ITI8801 · Cloud Architectures and DevOps

Practice 10 · Assignment 9: metrics and an alert

Siim Rebane · TalTech · Autumn 2026

What you build

Your Assignment 8 project, changed in three ways:

Part	Written by you
metrics	your service counts its requests and measures how long they take, and reports both at `GET /metrics`
scraping	the course Prometheus collects those numbers from your staging pods
two alert rules	one that fires when staging has no ready pod, one that would fire when staging gets slow; both deployed by your chart with staging

Then a drill: you take staging down on purpose, the alert arrives, you bring staging back, the alert clears. The validator watches the whole loop.

The two environments, their host names, the pipeline, the chart and the Assignment 5 API stay as they are in Assignment 8.

The metrics

Your service gets one new endpoint:

```
GET /metrics        →  200   text/plain, Prometheus text format
```

Requirement	Detail
a **counter** `http_requests_total`	one count per request handled, with a label `status` (the HTTP status code) and a label `path` (the route as written in your code, `/items/{id}`, never the actual id)
a **histogram** `http_request_duration_seconds`	the time each request took, in seconds; the default buckets of your client library are fine
how	the Prometheus client library of your language; a middleware or a handler wrapper counts every request, including `/metrics` itself if you like
both environments	the endpoint exists in the image, so dev has it too; only staging is checked

The series carry at most a few dozen label values: a label whose value is an id, a user or a random string makes one series per value, and the course Prometheus drops a pod that exposes more than 1000 series.

The scraping

The course runs one Prometheus for the whole cluster. It collects from a pod when the pod says so:

Requirement	Detail
two annotations on the **pods** of staging	`prometheus.io/scrape: "true"` and `prometheus.io/port: "<the port your service listens on>"`; `prometheus.io/path` only if your endpoint is not at `/metrics`
where they are written	in the Deployment template's pod template (`spec.template.metadata.annotations`), so that every pod the Deployment makes carries them; an annotation on the Deployment itself scrapes nothing
through the chart	the annotations come from your chart; whether dev has them too is your choice

Prometheus visits each annotated pod every 30 seconds. Your series appear with the labels `namespace` (yours), `pod` and `app`.

The course Prometheus's own page at `http://prometheus.k8s.digivader.com/` and Grafana at `http://grafana.k8s.digivader.com/` show what it holds. Both are for looking, nothing in them is checked.

The alert rules

A `PrometheusRule` object in your namespace, made by your chart for staging (a template that renders only when the values say staging, like the HPA and the PDB):

Rule	Requirements
alert `StagingDown`	`expr`: the staging Deployment has zero available replicas, from `kube_deployment_status_replicas_available{namespace="<yours>", deployment="<staging's Deployment>"}`; `for` between 30 seconds and 2 minutes; a label `severity: critical`; an annotation `summary` that says what is wrong in words
alert `StagingSlow`	`expr`: the 95th percentile of your request duration over the last 5 minutes is above a threshold you choose, computed with `histogram_quantile` over `http_request_duration_seconds_bucket` of your namespace; `for` of 2 minutes; `severity: warning`; a `summary`

`StagingDown` must fire in the drill. `StagingSlow` must load and parse; it is not made to fire. A rule whose expression does not parse is refused by the cluster when the chart is deployed.

Dev has no rules: the same chart with dev's values leaves the template out.

The drill

With everything above in place and the pipeline green:

1. `kubectl scale deployment <staging's Deployment> --replicas=0 -n <your namespace>` — staging answers nothing. The HPA stays quiet at zero replicas.
2. Within a few minutes `StagingDown` is **firing**: the course Alertmanager shows it with your namespace.
3. `kubectl scale deployment <staging's Deployment> --replicas=2 -n <your namespace>` — the pods come back, the HPA takes over again.
4. A few minutes later the alert is gone.

On the course cluster each of the two waits takes a minute or two; the validator allows five.

The end state, checked

- staging's pods carry the two annotations, Prometheus scrapes them (`up` is 1 for each) and holds `http_requests_total` and `http_request_duration_seconds_bucket` with your namespace label
- after 20 requests to staging's host the counter has grown by 20 for the paths they hit
- a `PrometheusRule` in your namespace, owned by the staging release, holds `StagingDown` and `StagingSlow` as specified; dev has none
- the chart in the repository holds the rule template and the annotations; no credential in the repository
- the drill: scaled to zero, `StagingDown` is active in Alertmanager within 5 minutes; scaled back, it is resolved within 5 minutes and the Assignment 5 API answers at staging's host again

The validator

`https://iti8801.digivader.com/lab9-validator/`: your student code and your GitLab project.

Box	Green when
Metrics	each staging pod is a scrape target with `up` 1; `http_requests_total` has a `status` and a `path` label and no label with more than 50 values; `http_request_duration_seconds` has buckets, a sum and a count; 20 requests to staging's host raise the counter by 20 within one scrape interval
Rules	a `PrometheusRule` in your namespace with Helm's release annotation of staging; `StagingDown` and `StagingSlow` with the expressions, `for`, labels and annotations above; Prometheus lists both rules; no `PrometheusRule` belongs to dev
Drill, down	the page says *scale staging to zero now* and waits up to 5 minutes for `StagingDown` with your namespace to be active in Alertmanager
Drill, up	the page says *scale staging back now* and waits up to 5 minutes for the alert to be resolved and the Assignment 5 API to answer at staging's host
Repository	the chart holds the rule template and the pod annotations; the commit staging runs has a green pipeline; no Secret manifest, kubeconfig or token in the repository

The validator reads the course Prometheus and Alertmanager and your namespace; it creates and changes nothing in your namespace or your project. The scaling in the drill is done by you. One check per namespace at a time; the drill boxes wait for you, so start a check when you are at your keyboard.

What to submit, and when

- The validator's code of a check that is **PASSED**. Paste it into Moodle.
- Deadline: Sunday 15 November 23:59. Late: Sunday 22 November, 50 %.
- A check is stored with its result: what you change in the project or the namespace afterwards does not change it.

Walkthrough

The steps in order, with the commands and what each should print. The outputs come from the test namespace `000007test` and its project, a Python service. Replace what is yours (`<…>`).

Step 0 · where you start

The project as Assignment 8 left it: the service, the chart, two values files, the pipeline that deploys `main` to dev and a `v*` tag to staging. Three things get added:

```
main.py                          ← the middleware and GET /metrics
chart/templates/deployment.yaml ← two annotations on the pod template
chart/templates/alerts.yaml     ← new: the PrometheusRule
chart/values.yaml               ← alerts: enabled: false
values-staging.yaml             ← alerts: enabled: true
```

Nothing is installed on your machine for this assignment: Prometheus, Alertmanager and Grafana run on the course cluster. `kubectl` and `helm` are the ones of practice 8.

Step 1 · count and time every request

A middleware runs around every request. In the test project's FastAPI service:

```python
from prometheus_client import CONTENT_TYPE_LATEST, Counter, Histogram, generate_latest

REQUESTS = Counter("http_requests", "Requests handled", ["path", "status"])
DURATION = Histogram("http_request_duration_seconds", "Time a request took", ["path"])

@app.middleware("http")
async def measure(request: Request, call_next):
    started = time.perf_counter()
    response = await call_next(request)
    route = request.scope.get("route")
    path = route.path if route is not None else "unmatched"
    REQUESTS.labels(path=path, status=str(response.status_code)).inc()
    DURATION.labels(path=path).observe(time.perf_counter() - started)
    return response

@app.get("/metrics")
def metrics():
    return Response(content=generate_latest(), media_type=CONTENT_TYPE_LATEST)
```

`Counter("http_requests", …)` becomes `http_requests_total`: the library adds the suffix. `path` is the route the framework matched (`/api/boardgames/{gid}`), never `request.url.path`: that one holds the id and makes a series per id. A request that matched no route gets a fixed word.

`requirements.txt` gets `prometheus-client==0.26.0`.

Step 1 · the same in other languages

Language	Library	Where the route template comes from
Python, FastAPI / Flask	`prometheus_client`	`request.scope["route"].path` / `request.url_rule.rule`
Node, Express	`prom-client`	`req.route.path` after the handler ran
Go, net/http or chi	`github.com/prometheus/client_golang`	the pattern you registered; `promhttp.Handler()` serves `/metrics`
Java, Spring Boot	Micrometer with the Prometheus registry; the endpoint is `/actuator/prometheus`, so the pod needs `prometheus.io/path` too	a `Counter` and a `Timer` of your own with the two names, in a filter or interceptor; the `Timer` publishes buckets only with `publishPercentileHistogram()`

Whatever the library: the two names and the two labels are fixed, the buckets are the library's defaults, `/metrics` answers `200` with the text format.

Step 2 · run it before you push

Add a line for `GET /metrics` to `tests/smoke.sh`, build the image and run it next to a database as the pipeline's `test` job does. Then look at the answer:

```
$ curl -s http://127.0.0.1:8000/metrics | grep '^http_' | grep -v bucket
http_requests_total{path="/health",status="200"} 2.0
http_requests_total{path="/info",status="200"} 1.0
http_requests_total{path="/metrics",status="200"} 1.0
http_requests_total{path="/api/boardgames",status="201"} 1.0
http_requests_total{path="/api/boardgames/{gid}",status="200"} 2.0
http_requests_total{path="/api/boardgames",status="200"} 4.0
http_requests_total{path="/api/boardgames",status="400"} 1.0
http_requests_total{path="/api/boardgames/{gid}",status="204"} 1.0
http_requests_total{path="/api/boardgames/{gid}",status="404"} 4.0
http_requests_total{path="unmatched",status="404"} 1.0
http_request_duration_seconds_count{path="/health"} 2.0
http_request_duration_seconds_sum{path="/health"} 0.003713637590408325
…
```

Without the `grep -v` the answer holds 15 `_bucket` lines per path, with `le` from `0.005` to `+Inf`. Every path label is a route, every status a number: that is what the check reads. Commit, push; the pipeline builds, tests, scans and deploys to dev.

Step 3 · the annotations

In `chart/templates/deployment.yaml`, next to the checksum annotation of practice 8, on the **pod template**:

```
template:
  metadata:
    labels:
      app: api-{{ .Release.Name }}
    annotations:
      checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") . | sha256sum }}
      prometheus.io/scrape: "true"
      prometheus.io/port: "8000"
  spec:
    containers:
    - name: service
```

The port is the one the container listens on, not the Service's. Both values are strings: `"true"` and `"8000"` in quotes, an annotation is never a number.

The test project adds them for dev too: a dev that is scraped costs nothing and shows up in Grafana as well.

Step 4 · the rule template

`chart/templates/alerts.yaml`, under a condition like the HPA's; the first rule:

```
{{- if .Values.alerts.enabled }}
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: api-{{ .Release.Name }}
spec:
  groups:
  - name: api-{{ .Release.Name }}
    rules:
    - alert: StagingDown
      expr: kube_deployment_status_replicas_available{namespace="{{ .Release.Namespace }}", deployment="api-{{ .Release.Name }}"} == 0
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: "{{ .Release.Name }} has no ready pod in {{ .Release.Namespace }}"
```

`.Release.Namespace` is the namespace Helm installs into, so the expression names your namespace without you writing it. `deployment="api-{{ .Release.Name }}"` must be the exact name of staging's Deployment, the one `kubectl get deployment` prints. The template's own `{{ }}` and the expression's `{ }` do not clash: Helm replaces its pairs first, what is left is the PromQL.

Step 4 · the second rule

The same file, continued after the first rule:

```
    - alert: StagingSlow
      expr: histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket{namespace="{{ .Release.Namespace }}",
              app="api-{{ .Release.Name }}"}[5m]))) > {{ .Values.alerts.p95Seconds }}
      for: 2m
      labels:
        severity: warning
      annotations:
        summary: "{{ .Release.Name }} in {{ .Release.Namespace }}:
          95 % of requests take more than {{ .Values.alerts.p95Seconds }} s"
{{- end }}
```

A value may continue on the next line, indented deeper: YAML joins the lines with a space, and the cluster prints the rule back the same way. `app="api-{{ .Release.Name }}"` picks staging's pods: `app` is the label Prometheus copies from the pod, the one the Service selects by. The threshold comes from the values, so staging's `p95Seconds` can differ from a future environment's without a change to the template. The two rules are one group; Prometheus evaluates a group every 30 s.

Step 4 · the values

```
# chart/values.yaml, added at the end
alerts:
  enabled: false           # staging only: a PrometheusRule for the course Prometheus
  p95Seconds: 0.5
```

```
# values-staging.yaml, added at the end
alerts:
  enabled: true
```

`values-dev.yaml` stays as it is: `enabled` is false by default, dev renders no rule. Raise the chart's `version` in `Chart.yaml` if you keep it in step with changes; Helm does not require it.

Step 5 · render and read before you deploy

```
$ helm template staging chart -f values-staging.yaml --set image.tag=342f48c | grep -n "namespace="
191:      expr: kube_deployment_status_replicas_available{namespace="default", deployment="api-staging"} == 0
198:      expr: histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket{namespace="default", app
```

`helm template` contacts no cluster and knows no namespace: without `-n` it renders `.Release.Namespace` as `default`. The pipeline's `helm upgrade` takes the namespace from the kubeconfig, so the deployed rule is right; to read what it will deploy, give the namespace:

```
$ helm template staging chart -n 000007test -f values-staging.yaml --set image.tag=342f48c | grep -n -A2 "prometheus.io\|kind: PrometheusRule\|alert:"
64:        prometheus.io/scrape: "true"
65:        prometheus.io/port: "8000"
--
183:kind: PrometheusRule
184-metadata:
185-  name: api-staging
--
190:    - alert: StagingDown
191-      expr: kube_deployment_status_replicas_available{namespace="000007test", deployment="api-staging"} == 0
192-      for: 1m
--
197:    - alert: StagingSlow
198-      expr: histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket{namespace="000007test", app="api-staging"}[5m]))) > 0.5
199-      for: 2m
$ helm template dev chart -n 000007test -f values-dev.yaml --set image.tag=342f48c | grep -c PrometheusRule
0
```

Step 6 · push, tag, the pipeline deploys staging

Commit the chart changes with the service change, push, wait for the pipeline on `main` to be green, then tag that commit with a tag that matches `v*`, as in Assignment 7. The tag pipeline's `deploy-staging` job:

```
$ helm upgrade --install $ENV chart -f values-$ENV.yaml --set image.tag=$CI_COMMIT_SHORT_SHA --wait --timeout 2m
Release "staging" does not exist. Installing it now.
NAME: staging
LAST DEPLOYED: Fri Oct 30 17:13:41 2026
NAMESPACE: 000007test
STATUS: deployed
REVISION: 1
Job succeeded
```

The test project's staging was installed fresh for this run; yours is a release since Assignment 8 and the first line reads `Release "staging" has been upgraded`, with the next revision number. Then, from your machine (`api` is the Assignment 6 Deployment the test namespace still has; yours shows dev and staging):

```
$ kubectl get deploy,prometheusrule
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api            2/2     2            2           27d
deployment.apps/api-dev        1/1     1            1           13d
deployment.apps/api-staging    2/2     2            2           2d5h

NAME                                              AGE
prometheusrule.monitoring.coreos.com/api-staging   2d5h
$ kubectl get pods -l app=api-staging -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}\
    {.metadata.annotations.prometheus\.io/scrape}{"\t"}{.metadata.annotations.prometheus\.io/port}{"\n"}{end}'
api-staging-6c94d5d5b6-ffzhz    true    8000
api-staging-6c94d5d5b6-qw7j5    true    8000
```

Step 6 · the rule as the cluster holds it

```
$ kubectl get prometheusrule api-staging -o yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  annotations:
    meta.helm.sh/release-name: staging
    meta.helm.sh/release-namespace: 000007test
    prometheus-operator-validated: "true"
  labels:
    app.kubernetes.io/managed-by: Helm
  name: api-staging
  namespace: 000007test
spec:
  groups:
  - name: api-staging
    rules:
    - alert: StagingDown
      annotations:
        summary: staging has no ready pod in 000007test
      expr: kube_deployment_status_replicas_available{namespace="000007test", deployment="api-staging"}
        == 0
      for: 1m
      labels:
        severity: critical
    …
```

`prometheus-operator-validated: "true"` says the cluster parsed both expressions. The release annotations are what the check reads to tell staging's rule from dev's.

Step 6 · a rule that does not parse

A rule with a broken expression never reaches the cluster: the operator checks every `PrometheusRule` on the way in and refuses it with the parser's message. Here a `)` too many:

```
$ kubectl apply -f bad-rule.yaml
The  "prometheusrules" is invalid: : 5:11: group "test", rule 1, "StagingDown": could not parse expression: 1:98: parse error: unexpected right parenthesis ')'
```

Through Helm the same rule fails the `deploy-staging` job, with less detail. Here it was tried as a throw-away release in the test namespace, with only the rule in the chart:

```
$ helm upgrade --install badrule chart --wait --timeout 2m
Error: UPGRADE FAILED: failed to create resource: server-side apply failed for object 000007test/badrule-test
  monitoring.coreos.com/v1, Kind=PrometheusRule: admission webhook "prometheusrulevalidate.monitoring.coreos.com"
  denied the request: Rules are not valid
```

`helm list` then shows the release as `failed`; for staging, the next green tag pipeline replaces it. To see which character is wrong, render the rule and apply it by hand: `helm template … -n <ns> -s templates/alerts.yaml | kubectl apply -f -` prints the parser's message above.

Step 7 · what the pod exposes

Prometheus reads the pod's port inside the cluster. You can read the same page through a port-forward:

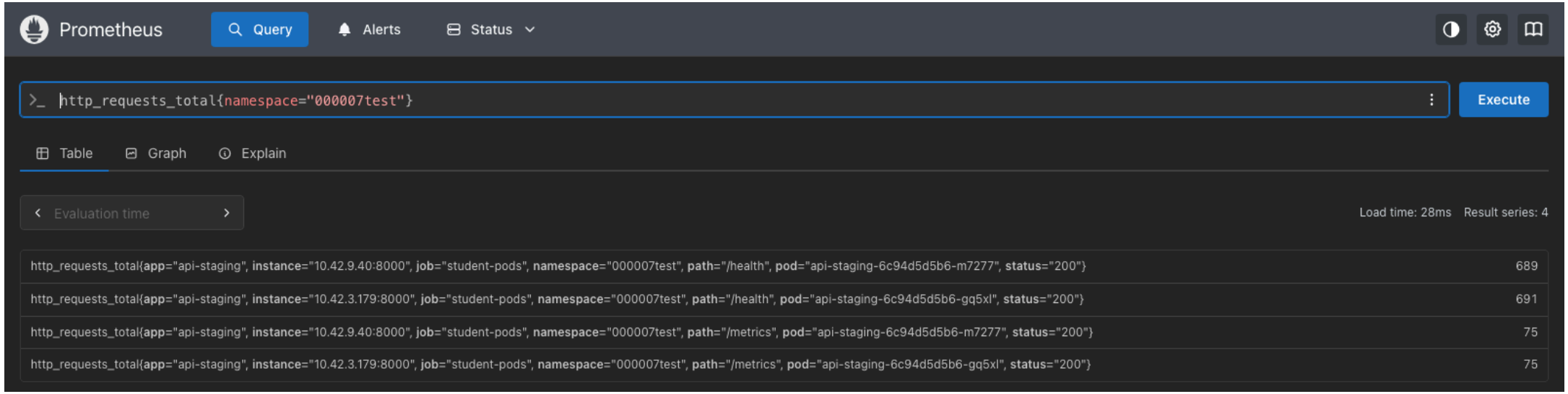
```
$ kubectl port-forward deployment/api-staging 8000:8000
Forwarding from 127.0.0.1:8000 -> 8000
Forwarding from [::1]:8000 -> 8000
```

in a second terminal:

```
$ curl -s http://127.0.0.1:8000/metrics | grep '^http_requests_total'
http_requests_total{path="/health",status="200"} 56739.0
http_requests_total{path="/metrics",status="200"} 6303.0
```

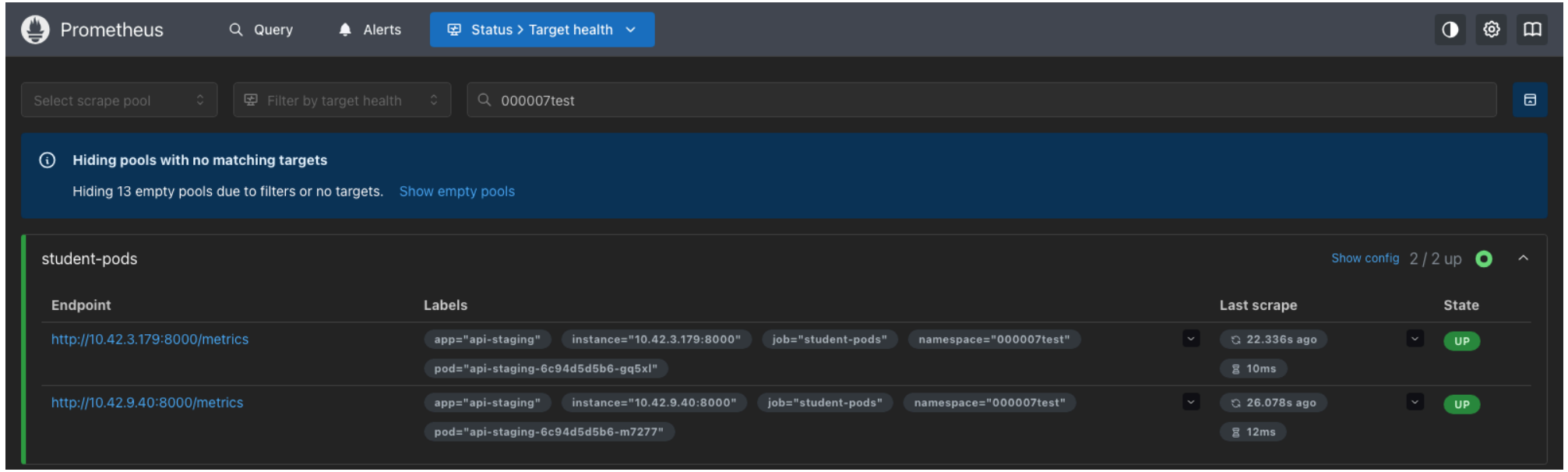
Two days after the deploy, nobody has called the API of this pod, and still it counts: Prometheus reads `/metrics` every 30 s, the kubelet calls `/health` for the readiness probe every 5 s and the liveness probe every 10 s, nine times as often. This is one pod of two; Prometheus holds both and `sum` adds them.

Step 8 · see it in Prometheus



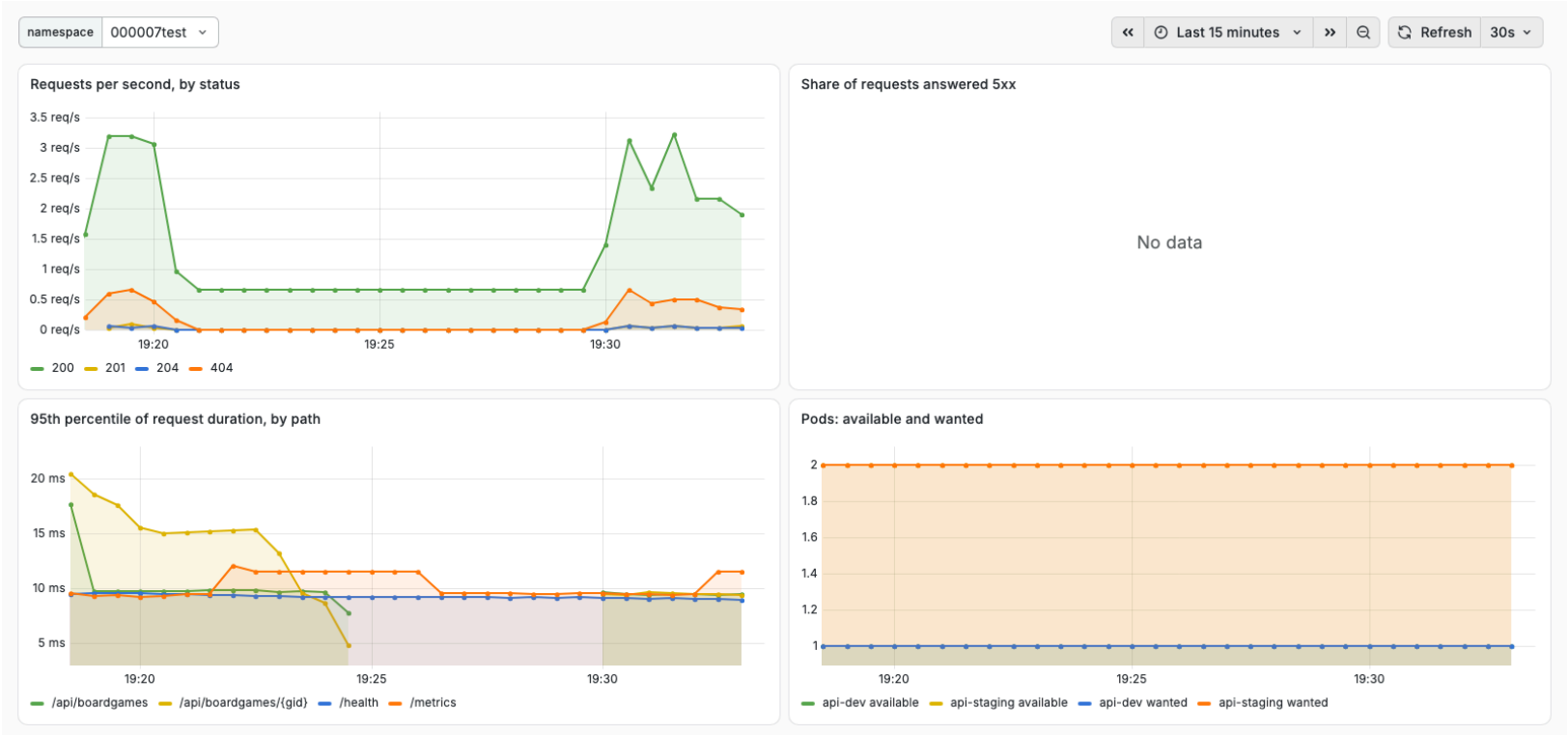
`http://prometheus.k8s.digivader.com/`, the query `http_requests_total{namespace="<your namespace>"}`, *Execute*: one row per series, both pods, the labels Prometheus added (`instance`, `job`, `namespace`, `pod`, `app`) next to yours. *Graph* draws the same over the last hour. The queries of the lecture work here with your namespace: `sum by (status) (rate(http_requests_total{namespace="<ns>"}[1m]))`, `up{namespace="<ns>"}`.

Step 8 · the targets



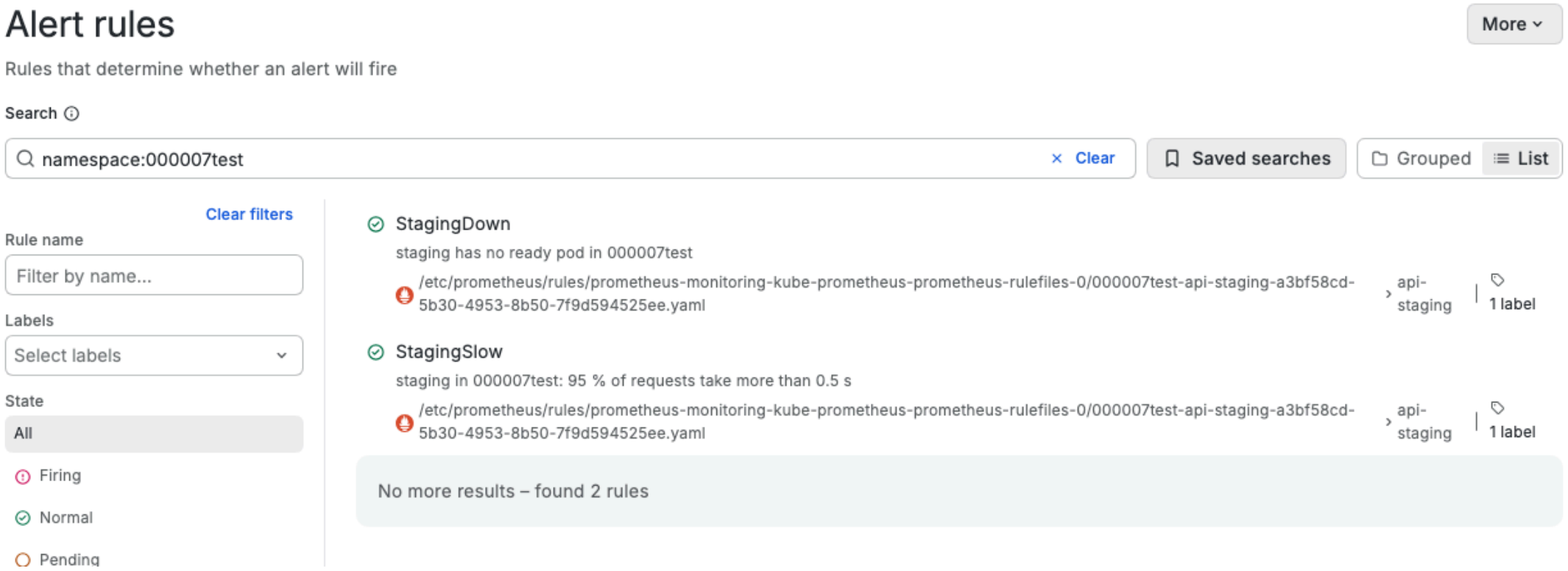
Status → Target health, your namespace in the search box: one line per pod, the address Prometheus reads, when it last did, `UP`. A pod that is missing here is not annotated; a pod that is `DOWN` is annotated and does not answer: the error text on its line says why.

Step 9 · see it in Grafana



`http://grafana.k8s.digivader.com/` → Dashboards → *ITI8801 · a service in staging*; pick your namespace at the top. The four panels are queries over `http_requests_total`, `http_request_duration_seconds_bucket` and `kube_deployment_status_replicas_available` with your namespace filled in; the pods panel shows staging's available and wanted replicas. A namespace missing from the list has no scraped series yet.

Step 10 · the rules in Grafana



Alerting → Alert rules, search `namespace:<your namespace>`, the *List* view. Both rules with their summary, the state *Normal* on the left. The file name is where the course Prometheus keeps a copy of your `PrometheusRule`; the operator rewrites it within a minute of a change.

Step 11 · the drill: down

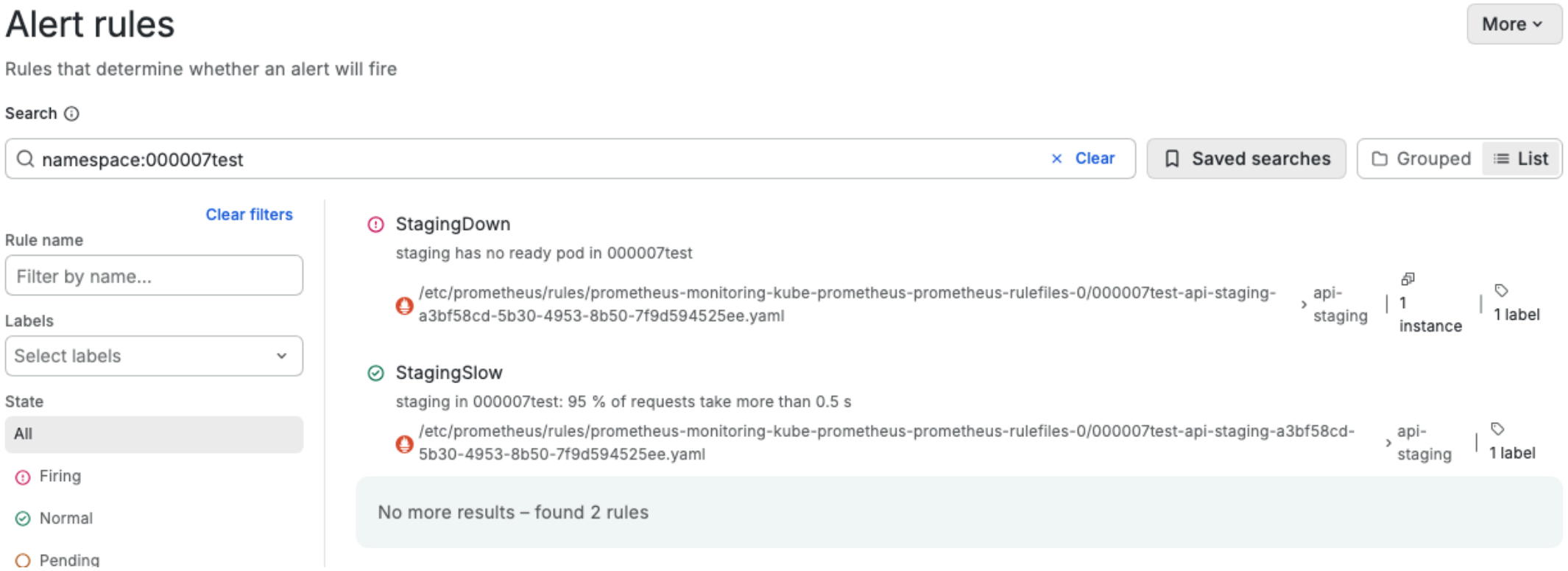
Everything green and nothing running against staging. Times are from a run in the test namespace, UTC:

```
18:03:30 $ kubectl scale deployment api-staging --replicas=0
deployment.apps/api-staging scaled
18:04:15 $ kubectl get deployment api-staging
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
api-staging   0/0     0            0           50m
```

Staging's host answers nothing now. Every 30 s kube-state-metrics reports zero available replicas, every 30 s the rule is evaluated: the first true evaluation makes the rule *Pending*, and when it has been true for the rule's `for` (1 minute here), *Firing*. In Grafana's rule list the icon on the left changes: a ring for pending, a red mark for firing. The alert reached Alertmanager at 18:05:05, 95 seconds after the scale.

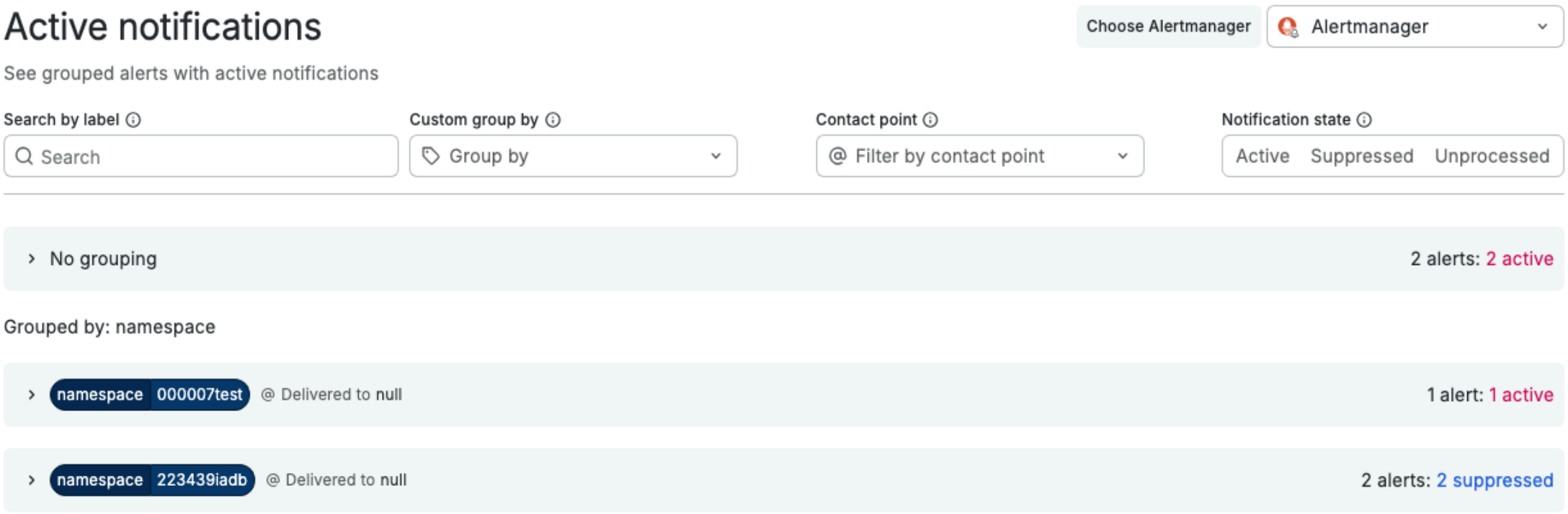
The HPA does not interfere: an autoscaler leaves a Deployment at zero alone.

Step 11 · the drill: firing



At 18:05:51: `StagingDown` firing with *1 instance*, `StagingSlow` still normal. The *State* filter on the left shows only the firing ones when the list is long.

Step 11 · the drill: in Alertmanager



Alerting → Active notifications, with *Alertmanager* chosen at the top right instead of *Grafana*: the alerts the course Alertmanager holds right now, grouped by namespace, yours with *1 active*. This is the page the validator's drill box reads, through Alertmanager's API. The suppressed ones of other namespaces are the cluster's own informational alerts.

Step 11 · the drill: back

```
18:07:04 $ kubectl scale deployment api-staging --replicas=2
deployment.apps/api-staging scaled
18:08:20 $ kubectl get deployment,hpa api-staging
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api-staging   2/2     2            2           54m

NAME                                                REFERENCE                TARGETS        MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/api-staging   Deployment/api-staging   cpu: 22%/60%   2         5         2          54m
18:08:21 $ curl -s -o /dev/null -w '%{http_code}' http://000007test-staging.k8s.digivader.com/health
200
```

Once a pod is ready the expression is false, the rule goes back to *Normal* at the next evaluation, and Alertmanager marks the alert resolved: at 18:08:20 it listed none, 76 seconds after the scale. The HPA shows `<unknown>` for a moment, then takes over at 2 replicas again.

`replicas=2` is the HPA's minimum: scaling to 1 would be raised to 2 by the HPA within a minute anyway.

Step 12 · the validator

- `https://iti8801.digivader.com/lab9-validator/` : your student code and your project.
- Five boxes. The first two and the last read; the two drill boxes wait for you: the page tells you when to scale to zero and when to scale back, and waits up to 5 minutes each time. Start a check with staging at 2 replicas and no alert of yours active.
- Paste the code of the **PASSED** check into Moodle.

```
$ kubectl get prometheusrule,deploy,hpa
$ kubectl get pods -l app=api-staging -o yaml | grep prometheus.io
```

show the rule, the Deployment it names and the annotations the pods carry.

When something does not work

You see	Look at
`/metrics` answers 404 at staging's host	staging runs the tagged commit: the tag was made before the change, or the tag pipeline has not finished; `helm list` shows the revision's time
your pods are not on the targets page	the annotations are on the Deployment, not on its pod template: `kubectl get pod <pod> -o yaml | grep prometheus.io` shows nothing
the pods are on the targets page, `DOWN`	the error on the line: the port is the Service's, not the container's; the service listens on `127.0.0.1` only (it must listen on `0.0.0.0`); the path is not `/metrics`
one series per id, hundreds of them	the `path` label holds the URL instead of the route; the pod is dropped above 1000 series
`Rules are not valid` from `helm upgrade`, the release `failed`	an expression does not parse: render the rule with `helm template -n <ns> -s templates/alerts.yaml` and `kubectl apply` it by hand for the parser's message
the rule renders with `namespace="default"`	`helm template` without `-n`; the pipeline's deploy is unaffected
`StagingDown` never fires at zero replicas	`deployment="…"` against `kubectl get deployment`; the namespace; `for` above 2 minutes
the rule is in `kubectl get prometheusrule` but not in Grafana	wait a minute; `prometheus-operator-validated` missing in the object's annotations
dev got a `PrometheusRule` too	the condition around the template; `alerts.enabled` in `values-dev.yaml` or in `values.yaml`

What stays where

Thing	Lives in
the counting and timing	your service, so in the image: both environments have `/metrics`
the annotations	the chart's Deployment template, on the pod template
the two rules	`chart/templates/alerts.yaml`, rendered for staging by `values-staging.yaml`
the numbers	the course Prometheus, 7 days back; nothing of Prometheus in your repository or namespace
the rule's state and the alert	the course Prometheus evaluates, the course Alertmanager holds the alert; Grafana shows both
the drill	your hands: `kubectl scale`, twice



Questions