

THE COMMITMENT LEDGER + PROOF-GATE

Your AI says "done" when it isn't. This makes "done" a fact the world proves, not a claim the agent makes.

THE PROBLEM

You hand an agent a task. It works, it reports back: "Done. I created the export script, wired it into the nightly job, and verified it runs clean." You move on. Three days later you go looking for the export and it isn't there. The script has a typo, the job never fired, and the "verification" was the model narrating what it intended to happen, not what happened.

This is the single most expensive failure mode of building with AI agents, and it has a name: "should have done, but didn't." It is not lying. The model genuinely predicted a plausible completion and wrote it down. But a claim in a chat log, a handoff note, or a status update is just prose. Nothing runs it. Nothing checks it. So half-finished work rolls forward silently, dressed as finished work, and you only find out when it costs you.

The concrete before-scenario: an agent promises "I'll set up the daily backup." It writes that promise into a task file. Tomorrow the agent (or the next session) reads the task file, sees the promise, assumes past-you handled it, and builds on top of a backup that does not exist. The promise was never wrong on paper. It was just never true in the world.

THE PRINCIPLE

A commitment is not closed by a claim. It is closed by an observation the world provides.

Every promise ships with a proof: a read-only command that exits successfully only once the deliverable actually exists. The agent does not

get to say "done." The proof says done, or nothing does. The claim and the check are two different things owned by two different parties, and only the check has authority.

The non-obvious part is the direction of trust. You are not asking the agent to verify its own work more carefully. You already know it will confidently verify work it never did. Instead you move the verdict outside the agent entirely, to a command whose exit code the agent cannot argue with. "Done" stops being a sentence the model generates and becomes a fact the filesystem, the git log, or the API either confirms or denies.

Receipts, not hype.

HOW IT WORKS

The whole system is a small ledger plus a gate that runs on a schedule.

A commitment is one record. When any promise is made, you write a record with, at minimum: an id, the promise text, a proof command, a due date, a status (open), and a miss counter. The proof is the load-bearing field. It is a short shell command that observes whether the deliverable exists. "The backup script exists and is executable": `test -x backups/nightly.sh`. "The report has at least two rows": `test $(wc -l < out/report.csv) -ge 2`. "The fix is committed": `git log --oneline -1 | grep -q FIX-123`.

A proof must observe, never write. This is the rule the entire design bends around, because a proof that can write can satisfy itself. If a proof were allowed to create the file it is checking for, the agent could pass any commitment by having the "check" do the work. So proofs are constrained to read-only commands. In our build the proof text is model-written and then executed by the shell, so we validate it against an allowlist of read-only commands (`test`, `grep`, `ls`, `stat`, `wc`, `cat`, `find`, `git`, `jq`, and similar) rather than trying to blacklist dangerous ones. More on why below.

A baseline check at creation. The moment a commitment is filed, its proof runs once. If the proof already passes before any work is done, the commitment is vacuous: the gate would close it as "done" tomorrow with zero work performed. That gets flagged immediately. A proof that is already green is not a proof, it is a rubber stamp.

The gate runs on a schedule. Once a day (pick your cadence) a single job walks every open commitment and runs its proof:

- Proof passes: the commitment is closed as done. The observation closed it, not a claim.

- Proof fails and the due date has passed: a miss is counted and the item goes overdue. It turns red wherever you watch your work, and it gets surfaced loudly, not buried.
- Proof fails but it is not due yet: still open, on track, left alone.

Nothing rolls forward silently. A commitment can only end two ways: done (proof passed) or dropped (an explicit, recorded reason). There is no third path where a promise just quietly disappears. If you want out of a commitment you drop it on purpose and the drop is visible, which means "commit, ignore, drop, re-commit" thrashing shows up as a pattern instead of hiding.

One open commitment per owner. An owner with an unmet promise cannot pile new promises on top. They finish it, or they drop it, then they commit to the next thing. This keeps the ledger honest about capacity. (You can allow stacking for lightweight, session-level promises, but the default is single-slot for anything that matters.)

Closing by hand still runs the proof. Even a manual "mark this done" runs the proof first. If it fails, you cannot close it unless you force it with a written note explaining why the proof itself is wrong. The claim never gets to override the observation silently.

THE GOTCHAS

These are the traps we hit and encoded. They are the part that saves you a week.

A denylist for proofs does not hold. Our first version blocked "dangerous" commands and allowed everything else. Skeptics walked straight through it: backticks, `find -delete`, `truncate`, `python3 -c "..."`, `/bin/rm`, `git -C . reset`. There are too many ways to write. The fix is to invert it: allow only a known set of read-only commands in command position, reject everything else. Allowlist, never denylist, for anything a model writes and a shell runs.

Pipes and logic are legitimate; you cannot just ban them. A real proof is often `test $(ls dir | wc -l) -ge 2`. So you cannot forbid pipes, `&&`, `||`, or `$()`. You have to actually parse: split the command on every operator that starts a new command, then check that the first word of each segment is on the allowlist. That is the only way to allow `grep -q "a|b" file` (a quoted pipe that is just a pattern) while still catching a real second command hiding after a pipe.

Quoted spans are arguments, not commands. `grep -q "a|b" f` has a pipe inside quotes. If you split naively you will "discover" a command called `b` and reject a perfectly good proof. Strip quoted spans out before you do the structural check. But watch the exception: a `$(...)` inside quotes can still reach command position through substitution, so you keep those visible.

Two allowed commands can still write. `find` and `git` are on the allowlist but each has write-capable modes. `find` can `-delete` or `-exec`. `git` can reset or checkout. So they get extra constraints: `find` may only search, and `git` may only run read-only subcommands (`log`, `show`, `status`, `diff`, `ls-files`, `rev-parse`, `grep`). Allowlisting the command name is not enough when the command is a Swiss army knife.

No redirects. `>` and `<` write files. A proof with a redirect is a proof that can create its own evidence. Ban them outright (after you have stripped quotes, so a `>` inside a search pattern does not trip it).

A corrupt ledger must refuse, not reset. If the ledger file is unreadable, the tempting move is to start from an empty list so the pipeline does not crash. That silently erases every open promise on the next save. Instead, refuse to write and preserve the file. Degrade by skipping, never by deleting evidence. (We raise a normal error the scheduled job can catch and skip, while the command-line tool hard-exits so a human notices.)

Write the ledger atomically. Write to a temp file, then rename over the real one. A crash mid-write must never leave a half-written ledger, because the ledger is the one file that remembers what was promised.

A promise without a deadline is a wish. Every commitment gets a due date, and the default is aggressive: due at the next gate run. An open-ended promise never becomes overdue, so it never generates pressure, so it never gets done. Aggressive-but-real beats comfortable-but-ignored.

Human steps do not belong in this ledger. The proof has to be a real command a machine can run. "Ask the client" or "wait for approval" cannot be proven by a shell, so those are not machine commitments and should not be filed here. Reject a proof that is just a person's name or "n/a."

IMPLEMENT IT YOURSELF

1. **Pick a store.** A single JSON file, a SQLite table, or a small collection. Each commitment record needs: id, text, proof, due, status, misses, created, closed, note. Keep it local and keep it simple.

2. **Define "proof" as a read-only command in your world.** Shell exit code is the cleanest, but it can be any check that returns pass or fail: an HTTP request that expects 200, a SQL query that expects a row, a test that must be green. The only hard requirement is that it observes the deliverable without creating it.
3. **Write a validator for proofs.** Allowlist the read-only operations you trust. Reject writes, redirects, and substitution tricks. If your proofs are model-generated, treat this as a security boundary, not a nicety, and parse rather than pattern-match. If your proofs are human-written and reviewed, you can be lighter here, but keep the read-only rule.
4. **Baseline-check on creation.** Run the proof the moment a commitment is filed. If it already passes, flag it as vacuous and make someone look.
5. **Build the gate.** One scheduled job that loads all open commitments, runs each proof, and applies the verdict: pass closes it, fail-past-due counts a miss and marks it overdue, fail-not-yet-due leaves it open. Print one line per item so there is a log.
6. **Make overdue loud.** Overdue commitments must show up where you actually look: the dashboard, the morning summary, the standup. A broken promise nobody sees is the same as no ledger at all.
7. **Force the two-exit rule.** Wire your "done" path to run the proof first and refuse on failure unless forced-with-a-reason. Make "drop" require a written reason. Remove every code path that lets a commitment vanish quietly.
8. **Enforce single-slot where it counts.** Block a new open commitment for an owner who already has one open. They close it or drop it first.

THE COPY-PASTE TEMPLATE

The commitment record:

```
{
  "id": "backup-2026-07-13-a1b2",
  "text": "Nightly backup script exists, is executable, and runs clean",
  "proof": "test -x backups/nightly.sh && backups/nightly.sh --dry-run",
  "created": "2026-07-13T09:00:00",
  "due": "2026-07-14",
  "status": "open",
  "misses": 0,
  "closed": null,
```

```
"note": ""  
}
```

Note: that proof runs the script, which is a write. Swap it for a pure observation, for example `test -x backups/nightly.sh && test -f backups/last-run.ok`, where the script itself drops the `.ok` marker. The proof observes the marker; it never creates it.

The proof validator (skeleton, adapt to your language):

```
ALLOWED = {"test", "[", "grep", "ls", "stat", "wc", "cat",  
           "head", "tail", "find", "git", "jq", "cut", "sort", "uniq"}  
ALLOWED_GIT = {"log", "show", "status", "diff", "ls-files", "rev-parse", "grep"}  
  
def validate_proof(proof: str) -> tuple[bool, str]:  
    proof = proof.strip()  
    if not proof:  
        return False, "empty proof"  
    if proof.lower() in {"none", "n/a", "-", "manual"}:  
        return False, "proof must be a real command, not a human step"  
    if "`" in proof:  
        return False, "no backticks; use $( ) if you need substitution"  
  
    # Quoted spans are arguments (patterns, paths). Drop them before the  
    # structural checks so a quoted pipe or > is not mistaken for an operator.  
    # But keep any span containing $( visible, since it can reach command position.  
    # TODO: strip quotes, preserving $(...) spans  
  
    if ">" in stripped or "<" in stripped:  
        return False, "no redirects; a proof observes, it never writes"  
  
    # TODO: split on every operator that starts a new command: || && ; | $( (  
    # For each segment, take the first word (command position) and require it in AL  
    for word in command_positions(stripped):  
        if word not in ALLOWED:  
            return False, f"'{word}' is not a read-only command"  
  
    # Extra constraints on the write-capable allowlist members:  
    # TODO: if 'find' appears, reject -delete/-exec/-execdir/-ok/-fprint  
    # TODO: if 'git' appears, require the next word be in ALLOWED_GIT  
  
    return True, ""
```

The gate (skeleton):

```
def gate(today):
    items = load_ledger()          # refuse, do not reset, if unreadable
    for c in items:
        if c["status"] != "open":
            continue
        passed = run_readonly(c["proof"])  # timeout it; capture exit code
        if passed:
            c["status"], c["closed"] = "done", now()
        elif c["due"] < today:
            c["misses"] += 1          # overdue: count it, surface it loud
            # else: still open, on track, leave it
    save_atomic(items)              # temp file then rename, never partial write
```

The pre-flight checklist for anyone filing a commitment:

- [] The text names one deliverable, not a paragraph of intent.
- [] The proof is read-only. It observes the result; it cannot create it.
- [] The proof fails right now (you ran it; it is red before the work).
- [] The proof will pass only when the deliverable genuinely exists.
- [] There is a due date, and it is real, not "someday."
- [] This is a machine step, not a human one, so a command can actually check it.