ITI8801 · Cloud Architectures and DevOps

Metrics and alerts

Siim Rebane · TalTech · Autumn 2026

What we built

- The boardgames service of assignment 8 in **staging** on the course cluster: a Helm release, two pods behind a Service and an Ingress, a database, an autoscaler, a disruption budget

```
$ helm -n 000007test list
NAME     NAMESPACE    REVISION  UPDATED                                  STATUS    CHART              APP VERSION
staging  000007test   1         2026-10-30 17:13:41.506496878 +0000 UTC deployed  boardgames-0.2.0

$ kubectl get deploy,hpa,pdb -n 000007test
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api-staging    2/2     2            2           2m
NAME                                                  REFERENCE                 TARGETS        MINPODS   MAXPODS   REPLICAS
horizontalpodautoscaler.autoscaling/api-staging   Deployment/api-staging   cpu: 24%/60%   2         5         2
NAME                                        MIN AVAILABLE   ALLOWED DISRUPTIONS
poddisruptionbudget.policy/api-staging    1               1
```

- It answers at `http://000007test-staging.k8s.digivader.com/api/boardgames`
- The question of this lecture: **is it working right now, and how would I know** before someone tells me?

How we find out today

- **Someone tells us.** A user, a teammate, the validator. The service has been broken for as long as it took them to notice and write
- **We look at the logs.** `kubectl logs` shows what happened, one line per event. Useful once we know something is wrong, and we have to be looking
- **We try it.** `curl` the endpoint. It answers now; it says nothing about the last hour, and nothing about the other pod
- All three are **after the fact** and need a person. A service runs 24 hours a day; a person does not

A number over time

- What we would want to see, drawn as a line over the last hour:
  - how many **requests** per second the service handles
  - how many of them **fail**
  - how **long** a request takes
  - how many **pods** are running, and how much memory and CPU they use
- Each of these is a **number** that can be read every few seconds and kept
- A line that drops to zero, or a line that climbs, is visible **before** anyone complains
- Numbers kept over time are called **metrics**; collecting and looking at them is **monitoring**

The four golden signals

Four numbers that describe almost any service. For ours:

Signal	Means	For the boardgames service
traffic	how much work arrives	requests per second
errors	how much of it fails	share of requests answered `5xx`
latency	how long work takes	the time a request takes, at the 95th percentile
saturation	how full the service is	CPU and memory of the pods against their limits

A dashboard with these four, and an alert on two of them, covers most of what goes wrong.

Source: Google, *Site Reliability Engineering* (2016), chapter 6, "Monitoring Distributed Systems".

Three lists that say the same thing

List	Where from	For what	The numbers
four golden signals	Google's SRE book, 2016	a service, as its users see it	traffic, errors, latency, saturation
RED	Tom Wilkie, 2015	a service that handles requests	rate, errors, duration
USE	Brendan Gregg, 2012	a resource: CPU, memory, disk, network	utilisation, saturation, errors

- RED is the golden signals without saturation, named for what a service's own metrics give. Our counter and histogram are exactly rate, errors and duration
- USE is for the thing the service runs on. Those numbers come from the kubelet and the node, not from the service: that is why the fourth panel of a service dashboard is about its pods, not its requests

Logs and metrics

	Logs	Metrics
what	one line per **event**: a request, an error, a start	one **number**, read again every few seconds
size	grows with traffic: 1000 requests, 1000 lines	the same size at 1 request per second and at 1000
question	*what happened to request 4711?*	*how many requests failed in the last 5 minutes?*
tool	a log store, searched by text	a time series database, queried by name and label

Both are needed. This lecture is the numbers; logs come with the incident process.

Prometheus

- **Prometheus** is an open-source monitoring system: a program that collects numbers from services, stores them over time and answers questions about them. It is the standard one for Kubernetes, so its format and its words are the common ones
- What it does, in one loop:
  - every 30 seconds it calls `GET /metrics` of every service it knows and reads the numbers in the answer
  - it stores each number with the time it was read, and keeps them (7 days on the course cluster)
  - it answers questions over the stored numbers, in its own query language
  - it evaluates **alert rules** on them and hands what fires to a second program, **Alertmanager**, which notifies people
- The course runs one Prometheus for the whole cluster, in the namespace `monitoring`: it reads every student's pods that ask to be read, and a few programs that describe the cluster itself
- It has a web page of its own, `http://prometheus.k8s.digivader.com/`, open to read: a query box, the targets, the rules, the alerts. Every Prometheus screenshot in this lecture is from that page

Source: prometheus.io, "Overview".

The programs

Monitoring on the course cluster is one Helm chart, `kube-prometheus-stack`, installed in the namespace `monitoring`. It brings five programs:

Program	Does	Page
Prometheus	calls every target's `/metrics`, stores the numbers, answers queries, evaluates alert rules	`prometheus.k8s.digivader.com`
Alertmanager	receives firing alerts from Prometheus; groups, silences and routes them to receivers	none exposed; Grafana reads its API
Grafana	draws Prometheus's answers as dashboards; shows Prometheus's rules and Alertmanager's alerts	`grafana.k8s.digivader.com`
kube-state-metrics	one pod that turns the cluster's objects into numbers for Prometheus	none
node-exporter	one pod per node that exposes the machine's numbers	none

- The requests between them: Prometheus **pulls** from every target. Prometheus **pushes** firing alerts to Alertmanager. Grafana **reads** Prometheus and Alertmanager while a page is open. Nothing else talks
- Alertmanager is a separate program by the Prometheus project: a Prometheus knows only its own data; grouping, silencing and sending belong in one place, whatever the number of Prometheuses
- Grafana stores no numbers; Alertmanager stores no numbers. Only Prometheus does

What Prometheus defines

Besides the program, Prometheus fixed a handful of definitions. Other monitoring tools adopted them, and they are the words of the rest of this lecture:

Definition	What it fixes
the **data model**	a metric has a name and labels; its values over time form a series of samples
the **text format**	what a service answers at `GET /metrics`; written down in 2020 as the **OpenMetrics** specification, so any tool can implement it
four **metric types**	counter, gauge, histogram, summary: what kind of number a metric is
naming conventions	how a metric's name is built, so that any name can be read without documentation
PromQL	the query language: how to ask for a rate, a sum, a percentile, over any set of series
client libraries	for Python, Go, Java, Node, .NET and others: they keep the numbers in the process and produce the text format correctly

A service that answers the text format can be read by Prometheus, by Grafana's own collector, by Datadog, by the cloud providers' monitoring agents. Our service will answer it; the course Prometheus reads it.

Source: prometheus.io, "Data model", "Metric types", "Exposition formats".

A metric, a label, a sample, a series

- A **metric** has a name: `http_requests_total`
- A **label** is a key and a value attached to it: `path="/api/boardgames"`, `status="200"`. Labels split one metric into several numbers
- A **sample** is one value with the time it was read: `305` at 17:31:52
- A **series** is one name with one set of labels and all its samples over time:

```
http_requests_total{path="/api/boardgames", status="200"}   …, 213, 239, 282, 305
```

- Every different label value makes a **new series**. `status="404"` is another series; `status="201"` a third

`GET /metrics`

- The service exposes its numbers at one endpoint, in a plain text format. What our service answered after a few requests:

```
$ curl http://127.0.0.1:8000/metrics
# HELP http_requests_total Requests handled
# TYPE http_requests_total counter
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
# HELP http_request_duration_seconds Time a request took
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.005",path="/api/boardgames"} 5.0
http_request_duration_seconds_bucket{le="0.01",path="/api/boardgames"} 6.0
…
```

- `# HELP` is a description, `# TYPE` says what kind of number it is, then one line per series: name, labels, the value **right now**
- Nothing is stored here. The endpoint shows the current numbers; keeping them is someone else's job

How a metric is named

Rule	Example
lowercase words joined by `_`; the first word says where the number comes from	`http_requests_total`, `process_cpu_seconds_total`, `python_gc_collections_total`
the unit is in the name, in base units: seconds, bytes, not milliseconds or kilobytes	`http_request_duration_seconds`, `container_memory_working_set_bytes`
a counter ends in `_total`	`http_requests_total`, `process_cpu_seconds_total`
what varies is a label, not part of the name	`http_requests_total{status="404"}`, not `http_404_requests_total`
a histogram is one name that becomes three series	`http_request_duration_seconds_bucket`, `_sum`, `_count`

- Conventions, not rules the format enforces. But every dashboard, every library and every alert in this lecture assumes them, and a service that follows them needs no documentation to be queried

Source: prometheus.io, "Metric and label naming".

Counter

- A **counter** only goes up: requests handled, bytes sent, errors seen. It starts at 0 when the process starts

```
http_requests_total{path="/api/boardgames",status="200"} 305.0
```

- The value itself says little: 305 since when? Since the pod started, which may be a minute or a month ago
- What we want is how fast it grows: requests **per second** over the last minute. That is computed later, from the samples
- The name ends in `_total`, as the naming conventions say

Gauge

- A **gauge** is a number that goes up and down: pods available, memory in use, items in a queue, temperature

```
kube_deployment_status_replicas_available{deployment="api-staging"} 2
container_memory_working_set_bytes{pod="api-staging-6c94d5d5b6-dxctf"} 43773952
```

- Here the value **is** the point: 2 pods available now; 42 MiB in use now
- The first line is not from our service: another program reads the cluster's objects and exposes them as numbers

Histogram

- A **histogram** counts observations into **buckets**: how many requests took less than 5 ms, less than 10 ms, less than 25 ms, …

```
http_request_duration_seconds_bucket{le="0.005",path="/api/boardgames"} 5.0
http_request_duration_seconds_bucket{le="0.01",path="/api/boardgames"} 6.0
http_request_duration_seconds_bucket{le="0.025",path="/api/boardgames"} 6.0
…
http_request_duration_seconds_bucket{le="+Inf",path="/api/boardgames"} 6.0
http_request_duration_seconds_count{path="/api/boardgames"} 6.0
http_request_duration_seconds_sum{path="/api/boardgames"} 0.015262885019183159
```

Line	Says
`_bucket{le="0.005"} 5`	5 requests took 0.005 s or less. `le` is *less than or equal*
`_bucket{le="0.01"} 6`	6 took 0.01 s or less: the 5 from above and one more, so that one took between 5 and 10 ms. The buckets are **cumulative**
`_bucket{le="+Inf"} 6`	all 6, whatever they took; the last bucket always equals the count
`_count 6`	6 requests were measured
`_sum 0.015`	together they took 0.015 s; divided by the count, 2.5 ms each

- Every line is a counter, so all of them can be turned into rates and added across pods
- A percentile ("95 % of requests took less than …") is computed later from the buckets; the histogram only counts

Summary

- A **summary** is the fourth type: the service computes percentiles itself and exposes them as numbers
- The percentile of one pod cannot be combined with the percentile of another pod: the median of two medians is not the median
- With two or more pods that makes a summary useless for the question we ask. Histograms add up; summaries do not
- The lecture, the assignment and the project use **histograms** for durations

Instrumenting the service

- Exposing the numbers is each service's own job: it keeps its counters and histograms in memory and answers `/metrics`. A **client library** does the keeping and the text format, one per language: Python has `prometheus_client`, Go `client_golang`, Node `prom-client`, Spring Boot has it built in through Micrometer. The library's documentation shows the few lines for your framework; ours, Python with FastAPI:

```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)
```

- A **middleware** wraps every request: time it, count it. The library adds `_total` to the counter's name by itself

The route, not the URL

- `/api/boardgames/4711` and `/api/boardgames/4712` are the same route, `/api/boardgames/{gid}`. The `path` label holds the **route**
- Why: every different label value is a new series. With the id in the label, every boardgame ever asked for is a series of its own, kept in memory for days. Ten thousand series from one small service is how a collector dies
- And it tells nothing: how often one boardgame was fetched is a question for the database, not for monitoring
- The same for every label: a **short, fixed list** of values. Routes, status codes, methods, environments: yes. Ids, names, addresses, timestamps: never
- The course collector drops a pod that exposes more than **1000 series**

What the library adds by itself

- The same `/metrics` answer carries numbers about the process, with no code written for them:

```
process_resident_memory_bytes 6.4622592e+07
process_cpu_seconds_total 0.96
process_open_fds 11.0
python_gc_collections_total{generation="0"} 51.0
```

- Memory, CPU time, open files, garbage collection: enough to see a leak or a runaway process
- Go, Java and Node libraries do the same under their own names

Pull, not push

- The service does not send its numbers anywhere. It **answers** `/metrics` when Prometheus asks, every 30 seconds. One call is a **scrape**; the pod that is called is a **target**
- Why pull:
  - the service needs no address, no credential, no retry logic for a collector that is down
  - a target that does not answer is itself a signal: Prometheus records `up` as 0
  - the collector decides how often, and can be replaced without touching any service
- Where push is used: batch jobs that end before the next scrape; they hand their numbers to a small gateway that is scraped instead

One Prometheus for the whole cluster

- The course Prometheus scrapes every annotated pod in every student namespace, and the programs that describe the cluster itself:

Source	Scraped from	Gives
your service	the annotated pods	`http_requests_total`, `http_request_duration_seconds`, `process_…`
kube-state-metrics	one pod that watches the API	`kube_deployment_status_replicas_available`, `kube_pod_status_phase`, … for every object in every namespace
cAdvisor, inside the kubelet	every node	`container_cpu_usage_seconds_total`, `container_memory_working_set_bytes` per container
node-exporter	every node	the machine: `node_cpu_seconds_total`, `node_memory_MemAvailable_bytes`, disk, network

- Every series carries the labels Prometheus attached: `namespace`, `pod`, and `job` for the source. Yours are the ones with your namespace
- One Prometheus holds about 400 000 series for this cluster in 2 to 3 GB of memory

Finding the pods

- Prometheus asks the Kubernetes API for the list of pods and reads their **annotations**. A pod that says so is scraped:

```
template:
  metadata:
    annotations:
      prometheus.io/scrape: "true"
      prometheus.io/port: "8000"
```

- The annotations sit on the **pod template** of the Deployment, in the chart, so every pod the Deployment makes carries them. An annotation on the Deployment object itself marks nothing: Prometheus looks at pods
- From the pod it takes the address, the namespace and the `app` label, and attaches them to every series it stores. That is how `namespace="000007test"` gets onto our numbers without the service knowing its namespace

The Prometheus page

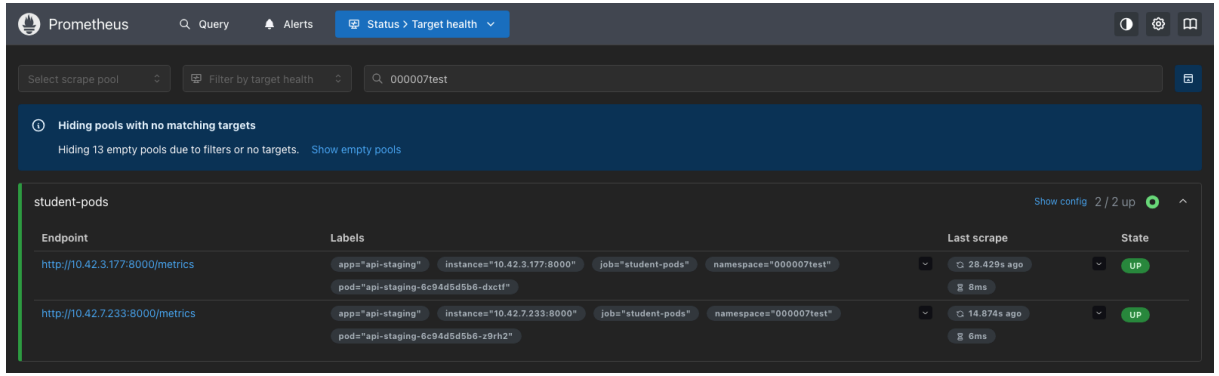
- `http://prometheus.k8s.digivader.com/` : the course Prometheus's own web page, open to read. Its menu, and what each part is for:

Menu	Shows
Query	a box for a query; the answer as a table of series or as a graph over a time range
Alerts	every alert rule it evaluates and its state right now
Status → Target health	every target it scrapes, with the labels it attaches and whether the last scrape worked
Status → Rules	the rules as it loaded them, with the time of the last evaluation
Status → Service discovery	every pod it found, and why it was kept or dropped

Status → Target health

A **target** is one endpoint Prometheus scrapes. This page lists every target, one row each, grouped by scrape job; the search box filters by any label.

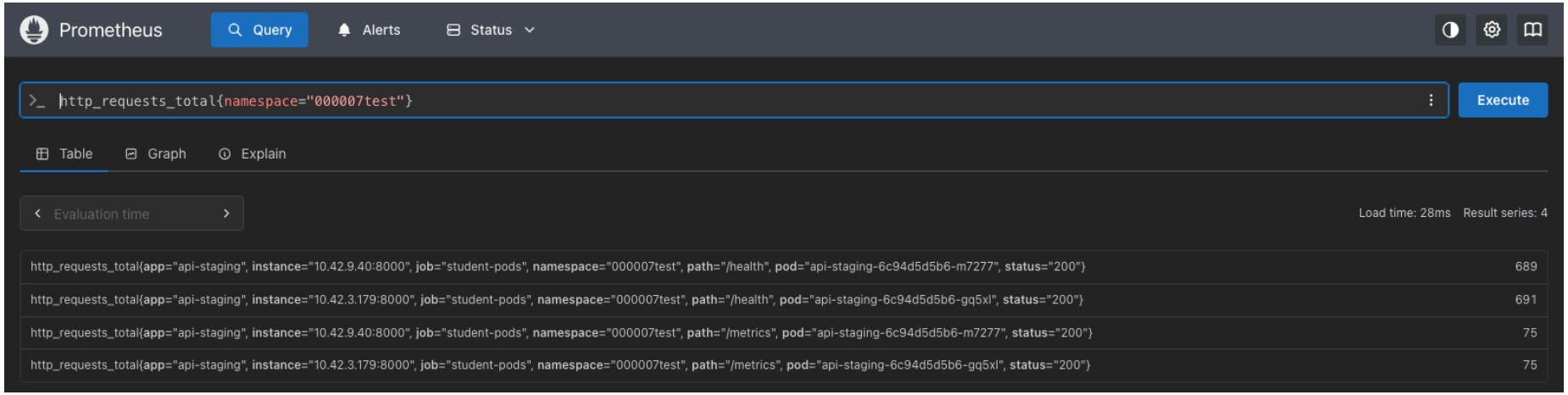
Column	Means
Endpoint	the URL called every 30 s: the pod's address, the port from the annotation, `/metrics`
Labels	the labels Prometheus attaches to every series from this endpoint
Last scrape	when it last called, and how long the answer took
State	`UP` if the last call answered; `DOWN` with the error text if not



Our namespace in the search box: the job `student-pods`, two rows, our two staging pods at their pod addresses on port 8000, both `UP`, the last scrape 8 and 6 ms. The labels: `app` and `pod` from the pod, `namespace`, `instance` for the address, `job` for the source.

Query

The **Query** page is where a question is typed and answered. A query goes into the box, *Execute* sends it to Prometheus, and the answer comes back in one of two tabs: *Table* lists the latest sample of every matching series, *Graph* draws them over a time range.



- The query here is `http_requests_total{namespace="000007test"}`, right after a quiet stretch. The answer is four series: two pods, only `/health` and `/metrics` being called, 689 and 691 and 75 and 75. The labels Prometheus attached, `instance`, `job`, `namespace`, `pod`, `app`, sit next to the service's own, `path` and `status`
- The language of that box is **PromQL**

The simplest query: a name and labels

The simplest question to Prometheus is a metric name, with label conditions in braces. The answer is every series that matches, each with its latest sample.

```
> http_requests_total{namespace="000007test", path="/api/boardgames", status="200"}
http_requests_total{app="api-staging", namespace="000007test", path="/api/boardgames", pod="api-staging-6c94d5d5b6-z9rh2", status="200"}  305
http_requests_total{app="api-staging", namespace="000007test", path="/api/boardgames", pod="api-staging-6c94d5d5b6-dxctf", status="200"}  291
```

- The name picks the metric. Each `label="value"` keeps only the series with that value: our namespace, the list route, status 200
- Two series come back, because the two pods differ in the `pod` label. Each line is one series with all its labels and the latest value: 305 requests counted by one pod, 291 by the other
- The line starting with `>` is what was typed; the lines below are the answer
- Prometheus calls this a selector. Every other query takes such a selection and computes on it

`rate`: per second

`rate` is a function that turns a counter into a speed: how much it grew per second, over a window of recent samples. The window is written after the selector in brackets.

```
> rate(http_requests_total{namespace="000007test", path="/api/boardgames", status="200"}[1m])
{app="api-staging", …, pod="api-staging-6c94d5d5b6-z9rh2", status="200"}  0.767
{app="api-staging", …, pod="api-staging-6c94d5d5b6-dxctf", status="200"}  0.800
```

- `[1m]` selects the samples of the **last minute** instead of the latest one; `rate` computes the growth per second over them
- 0.767 + 0.800 ≈ 1.6 list requests per second over the two pods. The test loop was written to send four per second and managed about 1.6: Prometheus reports what happened, not what was planned
- `rate` knows counters restart at 0 when a pod restarts, and does not count the drop
- The window must hold at least two samples: with a scrape every 30 s, `[1m]` is the shortest that works; `[5m]` is the usual choice for alerts
- `increase(…[2m])` is the same thing as a total instead of per second: 86.7 and 82.7 requests in two minutes
- `rate` of a gauge is meaningless. Pods available per second is not a number anyone wants

`sum`: across pods, by label

`sum` is a function that adds series together. Alone it adds everything into one number; with `by (label)` it adds everything that shares a value of that label and keeps one line per value.

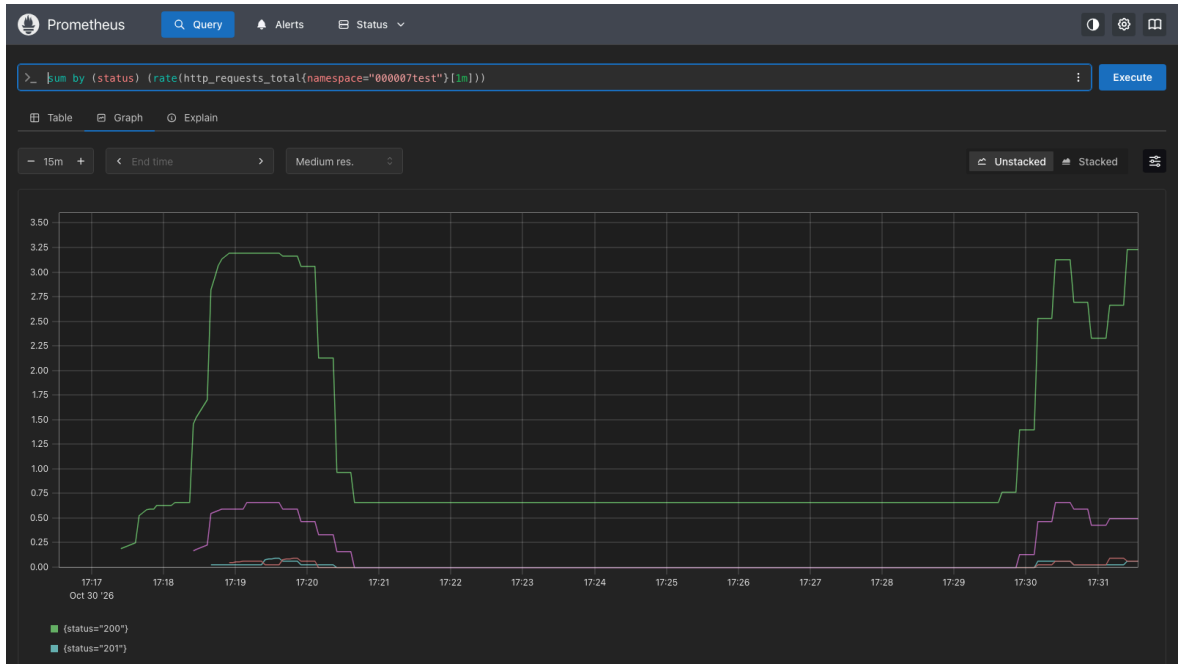
```
> sum(rate(http_requests_total{namespace="000007test", path="/api/boardgames", status="200"}[1m]))
{}  1.500

> sum by (status) (rate(http_requests_total{namespace="000007test"}[1m]))
{status="200"}  2.167
{status="404"}  0.367
{status="201"}  0.033
{status="204"}  0.033
```

- `sum` adds series together and drops the labels they differed in; `{}` is a single number with no labels left
- `sum by (status)` keeps one label and adds everything else: the pods, the paths. The four lines are the service's traffic by outcome, whatever the number of pods
- The order matters: `rate` first, per series, then `sum`. The sum of two counters that restart at different times is a counter that jumps

The same query, drawn

The *Graph* tab of the Query page draws every series of the answer as a line over a time range, chosen with the `15m` control; the legend below names each line by its labels.



The query is `sum by (status) (rate(http_requests_total{namespace="000007test"}[1m]))` over the last 15 minutes: two test runs with a quiet stretch between, `200` on top, `404` below it. One line per `status`, whatever the number of pods.

A ratio: the error share

Two queries divided by each other give a share: PromQL applies `/` to the two numbers, or series by series when both sides have labels.

```
> sum(rate(http_requests_total{namespace="000007test", status=~"4.."}[5m]))
  / sum(rate(http_requests_total{namespace="000007test"}[5m]))
{}  0.133

> sum(rate(http_requests_total{namespace="000007test", status=~"5.."}[5m]))
  / sum(rate(http_requests_total{namespace="000007test"}[5m]))
(empty)
```

- `=~` matches a regular expression: `"4.."` is every status from 400 to 499
- A number divided by a number: 13 % of the requests were answered `4xx`. One in five requests in the test asked for an unknown id, plus the `/health` probes that succeed
- The `5xx` share has **no answer at all**: there is no series with a `5xx` status, so the numerator is not 0, it is nothing. For a dashboard that should read 0: `(sum(…) or vector(0)) / sum(…)`
- `4xx` is the client's mistake, `5xx` is ours. The error signal of the golden four is the `5xx` share

`histogram_quantile`: the 95th percentile

`histogram_quantile` is the function that reads a percentile out of a histogram: given the rate of every bucket, it returns the value below which the chosen share of observations fall.

```
> histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket{namespace="000007test"}[5m])))
{}  0.009
> histogram_quantile(0.5, sum by (le) (rate(http_request_duration_seconds_bucket{namespace="000007test"}[5m])))
{}  0.004
```

- Read from the inside out: the per-second rate of every bucket, added across pods and paths but **kept apart by `le`**, then the value below which 95 % of the observations fall
- 95 % of requests took under 9 ms; half took under 4 ms. The number is interpolated inside the bucket it lands in, so it is as exact as the buckets are fine
- Kept apart by path, it is one number per route:

```
> histogram_quantile(0.95, sum by (le, path) (rate(http_request_duration_seconds_bucket{namespace="000007test"}[5m])))
{path="/api/boardgames/{gid}"}  0.009
{path="/api/boardgames"}        0.009
{path="/health"}                0.009
{path="/metrics"}               0.011
```

- Here the routes are within two milliseconds of each other; a route with a slow query behind it would stand out
- `le` must stay in the `by`: without it the buckets collapse into one number and there is no percentile to read

`up` : is the target there

`up` is a metric Prometheus makes itself, one series per target, with the value 1 or 0.

```
> up{namespace="000007test"}
up{app="api-staging", namespace="000007test", pod="api-staging-6c94d5d5b6-z9rh2"}  1
up{app="api-staging", namespace="000007test", pod="api-staging-6c94d5d5b6-dxctf"}  1

> up{namespace="000007test"} == 0
(empty)
```

- `up` is written by Prometheus itself after every scrape: 1 if the target answered, 0 if not. It exists for every target, whatever the target exposes
- `== 0` keeps only the series whose value is 0: a list of the pods that did not answer. Empty is the answer we want
- A pod that is deleted disappears from the target list, and with it its `up` series. "No series" and "`up` is 0" are different states: the first is *gone*, the second is *there but not answering*

Numbers about the pods, from the cluster

- The service's own numbers are half the picture. The other half the cluster collects for every pod, instrumented or not: kube-state-metrics gives what the API knows about Deployments and pods, cAdvisor gives each container's CPU and memory

```
> kube_deployment_status_replicas_available{namespace="000007test"}
{deployment="api"}          2
{deployment="api-dev"}      1
{deployment="api-staging"}  2

> kube_deployment_spec_replicas{namespace="000007test"}
{deployment="api"}          2
{deployment="api-dev"}      1
{deployment="api-staging"}  2

> container_memory_working_set_bytes{namespace="000007test", container="service"}
{pod="api-staging-6c94d5d5b6-dxctf", …}  43773952
{pod="api-staging-6c94d5d5b6-z9rh2", …}  44281856

> rate(container_cpu_usage_seconds_total{namespace="000007test", container="service"}[5m])
{pod="api-staging-6c94d5d5b6-dxctf", …}  0.013
{pod="api-staging-6c94d5d5b6-z9rh2", …}  0.014
```

- *available* against *wanted* is the health of a Deployment in two gauges. The autoscaler of lecture 8 reads the CPU line; so can we: 0.013 cores of the 0.025 requested
- The series carry the pod's labels, so `namespace="000007test"` selects ours out of 500 pods; `api` is the Deployment of assignment 6, still in the test namespace

Three mistakes that return a number

PromQL rarely refuses a query. It answers the question as asked, and when the question is wrong the answer is a number that means nothing. The three most common:

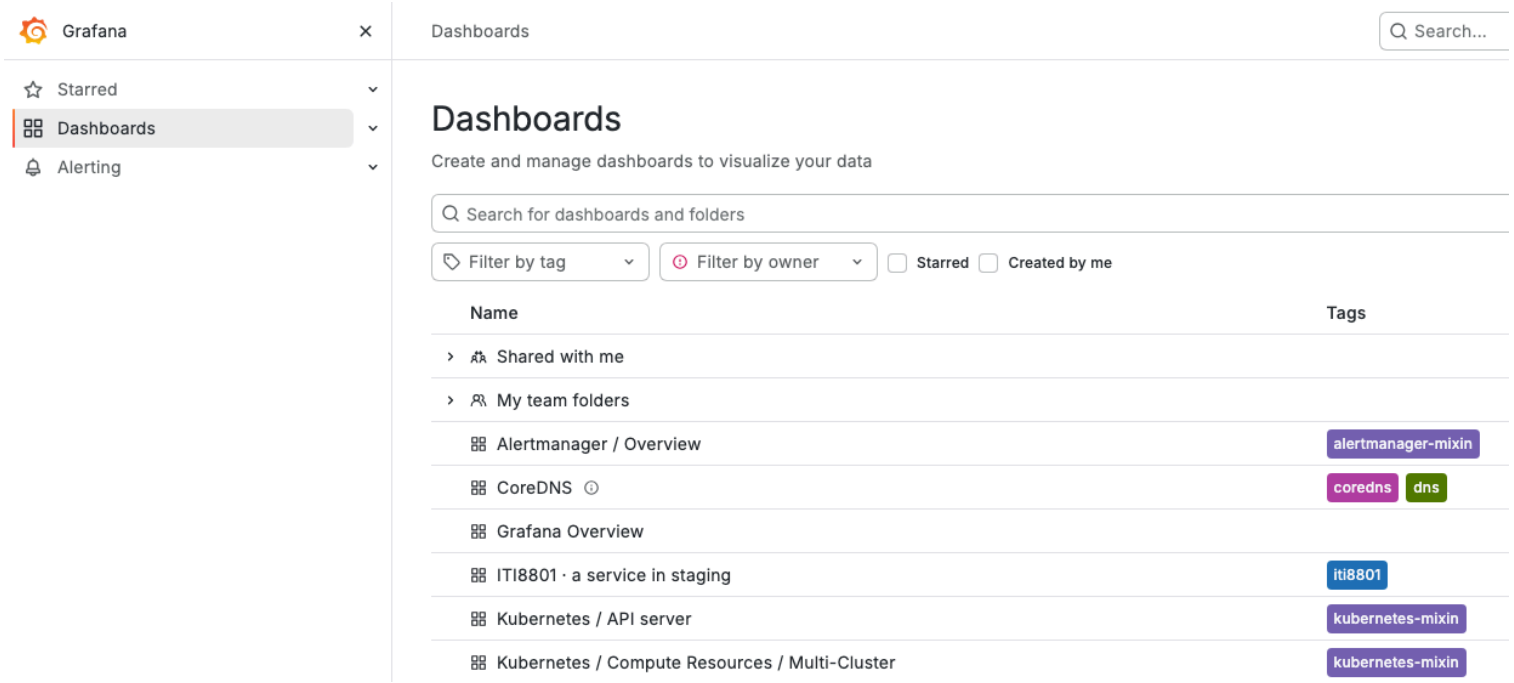
Query	What is wrong	What it returns
`rate(kube_deployment_status_replicas_available[5m])`	`rate` of a gauge	the slope of a line that is flat: 0, or nonsense after a change
`rate(sum(http_requests_total)[5m])`	`sum` before `rate`	a jump every time one pod restarts, and a syntax error in the usual spelling
`rate(http_requests_total[30s])`	a window shorter than two scrapes	no result, or a result from one sample: 0

Grafana

- **Grafana** draws queries. It stores no numbers of its own: a panel sends a PromQL query to Prometheus for the time range shown and plots the answer
- Its words:
  - a **data source**: a Prometheus to ask. The course Grafana has one
  - a **dashboard**: a page of panels
  - a **panel**: one chart on the page. It is drawn from one or more queries, and it knows the unit the numbers are in, seconds or requests per second, so the axis is labelled right
  - a **variable**: a value the user picks at the top, used inside the queries, such as the namespace
  - the **time range**: last 15 minutes, last day, or two fixed points
- The course Grafana is at `http://grafana.k8s.digivader.com/`, open to read without a login

The Grafana page

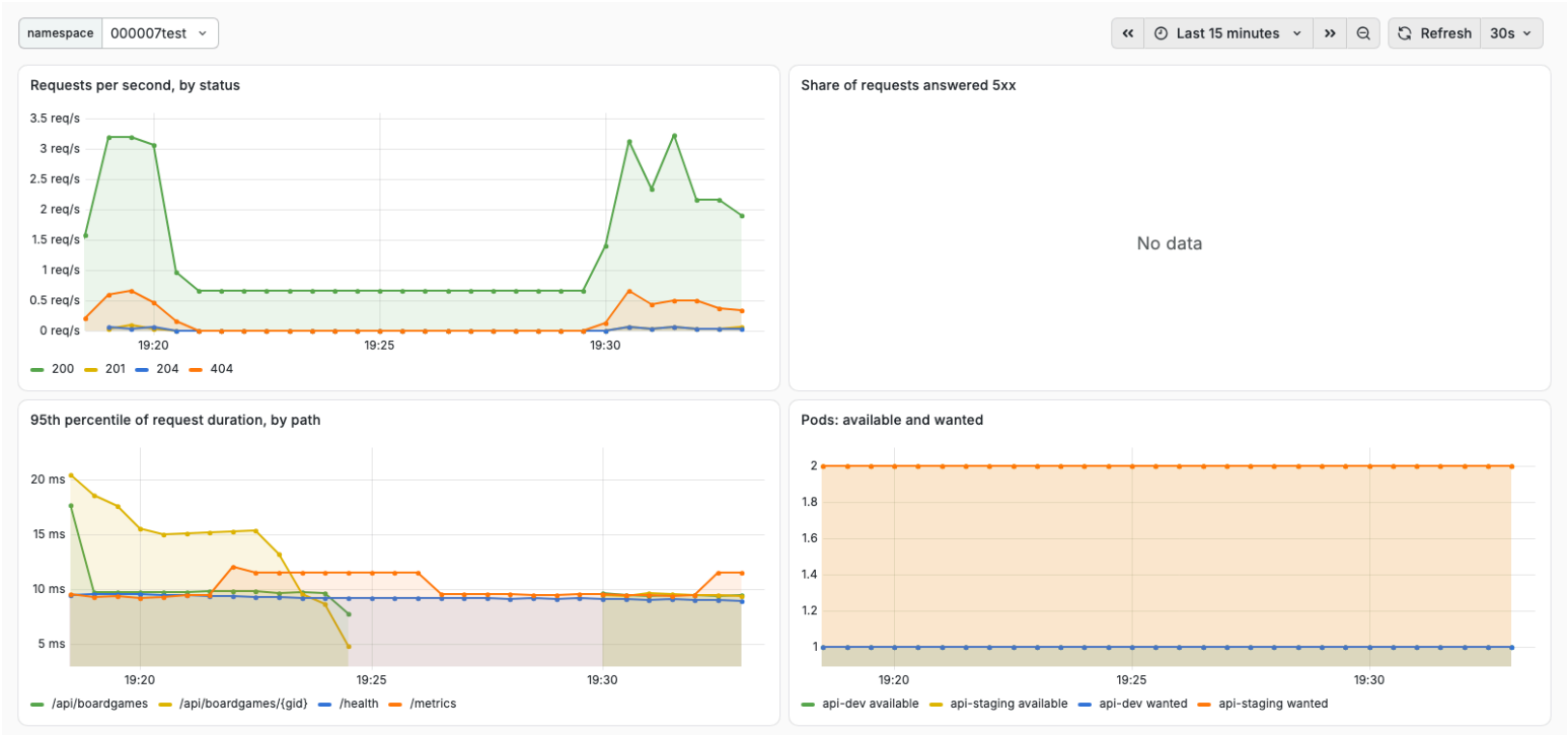
Grafana's own page: a menu on the left, the chosen item on the right. Two menu entries matter for this course.



- **Dashboards**: the list of every dashboard Grafana holds. Ours is *ITI8801 · a service in staging*; the *Kubernetes / Compute Resources* ones come with the monitoring stack and show a namespace's CPU and memory
- **Alerting → Alert rules**: every rule Prometheus holds and its state, searchable by namespace. **Alerting → Active notifications**: what Alertmanager holds right now, with *Alertmanager* chosen at the top right

The dashboard of this service

A **dashboard** is one page of panels, each panel one chart drawn from a query. This is the course dashboard, Dashboards → *ITI8801 · a service in staging*, with the namespace picked at the top and the last 15 minutes shown.



The four panels are the four golden signals, pods instead of saturation. The gap in the first panel is the minutes between two test runs; the `5xx` panel is empty because no request failed.

A panel is a query with a unit

Behind every panel of that dashboard is one PromQL query and a unit for the axis. The four of them:

Panel	Query	Unit
Requests per second, by status	`sum by (status) (rate(http_requests_total{namespace="$namespace"}[1m]))`	requests/s
Share of requests answered 5xx	`(sum(rate(http_requests_total{namespace="$namespace", status=~"5.."}[5m])) or vector(0)) / sum(rate(http_requests_total{namespace="$namespace"}[5m]))`	percent (0–1)
95th percentile of request duration, by path	`histogram_quantile(0.95, sum by (le, path) (rate(http_request_duration_seconds_bucket{namespace="$namespace"}[5m])))`	seconds
Pods: available and wanted	`kube_deployment_status_replicas_available{namespace="$namespace", deployment=~"api-.*"}` and `kube_deployment_spec_replicas{…}`	count

- `$namespace` is the variable at the top of the page; its list of values is itself a query: `label_values(http_requests_total, namespace)`
- The queries are the ones of the last pages, with the namespace replaced. A dashboard is nothing more than saved queries with a picture

A dashboard as code

A dashboard is a JSON file: every panel, query and unit is in it, and Grafana can load it from a file instead of from its own database.

- A dashboard built by clicking lives in Grafana's database. When Grafana is reinstalled, it is gone; when a teammate wants the same, they click again
- Grafana exports a dashboard as JSON; the JSON goes into git, and the course Grafana loads every ConfigMap that carries one label:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: dashboard-iti8801-service
  namespace: monitoring
  labels:
    grafana_dashboard: "1"
data:
  iti8801-service.json: |
    {"uid": "iti8801-service", "title": "ITI8801 · a service in staging",
     "panels": [{"type": "timeseries", "title": "Requests per second, by status",
                 "targets": [{"expr": "sum by (status) (rate(http_requests_total{namespace=\"$namespace\"}[1m]))"}], …
```

- The dashboard on the page before is this ConfigMap. A one-off question is typed into the Prometheus page's query box; what you look at often becomes a panel, and a panel goes into a file

Alerts

- A dashboard shows numbers to a person who is looking at it. Nobody looks at it all the time
- An **alert** is a query that Prometheus evaluates by itself, every 30 seconds. When the query returns a result for longer than a set time, the alert **fires**
- Three parts:
  - a **rule**: the query, how long it must stay true, a severity, a text for the reader
  - **Prometheus** evaluates the rules and sends the firing ones to Alertmanager
  - **Alertmanager** groups the alerts, drops the silenced ones, and sends the rest to a receiver: e-mail, chat, a phone
- The first alert of any service: no pod is serving

`PrometheusRule`

A `PrometheusRule` is a Kubernetes object that holds alert rules. On the course cluster it lives in your namespace, and the course Prometheus picks it up by itself. Ours, as the chart rendered it for staging; the first of its two rules:

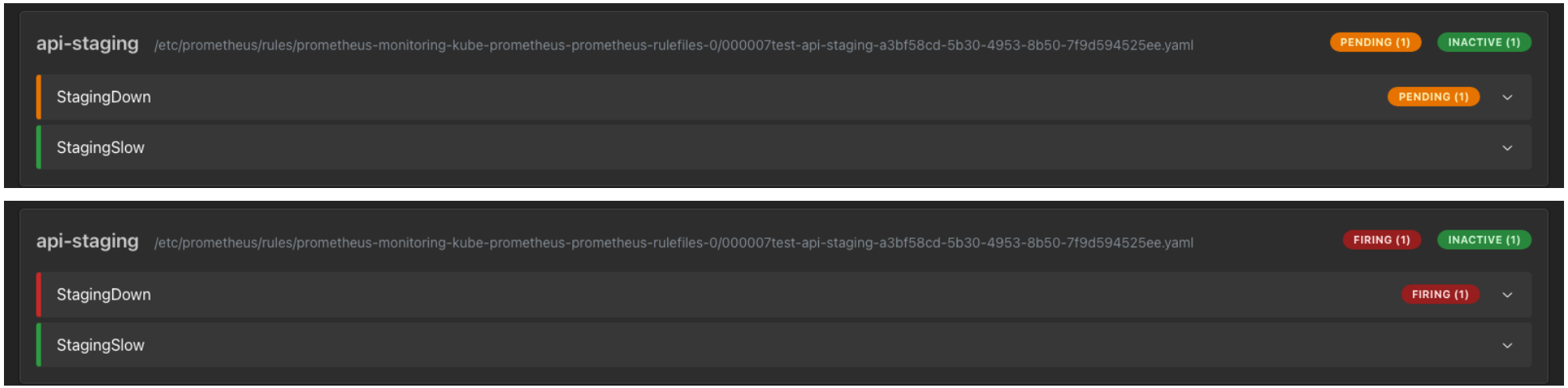
```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: api-staging
  namespace: 000007test
spec:
  groups:
  - name: api-staging
    rules:
    - alert: StagingDown
      expr: kube_deployment_status_replicas_available{namespace="000007test", deployment="api-staging"} == 0
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: staging has no ready pod in 000007test
```

Field	Does
`expr`	the query; the alert is **pending** while it returns something
`for`	how long it must keep returning something before the alert **fires**: a pod that is replaced in 10 s never fires
`labels`	attached to the alert; `severity` is what Alertmanager routes on
`annotations`	text for the person who gets it: what is wrong, where

Pending, then firing

An alert has three states. **Inactive**: the query returns nothing. **Pending**: it returns something, but for less than `for`. **Firing**: it has returned something for `for` or longer, and the alert is sent to Alertmanager. The *Alerts* page of the Prometheus site shows every rule with its state.

- `kubectl scale deployment api-staging --replicas=0`. Within 30 s the query returns a series with value 0; the alert turns pending. After `for: 1m` of that, firing:



- The query's own labels (`namespace`, `deployment`, the `job` of kube-state-metrics) travel with the alert, together with the rule's `severity`
- A rule whose expression does not parse is refused when the object is created: the cluster checks it before Prometheus sees it

Alertmanager

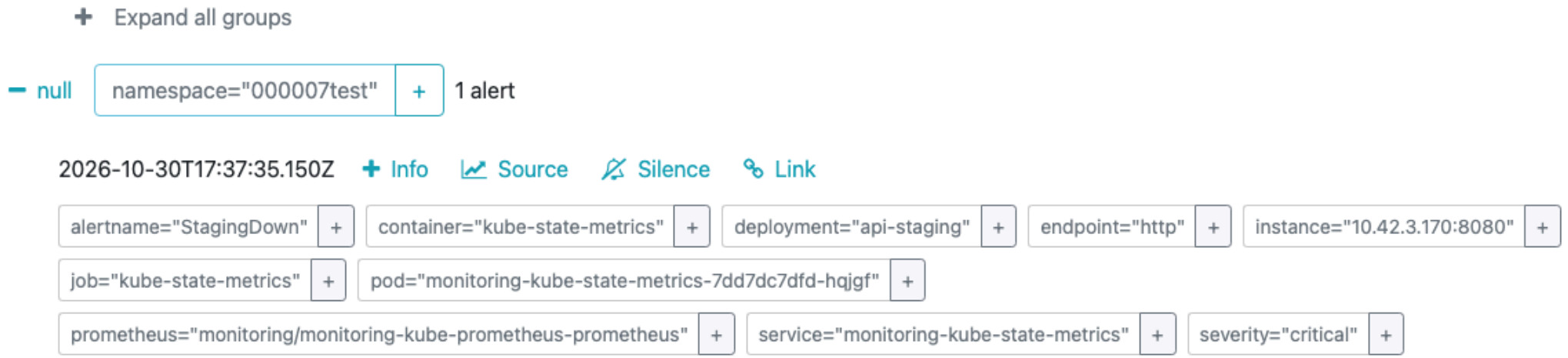
Alertmanager is the program, installed beside Prometheus, that receives the firing alerts and decides what happens to them. It keeps every alert it holds and answers for it on its API. Our alert, as it holds it:

```
$ curl -s 'localhost:9093/api/v2/alerts?filter=alertname=StagingDown'
{
 "labels": {"alertname": "StagingDown", "deployment": "api-staging", "namespace": "000007test",
            "severity": "critical", "job": "kube-state-metrics", …},
 "annotations": {"summary": "staging has no ready pod in 000007test"},
 "startsAt": "2026-10-30T17:37:35.150Z",
 "status": {"state": "active", "silencedBy": [], "inhibitedBy": []}
}
```

- `labels` are the rule's labels and the query's; `annotations` is the rule's text; `startsAt` is when it began; `status.state` is `active` until Prometheus stops sending it
- What Alertmanager does with alerts:
  - **groups** them: ten pods down in one namespace become one notification, not ten
  - **routes** them by label: `severity: critical` to one receiver, `warning` to another
  - **silences** them: a known outage is muted for a set time, by label match
  - **resolves** them: when Prometheus stops sending the alert, it ends and the receiver is told
- The course Alertmanager has no receivers. The alert is only read: from this API, from its page, from Grafana. Sending to a person comes in the team project

Alertmanager's page

Alertmanager has a web page that lists every alert it holds right now, in groups, the way its configuration groups them; here by namespace. On the course cluster the page is not exposed; Grafana's *Active notifications* shows the same list. Our alert on it:



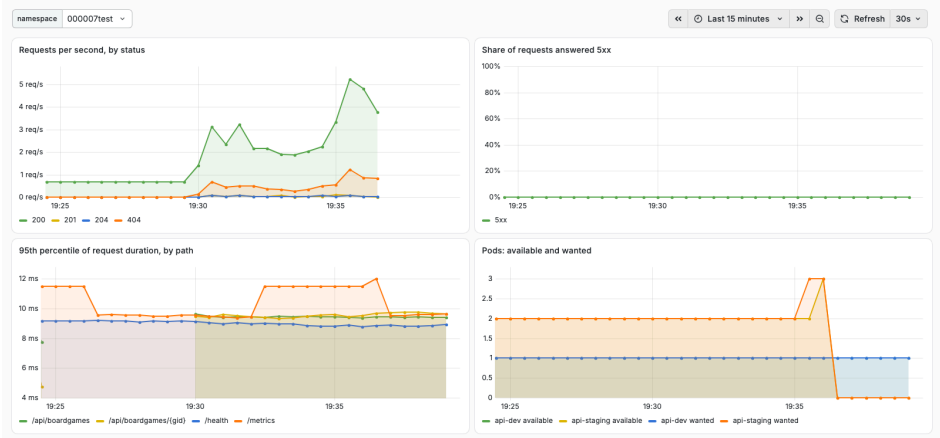
- One group, `namespace="000007test"`, with one alert in it
- The time is when the alert started: 17:37:35
- Every label is a chip. Clicking the `+` next to one adds it to the filter at the top
- *Silence* creates a silence for exactly these labels, for a chosen time. The alert stays, nothing is sent

The drill

A **drill** is a failure caused on purpose, in a controlled way, to check that the alert fires and resolves. The word is the same as in a fire drill. A rule is only known to work once it has fired. Ours, recorded:

```
17:15:05 $ kubectl -n 000007test scale deployment api-staging --replicas=0
17:16:35   StagingDown active in Alertmanager               after the for of 1 min and one evaluation
17:16:38 $ kubectl -n 000007test scale deployment api-staging --replicas=2
17:17:05   StagingDown resolved; GET /health answers 200     after the pods are back
```

- The autoscaler stays quiet at zero replicas and takes over again at two
- The drill repeated with the dashboard open: the request lines **stop** rather than fall, because no pod means no series; *available* goes to zero while *wanted* was set to zero too:



The rule on latency

The same `PrometheusRule` holds a second alert, on latency. Assignment 9 requires it; it is not made to fire in the drill:

```
    - alert: StagingSlow
      expr: histogram_quantile(0.95, sum by (le) (rate(http_request_duration_seconds_bucket{namespace="000007test", app="api-staging"}[5m]))) > 0.5
      for: 2m
      labels:
        severity: warning
      annotations:
        summary: 'staging in 000007test: 95 % of requests take more than 0.5 s'
```

- The 95th percentile over the last 5 minutes, compared with a threshold: `> 0.5` keeps the series only when the value is above half a second
- `for: 2m`: a slow second is not an incident; a slow two minutes is
- `warning`, not `critical`: the service answers, slowly. Someone looks today, nobody is woken
- The threshold is a judgement; a service with a database call per request would set it differently

Where alerts go

Alertmanager sends alerts out through **receivers** and decides which alert goes to which receiver through **routes**. Both are in its configuration file.

- A **receiver** in Alertmanager's configuration: e-mail, a Slack, Discord or Teams webhook, a paging service that calls a phone (PagerDuty, Opsgenie), or any URL that takes a POST
- A **route** sends by label: everything with `severity: critical` to the pager, `warning` to the channel, the rest nowhere
- The text the person reads is the rule's `summary`. *"staging has no ready pod in 000007test"* says what and where; *"alert fired"* says nothing
- Two alerts that go wrong the same way in every team:
  - **too sensitive**: fires on every pod restart, ignored within a week, then ignored the one time it matters
  - **too late**: `for: 1h` to stop the noise, and the outage is an hour old when the phone rings

What makes an alert worth waking up for

- It is about a **symptom** the user feels, not a cause: *no pod is serving*, *95 % of requests take over half a second*, *errors above 5 %*. A full disk is a cause; alert on what the disk being full breaks
- It needs a **person** now. Anything that can wait until morning is a ticket, not a page
- It has a `for` long enough to skip a flicker, short enough to matter: a minute for *down*, a few minutes for *slow*
- It says what to do: a `summary` a tired person understands, a `runbook_url` annotation when there are steps
- Four such alerts beat forty

SLI, SLO, SLA

- An **SLI**, a service level indicator, is one number that says how well the service serves: the share of requests answered `2xx` within 100 ms over the last 5 minutes

```
sum(rate(http_request_duration_seconds_bucket{namespace="000007test", le="0.1"}[5m]))
  / sum(rate(http_request_duration_seconds_count{namespace="000007test"}[5m]))
```

- For our service right now: 1.0. Every request was under 100 ms
- An **SLO**, a service level objective, is the target for it over a period: **99 %** of requests good, measured over 30 days
- The SLO is a decision, not a measurement: 99 % allows 7.2 hours of a bad service a month; 99.9 % allows 43 minutes. The second costs ten times more to keep
- An **SLA**, a service level agreement, is an SLO written into a contract with a customer, with a consequence when it is missed: money back, a penalty. The SLO is what the team aims at inside; the SLA is the lower bar promised outside, so a missed SLO is a warning, not a breach
- A number with a target is the first thing to put on the dashboard and the last thing to alert on directly

Source: Google, *Site Reliability Engineering* (2016), chapter 4, "Service Level Objectives".

Error budget

- 100 % minus the SLO is the **error budget**: at 99 % over 30 days, 1 % of requests may be bad, about 7 hours of total outage
- The budget is **spent** by every incident and by every bad deploy. A team that has spent it stops shipping features and fixes reliability; a team with budget left can take risks
- Alert on the **spending rate**, not on every blip: *the last hour burned 10 % of the month's budget* is worth a page; *one request in a thousand failed just now* is not
- The whole idea in one line: **decide how bad is acceptable, measure it, and act when the measurement says so**

The commands and queries of this lecture

`curl http://…/metrics`	what the service exposes right now
`kubectl get prometheusrule -n <ns>`	the rules in your namespace
`kubectl scale deployment api-staging --replicas=0`	the drill
`http_requests_total{namespace="…"}`	the latest sample of every matching series
`rate(x_total[1m])` · `increase(x_total[2m])`	per second · total, over the window
`sum by (status) (rate(…))`	add across pods, keep one label
`a / b`	a ratio; `or vector(0)` when the numerator may be empty
`histogram_quantile(0.95, sum by (le) (rate(x_bucket[5m])))`	the 95th percentile; `le` stays in the `by`
`up == 0`	targets that did not answer
`kube_deployment_status_replicas_available`	pods ready, from kube-state-metrics
`http://prometheus.k8s.digivader.com/`	the course Prometheus: a query, the targets, the rules
`http://grafana.k8s.digivader.com/`	the course Grafana: the dashboard, the rules' state, the alerts



Questions