Prometheus Alerting & Scrape Failure Runbook Pack

A senior engineer's incident runbook for scrapes that fail, metrics that vanish, and alerts that misbehave. DevOps AI ToolKit · devopsaitoolkit.com

When a dashboard goes blank or an alert fires (or *doesn't*), the fault is almost never Prometheus's core engine — it's the path between Prometheus and a target: a down endpoint, a wrong `metrics_path`, a relabel rule that dropped the target, a query that timed out, or an Alertmanager route that swallowed the notification. Work from the target outward: confirm what Prometheus *thinks* it's scraping (Status pages and the `up` metric), then walk toward service discovery, PromQL, and delivery. Most of these checks are read-only and safe to run mid-incident.

> Assumes a standard Prometheus + Alertmanager deployment (binaries or containers). Adjust `promtool` and config paths for your packaging. Reload config with SIGHUP or the reload endpoint — never restart to pick up a rule change if a reload will do.

1. Scrape/Alert Triage Checklist

1. **Start at the UI.** Prometheus **Status → Targets** shows every target's `State` (UP/DOWN), last scrape, duration, and the exact error. This is your ground truth.
2. **Query `up`.** In the expression browser: `up == 0` lists every currently-failing scrape by `job` and `instance`. `count by (job) (up == 0)` gives you blast radius.
3. **Confirm config is loaded.** A change on disk means nothing until reloaded. Check **Status → Runtime & Build**, and that your last `reload` returned 200.
4. **Read the target error verbatim.** "connection refused", "context deadline exceeded", "server returned HTTP 404", and "no such host" each point at a different section below.
5. **Scope it.** One instance down = that host/exporter. A whole `job` down = service discovery, relabeling, or a network/firewall change.
6. **Then chase delivery.** If scrapes are fine but no alert arrived, the problem is rule evaluation (§8/§9) or Alertmanager (§10), not scraping.

2. Target Down / Scrape Timeout

```
up == 0
scrape_duration_seconds > 0.9 * scrape_timeout_seconds   # near-timeout scrapes
```

```
# Reproduce the scrape exactly as Prometheus does, from the Prometheus host
curl -sS -m 10 -o /dev/null -w '%{http_code} %{time_total}s\n' http://node-1:9100/metrics

# Is anything even listening on that port?
curl -sS -v http://node-1:9100/metrics 2>&1 | grep -Ei "refused|timed out|no route|resolve"
```

- **connection refused** → the exporter isn't running or isn't bound to that interface/port. Fix the target, not Prometheus.
- **context deadline exceeded** → the scrape took longer than `scrape_timeout`. A slow exporter (huge payload, blocking collector) or a saturated host. Raising `scrape_timeout` is a stopgap; find the slow collector first.
- **no such host / i/o timeout** → DNS or firewall/security-group change between Prometheus and the target.
- Remember `scrape_timeout` must be ≤ `scrape_interval`; check both in the job config.

3. Metrics Endpoint 404 / 500

```
# 404 usually means wrong path or port — try the common ones
curl -sS -o /dev/null -w '%{http_code}\n' http://node-1:9100/metrics
curl -sS http://node-1:9100/metrics | head -5      # is it Prometheus exposition format?
```

```
scrape_configs:
  - job_name: "api"
    metrics_path: /actuator/prometheus   # NOT the default /metrics for Spring Boot
    scheme: http
    static_configs:
      - targets: ["node-1:8080"]
```

- **404** → wrong `metrics_path` (`/actuator/prometheus`, `/metrics/cadvisor`, custom mount) or wrong port. The Targets page shows the *effective* scrape URL — compare it to what actually serves metrics.
- **500** → the exporter itself is erroring while collecting (a broken collector, a DB it can't reach). Read the *exporter's* logs; Prometheus is only reporting what it received.
- **connection reset / EOF mid-body** → the exporter crashed or OOM'd partway through rendering. Check its memory and logs.

4. Service Discovery Failures

```
promtool check config /etc/prometheus/prometheus.yml     # validates SD + relabel syntax
```

- **Status → Service Discovery** in the UI is the key page: it shows **Discovered Labels** (raw `__meta_*` before relabeling) vs **Target Labels** (what survived). If a target is discovered but missing from Targets, a `relabel_configs` rule dropped it.

```
scrape_configs:
  - job_name: "kubernetes-pods"
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      # keep only pods that opted in
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: "true"
      # rewrite the scrape address to the annotated port
      - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
        action: replace
        regex: "([^:]+)(?::\\d+)?;(\\d+)"
        replacement: "$1:$2"
        target_label: __address__
```

- A `keep` regex that matches nothing → zero targets. A `drop` that's too broad → everything vanishes. Compare Discovered vs Target labels to see exactly which rule fired.
- `__address__`, `__scheme__`, `__metrics_path__` are the special labels that set *how* Prometheus scrapes. Rewriting `__address__` wrong (bad regex) silently produces an unscrapeable target.
- `relabel_configs` runs on the *target* (before scrape). Don't confuse it with `metric_relabel_configs`, which runs on *samples* after scrape — see §5.

---

5. Missing Labels & Label Joins

```
# Vector matching: attach the "team" label from a metadata metric onto request rate
sum by (instance) (rate(http_requests_total[5m]))
  * on (instance) group_left(team) node_meta
```

```
scrape_configs:
  - job_name: "pushproxy"
    honor_labels: true        # let the target's own job/instance labels win
    static_configs:
      - targets: ["node-1:9091"]
```

- `on (...)` / `ignoring (...)` control which labels are matched for a binary op. A join fails with "many-to-many matching not allowed" when the match set isn't unique — narrow it with `on` or use `group_left` / `group_right`.
- `group_left(<labels>)` pulls extra labels from the right side onto a many-to-one match. This is how you enrich metrics with metadata (team, env, region).
- `honor_labels: true` tells Prometheus *not* to overwrite a target's exported `job` / `instance` labels with the scrape config's — essential for pushgateways and federation. With it `false` (default), Prometheus wins and your pushed labels get clobbered.
- Missing labels after a `metric_relabel_configs` `labeldrop` / `labelkeep`? That's post-scrape sample surgery — check that section of the job.

---

6. PromQL Query Timeout & Too Many Samples

```
# EXPENSIVE — an unbounded high-cardinality regex over a long range
sum(rate({__name__=~".+"}[1h]))
```

- **"query timed out in expression evaluation"** → the query exceeded `--query.timeout` (default 2m). Usually a wide range × high cardinality. Narrow the range, add label selectors, or pre-aggregate with a recording rule.
- **"query processing would load too many samples into memory"** → you hit `--query.max-samples` (default 50,000,000). Same fix: constrain the selector, shorten `[range]`, or aggregate earlier.
- Symptoms of an expensive query: a dashboard panel spins, then errors; other queries slow down. Use recording rules to precompute the heavy aggregation:

```
groups:
  - name: precompute
    rules:
      - record: job:http_requests:rate5m
        expr: sum by (job) (rate(http_requests_total[5m]))
```

- Prefer a recording rule for anything a dashboard hits repeatedly — it turns an O(series) query into a single cheap lookup.

---

7. High Cardinality

```
# Top 10 metric names by series count (from the meta metrics)
topk(10, count by (__name__)({__name__=~".+"}))

# How many series is one job producing?
count by (job) ({job="api"})
```

```
# TSDB head stats: series count and the label churn driving it
curl -sS http://localhost:9090/api/v1/status/tsdb | \
  python3 -c 'import sys,json;d=json.load(sys.stdin)["data"];
print("head series:",d["headStats"]["numSeries"]);
[print(x["name"],x["value"]) for x in d["seriesCountByMetricName"][:10]]'
```

- High cardinality = a label with unbounded values (user IDs, full URLs, request UUIDs) multiplying series. It's the #1 cause of Prometheus OOM and slow queries.
- **Status → TSDB Status** page shows top metric names and top label-value counts — go straight there.
- Fix at the source (stop emitting the bad label) or drop it at ingest with `metric_relabel_configs`:

```
metric_relabel_configs:
  - source_labels: [__name__]
    regex: "go_gc_duration_seconds.*"   # example: drop noisy series
    action: drop
  - regex: "id|uuid|path"               # drop offending labels
    action: labeldrop
```

- `metric_relabel_configs` runs *after* the scrape, so it trims what's stored without touching discovery.

8. Alerts Not Firing / Stuck Pending

```
ALERTS{alertstate="pending"}    # firing soon, still inside its "for:" window
ALERTS{alertstate="firing"}
```

```
groups:
  - name: availability
    rules:
      - alert: InstanceDown
        expr: up == 0
        for: 5m            # must be TRUE continuously for 5m before firing
        labels:
          severity: critical
        annotations:
          summary: "{{ $labels.instance }} down"
```

- **Status → Rules** shows each rule's `State` (inactive/pending/firing), its last evaluation time, and any eval error. Start here.
- **Stuck in `pending`** → the condition is true but hasn't held for the full `for:` duration yet, *or* it keeps flapping true/false and the `for:` timer resets each time it goes false. Evaluate the raw `expr` in the browser and watch it.
- **Never fires at all** → run the `expr` manually; if it returns *no data*, the alert can't fire. A common trap: alerting on a metric that disappears when the target goes down (use `up == 0` or `absent()` instead).
- **`rule evaluation` errors** on the Rules page (e.g. bad function, missing metric) stop the rule cold. `promtool check rules /etc/prometheus/rules/*.yml` catches syntax before reload.

9. Alerts Firing Too Often / Flapping

- Flapping = the `expr` crosses the threshold repeatedly. Fixes live in three places:

```
# 1) Add a "for:" so brief spikes don't page
- alert: HighErrorRate
  expr: job:http_errors:rate5m / job:http_requests:rate5m > 0.05
  for: 10m

# 2) Add hysteresis / smoothing to the expr (rate over a longer window)
#    and compare a 5m rate, not an instantaneous value.
```

```
# 3) Alertmanager: control notification cadence, not firing
route:
  group_by: ["alertname", "cluster"]
  group_wait: 30s          # buffer before the first notification for a new group
  group_interval: 5m       # wait before sending an update for a changed group
  repeat_interval: 4h      # re-send an unchanged, still-firing alert this often
```

- **`for:`** stops transient blips from paging — the single highest-leverage fix for noise.
- **`repeat_interval`** too low (e.g. 5m) re-pages a still-firing alert relentlessly. Raise it to hours for anything non-critical.
- **`group_interval`** vs **`group_wait`**: `group_wait` delays the *first* page for a brand-new group; `group_interval` throttles follow-up notifications when the group's contents change. Tune both to batch related alerts.
- Threshold too tight against normal variance → widen it or alert on a rate/ratio instead of a raw gauge.

10. Alertmanager Notification Failures & Remote-Write

```
promtool check config /etc/alertmanager/alertmanager.yml   # (or amtool check-config)
amtool config routes test --config.file=/etc/alertmanager/alertmanager.yml \
  severity=critical alertname=InstanceDown          # which receiver would this hit?
amtool alert query --alertmanager.url=http://localhost:9093    # what AM currently holds
```

```
# Is Prometheus even reaching Alertmanager?
prometheus_notifications_dropped_total > 0
prometheus_notifications_errors_total
```

- Prometheus **Status → Runtime** and the `/alertmanagers` API show whether Prometheus has discovered a live Alertmanager. No Alertmanager = alerts fire in Prometheus but nobody is notified.
- **Silences and inhibition** are the silent killers: an over-broad silence or an inhibit rule can suppress a real page. `amtool silence query` lists active silences.
- Receiver errors (bad Slack webhook, SMTP auth, expired PagerDuty key) show in Alertmanager's own logs — grep for the receiver name.

Remote-write (if you ship samples to Thanos/Cortex/Mimir/a managed backend):

```
prometheus_remote_storage_samples_failed_total > 0
prometheus_remote_storage_samples_pending          # queue backing up = endpoint slow/down
prometheus_remote_storage_shards
```

```
remote_write:
  - url: https://metrics.example.com/api/v1/write
    queue_config:
      capacity: 10000
      max_shards: 200
      max_samples_per_send: 2000
```

- Pending samples climbing + failed samples > 0 → the remote endpoint is down, throttling (429), or unauthenticated. Prometheus buffers on disk (WAL) up to a point, then drops. Fix the endpoint or the auth; scaling `max_shards` only helps if the endpoint can actually keep up.

11. Decision Tree

```
Dashboard blank or alert wrong?
 ├─ Status→Targets shows DOWN? ── yes
 │    ├─ "connection refused"        → exporter down / wrong port (§2)
 │    ├─ "context deadline exceeded" → slow exporter / scrape_timeout (§2)
 │    ├─ HTTP 404                    → wrong metrics_path or port (§3)
 │    └─ HTTP 500 / reset            → exporter erroring; read ITS logs (§3)
 └─ Targets look UP, but data/alerts wrong
      ├─ target missing entirely     → service discovery / relabel dropped it (§4)
      ├─ metrics present, labels wrong → on/group_left / honor_labels (§5)
      ├─ query errors/timeouts       → range × cardinality; recording rule (§6/§7)
      ├─ alert stuck pending/never fires → for: window / expr returns no data (§8)
      ├─ alert pages constantly      → for:, thresholds, repeat_interval (§9)
      └─ alert fires but no notification → Alertmanager route/silence/receiver (§10)
```

12. Copy/Paste One-Shot Triage

```
PROM=http://localhost:9090
echo "== failing scrapes ==";   curl -sS "$PROM/api/v1/query?query=up==0" | python3 -c 'import sys,json;
[print(r["metric"].get("job"),r["metric"].get("instance")) for r in json.load(sys.stdin)["data"]["result"]]'
echo "== active alerts ==";     curl -sS "$PROM/api/v1/alerts" | python3 -c 'import sys,json;
[print(a["state"],a["labels"].get("alertname"),a["labels"].get("instance","")) for a in json.load(sys.stdin)["data"]["alerts"]]'
echo "== rule eval errors =="; curl -sS "$PROM/api/v1/rules" | python3 -c 'import sys,json;g=json.load(sys.stdin)["data"]["groups"];
[print(r["name"],r.get("lastError")) for x in g for r in x["rules"] if r.get("lastError")]'
echo "== top series ==";        curl -sS "$PROM/api/v1/status/tsdb" | python3 -c 'import sys,json;[print(x["name"],x["value"]) for x in
json.load(sys.stdin)["data"]["seriesCountByMetricName"][:5]]'
echo "== config valid? ==";     promtool check config /etc/prometheus/prometheus.yml >/dev/null && echo OK || echo FAILED
echo "== reload config ==";     curl -sS -XPOST "$PROM/-/reload" -o /dev/null -w 'reload: %{http_code}\n'
```

13. Incident Notes Template

```
INCIDENT: Prometheus Scrape / Alerting Failure
Started (UTC):          ____   Detected by: ____   Sev: ____
Symptom:                [ ] blank dashboard  [ ] no alert  [ ] alert storm  [ ] wrong data
Scope (up==0):          job(s) ____   instance(s) ____   ( __ of __ targets )
Target error (verbatim): _____________________________  (Status→Targets)
Endpoint check:         metrics_path ____  port ____  curl code ____  time ____s
Service discovery:      [ ] discovered but dropped by relabel rule: ____
Query/cardinality:      [ ] timeout  [ ] max-samples  top metric: ____ (__ series)
Alert rule:             name ____  state [ ]pending [ ]firing  for: ____  eval err: ____
Notification path:      [ ] AM reachable  [ ] silence active: ____  receiver err: ____
Remote-write:           samples_failed ____  pending ____
Action taken:           (config change / reload / exporter restart + who + when)
Result:                 [ ] recovered @ ____   [ ] escalated to ____
Root cause:             ____
Follow-ups:             (scrape_timeout, recording rules, cardinality caps, alert tuning) ____
```
