

THE PROCESS-CHECK REGISTRY

A process cannot grade its own homework. Every recurring job registers a check that reads the outcome in the world, never its own success log.

THE PROBLEM

You wire up a job that runs on a schedule. A morning digest email. A backup. A cron that pushes a file to a bucket. A bot that texts you a reminder. It runs every day, quietly, and you stop thinking about it. That is the whole point of automation.

Then one day it stops working and says nothing.

Here is the exact failure that this primitive was built to prevent. A job was supposed to send a voice memo to a phone every morning at 8:20. It ran. Its own log said "sent." For days the log kept saying "sent." Nothing ever reached the phone. The send call threw an exception, the exception got swallowed, the code moved on and wrote "sent" to the log anyway. The log was a record of the code reaching a line, not of anything happening in the world.

This is the default failure mode of every scheduled job you will ever write. The log is written by the same code that failed. A job that crashes politely will tell you it succeeded, because "succeeded" is just the line after the one that broke. You do not find out until a human notices the thing that was supposed to happen never happened, which is usually days later and usually at the worst time.

You cannot fix this by making the job log more carefully. The job is the unreliable narrator. You need a second witness.

THE PRINCIPLE

A check must read the outcome in the world, never the process's own success log.

If the job's purpose is to write a file, the check looks for the file, with today's date, actually on disk. If the job's purpose is to send a message, the check reads the messaging system's own database and confirms a message row exists. If the job's purpose is to keep a server up, the check opens a socket and gets a 200. If the job's purpose is to sync a record, the check reads the record from the destination.

The check and the job must not share a code path. The job's log can be used for exactly one thing: deciding whether the job was even supposed to do anything today (more on that below). It can never be used as proof that the job succeeded. The claim does not close the loop. The observable closes the loop.

That is the whole idea, and it is the thing you are paying for: **the verifier reads ground truth, from a source the job does not write.**

HOW IT WORKS

The mechanism is a registry plus a runner.

The **registry** is a list of check entries, stored as data (JSON), tracked in version control so that editing it is a reviewed change like any code. Every recurring job in your system gets at least one entry. A job that both produces an artifact and delivers a notification gets two entries, because those are two distinct outcomes that fail independently (the file can render fine while the send dies).

Each entry has:

- **id** – a short slug to name the check.
- **what** – one plain sentence naming the outcome being verified, in world terms ("the digest actually reached the phone"), not code terms ("notify() was called").
- **by** – a wall-clock time like "09:20". The outcome must exist by then. Before that time the check reports "not due" instead of failing, so you do not get a false alarm at 6am for a job that runs at 8. Always-on daemons omit this and get checked every run.
- **days** – optional weekday filter, for jobs that only run certain days.
- **check** – the verifier itself: a type plus its arguments.
- **skip_if** – optional. A read-only command that exits zero when the job legitimately had nothing to do today.

The **check types** are a small, fixed set of ground-truth readers. The real implementation ships five, and they cover almost everything:

- **file_today** – a file exists at a path (the path can contain a `{date}` token) AND its modification time is today. Existence alone is not enough; a stale file from last week is a failure.
- **message_in_window** – query the messaging system's own store for an outbound message inside a time window today. This is the one that caught the original failure.
- **http_ok** – open a URL, require a 200.
- **process_running** – a named process is actually in the process table.
- **cmd** – run an arbitrary read-only shell command; exit zero passes. This is the escape hatch for anything the four typed checkers cannot express (filter a JSON file by a field, query a remote API, check a DNS record).

Each checker returns a pass/fail and a one-line human detail. The rule they all obey: **ground truth only, never the job's own log**.

The **runner** walks the registry once per invocation (typically itself on a schedule, every 15 or 30 minutes). For each entry:

1. If it is not due yet (before its `by` time, or not its day), skip it silently.
2. If `skip_if` exits zero, mark it skipped. The job legitimately did nothing today.
3. Otherwise run the check. Record pass or fail with a timestamp into a small state file.
4. On a **new** failure, send an alert.

That state file is what makes alerting sane. It stores, per day, the last result of each check and which failures have already been alerted. You do not want the same failure texting you every 15 minutes all day, so once you have alerted a failure you dedup it. But there is a trap in that dedup, covered below.

THE GOTCHAS

These are the traps that were hit in production and encoded into the design. Each one cost a real failure to learn. This is the part that saves you the pain.

1. **The window-masking false pass.** The very first version of the message check false-passed. The check was "an outbound message exists between 8:00

and 9:00." A different job happened to send its own message at 9:00, inside that window. So the check went green while the 8:20 job it was supposed to watch had actually failed. A shared channel means any traffic in the window satisfies the check. Fix: tighten every message window to hug that specific job's send time, and know that a message check on a shared channel is inherently weaker than a check on a private artifact. Where you can, prefer checking the job's own unique output (a specific file, a record with an id) over checking a shared delivery channel.

2. A broken skip rule must never hide a failure. `skip_if` exists so a job that correctly did nothing ("no items to process today") does not alert. But if the skip command itself errors, you must treat that as "not skipped" and run the real check. If a crashing skip rule counted as "skip," a broken skip rule would silence a real outage forever. Fail open toward checking, never toward skipping.

3. `skip_if` may read intent, never success. The skip rule answers one question: was the job supposed to act today? It reads the same input the job reads to make that decision (the queue, the schedule, the ledger) and re-derives the answer read-only. It must not read the job's own "I did it" log, because then you are back to letting the job grade itself. Intent data is allowed. Success claims are not.

4. Confirm the alert actually sent before you dedup it. The nastiest recursion: your alerting channel is itself a process that can fail, and it might be the very thing that is down. So only mark a failure as "already alerted" after the notifier confirms delivery. If the alert send fails, log it and leave the failure un-deduped so the next run retries. Otherwise a wedged notifier swallows its own outage warning. (This is real. A messaging app wedged for ten minutes and an alert was lost outright, which is why a separate check watches for messages stuck undelivered longer than a threshold.)

5. `mtime` is not a heartbeat if the job only writes on change. One sync job only rewrote its output file when the data actually changed. Most days nothing changed, so the file's `mtime` was days old even though the job ran fine every hour. A `file_today` check would have screamed false failures daily. When a job is write-on-change, do not check its file's freshness; check the outcome directly (read the destination and confirm the records match). Only use `file_today` when the job writes unconditionally on every run.

6. Shared output files need a content filter, not just a timestamp. Several jobs wrote to the same file. Checking "was this file touched today" could not tell which job touched it. The fix was a `cmd` check that reads the file and filters for the specific job's fingerprint (a record with today's date

AND a particular slot/type field). If two jobs share a sink, your check has to distinguish them by content.

7. Pick a state artifact the job writes unconditionally. The strongest cheap proof that a whole chain ran is a state file the last step writes on every pass, incidents or not. If the final step saves state unconditionally, a fresh timestamp on that file proves the entire upstream chain executed, not just that the wrapper started. Look for the unconditional write and anchor your check there.

8. A crashing checker must report failure, not vanish. Wrap each check so an exception inside it returns a failure with the error text, never an uncaught crash that skips the entry. An unreliable narrator that dies mid-sentence still must not be scored as "passed."

9. A silent-by-design job needs a liveness proxy, not an output check. Some jobs correctly produce nothing on a normal day (a poller that only acts when new data arrives). You cannot check for an output that is supposed to be absent. Instead check that the job is loaded and its last run exited clean, or that its unconditional state write is fresh. Distinguish "did nothing because healthy" from "did nothing because dead" using something other than the missing output.

IMPLEMENT IT YOURSELF

You can build a minimal version in an afternoon, in any language.

- 1. Make a registry file.** A list of entries as plain data, in version control. Start with your three most important scheduled jobs.
- 2. Write four or five ground-truth checkers.** file-exists-and-fresh, http-returns-200, process-running, and a generic run-a-read-only-command. Each returns `(ok, detail)`. That covers most jobs. Add domain-specific ones (query your message store, read your database) as you need them.
- 3. For each job, name the outcome in world terms.** Not "the code ran." What should exist in the world if it worked? A file? A row? A reachable endpoint? That sentence is your `what`, and it dictates which checker to use.
- 4. Set a `by` time.** Before it, the check reports "not due." This kills 6am false alarms.
- 5. Add a `skip_if` only if the job legitimately no-ops some days.** It reads the job's inputs read-only and re-derives "was I supposed to act." Never let it read the job's success log. Make a crashing skip rule fall through to the real check.

6. **Write the runner.** Walk the registry. Skip not-due and legitimately-skipped entries. Run the rest. Store results and an alerted-list in a per-day state file. Alert only new failures. Confirm the alert delivered before marking it deduped.
7. **Schedule the runner itself** every 15 to 30 minutes. And here is the recursion you must break: something has to watch the watcher. Have the runner write its own fresh state file every pass, and let a dead-simple external check (or your eyes on a dashboard) confirm the runner is itself alive. The buck stops at one thing you trust; make it the simplest possible thing.
8. **Backfill the rule going forward:** no new recurring job ships without registering a check in the same change. The check is part of the definition of done, not a follow-up.

THE COPY-PASTE TEMPLATE

A registry entry schema. Adapt the field names to your stack.

```
{
  "id": "nightly-backup-uploaded",          // short unique slug
  "job": "backup.timer",                  // scheduler label, informational only
  "what": "Tonight's backup landed in the bucket, dated today",
  "by": "02:30",                          // outcome must exist by this local time
  "days": [0,1,2,3,4,5,6],              // optional weekday filter (0 = Sunday)

  "check": {
    "type": "cmd",                        // file_today | http_ok | process_runnin
    "shell": "aws s3 ls s3://my-bucket/backups/$(date +%F).tar.gz | grep -q ."
    // ^ reads the DESTINATION, not the backup job's log
  },

  // OPTIONAL. Exits 0 only when the job legitimately had nothing to do today.
  // Reads job INPUTS read-only. NEVER reads the job's success log.
  // If this command errors, the runner must fall through and run the real check.
  "skip_if": null,

  "notes": "Why this outcome is ground truth; any masking caveat.",
  "confidence": "high"                   // high = private unconditional artifact
}
```

A checker skeleton. Every checker has the same contract: read the world, return `(ok, detail)`.

```
def check_file_today(cfg):
    """Ground truth: a file exists AND was modified today. Nothing about the job's
    path = expand(cfg["path"].replace("{date}", today()))
    if not exists(path):
        return False, f"{path} does not exist"
    if mtime_date(path) != today():
        return False, f"{path} exists but is STALE (last modified {mtime_date(path)})"
    return True, f"{basename(path)} written today"

def run_check(entry):
    fn = CHECKERS.get(entry["check"]["type"])
    if not fn:
        return False, f"unknown check type {entry['check']['type']!r}"
    try:
        return fn(entry["check"])
    except Exception as e:
        # GOTCHA 8: a crashing checker fails loud, never silently passes.
        return False, f"checker crashed: {e}"

def is_skipped(entry):
    cmd = entry.get("skip_if")
    if not cmd:
        return False
    try:
        return run_readonly(cmd).exit_code == 0
    except Exception:
        # GOTCHA 2: a broken skip rule must NOT hide a real failure.
        return False

def runner():
    state = load_today_state()          # {results: {}, alerted: []}
    new_failures = []
    for entry in registry():
        if not due(entry):               # before 'by' time / wrong day -> not due
            continue
        if is_skipped(entry):            # legitimately no-op'd today
            state["results"][entry["id"]] = {"ok": None, "detail": "skipped"}
```

```
        continue
    ok, detail = run_check(entry)
    state["results"][entry["id"]] = {"ok": ok, "detail": detail}
    if not ok and entry["id"] not in state["alerted"]:
        new_failures.append((entry, detail))

if new_failures:
    delivered = send_alert(format(new_failures))
    if delivered:
        # GOTCHA 4: only dedup on CONFIRMED deliver
        state["alerted"] += [e["id"] for e, _ in new_failures]
    else:
        log("ALERT UNDELIVERED – will retry next run")
save_today_state(state)
```

Start with your three most important jobs. Name each outcome in world terms. Point each check at a source the job does not write. That is the whole primitive, and it is the difference between finding out a job died at 8:20am from a check at 8:35, versus finding out from a customer three days later.