PURETENSOR
INTELLIGENCE DISTILLED

A Published Metric Is Not a Consumed Metric

Controls that look healthy and cannot fail

Paper: PT-R-2026-025 v1.0

Author: PureTensor Ltd

Date: 21 September 2026

Status: Published

Abstract

A monitoring control has two ends. A producer emits a signal: a gauge, an exit code, a reloaded configuration, a passing test. A consumer reads that signal and acts when it goes bad. In earlier work we catalogued "false green" controls, meaning checks that report healthy in a state where they cannot report anything else. This paper adds four more traps to that catalogue, all found in one night, and describes the method that found them. It also examines one outage that shows the cost of leaving such a trap in place.

Case A is the off-site backup of a database, an append-only experience ledger for a long-running agent. The backup was built on 20 September 2026. On the night of 20 to 21 September we closed its three open residue items: retention, alert rules, and a restore drill against the real Postgres target. Closing them uncovered four traps. First, backup gauges had been published for about 7.5 hours with no alert rule consuming them. Second, a new rule group reloaded successfully without ever being loaded. Third, a staleness threshold read green on an empty repository, because the gauge's sentinel value (-1) sits below any threshold. Fourth, a retention policy ran, reported success, and deleted nothing. A further failure came from the restore rehearsal itself. It had passed against SQLite, and it failed twice against Postgres, the store production actually uses.

Case B is a CI runner fleet of 28 runner units on a two-GPU compute node. On 15 September a storage mount failed and returned about 1 h 40 min later. Nothing restarted the runners. No alert fired, and the fleet was found offline about 31 hours after the original failure, only because a pull request's checks sat pending. In each case the producer's own success signal was true and irrelevant. The reload metric read 1, the job exited 0, the suite was green, the drill passed, and the mount was verified OK.

The method that caught the Case A traps is the same each time: drive the control into the state it exists to report, and watch the consumer rather than the producer. We state two additions to the taxonomy no more strongly than the evidence allows. An explicit load list silently drops anything added to it later. A sentinel value is a failing branch that a threshold cannot see. These are case studies, and they support no claim about how common such traps are across the fleet.

1. Controls That Cannot Fail

The fleet is operated day-to-day by an orchestrating agent under human direction. It monitors itself with Prometheus, running as two instances (one inside the k3s cluster, one on the monitoring tier), with Alertmanager for routing, node_exporter textfile collectors (small files of metrics written by scripts and scraped by the node exporter), and systemd timers on each host.

On 28 July 2026 a fleet audit found production healthy but four controls that could not fail. It catalogued four shapes of false green, and a fifth shape was added on 3 September. The catalogue is the frame for everything that follows, so we reproduce it here.

#	Shape	The instance found	Fix shipped
1	Classifier with no failing branch	Network tests labelled for a 100G class on physically 25G links fell to a generic rule that passes on any positive throughput	A real 25G class with a threshold from measured line rate (23.16 to 23.35 Gbps); an unrecognised category now fails
2	Detector with no expectation	A cluster probe never checked desired topology; a cluster at 1 of 3 nodes reported zero anomalies	Expected node count and resource floor configured; negative test drove it red
3	Artifact nobody measured	A backup-status JSON file froze for 11 days while every backup signal stayed green	Guarded version installed (1.6 s runtime), call bounded at 30 s, file age exported as a metric
4	Job exit code standing in for the artifact	Off-site git mirror jobs reported Complete every cycle; the older script was a no-op that exited 0	An exporter measuring mirrored trees on disk; on an abandoned tree, 63 repos, 62 flagged stale
5 (3 Sept)	An alerting lane the observer cannot read	A summary reported "0 fleet alerts" while 19 were active (12 warning)	A separate read-only lane from Alertmanager

The July audit also recorded rules. A monitor is not done until it has been watched going red on the real failure shape. That rule was earned, because two of the four July fixes exposed bugs in our own new rules: a `$value` that reported an epoch timestamp as a duration, and a test fixture whose series ended before the `for:` window (the length of time a condition must hold before an alert fires). Absence of a series is not absence of a problem, so every staleness rule needs a companion rule for its own metric going missing. "Unknown" is honest, and a fabricated green is not. Check the installed copy, not the repository copy.

A related precedent came from 24 August. A panel-level audit of our dashboards executed 386 panel targets. It found that the live dashboard ConfigMap (a Kubernetes object holding configuration files as keys) had drifted from the committed one on 6 of 7 dashboards, and that one fix had been committed and not applied for 26 days. The same audit found a dashboard present in the ConfigMap and in git but absent from the dashboard server, because the Deployment mounted an explicit allow-list of keys that never listed it. Section 4 shows the same mechanism in a different component.

2. The Backup and Its Residue

The ledger is backed up with restic, a deduplicating backup tool, to an append-only repository on an off-site disaster-recovery host in another country. There are three snapshot cadences. A stream snapshot runs every 5 minutes (288 per day; deduplication makes each one a delta). A signed checkpoint runs hourly at two minutes past the hour. A dump runs daily.

When (2026)	Event
20 Sep, afternoon	Backup built: 63 red contract tests written first; integration at 149 tests green, 153 after live fixes
20 Sep 16:55 UTC	First live checkpoint snapshot: 3 files, 6,727,761 bytes added, 1.08 s
20 Sep, about 17:00 UTC	Off-site chain-check timer installed (every 15 min), writing fork, chain, staleness, check and ramp-up gauges. First reading: stale 9 s, chain ok 1, fork 0, check ok 1
20 Sep	Restore drill passes on the off-site host, against a SQLite target only (8 files, 6.413 MiB restored, signature verified)
20 Sep	Residue recorded as open: no retention; no alert rules; Postgres-target restore not yet run
21 Sep 00:25 to 00:31 UTC	All three residue items closed and committed; alert group confirmed live

At the end of 20 September the system looked finished by most ordinary tests. It had a passing contract suite, a live snapshot, a working check timer and a verified restore. The residue note was honest about the gaps, but it phrased the alerting gap in a way that invites complacency: "chat notification from the check script is the live alert path today". Closing the three items is where the four traps surfaced.

3. Trap 1: An Exporter With No Rule

The chain-check gauges had been published since about 17:00 UTC on 20 September. When we came to write alert rules, we first searched the whole alerting manifest tree for the metric prefix. The search returned nothing. No rule consumed any of the gauges. The only live path from a bad backup to a human was one chat-notification line inside the check script itself. That line depends on the same script running correctly, which is exactly what a consumer should not depend on.

Until the rules existed, a stalled or forked backup would have been a dashboard colour, not a page. In the taxonomy this is shape 2, a detector with no expectation: the gauges stated facts, but nothing stated what the facts should be. The gap lasted about 7.5 hours, from about 17:00 UTC on 20 September to about 00:26 UTC on 21 September. No backup fault occurred in that window, so nothing was missed. The trap was found before it cost anything.

4. Trap 2: A Successful Reload of a Group That Was Never Loaded

The new rule file went into the alerts ConfigMap as a new key. After ConfigMap propagation (about 50 s) we sent Prometheus SIGHUP, the signal that tells it to reread its configuration. Every signal on the producer side said the change had landed. The reload-success metric read 1. The file was present in the pod at the right path with the correct hash.

The group was not loaded. Only the live rules API, the endpoint that reports which rules Prometheus is actually evaluating, showed it absent, returning "group not found". The cause was that `rule_files` in our configuration is an explicit list of paths, not a glob. A new file loads nothing until it is listed, and a reload with nothing new to load is a successful reload.

This had happened before in the same configuration file. An earlier cost-alerting group sat authored but unloaded for three weeks by the same mechanism, and a comment recording that fact was already present in the file. The comment did not prevent a recurrence. After we added the path, the rules API showed all 7 rules live with health "ok". The rule we extract is narrow:

prove a new rule group against the live rules API, never against a successful apply or a successful reload.

5. Trap 3: A Sentinel That Defeats a Threshold

The staleness gauge reports the age in seconds of the newest snapshot, or -1 when the repository holds no countable snapshot. A sentinel of this kind is a reserved value meaning "no measurement" rather than a measurement. The obvious rule, `stale_seconds > 5400` (90 minutes), therefore reads green when the repository is empty. An empty repository is the state in which the backup is most broken. The threshold has no failing branch for it, because -1 is less than every positive threshold.

The fix is a separate rule on `== -1`, critical, with a 30-minute `for:` window. The fix is small. The trap lies in how natural the original rule looks: a staleness threshold on a staleness gauge reads as complete. The rule we extract is to read every metric's out-of-band values before writing a threshold on it.

6. Trap 4: Retention That Runs Clean and Deletes Nothing

restic's `forget` command, which applies a retention policy, groups snapshots by `host,paths` by default and applies keep rules within each group. Each of our snapshots carries a fresh temporary export path and per-snapshot checkpoint tags. The default grouping therefore put every snapshot in a group of one. Under that grouping, `--keep-daily 14` keeps the single snapshot in every group, which means it keeps everything forever. The policy ran, reported ok, and deleted nothing.

The fix was `--group-by host`. To prove it we did not rely on the real policy, whose effect would not be visible for days. We ran a deliberately tight dry run instead (`--keep-within 1h --dry-run` on stream snapshots). It reported "keep 12 / remove 78" in one group, and the chain was intact afterwards. A policy that should remove most of the history, and visibly does so, has been shown to work. A policy that removes nothing today tells us nothing, because on a young repository removing nothing is also the correct answer.

As built, the policy keeps stream and drill snapshots for 7 days. Daily dumps are kept at 14 daily, 8 weekly and 12 monthly. Signed checkpoints are never passed to `forget`, because they are the continuity chain. The prune runs weekly on the off-site host against the local repository path, so the backed-up system still holds no credential that can delete. A post-prune chain check fails the run if continuity broke. Stale-lock clearing and 3 retries were added after a killed run left a lock that failed every later prune.

We map this trap to shape 4, a job's clean exit standing in for the artifact. The mapping is ours for this paper; the source notes do not assign a shape.

7. The Method: Mutation-Test the Test

The alert rules shipped with a promtool unit suite. promtool is Prometheus's own tool for evaluating rules against synthetic time series. The suite has 8 cases, D1 to D8. D1 is a healthy steady state that must be silent on all 7 rules, a negative control. The remaining cases are one positive case per fault: fork; broken chain; 2 hours stale (warns, does not page); 7 hours stale (pages); empty repository; check failing during ramp-up; and the verdict metric vanishing (fires at 130 minutes, silent at 60 minutes). D6 asserts both halves of trap 3: with the gauge at -1 for 60 minutes, both staleness rules are silent and the empty-repo rule fires.

Rule (generic name)	Condition	For	Severity
Fork	fork detected = 1	5 m	critical
Chain broken	chain ok = 0	5 m	critical
Stale	staleness > 5,400 s (90 min)	15 m	warning
Stale, critical	staleness > 21,600 s (6 h)	15 m	critical
Repo empty	staleness = -1	30 m	critical
Check failed (catch-all)	check ok = 0	30 m	warning
Signal absent	verdict metric absent	2 h	critical

The signal-absent rule follows the July rule on missing series. If the check timer dies, the gauges stop updating, every other rule stops evaluating meaningfully, and the whole group goes quiet, which reads exactly like health.

The suite passed on first write. On its own that proved nothing, since a suite can pass because it asserts too little. We applied a mutation (a deliberate, plausible bug introduced into the code under test) to the rule the suite was meant to pin: we changed the empty-repo condition from `== -1` to `> 5400`. The suite turned red. That shows the suite is not vacuous for that rule. The rule we extract: a unit test that has never failed has not been shown to work.

The four traps and the mutation all follow one pattern. In each we stopped asking the producer whether it had succeeded and forced the consumer to show us the bad state: the rules API rather than the reload status, an empty repository rather than a stale one, a tight dry run rather than the real policy, a mutated rule rather than the passing suite.

8. The Rehearsal That Skipped the Real Target

The restore drill of 20 September passed against SQLite. The production store is Postgres, and the Postgres restore path had never been run. On 21 September we ran it end to end from a LAN host, because the off-site host cannot reach the Postgres service. It failed. The SQLite store creates its schema on open and the Postgres store does not, so the restore died on a missing table in exactly the place where the SQLite restore had passed.

The fix bootstraps the idempotent migrations (schema changes safe to apply more than once) for Postgres targets. It sits behind a guard that refuses any connection string not naming a drill database. We tested the guard with a positive control: pointed at the production database, the restore exits 2 with a refusal. The next run exited 0. It restored 12,816 events, with a head hash equal to the live ledger's hash at sequence 12,816.

Fixing the first failure exposed the other half of the same asymmetry. The importer refuses a non-empty store. The SQLite branch had always deleted its scratch file, and the Postgres branch had no equivalent. Repeating the drill against the same scratch database therefore failed with "cannot import into non-empty store". We fixed this by dropping the scratch database's tables first, behind the same guard. Two runs back to back then both exited 0. Each dropped 10 tables and reproduced the live head hash at sequence 12,816.

Both failures would otherwise have appeared first during a scheduled first-production session with the operator present. The second was the more likely of the two, because a drill is not run once. The rule we extract: a rehearsal that skips the real target is not a rehearsal.

9. Case B: All CI Runners Down for About 31 Hours, No Alert

Case A shows traps found before they cost anything. Case B shows one that was not found in time.

When (2026)	Event
14 Sep, about 17:38	A credential rotation elsewhere in the fleet leaves the compute node's kernel storage client unable to authenticate
15 Sep 04:57	The block-device mount under the runners' home directory fails; all 28 runner units stop with systemd Result=dependency
15 Sep 06:39	Mount restored during that morning's remediation; its verification lists the runners' home mount as OK
15 Sep 06:39 to 16 Sep about 12:00 UTC	Nothing restarts the runners; systemd does not retry a unit whose start failed on a dependency
16 Sep about 12:00 UTC	Found because a pull request's checks sat "pending"; its runner was offline, and so were the other 27
16 Sep	systemd reloaded, all runner units started; 28 active, 0 failed; queued jobs ran and passed

The real outage, the period in which the storage was actually unavailable, lasted about 1 h 40 min. The self-inflicted outage after the storage returned lasted 29 hours or more. The total was about 31 hours.

Two separate gaps combined. First, no runner-availability rule existed in either Prometheus instance. The source note assigns this to shape 3, an unmeasured artifact. We read it as also an absent expectation, shape 2, since nothing anywhere stated "28 runners should be online". Second, the storage repair on the 15th was verified against the mount, and the mount passed. The consumers of the mount were not in the verification. The verification's own success signal, mount OK, was true and did not address the question that mattered for the runners.

The hardening recorded as open on 16 September was an alert on runner count or offline runners, and restart-on-failure plus mount retry on the runner units.

10. What the Cases Have in Common

Every trap in this paper left the producer healthy and the consumer missing or blind. In Case A a gauge was published with no rule; a rule file was present and reloaded but never loaded; a rule evaluated but could not reach its worst state; a retention job ran and removed nothing; a test suite passed without having been shown able to fail; and a drill passed on a target production does not use. In Case B, 28 units changed state and no rule read them. Each control's own success signal was true: reload successful = 1, job exit 0, suite green, drill passed, mount verified OK. None of those signals said anything about the consumer.

Case B shows the cost when nobody drives the control to its bad state: about 1 h 40 min of real failure became about 31 hours, found by chance. Case A shows the same class of gap closed at the cost of one night's work, before any fault arrived.

We propose two additions to the taxonomy, and we state them narrowly. The first is the explicit load list. Prometheus `rule_files` in our configuration, and the Deployment key allow-list found in August, both silently drop anything added after the list was written. The reload or rollout reports success, and the artefact is present on disk, but the list decides what is used. Two instances, one of which recurred in the same file, justify naming the shape. They do not tell us

how many such lists exist elsewhere in the fleet. The second is the sentinel value: an out-of-band reading such as -1 is a failing branch that a threshold cannot see, and it is closely related to shape 1. We have one instance.

11. What These Numbers Will Not Carry

These are case studies: one backup system, one runner fleet, one night's residue close, and one incident of about 31 hours. No rate or prevalence claim across the fleet is supported. We cannot say what fraction of our controls have a missing consumer, only that a careful close of one system's residue found four such gaps and the restore rehearsal found a fifth failure.

The unconsumed window for the backup gauges, about 7.5 hours, did not coincide with any backup fault. No harm occurred in Case A, and we do not claim a missed incident there. One internal note describes the same gap as "a full day". The chronology does not support that description, and we use the 7.5-hour figure.

The mutation test was a single mutation on one rule. It shows the suite is not vacuous for the empty-repo rule. It does not show that each of the 7 rules is pinned by the suite, and we have not claimed that.

The Case B timing carries a time-zone ambiguity. The outage start (04:57) comes from the incident note, which states no time zone. The discovery time is given as about 12:00 UTC. "About 31 hours" is the note's figure, and it could be about 32 if 04:57 was local summer time. The system journal for that period no longer exists, so we cannot resolve this. The duration is either about 31 or about 32 hours; the conclusion does not depend on which.

The mapping of trap 3, trap 4 and Case B onto taxonomy shapes is our own classification, not an independent one. Another reader could draw the boundaries differently, and the shapes overlap: Case B is plausibly both shape 2 and shape 3.

The cause of the 15 September mount failure comes from that morning's incident report and was not independently re-established for this paper. The finding does not depend on it: any mount failure followed by recovery would have left the runners down, because systemd does not retry a dependency failure.

The claims that survive are these. On the evidence here, each of the named controls reported success in a state where its purpose was unmet. In every Case A instance, driving the control into its bad state and reading the consumer exposed the gap. In Case B, the absence of any consumer for runner state turned a short storage outage into a long CI outage. The claims that do not survive are any general frequency, any claim that the method finds every such trap, and any claim that the Case A traps would have caused harm on a particular date.

12. What We Changed, and What Comes Next

As of 21 September, the backup has 7 live rules, proven through the rules API, with an 8-case unit suite that has been shown to fail under mutation. Retention is proven by a dry run that removes history, and the restore has passed three consecutive runs against a Postgres drill target. Every rule and drill now has to be accepted against its consumer: the rules API, a driven bad state, or the real target. For Case B, the runner alert and the restart and retry hardening were recorded as open on 16 September. The next experiment is to apply the same drive-it-red method to the runner fleet's own availability signal once that consumer exists.

PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction. This paper is PT-R-2026-025. The primary measurements were taken on 15 to 16 September 2026 (the CI runner case) and 20 to 21 September 2026 (the backup case), against a framing audit of 28 July

2026. Raw artefacts (rule files, unit suites, dry-run and drill outputs, and rules API readings) are retained.