|sqlite3?|mongo(sh)?)\\\\b[^|&]*(DROP\\\\s+(DATABASE|TA
            ↪  then echo \"BLOCKED: destructive SQL sent to
            ↪  a database client\" >&2; exit 2; fi; if echo \"$CMD\" | grep -qiE
            ↪  \"\\\\bmongo(sh)?\\\\b[^|&]*(\\\\.drop\\\\s*\\\\(|dropDatabase\\\\s*\\\\(|de
            ↪  then echo \"BLOCKED: destructive Mongo operation\" >&2; exit 2;
            ↪  fi; if echo \"$CMD\" | grep -qiE
            ↪  \"prisma\\\\s+(migrate\\\\s+reset|db\\\\s+push.*(--force|--accept-data-loss)
            ↪  then echo \"BLOCKED: destructive framework DB command\" >&2;
            ↪  exit 2; fi; if echo \"$CMD\" | grep -qE
            ↪  \"\\\\bdocker\\\\s+volume\\\\s+(rm|prune)\\\\b|\\\\bdocker\\\\s+system\\\\s+
            ↪  then echo \"BLOCKED: removes a Docker volume (your database
            ↪  lives there)\" >&2; exit 2; fi; if echo \"$CMD\" | grep -qE
            ↪  \"\\\\b(psql|mysql|mariadb|mongo(sh)?)\\\\b[^|&]*([^-]-h\\\\s*|--host[=
            ↪  ])['\"'\"'\\\"]?[A-Za-z0-9$]\" && ! echo \"$CMD\" | grep
            ↪  -qE \"([^-]-h\\\\s*|--host[=
            ↪  ])['\"'\"'\\\"]?(localhost|127\\\\.0\\\\.0\\\\.1|::1|0\\\\.0\\\\.0\\\\.0
            ↪  then echo \"BLOCKED: direct remote DB connection\" >&2; exit 2;
            ↪  fi; exit 0'"
          }
        ]
      }
    ]
  }
}
```

Before you paste it: this guard needs `jq`. Run `command -v jq` first. Without it the hook exits `1` and prints `GUARD INACTIVE`, and because Claude Code treats every exit code other than 2 as "carry on", **the command still runs**. An inactive guard and a working guard look the same from the outside. §5.3 explains why that exit code was chosen.

This is not a suggestion the agent can decline. When the command matches, the tool call is denied and the agent is told why, so it can choose a safe alternative (`drizzle-kit generate` + `migrate`, or a reviewed migration).

If you would rather install the maintained versions as named files — with tests and updates — `cc-safe-setup` ships `drizzle-migrate-guard`, `prisma-migrate-guard`, and a database-connection guard as part of its free, MIT-licensed collection. **All three are in the example library, not the default install** — the line below installs the eight core guards, and you add these by name (`--install-example drizzle-migrate-guard`). §6.4 has the full explanation and one thing to know before you enable the checkpoint hook:

```
# free, runs locally, sends nothing anywhere
npx github:yurukusa/cc-safe-setup
```

One honest caveat: a hook can block a destructive *command*, but it cannot reach inside an ORM library call that your application code makes directly. The guard above covers the shell commands the agent runs — which is where every incident in §1.1 happened — not a `prisma.$executeRaw` buried in a script. For that layer, the defense is the same as it has always been: backups, which is the next section.

1.5 If it already ran — the recovery order

If you are reading this *after* the fact, do one thing before anything else: **stop new writes.** Every row written after the wipe lowers your odds of a clean restore. Then work in this order.

1. Your provider's point-in-time restore — your best shot

Most managed databases keep a continuous backup you can rewind to a timestamp seconds before the drop. Check this first.

- **Supabase** — Database □ Backups □ Point in Time (Pro plan and above).
- **Neon** — Time Travel, or branch from a past timestamp; the restore is near-instant.
- **PlanetScale** — Backups □ restore to a new branch.
- **AWS RDS / Aurora** — "Restore to point in time," which builds a *new* instance at the chosen moment.

- **Railway** – no point-in-time restore. The reporter in #27063 had no backup turned on; Railway's own documentation (checked 2026-09-03) describes **opt-in** volume backups. Whether that option was available on their plan in February 2026 is not something I have verified. Go and look at your own volume's backup setting now, before you need it.

2. A manual dump

Look for `pg_dump` / `mysqldump` output, a CI artifact, a `db/backup.sql`, or a staging copy. Note that `prisma migrate reset` and `db push --force-reset` re-seed from your `seed.ts` / `seed.sql` – so your seed file is the floor of what you can rebuild without a real backup.

3. Postgres, no backup — the last resort

Before any new write, you can try `pg_dirtyread` to read rows from pages the database has not yet vacuumed, or restore from the write-ahead log if WAL archiving was on. Both are partial and fragile; treat them as a salvage operation, not a restore.

The honest truth

A `DROP TABLE` or a `--force-reset` with no point-in-time restore and no dump is, in most cases, **unrecoverable**. Several of the incidents in §1.1 ended exactly this way. That is the whole reason the hook in §1.4 stops the command instead of trying to undo it: there is often nothing to undo. The two defenses are not interchangeable. The hook keeps the command from running; the backup is what saves you if something slips past the hook (an application-level call, a command shape you did not anticipate). You need both, and you need them in place *before* you leave an agent running unattended.

1.6 The one-paragraph version

Turn on point-in-time restore – or a nightly `pg_dump` – on every database your agent can reach. Put the `PreToolUse` guard from §1.4 in front of the destructive commands. Do not rely on a line in `CLAUDE.md`; it has been ignored,

in writing, more than once. With both layers in place, an agent that reaches for `drizzle-kit push --force` at three in the morning hits a wall instead of your production data — and on the rare chance something gets past it, you have a timestamp to rewind to. That combination is the difference between an incident and a catastrophe.

Chapter 2 — When your agent deletes your files

Files are more forgiving than a database. If a file is committed to git, a deletion is an inconvenience, not a loss. But there is one shape of file deletion that is every bit as final as `DROP TABLE`: a recursive force-delete pointed at the wrong directory. `rm -rf` does not ask, does not move things to a trash, and does not care whether the path was `./build` or `~`. This chapter is about keeping that one command from ever pointing somewhere fatal, and giving everything else a recycle bin.

2.1 What actually happens

The failure mode comes in two flavors.

The first is the **catastrophic target**: `rm -rf` aimed, through a bad variable expansion or a misread path, at `/`, at `~`, at `$HOME`, or up through `..` into a parent tree. An unset variable is the classic trigger — `rm -rf "$DIR/build"` becomes `rm -rf /build` when `$DIR` is empty, or `rm -rf "$HOME/$PROJECT"` wipes your home directory when `$PROJECT` is empty. An agent assembling a cleanup command from context is exactly the kind of actor that produces an empty variable and runs the result without flinching.

The second is **scope creep on a delete that was supposed to be small**. In one documented case, a user asked the agent to clear a specific set of telemetry rows; it deleted those and then an unrelated set of 12,547 rows nobody had mentioned. The same incident report describes the agent deleting an entire Docker database volume during what was meant to be a migration fix (anthropics/claude-code#46779). "Delete this one thing" became "delete more than you asked."

2.2 Why an agent does this

An agent cleaning up after itself is helpful right up until the path is wrong. "Let me remove the old build artifacts," "let me clear the cache," "let me reset the

working directory" – each is a reasonable intention that compiles down to a recursive delete. The model is not reckless; it simply does not have the human reflex of pausing when a delete path looks suspiciously short, or when a variable might be empty. It executes the command it composed, and `rm -rf` executes back, immediately and silently.

2.3 The block — stop the fatal target, recycle the rest

Two layers. First, a hard block on the catastrophic targets – a recursive delete that *names* `/`, a top-level directory, `~`, `$HOME`, or a parent-directory traversal is refused, whether or not the target is quoted. (What it cannot judge is a target the shell computes; see the limit at the end of this chapter.) Merge it into `~/.claude/settings.json` – add the entry to the existing `hooks.PreToolUse` array rather than pasting over the file, and check the count before and after, as §1.4 sets out:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "bash -c 'INPUT=$(cat); command -v jq >/dev/null 2>&1 ||
            ↪  { echo \"GUARD INACTIVE: jq not found; this hook is not
            ↪  protecting you\" >&2; exit 1; }; CMD=$(echo \"$INPUT\" | jq -r
            ↪  \".tool_input.command // empty\"); [ -z \"$CMD\" ] && exit 0;
            ↪  if echo \"$CMD\" | grep -qE
            ↪  \"\\\\brm\\\\s+([^;|&]*\\\\s)?['\"'\"'\\\"]?(/+\\\\*?|/+[^/
            ↪  '\"'\"'\\\"]+(/+\\\\*?)?|~(/+\\\\*?)?|~/+[^/
            ↪  '\"'\"'\\\"]+(/+\\\\*?)?|\\\\$\\\\{?HOME\\\\}?(/+\\\\*?)?|\\\\$\\\\{?HOME\\\
            ↪  '\"'\"'\\\"]+(/+\\\\*?)?|/+(home|Users)/+[^/
            ↪  '\"'\"'\\\\\\\"]+(/+\\\\*?)?|/+(home|Users)/+[^/
            ↪  '\"'\"'\\\\\\\"]+/+[^/
            ↪  '\"'\"'\\\"]+(/+\\\\*?)?)['\"'\"'\\\"]?\\\\s*(['\"'\"'\\\"]|$|\\\\s)\";
            ↪  then echo \"BLOCKED: rm targeting /, your home, or a directory
            ↪  directly beneath either\" >&2; exit 2; fi; if echo \"$CMD\" |
            ↪  grep -qE \"\\\\brm\\\\s+[^;|&]*(^|[^.])\\\\.\\\\./|\\\\s|$)\";
            ↪  then echo \"BLOCKED: rm with parent-directory traversal\" >&2;
            ↪  exit 2; fi; exit 0'"
          }
        ]
      }
```

```
      ]
    }
}
```

Before you paste it: this guard needs `jq`. Run `command -v jq` first. Without it the hook exits `1` and prints `GUARD INACTIVE`, and because Claude Code treats every exit code other than 2 as "carry on", **the command still runs**. An inactive guard and a working guard look the same from the outside. §5.3 explains why that exit code was chosen.

The rule has one shape, and it is worth stating exactly, because both the over-blocking and the under-blocking live in the details: **an `rm` is refused when its target is `/`, your home, or anything one level beneath either** — quoted or not, `~` or `$HOME`, long flags or short, with or without a trailing slash. So `rm -rf /`, `rm -rf "/"`, `rm -rf ~`, `rm -rf ~/Documents`, `rm -rf ~/Documents/`, `rm -rf $HOME/projects`, `rm -rf /home/you`, `rm -rf /build`, `rm -rf //` and `rm --recursive --force /` are all refused, and so is a parent-directory traversal like `rm -rf ../../anything`.

The target may be anywhere in the command, not only first. `rm -rf dist ~` is refused on the `~`, and `rm -rf node_modules /home/you` on the absolute home — an earlier edition read only the argument directly after the flags, so both of those went through while the guard watched. Spelling your home as an absolute path counts: `/home/you` and `/Users/you` are treated exactly like `~`.

Note what that sentence does *not* say: it does not say recursive. The rule reads the target, not the flags, so `rm ~/notes.txt` — one file, no `-r` — is refused too. That is deliberate; a single file directly in your home is usually a dotfile you did not mean to delete. It is also the most common false stop you will meet, so it belongs here rather than in a footnote.

Two levels down, it passes: `rm -rf ~/.cache/puppeteer` and `rm -rf /tmp/mycache` are two segments and go through, as do deeper ones like `rm -rf ~/projects/app/node_modules` and relative paths like `rm -rf ./build`. That boundary is a deliberate trade and the first edition of this guard got it wrong in both directions — first by missing quoted home paths entirely, then by refusing every path under `$HOME`, which would have made a developer disable the hook inside a day. A guard you turn off protects nothing, and Chapter 5 is about exactly that. Every case above was verified by piping it through the hook and reading the exit code, in both directions.

The quoted forms are worth calling out, because **the previous edition of this guard let all of them through.** `rm -rf "$HOME"` — your home directory, in the most ordinary quoting a shell script would use — returned exit code 0. The pattern expected an unquoted target and nothing told it otherwise. That is the failure mode this book is about, found in this book's own hook, so it goes in the book rather than into a quiet patch.

The second layer turns ordinary deletes into recoverable ones. `cc-safe-setup` ships `file-recycle-bin.sh`, which copies a target into `.claude/recycle-bin/` before the `rm` runs, so a normal delete can be undone with a plain `cp`. **Two limits worth knowing before you count on it**: it only fires when the command *starts* with `rm` — so `cd src && rm old.txt` is not saved, which is the compound shape 87% of real commands take — and it only copies regular files, so `rm -rf ./olddir` saves nothing. It also ships `rm-safety-net.sh` (blocks recursive deletes against critical paths and traversal, the maintained version of the hook above) and `bulk-file-delete-guard.sh` (catches a single `rm` that would remove a large number of files at once). **All three are examples, not part of the default install** — add each by name with `--install-example <name>` after running:

```
npx github:yurukusa/cc-safe-setup
```

The recycle bin is the difference between "I deleted the wrong file" being a thirty-second `cp` and being a recovery operation.

2.4 If it already ran — the recovery order

1. The recycle bin

If you had `file-recycle-bin.sh` installed before the delete, your file is sitting in `.claude/recycle-bin/`. Restore it:

```
cp .claude/recycle-bin/<file> <original-path>
```

2. Git

If the file was committed, it is in the repository regardless of the delete. Recover the last committed version:

```
git checkout HEAD -- path/to/file
```

If it was deleted *and* the deletion was committed, find the commit before the deletion and restore from there:

```
git log --oneline -- path/to/file     # find the last commit that had it
git checkout <commit> -- path/to/file
```

3. The filesystem's own safety nets

If the project lived on a filesystem with snapshots (APFS local snapshots on macOS, ZFS/Btrfs snapshots, a Time Machine backup, or a synced folder like Dropbox/OneDrive with version history), check there before giving up. These are outside git and outside the agent, so a delete inside the repo does not touch them.

The honest truth

A file that was deleted by `rm`, was never committed to git, had no recycle-bin hook, and lived on a plain filesystem with no snapshots, is **gone**. Undelete tools that scan raw disk blocks occasionally work, but only if you stop writing to the disk immediately, and they are unreliable for the small text files most projects are made of. The recycle bin and a committed git history are cheap to set up beforehand and decisive afterward; raw-block recovery is neither.

2.5 The one-paragraph version

Install the block so that a recursive delete naming `/`, a top-level directory, `~` or `$HOME` — quoted or not — is refused. Then add the recycle-bin hook so every other delete is a `cp` away from undone, and commit often, so git is a second net under the first.

One thing the block cannot do, and you should know it before you rely on it. The empty-variable accident — `rm -rf "$DIR/build"` becoming `rm -rf /build` when `$DIR` is unset — happens *after* the hook has answered. A `PreToolUse` hook is handed the command as text, before the shell expands

anything; at that moment the target is a variable name, not a path. I fired exactly that command at the block to be sure, and it passes. What closes that gap is `set -u` in your own scripts so an unset variable is an error, the recycle bin in §2.3, and the commit underneath. The block catches the *written* catastrophic target; the layers underneath catch the *computed* one.

Chapter 3 — When you lose your git history

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3.1 What actually happens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3.2 Why an agent does this

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3.3 The shield — save first, block the irreversible

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3.4 If it already ran — the recovery order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

1. The reflog — your branch's own history

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

2. Stashes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3. Dangling commits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

The honest truth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3.5 The one-paragraph version

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Chapter 4 — When `/rewind` eats your code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

4.1 What actually happens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

4.2 What `/rewind` cannot touch — your recovery map

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

4.3 If it already happened — the recovery order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

1. Check git first

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

2. Check your checkpoint hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

3. Check what was never tracked

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

The honest truth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

4.4 Prevention — and why no hook can confirm the menu

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

There is a second lever, and it is not a hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

4.5 The one-paragraph version

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Chapter 5 — Why a written rule is not a control point, and a hook is

Every chapter so far has reached the same conclusion from a different direction: do not rely on telling the model to be careful; put a guard in front of the command. This chapter explains *why* that is true, so the guards stop feeling like overkill and start feeling like the obvious floor. It also shows you how to confirm a guard actually works — because a hook you have not tested is just a different kind of false confidence.

5.1 What a written rule does, and what it cannot promise

The clearest single data point is issue #46779. The user's `CLAUDE.md` contained this, in plain language:

> "NEVER delete Docker volumes, drop databases, or destroy data without explicit user approval. Always pg_dump first. Dev data is not disposable."

The agent deleted a Docker database volume anyway. Later, asked to clear one specific set of rows, it deleted those and an unrelated set of 12,547 rows. The instruction was explicit, it was present, and it was ignored — more than once. This is not a story about a careless user who forgot to write the rule. The rule was there.

That is the strongest case in the public record, and it is one case. I have measured the other side of it myself, because selling you a hook on the strength of "rules get skipped" would be selling you something I had not checked.

Across 48 runs on two tasks (Claude Code 2.1.246, August 2026): with the prohibition written into the project's `CLAUDE.md`, the banned command was attempted **0 times in 25 runs**. Without it, **23 times in 23 runs**. Rule length, its position in the file, a conflicting instruction elsewhere in the same file, and

the severity of the operation produced no measurable difference. The protocol and the raw runs are in the companion book *CLAUDE.md Under Test*.

So the honest sentence is narrower than "instructions are ignored", and more useful. **A written rule buys you a "usually" that lives inside the model, and 0 out of 25 carries a 95% upper bound near 11%.** That is a number you can accept for a typo and cannot accept for a `DROP DATABASE`. A hook is not a better rule; it is a different kind of thing – it runs outside the model, and you can test it.

5.2 Why a clear instruction still gets ignored

Two mechanisms, both structural rather than occasional.

Context compaction (unverified – my runs did not reach it). A long session does not keep every token in view; when the context fills, earlier turns are summarized. Whether your `CLAUDE.md` rules survive that summary verbatim is something I have **not** measured, and the runs above were short enough that compaction never fired. I include it because it is the mechanism most often named, not because I have evidence for it. Read it as a reason the "usually" could get worse in long sessions – not as a demonstrated failure.

Model override (a mechanism, not a measurement). I have not isolated this one in a run either; #46779 is consistent with it, and "consistent with" is not "demonstrated". Even in view, an instruction is an input to the model's judgment, not a hard constraint on it. When the instruction ("never reset the database") conflicts with the immediate goal ("make this migration apply"), the model can decide, in the moment, that the goal wins – especially when a `--force` flag presents itself as the direct path. The instruction loses not because the model is defective but because that is what instructions are: guidance, weighed against everything else, not a wall.

A note in `CLAUDE.md` is a request. To stop an irreversible command you need something that is not a request.

5.3 What a hook is, and why it cannot be talked out of

A `PreToolUse` hook is a small program that Claude Code runs *before* it executes a tool call. It receives the tool input on stdin as JSON, and its exit code decides what happens:

- **exit 0** – allow the tool call to proceed.
- **exit 2** – block the tool call; the message your hook prints to stderr is shown to the agent so it can choose a different path.

The hook runs in the shell, at the system level, every time, with no dependence on what the model remembers or decides. It cannot be summarized away by compaction, and it cannot be overridden by the model's judgment, because the model never gets a vote – the command is intercepted before it runs. That is the entire reason the guards in this book are hooks and not paragraphs. A paragraph asks the model not to run `drizzle-kit push --force`. A hook makes the command not run.

With one precondition, which is not rhetorical. "Every time" assumes the hook can actually read the command, and the guards in this book read it with `jq`. On a machine without `jq`, a guard written the obvious way exits `0`, which Claude Code reads as permission. I tested this by stripping `jq` from the `PATH` and firing a recursive delete of `/` at a guard with no such check; it went straight through. Rather than take my word for which systems ship `jq`, run `command -v jq` on the machine you actually leave the agent running on. If it prints nothing, any guard that parses its input with `jq` and does not check for it is allowing everything. That is why every block printed in this book – Chapters 1, 2, 3 and 6 – opens by checking for `jq` and announcing itself inactive if it is missing – and why that check exits `1` rather than `0`. Per the hooks reference, stderr from a hook that exits 0 "goes to the debug log only, never the transcript", while a non-zero, non-2 exit lets the action proceed *and* surfaces the first line of stderr to the operator. A hook cannot be argued with, but it can be absent, and absence is silent unless you pick the exit code that makes it speak.

5.3.1 A hook only sees what Claude Code runs as a tool call

This is the boundary of everything in this book, and it cuts both ways.

It is why `/rewind` has no gate (Chapter 4): a slash command is not a tool call, so there is no `PreToolUse` interception point, and the defense there is the recoverable layer underneath. It is also why a command you type yourself, in your own terminal, is never seen by any of these guards. They are not a filesystem permission. They are a filter on one specific path.

The cut you will actually notice is the other way round, so it is worth saying plainly before it surprises you. **Several of the recovery commands in this book**

will be blocked by the guards in this book – if you installed the Chapter 3 or Chapter 6 guard, and you ask the agent to run them for you. (The single blocks printed in Chapters 1 and 2 do not stop them: I fired all four at those and every one came back `0`.) I fired every recovery command printed in Chapters 2, 3 and 4 at the installed guards, and four came back `2` – two of them the same `git checkout <commit> --` shape, printed once below:

```
git checkout HEAD -- path/to/file          exit 2
git checkout <commit> -- path/to/file      exit 2
git reset --hard HEAD@{1}                  exit 2
```

That is the guards working as designed – `git checkout --` and `git reset --hard` discard uncommitted work, which is exactly what they are there to stop, and a guard cannot tell "I am restoring" from "I am destroying" by reading the command string. But if you are recovering from an accident by asking the agent to do it, you will hit a wall at the worst possible moment.

Run recovery commands yourself, in your own shell. That is the answer, and it is a better one than any bypass: a recovery is the moment you least want an agent improvising.

If you would rather have the agent do it, check first whether the guard you actually installed has a disable switch, because the blocks printed in this book do not – they are deliberately small, and a small guard with an off switch is mostly an off switch. The maintained `destructive-guard` in `cc-safe-setup` reads `CC_ALLOW_DESTRUCTIVE=1`; if that is the one you have, set it for the single call, never for the session, and never in your shell profile.

5.4 Test your hook — or you have a paragraph with extra steps

A hook that does not actually block the command it claims to block is worse than no hook, because it gives you confidence you have not earned. So test every guard the same way Claude Code will invoke it: pipe a realistic tool input as JSON to the hook and read the exit code.

```
# should BLOCK – expect exit code 2
echo '{"tool_input":{"command":"drizzle-kit push --force"}}' \
  | bash ~/.claude/hooks/your-guard.sh ; echo "exit: $?"

# should ALLOW – expect exit code 0
echo '{"tool_input":{"command":"drizzle-kit generate"}}' \
  | bash ~/.claude/hooks/your-guard.sh ; echo "exit: $?"
```

Check both directions every time: the dangerous command must return `2`, and a safe lookalike must return `0`. It is easy to write a guard that blocks the dangerous command and *also* blocks half your normal workflow – a guard you will disable within a day. It is just as easy to write one whose pattern looks right but quietly matches nothing, so it returns `0` on the very command it was meant to stop. Only the exit codes tell you which one you have. Every guard in this book was checked this way, in both directions, before it was printed – and that process caught a real escaping bug in one of them that would otherwise have shipped a guard that looked correct and blocked nothing.

5.5 The one-paragraph version

Instructions in `CLAUDE.md` are guidance the model weighs against its goal. In my own 48 runs a written prohibition held every time it was present (0 of 25) and the command appeared every time it was absent (23 of 23) – but 0 of 25 is not zero risk, and there is a documented case of a written "never drop the database" rule being ignored twice. A `PreToolUse` hook is not guidance: it runs before the command, in the shell, and its exit code (`2` to block, `0` to allow) decides the outcome with no vote from the model. That is why guards beat paragraphs. And the only way to know a guard works is to pipe the command through it and read the exit code – in both directions – because an untested hook is just a paragraph with extra steps.

Chapter 6 — Building your safety net

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

6.1 The complete guard — one block, paste once

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

6.2 What it covers, and what it does not

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

What happened when I fired it, in both directions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

What it still does not catch, with the exact commands

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

The line that decides whether any of this runs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

6.3 The second layer — backups and checkpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

6.4 The maintained set — install in one line

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

6.5 The discipline, in one line

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Appendix — The incident catalog

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Database destruction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

File destruction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

`/rewind` and checkpoint data loss

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Widely reported industry incident

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

How to read this table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Where these issues stand now

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.

Appendix B: The test set, in full

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/claude-code-safety.