Prometheus y Grafana

Para instrumentar un servicio, escribir con la documentación abierta las consultas de un dashboard y de una alerta, y diagnosticar «no data», «target down» o «la alerta nunca sonó». En el C01, Prometheus pasa cada 15 s por cada servicio, le pide sus números y los guarda como series de tiempo; las reglas convierten una consulta en una alerta, Alertmanager decide a quién avisar y Grafana dibuja. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y un laboratorio opcional. En la versión web, al final hay un diagnosticador de síntomas, un revisor de reglas de alerta y un quiz de errores.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

GRAFANA Inspect › Query un lugar de la web: se abre, no se copia

EN LAS FIGURAS: counter (el odómetro) · gauge (el medidor de gasolina) · histogram (los cajones)

ok, up o resuelta · pending · firing, down o una cuenta que miente · silenciada, inhibida o ignorada

0 Números de bolsillo: puertos, intervalos, tipos de métrica y las funciones de PromQL que más usas

¿Qué puertos, intervalos, defaults y funciones conviene saberse de memoria para leer un sistema con Prometheus en segundos?

SINTAXIS CON VARIABLES

`curl -s` *host* `:` *puerto* `/metrics | head -n 20`

Lo mismo que lee Prometheus en cada **scrape** (la visita periódica en que pide los números): texto plano, una línea por **muestra** (el valor de una **serie**, la historia de un número, en ese momento). Puertos de bolsillo: gateway `8080`, Triton `8002`, node_exporter `9100` y el propio Prometheus en `9090`.

`up{job="`*job*`"} == 0`

`up` la escribe Prometheus después de cada scrape: **1** si pudo leer al target (cada URL que visita), **0** si falló. Es la primera consulta cuando un panel dice «No data»: si ni siquiera existe `up` para ese job, Prometheus no sabe que el target existe.

`sum by (`*label*`) (rate(`*contador*`_total[`*ventana*`]))`

Un **counter** (contador que solo sube) se lee con `rate()`: cuánto sube por segundo, en promedio, dentro de la **ventana** (el tramo de tiempo hacia atrás, entre corchetes). Primero `rate` y después `sum`: así se corrige el reinicio de cada pod. A Prometheus le bastan 2 muestras en la ventana; la regla práctica (la de `$__rate_interval` en Grafana) pide al menos 4 scrapes: `[1m]` con 15 s. Las alertas de C01 usan `[5m]`.

`histogram_quantile(`*q*`, sum by (le) (rate(`*histograma*`_bucket[`*ventana*`])))`

El **p99** (el tiempo bajo el que queda el 99 % de los requests) de todo el servicio: primero se suman los **buckets** (los cajones del histograma) de todos los pods y después se calcula el cuantil. Nunca promedies el p99 de cada pod. Es la expresión de `ClassifierHighP99Latency` en C01.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
host	dónde corre el target, visto desde donde haces curl	`localhost` (con `kubectl port-forward`) o `gateway` dentro del compose
puerto	puerto donde el target sirve /metrics	`8080` gateway · `8002` Triton · `9100` `node_exporter`
job	grupo de targets: el job_name de prometheus.yml	`gateway`
label	label por el que partes el resultado	`status`, `source` o `le`
contador	nombre del counter sin el _total	`classifier_requests`
ventana	cuánto tiempo hacia atrás mira la función	`5m` en las alertas de C01; `1m` como mínimo práctico con scrape de 15 s
q	cuantil, entre 0 y 1	`0.99`
histograma	nombre base del histograma	`classifier_request_latency_seconds`
x, w, t	x: el valor del counter; w: la ventana; t: la hora de cada muestra	lab de la sección 2: 0.371/s en 2 min, con un reinicio adentro

VARIABLE	QUÉ ES	EJEMPLO
Δ_{scrape}	cada cuánto lee Prometheus (scrape_interval)	15 s en C01; 1 min si no lo escribes
N_{series}	series que se guardan en cada ronda	≈ 205 por pod del gateway (sección 3): 615 con 3 pods
$t_{\text{retención}}$	cuánto guarda el disco local	15 días si nadie la cambia

FÓRMULAS

$$\text{rate}(x[w]) \approx \frac{x_{\text{última}} - x_{\text{primera}} + \sum x_{\text{antes de cada reinicio}}}{t_{\text{última}} - t_{\text{primera}}} \qquad \text{(y extrapola a los bordes de } w\text{)}$$

$$\text{increase}(x[w]) = \text{rate}(x[w]) \times w$$

$$w \geq 4 \times \Delta_{\text{scrape}} = 4 \times 15\,\text{s} = 1\,\text{min} \qquad \text{(regla práctica; Prometheus pide solo 2 muestras)}$$

$$\text{disco} \approx t_{\text{retención}} \times \frac{N_{\text{series}}}{\Delta_{\text{scrape}}} \times 1\text{–}2\,\text{B} = 15\,\text{d} \times \frac{615}{15\,\text{s}} \times 1\text{–}2\,\text{B} \approx 53\text{–}106\,\text{MB}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
9090	Prometheus: su web (consultas, alertas, targets), su API HTTP y sus propias métricas en /metrics.	localhost:9090/alerts en el compose de C01
9093	Alertmanager: recibe las alertas que disparan y decide a quién avisar.	kube-prometheus-stack (el paquete de Helm con Prometheus, Alertmanager y Grafana) lo instala junto a Prometheus
3000	Grafana: los dashboards. El primer login es admin / admin; cambia esa contraseña.	docker run -p 3000:3000 grafana/grafana:13.2.4
9100 · 8080 · 9400 · 9091	Exporters (programas que leen otro sistema y lo publican como métricas) y un buzón: node_exporter (la máquina), kube-state-metrics (los objetos de Kubernetes), dcgm-exporter (GPUs NVIDIA) y el Pushgateway (para jobs batch, trabajos que corren y terminan).	C01 lee la GPU desde Triton, sin dcgm-exporter
8002	Las métricas nv_* de Triton: counters de requests y tiempos, gauges (valores que suben y bajan) de GPU.	nv_inference_queue_duration_us, la señal de KEDA (el autoscaler de Triton)
scrape_interval: 1m	Default de cada cuánto se lee cada target si no lo escribes. El archivo de ejemplo (el que trae la imagen de Docker) pone 15s.	C01: 15s en local y en los ServiceMonitors (su config de scrape en EKS)
scrape_timeout: 10s	Cuánto espera cada scrape antes de darlo por fallido (up = 0). Nunca más que el intervalo.	un /metrics que tarda 12 s da up = 0 aunque el proceso viva
evaluation_interval: 1m	Cada cuánto se evalúan las reglas (las consultas que Prometheus corre solo: alertas y grabaciones). Un grupo de reglas puede fijar su propio interval.	C01: 15s; su grupo de grabación, 1m
Retención: 15d	Cuánto tiempo guarda el disco local. En 3.13 se cambia en prometheus.yml (storage: tsdb: retention: time); el flag --storage.tsdb.retention.time todavía funciona, pero está deprecado (se desaconseja y puede desaparecer).	ClassifierLowConfidenceSpike necesita 7 días de historia
Counter → rate()	Solo sube y vuelve a 0 al reiniciar: se lee con rate() o increase(), nunca crudo.	rate(classifier_requests_total[5m])
Gauge → directo	Sube y baja: se lee tal cual, o con avg_over_time / max_over_time (promedio o máximo de la ventana).	nv_gpu_utilization > 0.85 (GPUSaturated)
Histogram → histogram_quantile	Cajones acumulados (_bucket con le, «hasta») más _sum y _count: se suman entre pods y después se saca el cuantil.	p99 > 0.2 s (ClassifierHighP99Latency)

PIEZA	QUÉ HACE	EJEMPLO
Summary	El cuantil ya viene calculado en cada proceso: no se puede sumar entre pods. El de Python ni siquiera lo calcula.	**evítalo en servicios con réplicas**
for · group_wait	Una alerta espera en pending (pendiente) lo que dice for antes de pasar a firing (disparada); sin for (0s), dispara en la primera evaluación verdadera. Alertmanager espera group_wait (30 s por defecto) antes del primer aviso.	**for: 5m en ClassifierHigh5xxRate**

CAMPO	DEFAULT	EN C01	POR QUÉ IMPORTA
scrape_interval	1m (Prometheus Operator: 30s)	15s en prometheus.yml y en los ServiceMonitors	la resolución: con 1m, la regla práctica pide rate(x[4m]) o más
scrape_timeout	10s; si el intervalo es menor, el intervalo	sin cambiar (10s)	un /metrics lento da up = 0 aunque el proceso viva
evaluation_interval	1m (Prometheus Operator: 30s)	15s; el grupo de grabación usa interval: 1m	cada cuánto puede cambiar de estado una alerta
metrics_path · scheme	/metrics · http	/metrics en los dos jobs	otra ruta se escribe en el job, no en targets
Retención (storage.tsdb.retention.time)	15d	sin cambiar	≥ 7 días para el promedio de ClassifierLowConfidenceSpike
for	0s: firing en la primera evaluación verdadera	5m, 10m, 15m o 30m según la alerta	filtra picos cortos; la ventana del rate los estira, así que cuenta las dos
query.lookback-delta	5m	—	una consulta instantánea toma la última muestra de hasta 5 min atrás
rules.alert.resend-delay	1m	—	Prometheus reenvía a Alertmanager cada alerta que sigue en firing
group_wait · group_interval · repeat_interval	30s · 5m · 4h	sin Alertmanager en local	cuándo llega el primer aviso, los siguientes del grupo y el recordatorio
sample_limit · label_limit	0 (sin límite)	sin cambiar	con límite, un scrape que lo pasa se descarta entero: up = 0
Ventana de rate()	no tiene default	[5m] en casi todas las alertas, [30m] en la grabación y en la de confianza, [2m] en KEDA	Prometheus pide solo 2 muestras; la regla práctica es al menos 4 scrapes (la de $__rate_interval en Grafana)

Para ver los defaults de tu propio Prometheus: `/api/v1/status/config` devuelve la config cargada con los defaults ya escritos, y `/api/v1/status/flags`, los flags. Es el lab de esta sección.

Defaults de Prometheus 3.13 y Alertmanager 0.34 contra lo que fija C01. Verificados en la documentación oficial y en el código fuente, en octubre de 2026.

EJEMPLO REAL · EL PROMETHEUS DEL COMPOSE DE C01: VERSIÓN, PUERTO Y ARCHIVOS

YAML reference/c01/docker-compose.yml (líneas 46–55, el servicio prometheus)

```yaml
  prometheus:
    image: prom/prometheus:v3.13.3  # tag fijo (nunca latest): 3.13 es la LTS vigente (soporte hasta 2027-07-31); sube de parche dentro de 3.13.x
    ports:
      - "9090:9090"  # UI: http://localhost:9090/alerts muestra las 6 alertas y su estado
    volumes:  # montados en la ruta que usa el CMD por defecto de la imagen: no hace falta `command`
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - ./monitoring/alerts.yml:/etc/prometheus/alerts.yml:ro
    depends_on:
      - gateway
      - triton
```

- `v3.13.3` es de la línea **LTS** (soporte largo: un año de arreglos para fallas graves, hasta el 31 de julio de 2027). En octubre de 2026 la más nueva es la 3.15.1, y la 3.13 ya va en 3.13.5 (21 de octubre, con arreglos de seguridad): subir de parche es cambiar el tag, sin tocar la config.
- `9090:9090`: la web y la API. `/alerts` muestra cada alerta en inactive, pending o firing; `/targets` (menú Status → Target health) dice si cada scrape funciona y, si no, con qué error.
- Los dos archivos entran a `/etc/prometheus`, donde el `CMD` de la imagen (los flags con que arranca si no le pasas otros) ya busca la config (`--config.file=/etc/prometheus/prometheus.yml` y `--storage.tsdb.path=/prometheus`); `rule_files: - alerts.yml` se resuelve relativo a esa carpeta. Si agregas flags en `command:`, reemplazas ese CMD entero: repite los dos.
- No hay servicio de Alertmanager ni bloque `alerting:` en `prometheus.yml`: en local las alertas se ven en `/alerts`, pero nadie recibe un aviso. En EKS lo trae kube-prometheus-stack.
- La retención no se toca: 15 días, que alcanzan para los 7 que pide `ClassifierLowConfidenceSpike`. Pero la base vive en un volumen anónimo (lo cuenta el cheat sheet de Docker): tras un `down` + `up` empieza de cero, y el promedio «de 7 días» pasa a ser el de lo que haya.

CÓMO RECORDARLA

9090 Prometheus, 9093 Alertmanager, 3000 Grafana, 9100 la máquina, 8002 Triton. Un scrape cada 15 s (si no lo escribes, 1 min), 10 s para contestar, 15 días de memoria. Y el orden de lectura: `up` → `rate` → `sum by` → `histogram_quantile`.

TRAMPA

Creer que 15 s es el default porque está en todos los ejemplos, incluido el `prometheus.yml` que trae la imagen de Docker, cuyo comentario dice «Default is every 1 minute». Si tu archivo no escribe `scrape_interval`, es **1m**, y un `rate(x[1m])` casi siempre encuentra una sola muestra: devuelve vacío y el panel dice «No data». Con 15 s, `[1m]` tiene 4 muestras; con 1 min, usa `[4m]` o más.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · C01 · Monitoreo, alertas y reentrenamiento · Herramientas · Monitoreo: sistema y modelo · Docker y kubectl · docker compose · Hoja · Formulario de estimaciones

La cadena: de una línea de código a una alerta en tu teléfono

¿Qué pasa, en orden, entre un REQUESTS.inc() en el gateway y el aviso que despierta a alguien, y cuánto tarda cada eslabón?

SINTAXIS CON VARIABLES

`METRICA.labels(label=valor).inc()`

El primer eslabón: un número que sube en la memoria del proceso. Cuesta microsegundos y todavía no sale nada por la red.

`curl -s host:puerto/metrics | grep metrica`

El segundo: el valor acumulado, en texto, esperando a que alguien lo pida. Si aquí no sube, el problema está en tu código, no en Prometheus.

`curl -s -G localhost:9090/api/v1/query --data-urlencode 'query=expresion'`

Prometheus ya lo guardó como serie y lo consultas con **PromQL** (su lenguaje de consultas): la misma expresión que usan las reglas, Grafana y KEDA. `-G` con `--data-urlencode` codifica las llaves y comillas de la consulta.

`curl -s localhost:9090/api/v1/alerts`

Las alertas activas: **pending** (la condición se cumple, pero todavía no durante todo el `for`) o **firing** (ya disparó y se manda a Alertmanager).

`curl -s localhost:9093/api/v2/alerts`

Lo que recibió Alertmanager. Desde aquí agrupa, silencia y avisa a un **receiver** (el destino: PagerDuty, Slack, email o un webhook).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
METRICA	variable de Python que guarda el counter	`REQUESTS`
label = *valor*	la serie que sube	`status="503"`
host : *puerto*	el target	`localhost:8080 con kubectl port-forward`
metrica	nombre expuesto de la métrica	`classifier_requests_total`
expresion	la PromQL que quieres ver	`la expr de ClassifierHigh5xxRate`
t_{cruce}	lo que tarda la expresión en pasar el umbral desde que empiezan los errores	`30 s con 10 % de errores`
w	ventana del rate()	`5 min`
e	fracción de errores que empezó de golpe	`10 %: Triton rechaza 1 de cada 10`
Δ_{scrape}, Δ_{eval}	intervalos de scrape y de evaluación de reglas	`15 s y 15 s en el compose de C01; en EKS, 15 s (ServiceMonitor) y 30 s (default del Prometheus Operator, si nadie lo cambia)`
t_{for}	lo que la alerta espera en pending	`5 min en ClassifierHigh5xxRate`
t_{grupo}	group_wait: lo que Alertmanager espera antes del primer aviso de un grupo nuevo	`30 s por defecto`

FÓRMULAS

$$t_{\text{aviso}} \approx t_{\text{cruce}} + \underbrace{\Delta_{\text{scrape}} + \Delta_{\text{eval}}}_{\leq 30\text{ s en C01}} + t_{\text{for}} + t_{\text{grupo}}$$

$$t_{\text{cruce}} = w \times \frac{\text{umbral}}{e} = 300\text{ s} \times \frac{1\,\%}{10\,\%} = 30\text{ s}$$

$$t_{\text{aviso}} \approx 30 + 30 + 300 + 30\text{ s} \approx 6.5\text{ min} \qquad \text{(ClassifierHigh5xxRate con 10 \% de errores)}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Instrumentación	Las llamadas en tu código que suben counters y observan histogramas, en memoria.	`REQUESTS.labels(…).inc() en el finally de classify`
/metrics	La foto de esos números en texto, cada vez que alguien la pide.	`gateway:8080/metrics`
Scrape	Prometheus lee cada target cada scrape_interval y escribe up (1 o 0).	`cada 15 s en C01`

PIEZA	QUÉ HACE	EJEMPLO
TSDB: head y WAL	La base de series de Prometheus (TSDB, time series database). Las muestras recientes viven en memoria (head) y en un registro en disco (WAL, write-ahead log) que se relee si Prometheus reinicia.	`/prometheus/wal en la imagen de Docker`
Bloques de 2 h	Cada bloque de 2 h pasa de la memoria a un archivo en disco; después se junta con otros en bloques más grandes, y los más viejos que la retención se borran.	`15 días por defecto`
PromQL	Calcula sobre las series: tasas, cocientes, sumas, cuantiles.	`la fracción de 5xx sobre requests válidos`
Regla de grabación	Guarda el resultado de una expresión cara como una serie nueva, en cada evaluación.	`classifier:low_confidence_share:rate30m`
Regla de alerta	Evalúa su expr cada evaluation_interval: cada serie que devuelve es una alerta activa.	`ClassifierHigh5xxRate`
pending → firing	pending mientras la condición no cumple el for completo; firing cuando sí. Si deja de cumplirse, vuelve a inactive.	`for: 5m`
ALERTS	Serie que escribe Prometheus por cada alerta activa, con alertstate pending o firing.	`ALERTS{alertname="ClassifierHigh5xxRate"}`
alerting: alertmanagers	A qué Alertmanager manda Prometheus las alertas en firing; las reenvía cada minuto mientras sigan.	`no está en el prometheus.yml de C01`
Alertmanager	Agrupa, quita duplicados, silencia, inhibe (calla una alerta si otra más grave ya avisa) y elige el receiver según los labels (severity).	`page a quien está de guardia, ticket al equipo de ML`
Receiver y runbook	El destino del aviso, y el link con los pasos a seguir (runbook), que viaja en las annotations de la alerta.	`runbook_url en las 6 alertas de C01`
Grafana	Una rama aparte: consulta la misma base con PromQL para dibujar. En C01 no participa del aviso (Grafana también puede evaluar alertas, pero C01 las tiene en Prometheus).	`dashboard de servicio de C01`



1 Código TU SERVICIO

PYTHON

```
REQUESTS.labels(status="503", …).inc()
```

Suma 1 en la memoria del proceso: microsegundos, sin red.

2 /metrics TU SERVICIO

TERMINAL

```
curl -s localhost:8080/metrics | grep 'status="503"'
```

El valor acumulado, en texto, cuando alguien lo pide.

3 Scrape PROMETHEUS

PROMQL `up{job="gateway"}`

Cada 15 s Prometheus lee cada pod (10 s de timeout) y escribe up.

4 TSDB PROMETHEUS

PROMQL

```
prometheus_tsdb_head_series
```

La muestra entra a su serie: memoria (head) y WAL en disco; bloques de 2 h; 15 días de retención.

5 PromQL PROMETHEUS

PROMQL

```
sum(rate(classifier_requests_total{status=~"5.."}[5m])) / sum(rate(classifier_requests_total{status!~"4.."}[5m]))
```

Tasas en ventanas de 5 min: un salto a 10 % de errores cruza el 1 % en ≈ 30 s.

6 Regla PROMETHEUS

REGLA `ClassifierHigh5xxRate`

Se evalúa cada 15 s (evaluation_interval). La primera vez que la expr devuelve algo: pending.

7 for PROMETHEUS

PROMQL

```
ALERTS{alertname="ClassifierHigh5xxRate"}
```

5 min seguidos en pending → firing. Si la condición se apaga antes, vuelve a inactive y no avisa.

8 Envío PROMETHEUS

YAML `alerting: alertmanagers`

Prometheus manda las alertas en firing a Alertmanager y las reenvía cada 1 min mientras sigan. En el compose de C01, este eslabón no existe.

9 Alertmanager ALERTMANAGER

TERMINAL

```
curl -s localhost:9093/api/v2/alerts
```

Agrupa por labels, espera group_wait (30 s) por si llegan más alertas del mismo incidente, aplica silencios e inhibiciones y elige el receiver por severity.

10 Aviso ALERTMANAGER

YAML `severity: page` → receiver de guardia

PagerDuty, Slack o email, con summary, description y runbook_url. Con 10 % de errores, ≈ 6.5 min después del primer error en el peor caso.

RAMA APARTE **Grafana**

PROMQL

```
histogram_quantile(0.99, sum by (le) (rate(classifier_request_latency_seconds_bucket[5m])))
```

Consulta la misma base cada vez que refresca un panel. Si Grafana se cae, las alertas de Prometheus siguen.

PYTHON una línea de Python de tu servicio · TERMINAL un comando: va en la terminal

PROMQL una consulta: va en la caja de consultas de Prometheus (o en Grafana) · REGLA el nombre de una regla de alerta

YAML una clave de la configuración (YAML)

De la línea 244 de main.py al aviso: diez eslabones, con los tiempos de C01 y una rama aparte para Grafana, que lee la misma base y no participa del aviso. En el compose local la cadena se corta en /alerts: no hay Alertmanager.

EJEMPLO REAL · LA ALERTA DE 5XX DE C01: LA EXPRESIÓN, EL FOR Y LO QUE VIAJA CON EL AVISO

YAML reference/c01/monitoring/alerts.yml (líneas 41–60)

```yaml
    # Errores del servidor > 1 % de los requests válidos: los 4xx no cuentan ni en el numerador ni en el
    # denominador (si no, una ráfaga de 4xx diluye los 5xx).
    # Punto ciego: si TODOS los gateways quedan NotReady, el ALB responde 503 sin llegar al
    # gateway y esta métrica no lo ve. Lo cubren TritonNotReady y el HTTPCode_ELB_5XX_Count del ALB.
    - alert: ClassifierHigh5xxRate
      expr: |
        (
          sum(rate(classifier_requests_total{status=~"5.."}[5m]))
            /
          sum(rate(classifier_requests_total{status!~"4.."}[5m]))
        ) > 0.01
```

```
          and sum(rate(classifier_requests_total[5m])) > 1
        # La segunda condición exige ≥ 1 req/s: con tráfico casi nulo, 1 error de 20 dispararía un page falso
        for: 5m
        labels:
          severity: page
        annotations:
          summary: "Más de 1 % de 5xx en /v1/classify"
          description: "Tasa de 5xx = {{ $value | humanizePercentage }}. 503 = Triton caído o rechazando; 504 = Triton lento (mira TritonQueueTimeHigh)."
          runbook_url: "https://runbooks.example.com/classifier/high-5xx-rate"
```

- Eslabón 1: cada request suma 1 en `REQUESTS.labels(status=str(status), source=source, model_version=MODEL_VERSION).inc()` (main.py, línea 244, en el `finally`); los 422 los cuenta aparte el handler de errores de validación (línea 329), porque FastAPI los rechaza antes de llamar a `classify`. Eslabones 2 y 3: Prometheus lee `gateway:8080/metrics` cada 15 s, según `prometheus.yml`.
- La `expr` es PromQL sobre esas series: 5xx por segundo ÷ requests válidos por segundo (sin 4xx), en ventanas de 5 min. La segunda condición pide al menos 1 req/s: con 20 requests de madrugada, un solo error sería el 5 %.
- `for: 5m`: la alerta pasa a **pending** en la primera evaluación en que la expr devuelve algo, y a **firing** cuando lleva 5 min así. Ojo: la ventana de 5 min estira cada pico. Un pico de 2 min con 10 % de errores deja la fracción sobre 1 % durante ≈ 6 min, así que dispara igual; uno de 2 min con 2 % nunca pasa el 1 %.
- `severity: page` es solo un label: lo que lo convierte en «despertar a alguien» es la ruta de Alertmanager (sección 8). `{{ $value | humanizePercentage }}` escribe el valor de la expr en el texto del aviso, y `runbook_url` dice qué hacer.
- En el compose de C01 la cadena se corta antes: `prometheus.yml` no tiene bloque `alerting:` ni hay Alertmanager, así que la alerta llega a firing en `/alerts` y ahí se queda. En EKS, kube-prometheus-stack conecta Prometheus con su Alertmanager; las rutas y los receivers los escribe tu equipo.

CÓMO RECORDARLA

El código cuenta, Prometheus pregunta y guarda, PromQL calcula, la regla espera (`for`) y Alertmanager decide a quién avisar; Grafana solo mira. Si un aviso no llegó, recorre la cadena en orden: ¿sube el número en `/metrics` ? ¿ `up` = 1? ¿la expr devuelve algo? ¿pending o firing? ¿llegó a Alertmanager?

TRAMPA

Esperar el aviso en el instante del primer error. Se suman la ventana del `rate` , el scrape, la evaluación, el `for` y el `group_wait` : con 10 % de errores, ≈ 6.5 min para la alerta de 5xx de C01. Y creer que `firing` en `/alerts` significa que alguien se enteró: sin bloque `alerting:` y sin una ruta en Alertmanager, la alerta dispara y nadie recibe nada. El compose de C01 está así.

En el curso: C01 · Monitoreo, alertas y reentrenamiento · M12 · Prometheus, Grafana y alertas por SLO · M12 · Dashboards, runbooks y on-call · C01 · Archivos de referencia · Hoja · Del dato a producción · la estación de monitoreo

1 Qué es el monitoreo y qué hacen Prometheus y Grafana

¿Para qué se monitorea un sistema, qué diferencia hay entre métricas, logs y trazas, y quién hace qué entre Prometheus, Alertmanager y Grafana?

SINTAXIS CON VARIABLES

`curl -s` *servicio* `:` *puerto* `/metrics`

Lo que expone un servicio para Prometheus: un **endpoint** (una URL) con sus números en texto plano. El servicio no le manda nada a nadie: espera a que lo lean.

`docker run -d --name prom-lab -p 9090:9090 prom/prometheus:` *version*

Prometheus en tu Mac en un comando. Sin config propia usa la de ejemplo que trae la imagen: se lee a sí mismo cada 15 s. Abre `http://localhost:9090` y consulta `up`.

`rate(prometheus_http_requests_total[1m])`

Una primera consulta de **PromQL** (el lenguaje de consultas de Prometheus) sobre las métricas del propio Prometheus: cuántos requests por segundo recibe su servidor web (la UI, la API y su propio /metrics), por ruta (`handler`) y código (`code`).

`http://localhost:9090` · `http://localhost:9093` · `http://localhost:3000`

Las tres webs: Prometheus (consultas, alertas, targets), Alertmanager (avisos agrupados y silencios) y Grafana (dashboards). Grafana no guarda métricas: le pregunta a Prometheus con PromQL.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
servicio	dónde está el target	`gateway en el compose de C01`
puerto	puerto donde sirve /metrics	`8080`
version	tag fijo de la imagen	`v3.13.3, la de C01`
λ	requests por día del servicio	`10 M en C01`
$B_{\text{línea}}$	bytes de una línea de log	`≈ 300 B en C01`
N_{series}	series que guarda Prometheus del gateway	`≈ 615 con 3 pods (sección 3)`
Δ_{scrape}	intervalo de scrape	`15 s → 5,760 rondas por día`

FÓRMULAS

$$\text{logs por día} = \lambda \times B_{\text{línea}} = 10\ \text{M} \times 300\ \text{B} \approx 3\ \text{GB}$$

$$\text{muestras por día} = N_{\text{series}} \times \frac{86{,}400\ \text{s}}{\Delta_{\text{scrape}}} = 615 \times 5{,}760 \approx 3.5\ \text{M} \qquad (\times 1\text{–}2\ \text{B} \approx 3.5\text{–}7\ \text{MB})$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Monitoreo	Saber, sin estar mirando, si el sistema funciona, y que te avise cuando no.	`page si el p99 pasa de 200 ms durante 10 min`
Métrica	Un número medido en el tiempo y agregado: cuántos, qué tan rápido, cuántos errores. Barata y fácil de alertar.	`classifier_requests_total`
Log	Un evento con detalle, normalmente una línea por request. Responde qué le pasó a ese request.	`la línea JSON del gateway, ≈ 300 B`
Traza (trace)	El viaje de un request por varios servicios, partido en tramos con inicio y fin (spans).	`gateway → Triton, muestreada ≈ 1 % (propuesta del curso)`
Prometheus	El servidor que lee métricas (scrape), las guarda como series de tiempo y evalúa reglas. Un binario autónomo, con su disco local.	`prom/prometheus:v3.13.3 en el compose de C01`
Pull (scrape)	Prometheus pregunta a cada target cada cierto tiempo; el target solo contesta.	`GET /metrics cada 15 s`
Target · job · instance	Target: una URL que leer. Job: el grupo de targets con el mismo propósito. Instance: cuál de ellos.	`job="gateway", instance="10.0.3.17:8080" (ilustrativo)`
Serie de tiempo (time series)	La historia de un solo número: un nombre más un juego de labels, con una muestra por scrape.	`classifier_requests_total{status="503", source="web", …}`
TSDB	La base de datos de series de Prometheus (time series database), en el disco local: 15 días por defecto, sin réplica.	`/prometheus en la imagen de Docker`

PIEZA	QUÉ HACE	EJEMPLO
PromQL	El lenguaje para consultar series: tasas, sumas, cuantiles. Lo usan los dashboards, las reglas y KEDA.	`sum(rate(classifier_requests_total[5m]))`
Reglas (rules)	Consultas que Prometheus evalúa solo, cada cierto tiempo: las de grabación guardan un resultado; las de alerta disparan.	`1 de grabación y 6 alertas en alerts.yml`
Alertmanager	Recibe las alertas, las agrupa, silencia e inhibe, y decide a quién avisar y cómo (page o ticket).	`lo trae kube-prometheus-stack`
Grafana	Dibuja dashboards consultando a Prometheus con PromQL. No guarda métricas.	`servicio, Triton y GPU, modelo y negocio (el curso)`
Exporter · Pushgateway	Exporter: traduce las métricas de algo que no habla Prometheus. Pushgateway: un buzón donde un job batch (un trabajo que corre y termina) deja sus números para que Prometheus los lea.	`node_exporter · el PSI del job diario de drift (M12)`

Pull: Prometheus hace la ronda

Prometheus `:9090` pasa por cada target cada **15 s**, pide `/metrics` y espera hasta **10 s**. En cada visita anota `up` :

CADA RONDA, EN SEGUNDOS	0	15	30	45	60
gateway · 3 pods `:8080/metrics job="gateway"`	1	1	1	1	1
triton · 2 pods `:8002/metrics job="triton"`	1	1	1	1	1
kube-state-metrics `:8080/metrics`	1	1	1	1	1
node_exporter · 1 por nodo `:9100/metrics`	1	1	1	1	1
un pod del gateway que acaba de morir `:8080/metrics job="gateway"`	0	0	0	0	0

Prometheus → GET /metrics → target target → 200 con el texto

Prometheus abre la conexión y pregunta; el target solo contesta. Si no contesta en 10 s, Prometheus anota up = 0: el silencio también es un dato.

Push: (Pushgateway u OTLP)

QUIÉN ABRE LA CONEXIÓN: cada proceso manda sus números a un receptor (OTLP es el protocolo de OpenTelemetry para eso)

SI ESTOS MISMOS MANDARAN	0	15	30	45	60
gateway · 3 pods `:8080/metrics job="gateway"`	↑	↑	↑	↑	↑
triton · 2 pods `:8002/metrics job="triton"`	↑	↑	↑	↑	↑
kube-state-metrics `:8080/metrics`	↑	↑	↑	↑	↑
node_exporter · 1 por nodo `:9100/metrics`	↑	↑	↑	↑	↑
un pod del gateway que acaba de morir `:8080/metrics job="gateway"`					

CUÁNDO CONVIENE: jobs batch que terminan antes del próximo scrape

EL PROBLEMA: un servicio muerto solo deja de mandar: nadie distingue eso de «no hubo tráfico»

up = 1: contestó up = 0: no contestó, y queda anotado con push: mandó con push: nada (¿murió o no hubo tráfico?)

En EKS, el label `job` de cada target del Operator es el nombre de su Service: `gateway` y `triton` en C01, igual que en el compose local.

Pull: Prometheus visita cada target cada 15 s y anota lo que encuentra, también el silencio (up = 0). Push: cada servicio manda sus números, y uno muerto simplemente deja de mandar. Pods y puertos de C01 en EKS; el gateway caído es ilustrativo.

EJEMPLO REAL · QUIÉN LE PREGUNTA A QUIÉN: EL SERVICEMONITOR DEL GATEWAY DE C01

YAML reference/c01/k8s/monitoring.yaml (líneas 1–20: el encabezado y el ServiceMonitor del gateway)

```yaml
# Scrape con el Prometheus Operator (kube-prometheus-stack) (caso c01). Sin esto, Prometheus nunca
# lee el gateway ni Triton: la consulta de KEDA vuelve vacía (k8s/keda-triton.yaml la marca como
# error), Triton se queda en 2 réplicas y ninguna alerta evalúa. Falla en silencio.
# Las reglas de monitoring/alerts.yml van aparte, como PrometheusRule (el Operator no lee archivos
# de reglas): el comando está en el README.
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: gateway
  namespace: classifier
  labels:
    release: kube-prometheus-stack  # = serviceMonitorSelector de tu Prometheus (nombre del release de Helm)
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: gateway  # labels del Service de k8s/gateway.yaml
  endpoints:
    - port: http  # nombre del puerto del Service: 8080/metrics del gateway (no expuesto en el Ingress)
```

```
    path: /metrics
    interval: 15s  # igual que en local: suficiente para alertas de 5–30 min
```

- Pull en una frase: el gateway no sabe que Prometheus existe. Solo atiende `GET /metrics` en su puerto 8080; quien decide qué leer y cada cuánto es Prometheus. Este objeto es la forma de decírselo en Kubernetes.
- «Falla en silencio»: si nadie le dice a Prometheus que lea un servicio, no hay error en ningún lado. No hay series ni `up`, y una alerta sobre una serie que no existe no dispara: su expresión no devuelve nada.
- Un **ServiceMonitor** es un objeto del Prometheus Operator (un programa que corre en el cluster, vigila estos objetos y con ellos arma la config de Prometheus): busca los Services con esas labels y lee el puerto llamado `http` de cada pod detrás. El label `release: kube-prometheus-stack` hace que tu Prometheus lo tome en cuenta. La sección 11 lo cuenta entero.
- El pull también se ve en la red: la NetworkPolicy de Triton (la regla de red de Kubernetes que dice quién puede conectarse; `k8s/triton.yaml`) solo deja entrar al puerto 8002 desde el namespace `monitoring`, porque es Prometheus el que abre la conexión. Con push haría falta lo contrario: dejar salir a Triton hacia el receptor.
- Lo que Prometheus no hace: guardar eventos uno por uno (para eso están los logs) ni dar números exactos al 100 %, como para cobrar por request; su documentación lo dice. Es un sistema de números agregados, pensado para ser lo que consultas durante una caída: cada servidor es independiente y no depende de otra base ni de otro servicio para guardar.

CÓMO RECORDARLA

Prometheus es **una ronda de enfermería cada 15 s**: pasa por cada cuarto (target) y anota los signos vitales; si en un cuarto nadie contesta, anota eso también (`up = 0`). Alertmanager llama a quien está de guardia; Grafana es el tablero de la pared. **Métricas para saber que algo anda mal, logs para saber qué le pasó a un request, trazas para saber dónde se fue el tiempo.**

TRAMPA

Pensar que tu servicio «manda» sus métricas a Prometheus. Con pull, si nadie le dice a Prometheus que lo lea (falta el ServiceMonitor, el puerto se llama distinto, el label `release` no coincide), no hay error en ningún lado: no hay series, ninguna alerta evalúa y KEDA no escala (en C01, Triton se queda en 2 réplicas). Lo primero que miras es si existe `up` para ese job.

En el curso: M12 · Por qué un modelo falla en silencio · M12 · Métricas de sistema: golden signals con detalles de ML · M12 · Prometheus, Grafana y alertas por SLO · Herramientas · Monitoreo: sistema y modelo

2 El modelo de datos: nombre, labels, series y muestras; counter, gauge, histogram y summary; nombres y cardinalidad

¿Qué guarda Prometheus exactamente, qué tipo de métrica uso para cada cosa y cómo evito que la cantidad de series explote?

SINTAXIS CON VARIABLES

metrica **{** *label* **="** *valor* **",** *label2* **="** *valor2* **"}** *numero*

Una línea del formato de texto = una **muestra** (sample: un número en un instante) de una **serie** (time series: la historia de un número). La serie es el nombre más el juego exacto de **labels** (etiquetas clave=valor). Cambia, agrega o quita un label y es otra serie.

```
# TYPE metrica counter|gauge|histogram|summary
```

El tipo viaja como comentario: lo usan la librería que mide y quien lee el texto. El servidor guarda todas las series igual, como números con hora (salvo los native histograms): eres tú quien elige `rate()` para un counter o la lectura directa para un gauge.

```
rate( contador _total[ ventana ])
# counter: cuánto sube por segundo
gauge
# gauge: lo que marca ahora
max_over_time( gauge [ ventana ])
# gauge: el pico de la ventana
rate( hist _sum[ ventana ]) /
rate( hist _count[ ventana ])  # histogram: el
promedio
```

Cada tipo se lee distinto. El promedio de un histograma sale de `_sum` ÷ `_count`; los percentiles, de los `_bucket` (sección 6).

```
prefijo _ que_mide _ unidad _total     # counter
prefijo _ que_mide _ unidad            # gauge o
histogram
```

Nombres: un prefijo del dominio, lo que mide, la unidad **base** en plural (`seconds`, `bytes`; las proporciones de 0 a 1, `ratio`) y `_total` si es un counter. Nunca milisegundos en una métrica y segundos en otra.

```
topk(10, count by (__name__)
({__name__=~"prefijo _.*"}))
```

¿Qué métrica tiene más series? Cuenta series por nombre. En la web, Status → TSDB status (el estado de la base de series) muestra el ranking de todo Prometheus. Usa siempre un filtro: sin él recorre todas las series.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
metrica	nombre de la métrica	`classifier_requests_total`
label, *valor* · *label2*, *valor2*	dimensiones y sus valores: la serie lleva todos sus labels	`status="503", source="web"`
numero	el valor: siempre un float de 64 bits	`17.0`
contador, *gauge*, *hist*	nombres base de cada tipo	`classifier_requests` · `nv_gpu_utilization` · `classifier_request_latency_seconds`
ventana	cuánto mira hacia atrás	`5m`
prefijo, *que_mide*, *unidad*	las partes del nombre	`classifier` · `request_latency` · `seconds`
N_{series}	series que puede tener una métrica	`72 para classifier_requests_total por pod, como máximo`
L_i	el label i; |L_i| es cuántos valores distintos puede tomar	`status 9, source 4, model_version 1 por pod`
b, v	buckets con borde finito y combinaciones de labels; el + 4 son +Inf, _sum, _count y _created	`11 y 4 en la latencia de C01`
$\bar{x}$	promedio de lo observado en la ventana	`latencia media por request`

FÓRMULAS

$$N_{\text{series}} \le \prod_i |L_i| \qquad \text{classifier_requests_total: } 9 \times 4 \times 1 = 36 \text{ combinaciones (72 series con _created)}$$

$$\text{series de un histograma} = (b + 4) \times v = (11 + 4) \times 4 = 60 \text{ por pod}$$

$$\bar{x} = \frac{\text{rate(x_sum[5m])}}{\text{rate(x_count[5m])}} \qquad \text{(promedio de lo observado en la ventana)}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Muestra (sample)	Un número float64 (decimal de 64 bits) con su hora en milisegundos. Es lo que deja cada scrape en cada serie.	`17.0 a las 14:03:15.123`
Serie (time series)	La historia de un número: nombre + juego exacto de labels. Cambiar un label crea otra serie.	`classifier_requests_total{status="200", source="web", …}`

PIEZA	QUÉ HACE	EJEMPLO
Nombre de métrica	Qué se mide. Letras, dígitos, _ y :, sin empezar con un dígito (los : quedan para las reglas de grabación). Desde Prometheus 3.0 acepta UTF-8, pero el formato clásico evita sorpresas.	`classifier_request_latency_seconds`
Label	Una dimensión clave=valor para partir la métrica. Los que empiezan con __ son internos; un valor vacío equivale a no tener el label.	`source="mobile"`
Counter	Solo sube; vuelve a 0 si el proceso reinicia. Nombre con _total. Se lee con rate() o increase().	`classifier_requests_total; nv_inference_count (Triton no le pone _total)`
Gauge	Sube y baja: vale lo que marca ahora. Se lee directo o con *_over_time.	`nv_gpu_utilization, kube_deployment_status_replicas_available`
Histogram	Cuenta observaciones en cajones acumulados (_bucket con le) y guarda _sum y _count. Se suma entre pods.	`classifier_request_latency_seconds`
le (less or equal)	El borde de cada cajón: le="0.2" cuenta todo lo que tardó 0.2 s o menos. El último, le="+Inf", es igual a _count.	`11 bordes de 0.01 a 2.5 s en C01`
Summary	Calcula cuantiles en el proceso (quantile="0.99"). Esos cuantiles no se pueden sumar ni promediar entre pods. El de Python solo da _sum y _count.	`C01 no usa ninguno`
Native histogram	Histograma con buckets automáticos, guardado en una sola serie por combinación de labels. Estable desde 3.8, pero hay que activar scrape_native_histograms y que el cliente lo emita: el Histogram de Python emite solo clásicos.	`C01 usa histogramas clásicos`
_created	La hora en que nació la serie. prometheus_client la agrega a counters, histogramas y summaries como una serie más (se apaga con PROMETHEUS_DISABLE_CREATED_SERIES=True).	`classifier_requests_created`
Unidades base	Segundos, bytes y proporciones de 0 a 1, con la unidad al final del nombre.	`_seconds, _bytes; nv_inference_queue_duration_us (µs) es la excepción de Triton`
Métrica _info	Un gauge que siempre vale 1 y lleva datos en sus labels: versión, build.	`python_info{implementation="CPython", …, version="3.13.16"} 1.0`
Cardinalidad	Cuántas series tiene una métrica: hasta el producto de los valores de cada label. Cada serie ocupa memoria e índice en Prometheus. La guía oficial: si puedes, menos de 10 series por métrica (y pocas métricas en todo el sistema que pasen de eso); nunca ids ni valores sin límite.	`request_id como label: una serie nueva por request`

Counter · el odómetro: solo sube

`classifier_requests_total` (un pod)



`001060` → `000003`

el proceso nuevo empieza de 0

`rate(x[2m])` **0.371/s** corrige el reinicio

`increase(x[2m])` **44.6**
requests en la ventana

última – primera **−9.72/s**
negativo: no ve el reinicio

Como cambiar de auto a mitad de viaje: el odómetro nuevo empieza en 0. rate() suma lo que marcaba el viejo antes de caer (1,060) y no da negativo.

Gauge · el medidor de gasolina: sube y baja

requests en vuelo (gauge)



`avg_over_time(x[2m])` **3.62**
lo que marcó en promedio: la línea punteada

`max_over_time(x[2m])` **6**
el pico, en el círculo

Como el medidor de gasolina: se lee lo que marca. rate() sobre esto da 0.158/s, un número sin sentido.

Histogram · cajones acumulados: «hasta»

`classifier_request_latency_seconds` (1,000 requests simulados, mediana ≈ 40 ms)

le="0.01"	4
le="0.025"	175
le="0.05"	702
le="0.075"	897
le="0.1"	968
le="0.15"	995
le="0.2"	999
le="0.3"	1,000
le="0.5"	1,000
le="1.0"	1,000
le="2.5"	1,000
le="+Inf"	1,000

`_sum` **44.7 s** `_count` **1,000**
= `le="+Inf"`

Cada cajón cuenta todo lo que cayó por debajo de su borde (le): por eso los números solo suben y +Inf = _count. El p99 cae entre 0.1 y 0.15 s (sección 6).

SUMMARY: No sale en C01: trae el cuantil ya calculado en cada pod (quantile="0.99") y dos pods no se pueden mezclar.

counter · gauge · histogram: lo nuevo de cada cajón · lo que ya traía del cajón anterior · la ventana de rate()

Los tres tipos que usa C01, a escala. Counter: un pod que reinicia entre t = 157 y 172 s; rate() en 2 min da 0.371/s. Gauge: requests en vuelo, que suben y bajan. Histogram: 1,000 latencias simuladas en los buckets de C01, acumulados. El counter y el gauge son los del lab de esta sección.

EJEMPLO REAL · LAS SEIS MÉTRICAS DEL GATEWAY DE C01: NOMBRES, TIPOS, LABELS Y BUCKETS

PYTHON reference/c01/gateway/app/main.py (líneas 62–88)

```python
# ---------------------------------------------------------------------------- métricas
# Nombres = contrato con monitoring/alerts.yml: si cambias uno aquí, rompes una alerta en silencio.
# Labels acotados: status (≈ 10 códigos) × source (4) × model_version (1–2 durante un canary)
REQUESTS = Counter(  # el cliente de Python agrega el sufijo _total; el nombre expuesto es este mismo
    "classifier_requests_total", "Requests a /v1/classify por código HTTP, caller y modelo",
    ["status", "source", "model_version"],  # status: "200", "503"… · source: mobile/web/internal/other
)
LATENCY = Histogram(  # latencia server-side; los buckets rodean al SLO (0.2 s) para que el p99 sea preciso
    "classifier_request_latency_seconds", "Latencia total de /v1/classify", ["source"],
    buckets=(0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.5, 1.0, 2.5),
)
PREPROCESS = Histogram(  # separar preprocess e inferencia dice a qué capa escalar cuando sube la latencia
    "classifier_preprocess_seconds", "Decode + resize + normalize (CPU)",
    buckets=(0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.25, 0.5),
)
INFERENCE = Histogram(  # ida y vuelta a Triton: red + cola de dynamic batching + cómputo en GPU
    "classifier_inference_seconds", "Llamada a Triton (round trip)",
    buckets=(0.005, 0.01, 0.02, 0.035, 0.05, 0.075, 0.1, 0.15, 0.3),
)
TOP1 = Histogram(  # distribución de confianza: la señal de drift más barata que existe (sin labels)
    "classifier_top1_score", "Score top-1 tras softmax", ["model_version"],  # separa canary de estable
    buckets=tuple(round(0.1 * i, 1) for i in range(1, 11)),  # 0.1 … 1.0 → el bucket le="0.3" alimenta una alerta
)
IMAGE_BYTES = Histogram(  # tamaño de entrada: explica cambios de latencia (p. ej. la web empieza a subir 12 MP)
    "classifier_image_bytes", "Bytes de la imagen recibida",
    buckets=(1e4, 5e4, 1e5, 2.5e5, 5e5, 1e6, 2.5e6, 5e6, 1e7),
)
```

- Un counter y cinco histogramas, ningún gauge. Los nombres siguen la guía: prefijo `classifier_`, unidad base al final (`_seconds`, `_bytes`) y `_total` en el counter. `classifier_top1_score` no lleva unidad: es una proporción de 0 a 1.
- «Contrato con alerts.yml» (línea 63): las alertas buscan estos nombres exactos. Si renombras uno, la alerta no falla: su expresión deja de encontrar series y nunca dispara.

- La cardinalidad está en el comentario de la línea 64: status × source × model_version. Para la latencia, (11 + 4) × 4 = 60 series por pod; con el counter y los otros cuatro histogramas, hasta 184 series `classifier_*` por pod (sección 3).
- El sufijo: `prometheus_client` quita `_total` del nombre que le das y lo vuelve a poner al exponer, así que la serie de nacimiento se llama `classifier_requests_created`, sin `_total` (verificado con prometheus-client 0.26.0, la que fija C01).
- `TOP1` lleva `model_version` para comparar el canary (la versión nueva, que recibe una parte del tráfico) con el estable. Sus bordes salen de `round(0.1 * i, 1)`: 0.1 … 1.0 exactos, y `le="0.3"` es el cajón que lee la regla de grabación de baja confianza.

CÓMO RECORDARLA

Un **counter es el odómetro** del auto: solo sube, y `rate()` lo convierte en velocidad. Un **gauge es el medidor de gasolina**: lo lees tal cual. Un **histogram son cajones acumulados** (`le` = «hasta»), y un **summary** es un p99 ya cocinado en cada pod, que no se mezcla con el de otro.

TRAMPA

Pasar un gauge por `rate()` o mostrar un counter crudo. `rate()` toma cada bajada de un gauge por un reinicio y suma basura (en el lab, 0.158 «requests en vuelo por segundo»); un counter crudo en un panel solo sube y cae a 0 en cada deploy. Y un label con valores sin límite (`request_id`, `user_id`): cada valor nuevo es una serie nueva que Prometheus indexa y guarda.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Métricas de sistema: golden signals con detalles de ML · M3 · p50, p95, p99: por qué el promedio miente · C01 · Monitoreo, alertas y reentrenamiento · Hoja · El gateway · observabilidad

3 Instrumentar tu código y usar exporters: prometheus_client, el gateway de C01, Triton, node_exporter, kube-state-metrics y cAdvisor

¿Cómo hago que mi servicio exponga métricas, dónde van las llamadas en el código y qué métricas me dan otros sin escribir nada?

SINTAXIS CON VARIABLES

```
from prometheus_client import Counter, Gauge, Histogram
CONTADOR = Counter("prefijo_cosa_total", "ayuda", ["label"])
HISTO = Histogram("prefijo_cosa_seconds", "ayuda", buckets=(bordes))
GAUGE = Gauge("prefijo_cosa", "ayuda")
```

Se definen **una vez**, al importar el módulo, y quedan en el registro global (`REGISTRY`); definirlas dos veces da `DuplicateTimeseries` («Duplicated timeseries in CollectorRegistry», un `ValueError`). Sin `buckets` (los bordes de los cajones), el histograma usa los default (de 5 ms a 10 s), pensados para «requests web típicos», no para tu **SLO** (el objetivo de servicio: p99 ≤ 200 ms en C01).

```
t0 = time.perf_counter()
status = "500"
try:
    ...  # el trabajo
    status = "200"
finally:
    CONTADOR.labels(label=status).inc()
    HISTO.observe(time.perf_counter() - t0)
```

Las llamadas van en el `finally`: así cuentan también los errores. `.labels()` elige la serie, `.inc()` suma 1 y `.observe()` mete el valor en su bucket. Un gauge usa `.set()`, `.inc()` / `.dec()` o el decorador `track_inprogress()`, que sube al entrar y baja al salir.

```
@app.get("/metrics", include_in_schema=False)
def metrics() -> Response:
    return Response(generate_latest(),
media_type=CONTENT_TYPE_LATEST)
```

```
start_http_server(puerto)  # en un script o un worker sin web
```

Exponer: en FastAPI, una ruta propia como la de C01 (montar `make_asgi_app()` en `/metrics` responde 307, una redirección, hacia `/metrics/`). En un script, `start_http_server` abre un servidor de `/metrics` en un hilo aparte.

```
PROMETHEUS_MULTIPROC_DIR=carpeta uvicorn app --workers n
```

Con varios procesos por pod (uvicorn con `--workers N`, o gunicorn) cada uno tiene su propio registro: hace falta el **modo multiproceso** (una carpeta compartida, vaciada en cada arranque, y un `MultiProcessCollector` en la ruta de `/metrics`). C01 corre 1 worker por pod y escala con réplicas.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
CONTADOR, *HISTO*, *GAUGE*	variables de Python con cada métrica	REQUESTS y LATENCY en C01 (no tiene gauges)
prefijo, *cosa*	las partes del nombre	classifier · request_latency
ayuda	texto de # HELP	Latencia total de /v1/classify
label	dimensión con valores de una lista cerrada	status
bordes	bordes de los buckets, varios cerca del SLO	0.01, 0.025 … 2.5 (11 en C01)
puerto	puerto donde escucha /metrics	8080 en el gateway; 8000 en el lab
carpeta	carpeta del modo multiproceso, vacía al arrancar	/tmp/prom-multiproc
n, *app*	procesos del servidor y la app que corre	1 worker en C01 · app.main:app
version	tag de la imagen de Prometheus	v3.13.3
N_{pod}	series que expone un pod del gateway en régimen	200: 184 classifier_* + 16 del cliente, en Linux
n_{pods}	réplicas del gateway (las mueve el HPA, el autoscaler de Kubernetes)	de 3 a 12
$+5$	series que Prometheus agrega por target	up y las cuatro scrape_*

curl -s localhost:*puerto*/metrics | docker run --rm -i --entrypoint promtool prom/prometheus:*version* check metrics

`promtool`, la herramienta de línea de comandos de Prometheus, revisa con `check metrics` el texto que le llega por stdin (por el `|`) y marca problemas de estilo, como `counter metrics should have "_total" suffix` o `no help text`. Viene dentro de la imagen de Prometheus: no instalas nada.

FÓRMULAS

$$N_{\text{pod}} = \underbrace{2 \times 36}_{\text{requests}} + \underbrace{15 \times 4}_{\text{latencia}} + \underbrace{12 + 13 + 14 + 13}_{\text{preproceso, inferencia, top-1, bytes}} + \underbrace{16}_{\text{cliente}} = 200$$

$$N_{\text{servicio}} = n_{\text{pods}} \times (N_{\text{pod}} + 5) = 3 \times 205 = 615\text{a}12 \times 205 = 2{,}460$$

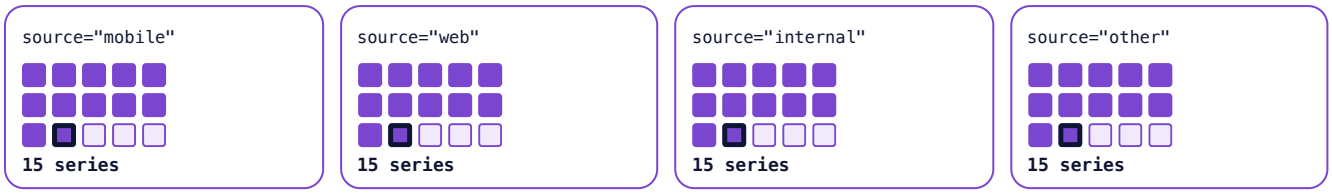
request_id como label: $15 \frac{\text{series}}{\text{request}} \times 10\ \text{M} \frac{\text{requests}}{\text{día}} = 150\ \text{M series nuevas al día}$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
prometheus_client	La librería oficial de Python: define métricas, las guarda en memoria y las escribe en el formato de texto.	**0.26.0 en C01, la última en octubre de 2026**
REGISTRY · generate_latest()	El registro global donde vive cada métrica; generate_latest() lo vuelca entero como texto.	**Response(generate_latest(), media_type=CONTENT_TYPE_LATEST)**
.labels(…)	Elige (o crea) la serie de esa combinación. Una serie con labels no existe hasta su primer uso: llámala sin .inc() al arrancar si quieres verla en 0.	**REQUESTS.labels(status="500", …)**
Counter: .inc()	Suma 1, o lo que le pases. count_exceptions() cuenta las excepciones de un bloque.	**REQUESTS…inc() en el finally de classify**
Histogram: .observe() · .time()	Mete una observación en su bucket y la suma a _sum y _count; .time() mide un bloque en segundos.	**LATENCY.labels(source=source).observe(elapsed)**
Gauge: .set() · track_inprogress()	Fija o mueve un valor que sube y baja; track_inprogress() cuenta lo que está en curso.	**C01 lleva state.inflight, pero no lo exporta**
Colectores por defecto	process_* (CPU, RAM, archivos abiertos; solo en Linux), python_gc_* y python_info, sin escribir nada.	**16 series en un pod de C01; 10 en tu Mac**
start_http_server(puerto)	Un servidor de /metrics en un hilo aparte, para scripts y workers sin web propia.	**start_http_server(8000) en el lab**
Modo multiproceso	Para varios procesos por pod: PROMETHEUS_MULTIPROC_DIR y MultiProcessCollector. Sin él, cada scrape cae en un proceso distinto.	**C01 lo evita con --workers 1**
Exporter	Un programa aparte que lee algo que no habla Prometheus y lo traduce a /metrics.	**node_exporter en :9100 lee /proc y /sys de cada nodo**
Métricas propias del software	Programas que ya exponen /metrics por su cuenta: no hace falta exporter.	**Triton en :8002 (nv_*); el propio Prometheus en :9090**
kube-state-metrics	Lee la API de Kubernetes y publica el estado de los objetos: réplicas, fases, reinicios.	**kube_deployment_status_replicas_available → TritonNotReady**
cAdvisor (dentro del kubelet)	El uso real de CPU, memoria y throttling (los frenos cuando un contenedor agota su cuota de CPU) de cada contenedor, que el kubelet (el agente de Kubernetes en cada nodo) sirve en /metrics/cadvisor.	**container_memory_working_set_bytes**
Pushgateway	Un buzón para jobs batch que terminan antes del próximo scrape: empujan su resultado y Prometheus lee el buzón. No es para servicios.	**el PSI del job diario de drift (M12)**

HISTOGRAM `classifier_request_latency_seconds`

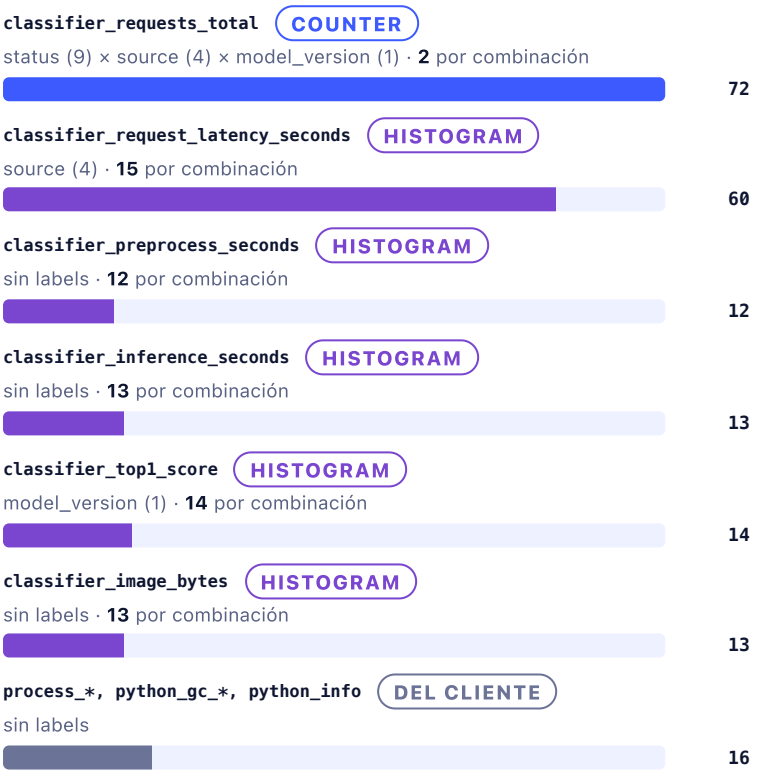
se abre por el label `source`, que tiene 4 valores:



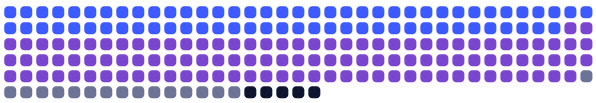
4 × 15 = **60 series** de una sola métrica, en un solo pod

`_bucket{le="0.01"} … _bucket{le="2.5"}` (11) · `_bucket{le="+Inf"}` · `_sum` · `_count` · `_created`

SERIES POR MÉTRICA EN UN POD (A ESCALA, MÁXIMO 72)



UN POD DEL GATEWAY, UN CUADRITO POR SERIE: 200 + 5 = 205



3 a 12 pods: de 615 a 2,460 series

Un label sin techo: `request_id`

2 series por request en un counter (_total y _created), 15 en un histograma: ≈ 1,600 nuevas en los primeros 10 s del lab

MEDIDO: lab: 144 series classifier_* con 4 status × 4 source; 154 en total en macOS (sin las 6 process_*)

counter · histogram · del cliente · las agrega Prometheus

Los 9 status son los que `main.py` puede anotar: 200, 400, 413, 415, 422, 499, 500, 503 y 504. Un label de lista cerrada multiplica; un id no tiene techo.

Cuántas series expone un pod del gateway de C01, contadas con las definiciones exactas de main.py y prometheus-client 0.26.0. Cada métrica se abre en una serie por combinación de labels; un histograma, en 15 por combinación (11 buckets + +Inf + _sum + _count + _created).

EJEMPLO REAL · MAIN.PY: LA RUTA /METRICS Y LAS DOS LLAMADAS DEL FINALLY

PYTHON reference/c01/gateway/app/main.py (líneas 30, 119–127 y 240–246; # … marca líneas omitidas)

```python
from prometheus_client import CONTENT_TYPE_LATEST, Counter, Histogram, generate_latest  # métricas Prometheus
# …
@app.get("/metrics", include_in_schema=False)
def metrics() -> Response:
    """Endpoint que scrapea Prometheus. Con 1 proceso por contenedor basta el registro por defecto.

    Verificado: app.mount("/metrics", make_asgi_app()) (lo que sugiere la doc) responde 307 →
    /metrics/ en Starlette; Prometheus sigue el redirect, pero cada scrape serían dos requests.
    Con gunicorn/uvicorn --workers N necesitarías el modo multiproceso de prometheus_client.
    """
```

```
    return Response(generate_latest(), media_type=CONTENT_TYPE_LATEST)  # formato de texto de Prometheus
# …
    finally:
        elapsed = time.perf_counter() - t0
        source = record["source"] if record["source"] in SOURCES else "other"  # cardinalidad acotada
        # Alimenta la alerta de 5xx y los paneles por caller y por modelo (canary vs estable)
        REQUESTS.labels(status=str(status), source=source, model_version=MODEL_VERSION).inc()
        if not 400 <= status < 500:  # los 4xx son rechazos rápidos: incluirlos "mejoraría" el p99 falsamente
            LATENCY.labels(source=source).observe(elapsed)
```

- La ruta es una función normal (`def` , no `async def`): FastAPI la corre en su threadpool (un grupo de hilos aparte, para no frenar al resto). Devuelve el registro entero con `CONTENT_TYPE_LATEST` , que en prometheus-client 0.26.0 es `text/plain; version=1.0.0; charset=utf-8` .
- Por qué no `make_asgi_app()` : montado en `/metrics` , Starlette responde 307 hacia `/metrics/` y cada scrape serían dos requests. Lo dice el docstring y se repitió con Starlette 1.7.0 (la de C01, que es quien redirige) y FastAPI 0.143.0: `307` con `location: …/metrics/` .
- El `finally` de `classify` corre siempre: el counter cuenta 200, 4xx y 5xx. Los 422 son la excepción: FastAPI los rechaza antes de llamar a `classify` , y los cuenta aparte el handler (la función que atiende ese error) de `RequestValidationError` , en la línea 329. `status` va como texto (`str(status)`): los valores de un label siempre son strings.
- `source` pasa por una lista cerrada antes de ser label: un header libre (`x-client-source`) crearía una serie por cada valor que invente un cliente. Los 4xx no entran al histograma de latencia: un rechazo de 1 ms bajaría el p99 sin que nadie esté mejor.
- Lo que el gateway no exporta: `state.inflight` (requests en vuelo) vive en la app pero no es un gauge, aunque el curso lo lista como señal de saturación del gateway. Para verlo en un panel haría falta un `Gauge` que se mueva junto con `state.inflight` .

CÓMO RECORDARLA

Define al importar, cuenta en el finally, expón en /metrics. Labels de una lista cerrada, buckets alrededor del SLO y un proceso por pod. Lo que no es tuyo (la máquina, Kubernetes, Triton) ya trae su exporter o su propio `/metrics` .

TRAMPA

Correr el servicio con varios workers (`--workers 4`) sin el modo multiproceso. Cada worker tiene sus propios counters y cada scrape cae en uno distinto: la serie salta, por ejemplo, 1,200 → 310 → 1,215, y `rate()` lee cada bajada como un reinicio y suma de más. Por eso C01 corre `--workers 1` y escala con réplicas. La otra: buscar la serie de los 500 antes del primer 500; una serie con labels no existe hasta su primer uso, y el panel dice «No data» en vez de 0.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Métricas de sistema: golden signals con detalles de ML · M7 · FastAPI + PyTorch: el punto de partida · C01 · Archivos de referencia · Hoja · El gateway · observabilidad · Hoja · De PyTorch a Triton · medir

4 Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

¿Cómo le dices a Prometheus qué leer, cada cuánto y con qué labels, y cómo sabes si lo está logrando?

SINTAXIS CON VARIABLES

```
global:
  scrape_interval: intervalo
  evaluation_interval: intervalo
rule_files:
  - reglas.yml
scrape_configs:
  - job_name: job
    metrics_path: ruta
    static_configs:
      - targets: ["host:puerto"]
        labels:
          label: valor
```

global pone los valores por defecto de todos los jobs (cada job puede cambiarlos). Cada **job** (grupo de targets con el mismo propósito) lleva un nombre único, que termina como label `job` en cada serie. `targets` son `host:puerto`, sin `http://` ni ruta: la ruta va en `metrics_path` (default `/metrics`).

```
    relabel_configs:
      - source_labels: [label_origen]
        regex: "regex"
        action: drop
      - target_label: label
        replacement: valor
```

Relabeling: reescribir los labels de cada target **antes** del scrape. `keep` y `drop` deciden si el target se lee; sin `action` es `replace`, que escribe `target_label`. El regex se ancla entero (default `(.*)`). Los labels que empiezan con `__` (`__address__`, los `__meta_*` del descubrimiento) se borran al terminar.

```
    metric_relabel_configs:
      - source_labels: [__name__]
        regex: "patron"
        action: drop
```

`metric_relabel_configs` corre **después** del scrape, sobre cada muestra y antes de guardarla: sirve para tirar series caras. No toca `up` ni las otras series que genera Prometheus.

```
docker run --rm -v "$PWD":/cfg:ro --entrypoint
promtool prom/prometheus:version check config
/cfg/prometheus.yml
```

promtool (el validador que viene en la imagen) revisa la sintaxis, que existan los archivos que nombra y que las reglas de `rule_files` sean válidas. Úsalo antes de cada cambio: un error aquí es un Prometheus que no arranca o una recarga que falla.

```
docker kill -s HUP contenedor
curl -X POST localhost:9090/-/reload   # solo con
--web.enable-lifecycle
```

Recargar sin reiniciar: la señal SIGHUP (el aviso de «relee tu config»), o un POST a `/-/reload` si Prometheus arrancó con `--web.enable-lifecycle`. Si la config nueva tiene un error, sigue la anterior y `prometheus_config_last_reload_successful` queda en 0.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
intervalo	cada cuánto lee y evalúa	15s en C01
reglas	archivo de reglas, relativo a prometheus.yml	alerts
job	nombre único del grupo de targets	gateway
ruta	ruta de las métricas en el target	/metrics (el default)
host:*puerto*	dónde está el target, visto desde Prometheus	gateway:8080 en el compose; host.docker.internal:8000 hacia tu Mac
label:*valor*	label fijo para todas las series de esos targets	env: lab
label_origen	label que miras para decidir; en Kubernetes, un __meta_*	__address__ · __meta_kubernetes_namespace
regex	expresión regular, anclada al inicio y al final	.*:8001
patron	nombres de métricas que no quieres guardar	go_.*
version	tag de la imagen de Prometheus	v3.13.3 (la de C01)
contenedor	nombre del contenedor de Prometheus	prom-lab
$\Delta_{timeout}$, Δ_{scrape}	scrape_timeout y scrape_interval	10 s y 15 s en C01
N_{series}	series que expone cada target, más las 5 que agrega Prometheus	205 por pod del gateway

FÓRMULAS

$$\Delta_{\text{timeout}} \le \Delta_{\text{scrape}} \qquad \text{si no lo escribes: } \Delta_{\text{timeout}} = \min(10\ \text{s},\ \Delta_{\text{scrape}})$$

$$\frac{\text{muestras}}{\text{s}} = \sum_{\text{targets}} \frac{N_{\text{series}}}{\Delta_{\text{scrape}}} = \frac{3 \times 205}{15\ \text{s}} = 41\ \text{muestras/s} \quad \text{(el gateway de C01 con 3 pods)}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
global	Valores por defecto para todos los jobs (intervalos y timeout), más los external_labels de este Prometheus.	**scrape_interval: 15s en C01**
external_labels	Labels que se pegan a lo que sale de este Prometheus: alertas, federación (otro Prometheus que lee de este) y remote write (copias a otra base). No aparecen en tus consultas locales.	**env: local en C01**
job_name	Nombre único del job; termina como label job en cada serie.	**job="gateway"**
static_configs.targets	Lista fija de host:puerto. Con http:// adelante, la config ni carga; si el job tiene relabel_configs, carga, pero ese target nunca se crea.	**["gateway:8080"]**
instance	Label que dice qué target fue: por defecto, el __address__ (host:puerto).	**instance="gateway:8080"**
metrics_path · scheme	Ruta y protocolo del scrape: /metrics y http por defecto.	**C01 escribe metrics_path: /metrics aunque es el default**
scrape_timeout	Cuánto espera cada scrape; no puede pasar el intervalo.	**10 s por defecto**
Descubrimiento (service discovery)	Que Prometheus encuentre los targets solo, sin lista fija. En Kubernetes, kubernetes_sd_configs con role pod, service, endpoints, endpointslice, node o ingress.	**en EKS lo arma el Prometheus Operator desde los ServiceMonitors (sección 11)**
relabel_configs	Reglas que reescriben los labels de cada target antes del scrape: keep, drop, replace, labelmap (copia labels con otro nombre)…	**tirar un namespace entero, renombrar instance**
__address__ · __meta_*	Labels temporales del target: la dirección y lo que sabe el descubrimiento. Se borran después del relabeling.	**__meta_kubernetes_pod_name**
metric_relabel_configs	Reglas que corren después del scrape, sobre cada muestra: tiran o renombran series antes de guardarlas.	**drop de go_.* en el lab**
honor_labels	Si el target trae un label que Prometheus también pone (job, instance): con false (default) el del target pasa a llamarse exported_job; con true gana el del target.	**true para el Pushgateway**
up y scrape_*	Series que escribe Prometheus en cada scrape: up, scrape_duration_seconds, scrape_samples_scraped, scrape_samples_post_metric_relabeling y scrape_series_added.	**up{job="triton"} == 0**
Status → Target health	La página /targets: cada target con su estado (UP o DOWN), labels, último scrape, duración y error. Status → Service discovery muestra los labels de antes del relabeling y los targets descartados.	**connection refused · context deadline exceeded**



1 Descubrimiento ANTES DEL SCRAPE

YAML

```
static_configs · kubernetes_sd_configs
```

Una lista fija (local) o la API de Kubernetes (EKS). Cada target nace con __address__ (host:puerto), job y, con descubrimiento, labels __meta_*.

2 relabel_configs ANTES DEL SCRAPE

YAML

```
action: keep · drop · replace · labelmap
```

Decide si el target se lee y con qué labels. Lo que empieza con __ se borra al terminar; instance = __address__ si nadie lo fijó.

3 Target listo ANTES DEL SCRAPE

TERMINAL

```
curl -s localhost:9090/api/v1/targets
```

discoveredLabels (antes) y labels (después del relabeling). Los descartados quedan como dropped en Status → Service discovery.

4 Scrape LA VISITA

HTTP

```
GET http://gateway:8080/metrics
```

Cada scrape_interval (15 s en C01), con scrape_timeout (10 s). El formato se negocia con el header Accept.

5 metric_relabel_configs DESPUÉS DEL SCRAPE

YAML `action: drop` sobre __name__

Corre sobre cada muestra leída: tira series caras antes de guardarlas. No toca up.

6 Límites DESPUÉS DEL SCRAPE

YAML

```
sample_limit · label_limit
```

Si una ronda los pasa, se descarta entera y up = 0. Default 0: sin límite.

7 up y scrape_* DESPUÉS DEL SCRAPE

PROMQL `up{job="gateway"}`

Al cerrar la ronda, Prometheus escribe up (1 si salió bien; 0 si el scrape falló o pasó un límite), scrape_duration_seconds, scrape_samples_scraped…

8 TSDB DESPUÉS DEL SCRAPE

PROMQL

```
prometheus_tsdb_head_series
```

Cada muestra entra a su serie (nombre + labels). Una combinación nueva de labels es una serie nueva en memoria.

YAML una clave de la configuración (YAML) TERMINAL un comando: va en la terminal HTTP el request que hace Prometheus
PROMQL una consulta: va en la caja de consultas de Prometheus (o en Grafana)

La vida de un target en cada ronda, del descubrimiento a la muestra guardada. Hay dos lugares para reescribir labels: relabel_configs, antes del scrape, y metric_relabel_configs, después.

EJEMPLO REAL · EL PROMETHEUS.YML DE C01: DOS JOBS, CADA 15 S

YAML reference/c01/monitoring/prometheus.yml (completo)

```yaml
# Configuración de Prometheus para correr el sistema en local con docker compose (caso c01).
# En Kubernetes NO se escribe a mano: el Prometheus Operator descubre los pods con
# ServiceMonitor/PodMonitor (o kubernetes_sd_configs), y en AWS puedes usar Amazon Managed
```

```yaml
# Prometheus. alerts.yml se reutiliza tal cual como spec de un PrometheusRule (ver README).

global:
  scrape_interval: 15s  # cada cuánto lee /metrics: suficiente para alertas de 5–30 min, barato en series
  evaluation_interval: 15s  # cada cuánto evalúa reglas y alertas
  external_labels:
    env: local  # en producción, env/cluster distinguen de dónde viene cada alerta

rule_files:
  - alerts.yml  # ruta relativa a este archivo: reglas de grabación + las 6 alertas del caso

scrape_configs:
  - job_name: gateway  # el label job="gateway" aparece en cada serie del gateway
    metrics_path: /metrics  # endpoint propio (sin redirect; ver main.py)
    static_configs:
      - targets: ["gateway:8080"]  # nombre del servicio en docker-compose.yml, puerto del gateway

  - job_name: triton  # métricas nv_* (cola, cómputo, batches, GPU) que expone Triton
    static_configs:
      - targets: ["triton:8002"]  # 8002 = --metrics-port de Triton
```

- Un archivo para el compose local: cada target es `servicio:puerto` en la red de Docker (`gateway:8080`, `triton:8002`). Desde el contenedor de Prometheus, `localhost` sería el propio Prometheus.
- Dos jobs, dos valores del label `job`: `gateway` y `triton`. El job de Triton no escribe `metrics_path` y usa el default, `/metrics`.
- `external_labels` no aparece en tus consultas: se pega a lo que sale de este Prometheus (alertas a Alertmanager, federación, remote write). En producción, `env` y `cluster` dicen de dónde viene cada alerta.
- En EKS este archivo no existe (lo dice su comentario): el Operator arma el scrape desde `k8s/monitoring.yaml`. Cada ServiceMonitor se vuelve un job (scrape pool) llamado `serviceMonitor/classifier/gateway/0`, pero el label `job` de las series es el nombre del Service: `gateway` y `triton`, los mismos que en local. El Operator agrega además `namespace`, `service`, `pod`, `container` y `endpoint` (el nombre del puerto).
- Los ServiceMonitors repiten `interval: 15s`: sin él usarían el intervalo de su Prometheus, que en el Prometheus Operator es 30 s por defecto. Con 15 s, las ventanas de 2 a 30 min de KEDA y de las alertas tienen de 8 a 120 muestras.

CÓMO RECORDARLA

Un **job es una ronda con su lista de cuartos** (targets): `relabel_configs` decide en qué cuartos entrar y con qué nombre anotarlos, `metric_relabel_configs` qué anotar, y `up` es la línea que dice «entré y alguien contestó».

TRAMPA

Prometheus en un contenedor con `targets: ["localhost:8080"]`: `localhost` es el propio contenedor de Prometheus, y el target queda DOWN con `connection refused`. Usa el nombre del servicio en compose (`gateway:8080`) o `host.docker.internal` para tu Mac. Y escribir `http://gateway:8080` en `targets`: la config ni carga (`"http://gateway:8080" is not a valid hostname`), y si el job tiene `relabel_configs`, carga pero ese target nunca se crea. El esquema va en `scheme` y la ruta en `metrics_path`.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · C01 · Archivos de referencia · M5 · docker compose para desarrollo local · Docker y kubectl · docker compose · Hoja · Docker · redes

5 PromQL básico: selectores, rangos, rate y agregaciones

¿Cómo le preguntas a Prometheus «cuántos requests por segundo», «qué fracción falla» o «qué pod va peor», sin que la respuesta te engañe?

SINTAXIS CON VARIABLES

métrica`{source="`*valor*`", status=~"`*regex*`"}`

Un **selector** elige **series** (cada serie es la historia de un número en el tiempo, con sus labels): el nombre de la métrica y, entre llaves, **matchers** (filtros por label, las etiquetas clave-valor de cada serie). `=` igual, `!=` distinto, `=~` cumple la regex (expresión regular), `!~` no la cumple. La regex tiene que calzar con el valor completo: `status=~"5.."` toma 500 y 503, no 1500. El resultado es un **instant vector** (vector instantáneo): el último valor de cada serie.

métrica`{…}[`*ventana*`]`

Con `[5m]` pides todas las **muestras** (cada lectura guardada: una hora y un valor) de los últimos 5 minutos de cada serie: un **range vector** (vector de rango). No se grafica directo: es lo que reciben `rate()` y las funciones `*_over_time`. La ventana deja afuera el borde viejo e incluye el de ahora.

`rate(`*counter*`[`*ventana*`])`
`increase(`*counter*`[`*ventana*`])`
`irate(`*counter*`[`*ventana*`])`

Para **counters** (contadores que solo suben, como el odómetro de un auto). `rate` da la velocidad: cuánto sube por segundo, en promedio, dentro de la ventana. `increase` da cuánto subió en toda la ventana. `irate` mira solo las dos últimas muestras. Los tres corrigen los **reinicios** (el proceso arranca de nuevo y el contador vuelve a 0). Prometheus solo exige 2 muestras dentro de la ventana; la regla práctica es una ventana de al menos 4 scrapes (4 lecturas de Prometheus): 1m con los 15 s de C01. Es la regla que usa Grafana en su ventana automática, `$__rate_interval`.

`sum by (`*label*`) (rate(`*counter*`[`*ventana*`]))`
`topk(`*k*`, sum by (`*label*`) (rate(`*counter*`[`*ventana*`])))`

Agregar junta muchas series en una. `sum by (source)` suma y deja solo el label `source`; `without (pod)` quita los labels que nombras y deja el resto. `topk(3, …)` se queda con las 3 series más altas, con sus labels. Siempre `rate` primero y `sum` después: si sumas los counters crudos, el reinicio de un pod hace bajar la suma y `rate` lo toma como un reinicio del total: sale un pico falso.

`sum(rate(`*errores*`[`*ventana*`])) /`
`sum(rate(`*total*`[`*ventana*`]))`

Una **proporción** (ratio). Al dividir, Prometheus empareja las series que tienen exactamente los mismos labels; las que no tienen pareja desaparecen, sin error. Por eso se agrega igual de los dos lados. `on(…)` o `ignoring(…)` eligen con qué labels emparejar, y `group_left` deja que varias series de la izquierda usen la misma de la derecha.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
métrica	nombre de la métrica	`classifier_requests_total`
valor	valor exacto de un label (los valores son texto)	`"mobile"`
regex	expresión regular que tiene que calzar con todo el valor	`"5.." (de 500 a 599)`
ventana	largo del rango: número y unidad (ms, s, m, h, d, w, y)	`5m en las alertas de C01; 2m en la query de KEDA (el que escala Triton)`
counter	un contador: suele terminar en _total, _count, _sum o _bucket (varios de Triton no llevan sufijo)	`classifier_requests_total, nv_inference_request_success`
label	label que queda después de agregar	`source, model_version, le`
k	cuántas series dejar	`3`
errores, *total*	selectores del numerador y del denominador	`classifier_requests_total{status=~"5.."} y classifier_requests_total{status!~"4.."}`
expresión, *umbral*	lo que comparas y el número contra el que comparas	`… > 0.01 (1 %)`
duración	cuánto hacia atrás	`1d, 1w`
x_1, x_n	primer y último valor del counter dentro de la ventana	`figura: 1,320 y 480`
t_1, t_n	segundo de la primera y de la última muestra	`figura: 45 y 150`
Σ reinicios	lo que llevaba el contador justo antes de cada reinicio	`figura: 1,680`
w	la ventana, en segundos	`120 con [2m]`
$\approx$	Prometheus extrapola hasta los bordes de la ventana; si las muestras son regulares y la cubren, da exactamente Δx / (t_n − t_1)	`figura: 8.0`

expresión **>** *umbral*
expresión **> bool** *umbral*
rate(*counter* **[** *ventana* **] offset** *duración* **)**

Una comparación **filtra**: quedan solo las series que la cumplen; en una regla de alerta, cada serie que queda es una alerta. Con `bool` devuelve 1 o 0 para todas, sin filtrar. `offset 1w` corre la consulta una semana hacia atrás (para comparar con el mismo día de la semana pasada) y va pegado al selector, dentro del `rate(…)`.

FÓRMULAS

$$\Delta x = x_n - x_1 + \sum_{\text{reinicios}} x_{\text{antes del reinicio}}$$

$$\text{rate}(x[w]) \approx \frac{\Delta x}{t_n - t_1} \qquad \text{increase}(x[w]) = \text{rate}(x[w]) \times w$$

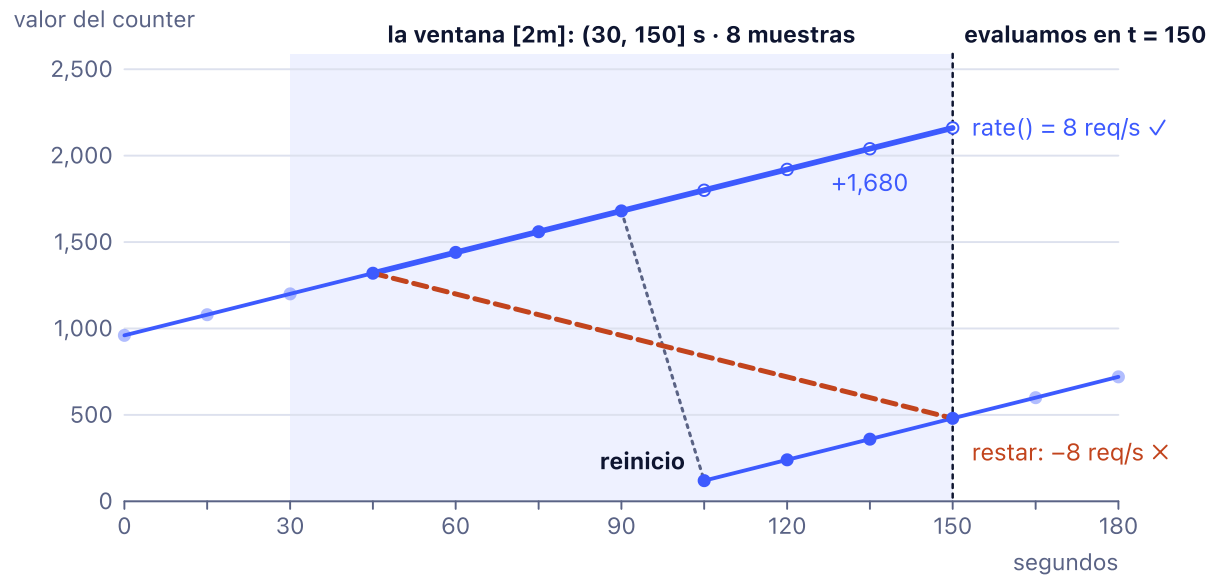
$$\text{irate}(x[w]) = \frac{x_n - x_{n-1}}{t_n - t_{n-1}}$$

$$w \geq 4 \times 15\text{s} = 1\text{min (regla práctica)} \qquad \text{figura: } \frac{480 - 1320 + 1680}{150\text{s} - 45\text{s}} = \frac{840}{105\text{s}} = 8 \text{ req/s}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Selector y matchers	Elige series por nombre y labels: = igual, != distinto, =~ cumple la regex, !~ no la cumple. La regex se ancla a todo el valor.	`classifier_requests_total{status!~"4.."}`
Label vacío	Un matcher que acepta "" también elige las series que no tienen ese label. Un selector sin nombre necesita al menos un matcher que no acepte "".	`classifier_requests_total{model_version=""}: las que no lo tienen`
Instant vector	Un valor por serie: el último que hay hasta 5 min atrás (el lookback). Una serie sin muestras en esos 5 min no aparece.	`nv_gpu_utilization`
Range vector	Todas las muestras de la ventana, por serie. No se grafica: va dentro de una función.	`classifier_requests_total[5m]: unas 20 muestras a 15 s`
rate()	Velocidad media de un counter en la ventana, por segundo. Corrige reinicios y estira la cuenta hasta los bordes de la ventana (extrapola). La de las alertas y los paneles.	`rate(classifier_requests_total[5m]) → req/s`
increase()	Cuánto subió en la ventana: rate × segundos de la ventana. Puede dar decimales aunque cuentes de a uno.	`increase(classifier_requests_total[1h])`
irate()	Velocidad entre las dos últimas muestras. Nerviosa: para graficar counters muy rápidos, no para alertas.	`irate(nv_inference_count[1m])`
***_over_time()**	Resumen de un gauge (un valor que sube y baja) dentro de la ventana: promedio, máximo, mínimo.	`max_over_time(nv_gpu_utilization[10m])`
sum · avg · max · min · count	Agregan muchas series en una (o en una por grupo). Promediar proporciones o percentiles no sirve: suma numeradores y denominadores.	`sum by (source) (rate(classifier_requests_total[5m]))`
by (…) · without (…)	by deja solo esos labels; without quita esos y deja el resto. Se puede escribir antes o después del paréntesis.	`sum by (le) (…) en un histograma`
topk(k, …) · bottomk	Las k series más altas o más bajas, con sus labels intactos.	`topk(3, sum by (source) (rate(classifier_requests_total[5m])))`
+ − * / (aritmética)	Con un número, operan sobre cada serie. Entre dos vectores, emparejan series con exactamente los mismos labels y lo que no tiene pareja se cae; on (…) e ignoring (…) eligen los labels y group_left permite muchos a uno. El resultado pierde el nombre de la métrica.	`… / 1000: de µs a ms · errores / total: una proporción`

PIEZA	QUÉ HACE	EJEMPLO
`> < == != >= <=`	Filtran: queda lo que cumple. Con bool devuelven 0 o 1 para todo.	`… > 0.01: la condición de una alerta`
`and · or · unless`	Operan con conjuntos de series: and deja lo de la izquierda que tiene pareja a la derecha; unless, lo de la izquierda que no la tiene; or deja todo lo de la izquierda y suma lo de la derecha que no tiene pareja.	`… > 0.01 and sum(rate(…[5m])) > 1 (C01)`
`offset`	Corre una consulta hacia el pasado. Va pegado al selector.	`rate(classifier_requests_total[5m] offset 1w)`
`clamp_min(v, m)`	Ningún valor queda por debajo de m: evita dividir por 0.	`clamp_min(…, 1e-9) en la query de KEDA`
`absent(v)`	Devuelve 1 cuando v no tiene ninguna serie, y nada cuando existe: avisa si una métrica desaparece.	`absent(up{job="gateway"})`



Restar el primero al último: no ve el reinicio

`(480 − 1320) / (150 − 45)`

−8 req/s

rate(): suma lo que el counter llevaba antes del reinicio

`(480 − 1320 + 1680) / (150 − 45) = 840 / 105`

8 req/s

increase(): rate × los segundos de la ventana

`8 req/s × 120 s`

960 requests

irate() en t = 105: solo las dos últimas muestras

`120 / 15: bajó, así que toma el último valor solo`

8 req/s

classifier_requests_total de un pod, una muestra cada 15 s — lo que ve rate(): el viejo + el nuevo — la resta que miente

la ventana: abierta a la izquierda

Un pod atiende 8 req/s y Prometheus lo lee cada 15 s. Justo después de la lectura de t = 90 s el pod se reinicia y su counter vuelve a 0. Restar el primer valor de la ventana al último da −8 req/s; rate() suma lo que el contador llevaba antes del reinicio y da 8. Cifras calculadas con la regla de Prometheus, la misma que el lab de la sección 2 hace a mano.

EJEMPLO REAL · LA QUERY DE KEDA: ESPERA MEDIA EN LA COLA DE TRITON, EN MS

YAML `reference/c01/k8s/keda-triton.yaml (líneas 53–59)`

```yaml
    # Espera media en cola en ms (misma fórmula que la alerta TritonQueueTimeHigh).
    # clamp_min evita 0/0 = NaN sin tráfico; si todo falla (0 éxitos) el valor se dispara y escala.
    query: |
      sum(rate(nv_inference_queue_duration_us{model="classifier"}[2m]))
        /
      clamp_min(sum(rate(nv_inference_request_success{model="classifier"}[2m])), 1e-9)
        / 1000
```

- Se lee de adentro hacia afuera. `nv_inference_queue_duration_us{model="classifier"}` es un counter: los µs acumulados en la cola desde que arrancó Triton. Triton no le pone `_total`, pero sube como un odómetro (la hoja de Triton separa sus counters de sus gauges). Por el nombre, Prometheus 3 muestra un aviso de nivel info al hacerle `rate` (`metric might not be a counter…`): aquí no importa.
- `rate(…[2m])`: cuántos µs de cola se acumulan por segundo en los últimos 2 minutos (8 muestras a 15 s). `sum(…)` junta los pods de Triton en un solo número. Primero rate, después sum.
- Dividir dos rates da un promedio: µs de cola por segundo ÷ requests exitosos por segundo = µs de espera por request, y `/ 1000` lo pasa a ms. Es la cuenta de la alerta `TritonQueueTimeHigh`, que usa `[5m]` y avisa sobre 20 ms; KEDA usa 2 minutos para reaccionar antes, con objetivo de 10 ms.
- `clamp_min(…, 1e-9)`: sin tráfico el denominador es 0 y 0/0 da NaN (not a number). Con el piso, 0 / 1e-9 = 0: sin cola, no escala. Y si todo falla (0 éxitos pero tiempo en cola), el valor se dispara y KEDA escala.
- Si la query vuelve vacía (Prometheus no lee Triton), no hay número: con `ignoreNullValues: "false"` KEDA lo marca como error en vez de tomarlo como 0 (hoja de Kubernetes, autoscaling). En C01, la señal de tráfico del gateway es la misma cuenta sobre un counter: `rate(classifier_requests_total[5m])`, por caller (quién llama: el label `source`).

CÓMO RECORDARLA

selector → [ventana] → rate → sum by → comparación, de adentro hacia afuera. Un counter es el odómetro del auto: el número suelto no dice nada, su velocidad (`rate`) sí. Un gauge es el indicador de gasolina: se lee directo, o con `avg_over_time`. Ventana de al menos 4 scrapes (la regla práctica de Grafana; Prometheus solo exige 2 muestras); `rate` antes de `sum`.

TRAMPA

escribir `status="5.."` con `=`. Busca el texto literal «5..»: no hay error, la consulta vuelve vacía y una alerta así nunca dispara. Para una regex va `=~`. Otra que tampoco avisa: dividir dos lados con labels distintos, como `sum by (source) (…) / sum(…)`. Sin pareja, el resultado es vacío; usa `/ ignoring(source) group_left sum(…)` o agrega igual en los dos lados.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · C01 · Monitoreo, alertas y reentrenamiento · M6 · KEDA: escalar por eventos y hasta cero · M12 · Métricas de sistema: golden signals con detalles de ML · Hoja · Kubernetes · autoscaling

6 Histogramas y percentiles: buckets, le y histogram_quantile

¿Cómo sacas el p99 de todos los pods juntos, qué tan exacto sale y por qué nunca se promedian los p99?

SINTAXIS CON VARIABLES

```
nombre_bucket{le="borde"}   # observaciones ≤ borde, acumuladas desde que arrancó el proceso
nombre_bucket{le="+Inf"}    # todas
nombre_count                # cuántas (igual al bucket +Inf)
nombre_sum                  # la suma de los valores observados
```

Un **histograma** (histogram) no guarda cada latencia: cuenta cuántas cayeron bajo cada **borde** (bucket). El label `le` significa «menor o igual». Los buckets son **acumulados**: `le="0.2"` incluye todo lo de `le="0.1"`. Cada línea es un counter.

```
histogram_quantile(φ, sum by (le) (rate(nombre_bucket[ventana])))
```

El percentil de todo el servicio, en tres pasos: `rate` (solo lo de la ventana, no desde que arrancó cada pod), `sum by (le)` (junta los pods y conserva `le`) y recién ahí el cuantil. Con φ = 0.99 sale el **p99**: el tiempo bajo el que queda el 99 % de los requests.

```
histogram_quantile(φ, sum by (le, label) (rate(nombre_bucket[ventana])))
```

Un p99 por grupo: agrega el label al `by`, sin quitar `le`. En C01, la latencia se parte por `source`; el score top-1 (otro histograma) se parte por `model_version` durante un canary, cuando una versión nueva recibe una parte del tráfico.

```
sum(rate(nombre_sum[ventana])) / sum(rate(nombre_count[ventana]))
```

La **media**: suma de los valores ÷ cantidad. Sirve, pero esconde la cola (los requests más lentos); el p99 la muestra.

```
sum(rate(nombre_bucket{le="borde"}[ventana])) / sum(rate(nombre_count[ventana]))
```

La **fracción** de observaciones ≤ borde, sin interpolar: exacta si el borde es uno de tus buckets. Así cuenta C01 las predicciones con score ≤ 0.3, y así se mide un SLO de latencia (la meta de servicio: qué fracción queda bajo 0.2 s).

```
histogram_quantile(φ, sum(rate(nombre[ventana])))   # native histogram: sin _bucket ni le
```

Los **native histograms** (buckets exponenciales automáticos, todo en una sola serie) no tienen `_bucket` ni `le`. Son estables desde Prometheus 3.8, pero Prometheus solo los lee si lo activas (`scrape_native_histograms: true`) y el cliente tiene que emitirlos: lo hacen las librerías oficiales de Go, Java y Rust. C01 (Python) usa histogramas clásicos.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre base del histograma	`classifier_request_latency_seconds`
borde	un borde de bucket, en la unidad de la métrica	`"0.2" (200 ms)`
ϕ	el cuantil, entre 0 y 1	`0.99 (p99), 0.5 (mediana)`
ventana	ventana del rate	`5m`
label	label que quieres conservar además de le	`source`
$\hat{q}$	cuantil estimado por histogram_quantile	`lab: 275 ms`
r, N	posición buscada y total de observaciones (el bucket +Inf)	`lab: 99 de 100`
ℓ, u	bordes inferior y superior del bucket donde cae r	`0.2 y 0.3 s`
c_ℓ, c_u	conteos acumulados en esos dos bordes	`lab: 96 y 100`
q_{real}	el percentil exacto, que Prometheus no conoce: siempre dentro del bucket	`lab: 260 ms`
$\bar{x}$	la media: crecimiento de _sum ÷ crecimiento de _count en la ventana	`con los datos del lab: 3.16 s / 50 = 63 ms en pod-b`
b, v	buckets finitos y combinaciones de labels; el + 4 son +Inf, _sum, _count y _created	`11 y 4 (los valores de source)`

FÓRMULAS

$$\hat{q} = \ell + (u - \ell)\frac{r - c_\ell}{c_u - c_\ell} \qquad r = \varphi \times N$$

$$\text{lab: } r = 0.99 \times 100 = 99 \Rightarrow \hat{q} = 0.2 + 0.1 \times \frac{99 - 96}{100 - 96} = 0.275 \text{ s}$$

$$\ell < q_{\text{real}} \leq u \qquad (\text{lab: } 0.2 < 0.26 \leq 0.3)$$

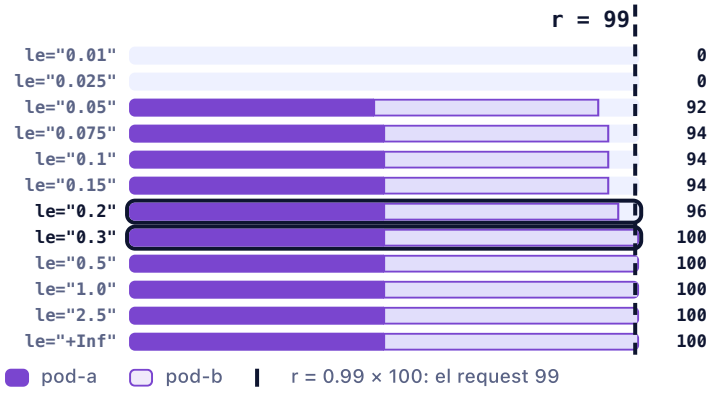
$$\bar{x} = \frac{\Delta_\text{sum}}{\Delta_\text{count}}$$

$$\text{series} = (b + 4) \times v = (11 + 4) \times 4 = 60 \text{ por pod}$$

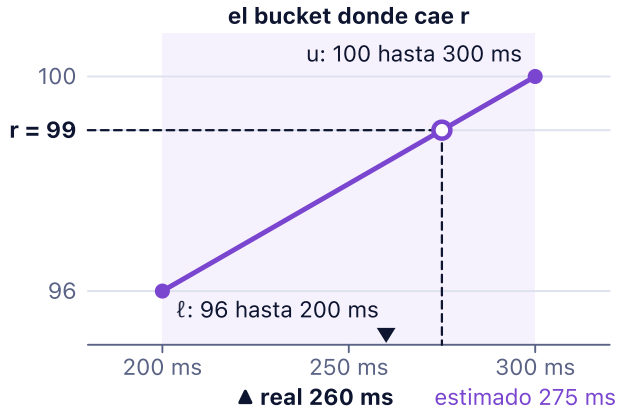
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`_bucket{le="…"}`	Un counter por borde: cuántas observaciones fueron ≤ ese borde. Acumulado.	`classifier_request_latency_seconds _bucket{le="0.2"}`
`le="+Inf"`	El último bucket: cuenta todo, igual que _count. Sin él, histogram_quantile da NaN.	el cliente lo agrega solo
`_sum · _count`	La suma de los valores y la cantidad de observaciones: su cociente es la media.	`rate(_sum) / rate(_count)`
`_created`	La hora en que nació cada serie; prometheus_client la agrega por cada combinación de labels.	se apaga con `PROMETHEUS_DISABLE_CREATED_SERIES=True`
`buckets=(…)`	Los bordes los decides en el código: densos cerca del SLO. Cambiarlos después rompe la comparación con la historia.	C01: 0.15, 0.2 y 0.3 s alrededor de 200 ms
Buckets por defecto	Sin buckets=, prometheus_client usa .005, .01, .025, .05, .075, .1, .25, .5, .75, 1, 2.5, 5, 7.5 y 10 s: de 0.1 salta a 0.25.	un p99 cerca de 200 ms queda en un bucket de 150 ms de ancho
`histogram_quantile(φ, v)`	Estima el cuantil φ con una línea recta dentro del bucket donde cae: supone que las observaciones están repartidas parejo dentro del bucket.	0.99 → p99
`sum by (le)`	Junta los buckets de todos los pods conservando le. Agregues como agregues (by o without), le tiene que sobrevivir.	obligatorio antes de histogram_quantile
Cae en +Inf	Si el cuantil cae en el último bucket, devuelve el borde finito más alto: no sabe cuánto más.	C01 nunca dirá más de 2.5 s
Sin tráfico	Con 0 observaciones en la ventana sale NaN (not a number): la comparación da falso y la alerta calla.	rate = 0 en todos los buckets
Summary	Calcula cuantiles en el cliente: no se pueden sumar entre pods. El de Python ni los calcula: solo _sum y _count.	avg(x{quantile="0.99"}) no significa nada
Native histogram	Buckets exponenciales automáticos en una sola serie: más precisos y baratos. Estables desde Prometheus 3.8, pero hay que activar su lectura.	scrape_native_histograms: true
Heatmap de Grafana	Dibuja los buckets en el tiempo: ves toda la distribución, no un solo percentil.	sección 9

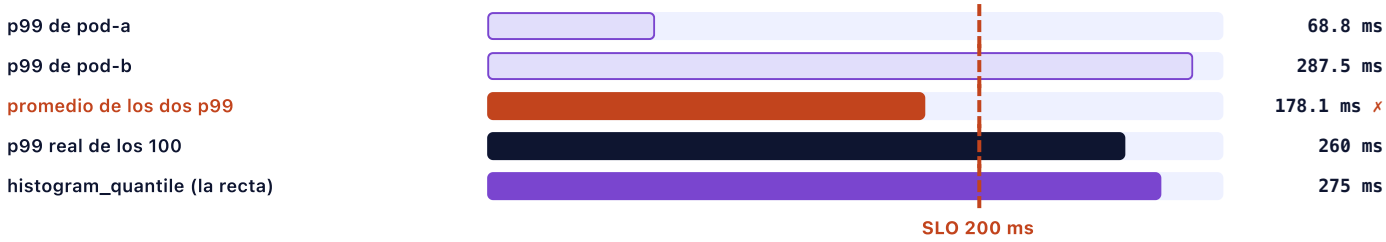
LOS BUCKETS ACUMULADOS DE LOS DOS PODS, SUMADOS POR LE (100 REQUESTS)



HISTOGRAM_QUANTILE: UNA RECTA DENTRO DEL BUCKET



¿CUMPLE EL SLO DE 200 MS? A ESCALA, DE 0 A 300 MS



El promedio dice 178.1 ms («cumple»); el p99 de los 100 requests es 260 ms. Sumar los buckets y sacar el cuantil una vez no miente.

100 requests de dos pods (los del lab): las barras son los buckets acumulados después de sum by (le). r = 0.99 × 100 = 99 cae entre le="0.2" (96) y le="0.3" (100); la línea recta da 275 ms. El p99 real es 260 ms: siempre dentro de ese bucket. Promediar el p99 de cada pod (68.8 y 287.5 ms) daría 178 ms, bajo el SLO de 200 ms.

EJEMPLO REAL · LA ALERTA DE P99 DE C01: BUCKETS SUMADOS ENTRE PODS, DESPUÉS EL CUANTIL

YAML reference/c01/monitoring/alerts.yml (líneas 26–39)

```yaml
      # SLO de latencia: p99 server-side ≤ 200 ms. Histograma agregado entre pods ANTES del quantile
      # (sum by le): promediar p99 de cada pod da un número matemáticamente sin sentido.
      - alert: ClassifierHighP99Latency
        expr: |
          histogram_quantile(0.99,
            sum by (le) (rate(classifier_request_latency_seconds_bucket[5m]))
          ) > 0.2
        for: 10m  # 10 min sostenido: ignora picos de un GC o de un deploy
        labels:
          severity: page
        annotations:
          summary: "p99 del clasificador sobre 200 ms"
          description: "p99 = {{ $value | humanizeDuration }} (SLO 200 ms). Revisa classifier_preprocess_seconds vs
classifier_inference_seconds para saber si es CPU o GPU."
          runbook_url: "https://runbooks.example.com/classifier/high-p99-latency"
```

- Los bordes salen de `main.py`: `buckets=(0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.5, 1.0, 2.5)`. Hay bordes en 0.15, 0.2 y 0.3 s: si el p99 anda cerca del SLO (200 ms), cae en un bucket de 50 o 100 ms de ancho y la estimación es fina.
- `rate(…_bucket[5m])`: cada bucket es un counter, y rate deja solo lo de los últimos 5 minutos. Sin rate, el p99 sería el de todo lo que pasó desde que arrancó cada pod (en el lab, 497.8 ms contra 49.8 ms de los últimos 5 minutos).
- `sum by (le)` suma el mismo bucket de todos los pods (y de todos los valores de `source`) y conserva `le`. El comentario lo dice: promediar el p99 de cada pod «da un número matemáticamente sin sentido». En el lab, dos pods con p99 de 68.8 y 287.5 ms promedian 178 ms («cumplimos»), pero sumando buckets el p99 es 275 ms (el real, 260 ms).
- `> 0.2` está en segundos: las métricas de Prometheus van en unidades base. Si el p99 pasa de 2.5 s, histogram_quantile devuelve 2.5 (el último borde finito): la alerta igual dispara, pero el número no dice cuánto más. `humanizeDuration` lo escribe legible: 0.2134 → 213.4ms.
- Sin tráfico en la ventana, el p99 es NaN y la alerta no dispara: una caída del tráfico se detecta con otra señal (por ejemplo, la tasa de requests contra la de la semana pasada, con `offset 1w`).

CÓMO RECORDARLA

rate, sum by (le), histogram_quantile, en ese orden y sin saltarte ninguno. Un histograma es como medir la estatura de un grupo con marcas en la puerta: «¿cuántos miden hasta 1.50 m, hasta 1.60, hasta 1.70?». Para el p99 buscas la marca donde ya pasaste el 99 % y estimas con una regla recta entre esa marca y la anterior: el resultado siempre queda entre las dos.

TRAMPA

agregar sin `le`. `histogram_quantile(0.99, sum(rate(x_bucket[5m])))` junta todos los bordes en una sola serie sin `le`, histogram_quantile la descarta y el resultado sale vacío. En la UI de Prometheus 3 ves un aviso (`bucket label "le" is missing or has a malformed value`), pero una regla no muestra avisos: la alerta calla para siempre. La otra ya la viste en la hoja del gateway: promediar los p99 de cada pod (en el lab, 178 ms «cumple» y el real es 260 ms).

En el curso: M12 · Métricas de sistema: golden signals con detalles de ML · M3 · p50, p95, p99: por qué el promedio miente · C01 · Monitoreo, alertas y reentrenamiento · M12 · Prometheus, Grafana y alertas por SLO · Hoja · El gateway · observabilidad

7 Recording rules y alerting rules: expr, for, labels y annotations

¿Cómo conviertes una consulta en una alerta que avisa a tiempo y sin falsas alarmas, y en una serie precalculada que se consulta barato?

SINTAXIS CON VARIABLES

```
groups:
  - name: grupo
    interval: intervalo
    rules:
      - record: nivel:métrica:operaciones
        expr: consulta
```

Una **regla de grabación** (recording rule) corre la consulta cada `interval` y guarda el resultado como una serie nueva con ese nombre: las alertas y los paneles leen esa serie, que sale barata. Sin `interval` usa el `evaluation_interval` global: 1m si nadie lo fija, 15 s en el prometheus.yml local de C01 y 30 s en EKS (el Kubernetes de AWS donde corre C01), porque es el default del Prometheus Operator, el programa que instala y configura Prometheus en el cluster. Las reglas de un grupo corren en orden, una tras otra.

```
      - alert: NombreDeAlerta
        expr: consulta > umbral
        for: espera
        keep_firing_for: retención
        labels:
          severity: severidad
        annotations:
          summary: "resumen"
          description: "detalle: {{ $value }}"
          runbook_url: "url"
```

Una **regla de alerta** (alerting rule) crea una alerta por cada serie que devuelve la expr. `for`: cuánto tiene que sostenerse antes de disparar; mientras, la alerta está **pending** (pendiente). Sin `for`, dispara en la primera evaluación en que la expr devuelve algo. `keep_firing_for`: cuánto sigue **firing** (disparada) después de que la condición se apaga, para que no parpadee; sin él, se resuelve en la primera evaluación que ya no cumple. `labels` deciden a quién le llega en Alertmanager (el que reparte los avisos, sección 8); `annotations` son el texto para la persona.

```
promtool check rules archivo.yml
promtool test rules pruebas.yml
```

`check` valida el YAML, el PromQL y las plantillas; sale con código 0 si todo está bien. `test` corre **pruebas unitarias**: series inventadas y las alertas que esperas a cierta hora. promtool viene en la imagen `prom/prometheus`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
grupo	nombre del grupo, único en el archivo	`classifier-recording`
intervalo	cada cuánto corre el grupo	`1m`
nivel : métrica : operaciones	los labels que quedan, la métrica de origen y lo que se le hizo (la última operación primero)	`classifier:low_confidence_share:rate30m`
consulta	la expresión PromQL	`histogram_quantile(0.99, sum by (le) (rate(…[5m])))`
NombreDeAlerta	en CamelCase; nombra el síntoma	`ClassifierHighP99Latency`
umbral	en la unidad base de la métrica	`0.2 (segundos)`
espera	el for	`10m`
retención	el keep_firing_for	`5m (C01 no lo usa)`
severidad	el valor con el que rutea tu Alertmanager. En C01: page despierta a alguien, warning se mira en horario laboral, ticket se investiga esa semana	`page, warning o ticket`
estado	estado de la alerta en la serie ALERTS	`pending o firing`
resumen, detalle, url	qué pasa, el valor y por dónde empezar, y los pasos	`summary, description y runbook_url de C01`
archivo, pruebas	el archivo de reglas y el de pruebas	`alerts.yml, prueba.yml`
nombre, namespace, release	nombre, namespace y label release del PrometheusRule	`classifier, classifier, kube-prometheus-stack`
t_{cruce}	cuándo la expr pasa el umbral; depende también de la ventana del rate	`unos minutos después de que empieza el problema`
Δt_{eval}	cada cuánto se evalúa el grupo	`15 s en el compose de C01; 30 s en EKS (default del Operator)`
group_wait	espera de Alertmanager antes de la primera notificación de un grupo (sección 8)	`30 s por defecto`

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: nombre
  namespace: namespace
  labels:
    release: release
spec:
  groups:
    - name: grupo
      rules: […]
```

En Kubernetes con el Prometheus Operator (el controlador que configura Prometheus a partir de objetos del cluster), el mismo YAML va bajo `spec` de un **PrometheusRule**. La label `release` tiene que coincidir con el `ruleSelector` (el filtro de labels con que tu Prometheus elige sus reglas); en kube-prometheus-stack es el nombre con que instalaste el chart de Helm. Si no coincide, el Operator no lo carga y nadie avisa.

```
ALERTS{alertname="NombreDeAlerta",
alertstate="estado"}
```

Prometheus escribe una serie `ALERTS` (vale 1) por cada alerta pending o firing: la puedes consultar y graficar. La página `/alerts` (puerto 9090) muestra cada regla como inactive, pending o firing.

VARIABLE	QUÉ ES	EJEMPLO
43.2 min	presupuesto de errores: lo que puede fallar en un mes sin romper el SLO (la meta) de 99.9 % a 30 días	`(1 − 0.999) × 43,200 min`

FÓRMULAS

$$t_{\text{page}} \approx t_{\text{cruce}} + \text{for} + \Delta t_{\text{eval}} + \text{group_wait} = t_{\text{cruce}} + 10\text{min} + 30\text{s} + 30\text{s (EKS)}$$

$$\text{apagón total: presupuesto gastado antes del page} \approx \frac{\text{for}}{43.2\text{ min}} = \frac{5\text{ min}}{43.2\text{ min}} \approx 11.6\%$$

$$\frac{0}{0} = \text{NaN} \Rightarrow \text{avg_over_time}(\ \ldots\ [7d]) = \text{NaN durante 7 días}$$

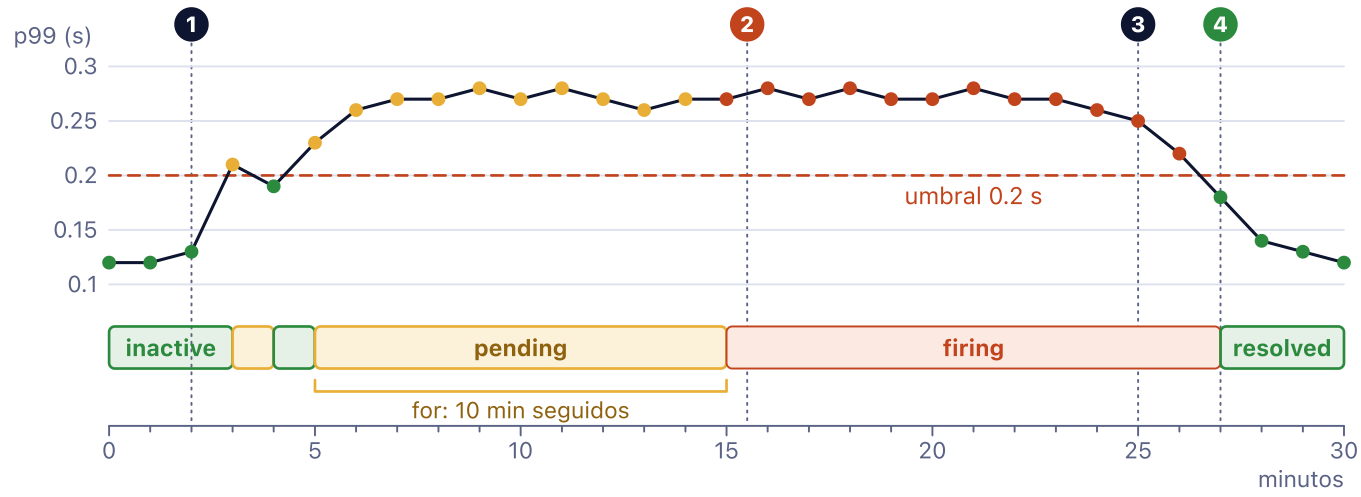
$$7\text{d} \div 1\text{min} = 10{,}080 \text{ muestras de la serie grabada por evaluación}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`groups[].name`	Nombre del grupo, único en el archivo. Sus reglas corren en orden, con la misma hora de evaluación.	`classifier-recording y classifier-alerts`
`interval`	Cada cuánto corre el grupo. Si falta, el evaluation_interval global: 1m si nadie lo fija, 30 s con el Prometheus Operator.	`1m en classifier-recording`
`record`	Nombre de la serie que se graba: nivel:métrica:operaciones (con rate, la métrica pierde su _total). Se graba rate y no increase: la documentación lo pide para que todo quede por segundo.	`classifier:low_confidence_share:rate30m`
`alert`	Nombre de la alerta: llega a Alertmanager como el label alertname.	`ClassifierHighP99Latency`
`expr`	La consulta. En una alerta, cada serie del resultado es una alerta: por eso termina en una comparación que filtra.	`histogram_quantile(…) > 0.2`
`for`	Cuánto tiene que sostenerse; mientras, pending. Si una evaluación sale bajo el umbral, vuelve a inactive y el conteo empieza de cero. Sin for, dispara en la primera evaluación.	`for: 10m`
`for + ventana del rate`	La ventana y el for se suman: la ventana estira cada pico. Con [5m], un pico de 2 min al 10 % de 5xx deja la proporción sobre 1 % unos 6 min, así que una alerta con for: 5m dispara igual.	`ClassifierHigh5xxRate: [5m] y for: 5m`

PIEZA	QUÉ HACE	EJEMPLO
keep_firing_for	Cuánto sigue firing después de que la condición se apaga: evita resolver y redisparar en el borde.	`keep_firing_for: 5m`
labels	Labels que se agregan a la alerta; Alertmanager rutea con ellos.	`severity: page`
annotations	Texto para quien recibe la alerta: summary corto, description con valores y runbook_url, el link al runbook (la guía de pasos para resolverla).	`las tres en cada alerta de C01`
$value · $labels	En las plantillas: el valor de la expr y los labels de la serie que disparó. Después de un sum sin by, $labels ya no trae pod ni instance.	`{{ $labels.gpu_uuid }} en GPUSaturated`
humanize*	Funciones que formatean en las plantillas: humanizeDuration (0.2134 → 213.4ms), humanizePercentage (0.0123 → 1.23%).	`description de ClassifierHighP99Latency`
Guarda de tráfico	Una condición extra con and que exige tráfico mínimo: con 20 requests de madrugada, un error ya es el 5 %.	`and sum(rate(classifier_requests_total[5m])) > 1`
ALERTS	Serie que Prometheus crea por cada alerta pending o firing.	`ALERTS{alertstate="firing"}`
promtool	check rules valida; test rules prueba con series inventadas antes del deploy.	`promtool check rules alerts.yml`
PrometheusRule	El mismo YAML como objeto de Kubernetes; el Operator lo carga si su label calza con el ruleSelector.	`release: kube-prometheus-stack`
Recarga	Prometheus vuelve a leer las reglas cuando recibe la señal SIGHUP (un aviso del sistema operativo) o un POST a /-/reload, si arrancó con --web.enable-lifecycle. Si un archivo tiene un error, se queda con las reglas anteriores.	`kill -HUP al proceso de Prometheus`

`ClassifierHighP99Latency` · `for: 10m` · se evalúa cada 15 s en el compose y 30 s en EKS



- INACTIVE **0–3 min**
- PENDING **3–4 min** primera evaluación sobre 0.2 s
- INACTIVE **4–5 min** una evaluación bajo 0.2 s: el for empieza de cero
- PENDING **5–15 min** 10 min seguidos sobre 0.2 s
- FIRING **15–27 min** Prometheus la manda a Alertmanager y la reenvía cada 1 min
- RESOLVED **27–30 min** resolved: Alertmanager avisa la resolución si el receiver tiene send_resolved

ok · up · inactive · pending · firing

1. **2 min** deploy
2. **15.5 min** page: firing + group_wait de 30 s
3. **25 min** rollback
4. **27 min** resolved (con keep_firing_for: 5m seguiría firing hasta el minuto 32)

ClassifierHighP99Latency (for: 10m) durante un deploy que sube el p99. Un punto por minuto, valores ilustrativos; Prometheus evalúa cada 15 s en el compose de C01 y cada 30 s en EKS. Una sola evaluación bajo 0.2 s reinicia el for; el page sale 10 min después de la última entrada a pending, más los 30 s de group_wait de Alertmanager.

EJEMPLO REAL · UNA REGLA DE GRABACIÓN Y LA ALERTA QUE LA LEE: LA SEÑAL DE CALIDAD DE C01

YAML reference/c01/monitoring/alerts.yml (líneas 8–21 y 62–76; # … marca las omitidas)

```yaml
groups:
  # ------------------------------------------------------------------ reglas de grabación
  - name: classifier-recording
    interval: 1m  # precalcula cada minuto: la alerta de confianza compara contra 7 días de esta serie
    rules:
      # Fracción de predicciones con score top-1 < 0.3 (el bucket le="0.3" del histograma).
      # Se graba porque avg_over_time(...[7d]) sobre una expresión cruda sería carísimo en cada evaluación.
      # `> 0` en el denominador: sin tráfico en 30 min no se graba nada. Sin él, 0/0 = NaN quedaría grabado
      # y avg_over_time(...[7d]) daría NaN durante 7 días: la alerta de abajo nunca dispararía.
      - record: classifier:low_confidence_share:rate30m
        expr: |
          sum(rate(classifier_top1_score_bucket{le="0.3"}[30m]))
            /
          (sum(rate(classifier_top1_score_count[30m])) > 0)
  # …
      # Señal de calidad SIN labels: si se duplica la fracción de predicciones de baja confianza,
      # algo cambió en la entrada (nueva cámara, nuevo tipo de producto, bug de preprocesamiento).
      - alert: ClassifierLowConfidenceSpike
        expr: |
          classifier:low_confidence_share:rate30m
            > 2 * avg_over_time(classifier:low_confidence_share:rate30m[7d])
          and sum(rate(classifier_top1_score_count[30m])) > 1
        # Necesita 7 días de historia de la regla de grabación (retención de Prometheus ≥ 7 d)
        for: 30m
        labels:
          severity: ticket
        annotations:
          summary: "Se duplicó la fracción de predicciones con score top-1 < 0.3"
          description: "Fracción actual = {{ $value | humanizePercentage }} (> 2× su promedio de 7 días). Compara por source (mobile/web) en los logs y revisa deploys recientes del gateway."
          runbook_url: "https://runbooks.example.com/classifier/low-confidence-spike"
```

- El grupo `classifier-recording` corre cada minuto (`interval: 1m`) y graba `classifier:low_confidence_share:rate30m` : de las predicciones de los últimos 30 minutos, la fracción con score top-1 ≤ 0.3 (el bucket `le="0.3"` ÷ el total). El nombre dice el nivel (`classifier`), la métrica y la operación con su ventana.
- `(… > 0)` en el denominador: sin tráfico, la comparación descarta el 0 y no se graba nada. Sin ese filtro, 0/0 = NaN quedaría grabado y el `avg_over_time(…[7d])` de la alerta daría NaN durante 7 días: nunca dispararía.
- La alerta compara la fracción actual con el doble de su promedio de 7 días. Sobre la serie grabada son 10,080 muestras (una por minuto); sobre la expresión cruda habría que recalcular dos rates de 30 minutos en cada punto de esos 7 días, en cada evaluación. Necesita 7 días de historia: la retención por defecto de Prometheus es 15 d, y la de kube-prometheus-stack, 10 d.
- `and sum(rate(classifier_top1_score_count[30m])) > 1` es la guarda de tráfico: de madrugada, con pocas predicciones, la fracción salta sola. `for: 30m` y `severity: ticket` : la calidad del modelo se revisa en horario laboral; nadie se despierta por esto.
- Las annotations llevan el valor (`humanizePercentage` : 0.065 → 6.5%), qué mirar primero y el runbook. En el compose, Prometheus lee este archivo por `rule_files` de prometheus.yml; en EKS, el README lo envuelve en un PrometheusRule con `release: kube-prometheus-stack` . C01 lo validó con un parser de PromQL, no con promtool: el lab lo hace.

CÓMO RECORDARLA

expr dice qué, for dice cuánto tiempo, labels dicen a quién, annotations dicen qué hacer. Es la indicación médica junto a la cama: «si la temperatura pasa de 38 °C durante 10 minutos, llama a quien esté de guardia». La ronda de enfermería (el scrape) anota cada 15 s; una sola medición alta no basta (for), y la gravedad (severity) decide si es una llamada urgente (page) o una nota para la mañana (ticket).

TRAMPA

una alerta sobre una serie que no existe nunca dispara. Si Prometheus deja de leer el gateway, o alguien renombra una métrica en main.py, `histogram_quantile(…) > 0.2` devuelve vacío: ni pending ni error, solo silencio. Por eso main.py dice que sus nombres son un contrato con alerts.yml, y por eso existen `absent()` y las alertas sobre `up` . Prueba las reglas con `promtool test rules` , que falla si la alerta esperada no sale.

En el curso: C01 · Monitoreo, alertas y reentrenamiento | C01 · Archivos de referencia | M12 · Prometheus, Grafana y alertas por SLO | M12 · Dashboards, runbooks y on-call

8 Alertmanager: ruteo, agrupación, inhibición, silencios y receivers

Cuando una regla pasa a firing, ¿cómo llega a la persona correcta, una sola vez, y no a las 3 a. m. por algo que puede esperar?

SINTAXIS CON VARIABLES

```
route:
  receiver: receiver_por_defecto
  group_by: [ label , …]
  group_wait: espera
  group_interval: intervalo
  repeat_interval: repetición
  routes:
    - matchers: [' label =" valor "']
      receiver: receiver
      continue: false
```

El **árbol de ruteo**: cada alerta entra por la raíz, que acepta todas, y baja al primer hijo cuyos **matchers** (filtros por label: `=`, `!=`, `=~`, `!~`) cumple. Con `continue: true` sigue probando los hermanos. Si no calza ningún hijo, la maneja el nodo donde está. Los hijos heredan lo que no definen. Por defecto: group_wait 30s, group_interval 5m, repeat_interval 4h.

```
receivers:
  - name: receiver
    pagerduty_configs:
      - routing_key_file: archivo
  - name: otro_receiver
    slack_configs:
      - api_url_file: archivo
        channel: ' canal '
        send_resolved: true
```

Un **receiver** es un destino con nombre: PagerDuty u Opsgenie (servicios que hacen sonar el teléfono de quien está de guardia, el on-call: eso es un **page**), Slack, Teams, email, un webhook (un POST a una URL tuya) y otros. Los secretos van en archivos (`*_file`), no en el YAML. `send_resolved` avisa también cuando la alerta se resuelve: true por defecto en PagerDuty y webhook, false en Slack y email.

```
inhibit_rules:
  - source_matchers: [' label =" valor "']
    target_matchers: [' label =~" regex "']
    equal: [ label , …]
```

Inhibición (inhibition): mientras haya una alerta activa que cumple `source_matchers`, se callan las que cumplen `target_matchers` y tienen los mismos valores en los labels de `equal`. Un label que falta cuenta como vacío: si falta en las dos, son iguales.

```
amtool silence add label = valor --duration= duración
--comment=" motivo "
--alertmanager.url=http://localhost:9093
amtool silence query
--alertmanager.url=http://localhost:9093
amtool silence expire id
--alertmanager.url=http://localhost:9093
```

Un **silencio** (silence) calla por un tiempo las alertas que calzan con unos matchers: un mantenimiento, un falso positivo ya conocido. La alerta sigue firing en Prometheus; solo no se notifica. amtool exige un comentario y la duración por defecto es 1h. También se crean en la UI, en el puerto 9093.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
receiver_por_defecto	receiver de la raíz: lo que no calce con ningún hijo	`equipo-slack`
label, *valor*, *regex*	label de la alerta y el valor o la regex que pide el matcher	`severity, "page", "TritonQueueTimeHigh|GPUSaturated"`
espera	el group_wait	`30s`
intervalo	el group_interval	`5m`
repetición	el repeat_interval	`4h (1h en la ruta de page)`
receiver, *otro_receiver*	nombres de destinos definidos en receivers	`oncall-pagerduty, equipo-slack`
archivo	archivo con el secreto, montado desde un Secret	`/etc/alertmanager/secrets/pagerduty-routing-key`
canal	canal de Slack	`#classifier`
duración, *motivo*	cuánto dura el silencio y por qué	`2h, "mantenimiento del nodo GPU"`
id	el id que devuelve silence add	`un UUID`
alertmanager.yml	el archivo de configuración	`alertmanager.yml`
host	dónde escucha Alertmanager	`alertmanager (compose) o el Service del Operator`

```
amtool check-config alertmanager.yml
amtool config routes test
--config.file=alertmanager.yml --tree label=valor
…
```

Valida el archivo y te dice a qué receiver iría una alerta con esos labels, sin mandar nada: la prueba unitaria del ruteo.

```
alerting:
  alertmanagers:
    - static_configs:
        - targets: ['host:9093']
```

Esto va en el prometheus.yml y conecta Prometheus con Alertmanager. El prometheus.yml local de C01 no lo trae: en el compose las alertas se ven en `:9090/alerts`, pero no le llegan a nadie. En kube-prometheus-stack (el chart de Helm que instala todo esto en Kubernetes) lo configura el Operator.

FÓRMULAS

$$t_{\text{primera notificación}} = t_{\text{llega la primera alerta del grupo}} + \text{group_wait (30s)}$$

$$\Delta t_{\text{entre avisos de un grupo}} \geq \text{group_interval} = 5\text{min} \qquad \Delta t_{\text{recordatorio}} = \text{repeat_interval} = 4\text{h}$$

$$N_{\text{notificaciones}} = N_{\text{grupos}} \neq N_{\text{alertas}}: \quad \text{4 GPUs saturadas con group_by: [alertname, env]} \Rightarrow 1$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
route (raíz)	Entrada de todas las alertas: no lleva matchers y tiene el receiver de reserva.	`receiver: equipo-slack`
routes · matchers	Hijos que eligen alertas por labels. Gana el primero que calza.	`severity="page" → oncall-pagerduty`
continue	Si es true, la alerta sigue por los hermanos y puede llegar a varios receivers. Default false.	`page al on-call y copia a Slack`
group_by	Labels que definen un grupo: las alertas con los mismos valores salen en una sola notificación. ['...'] agrupa por todos los labels (cada alerta va sola).	`[alertname, env]`
group_wait	Espera antes de la primera notificación de un grupo nuevo, para juntar las que llegan casi a la vez. Default 30s.	`30s`
group_interval	Espera antes de avisar cambios en un grupo ya notificado (alertas nuevas o resueltas). Default 5m.	`5m`
repeat_interval	Cada cuánto se repite el aviso si nada cambió. Default 4h; kube-prometheus-stack trae 12h.	`1h en la ruta de page`
receivers	Destinos con nombre: pagerduty_configs, slack_configs, email_configs, webhook_configs, opsgenie_configs, msteams_configs y más.	`oncall-pagerduty`
send_resolved	Avisa también cuando la alerta se resuelve. Default true en PagerDuty y webhook, false en Slack y email.	`send_resolved: true en Slack`
inhibit_rules	Mientras una alerta fuente esté activa, calla a las objetivo que coinciden en los labels de equal.	`TritonNotReady calla a TritonQueueTimeHigh`
Silencio	Calla por un tiempo las alertas que calzan con unos matchers, con autor y comentario. Se crea en la UI (:9093) o con amtool.	`amtool silence add alertname=GPUSaturated --duration=2h --comment="nodo en mantenimiento"`
time_intervals	Horarios con nombre. Una ruta con active_time_intervals solo notifica en ese horario; con mute_time_intervals, calla en él.	`warnings de lunes a viernes, 9 a 18`
Deduplicación y HA	Alertas idénticas (mismos labels) se notifican una vez. En alta disponibilidad, Prometheus manda a todos los Alertmanagers, sin balanceador, y ellos se coordinan (puerto 9094).	`2 réplicas de Prometheus, 1 solo page`

PIEZA	QUÉ HACE	EJEMPLO
external_labels	Labels que Prometheus pega a cada alerta que manda: dicen de qué entorno o cluster viene.	`env: local en el prometheus.yml de C01`
Watchdog	Alerta que siempre está firing (expr: vector(1)) para probar que toda la cadena funciona. kube-prometheus-stack la trae y la manda al receiver 'null'; si la mandas a un servicio externo, ese servicio avisa cuando deja de llegar.	`severity: none`
Webhook	Un POST con JSON (version "4": receiver, status, alerts…) a tu URL: el receiver más fácil de probar.	`webhook_configs: - url: http://host.docker.internal:5001/page`

route (la raíz): entran todas las alertas

si ningún hijo calza → **equipo-slack**

`GROUP_BY [alertname, env]` `GROUP_WAIT 30s` `GROUP_INTERVAL 5m` `REPEAT_INTERVAL 4h`

↓ baja al primer hijo que calza

1 `severity="page"`

→ **oncall-pagerduty**

`repeat_interval: 1h`

`ClassifierHighP99Latency`

`ClassifierHigh5xxRate`

`TritonNotReady`

2 `severity="warning"`

→ **equipo-slack**

`active_time_intervals: horario-laboral`

`TritonQueueTimeHigh` `GPUSaturated`

3 `severity="ticket"`

→ **ml-tickets**

`ClassifierLowConfidenceSpike`

`INHIBICIÓN:` mientras esté activa una alerta con `alertname="TritonNotReady"`, se callan las que cumplen `alertname=~"TritonQueueTimeHigh|GPUSaturated"` y tienen el mismo `env`.

UN INCIDENTE (15:10): SE CAE EL NODO DE UNA DE LAS 2 GPUS: QUEDA 1 POD DE TRITON CON TODO EL TRÁFICO DE LA HORA PICO

15:15 TritonNotReady **PAGE** firing (for: 5m)
✓ page a oncall-pagerduty a las 15:15:30 (group_wait 30 s)

15:20 TritonQueueTimeHigh **WARNING** firing (for: 10m)
⊘ inhibida: no se notifica

15:25 GPUSaturated **WARNING** firing (for: 15m)
⊘ inhibida: no se notifica

3 alertas en firing → **1 notificación**

page: despierta a alguien · warning: en horario laboral · ticket: esta semana · inhibida: no se notifica

El árbol de la configuración de ejemplo, y lo que pasa cuando se cae el nodo de una de las dos GPUs en hora pico (horas ilustrativas): disparan tres alertas, pero llega un solo page. La cola y la GPU quedan inhibidas por TritonNotReady.

EJEMPLO REAL · LAS SEVERIDADES DE C01 CONVERTIDAS EN RUTAS, UNA INHIBICIÓN Y UN HORARIO

YAML — ilustrativo: arma en YAML la columna «quién la recibe» de C01 monitoreo y la tabla de severidades de M12; C01 no trae alertmanager.yml

```yaml
# alertmanager.yml ILUSTRATIVO: la columna «quién la recibe» de C01, en YAML (C01 no trae este archivo)
route:
  receiver: equipo-slack  # lo que no calce con ningún hijo va al canal del equipo
  group_by: [alertname, env]  # una notificación por alerta y entorno, no una por pod o por GPU
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
    - matchers: ['severity="page"']  # p99, 5xx, Triton no listo: quien usa el servicio ya lo nota
      receiver: oncall-pagerduty
```

```yaml
      repeat_interval: 1h
    - matchers: ['severity="warning"']  # cola de Triton, GPU: anticipan un page
      receiver: equipo-slack
      active_time_intervals: [horario-laboral]
    - matchers: ['severity="ticket"']  # calidad del modelo: se analiza con datos, no a las 3 a. m.
      receiver: ml-tickets
inhibit_rules:
  - source_matchers: ['alertname="TritonNotReady"']  # con Triton caído, la cola y la GPU son consecuencia
    target_matchers: ['alertname=~"TritonQueueTimeHigh|GPUSaturated"']
    equal: [env]
time_intervals:
  - name: horario-laboral
    time_intervals:
      - weekdays: ['monday:friday']
        times:
          - start_time: '09:00'
            end_time: '18:00'
        location: America/Mexico_City
receivers:
  - name: oncall-pagerduty
    pagerduty_configs:
      - routing_key_file: /etc/alertmanager/secrets/pagerduty-routing-key
  - name: equipo-slack
    slack_configs:
      - api_url_file: /etc/alertmanager/secrets/slack-webhook-url
        channel: '#classifier'
        send_resolved: true
  - name: ml-tickets
    webhook_configs:
      - url: http://ticket-bridge.monitoring.svc:8080/alertmanager
```

- Las tres rutas salen de C01: page al on-call del servicio (quien está de guardia), warning al canal del equipo y ticket al equipo de ML. La raíz no tiene matchers (acepta todas) y su receiver es el de reserva para lo que no calce.
- `group_by: [alertname, env]`: si se saturan las 4 GPUs a la vez, llega una notificación con las cuatro, no cuatro. `group_wait: 30s` espera a las hermanas antes del primer aviso; `group_interval: 5m` espacia los avisos de cambios; `repeat_interval` repite si nada cambió (4 h, 1 h en la ruta de page).
- La inhibición: mientras `TritonNotReady` esté activa, se callan `TritonQueueTimeHigh` y `GPUSaturated` del mismo `env`. Con un pod de Triton caído, la cola y la GPU son consecuencia, y el on-call ya tiene el page de la causa. `env` sale de `external_labels` del prometheus.yml (`env: local`). No pongas en `equal` un label que la expr pierde al agregar: TritonQueueTimeHigh hace `sum` sin `by` y no tiene `namespace`, así que con `equal: [namespace]` nunca se inhibiría.
- `active_time_intervals` deja que la ruta de warning avise solo de lunes a viernes, de 9 a 18 (hora de Ciudad de México). Los secretos van en archivos (`routing_key_file`, `api_url_file`), montados desde un Secret: nunca en el YAML que va a git.
- Para que algo llegue, Prometheus necesita el bloque `alerting.alertmanagers`. El Alertmanager de kube-prometheus-stack viene con `group_by: ['namespace']`, `repeat_interval: 12h`, todo al receiver `'null'` (que descarta) e inhibiciones por severidad: una critical calla a las warning e info con el mismo alertname y namespace, y una warning, a las info. Hay que agregarle tus receivers y tus rutas.

CÓMO RECORDARLA

Prometheus decide que algo está mal; Alertmanager decide a quién, cuándo y cuántas veces avisar. Es la central de enfermería de un hospital: junta los avisos de un mismo paciente en una sola llamada (agrupar), no llama por la presión si ya llamó por el paro (inhibir), no llama por la cama en mantenimiento (silenciar) y elige a quién llamar según la gravedad (rutear).

TRAMPA

creer que un silencio o una inhibición apaga la alerta. Prometheus la sigue evaluando y la ves firing en `:9090/alerts`; Alertmanager solo no la notifica. Un silencio largo «por si acaso» tapa la próxima falla real: dale una duración corta y un comentario. Y en Slack `send_resolved` es false por defecto: nadie se entera de que se resolvió.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Dashboards, runbooks y on-call · C01 · Monitoreo, alertas y reentrenamiento · Herramientas · Monitoreo: sistema y modelo

9 Grafana: data sources, dashboards, paneles, variables y dashboards como código

¿Cómo conviertes las consultas en un tablero que quien está de guardia entiende en 10 segundos, y cómo evitas que ese tablero se pierda o se rompa?

SINTAXIS CON VARIABLES

```
apiVersion: 1
datasources:
  - name: nombre
    type: prometheus
    uid: uid
    access: proxy
    url: http://host:9090
    isDefault: true
    jsonData:
      timeInterval: scrape_interval
```

Un **data source** (fuente de datos) le dice a Grafana dónde está Prometheus. Grafana no guarda métricas: le manda PromQL a Prometheus y dibuja la respuesta. Este archivo va en `/etc/grafana/provisioning/datasources` y se carga al arrancar. `timeInterval` es el «Scrape interval» del data source: ponle el de tu Prometheus (15s en C01) para que `$__rate_interval` salga bien.

```
rate(counter[$__rate_interval])
histogram_quantile(0.99, sum by (le)
(rate(nombre_bucket[$__rate_interval])))
```

`$__rate_interval` es una variable de Grafana: una ventana que se ajusta al zoom y que nunca baja de 4 scrapes, así `rate` siempre tiene muestras. En las reglas de Prometheus (alertas y grabaciones) no existe: ahí va una ventana fija, como `[5m]`.

```
Variable: variable · Query type: Label values ·
Label: label · Metric: métrica
sum by (model_version)
(rate(classifier_requests_total{model_version=~"$mod
el_version"}[$__rate_interval]))
```

Una **variable** es un menú arriba del dashboard (model_version, source) que se reemplaza en todas las consultas. Con selección múltiple o «All», Grafana une los valores con `|`: usa `=~`, no `=`.

```
Visualización: Time
series | Stat | Gauge | Heatmap | Table
Unit: s | ms | percentunit | reqps
Thresholds: Base verde · valor rojo
```

Cada **panel** es una o más consultas, una visualización, una **unidad** (con Unit `s`, Grafana muestra 0.2134 como 213 ms) y **umbrales** (thresholds: colores a partir de un valor). Los umbrales pintan; no le avisan a nadie. En Legend, `{{source}}` pone el valor de ese label como nombre de cada línea.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre del data source o del provider	`Prometheus`
uid	id fijo del data source: los dashboards lo referencian	`prom-lab`
host	dónde escucha Prometheus, visto desde Grafana	`prometheus (compose), prometheus-operated.monitoring.svc (EKS)`
scrape_interval	el intervalo de scrape real de tu Prometheus	`15s`
counter, *nombre*_bucket	un counter y los buckets de un histograma	`classifier_requests_total, classifier_request_latency_seconds_bucket`
variable, *label*, *métrica*	nombre de la variable, label cuyos valores ofrece y métrica de donde los saca	`model_version, model_version, classifier_requests_total`
valor	valor donde empieza el color del umbral	`0.2 (s) en el p99, 0.01 en la tasa de 5xx`
carpeta	carpeta de Grafana donde aparece el dashboard	`Clasificador`
ruta_de_los_json	carpeta del contenedor con los archivos JSON	`/var/lib/grafana/dashboards`
s	el Scrape interval configurado en el data source	`15 s`
$__interval	el paso entre puntos de un panel: crece con el rango de tiempo y baja con el ancho del panel	`más grande con 7 días que con 1 hora`
$__rate_interval	la ventana para rate en un panel	`nunca menos de 4s = 60 s`

```yaml
apiVersion: 1
providers:
  - name: nombre
    type: file
    folder: carpeta
    allowUiUpdates: false
    options:
      path: ruta_de_los_json
```

Dashboards como código: el JSON del dashboard (su modelo completo: paneles, consultas, variables) vive en git y Grafana lo carga de una carpeta. Con `allowUiUpdates: false` no se puede guardar un cambio desde la UI: el cambio entra por git. En kube-prometheus-stack (el chart de Helm que instala Prometheus, Alertmanager y Grafana en Kubernetes), un ConfigMap con la label `grafana_dashboard: "1"` hace lo mismo: Grafana lo encuentra y lo carga.

FÓRMULAS

$$\$__rate_interval = \max(\$__interval + s, 4s)$$

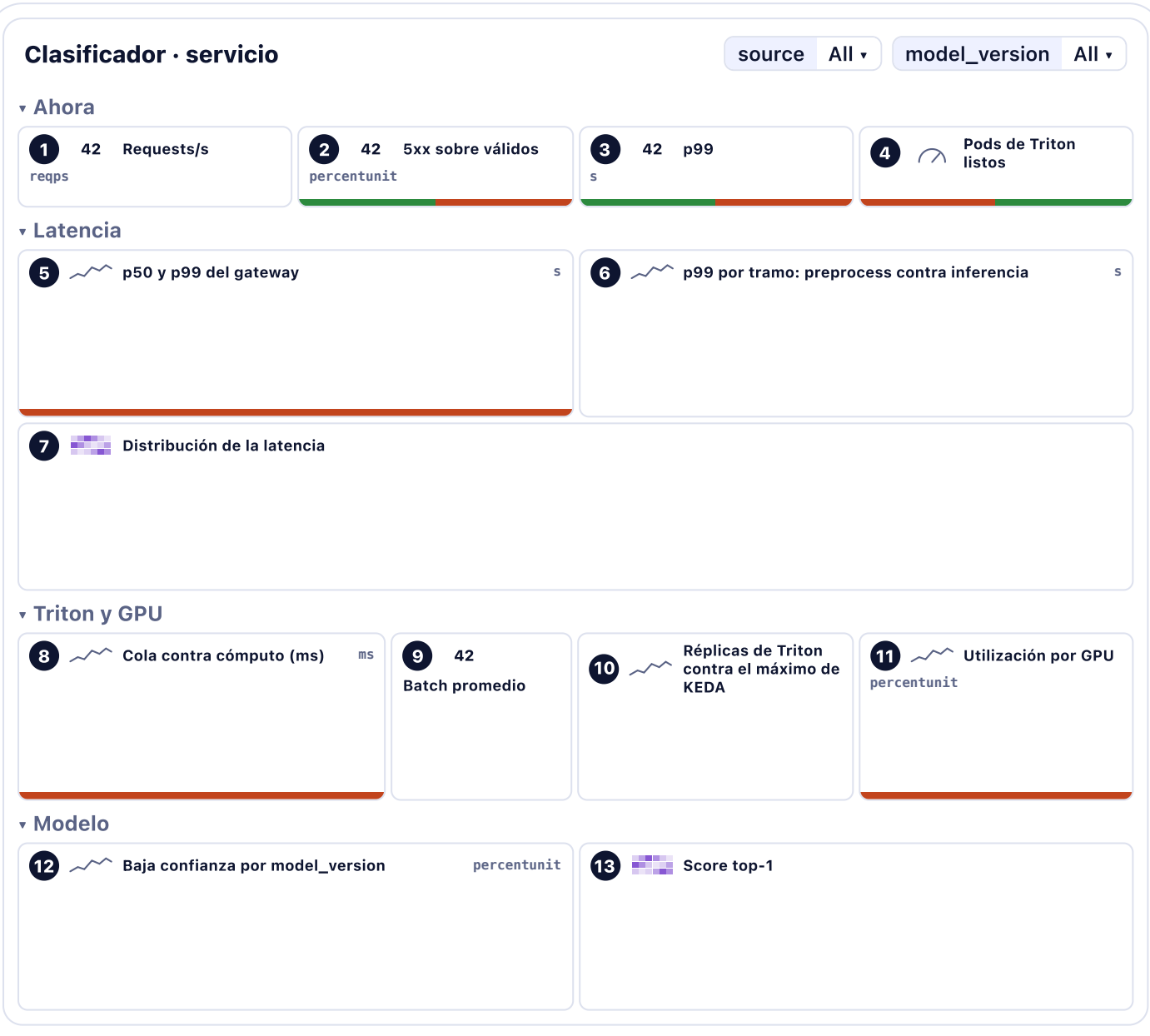
$s = 15s$: $\$__interval = 30s \Rightarrow 60s$ $\$__interval = 2min \Rightarrow 135s$

$$\$__interval \approx \frac{\text{rango de tiempo del dashboard}}{\text{puntos del panel}} \quad \text{(redondeado y nunca menor que } s\text{)}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Data source	La conexión a Prometheus: tipo y URL. Grafana consulta; no guarda métricas.	`http://prometheus-operated.monitoring.svc:9090`
Dashboard	Una página de paneles con un rango de tiempo y un refresco comunes. Por dentro es un JSON.	`Servicio, Triton y GPU, Modelo, Negocio (C01)`
Row	Una fila plegable que agrupa paneles.	`Ahora · Latencia · Triton y GPU · Modelo`
Time series	Líneas en el tiempo: tasas, latencias, cola. La visualización por defecto.	`p50 y p99 del gateway`
Stat	Un número grande con color por umbral (por defecto, el último valor).	`tasa de 5xx de ahora`
Gauge · Bar gauge	El valor contra un mínimo y un máximo que fijas.	`pods de Triton listos, de 0 a 4`
Heatmap	La distribución en el tiempo: cada columna son los buckets de un histograma. La consulta va con Format: Heatmap.	`buckets de classifier_request_latency_seconds`
Table	Una fila por serie, con sus valores y labels.	`top 5 de callers (quién llama: el label source) por req/s`
Variable	Un menú del dashboard que se reemplaza en las consultas; tipo Query → Label values.	`$model_version: estable contra canary (la versión nueva a prueba)`
$__rate_interval	La ventana para rate que se ajusta al zoom y nunca baja de 4 scrapes.	`rate(classifier_requests_total[$__rate_interval])`
Unit	Cómo se muestra el número: s, ms, percentunit (0.0–1.0 como %), reqps.	`percentunit en la tasa de 5xx`
Thresholds	Colores a partir de un valor; por defecto, base verde y 80 rojo. En Time series se pueden dibujar como líneas. No mandan alertas.	`línea roja en 0.2 s`
Legend	El nombre de cada serie; {{label}} usa el valor de ese label.	`{{gpu_uuid}} en la utilización por GPU`
Annotations	Marcas verticales que salen de una consulta: deploys, reinicios, alertas.	`changes(process_start_time_seconds{job="gateway"}[5m]) > 0`
Provisioning	Data sources y dashboards desde archivos al arrancar: viven en git, no en un clic.	`/etc/grafana/provisioning`

PIEZA	QUÉ HACE	EJEMPLO
Explore	Pantalla para consultas sueltas, sin armar un panel: el lugar para probar una query.	`Builder o Code`
Alertas de Grafana	Grafana también puede evaluar alertas. Elige un solo lugar para las reglas (en C01, Prometheus), versionado en git.	`alerts.yml de C01`



A escala: 24 columnas de ancho; cada fila de la grilla, una unidad de alto.

1 **Requests/s** Stat · unit `reqps`

PROMQL

```
sum(rate(classifier_requests_total{source=~"$source"}
[$__rate_interval]))
```

2 **5xx sobre válidos** Stat · unit `percentunit`

base: verde · 0.01: rojo

PROMQL

```
sum(rate(classifier_requests_total{status=~"5..",
source=~"$source"}[$__rate_interval])) /
sum(rate(classifier_requests_total{status!~"4..",
source=~"$source"}[$__rate_interval]))
```

3 **p99** Stat · unit `s` base: verde · 0.2: rojo

PROMQL

```
histogram_quantile(0.99, sum by (le)
(rate(classifier_request_latency_seconds_bucket{source=~"$sour
ce"}[$__rate_interval])))
```

4 **Pods de Triton listos** Gauge · de 0 a 4 base: rojo · 2: verde

PROMQL

```
kube_deployment_status_replicas_available{namespace="classifie
r", deployment="triton"}
```

5 p50 y p99 del gateway

Time series · unit `s`

0.2: rojo, como línea

PROMQL

```
histogram_quantile(0.5, sum by (le)
(rate(classifier_request_latency_seconds_bucket{source=~"$sour
ce"}[$__rate_interval])))
```

PROMQL

```
histogram_quantile(0.99, sum by (le)
(rate(classifier_request_latency_seconds_bucket{source=~"$sour
ce"}[$__rate_interval])))
```

6 p99 por tramo: preprocess contra inferencia

Time series · unit `s`

PROMQL

```
histogram_quantile(0.99, sum by (le)
(rate(classifier_preprocess_seconds_bucket[$__rate_interval]))
)
```

PROMQL

```
histogram_quantile(0.99, sum by (le)
(rate(classifier_inference_seconds_bucket[$__rate_interval])))
```

7 Distribución de la latencia

Heatmap · Format: Heatmap

PROMQL

```
sum by (le)
(rate(classifier_request_latency_seconds_bucket{source=~"$sour
ce"}[$__rate_interval]))
```

8 Cola contra cómputo (ms)

Time series · unit `ms`

20: rojo, como línea

PROMQL

```
sum(rate(nv_inference_queue_duration_us{model="classifier"}
[$__rate_interval])) /
sum(rate(nv_inference_request_success{model="classifier"}
[$__rate_interval])) / 1000
```

PROMQL

```
sum(rate(nv_inference_compute_infer_duration_us{model="classif
ier"}[$__rate_interval])) /
sum(rate(nv_inference_request_success{model="classifier"}
[$__rate_interval])) / 1000
```

9 Batch promedio

Stat

PROMQL

```
sum(rate(nv_inference_count{model="classifier"}
[$__rate_interval])) /
sum(rate(nv_inference_exec_count{model="classifier"}
[$__rate_interval]))
```

10 Réplicas de Triton contra el máximo de KEDA

Time series

PROMQL

```
kube_horizontalpodautoscaler_status_current_replicas{horizonta
lpodautoscaler="keda-hpa-triton"}
```

PROMQL

```
kube_horizontalpodautoscaler_spec_max_replicas{horizontalpodau
toscaler="keda-hpa-triton"}
```

11 Utilización por GPU

Time series · unit `percentunit` · Legend `{{gpu_uuid}}`

0.85: rojo, como línea

PROMQL `nv_gpu_utilization`

12 Baja confianza por model_version

Time series · unit `percentunit` · Legend `{{model_version}}`

PROMQL

```
sum by (model_version)
(rate(classifier_top1_score_bucket{le="0.3",
model_version=~"$model_version"}[$__rate_interval])) / sum by
(model_version)
(rate(classifier_top1_score_count{model_version=~"$model_versi
on"}[$__rate_interval]))
```

13 Score top-1

Heatmap · Format: Heatmap

PROMQL

```
sum by (le)
(rate(classifier_top1_score_bucket{model_version=~"$model_vers
ion"}[$__rate_interval]))
```

VARIABLES: LOS MENÚS DE ARRIBA

`$source` · Label values: source de classifier_requests_total · selección múltiple · con All

`$model_version` · Label values: model_version de classifier_requests_total · selección múltiple · con All

ANOTACIONES: MARCAS VERTICALES EN TODOS LOS PANELES

Reinicios del gateway

PROMQL

```
changes(process_start_time_seconds{job="gateway"}[5m]) > 0
```

Alertas firing

PROMQL

```
ALERTS{alertstate="firing"}
```

umbral verde umbral rojo (pinta; no avisa)

Un dashboard de servicio para el on-call de C01 (quien está de guardia), en la grilla de 24 columnas de Grafana (x, y, w y h en celdas): lo que importa ahora arriba, después la latencia por tramo, Triton y el modelo. Las consultas son reales sobre las métricas de C01; el layout es una propuesta, porque el repo no trae dashboards.

EJEMPLO REAL · LAS TRES CONSULTAS DE M12 QUE SE VUELVEN PANELES DE C01

PROMQL `js/content/m12-monitoreo.js, sección «Prometheus, Grafana y alertas por SLO»: p99.promql, error-ratio.promql y batch-promedio.promql, tal cual`

```
# p99 del servicio en ventanas de 5 min: primero sumar buckets entre pods, después el cuantil
histogram_quantile(0.99,
  sum by (le) (rate(classifier_request_latency_seconds_bucket[5m])))

# Fracción de 5xx sobre requests válidos (los 4xx no cuentan en ningún lado)
sum(rate(classifier_requests_total{status=~"5.."}[5m]))
  / sum(rate(classifier_requests_total{status!~"4.."}[5m]))

# Batch promedio en Triton: inferencias por ejecución (cerca de max_batch_size = sin margen)
sum(rate(nv_inference_count{model="classifier"}[5m]))
  / sum(rate(nv_inference_exec_count{model="classifier"}[5m]))
```

- C01 describe cuatro dashboards: **Servicio** (tasa, errores y latencia por tier, la capa: ALB, gateway y Triton, y por caller, con el desglose preprocess / cola / cómputo y marcadores de deploy), **Triton y GPU** (cola contra cómputo, batch promedio, utilización y memoria por GPU, réplicas contra el máximo de KEDA), **Modelo** (score top-1, baja confianza, top-20 de categorías, todo partido por `model_version`) y **Negocio** (diario). El repo no trae sus JSON: estas consultas son el punto de partida.
- p99 → un **Time series** del dashboard Servicio, con Unit `s` y el umbral 0.2 dibujado como línea. En un panel conviene `[$__rate_interval]` en vez de `[5m]`: se ajusta al zoom. Para el valor de ahora, un **Stat** con la misma consulta.
- Fracción de 5xx → Unit `percentunit` (0.012 se ve como 1.20%) y un umbral rojo en 0.01, el de la alerta. Partida por caller: `sum by (source)` arriba y abajo de la división.
- Batch promedio → dashboard Triton y GPU. Si se acerca a `max_batch_size`, el dynamic batching (Triton juntando requests en un solo batch) ya no tiene margen (hoja de Triton). Va junto a la cola (la query de KEDA de la sección 5) y a `nv_gpu_utilization` con Legend `{{gpu_uuid}}`: una línea por GPU.
- El top-20 de categorías no sale de Prometheus: el gateway no tiene un label de categoría (serían 500 valores, cada uno con sus propias series: demasiada cardinalidad). Sale de los logs de predicción, en otro data source; un mismo dashboard puede mezclar paneles de los dos.

CÓMO RECORDARLA

Grafana no mide nada: pregunta y dibuja. Es el tablero del auto: muestra lo que miden los sensores (Prometheus guarda, Grafana pinta). Cada panel es una consulta, una visualización, una unidad y unos umbrales; las variables cambian el filtro de todo el tablero a la vez, y el JSON del dashboard va a git como cualquier código.

TRAMPA

dejar `rate(x[1m])` fijo en un panel. Con un rango de 7 días, Grafana pide un punto cada varios minutos, pero cada punto mira solo el último minuto: lo que pasó entre punto y punto no se ve. `$__rate_interval` crece con el paso, pero depende del Scrape interval del data source: si dice 15s y tu Prometheus lee cada 60 s, la ventana sale corta y el panel queda vacío. Y un umbral rojo no avisa a nadie: las alertas viven en las reglas de Prometheus.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Dashboards, runbooks y on-call · C01 · Monitoreo, alertas y reentrenamiento · Herramientas · Monitoreo: sistema y modelo · Hoja · De PyTorch a Triton · batching

10 SLOs: golden signals, RED y USE, error budget y alertas por burn rate

¿Qué mides para saber si el servicio cumple, cuánto puede fallar en un mes y cuándo una alerta tiene que despertar a alguien?

SINTAXIS CON VARIABLES

```
- record: servicio:error_ratio:rate ventana
  expr: |
    sum(rate( contador {status=~"5.."}[ ventana ]))
      /
    sum(rate( contador {status!~"4.."}[ ventana ]))
```

El **SLI** (service level indicator: la fracción de eventos buenos sobre los válidos) se alerta al revés, por la fracción mala. Aquí, 5xx (errores del servidor) sobre los requests sin 4xx (errores del cliente, como una imagen inválida): los 4xx no cuentan ni arriba ni abajo. Va en una **recording rule** (una regla que corre la query en cada evaluación, cada 15 s en el compose del C01, y guarda el resultado como una serie nueva), una por ventana.

```
- record: servicio:slow_ratio:rate ventana
  expr: |
    1 - (
      sum(rate( histograma _bucket{le=" umbral "}
[ ventana ]))
        /
      sum(rate( histograma _count[ ventana ]))
    )
```

El SLI de latencia sale de un solo **bucket** (el cajón del histograma que cuenta los requests que tardaron hasta ese borde): lo que no cayó en `le="` *umbral* `"` tardó más. El umbral tiene que ser un borde exacto del histograma; el gateway del C01 tiene 0.2 y 0.3.

```
- alert: Alerta
  expr: |
    servicio:error_ratio:rate larga > ( factor *
presupuesto )
    and
    servicio:error_ratio:rate corta > ( factor *
presupuesto )
  labels:
    severity: severidad
```

Suena si el **error budget** (el presupuesto de errores: lo que el SLO deja fallar) se quema *factor* veces más rápido de lo permitido en la ventana larga **y** en la corta. Las reglas multi-ventana del SRE Workbook (el libro de Google sobre cómo operar servicios confiables) no llevan `for` (la espera antes de disparar): las dos ventanas ya filtran los picos, y el libro desaconseja sumar esperas.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
servicio	prefijo de las series grabadas: el nivel del nombre nivel:métrica:operaciones	`classifier`
contador	el counter de requests, con el label status	`classifier_requests_total`
ventana, *larga*, *corta*	ventanas de rate(); la larga confirma, la corta se apaga rápido	`5m, 30m, 1h, 6h · larga 1h con corta 5m`
histograma	nombre base del histograma de latencia	`classifier_request_latency_seconds`
umbral	borde del bucket que separa rápido de lento, en segundos	`0.2 (p99 ≤ 200 ms) · 0.3 (el SLI de requisitos)`
factor	burn rate que dispara la alerta	`14.4, 6 o 1`
presupuesto	1 − SLO, como fracción	`0.001 (errores) · 0.01 (latencia: 99 % bajo 200 ms)`
Alerta, *severidad*	nombre de la alerta y a quién le llega: page despierta a quien está de guardia; ticket se revisa en horario laboral	`ClassifierErrorBudgetFastBurn · page`
SLO	la meta del SLI en una ventana de tiempo	`99.9 % en 30 días`
burn rate	qué tan rápido gastas el presupuesto; 1 = se acaba justo al final del mes	`5 % de errores = 50×`
g	fracción del presupuesto del mes que aceptas gastar antes del aviso	`2 %, 5 % o 10 %`
P	el período del SLO	`720 h (30 días)`
L	la ventana larga de la alerta	`1 h, 6 h o 3 días`
e	la tasa de error real durante el incidente	`5 % (el deploy malo de M15)`
t_{agotar}	en cuánto se acaba el presupuesto a ese ritmo	`50 h a 14.4×`
t_{detectar}	cuánto tarda la ventana larga en cruzar el umbral	`≈ 17 min con 5 % de errores`

```
1 - (
  sum(increase( contador {status=~"5.."}[30d]))
    /
  sum(increase( contador {status!~"4.."}[30d]))
) / presupuesto
```

Cuánto queda del presupuesto del mes: 1 = intacto, 0 = gastado, negativo = SLO roto. Va arriba en el dashboard de quien está de guardia (el on-call). Necesita 30 días guardados: con la retención por defecto (15 días; 10 en kube-prometheus-stack) mide solo los que hay. Es cara: con scrape cada 15 s, cada serie aporta 172,800 muestras. Igual corre (tarda, pero corre), porque Prometheus las carga de a una serie; una subquery de 30 días (`rate(…[5m])[30d:15s]` : la query de adentro, recalculada cada 15 s) las junta todas a la vez y, con unas 300 series, pasa el tope de 50 millones (`--query.max-samples`): `query processing would load too many samples` . Lo barato: grabar la fracción con recording rules.

FÓRMULAS

$$\text{presupuesto} = 1 - \text{SLO} = 1 - 0.999 = 0.001 \Rightarrow 0.001 \times 300\text{ M} = 300{,}000 \text{ requests al mes}$$

$$\text{burn rate} = \frac{\text{tasa de error en la ventana}}{1 - \text{SLO}} \qquad t_{\text{agotar}} = \frac{720\text{ h}}{\text{burn rate}}$$

$$\text{factor} = \frac{g \times P}{L} = \frac{0.02 \times 720\text{ h}}{1\text{ h}} = 14.4 \qquad \text{umbral} = 14.4 \times 0.001 = 1.44\%$$

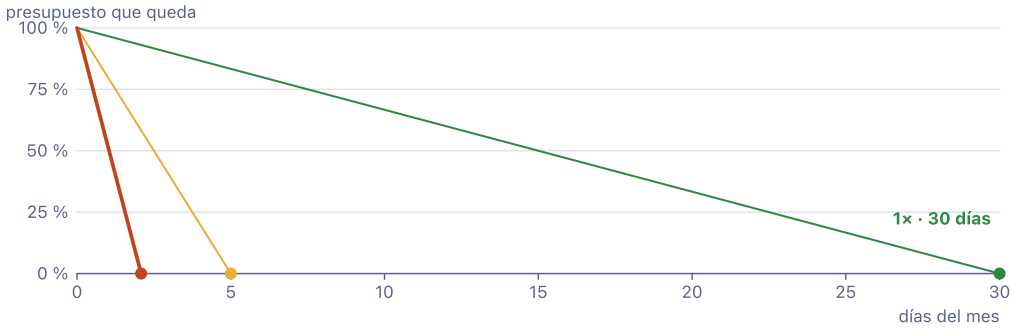
$$t_{\text{detectar}} = \frac{1 - \text{SLO}}{e} \times L \times \text{factor} = \frac{0.001}{0.05} \times 60\text{ min} \times 14.4 \approx 17\text{ min}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Golden signals	Las cuatro señales del libro de SRE de Google: latencia, tráfico, errores y saturación (qué tan lleno está el servicio).	un panel por señal y por tier
RED	Rate, Errors, Duration: requests por segundo, errores y latencia. La versión corta para servicios que responden requests.	gateway: req/s, 5xx, p99
USE	Utilization, Saturation, Errors: cuánto se usa, cuánto trabajo queda esperando y cuánto falla cada recurso (CPU, memoria, GPU, disco).	GPU: utilización y cola de Triton
SLI	La medida: eventos buenos sobre eventos válidos. Para alertar se usa la fracción mala.	requests sin 5xx y en ≤ 300 ms ÷ requests sin 4xx
SLO	La meta interna del SLI en una ventana. Sale de requisitos, no de lo que el sistema logra hoy.	99.9 % en 30 días · p99 ≤ 200 ms
SLA	Un contrato con créditos o multas si no se cumple. Más laxo que el SLO, para tener margen.	el C01 no tiene: sus callers son internos
Error budget	Lo que el SLO deja fallar: 1 – SLO. Se gasta en deploys y experimentos; si se acaba, se congelan los lanzamientos riesgosos.	0.1 % = 300,000 requests al mes
Burn rate	Tasa de error ÷ presupuesto. A 1× el presupuesto dura justo el mes; a 14.4×, 50 h.	5 % de errores = 50×
Ventana larga y corta	La larga confirma que el gasto es real; la corta (1/12 de la larga) hace que la alerta se apague minutos después del arreglo.	1 h y 5 min · 6 h y 30 min · 3 días y 6 h
14.4× · 6× · 1×	Los tres pares del SRE Workbook para 99.9 %: 2 % del presupuesto en 1 h (page), 5 % en 6 h (page) y 10 % en 3 días (ticket).	el C01 manda el de 6× a ticket: existe el selector manual
Recording rule por ventana	Una serie grabada por ventana: cada alerta compara dos números ya calculados en vez de recalcular horas de datos.	classifier:error_ratio:rate1h

PIEZA	QUÉ HACE	EJEMPLO
Mínimo de tráfico	Con pocos requests, un solo error es un porcentaje enorme. Exige un caudal mínimo antes de alertar.	`and sum(rate(classifier_requests_total[5m])) > 1`
Política de error budget	Un acuerdo previo con producto: qué se hace con más de 50 %, entre 10 y 50 % y con menos de 10 % de presupuesto.	`bajo 10 %: solo cambios de confiabilidad`
Sloth · Pyrra	Herramientas que generan todas estas reglas desde una definición corta del SLO.	`un YAML con el objetivo y el SLI`

SLO **99.9 %** en 30 días presupuesto **0.1 %** de 300,000,000 requests al mes: **300,000** pueden fallar



- **1×: dura justo los 30 días**
- **6×: se acaba en 5 días**
- **14.4×: se acaba en 50 h**

LAS ALERTAS DE DOS VENTANAS: LA LARGA CONFIRMA, LA CORTA (1/12) SE APAGA RÁPIDO

14.4× PAGE

suena si el error pasa **1.44 %** en **1h** y en **5m**

1h — 5m

gasta el **2 %** del presupuesto del mes antes de sonar

6× PAGE (TICKET EN EL C01)

suena si el error pasa **0.6 %** en **6h** y en **30m**

6h — 30m

gasta el **5 %** del presupuesto del mes antes de sonar

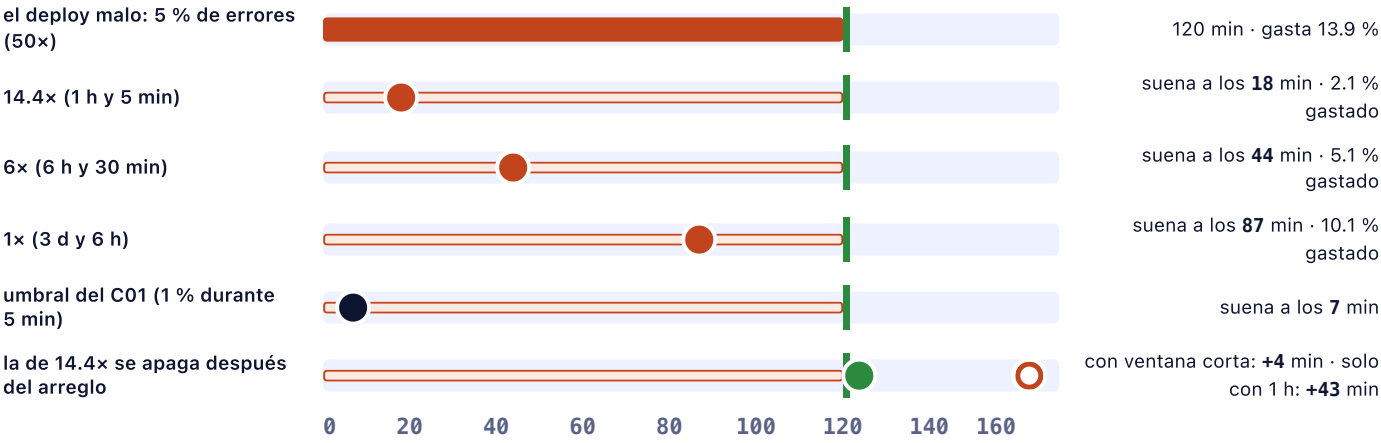
1× TICKET

suena si el error pasa **0.1 %** en **3d** y en **6h**

3d — 6h

gasta el **10 %** del presupuesto del mes antes de sonar

UN DEPLOY MALO, MINUTO A MINUTO (A ESCALA, DE 0 A 170 MIN)



El presupuesto del mes del C01 (300,000 requests fallidos) gastado a tres ritmos, y un deploy malo con 5 % de errores durante 2 h: la alerta de 14.4× suena a los 18 min y se apaga 4 min después del arreglo gracias a la ventana corta; solo con la de 1 h, se apagaría 43 min después del arreglo. Números del lab de esta sección.

EJEMPLO REAL · C01: LAS ALERTAS POR BURN RATE DEL CURSO PARA EL SLO DE 99.9 %

YAML — M12 · Prometheus, Grafana y alertas por SLO (slo-rules.yml del curso, extracto; la línea # … resume las tres reglas omitidas)

```yaml
# Parte de errores del SLO de c01: 99.9 % a 30 días (su SLI también exige ≤ 300 ms) → presupuesto 0.1 %
groups:
  - name: classifier-slo-recording
```

```yaml
    rules:
      # Una regla por ventana: la alerta compara ventanas ya calculadas (barato y legible)
      - record: classifier:error_ratio:rate5m
        expr: |
          sum(rate(classifier_requests_total{status=~"5.."}[5m]))
            / sum(rate(classifier_requests_total{status!~"4.."}[5m]))
      # … classifier:error_ratio:rate30m, :rate1h y :rate6h: la misma expr con [30m], [1h] y [6h]
  - name: classifier-slo-alerts
    rules:
      # 14.4× durante 1 h = 2 % del presupuesto mensual; la ventana de 5 min hace que se apague rápido
      - alert: ClassifierErrorBudgetFastBurn
        expr: |
          classifier:error_ratio:rate1h > (14.4 * 0.001)
          and classifier:error_ratio:rate5m > (14.4 * 0.001)
        labels:
          severity: page
        annotations:
          summary: "Quemando el presupuesto de errores 14.4× más rápido de lo permitido"
          description: "Error ratio 1 h = {{ $value | humanizePercentage }} (umbral 1.44 %)."
          runbook_url: "https://runbooks.example.com/classifier/error-budget-burn"
      # 6× durante 6 h = 5 % del presupuesto; el SRE Workbook lo manda a page, c01 a ticket (hay fallback)
      - alert: ClassifierErrorBudgetSlowBurn
        expr: |
          classifier:error_ratio:rate6h > (6 * 0.001)
          and classifier:error_ratio:rate30m > (6 * 0.001)
        labels:
          severity: ticket
        annotations:
          summary: "Quemando el presupuesto de errores 6× más rápido de lo permitido"
          runbook_url: "https://runbooks.example.com/classifier/error-budget-burn"
```

- Una regla de grabación por ventana (5m, 30m, 1h y 6h): cada alerta compara dos series ya calculadas. Sin ellas, cada evaluación recalcularía `rate()` sobre 6 h de datos crudos de todos los pods.
- `14.4 * 0.001` es 1.44 %: más de 1.44 % de 5xx en la última hora **y** en los últimos 5 min. Con el deploy malo de M15 (5 % de errores) suena a los 17–18 min, con ≈ 2 % del presupuesto gastado, y se apaga 4 min después del arreglo. Lo calculas en el lab.
- La de 6× va a `ticket` y no a page: el C01 no está en el camino crítico (quien publica puede elegir la categoría a mano). El SRE Workbook la manda a page.
- Es solo la parte de errores. El SLI de requisitos también exige ≤ 300 ms: la fracción lenta sale del bucket `le="0.3"` con la segunda plantilla. Sumar las dos fracciones puede contar dos veces un 5xx que además tardó más de 300 ms; como el gateway se da 250 ms por request (su deadline: el tiempo máximo que se permite), casi no pasa.
- Comparada con `ClassifierHigh5xxRate` de `alerts.yml` (más de 1 % durante 5 min): ese umbral fijo es un burn rate de 10×. Con 5 % de errores suena antes (a los 7 min, contra 18); en una caída total, no: la de 14.4× suena al primer minuto y el `for: 5m` retrasa la fija hasta el minuto 6. Y la fija nunca suena con un 0.8 % sostenido, que agota el mes en 90 h; la de 6× lo atrapa a las 4.5 h.
- Sin tráfico, la fracción es 0/0 = NaN y la comparación da falso: no dispara. Con muy poco tráfico, en cambio, un error mueve mucho la fracción; el extracto del curso no trae el mínimo de requests que sí usa `ClassifierHigh5xxRate`.

CÓMO RECORDARLA

14.4 en 1 h, 6 en 6 h, 1 en 3 días: el 2 %, el 5 % y el 10 % del presupuesto del mes. Cada ventana larga lleva una corta de 1/12 (5 min, 30 min, 6 h): la larga dice que el gasto es real; la corta, que sigue pasando.

TRAMPA

creer que una alerta de umbral fijo ya es el SLO. `ClassifierHigh5xxRate` (más de 1 % durante 5 min) es un burn rate de 10×: un 0.8 % sostenido nunca la dispara y se come el presupuesto del mes en menos de 4 días. Y con poco tráfico pasa lo contrario: un solo error ya es un porcentaje enorme, así que exige un mínimo de requests.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M15 · SLOs, error budgets y disponibilidad compuesta · C01 · Requisitos · C01 · Monitoreo, alertas y reentrenamiento · M12 · Métricas de sistema: golden signals con detalles de ML · M12 · Dashboards, runbooks y on-call

11 En Kubernetes: kube-prometheus-stack, el Operator, ServiceMonitor, PrometheusRule y KEDA

En un cluster, ¿cómo encuentra Prometheus tus pods, quién mide los nodos, los contenedores y los objetos, y cómo lee KEDA una query para escalar?

SINTAXIS CON VARIABLES

```
helm install release oci://ghcr.io/prometheus-community/charts/kube-prometheus-stack \
  --version version -n ns --create-namespace
```

kube-prometheus-stack es un chart de Helm (un paquete de manifests con valores por defecto) que instala el **Prometheus Operator** (un controlador que convierte objetos de Kubernetes en configuración de Prometheus) con sus CRDs (las definiciones de los tipos nuevos: ServiceMonitor, PodMonitor, PrometheusRule…), Prometheus, Alertmanager, Grafana, kube-state-metrics, node_exporter, reglas y dashboards. Fija `--version`.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: nombre
  namespace: ns_app
  labels:
    release: release
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: app
  endpoints:
    - port: puerto
      path: /metrics
      interval: intervalo
```

Un **ServiceMonitor** le dice al Operator: lee `/metrics` de los pods que están detrás de los Services con estas labels, en el puerto con este nombre. La label `release` tiene que coincidir con el `serviceMonitorSelector` de tu Prometheus (el filtro con el que elige ServiceMonitors); si no, se ignora sin avisar.

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: nombre
  namespace: ns_app
  labels:
    release: release
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: app
  podMetricsEndpoints:
    - port: puerto_pod
      interval: intervalo
```

Un **PodMonitor** elige pods directo, sin Service: sirve para workers o jobs que nadie llama. `port` es el nombre del puerto del contenedor; si el pod no declara ese nombre, no aparece ningún target.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
release	nombre del release de Helm; el chart lo usa como label para elegir objetos	kube-prometheus-stack
version	versión fija del chart	92.11.0 (trae el Operator v0.94.1)
ns	namespace de Prometheus	monitoring (el C01 lo da por hecho)
nombre, ns_app	nombre y namespace del ServiceMonitor, PodMonitor o PrometheusRule	gateway, triton · classifier
app	label que comparten el Service (o los pods) que quieres medir	gateway
puerto	el NOMBRE del puerto del Service, no el número	http (8080) · metrics (8002)
puerto_pod	el nombre del puerto del contenedor	metrics
intervalo	cada cuánto se lee; sin él, el de Prometheus (30 s con el Operator)	15s
grupo, Alerta, expr, for, severidad	lo mismo que en alerts.yml	classifier-alerts · TritonNotReady · … · 5m · page
Ready	réplicas listas del Deployment	2
valor de la query	lo que devuelve la query de KEDA: un solo número	25 (ms de espera en la cola)
threshold	el objetivo del trigger	10 (ms)
ventana	el rango de rate() dentro de la query: Prometheus necesita 2 muestras adentro; la regla práctica de Grafana ($__rate_interval) pide 4 intervalos	[2m] en KEDA (8 muestras), [5m] en las alertas (20)
interval	intervalo de scrape del target	15 s (los ServiceMonitors del C01)

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: nombre
  namespace: ns_app
  labels:
    release: release
spec:
  groups:
    - name: grupo
      rules:
        - alert: Alerta
          expr: expr
          for: for
          labels:
            severity: severidad
```

Las reglas van como **PrometheusRule**: el Operator no lee archivos sueltos. El `spec` es el mismo YAML de `groups` de `alerts.yml`, también con la label `release` (para el `ruleSelector`, el filtro de reglas). Un webhook del chart (un chequeo que el API server hace antes de guardar el objeto) rechaza las reglas que no parsean: Helm muestra `Rules are not valid`; `kubectl apply`, `"prometheusrules" is invalid` y la regla que falló.

```
kubectl get
servicemonitors,podmonitors,prometheusrules -A
--show-labels
kubectl get prometheus -n ns -o
jsonpath='{.items[0].spec.serviceMonitorSelector}'
kubectl port-forward -n ns svc/prometheus-operated
9090
```

Primero, ¿existe el objeto y trae la label? Después, ¿qué selector usa tu Prometheus? `prometheus-operated` es el Service que crea el Operator: el mismo que consulta KEDA en el C01.

FÓRMULAS

$$\text{deseadas} = \left\lceil \text{Ready} \times \frac{\text{valor de la query}}{\text{threshold}} \right\rceil = \left\lceil 2 \times \frac{25}{10} \right\rceil = 5 \rightarrow 4 \ (\text{maxReplicaCount})$$

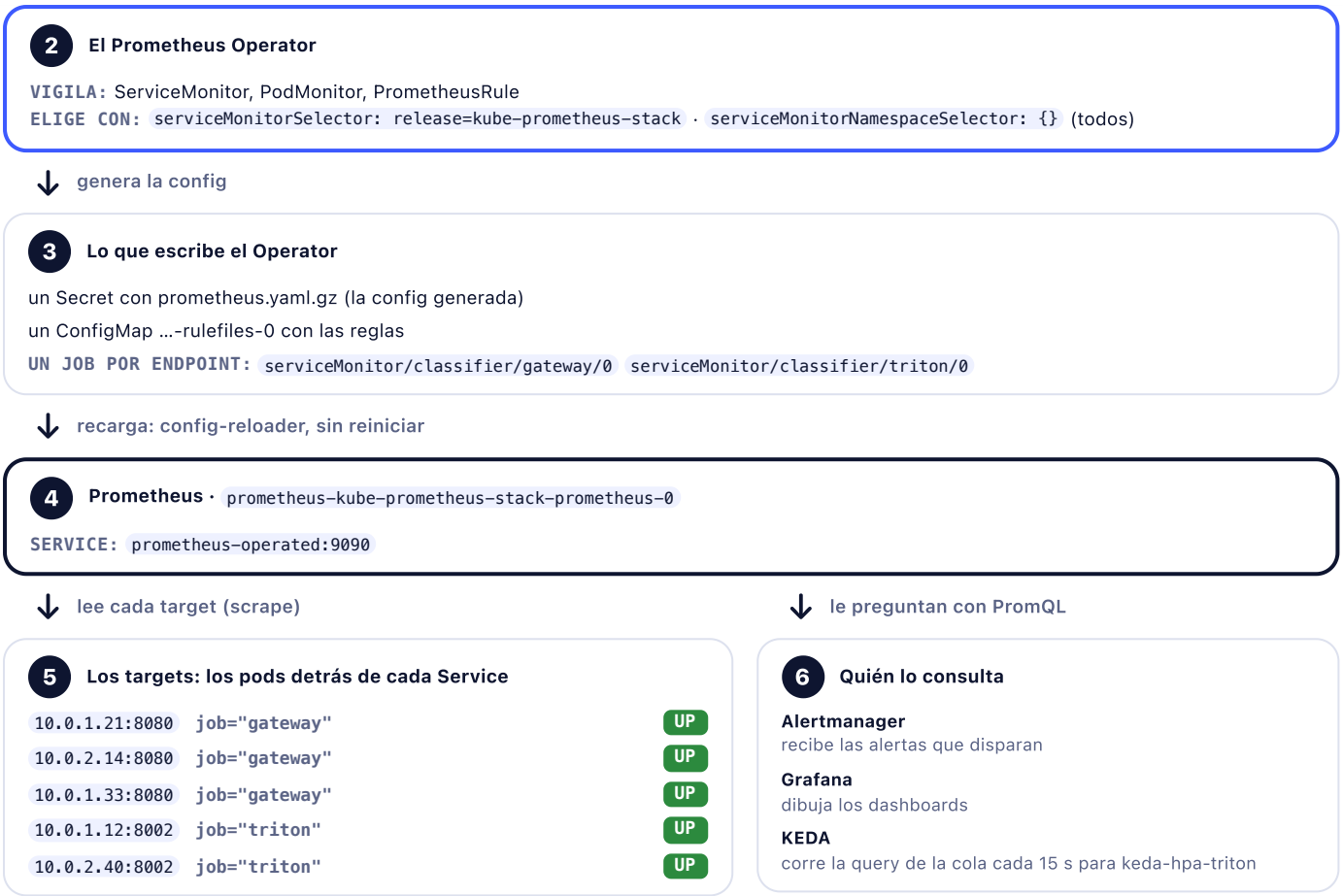
$$\text{muestras} \approx \frac{\text{ventana}}{\text{interval}} \geq 2 \ (\text{mínimo de Prometheus}) \qquad \text{ventana} \geq 4 \times \text{interval (regla de Grafana)} \qquad [2\text{m}] \geq 4 \times 15\,\text{s} = 1\,\text{min}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
kube-prometheus-stack	Chart de Helm con el Operator, Prometheus, Alertmanager, Grafana, kube-state-metrics y node_exporter, más reglas y dashboards por defecto.	`chart 92.11.0 (Operator v0.94.1)`
Prometheus Operator	Controlador que vigila ServiceMonitors, PodMonitors y PrometheusRules y reescribe la config de Prometheus. No le escribes prometheus.yml a mano.	`deploy kube-prometheus-stack-operator`
Prometheus (el objeto)	Recurso que describe tu Prometheus: réplicas, retención y selectores (qué objetos toma). El Operator crea el StatefulSet (el objeto que mantiene pods con nombre fijo y disco propio) y el Service prometheus-operated. En el chart, la retención es de 10 días (sin el chart, Prometheus guarda 15): alcanza para los 7 días que pide ClassifierLowConfidenceSpike, pero no para un presupuesto de 30 días.	`pod prometheus-kube-prometheus-stack-prometheus-0`

PIEZA	QUÉ HACE	EJEMPLO
serviceMonitorSelector	Qué ServiceMonitors toma ese Prometheus. El chart pone release con el nombre del release: un ServiceMonitor sin esa label no existe para él.	`{"matchLabels":{"release":"kube-prometheus-stack"}}`
serviceMonitorNamespaceSelector	En qué namespaces los busca. Vacío ({}) = todos, el default del chart; sin definir = solo el namespace de Prometheus.	`{} en kube-prometheus-stack`
ServiceMonitor	Elige Services por label y lee los pods que tienen detrás. El job queda con el nombre del Service; el scrape pool, serviceMonitor/namespace/nombre/número de endpoint.	`serviceMonitor/classifier/gateway/0`
PodMonitor	Elige pods sin pasar por un Service.	`workers que leen de una cola`
PrometheusRule	Reglas de grabación y alertas como objeto. El Operator las monta en Prometheus; el webhook del chart rechaza las que no parsean.	`classifier: 1 regla de grabación + 6 alertas`
config-reloader	Contenedor al lado de Prometheus que lo recarga cuando cambia la config generada, sin reiniciar el pod.	`un ServiceMonitor nuevo aparece en uno o dos minutos`
kube-state-metrics	Lee el API server y publica el estado de los objetos: réplicas, fases, reinicios y motivos.	`kube_deployment_status_replicas_available → TritonNotReady`
cAdvisor (dentro del kubelet)	El uso real de cada contenedor: CPU y memoria, y throttling (las pausas que impone un límite de CPU) solo si el contenedor tiene ese límite. El chart lo lee en /metrics/cadvisor de cada nodo. El gateway del C01 corre sin límite de CPU: no tiene serie de throttling.	`container_cpu_usage_seconds_total del gateway`
node_exporter	Un pod por nodo (un DaemonSet) en el puerto 9100 con la máquina: CPU, memoria, disco y red.	`node_memory_MemAvailable_bytes`
metrics-server	No es Prometheus: publica CPU y memoria para kubectl top y para el HPA por CPU.	`el HPA del gateway`
KEDA (trigger prometheus)	Corre tu query en Prometheus y le pasa el número al HPA que crea (keda-hpa- más el nombre). La query tiene que devolver una sola serie.	`serverAddress: http://prometheus-operated.monitoring.svc.cluster.local:9090`
ignoreNullValues	Con true (el default), una query vacía, NaN o +Inf vale 0 en silencio; con "false", es un error visible en los eventos del ScaledObject.	`"false" en el C01`
TargetDown · Watchdog	Dos alertas que trae el chart: TargetDown (más del 10 % de los targets de un job caídos durante 10 min) y Watchdog (siempre encendida, para probar que la cadena de alertas funciona).	`Watchdog va al receiver null (descarta todo) por defecto`





El Operator convierte objetos en config: toma los ServiceMonitors y PrometheusRules con la label release, genera un job por endpoint y el config-reloader recarga Prometheus. Los targets son los pods detrás de cada Service. IPs ilustrativas.

QUIÉN MIDE QUÉ EN UN CLUSTER

FUENTE	QUÉ MIDE	EJEMPLO	EN EL C01
Tu app (prometheus_client)	lo que solo tu código sabe: requests, latencia por tramo, scores	**classifier_requests_total**	el gateway, en :8080/metrics; los requests en vuelo que el curso cuenta como saturación no salen ahí: main.py no los exporta como gauge
Triton	cola, cómputo y batches por modelo, y la GPU	**nv_inference_queue_duration_us**	puerto 8002; la query de KEDA y TritonQueueTimeHigh
kube-state-metrics	el estado de los objetos de Kubernetes, leído del API server	**kube_deployment_status_replicas_available**	TritonNotReady (no existe en el compose local)
cAdvisor (kubelet)	cuánto usa de verdad cada contenedor	**container_memory_working_set_bytes**	la CPU del gateway: hoy, su única señal de saturación exportada (sin límite de CPU no hay throttling)
node_exporter	la máquina: CPU, memoria, disco y red del nodo	**node_filesystem_avail_bytes**	sin alerta propia en el C01
DCGM exporter	la GPU vista por NVIDIA: utilización, memoria, temperatura, errores	**DCGM_FI_DEV_GPU_UTIL**	lo instala el GPU Operator; el C01 usa nv_gpu_utilization de Triton
metrics-server	CPU y memoria al instante para kubectl top y el HPA; no guarda historia	–	el HPA del gateway al 60 % de su request

La pregunta elige la fuente: ¿el Deployment tiene sus réplicas listas? kube-state-metrics. ¿El contenedor se acerca a su límite de memoria? cAdvisor. ¿Se llena el disco del nodo? node_exporter. ¿Fallan los requests? Tu app.

EJEMPLO REAL · C01: LOS DOS SERVICEMONITORS

QUE CONECTAN EL GATEWAY Y TRITON CON PROMETHEUS

YAML reference/c01/k8s/monitoring.yaml (completo)

```yaml
# Scrape con el Prometheus Operator (kube-prometheus-stack) (caso c01). Sin esto, Prometheus nunca
# lee el gateway ni Triton: la consulta de KEDA vuelve vacía (k8s/keda-triton.yaml la marca como
# error), Triton se queda en 2 réplicas y ninguna alerta evalúa. Falla en silencio.
# Las reglas de monitoring/alerts.yml van aparte, como PrometheusRule (el Operator no lee archivos
# de reglas): el comando está en el README.
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: gateway
  namespace: classifier
  labels:
    release: kube-prometheus-stack  # = serviceMonitorSelector de tu Prometheus (nombre del release de Helm)
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: gateway  # labels del Service de k8s/gateway.yaml
  endpoints:
    - port: http  # nombre del puerto del Service: 8080/metrics del gateway (no expuesto en el Ingress)
      path: /metrics
      interval: 15s  # igual que en local: suficiente para alertas de 5–30 min
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: triton
  namespace: classifier
  labels:
    release: kube-prometheus-stack
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: triton  # labels del Service de k8s/triton.yaml
  endpoints:
    - port: metrics  # 8002: métricas nv_* (las que consulta KEDA); la NetworkPolicy solo deja entrar a "monitoring"
      interval: 15s
```

- `release: kube-prometheus-stack` es la label que busca el `serviceMonitorSelector` que arma el chart (release igual al nombre del release de Helm). Si instalas el chart con otro nombre, cambia esta label: si no, el ServiceMonitor no existe para Prometheus, ni UP ni DOWN.
- `port: http` y `port: metrics` son **nombres** de puertos de los Services de `gateway.yaml` y `triton.yaml`, no números. El job queda con el nombre del Service, así que `up{job="gateway"}` funciona igual que en el compose local.
- `interval: 15s` pisa el default del Operator (30 s) y queda igual que `prometheus.yml` en local: la ventana de 2 min de KEDA y las de 5 min de las alertas tienen 8 y 20 muestras por serie. Ojo: las reglas sin `interval` propio se evalúan cada 30 s en el cluster (el default del Operator), no cada 15 s como en local.
- La NetworkPolicy de `triton.yaml` solo deja leer el puerto 8002 desde el namespace `monitoring` (si el plugin de red del cluster, el CNI, aplica NetworkPolicies, como pide ese archivo). Si Prometheus vive en otro namespace, Triton sale DOWN con `context deadline exceeded`: el ejemplo de la sección 12.
- Las reglas no van aquí: el README del C01 envuelve `alerts.yml` en un PrometheusRule con la misma label `release`. Sin ese paso, las 6 alertas no existen en el cluster.
- KEDA lee este mismo Prometheus por `prometheus-operated.monitoring.svc.cluster.local:9090`. Si falta el ServiceMonitor de Triton, su query vuelve vacía y, con `ignoreNullValues: "false"`, el ScaledObject muestra `prometheus metrics 'prometheus' target may be lost, the result is empty` en sus eventos.

CÓMO RECORDARLA

no escribes prometheus.yml: escribes objetos. ServiceMonitor = qué leer (por Service), PodMonitor = qué leer (por pod), PrometheusRule = qué evaluar; la label `release` es la llave. kube-state-metrics dice cómo están los objetos; cAdvisor, cuánto gasta cada contenedor; node_exporter, cómo está la máquina.

TRAMPA

un ServiceMonitor que nadie elige no da error. Sin la label `release`, con un `selector` que copia las labels del pod en vez de las del Service, o con un `port` que el Service no tiene, el target simplemente no aparece: ni UP ni DOWN. En el C01 eso rompe dos cosas a la vez: las alertas se quedan sin datos y KEDA no ve la cola.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M6 · KEDA: escalar por eventos y hasta cero · M6 · HPA: por qué escalar por CPU no sirve para un servicio de GPU · Docker y kubectl · Autoscaling y capacidad · C01 · Monitoreo, alertas y reentrenamiento · C01 · Archivos de referencia · Hoja · Kubernetes · autoscaling

12 Diagnóstico: el método y la tabla de estados

No hay datos, un target está DOWN o la alerta nunca sonó: ¿qué miras primero, qué después y qué te dice cada estado?

SINTAXIS CON VARIABLES

```
curl -s 'http://prometheus/api/v1/targets?state=active' \
  | jq -r '.data.activeTargets[] | [.labels.job, .labels.instance, .health, .lastError] | @tsv'
```

¿Prometheus lee el target? `health` es up, down o unknown (todavía no lo leyó) y `lastError` dice por qué falló. Es lo mismo que Status → Target health en la UI. Si el target ni aparece, el problema está antes: la config o el ServiceMonitor.

```
curl -s http://host:puerto/metrics | grep '^metrica'
```

¿El proceso emite la métrica? Lo que no está aquí no puede estar en Prometheus. Una serie con labels nace con su primera observación: si nadie recibió un 503, no hay línea de 503. En Kubernetes, llega con `kubectl port-forward`.

```
curl -s http://prometheus/api/v1/query --data-urlencode 'query=consulta' | jq '.data.result, .warnings, .infos'
```

¿La query devuelve algo? Una lista vacía no es un 0: no hay serie con ese nombre y esas labels en los últimos 5 min (el **lookback**: cuánto mira hacia atrás una query para encontrar la última muestra). Y lee `warnings` e `infos`: la query pudo correr con una sospecha.

```
curl -s 'http://prometheus/api/v1/rules?type=alert' \
  | jq -r '.data.groups[].rules[] | [.name, .state, .health, .lastError] | @tsv'
amtool --alertmanager.url=http://alertmanager alert query alertname=alerta
```

¿La regla evalúa y en qué estado está? `health` ok o err (con `lastError`) y `state` inactive, pending o firing. Después, **¿la recibió Alertmanager?** Sin flags, `amtool` lista solo las activas (las que sí notifica); `-s` e `-i` muestran solo las silenciadas y las inhibidas, y `-a -s -i`, todas, con su estado.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
prometheus	host y puerto de Prometheus	`localhost:9090` (compose o port-forward)
host, *puerto*	el target, visto desde donde corres curl (Prometheus lo lee como gateway:8080 o triton:8002)	`localhost:8080` · `localhost:8002` (puertos publicados del compose o port-forward)
metrica	nombre de la métrica (o su comienzo)	`classifier_requests_total`
consulta	la query que no devuelve lo esperado	`up{job="triton"}`
alertmanager	host y puerto de Alertmanager	`localhost:9093`
alerta	nombre de la alerta	`ClassifierHigh5xxRate`
t_{aviso}	del primer error a la notificación	≈ 6 min para `ClassifierHigh5xxRate` en una caída total
t_{cruce}	hasta que la expresión es verdadera: un scrape y la ventana	≈ 15–30 s con 100 % de errores (rate necesita dos muestras de la serie de 5xx); ≈ 17 min con 5 % en la de 14.4×
t_{for}	el for de la regla: el tiempo en pending	5 min
t_{eval}	hasta la siguiente evaluación de la regla (evaluation_interval: 1 min si no lo fijas)	≤ 15 s con el `prometheus.yml` del C01; ≤ 30 s con el Operator (EKS)
t_{grupo}	group_wait de Alertmanager: espera antes de la primera notificación de un grupo	30 s (default)
s	intervalo de scrape que Grafana supone para el data source	15 s (el default de Grafana y el del C01)
u	el umbral de la alerta	0.01 (1 % de 5xx)

FÓRMULAS

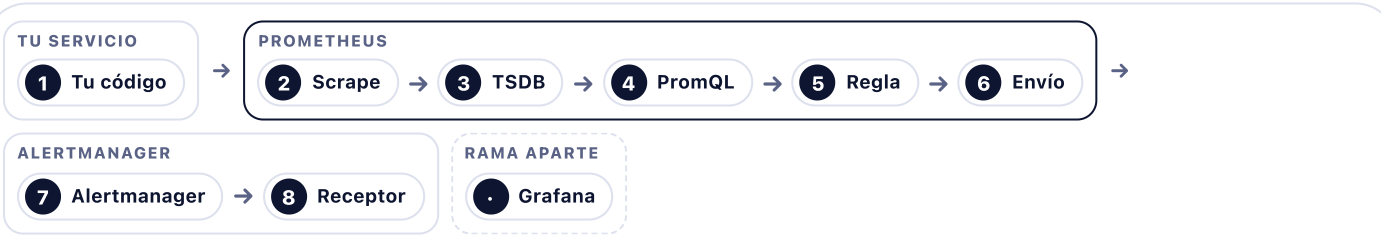
$$t_{aviso} \approx t_{cruce} + t_{for} + t_{eval} + t_{grupo} \approx 15\,\text{s} + 5\,\text{min} + 15\,\text{s} + 30\,\text{s} \approx 6\,\text{min}$$

$$\$__rate_interval = \max(\$__interval + s, 4s)$$

$$\frac{0}{0} = \text{NaN} \qquad \frac{x}{0} = +\infty \qquad \text{NaN} > u \Rightarrow \text{falso: la alerta no dispara}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`up = 1`	El último scrape funcionó: Prometheus leyó /metrics completo.	`up{job="gateway"} 1`
`up = 0 (DOWN)`	El scrape falló. La causa está en lastError: red, puerto, path, timeout o Content-Type.	`context deadline exceeded`
`sin serie up`	No hay target: la config, el ServiceMonitor, su selector o el nombre del puerto no lo generan.	`un ServiceMonitor sin la label release`
`health: unknown`	Target recién descubierto, todavía sin scrape.	`los primeros segundos`
`resultado vacío`	No hay serie con ese nombre y esas labels en los últimos 5 min. No es lo mismo que 0.	`la serie de 503 cuando no hubo ningún 503`
`NaN`	Dividiste 0 entre 0, o histogram_quantile no tuvo observaciones en la ventana.	`la fracción de 5xx sin tráfico`
`+Inf`	Dividiste algo positivo entre 0.	`espera en cola ÷ 0 éxitos, sin clamp_min`
`caída en un counter`	El proceso reinició y el counter volvió a 0 (un counter reset). rate() lo corrige si va antes del sum.	`un pod reemplazado en un deploy`
`hueco en la gráfica`	Scrapes que fallaron: Prometheus marca la serie como stale (vencida) y no inventa valores.	`un /metrics que tardó más que scrape_timeout`
`regla con health err`	La regla no se puede evaluar; lastError trae el mensaje. Mientras falle, no dispara.	`many-to-many matching not allowed`
`alerta pending`	La condición es verdadera, pero no cumplió el for. Si se apaga un instante, el for empieza de cero.	`for: 10m en ClassifierHighP99Latency`
`alerta firing`	Cumplió el for: Prometheus la manda a Alertmanager y la reenvía mientras siga activa.	`ALERTS{alertstate="firing"}`
`silenciada · inhibida`	Alertmanager la tiene, pero no notifica: un silencio la tapa u otra alerta la inhibe.	`amtool alert query -s -i`
`No data (Grafana)`	La query del panel no devolvió series: nombre, labels, rango de tiempo, variables o data source.	`Inspect → Query del panel`



1 Tu código TU SERVICIO

¿El proceso emite la métrica, con esos labels?

TERMINAL

```
curl -s localhost:8080/metrics | grep '^classifier_requests_total'
```

Sin línea: el nombre es otro, o esa combinación de labels todavía no tuvo ninguna observación.

2 Scrape PROMETHEUS

¿Prometheus lo lee?

PROMQL `up{job="gateway"}`

1 = bien; 0 = DOWN, lee lastError; nada = no hay target (config o ServiceMonitor).

3 TSDB PROMETHEUS

¿La serie llegó y sigue viva?

PROMQL

```
count by (job) (classifier_requests_total)
```

Si un deploy cambió el nombre o el target cayó, la serie quedó stale y desaparece de las queries.

4 **PromQL** PROMETHEUS

¿La query hace lo que crees?

PROMQL

```
sum(rate(classifier_requests_total{status=~"5.."}[5m]))
```

Vacío no es 0; NaN es 0/0; lee warnings e infos.

5 **Regla** PROMETHEUS

¿Está cargada, sana y en qué estado?

PROMQL

```
ALERTS{alertname="ClassifierHigh5xxRate"}
```

Rule health: ok o err. Pending espera el for; firing ya salió hacia Alertmanager.

6 **Envío** PROMETHEUS

¿Prometheus conoce y alcanza un Alertmanager?

PROMQL

```
prometheus_notifications_alertmanagers_discovered
```

0 = no hay a quién mandar (el compose del C01). Los fallos suman en prometheus_notifications_errors_total.

7 **Alertmanager** ALERTMANAGER

¿La recibió, la silenció, a qué receiver la mandó?

TERMINAL

```
amtool --alertmanager.url=http://localhost:9093 alert query -a -s -i
```

Estado active = notifica; suppressed = silenciada o inhibida, no notifica. La ruta decide el receiver.

8 **Receptor** ALERTMANAGER

¿El envío final funcionó?

TERMINAL

```
kubectl logs -n monitoring alertmanager-kube-prometheus-stack-alertmanager-0 -c alertmanager | grep 'Notify for alerts failed'
```

Un webhook vencido o sin red de salida: Alertmanager reintenta y lo deja en el log.

RAMA APARTE **Grafana**

¿El panel pregunta lo mismo que tú?

GRAFANA Inspect › Query

Data source, variables, rango de tiempo y $__rate_interval.

El primer eslabón que falla es la causa.

TERMINAL un comando: va en la terminal PROMQL una consulta: va en la caja de consultas de Prometheus (o en Grafana)

un lugar de la web: se abre, no se copia

El método: recorre la cadena en orden, de tu código al receptor. El primer eslabón que falla es la causa; lo que viene después son consecuencias. Grafana es una rama aparte: lee de Prometheus como tú.

EJEMPLO REAL · C01: TRITON DOWN POR LA RED, NO POR TRITON

BASH · ZSH Comandos reales con los nombres de reference/c01/k8s/triton.yaml (NetworkPolicy triton-ingress), monitoring.yaml y keda-triton.yaml; las salidas son ilustrativas, con la forma de Prometheus 3.13 y KEDA 2.21

```
# Síntoma: el panel de la cola de Triton dice No data y KEDA no pasa de 2 réplicas.
# Este Prometheus (kube-prometheus-stack) quedó instalado en el namespace "observability".
kubectl port-forward -n observability svc/prometheus-operated 9090 &
curl -s 'localhost:9090/api/v1/targets?state=active' \
  | jq -r '.data.activeTargets[] | select(.labels.namespace=="classifier") | [.labels.job, .health, .lastError] | @tsv'
# gateway  up
# gateway  up
# gateway  up
# triton   down  Get "http://10.0.1.12:8002/metrics": context deadline exceeded
# triton   down  Get "http://10.0.2.40:8002/metrics": context deadline exceeded

# El target existe (el ServiceMonitor está bien), pero no contesta a tiempo: ¿el proceso o la red?
kubectl port-forward -n classifier svc/triton 8002 &
curl -s localhost:8002/metrics | grep -c '^nv_inference_queue_duration_us'
```

```
# 1          ← por el port-forward sí contesta: el proceso está bien
kubectl get networkpolicy triton-ingress -n classifier -o jsonpath='{.spec.ingress[1].from}'
# [{"namespaceSelector":{"matchLabels":{"kubernetes.io/metadata.name":"monitoring"}}}]

# Causa: solo el namespace "monitoring" puede leer el 8002. KEDA tampoco llega a Prometheus:
kubectl describe scaledobject triton -n classifier | grep KEDAScalerFailed
# Warning  KEDAScalerFailed  …  …: lookup prometheus-operated.monitoring.svc.cluster.local …: no such host
```

- Lee los tres estados: el gateway sale UP (se lee bien), Triton sale DOWN con un error (el target existe y el scrape falla) y ningún target falta. Si faltara, el problema sería de selección (sección 11); aquí el ServiceMonitor está bien.
- `context deadline exceeded` = nadie contestó antes de `scrape_timeout` (10 s por defecto). Una NetworkPolicy descarta los paquetes sin responder: por eso es un timeout y no un `connection refused`.
- El `port-forward` entra al pod a través del kubelet y no pasa por la NetworkPolicy: si así contesta y Prometheus no, el problema es la red, no el proceso.
- El namespace `monitoring` es parte del contrato del C01: lo usan la NetworkPolicy de `triton.yaml` y el `serverAddress` de `keda-triton.yaml`. O instalas Prometheus ahí, o cambias los dos archivos en git.
- Mientras tanto, `TritonQueueTimeHigh` y `GPUSaturated` no pueden disparar (sus series de Triton no existen: la expresión vuelve vacía) y KEDA no escala: un error de monitoreo deja ciegos a la vez a las alertas y al autoscaling. Una alerta sobre `up == 0` (o la TargetDown del chart, a los 10 min) lo habría avisado, si Alertmanager tiene a quién mandarla.

CÓMO RECORDARLA

código → scrape → serie → query → regla → envío → Alertmanager → receptor. Sigue la cadena en orden y lee el estado exacto: DOWN (falla el scrape), ausente (no hay target), vacío (no hay serie), NaN (dividiste 0 entre 0), pending (esperando el for).

TRAMPA

leer **vacío** como **cero** y **sin alerta** como **todo bien**. Cuando un target cae, Prometheus marca sus series como stale (vencidas) en el scrape siguiente: una consulta directa vuelve vacía enseguida, y una con `rate(…[5m])`, cuando su ventana se queda sin muestras. Las alertas sobre esas series se apagan justo cuando todo está caído. Por eso hacen falta `up == 0`, `absent()` y una alerta que siempre suene (el Watchdog del chart) mandada a un servicio de afuera que avise cuando deja de llegar.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Dashboards, runbooks y on-call · C01 · Monitoreo, alertas y reentrenamiento · C01 · Modos de falla y degradación · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · Autoscaling y capacidad · Hoja · Kubernetes · diagnóstico

13 Errores que debes reconocer rápido

Lees el mensaje: ¿quién lo dijo (el scrape, PromQL, Alertmanager, Grafana, tu código o el cluster), qué significa, qué corres primero y dónde está la explicación?

SINTAXIS CON VARIABLES

`Get "url": error_de_red`

Lo dijo el **scrape**: Prometheus como cliente HTTP de tu target. Aparece en Status → Target health y en `lastError`, y `up` vale 0.

`invalid parameter "query": fila:columna: parse error: detalle`

Lo dijo el **parser** de PromQL (la parte que lee la query antes de correrla; errorType `bad_data`): tu query no llegó a correr. Fila y columna señalan dónde se perdió. Un error del motor, en cambio, sale con errorType `execution`.

`PromQL info: detalle`
`PromQL warning: detalle`

Una **anotación** (desde Prometheus 2.48): la query sí corrió, pero algo es sospechoso. La UI la muestra sobre el resultado; la API, en los campos `infos` y `warnings`.

`time=hora level=nivel source=archivo.go:linea msg="mensaje" err="detalle"`

Una línea de log de Prometheus 3 o de Alertmanager: la escribe el servidor (config, disco, envío de alertas). Léela con `docker logs` o `kubectl logs`.

`modulo.Excepcion: mensaje`

Un traceback de Python: lo dijo **prometheus_client**, dentro de tu proceso, al crear o usar una métrica. Pasa en cada arranque o en cada request que toca esa línea.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
url	lo que Prometheus intentó leer	`http://10.0.1.12:8002/metrics`
error_de_red	el error de red de Go	`context deadline exceeded · connect: connection refused · no such host`
fila : *columna*	dónde se trabó el parser, contando desde 1	`1:6`
detalle	el porqué, en una frase	`unclosed left parenthesis`
hora , *nivel*	cuándo y qué tan grave	`2026-10-23T10:00:00.000Z · ERROR, WARN o INFO`
archivo.go: *linea*	de qué parte del código salió	`main.go:745`
mensaje	qué pasó	`Error loading config (--config.file=/etc/prometheus/prometheus.yml)`
modulo.*Excepcion*	la clase de la excepción	`prometheus_client.registry.DuplicateTimeseries`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Target health · lastError	El scrape: Prometheus no pudo leer /metrics. up vale 0 para ese target.	`server returned HTTP status 404 Not Found`
parse error · bad_data	El parser de PromQL: la query no corrió.	`expected type range vector in call to function "rate"`
errorType execution	El motor: la query parseó, pero no se pudo calcular.	`many-to-many matching not allowed`
errorType timeout	La query pasó --query.timeout (2 min por defecto).	`query timed out in expression evaluation`
PromQL info · PromQL warning	Anotaciones: corrió, pero algo es sospechoso.	`metric might not be a counter`
promtool check config · check rules	El validador: encuentra YAML, campos y PromQL inválidos antes de cargarlos.	`could not parse expression`
Log de Prometheus	El servidor: config, almacenamiento, reglas y envío a Alertmanager.	`msg="Error sending alerts"`
amtool · log de Alertmanager	La config de Alertmanager y las notificaciones que fallan.	`undefined receiver "pagerduty-ml" used in route`
Panel o data source de Grafana	La query del panel o la conexión; el error de Prometheus llega tal cual.	`There was an error returned querying the Prometheus API`
Traceback de Python	prometheus_client en tu proceso.	`Duplicated timeseries in CollectorRegistry`
kubectl · API server	El cluster no conoce el tipo (faltan los CRDs) o un webhook rechazó el objeto.	`no matches for kind "ServiceMonitor"`

PIEZA	QUÉ HACE	EJEMPLO
Eventos del ScaledObject	KEDA no pudo leer su métrica (KEDAScalerFailed).	`target may be lost, the result is empty`
Eventos del ServiceMonitor	El Operator rechazó el objeto (InvalidConfiguration): no genera config para él.	`was rejected due to invalid configuration`

LOS 26 MENSAJES

`server returned HTTP status 404 Not Found`

Prometheus llegó al target, pero en ese path no hay métricas: metrics_path (por defecto /metrics) no coincide con la ruta de tu app. La red y el proceso están bien; up vale 0 igual. Un redirect (un 307 a /metrics/) no da este error, porque Prometheus lo sigue, pero cada scrape son dos requests: por eso el gateway del C01 sirve /metrics directo.

PRIMERO:

```
curl -s -o /dev/null -w '%{http_code}\n'
localhost:8080/metrics
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

`Get "http://10.0.1.12:8002/metrics": context deadline exceeded`

Nadie terminó de contestar antes de scrape_timeout (10 s por defecto, nunca mayor que el intervalo). O la red descarta los paquetes (una NetworkPolicy o un security group, el firewall de AWS; Triton en el C01 si Prometheus no vive en el namespace monitoring), o el proceso está tan cargado que /metrics tarda más que eso.

PRIMERO: `kubectl get networkpolicy -n classifier` ·

```
curl -s localhost:9090/api/v1/query --data-urlencode
'query=scrape_duration_seconds{job="triton"}' | jq -r
'.data.result[].value[1]'
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

`dial tcp 10.0.1.21:8080: connect: connection refused`

La máquina existe, pero nadie escucha en ese puerto: el proceso no arrancó, se cayó, o el puerto del target es otro. El rechazo es inmediato: a diferencia del timeout, no hay nada que esperar. En Kubernetes, revisa el puerto del Service que nombra el ServiceMonitor.

PRIMERO:

```
kubectl get svc gateway -n classifier -o
jsonpath='{.spec.ports}'
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

`dial tcp: lookup gateway on 127.0.0.11:53: no such host`

El nombre del target no resuelve en el DNS de donde corre Prometheus. 127.0.0.11 es el DNS interno de Docker en las redes de compose: Prometheus está en una, pero gateway no (otro proyecto de compose, o el servicio se llama distinto). Desde tu Mac o desde un contenedor lanzado aparte tampoco resuelve. En Kubernetes: un Service o un namespace mal escrito.

PRIMERO:

```
docker network inspect classifier-c01_default --format
'{{range .Containers}}{{.Name}} {{end}}'
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

`received unsupported Content-Type "application/json" and no fallback_scrape_protocol specified for target`

Desde Prometheus 3.0 el scrape falla si el Content-Type no es de métricas (en 2.x se leía como texto igual). El caso típico en FastAPI: una ruta /metrics que hace return generate_latest() sin Response, y FastAPI lo manda como JSON. Sírvelo con Response(generate_latest(), media_type=CONTENT_TYPE_LATEST), como main.py del C01. fallback_scrape_protocol es para exporters viejos con un header malo, no para esto.

PRIMERO:

```
curl -sI localhost:8080/metrics | grep -i content-type
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

`line 12: field scrape_config not found in type config.plain`

El YAML trae un campo que Prometheus no conoce: aquí falta la s de scrape_configs (config.plain es el tipo interno de la config). En la línea de arriba viene parsing YAML file …: yaml: unmarshal errors. Al arrancar, Prometheus sale con Error loading config; en un reload, se queda con la config anterior y prometheus_config_last_reload_successful vale 0.

PRIMERO:

```
docker run --rm --entrypoint promtool -v
"$PWD/monitoring":/etc/prometheus:ro
prom/prometheus:v3.13.3 check config
/etc/prometheus/prometheus.yml
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

`Lifecycle API is not enabled.`

Pediste un reload por HTTP (POST /-/reload) y Prometheus corre sin --web.enable-lifecycle, que viene apagado. La config nueva no se cargó. Mándale SIGHUP, reinícialo o arráncalo con esa flag. Con el Operator no hace falta: el config-reloader recarga solo.

PRIMERO: `docker compose kill -s SIGHUP prometheus` ·

```
docker compose logs prometheus | grep 'Completed loading
of configuration file'
```

→ Sección 4 · Configurar el scrape: prometheus.yml, jobs, targets, relabeling básico, up y la página de targets

found duplicate series for the match group {pod="…"} on the right hand-side of the operation: […];many-to-many matching not allowed: matching labels must be unique on one side

Una operación entre dos vectores (una división, una suma) encontró varias series del mismo lado para la misma combinación de labels del on(...). Casi siempre falta agregar un lado: sum by (pod) arriba y abajo. El mensaje dice qué lado tiene los repetidos y muestra dos de ellos.

PRIMERO:

```
curl -s localhost:9090/api/v1/query --data-urlencode
'query=count by (pod)
(rate(classifier_requests_total[5m]))' | jq -r
'.data.result[] | [.metric.pod, .value[1]] | @tsv'
```

→ Sección 5 · PromQL básico: selectores, rangos, rate y agregaciones

vector cannot contain metrics with the same labelset

El resultado quedó con dos series de labels idénticas. El caso típico: rate() sobre varias métricas a la vez ({__name__=~"..."}) borra el nombre, y dos series que solo se distinguían por él chocan. Consulta cada métrica por separado, o agrega por un label que las distinga.

PRIMERO:

```
curl -s localhost:9090/api/v1/query --data-urlencode
'query=count by (__name__)
({__name__=~"nv_inference_(count|exec_count)"})' | jq -r
'.data.result[].metric.__name__'
```

→ Sección 5 · PromQL básico: selectores, rangos, rate y agregaciones

invalid parameter "query": 1:6: parse error: expected type range vector in call to function "rate", got instant vector

rate(), increase() y las funciones _over_time necesitan una ventana: rate(x[5m]), no rate(x). Es un error de sintaxis (errorType bad_data): la query no llegó a correr. 1:6 es fila y columna dentro de la query.

PRIMERO:

```
curl -s localhost:9090/api/v1/query --data-urlencode
'query=rate(classifier_requests_total[5m])' | jq -r
.status
```

→ Sección 5 · PromQL básico: selectores, rangos, rate y agregaciones

PromQL info: metric might not be a counter, name does not end in _total/_sum/_count/_bucket: "nv_inference_queue_duration_us"

Una anotación, no un error: la query corrió. Prometheus avisa que hiciste rate() o increase() sobre algo cuyo nombre no parece de counter. Sobre un gauge (nv_gpu_utilization) es un bug: el rate de un gauge no significa nada. Con Triton es una falsa alarma: sus counters no terminan en _total. Existe desde Prometheus 2.48.

PRIMERO:

```
curl -s localhost:8002/metrics | grep '^# TYPE
nv_inference_queue_duration_us'
```

→ Sección 5 · PromQL básico: selectores, rangos, rate y agregaciones

query processing would load too many samples into memory in query execution

La query tuvo que tener en memoria, a la vez, más muestras que --query.max-samples (50 millones por defecto). Una ventana larga dentro de rate() o increase() no llega sola: Prometheus la carga de a una serie (172,800 muestras para [30d] con scrape cada 15 s), así que tarda pero corre. Lo que choca es juntarlo todo: una subquery de 30 días (rate(...[5m])[30d:15s] guarda cada paso de cada serie: con unas 300 series ya pasa el tope), una ventana cruda pedida entera a la API (x[30d]) o un rango largo con miles de series. Para el presupuesto del mes, graba sum(rate(...[5m])) del numerador y del denominador y divide sus sum_over_time(...[30d]).

PRIMERO:

```
curl -s localhost:9090/api/v1/query --data-urlencode
'query=count(last_over_time(classifier_requests_total[30
d]))' | jq -r '.data.result[0].value[1]'
```

→ Sección 10 · SLOs: golden signals, RED y USE, error budget y alertas por burn rate

PromQL warning: bucket label "le" is missing or has a malformed value of ""

histogram_quantile recibió series sin le: lo perdiste al agregar (sum by (source) sin le) o le pasaste _sum o _count en vez de _bucket. El resultado sale vacío. El le tiene que sobrevivir a la agregación: sum by (le), o sum by (le, source) si quieres un p99 por source. Es una anotación desde Prometheus 2.48.

PRIMERO:

```
curl -s localhost:9090/api/v1/query --data-urlencode
'query=histogram_quantile(0.99, sum by (le)
(rate(classifier_request_latency_seconds_bucket[5m])))'
| jq '.data.result, .warnings'
```

→ Sección 6 · Histogramas y percentiles: buckets, le y histogram_quantile

group "classifier-alerts", rule 1, "ClassifierHighP99Latency": could not parse expression: …

promtool (o Prometheus al cargar las reglas) no pudo parsear la expr de esa regla. Antes de group vienen la línea y la columna del expr en el YAML; después de could not parse expression, el error de PromQL con fila y columna dentro de la expr. Prometheus no carga ese archivo: al arrancar sale con error y en un reload sigue con las reglas anteriores.

PRIMERO:

```
docker run --rm --entrypoint promtool -v
"$PWD/monitoring":/etc/prometheus:ro
prom/prometheus:v3.13.3 check rules
/etc/prometheus/alerts.yml
```

→ Sección 7 · Recording rules y alerting rules: expr, for, labels y annotations

undefined receiver "pagerduty-ml" used in route

Una ruta de Alertmanager manda a un receiver que no está en la lista receivers: un typo, o lo borraste. Alertmanager no arranca con esa config, y en un reload sigue con la anterior. amtool check-config lo atrapa antes de aplicar.

PRIMERO: `amtool check-config alertmanager.yml`

→ Sección 8 · Alertmanager: ruteo, agrupación, inhibición, silencios y receivers

msg="Notify for alerts failed" … err="slack-ml/slack[0]: notify retry canceled after 4 attempts: …"

Alertmanager tiene la alerta y la ruteó, pero el envío al receptor falló: la URL o el token del webhook (la dirección que recibe el aviso), la red de salida o un rechazo del servicio. Reintenta hasta que vence el plazo y lo deja en su log; alertmanager_notifications_failed_total sube. Lo que sigue a attempts: es el último error (la respuesta del receptor o el de la red). Si el receptor contesta con un error que reintentar no arregla (por ejemplo un 4xx), el texto es notify retry canceled due to unrecoverable error after 1 attempts.

PRIMERO:

```
kubectl logs -n monitoring alertmanager-kube-prometheus-stack-alertmanager-0 -c alertmanager | grep 'Notify for alerts failed'
```

→ Sección 8 · Alertmanager: ruteo, agrupación, inhibición, silencios y receivers

msg="Error sending alerts" … err="Post \"http://alertmanager:9093/api/v2/alerts\": dial tcp …: connect: connection refused"

Lo dice Prometheus: no pudo entregar las alertas a Alertmanager (la dirección o el puerto de alerting.alertmanagers están mal, o Alertmanager está caído). Las alertas salen firing en /alerts, pero nadie las recibe; prometheus_notifications_errors_total sube. Hasta Prometheus 3.0 el texto era Error sending alert, en singular.

PRIMERO:

```
curl -s localhost:9090/api/v1/alertmanagers | jq '.data.activeAlertmanagers'
```

→ Sección 8 · Alertmanager: ruteo, agrupación, inhibición, silencios y receivers

There was an error returned querying the Prometheus API

Grafana no pudo hablar con Prometheus: la URL, el protocolo o el puerto del data source, o Prometheus no se alcanza desde donde corre Grafana. El clásico: Grafana en un contenedor con http://localhost:9090, y ahí localhost es el propio Grafana. Usa el nombre del servicio (http://prometheus:9090 en compose) o el Service del cluster.

PRIMERO: `curl -s localhost:9090/-/ready` ·

GRAFANA Connections › Data sources › Prometheus › Save & test

→ Sección 9 · Grafana: data sources, dashboards, paneles, variables y dashboards como código

Datasource ${DS_PROMETHEUS} was not found

El dashboard viene de un JSON exportado para compartir afuera: trae la entrada ${DS_PROMETHEUS}, que solo se llena al importarlo a mano. Por provisioning (dashboards como código) nadie la llena. Exporta sin esa opción, o reemplaza la referencia por el uid (el id fijo) de tu data source.

PRIMERO: `grep -rn 'DS_PROMETHEUS' dashboards/`

→ Sección 9 · Grafana: data sources, dashboards, paneles, variables y dashboards como código

prometheus_client.registry.DuplicateTimeseries: Duplicated timeseries in CollectorRegistry: {'classifier_requests_total', …}

Registraste dos veces una métrica con el mismo nombre en el registro global de prometheus_client: el módulo se importó dos veces (con nombres distintos, o un test que lo recarga) o re-ejecutaste la celda del notebook. Un counter reserva tres nombres (sin sufijo, _total y _created). Desde prometheus-client 0.26.0 (la del C01) la excepción es DuplicateTimeseries, una subclase de ValueError; antes salía ValueError: con el mismo texto. Las métricas se crean una vez, al nivel del módulo.

PRIMERO: `grep -rnE "Counter\(|Histogram\(" app/`

→ Sección 3 · Instrumentar tu código y usar exporters: prometheus_client, el gateway de C01, Triton, node_exporter, kube-state-metrics y cAdvisor

ValueError: counter metric is missing label values

Llamaste .inc() directo sobre una métrica que declaró labels: primero va .labels(status=…, source=…). Si los nombres de .labels() no son los declarados, el error es Incorrect label names. Revienta en el request, no al arrancar: un test que pase por ese camino lo atrapa.

PRIMERO: `grep -rnE "\.(inc|observe)\(" app/`

→ Sección 3 · Instrumentar tu código y usar exporters: prometheus_client, el gateway de C01, Triton, node_exporter, kube-state-metrics y cAdvisor

no matches for kind "ServiceMonitor" in version "monitoring.coreos.com/v1"

El cluster no conoce el tipo ServiceMonitor: faltan los CRDs (las definiciones de tipos nuevos) del Prometheus Operator, porque kube-prometheus-stack no está instalado o se instaló después. kubectl termina con ensure CRDs are installed first. Aplica monitoring.yaml después del chart.

PRIMERO:

```
kubectl get crd servicemonitors.monitoring.coreos.com
```

→ Sección 11 · En Kubernetes: kube-prometheus-stack, el Operator, ServiceMonitor, PrometheusRule y KEDA

prometheus metrics 'prometheus' target may be lost, the result is empty

Lo dice KEDA con ignoreNullValues: "false", como en el C01: la query de su trigger no devolvió ninguna serie. Casi siempre Prometheus no scrapea a Triton (falta el ServiceMonitor o su label release) o la query pide un label que no existe. Con el default (true), KEDA lo tomaría como 0 y nada avisaría. Sale como evento KEDAScalerFailed del ScaledObject.

PRIMERO:

```
kubectl describe scaledobject triton -n classifier
```

→ Sección 11 · En Kubernetes: kube-prometheus-stack, el Operator, ServiceMonitor, PrometheusRule y KEDA

prometheus query rate(nv_inference_queue_duration_us{model="classifier"}[2m]) returned multiple elements

La query de KEDA devolvió más de una serie, y KEDA necesita un solo número. Pasa al quitar el sum() de afuera: queda una serie por pod de Triton. Agrega con sum() (o max()) hasta que devuelva una sola serie.

PRIMERO:

```
curl -s localhost:9090/api/v1/query --data-urlencode 'query=rate(nv_inference_queue_duration_us{model="classifier"}[2m])' | jq '.data.result | length'
```

→ Sección 11 · En Kubernetes: kube-prometheus-stack, el Operator, ServiceMonitor, PrometheusRule y KEDA

admission webhook "prometheusrulevalidate.monitoring.coreos.com" denied the request: Rules are not valid

El webhook de validación (un chequeo que el API server consulta antes de guardar el objeto) que instala kube-prometheus-stack rechazó tu PrometheusRule: alguna regla no parsea. Así lo muestra Helm; kubectl apply muestra en cambio "prometheusrules" is invalid y debajo la regla que falló (group …, rule …: could not parse expression). El objeto no se creó ni cambió y siguen las reglas anteriores; el log del Operator repite el detalle (Invalid rule).

PRIMERO:

```
docker run --rm --entrypoint promtool -v "$PWD/monitoring":/etc/prometheus:ro prom/prometheus:v3.13.3 check rules /etc/prometheus/alerts.yml
```

.

```
kubectl logs -n monitoring deploy/kube-prometheus-stack-operator | grep 'Invalid rule'
```

→ Sección 11 · En Kubernetes: kube-prometheus-stack, el Operator, ServiceMonitor, PrometheusRule y KEDA

nv_inference_queue_duration_us metric names should not contain abbreviated units

promtool check metrics revisa los nombres contra las convenciones de Prometheus: la unidad va completa y en base (_seconds, no _us ni _ms). Para la misma métrica marca también counter metrics should have "_total" suffix: los counters terminan en _total. Triton no sigue ninguna de las dos y funciona igual; en tu código, síguelas: con _seconds nadie confunde µs con ms, y _total dice counter de un vistazo.

PRIMERO:

```
curl -s localhost:8002/metrics | docker run --rm -i --entrypoint promtool prom/prometheus:v3.13.3 check metrics
```

→ Sección 2 · El modelo de datos: nombre, labels, series y muestras; counter, gauge, histogram y summary; nombres y cardinalidad

EJEMPLO REAL · CÓMO LEER UN MANY-TO-MANY: LA FRACCIÓN DE 5XX POR POD

TEXTO — Forma exacta del mensaje en Prometheus 3.13 (promql/engine.go); las labels son las que agrega un ServiceMonitor del C01; el pod y la IP son ilustrativos

```
$ curl -s localhost:9090/api/v1/query --data-urlencode 'query=
    sum by (pod) (rate(classifier_requests_total{status=~"5.."}[5m]))
      / on (pod)
    rate(classifier_requests_total[5m])' | jq -r '.errorType, .error'
execution
found duplicate series for the match group {pod="gateway-5f6b9c7d48-7tq2m"} on the right hand-side of the operation: [{container="gateway", endpoint="http", instance="10.0.1.21:8080", job="gateway", model_version="2026-09-01-resnet50-fp16", namespace="classifier", pod="gateway-5f6b9c7d48-7tq2m", service="gateway", source="web", status="200"}, {container="gateway", endpoint="http", instance="10.0.1.21:8080", job="gateway", model_version="2026-09-01-resnet50-fp16", namespace="classifier", pod="gateway-5f6b9c7d48-7tq2m", service="gateway", source="mobile", status="200"}];many-to-many matching not allowed: matching labels must be unique on one side
```

- Léelo por partes: `match group {pod=…}` son las labels de tu `on (pod)`; `right hand-side` dice de qué lado están los repetidos; la lista muestra dos series de ese lado que solo difieren en `source`.
- La causa: el lado derecho no está agregado. Para cada pod hay una serie por cada combinación de status, source y model_version, y PromQL no sabe con cuál dividir.
- Arreglo: agrega los dos lados igual, `sum by (pod) (…) / sum by (pod) (…)`. `group_left` es para otra cosa: unir muchas series con una sola, por ejemplo para copiar labels de `kube_pod_info`.
- Sale con errorType `execution`: la query parseó bien y falló al calcularse. Un error de sintaxis habría sido `bad_data`.
- En una regla, el mismo error no se ve en la página de alertas: la regla queda con `health` `err` y el texto en `lastError` (Status → Rule health), y la alerta nunca dispara.

CÓMO RECORDARLA

quién habla dice dónde mirar. `Get "…"` en lastError: el scrape (red, puerto, path). `parse error` o `bad_data`: tu PromQL no corrió. `PromQL info` o `warning`: corrió, pero es sospechosa. Un log `level=ERROR`: el servidor. Un traceback: tu instrumentación.

TRAMPA

buscar el error donde no está. El de un target no sale en tu app ni en la query: está en Target health. El de una regla que no evalúa tampoco grita: queda en Rule health. Y algunos textos dependen de la versión: en Prometheus 2.x un Content-Type raro se toleraba; desde 3.0 el scrape falla con `received unsupported Content-Type`, y un `le="1"` se guarda como `le="1.0"`.

En el curso: M12 · Prometheus, Grafana y alertas por SLO · M12 · Métricas de sistema: golden signals con detalles de ML · C01 · Monitoreo, alertas y reentrenamiento · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Herramientas · Monitoreo: sistema y modelo

---

Funciones, campos, defaults, puertos y mensajes revisados en octubre de 2026 contra la documentación oficial de Prometheus, Alertmanager, el Prometheus Operator, Grafana, prometheus_client y KEDA, contra el capítulo «Alerting on SLOs» del SRE Workbook de Google y contra los archivos de `reference/c01` del curso. Las versiones son las del C01 (Prometheus 3.13.3, prometheus-client 0.26.0) y las de los laboratorios (Alertmanager 0.34.1, Grafana 13.2.4, kube-prometheus-stack 92.11.0, KEDA 2.21.0); cuando la versión actual cambia algo, la sección lo dice. De los 15 laboratorios, 6 se corrieron de verdad en un Mac; los demás se verificaron contra la documentación. Si tu terminal dice otra cosa, créele a tu terminal.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error