nyrvex.com

Kubernetes Observability

Hands-On

Prometheus, Alertmanager, Grafana, Thanos, Mimir, Loki, Tempo und S3 für die alten Daten



Mit wenig Text fit fürs Interview

Kubernetes Observability

Hands-On

Prometheus, Alertmanager, Grafana, Thanos, Mimir, Loki, Tempo und S3 für die alten Daten

Mit wenig Text fit fürs Interview

Thomas Zachmann

Kubernetes Observability Hands-On. Prometheus, Alertmanager, Grafana, Thanos, Mimir, Loki, Tempo und S3 für die alten Daten.
Stand: 4. November 2026. © 2026 Thomas Zachmann. Lizenz: CC BY-ND 4.0.

Geschrieben für Prometheus 3.15, Alertmanager 0.34, Prometheus Operator 0.94, Grafana 13.3, Thanos 0.42, Mimir 3.2, Loki 3.7, Tempo 3.1 und Alloy 1.21. Charts: kube-prometheus-stack 92.18, mimir-distributed 6.2, loki 20.4, tempo 3.2, tempo-distributed 3.22, alloy 1.15. Die Befehle, Konfigurationen und Beispielausgaben in diesem Buch wurden nicht auf einem Cluster ausgeführt. Ausgaben sind beispielhaft, und Details können sich je nach Version unterscheiden. Probiere alles zuerst im Labor aus, nie auf einem produktiven Cluster.

Schriften: Charter, Fira Sans, JetBrains Mono.

Inhalt

So liest du dieses Buch 5

Grundlagen **7**
1 Warum Observability? 8
2 Die Landkarte 10
3 Dein Labor 12
4 Labels und Kardinalität 16

Prometheus **19**
5 Das Prometheus-Modell 20
6 Metrik-Typen 22
7 PromQL 24
8 Die TSDB 27
9 Scrape und Relabeling 30
10 Der Prometheus Operator 32
11 Regeln 36
12 Kubernetes messen 39

Alerting **43**
13 Alertmanager 44
14 Gute Alarme 48

Grafana **51**
15 Grafana 52
16 Dashboards, die helfen 55

Langzeit-Metriken und S3 **57**
17 S3 für alte Daten 58
18 Thanos verstehen 62
19 Thanos betreiben 65
20 Mimir verstehen 70
21 Mimir betreiben 73
22 Thanos oder Mimir? 77

Logs **79**
23 Loki verstehen 80
24 LogQL 83
25 Alloy 85
26 Loki betreiben 88

Traces und Korrelation **93**
27 Tracing-Grundlagen 94
28 Tempo verstehen 97
29 Tempo betreiben 100
30 Signale verbinden 104

Produktion und Interview **107**
31 Meta-Monitoring 108
32 Kapazität und Kosten 110
33 Fehlersuche in Szenarien 112
34 Die Interviewfragen 115
35 Aufgaben 120
36 Spickzettel 122
Stichwörter 125

So liest du dieses Buch

Du hast bald ein Interview und musst Observability auf Kubernetes können. Nicht irgendwann, sondern bald. Dieses Buch ist genau dafür gebaut: wenig Text, viele kleine Beispiele, und zu jedem Thema die Frage, die dir wirklich gestellt wird.

Die Regeln

- **Tippe alles selbst.** Nichts kopieren. Deine Finger lernen mit.
- **Jeder Befehl einzeln.** Du tippst, du schaust, was passiert.
- **Nur im Labor.** Ein kind-Cluster reicht für fast alles, siehe Kapitel 3.
- **Verstehe zuerst das Warum.** Interviews prüfen selten YAML-Felder. Sie prüfen, ob du erklären kannst, warum ein Alarm nicht kam oder warum der Speicher explodiert.

> **MERKE**
>
> Die Befehle, Konfigurationen und Ausgaben in diesem Buch wurden nicht ausgeführt. Die Ausgaben sind gekürzte Beispiele. Deine Texte werden abweichen.

Was sich gerade ändert

Viele Anleitungen im Netz sind veraltet. Diese Punkte zeigen im Interview, dass du aktuell bist:

- **Promtail ist tot.** Seit dem 2. März 2026 ist es End of Life. Der Nachfolger heißt Grafana Alloy.
- **Mimir 3 bevorzugt Kafka.** Die Ingest-Storage-Architektur trennt Schreib- und Lesepfad.
- **Tempo 3 hat keine Ingester mehr.** Block-Builder und Live-Stores ersetzen sie. Im Microservices-Modus ist Kafka Pflicht.
- **Loki verliert einen Modus.** Simple Scalable fällt mit Loki 4.0 weg.
- **Native Histograms sind stabil,** seit Prometheus 3.8.
- **Charts sind umgezogen.** Die Charts für Grafana, Loki und Tempo liegen jetzt im Repo `grafana-community`.

Die Kästen

> **DAS BILD IM KOPF**
>
> *Ein Vergleich aus dem Alltag. Wenn du den behältst, kannst du das Thema herleiten.*

```
kubectl get servicemonitors -A
```

Eselsbrücke: Ein Befehl zum Tippen, mit Merkhilfe darunter.

> **STOLPERFALLE**
>
> Der Fehler, den fast alle machen. Und den Interviewer gern abfragen.

> **IM INTERVIEW**
>
> **Eine typische Frage?**
>
> Eine Antwort in zwei, drei Sätzen. So kurz darf sie sein.

Dein Lernplan

Du hast …	Dann lies …
1 Abend	Kapitel 1, 2, 4, 5, 7, 13, 17, 18, 20, 23, 28 und 30, dann Kapitel 34 (die Fragen) und 36 (Spickzettel).
2 Tage	Tag 1: Teil I, II und III. Tag 2: Teil V, die Kapitel 23 und 28, dann Kapitel 34.
1 Woche	Tag 1 und 2: Teil I bis III. Tag 3: Teil IV und V. Tag 4: Teil VI. Tag 5: Teil VII. Tag 6: die Aufgaben aus Kapitel 35. Tag 7: Fragen wiederholen.

TEIL I

Grundlagen

Signale, die Landkarte, dein Labor und das wichtigste Wort: Kardinalität.

1

Warum Observability?

Monitoring sagt dir, dass etwas kaputt ist. Observability hilft dir herauszufinden, warum.

Der Unterschied

Monitoring beantwortet Fragen, die du vorher gestellt hast: Ist die CPU über 90 Prozent? Läuft der Pod? **Observability** erlaubt Fragen, an die du vorher nicht gedacht hast: Warum sind nur Bestellungen aus Frankreich über Zahlungsanbieter B langsam, und das nur seit dem Deployment um 14 Uhr?

> **DAS BILD IM KOPF**
>
> *Monitoring ist die Warnleuchte im Auto. Observability ist die Werkstatt, die den Fehlerspeicher ausliest, den Motor abhört und den Weg des Problems zurückverfolgt.*

Die drei Signale

Signal	Frage	Form	Werkzeug
Metriken	Wie viel? Wie schnell? Seit wann?	Zahlen über die Zeit, billig	Prometheus
Logs	Was genau ist passiert?	Text-Ereignisse, teuer in Masse	Loki
Traces	Wo ging die Zeit verloren?	Weg einer Anfrage durch Dienste	Tempo

Eine typische Suche läuft in dieser Reihenfolge:

1. Ein **Alarm** aus Metriken sagt: Fehlerrate zu hoch.
2. Ein **Trace** zeigt: Der Dienst `payment` ist die Ursache.
3. Die **Logs** von `payment` zeigen: Das Zertifikat ist abgelaufen.

Ein viertes Signal wächst gerade: **Profile**. Sie zeigen, welche Funktion die CPU verbraucht. In der Grafana-Welt macht das Pyroscope. Dieses Buch lässt es weg.

MERKE

Die Stärke entsteht erst durch die Verbindung. Gleiche Labels wie `namespace` und `pod` in allen Signalen und die Trace-ID in den Logs machen aus drei Werkzeugen ein System.

Vier Methoden, die du nennen können musst

Methode	Bestandteile	Für wen
Golden Signals	Latenz, Traffic, Fehler, Sättigung	Jeder Dienst
RED	Rate, Errors, Duration	Dienste, von außen gesehen
USE	Utilization, Saturation, Errors	Ressourcen wie CPU und Platte
SLI/SLO	Messgröße und Zielwert	Das, was Nutzer spüren

IM INTERVIEW

Was ist der Unterschied zwischen Monitoring und Observability?

Monitoring prüft bekannte Fehlerbilder mit festen Schwellwerten. Observability bedeutet, dass ich das Innere eines Systems aus seinen Signalen verstehen kann, auch bei Fehlern, die ich nicht vorhergesehen habe. Dafür brauche ich Metriken, Logs und Traces, die miteinander verknüpft sind.

IM INTERVIEW

Welches Signal nimmst du wofür?

Metriken für Alarme und Trends, weil sie billig sind. Traces, um den schuldigen Dienst zu finden. Logs für das konkrete Detail. Ich gehe von grob nach fein.

DAS NIMMST DU MIT

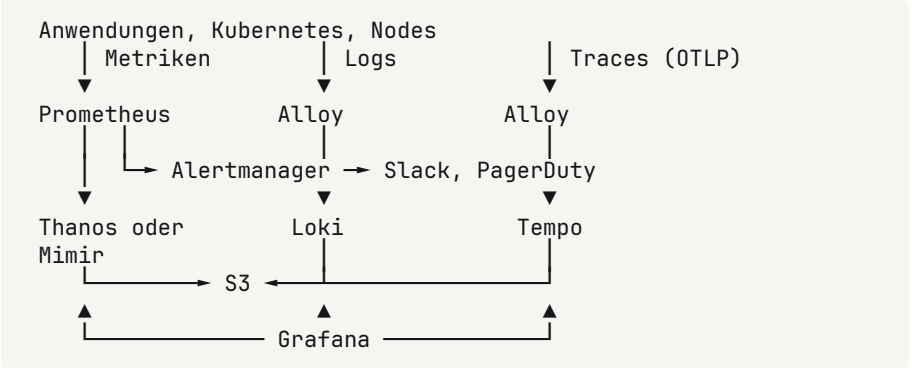
- Metriken, Logs, Traces: wie viel, was, wo.
- RED für Dienste, USE für Ressourcen.
- Der Wert liegt in der Verknüpfung.

2

Die Landkarte

Acht Namen, eine einfache Idee: sammeln, speichern, zeigen, alarmieren.

Der Stack auf einen Blick



Wer macht was?

Werkzeug	Aufgabe in einem Satz
Prometheus	Holt Metriken ab, speichert sie kurz, wertet Regeln aus.
Alertmanager	Bündelt Alarme, unterdrückt Lärm, schickt Nachrichten.
Grafana	Zeigt alles an und verbindet die Signale. Speichert selbst keine Messdaten.
Thanos	Hängt sich an Prometheus und legt Blöcke für lange Zeit in S3 ab.
Mimir	Eigene, mandantenfähige Metrik-Datenbank mit Remote Write und S3.
Loki	Speichert Logs günstig, indiziert nur die Labels.
Tempo	Speichert Traces günstig, fast nur in S3.
Alloy	Sammler für alle Signale. Grafanas Variante des OpenTelemetry Collectors.
S3	Billiger, fast unbegrenzter Objektspeicher für alte Daten.

LGTM steht für Loki, Grafana, Tempo, Mimir. Das Muster ist bei allen gleich:

> **DAS BILD IM KOPF**
>
> *Eine Kneipe mit kleiner Theke und großem Keller. Vorne (RAM, lokale Platte) liegt, was gerade gebraucht wird. Alles Ältere wandert ordentlich beschriftet in den Keller (S3). Der Keller ist riesig und billig, aber der Weg dorthin dauert länger.*

Thanos und Mimir sind Alternativen

Das verwechseln viele. Du brauchst für Langzeit-Metriken **eines** von beiden, nicht beide:

- **Thanos:** Prometheus bleibt das Zentrum, die Blöcke wandern in S3.
- **Mimir:** Prometheus liefert nur noch per Remote Write ab. Mimir ist das neue Zentrum.

Kapitel 22 hilft bei der Wahl.

IM INTERVIEW

Beschreibe einen Observability-Stack für Kubernetes.

Metriken holt Prometheus über den Operator, Alarme gehen an den Alertmanager. Logs und Traces sammelt Alloy pro Node und schickt sie an Loki und Tempo. Langzeit-Metriken liegen über Thanos oder Mimir in S3, wo auch Loki und Tempo speichern. Grafana zeigt alles an und verlinkt die Signale.

DAS NIMMST DU MIT

- Muster überall gleich: heiße Daten lokal, alte Daten in S3.
- Thanos oder Mimir, nicht beides.
- Grafana zeigt nur, Daten liegen woanders.

3

Dein Labor

Ein kind-Cluster, ein Chart und ein kleiner S3-Speicher.

Was du brauchst

`docker`, `kind`, `kubectl`, `helm` und mindestens 8 GB RAM für Docker.

```
kind create cluster --name obs
```

Eselsbrücke: kind heißt Kubernetes in Docker. Ein Cluster namens obs.

```
kubectl create namespace monitoring
```

Eselsbrücke: Hier wohnt alles, was beobachtet.

kube-prometheus-stack

Dieses Chart installiert den Prometheus Operator, Prometheus, Alertmanager, Grafana, kube-state-metrics, node-exporter sowie fertige Regeln und Dashboards.

values-kps.yaml

```yaml
grafana:
  adminPassword: labor-only
prometheus:
  prometheusSpec:
    retention: 2d
    serviceMonitorSelectorNilUsesHelmValues: false
    podMonitorSelectorNilUsesHelmValues: false
    ruleSelectorNilUsesHelmValues: false
```

Die drei `NilUsesHelmValues`-Zeilen erklärt Kapitel 10. Sie sparen dir viel Ärger.

```
helm install kps oci://ghcr.io/prometheus-community/charts/
kube-prometheus-stack -n monitoring --version 92.18.0 -f
values-kps.yaml
```

Eselsbrücke: kps ist der Release-Name. Daraus entstehen alle Objektnamen.

```
kubectl -n monitoring get pods
```

Eselsbrücke: Warte, bis alles läuft.

```
NAME                                            READY   STATUS
alertmanager-kps-kube-prometheus-alertmanager-0  2/2   Running
kps-grafana-6d9f7c8b5-x2k4z                     3/3     Running
kps-kube-prometheus-operator-5b7c9d-qwe12       1/1     Running
kps-kube-state-metrics-7f6d5c-abc34             1/1     Running
kps-prometheus-node-exporter-m8t2k              1/1     Running
prometheus-kps-kube-prometheus-prometheus-0   2/2   Running
```

Die Oberflächen öffnen

```
kubectl -n monitoring port-forward svc/kps-kube-prometheus-
prometheus 9090
```

Eselsbrücke: Prometheus: localhost:9090.

```
kubectl -n monitoring port-forward svc/kps-kube-prometheus-
alertmanager 9093
```

Eselsbrücke: Alertmanager: localhost:9093.

```
kubectl -n monitoring port-forward svc/kps-grafana 3000:80
```

Eselsbrücke: Grafana: localhost:3000, Nutzer admin.

> **STOLPERFALLE**
>
> Auf kind melden Alarme oft `etcd`, `kube-scheduler` oder `kube-controller-manager` als nicht erreichbar. Sie lauschen dort nur auf localhost. Im Labor ist das normal.

S3 im Labor

Für Teil V bis VII brauchst du einen S3-kompatiblen Speicher. Das Projekt MinIO ist dafür kaum noch geeignet: Die freien Images sind aus den öffentlichen Registries verschwunden, sodass das Mimir-Chart inzwischen ein Fork-Image nutzt. Im Labor nehmen wir SeaweedFS.

s3-labor.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: s3
  namespace: monitoring
spec:
  selector:
    matchLabels: { app: s3 }
  template:
    metadata:
      labels: { app: s3 }
    spec:
      containers:
        - name: seaweedfs
          image: chrislusf/seaweedfs:latest
          args: ["server", "-dir=/data", "-s3"]
          ports:
            - containerPort: 8333
          volumeMounts:
            - { name: data, mountPath: /data }
      volumes:
        - name: data
          emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: s3
  namespace: monitoring
spec:
  selector: { app: s3 }
  ports:
    - port: 8333
```

Dann die Buckets anlegen:

```
kubectl -n monitoring run aws --rm -it --image=amazon/aws-
cli --env=AWS_ACCESS_KEY_ID=x --env=AWS_SECRET_ACCESS_KEY=x
--env=AWS_DEFAULT_REGION=us-east-1 --command -- sh
```

Eselsbrücke: Ein Wegwerf-Pod mit AWS-CLI.

```
for b in thanos loki tempo; do
  aws --endpoint-url http://s3:8333 s3 mb s3://$b
done
```

STOLPERFALLE

Das ist nur ein Labor: `latest`-Image, ohne Zugriffskontrolle, Daten in `emptyDir`. Nach einem Neustart ist alles weg.

DAS NIMMST DU MIT

- `kube-prometheus-stack` liefert Metriken und Alarme fertig aus.
- Release-Name `kps` bestimmt die Objektnamen.
- Für später brauchst du S3, im Labor reicht SeaweedFS.

4

Labels und Kardinalität

Das wichtigste Kapitel: Fast jedes Kostenproblem ist ein Label-Problem.

Eine Zeitreihe

Eine Zeitreihe ist ein Metrikname plus eine Kombination aus Labels:

```
http_requests_total{method="GET",  status="200", pod="api-1"}
http_requests_total{method="GET",  status="500", pod="api-1"}
http_requests_total{method="POST", status="200", pod="api-1"}
```

Das sind drei Zeitreihen, obwohl es nur eine Metrik ist. **Kardinalität** ist die Anzahl der Zeitreihen.

Die Multiplikation

```
  20 Endpunkte
×  5 Methoden
× 10 Statuscodes
× 50 Pods
= 50.000 Zeitreihen, im schlimmsten Fall
```

Kommt ein Label `user_id` mit 100.000 Werten dazu, wird daraus eine Zahl, die jeden Prometheus umbringt.

> **DAS BILD IM KOPF**
>
> *Ein Label ist wie eine weitere Spalte im Bestellformular. Größe mal Farbe ergibt ein paar Varianten. Schreibst du aber die Kundennummer dazu, hat plötzlich jeder Kunde seine eigene Schublade im Lager.*

Gute und schlechte Labels

Gut: wenige, feste Werte	Schlecht: unbegrenzte Werte
`namespace`, `service`, `method`, `status`	`user_id`, `session_id`, `email`
`region`, `cluster`, `env`	`trace_id`, `request_id`
`route="/users/:id"`	`path="/users/48213"`

> **MERKE**
>
> Diese Regel gilt für alle Werkzeuge: für Prometheus-Labels, für Loki-Stream-Labels und für Tempo-Dimensionen in abgeleiteten Metriken. Hohe Kardinalität gehört in Logs und Traces, aber nicht in Labels.

Churn: die versteckte Kardinalität

Jeder neue Pod hat einen neuen Namen, also neue Zeitreihen. Die alten stehen noch bis zur Komprimierung im Head-Block. Wer zehnmal am Tag ausrollt, hat auch ohne böse Labels viele Zeitreihen. Das nennt man **Series Churn**.

Finden, was teuer ist

In Prometheus unter **Status → TSDB Status** oder mit PromQL:

```
# Die zehn Metriken mit den meisten Zeitreihen
topk(10, count by (__name__) ({__name__=~".+"}))

# Welches Label treibt diese Metrik?
count(count by (pod) (http_requests_total))

# Wie viele Zeitreihen hat jeder Job?
sum by (job) (scrape_series_added)
```

> **STOLPERFALLE**
>
> Die erste Abfrage ist selbst teuer. Führe sie auf einem großen System nur mit Bedacht aus, oder nutze die TSDB-Statusseite.

Reparieren

1. Im Code das Label entfernen oder normalisieren. Das ist die beste Lösung.
2. Beim Abholen wegwerfen, mit `metric_relabel_configs` (Kapitel 9).
3. Beim Schreiben begrenzen, mit `sample_limit` oder den Limits in Mimir.

> **IM INTERVIEW**
>
> **Was ist Kardinalität, und warum ist sie ein Problem?**
>
> Kardinalität ist die Zahl der einzelnen Zeitreihen, also jeder Kombination aus Metrik und Labelwerten. Prometheus hält aktive Zeitreihen im RAM, deshalb kostet jede neue Reihe Speicher, und Abfragen werden langsamer. Unbegrenzte Labels wie Nutzer-IDs führen zum OOM-Kill.

IM INTERVIEW

Prometheus braucht plötzlich doppelt so viel RAM. Was tust du?

Ich schaue in die TSDB-Status-Seite und vergleiche die Top-Metriken und Labels. Mit `scrape_series_added` finde ich den Job. Danach korrigiere ich die Anwendung oder werfe die Metrik per `metric_relabel_configs` weg und setze ein `sample_limit` als Schutz.

DAS NIMMST DU MIT

- Zeitreihen sind Multiplikation.
- Keine unbegrenzten Werte in Labels.
- Churn zählt mit: Neue Pods sind neue Reihen.

TEIL II

Prometheus

Pull-Modell, PromQL, Speicher, Operator und Regeln.

5

Das Prometheus-Modell

Prometheus holt ab. Niemand muss liefern.

Pull statt Push

Anwendungen zeigen ihre Zahlen unter `/metrics` als Text. Prometheus ruft diese Adresse alle paar Sekunden ab. Das nennt man **Scrape**.

```
# HELP http_requests_total Anzahl der Anfragen
# TYPE http_requests_total counter
http_requests_total{method="GET",status="200"} 1027
http_requests_total{method="GET",status="500"} 3
```

> **DAS BILD IM KOPF**
>
> *Der Stromzähler im Keller. Er zählt einfach weiter. Der Ableser kommt regelmäßig und notiert den Stand. Kommt er nicht, weiß er sofort, dass etwas nicht stimmt.*

Pull hat zwei Vorteile: Prometheus erkennt sofort, wenn ein Ziel fehlt, und eine langsame Anwendung kann Prometheus nicht überfluten.

Zu jedem Ziel erzeugt Prometheus automatisch die Metrik `up`. Sie ist 1, wenn der Scrape klappte, und sonst 0.

Service Discovery

In Kubernetes ändern sich Ziele ständig. Deshalb fragt Prometheus die Kubernetes-API, welche Pods und Endpoints es gibt. Niemand pflegt IP-Adressen von Hand.

Exporter

Software ohne `/metrics` bekommt einen Übersetzer:

Exporter	Liefert
node-exporter	CPU, RAM, Platte, Netz eines Nodes.
kube-state-metrics	Zustand von Kubernetes-Objekten.
cAdvisor (im Kubelet)	CPU und RAM pro Container.
blackbox-exporter	Prüft von außen: HTTP, TCP, DNS, ICMP.
postgres-exporter usw.	Datenbanken und andere Software.

Sonderfälle

- **Pushgateway:** Für kurze Batch-Jobs, die vor dem Scrape verschwinden. Nur dafür.
- **Agent-Modus:** Mit `--agent` sammelt Prometheus nur und leitet per Remote Write weiter. Abfragen und Regeln entfallen. Das ist ideal für Edge-Cluster.
- **Remote Write:** Prometheus schickt Daten weiter, etwa an Mimir.
- **OTLP:** Prometheus nimmt mit `--web.enable-otlp-receiver` auch OpenTelemetry-Metriken an.

> **STOLPERFALLE**
>
> Das Pushgateway vergisst nichts. Ein einmal gesendeter Wert bleibt stehen, auch wenn der Job seit Tagen tot ist. `up` sagt dir dort nur etwas über das Gateway selbst.

IM INTERVIEW

Warum Pull und nicht Push?

Mit Pull weiß Prometheus, welche Ziele existieren sollen, und erkennt Ausfälle über `up`. Die Last steuert Prometheus selbst. Für kurzlebige Jobs oder Netze hinter einer Firewall gibt es Pushgateway, Agent-Modus und Remote Write.

DAS NIMMST DU MIT

- `/metrics` ist Text, Prometheus holt ihn ab.
- `up` ist die erste Metrik, die du prüfst.
- Pushgateway nur für Batch-Jobs.

6

Metrik-Typen

Vier Typen, und einer davon ist die häufigste Interviewfrage.

Die vier Typen

Typ	Verhalten	Beispiel
Counter	Steigt nur. Fängt nach Neustart bei 0 an.	`http_requests_total`
Gauge	Geht hoch und runter.	`node_memory_MemAvailable_bytes`
Histogram	Zählt Werte in Eimern. Serverseitige Quantile.	`http_request_duration_seconds`
Summary	Rechnet Quantile in der Anwendung.	`go_gc_duration_seconds`

> **MERKE**
>
> Einen Counter zeigst du nie roh an. Du nimmst immer `rate()`. Der rohe Wert hängt nur davon ab, wann der Pod zuletzt neu gestartet wurde.

Histogram gegen Summary

Ein Histogram besteht aus mehreren Zeitreihen:

```
http_request_duration_seconds_bucket{le="0.1"}  240
http_request_duration_seconds_bucket{le="0.5"}  310
http_request_duration_seconds_bucket{le="+Inf"} 320
http_request_duration_seconds_sum   88.2
http_request_duration_seconds_count 320
```

`le` heißt **less or equal**. Die Eimer sind kumulativ: Der Eimer 0.5 enthält auch alles aus 0.1.

	Histogram	Summary
Quantil berechnet	Beim Abfragen	In der Anwendung
Über Pods zusammenfassbar	Ja	Nein
Genauigkeit	Hängt von den Eimern ab	Fest vorgegeben
Empfehlung	Standard	Nur in Sonderfällen

> **DAS BILD IM KOPF**
>
> *Summary ist, als hätte jede Filiale schon ihren eigenen Durchschnitt berechnet. Aus zehn Durchschnitten bekommst du keinen echten Gesamtwert. Histogram liefert die Strichlisten, die du beliebig zusammenzählen kannst.*

Native Histograms

Klassische Histogramme erzeugen pro Eimer eine Zeitreihe, und du musst die Grenzen vorher festlegen. **Native Histograms** speichern alle Eimer in einem einzigen Wert und wählen die Grenzen automatisch. Das spart Zeitreihen und ist genauer.

Sie sind seit Prometheus 3.8 stabil. Das Abholen musst du trotzdem ausdrücklich einschalten:

```
global:
  scrape_native_histograms: true
```

> **STOLPERFALLE**
>
> Das alte Flag `--enable-feature=native-histograms` tut seit Version 3.9 nichts mehr. Erst mit Prometheus 4 soll das Abholen automatisch aktiv sein.

> **IM INTERVIEW**
>
> **Histogram oder Summary?**
>
> Histogram, denn nur das kann ich über mehrere Pods zusammenrechnen. Ein Summary-Quantil lässt sich nicht sinnvoll mitteln. Heute nehme ich möglichst Native Histograms, weil sie weniger Zeitreihen brauchen und keine festen Grenzen.

DAS NIMMST DU MIT

- Counter nur mit `rate()`.
- Histogram lässt sich aggregieren, Summary nicht.
- Native Histograms sind stabil, aber Opt-in.

7

PromQL

Die wenigen Muster, die in fast jeder Abfrage stecken.

Auswählen

```
http_requests_total                          # Instant Vector
http_requests_total{status=~"5..", job="api"}  # Regex
http_requests_total[5m]                      # Range Vector
http_requests_total offset 1h                # vor einer Stunde
```

Ein **Instant Vector** liefert pro Zeitreihe einen Wert. Ein **Range Vector** liefert pro Zeitreihe alle Werte eines Zeitfensters. Letzteren kannst du nicht direkt zeichnen, sondern nur in Funktionen wie `rate()` stecken.

rate und increase

```
rate(http_requests_total[5m])       # pro Sekunde
increase(http_requests_total[1h])   # Zuwachs in der Stunde
irate(http_requests_total[5m])      # nur die letzten zwei Werte
```

`rate()` erkennt Counter-Resets selbst. Das Fenster sollte mindestens das Vierfache des Scrape-Intervalls sein. Bei 30 Sekunden Intervall heißt das: mindestens `[2m]`.

> **STOLPERFALLE**
>
> Zuerst `rate`, dann `sum`. Falsch: `rate(sum(x)[5m:])`. Richtig: `sum(rate(x[5m]))`. Summierst du zuerst, sieht `rate` nach einem Pod-Neustart einen Abfall und deutet ihn falsch.

Aggregieren

```
sum by (namespace) (rate(http_requests_total[5m]))
sum without (pod, instance) (rate(http_requests_total[5m]))
topk(5, sum by (pod)
(rate(container_cpu_usage_seconds_total[5m])))
```

Die wichtigsten Rezepte

Fehlerquote:

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

p99-Latenz:

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

Das `le` muss in der Aggregation bleiben, sonst geht es nicht. Bei Native Histograms entfällt sowohl `_bucket` als auch `le`.

Durchschnittliche Latenz:

```
rate(http_request_duration_seconds_sum[5m])
  /
rate(http_request_duration_seconds_count[5m])
```

Platte in vier Stunden voll?

```
predict_linear(node_filesystem_avail_bytes[6h], 4 * 3600) < 0
```

Ein Job fehlt ganz:

```
absent(up{job="api"})
```

Verknüpfen zweier Metriken

Bei Rechnungen zwischen Metriken müssen die Labels passen. Sonst hilft `on` und `group_left`:

```
sum by (pod, namespace)
(rate(container_cpu_usage_seconds_total[5m]))
  * on (pod, namespace) group_left (node)
kube_pod_info
```

> **DAS BILD IM KOPF**
>
> *`group_left` ist wie ein SVERWEIS in Excel: Viele Zeilen links holen sich eine Spalte aus einer kleinen Tabelle rechts.*

STOLPERFALLE

`rate()` ist ein Durchschnitt über das Fenster. Spitzen unterhalb des Fensters verschwinden. Und `histogram_quantile` schätzt nur: Das Ergebnis kann nie genauer sein als deine Eimer.

IM INTERVIEW

Warum liefert `rate` manchmal keine Daten?

`rate` braucht mindestens zwei Werte im Fenster. Ist das Fenster kürzer als zweimal das Scrape-Intervall oder fehlte ein Scrape, bleibt es leer. Deshalb nehme ich mindestens das Vierfache des Intervalls, in Grafana am besten `$__rate_interval`.

IM INTERVIEW

Was ist der Unterschied zwischen `rate` und `irate`?

`rate` mittelt über das ganze Fenster und eignet sich für Alarme und Trends. `irate` schaut nur auf die letzten zwei Werte. Das zeigt Spitzen, flackert aber. Für Alarme nehme ich `rate`.

DAS NIMMST DU MIT

- Erst `rate`, dann `sum`.
- `by (le)` nicht vergessen.
- Fenster mindestens vierfaches Scrape-Intervall.

8

Die TSDB

Warum Prometheus so effizient ist und wo es stirbt.

Der Weg eines Messwerts

```
Scrape ──► WAL (Platte, zum Wiederherstellen)
       └─► Head Block (RAM, aktuelle 2 Stunden)
              │ alle 2 Stunden
              ▼
           Block auf Platte (unveränderlich)
              │ Compaction
              ▼
           größere Blöcke ──► Retention löscht alte
```

Teil	Aufgabe
WAL	Write-Ahead Log: Schützt den Head vor Datenverlust bei einem Absturz.
Head Block	Die jüngsten Daten im Speicher, sofort abfragbar.
Block	Ordner mit Chunks, Index und `meta.json`. Standardlänge zwei Stunden.
Compaction	Fasst kleine Blöcke zu größeren zusammen.
Tombstones	Löschmarken. Blöcke selbst ändern sich nicht.

> **DAS BILD IM KOPF**
>
> *Ein Kassenbuch: Jede Buchung steht erst im Notizbuch (WAL), dann auf dem Schreibtisch (Head). Alle zwei Stunden wird ein fertiger Ordner ins Regal gestellt (Block). Einmal im Regal, wird nichts mehr geändert.*

Genau diese unveränderlichen Blöcke machen Thanos und Mimir möglich. Man kann sie einfach nach S3 kopieren.

Rechnen

Prometheus braucht laut Doku rund 1 bis 2 Byte pro Sample.

```
1.000.000 Zeitreihen, alle 30 Sekunden
= 33.333 Samples pro Sekunde
```

```
= 2,88 Milliarden Samples pro Tag
× 1,5 Byte = etwa 4,3 GB pro Tag
× 15 Tage  = etwa 65 GB
```

RAM ist der Engpass: Jede aktive Zeitreihe kostet einige Kilobyte im Head.

Aufbewahrung

Prometheus hält standardmäßig 15 Tage. Das Chart `kube-prometheus-stack` setzt 10 Tage.

```
--storage.tsdb.retention.time=15d
--storage.tsdb.retention.size=50GB
```

Es gilt, was zuerst eintritt.

Hochverfügbarkeit

Prometheus hat keine eingebaute Replikation. Die Lösung ist schlicht: Zwei identische Instanzen scrapen unabhängig voneinander dieselben Ziele. Grafana oder Thanos zeigen eine davon. Die Werte unterscheiden sich minimal, weil beide zu anderen Zeitpunkten scrapen.

> **STOLPERFALLE**
>
> Nach einem OOM spielt Prometheus beim Start das WAL ein. Das braucht oft noch mehr RAM. Der Pod stirbt erneut, und so entsteht eine `CrashLoop`. Die Ursache ist fast immer Kardinalität. Vermeide kurzfristige Lösungen, die das WAL löschen, solange du den Datenverlust nicht akzeptierst.

> **STOLPERFALLE**
>
> NFS ist für die TSDB nicht geeignet. Die Doku empfiehlt lokalen, POSIX-konformen Speicher. Auf Kubernetes nimmst du ein Blockgerät über ein PVC.

IM INTERVIEW

Wie speichert Prometheus Daten?

Neue Samples landen im WAL und im Head im RAM. Alle zwei Stunden entsteht ein unveränderlicher Block auf der Platte. Die Compaction fasst Blöcke zusammen, und die Retention löscht alte. Für lange Aufbewahrung und mehrere Cluster ergänze ich Thanos oder Mimir, weil die lokale TSDB nicht geclustert ist.

IM INTERVIEW

Wie machst du Prometheus hochverfügbar?

Zwei Replikate mit identischer Konfiguration und unterschiedlichem Replica-Label. Beide scrapen alles und schicken Alarme an den Alertmanager, der dedupliziert. Thanos Query oder der HA-Tracker von Mimir entfernen doppelte Daten.

DAS NIMMST DU MIT

- WAL, Head, Blöcke, Compaction, Retention.
- Blöcke sind unveränderlich, darum passen sie nach S3.
- HA heißt: zwei Prometheus, die dasselbe tun.

9

Scrape und Relabeling

Zwei Schritte, die zu den am häufigsten verwechselten gehören.

Eine rohe Scrape-Konfiguration

```yaml
global:
  scrape_interval: 30s
  evaluation_interval: 30s
  external_labels:
    cluster: prod-eu

scrape_configs:
  - job_name: api
    kubernetes_sd_configs:
      - role: pod
    sample_limit: 50000
    relabel_configs:
      - source_labels: [__meta_kubernetes_namespace]
        target_label: namespace
```

Der Unterschied

	`relabel_configs`	`metric_relabel_configs`
Wann	Vor dem Scrape	Nach dem Scrape, vor dem Speichern
Arbeitet auf	Zielen	Einzelnen Zeitreihen
Zweck	Ziele wählen, Labels setzen	Teure Metriken wegwerfen

> **DAS BILD IM KOPF**
>
> `relabel_configs` *ist die Gästeliste an der Tür.* `metric_relabel_configs` *ist der Türsteher am Buffet, der entscheidet, was auf den Teller darf.*

Aktionen

Aktion	Wirkung
`keep`	Behält nur Passendes.
`drop`	Wirft Passendes weg.
`replace`	Setzt ein Label, oft aus einem anderen.
`labeldrop`	Entfernt ein Label von der Reihe.
`labelmap`	Benennt Labels per Regex um.
`hashmod`	Verteilt Ziele auf mehrere Scraper.

Eine teure Metrik wegwerfen

```yaml
metric_relabel_configs:
  - source_labels: [__name__]
    regex: "apiserver_request_duration_seconds_bucket"
    action: drop
  - regex: "pod_ip"
    action: labeldrop
```

STOLPERFALLE

Vorsicht mit `labeldrop`: Bleiben danach zwei Reihen mit identischen Labels übrig, kollidieren sie. Prometheus meldet dann doppelte Samples.

MERKE

Labels mit `__` am Anfang sind intern und verschwinden nach dem Relabeling. Darum musst du `__meta_kubernetes_namespace` in `namespace` kopieren, wenn du es behalten willst.

IM INTERVIEW

Was ist der Unterschied zwischen `relabel_configs` und `metric_relabel_configs`?

`relabel_configs` läuft vor dem Scrape und entscheidet, welche Ziele abgeholt werden und welche Labels sie bekommen. `metric_relabel_configs` läuft danach auf jeder Zeitreihe. Damit werfe ich teure Metriken weg. Sie kosten aber trotzdem Netz und Parsing.

DAS NIMMST DU MIT

- Vor dem Scrape: Ziele. Nach dem Scrape: Reihen.
- `sample_limit` ist deine Sicherung.
- `__meta_`-Labels musst du retten.

10

Der Prometheus Operator

Prometheus als Kubernetes-Objekte, und die berühmteste Falle.

Die CRDs

CRD	Bedeutung
`Prometheus`	Eine Prometheus-Instanz.
`PrometheusAgent`	Prometheus im Agent-Modus.
`Alertmanager`	Eine Alertmanager-Instanz.
`ServiceMonitor`	Scrape-Ziele über einen Service.
`PodMonitor`	Scrape-Ziele direkt über Pods.
`Probe`	Blackbox-Prüfungen.
`ScrapeConfig`	Freie Scrape-Konfiguration, auch für Ziele außerhalb des Clusters.
`PrometheusRule`	Alarm- und Recording-Regeln.
`AlertmanagerConfig`	Routing pro Namespace.
`ThanosRuler`	Thanos Ruler als Objekt.

Ein Dienst wird gescrapt

`app.yaml`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api
  namespace: shop
  labels:
    app: api
spec:
  selector:
    app: api
  ports:
    - name: metrics
      port: 8080
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
```

```yaml
metadata:
  name: api
  namespace: shop
  labels:
    release: kps
spec:
  selector:
    matchLabels:
      app: api
  namespaceSelector:
    matchNames: [shop]
  endpoints:
    - port: metrics
      path: /metrics
      interval: 30s
```

DAS BILD IM KOPF

Eine Kette aus drei Etiketten: Prometheus sucht ServiceMonitors mit `release: kps`. Der ServiceMonitor sucht Services mit `app: api`. Der Service sucht Pods mit `app: api`. Reißt ein Glied, ist das Ziel unsichtbar und niemand beschwert sich.

STOLPERFALLE

Das ist die berühmteste Falle: `port: metrics` meint den **Namen** des Service-Ports, nicht eine Nummer. Und ohne das passende Label `release: kps` ignoriert Prometheus den ServiceMonitor still.

Warum das Label nötig ist

Ohne weitere Angaben sucht das Chart nur ServiceMonitors mit dem Release-Label. Mit dieser Zeile nimmt Prometheus alle:

```yaml
prometheus:
  prometheusSpec:
    serviceMonitorSelectorNilUsesHelmValues: false
```

Das Gleiche gilt für `podMonitorSelector…`, `ruleSelector…`, `probeSelector…` und `scrapeConfigSelector…`.

Prüfen

```
kubectl -n monitoring get prometheus -o jsonpath='{.items[0].spec.serviceMonitorSelector}'
```

Eselsbrücke: Wonach sucht Prometheus?

```
kubectl -n shop get servicemonitor api -o yaml
```

Eselsbrücke: Hat er das richtige Label?

```
kubectl -n shop get endpoints api
```

Eselsbrücke: Hat der Service überhaupt Pods?

Danach öffne **Status → Targets** in Prometheus. Steht das Ziel nicht in der Liste, liegt es am Label. Steht es dort als `down`, liegt es am Netz oder an der Anwendung.

Rechte

Prometheus braucht RBAC für den Ziel-Namespace. Das Chart gibt ihm clusterweite Leserechte. Bei selbst gebauten Instanzen kommt der Fehler `403 forbidden` im Log häufig vor.

IM INTERVIEW

Ein ServiceMonitor wird nicht aufgenommen. Wie gehst du vor?

Ich prüfe die Kette: Passt das Label zum `serviceMonitorSelector`, und passt der Namespace zum `serviceMonitorNamespaceSelector`? Passt der Selector des ServiceMonitors zum Service, und trägt der Port den richtigen Namen? Hat der Service Endpoints? Zum Schluss schaue ich in die Target-Seite und ins Operator-Log.

IM INTERVIEW

ServiceMonitor oder PodMonitor?

ServiceMonitor, wenn es einen Service gibt, das ist der Normalfall. PodMonitor für Workloads ohne Service, etwa DaemonSets oder Jobs. Für Ziele außerhalb des Clusters nehme ich `ScrapeConfig`.

DAS NIMMST DU MIT

- Selektor-Kette: Prometheus, ServiceMonitor, Service, Pod.
- `port` ist ein Name.
- Fehlt das Target ganz: Labels prüfen.

11

Regeln

Recording Rules machen schnell, Alerting Rules machen wach.

Recording Rules

Sie rechnen teure Abfragen regelmäßig vor und speichern das Ergebnis als neue Metrik.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: api-rules
  namespace: shop
  labels:
    release: kps
spec:
  groups:
    - name: api.rules
      interval: 30s
      rules:
        - record: job:http_requests:rate5m
          expr: sum by (job) (rate(http_requests_total[5m]))
```

Die Konvention für Namen lautet `ebene:metrik:operation`.

Alerting Rules

```yaml
    - name: api.alerts
      rules:
        - alert: HighErrorRate
          expr: |
            sum(rate(http_requests_total{status=~"5.."}[5m]))
              / sum(rate(http_requests_total[5m])) > 0.05
          for: 10m
          keep_firing_for: 5m
          labels:
            severity: critical
            team: shop
          annotations:
            summary: "Fehlerquote über 5 Prozent"
            description: "Aktuell {{ $value |
humanizePercentage }}"
```

```
            runbook_url: "https://wiki.example.com/high-error-
rate"
```

Der Lebenslauf eines Alarms

```
inactive → pending (Bedingung wahr, for läuft)
         → firing  (for abgelaufen, geht an Alertmanager)
         → resolved
```

Feld	Bedeutung
`for`	So lange muss die Bedingung gelten. Filtert kurze Ausreißer.
`keep_firing_for`	So lange bleibt der Alarm aktiv, nachdem die Bedingung endet. Verhindert Flackern.
`labels`	Für das Routing. Ändern die Identität des Alarms.
`annotations`	Für Menschen. Text, Links, Werte.

STOLPERFALLE

Schreibe keine sich ändernden Werte in `labels`. Jeder neue Wert erzeugt einen neuen Alarm. Die aktuelle Zahl gehört in `annotations`.

MERKE

Startet Prometheus neu, stellt es laufende `for`-Zeiten aus der Reihe `ALERTS_FOR_STATE` wieder her. Nach einem langen Ausfall gilt das nur begrenzt, standardmäßig bis zu einer Stunde.

Testen

```
promtool check rules regeln.yaml
```

Eselsbrücke: Syntax prüfen.

```
promtool test rules test.yaml
```

Eselsbrücke: Regeln mit erfundenen Daten testen.

`test.yaml`

```
rule_files: [regeln.yaml]
evaluation_interval: 1m
tests:
  - interval: 1m
    input_series:
```

```
      - series: 'up{job="api"}'
        values: "0x15"
    alert_rule_test:
      - eval_time: 12m
        alertname: ApiDown
        exp_alerts:
          - exp_labels: { severity: critical, job: api }
```

IM INTERVIEW

Wozu dienen Recording Rules?

Sie berechnen teure Abfragen im Voraus. Dashboards werden schneller, und Alarme werden einfacher. Außerdem kann ich bei langer Speicherung die vorberechneten Reihen abfragen, statt jedes Mal Millionen Samples zu lesen.

IM INTERVIEW

Was bewirkt `for` in einer Regel?

Der Alarm geht erst nach dieser Zeit auf `firing`. Das filtert kurze Ausreißer, verzögert aber auch die Meldung. Bei echten Totalausfällen wähle ich es kurz, bei langsamen Trends länger.

DAS NIMMST DU MIT

- Recording Rules sparen Rechenzeit.
- `labels` routen, `annotations` erklären.
- Regeln mit `promtool test rules` testen.

12

Kubernetes messen

Woher die Zahlen kommen und welche du kennen musst.

Vier Quellen

Quelle	Liefert	Präfix
kube-state-metrics	Wie die API die Objekte sieht: gewünschter und aktueller Zustand.	`kube_`
cAdvisor	Wie viel die Container wirklich verbrauchen.	`container_`
node-exporter	Wie es dem Host geht.	`node_`
Kubelet, API-Server	Wie es den Komponenten selbst geht.	`kubelet_`, `apiserver_`

> **STOLPERFALLE**
>
> `metrics-server` ist etwas anderes. Er hält nur aktuelle Werte im RAM und bedient `kubectl top` und den HPA. Für Verlauf und Alarme taugt er nicht. Prometheus ersetzt ihn auch nicht automatisch: Für den HPA mit eigenen Metriken brauchst du `prometheus-adapter` oder KEDA.

Die wichtigsten Abfragen

CPU pro Pod:

```
sum by (namespace, pod) (
  rate(container_cpu_usage_seconds_total{container!=""}[5m]))
```

Speicher pro Pod:

```
sum by (namespace, pod) (
  container_memory_working_set_bytes{container!=""})
```

`container_memory_working_set_bytes` ist der richtige Wert: Danach entscheidet der Kubelet über Evictions, und er liegt nah am OOM-Killer.

CPU-Drosselung:

```
sum by (pod)
(rate(container_cpu_cfs_throttled_periods_total[5m]))
  /
sum by (pod) (rate(container_cpu_cfs_periods_total[5m]))
```

Pods mit Neustarts:

```
increase(kube_pod_container_status_restarts_total[1h]) > 3
```

OOM-Kills:

```
kube_pod_container_status_last_terminated_reason{reason="OOMKilled"}
```

Fehlende Replikate:

```
kube_deployment_spec_replicas
  - kube_deployment_status_replicas_available > 0
```

Requests gegen echten Verbrauch:

```
sum by (namespace)
(kube_pod_container_resource_requests{resource="cpu"})
  -
sum by (namespace)
(rate(container_cpu_usage_seconds_total{container!=""}[5m]))
```

> **MERKE**
>
> Das Filter `container!=""` entfernt die Summenzeile des Pods aus cAdvisor. Ohne es zählst du alles doppelt.

IM INTERVIEW

Wie erkennst du, dass ein Pod CPU-gedrosselt wird?

Der CPU-Graph sieht ruhig aus, weil das Limit ihn kappt, die Latenz steigt aber. Ich schaue auf das Verhältnis von `container_cpu_cfs_throttled_periods_total` zu `container_cpu_cfs_periods_total`. Ist es hoch, erhöhe ich das Limit oder entferne es.

IM INTERVIEW

Welche Quelle nimmst du für was?

Für den Objektzustand kube-state-metrics, für den Verbrauch cAdvisor, für den Host node-exporter. `metrics-server` dient nur dem Autoscaling und `kubectl top`.

DAS NIMMST DU MIT

- `kube_` Zustand, `container_` Verbrauch, `node_` Host.
- Working Set ist der richtige Speicherwert.
- `container!=""` verhindert Doppelzählung.

TEIL III

Alerting

Alertmanager, gute Alarme, SLOs und Error Budgets.

13

Alertmanager

Prometheus entscheidet, dass etwas brennt. Alertmanager entscheidet, wer es erfährt.

Was er tut

- **Gruppieren:** 200 Pods fallen aus, du bekommst eine Nachricht.
- **Unterdrücken (Inhibition):** Ist der Node tot, schweigen die Pod-Alarme.
- **Stummschalten (Silence):** Für Wartungen.
- **Routen:** Team A ins Slack, kritisch an PagerDuty.
- **Deduplizieren:** Zwei Prometheus melden dasselbe, du hörst es einmal.

> **DAS BILD IM KOPF**
>
> *Die Telefonzentrale eines Krankenhauses. Hundert Anrufe zum selben Unfall werden zu einem Einsatz zusammengefasst, an die richtige Station gestellt und während einer geplanten Übung gar nicht erst durchgestellt.*

Die Konfiguration

`alertmanager.yaml`

```yaml
route:
  receiver: default
  group_by: [alertname, namespace]
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
    - matchers:
        - severity = "critical"
      receiver: pager
    - matchers:
        - team = "shop"
      receiver: shop-slack
      continue: true
    - matchers:
        - alertname = "Watchdog"
      receiver: "null"
```

```yaml
receivers:
  - name: default
    slack_configs:
      - channel: "#alerts"
        api_url_file: /etc/alertmanager/secrets/slack/url
        send_resolved: true
  - name: pager
    pagerduty_configs:
      - routing_key_file: /etc/alertmanager/secrets/pd/key
  - name: shop-slack
    slack_configs:
      - channel: "#shop"
        api_url_file: /etc/alertmanager/secrets/slack/url
  - name: "null"

inhibit_rules:
  - source_matchers: [severity = "critical"]
    target_matchers: [severity = "warning"]
    equal: [namespace, alertname]
```

Die drei Zeiten

Feld	Standard	Bedeutung
`group_wait`	30 s	So lange sammeln, bevor die erste Nachricht einer neuen Gruppe rausgeht.
`group_interval`	5 min	So lange warten, bevor neue Alarme einer bestehenden Gruppe gemeldet werden.
`repeat_interval`	4 h	So lange warten, bevor unveränderte Alarme erneut gemeldet werden.

MERKE

Die Route-Baum-Regel: Die erste passende Route gewinnt, es sei denn, sie setzt `continue: true`. Die Wurzelroute fängt alles auf und darf keine Matcher haben.

STOLPERFALLE

Die alten Felder `match` und `match_re` sowie `source_match` und `target_match` sind abgekündigt. Nimm `matchers`, `source_matchers` und `target_matchers`. Alte Beispiele im Netz nutzen sie noch.

Hochverfügbarkeit

Alertmanager-Instanzen bilden per Gossip (Port 9094) einen Cluster. Prometheus sendet an **alle** Instanzen, und diese stimmen untereinander ab, wer sendet.

> **STOLPERFALLE**
>
> Stelle keinen Load Balancer zwischen Prometheus und Alertmanager. Prometheus muss alle Instanzen direkt kennen. Der Operator erledigt das über den Service und die Endpoints.

Per Namespace: AlertmanagerConfig

Mit dem Operator kann jedes Team sein Routing selbst schreiben:

```yaml
apiVersion: monitoring.coreos.com/v1alpha1
kind: AlertmanagerConfig
metadata:
  name: shop
  namespace: shop
spec:
  route:
    receiver: slack
    groupBy: [alertname]
  receivers:
    - name: slack
      slackConfigs:
        - channel: "#shop"
          apiURL:
            name: slack-secret
            key: url
```

Der Operator fügt automatisch einen Matcher auf den Namespace hinzu. So kann ein Team nur seine eigenen Alarme routen. Welche Objekte gelten, regelt `alertmanagerConfigSelector`.

Prüfen

```
amtool check-config alertmanager.yaml
```

Eselsbrücke: am wie Alertmanager, tool wie Werkzeug. Prüft die Datei.

```
amtool config routes test --config.file=alertmanager.yaml
severity=critical team=shop
```

Eselsbrücke: Wohin ginge dieser Alarm?

```
amtool silence add alertname=NodeDown --duration=2h --com-
ment="Wartung" --alertmanager.url=http://localhost:9093
```

Eselsbrücke: Zwei Stunden Ruhe, mit Begründung.

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

Eselsbrücke: Was feuert gerade?

IM INTERVIEW

Wie vermeidest du Alarmstürme?

Mit `group_by` fasse ich zusammen, zum Beispiel nach Alarmname und Namespace. Mit Inhibition schweigen Folgealarme, wenn die Ursache schon gemeldet ist. Dazu sinnvolle Zeiten und nur handlungsrelevante Alarme.

IM INTERVIEW

Ein Alarm feuert in Prometheus, kommt aber nicht in Slack an. Wie prüfst du das?

Erst schaue ich auf `/alerts` in Prometheus, ob er `firing` ist. Dann im Alertmanager, ob er dort ankommt und ob eine Silence oder Inhibition greift. Mit `amtool config routes test` prüfe ich die Route. Zum Schluss schaue ich auf `alertmanager_notifications_failed_total` und ins Log.

IM INTERVIEW

Warum kein Load Balancer vor dem Alertmanager?

Prometheus soll jede Instanz erreichen. Die Instanzen stimmen sich per Gossip ab und deduplizieren. Würde ein Load Balancer nur an eine Instanz senden, hinge ein Alarm an genau dieser einen Instanz.

DAS NIMMST DU MIT

- Gruppieren, unterdrücken, stummschalten, routen.
- Erste passende Route gewinnt.
- HA per Gossip, Prometheus sendet an alle.

14

Gute Alarme

Jeder Alarm muss eine Handlung auslösen. Sonst ist er Lärm.

Die Regeln

- Alarmiere auf **Symptome**, nicht auf Ursachen. Nutzer spüren Fehler, nicht CPU.
- Jeder Alarm hat ein **Runbook**.
- Jede Benachrichtigung verlangt eine Handlung.
- Was warten kann, ist ein Ticket und kein Pager.
- Wer zu oft geweckt wird, ignoriert Alarme. Das nennt man **Alert Fatigue**.

Schlecht: Ursache	Gut: Symptom
CPU über 90 Prozent	p99 über 500 ms seit 10 Minuten
Ein Pod ist neu gestartet	Fehlerquote über 1 Prozent
RAM über 80 Prozent	Warteschlange wächst seit 15 Minuten

SLI, SLO, Error Budget

Begriff	Bedeutung
SLI	Die Messgröße. Zum Beispiel: Anteil der Anfragen unter 300 ms.
SLO	Das Ziel. Zum Beispiel: 99,9 Prozent in 30 Tagen.
SLA	Der Vertrag. Bei Bruch gibt es Geld zurück.
Error Budget	Der erlaubte Rest: 0,1 Prozent, also 43,2 Minuten pro 30 Tage.

> **DAS BILD IM KOPF**
>
> *Das Error Budget ist ein Taschengeld. Solange noch etwas da ist, darfst du riskant ausrollen. Ist es aufgebraucht, wird gespart: keine neuen Features, erst Stabilität.*

Burn Rate

Die Burn Rate sagt, wie schnell du das Budget verbrauchst. 1 heißt: Es reicht genau bis zum Ende der 30 Tage. 14,4 heißt: In einer Stunde sind 2 Prozent des Budgets weg.

```
Fehlerquote / erlaubte Fehlerquote = Burn Rate
1,44 Prozent / 0,1 Prozent = 14,4
```

Das Google-SRE-Workbook empfiehlt Alarme über zwei Fenster:

Langes Fenster	Kurzes Fenster	Burn Rate	Budget verbraucht
1 h	5 min	14,4	2 Prozent
6 h	30 min	6	5 Prozent
3 Tage	6 h	1	10 Prozent

Das lange Fenster belegt, dass das Problem ernst ist. Das kurze sorgt dafür, dass der Alarm schnell endet, sobald es behoben ist.

```
- alert: ErrorBudgetBurnFast
  expr: |
    job:slo_errors:ratio_rate1h > (14.4 * 0.001)
    and
    job:slo_errors:ratio_rate5m > (14.4 * 0.001)
  labels:
    severity: critical
```

Werkzeuge wie **Sloth** oder **Pyrra** erzeugen solche Regeln aus einer kurzen SLO-Beschreibung.

Der Watchdog

Was passiert, wenn das Alarmsystem selbst tot ist? Dann bleibt alles still. `kube-prometheus-stack` bringt deshalb den Alarm `Watchdog` mit, der **immer** feuert. Du leitest ihn an einen externen Dienst weiter, der Alarm schlägt, sobald die Meldungen **ausbleiben**. Das nennt man **Dead Man's Switch**.

> **DAS BILD IM KOPF**
>
> *Der Totmannschalter in der Lok. Der Fahrer muss regelmäßig drücken. Bleibt das Signal aus, bremst der Zug.*

IM INTERVIEW

Wann soll ein Alarm einen Menschen wecken?

Nur wenn Nutzer betroffen sind oder bald betroffen sein werden und ein Mensch etwas tun kann. Alles andere wird zum Ticket oder Dashboard. Ich alarmiere auf Symptome und nutze die Ursachen für die Fehlersuche.

IM INTERVIEW

Was ist eine Multi-Window-Burn-Rate?

Ich vergleiche die Fehlerquote mit dem erlaubten Budget, über ein langes und ein kurzes Fenster. Beide müssen überschritten sein. So reagiert der Alarm schnell auf große Probleme, feuert nicht bei kurzen Spitzen und endet zügig nach der Lösung.

IM INTERVIEW

Wer überwacht das Monitoring?

Ein immer feuernder Watchdog-Alarm geht an einen externen Dead-Man's-Switch-Dienst. Bleibt er aus, werde ich benachrichtigt. Dazu gehören Alarme auf die Pipeline selbst, siehe Kapitel 31.

DAS NIMMST DU MIT

- Symptome statt Ursachen.
- Burn Rate über zwei Fenster.
- Watchdog nach außen führen.

TEIL IV

Grafana

Datenquellen, Dashboards und alles als Code.

15

Grafana

Das Fenster zu allen Daten, aber nicht der Speicher.

Was Grafana ist

Grafana speichert keine Messwerte. Es fragt Datenquellen ab und zeichnet das Ergebnis. Selbst speichert es nur Dashboards, Nutzer, Regeln und Einstellungen.

Betrieb	Hinweis
Ein Pod, SQLite	Standard. Genügt für das Labor.
Mehrere Pods	Braucht eine gemeinsame Datenbank, etwa PostgreSQL.
Dashboards	Gehören in Git, nicht nur in die Datenbank.

> **STOLPERFALLE**
>
> Das Chart `grafana/grafana` ist abgekündigt und wandert nach `grafana-community/grafana`. Alte Anleitungen funktionieren eine Weile noch, bekommen aber keine neuen Versionen mehr. Das Chart `kube-prometheus-stack` nutzt schon die Community-Quelle.

Datenquellen als Code

Klicke keine Datenquellen zusammen. In `kube-prometheus-stack` sammelt ein Sidecar ConfigMaps mit dem Label `grafana_datasource: "1"` ein:

`datasources.yaml`

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: obs-datasources
  namespace: monitoring
  labels:
    grafana_datasource: "1"
data:
  datasources.yaml: |
    apiVersion: 1
    datasources:
      - name: Loki
```

```
      type: loki
      uid: loki
      url: http://loki-gateway.loki.svc
      access: proxy
    - name: Tempo
      type: tempo
      uid: tempo
      url: http://tempo.tempo.svc:3200
      access: proxy
```

STOLPERFALLE

In Grafanas Provisioning-Dateien ist `$` ein Ersetzungszeichen für Umgebungsvariablen. Willst du ein echtes `$` schreiben, etwa in `$${__value.raw}`, verdopple es.

Die Oberfläche

- **Explore:** Freies Suchen ohne Dashboard. Beginne die Fehlersuche hier.
- **Dashboards:** Feste Ansichten.
- **Alerting:** Grafanas eigene Alarme. Sie sind eine Alternative zu Regeln in Prometheus.
- **Drilldown:** Klicken statt Abfragen schreiben, für Metriken, Logs und Traces.

Grafana Alerting oder Prometheus-Regeln?

Prometheus-Regeln	Grafana Alerting
Als Code, leicht in Git	Auch per Provisioning, aber schwerer
Laufen nah an den Daten	Hängen von Grafana und der Datenquelle ab
Nur Metriken	Über alle Datenquellen

Faustregel: Infrastruktur-Alarme bleiben in Prometheus und im Alertmanager. Grafana Alerting nimmst du für Datenquellen, die keine Regeln können.

IM INTERVIEW

Wie betreibst du Grafana hochverfügbar?

Mehrere Replikate hinter einem Service mit einer gemeinsamen Datenbank wie PostgreSQL statt SQLite. Dashboards und Datenquellen kommen per Provisioning aus Git, damit Grafana selbst ersetzbar bleibt. Bei Grafana Alerting müssen sich die Instanzen zusätzlich als Cluster finden.

DAS NIMMST DU MIT

- Grafana zeigt nur, Daten liegen anderswo.
- Datenquellen per ConfigMap oder Provisioning.
- Infrastruktur-Alarme bleiben in Prometheus.

16

Dashboards, die helfen

Weniger Kacheln, klare Geschichte, alles aus Git.

Der Aufbau

Ein gutes Dashboard beantwortet eine Frage von oben nach unten:

1. Oben: Geht es dem Dienst gut? Rate, Fehler, Latenz.
2. Mitte: Wer ist betroffen? Aufteilung nach Route und Pod.
3. Unten: Warum? CPU, RAM, Abhängigkeiten.

> **DAS BILD IM KOPF**
>
> *Eine Zeitung: Schlagzeile, Zusammenfassung, Details. Wer nach zehn Sekunden nicht weiß, ob alles in Ordnung ist, hat ein schlechtes Dashboard.*

Variablen

```
Name:  namespace
Query: label_values(kube_pod_info, namespace)

Name:  pod
Query: label_values(kube_pod_info{namespace="$namespace"}, pod)
```

In der Abfrage:

```
sum by (pod) (
  rate(http_requests_total{namespace="$namespace"}
[$__rate_interval]))
```

> **MERKE**
>
> Nimm `$__rate_interval` statt fester Fenster. Grafana rechnet aus Zoom und Scrape-Intervall ein passendes Fenster aus. Dafür muss das Scrape-Intervall in der Datenquelle stimmen.

Dashboards als Code

Der Sidecar macht aus jeder ConfigMap mit Label ein Dashboard:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-dashboard
  namespace: monitoring
  labels:
    grafana_dashboard: "1"
data:
  api.json: |
    { "title": "API", "panels": [] }
```

Weitere Wege:

- **Grafana Operator:** CRDs wie `GrafanaDashboard`.
- **Grafonnet:** Dashboards in Jsonnet generieren.
- **Terraform:** Der Grafana-Provider verwaltet Ordner und Rechte.

Schnell bleiben

- Teure Abfragen werden Recording Rules.
- Wenige Panels, kurze Standard-Zeiträume.
- Keine Regex über alle Reihen.
- Bei Thanos: Automatisches Downsampling nutzt die Auflösung des Zeitraums.

> **STOLPERFALLE**
>
> Ein Dashboard mit 40 Panels, das alle 5 Sekunden lädt und von 30 Leuten offen gehalten wird, kann einen Prometheus in die Knie zwingen. Ein Dashboard ist Last.

IM INTERVIEW

Wie hältst du Dashboards wartbar?

Dashboards liegen versioniert in Git und werden per ConfigMap oder Operator ausgerollt. Ich nutze Variablen statt Kopien pro Dienst und baue von Überblick nach Detail. Händische Änderungen sind nur ein Experiment, bis sie zurück in Git sind.

DAS NIMMST DU MIT

- Von oben nach unten: gut, wer, warum.
- `$__rate_interval` statt fester Fenster.
- Dashboards gehören in Git.

TEIL V

Langzeit-Metriken und S3

Warum Objektspeicher, Thanos, Mimir und die Entscheidung dazwischen.

17

S3 für alte Daten

Billig, riesig, langsam: der Keller des ganzen Stacks.

Warum S3?

Lokale Platten sind schnell, aber teuer, und jede Größenänderung ist Arbeit. Objektspeicher hat andere Stärken:

- Du musst keine Kapazität planen.
- Der Anbieter kümmert sich um Redundanz.
- Pods bleiben fast zustandslos und lassen sich leicht ersetzen.
- Alle Replikate sehen dieselben Daten.

Dafür ist er langsam und rechnet pro Anfrage ab. Darum haben alle Werkzeuge Caches.

Werkzeug	Legt in S3 ab	Schreibt wann
Thanos	TSDB-Blöcke	Sidecar: alle 2 Stunden
Mimir	TSDB-Blöcke pro Mandant	Ingester: alle 2 Stunden
Loki	Chunks und TSDB-Index	Bei voller Chunk-Größe oder Zeitablauf
Tempo	Parquet-Blöcke	Laufend, gesteuert durch die Block-Grenzen

> **MERKE**
>
> Auch Ceph spricht über das RADOS Gateway S3. Du brauchst also keine Cloud. Das Protokoll ist der Standard, nicht Amazon.

Regeln für Buckets

- **Pro Werkzeug eigene Buckets.** Mimir startet nicht, wenn Blocks, Ruler und Alertmanager denselben Bucket samt Präfix teilen.
- **Region passend zum Cluster.** Sonst zahlst du Übertragung und wartest länger.
- **Buckets selbst anlegen.** Mimir und Tempo legen sie nicht an.

- **Unvollständige Multipart-Uploads löschen.** Eine Lifecycle-Regel mit einem Tag genügt.

Wer löscht alte Daten?

Immer das Werkzeug, nicht S3. Das ist die wichtigste Regel in diesem Kapitel.

Werkzeug	Einstellung	Standard
Thanos	Compactor: `--retention.resolution-raw` usw.	`0d`, also für immer
Mimir	`compactor_blocks_retention_period`	nie löschen
Loki	`retention_period` und `retention_enabled`	nie löschen
Tempo	`compaction.block_retention`	336 h, also 14 Tage

> **STOLPERFALLE**
>
> Eine S3-Lifecycle-Regel, die einfach alles nach 30 Tagen löscht, zerstört die Werkzeuge. Index und Daten laufen auseinander, und Abfragen schlagen fehl. Loki erlaubt Lifecycle-Regeln nur als Rückfallnetz: begrenzt auf Chunk-Präfixe, länger als die Retention, und nie für Index-Objekte.

Speicherklassen

Klasse	Bewertung
Standard	Die sichere Wahl.
Intelligent-Tiering	Möglich, solange die Archiv-Stufen aus bleiben.
Standard-IA	Zugriffsgebühr und Mindestdauer. Compaction liest viel und löscht früh.
Glacier	Nein. Die Werkzeuge brauchen sofortigen Zugriff auf die Objekte.

> **DAS BILD IM KOPF**
>
> *Glacier ist ein Lager, das erst nach Stunden liefert. Deine Abfrage steht aber an der Theke und wartet Sekunden. Der Keller muss begehbar bleiben.*

Zugriff ohne Schlüssel

Lege keine Access Keys in Secrets, wenn es anders geht. Auf EKS nutzt du **IRSA** oder **EKS Pod Identity**: Der ServiceAccount bekommt eine IAM-Rolle.

```
serviceAccount:
  annotations:
```

```
    eks.amazonaws.com/role-arn: arn:aws:iam::123456789012:role/
thanos
```

Die Rolle braucht nur Zugriff auf den eigenen Bucket:

```
s3:ListBucket, s3:GetObject, s3:PutObject, s3:DeleteObject
```

Tempo braucht zusätzlich `s3:GetObjectTagging` und `s3:PutObjectTagging`.

Kosten

Du zahlst drei Dinge: gespeicherte Gigabytes, Anfragen und Datenübertragung. Bei vielen kleinen Objekten werden Anfragen schnell teurer als der Speicher. Compaction hilft, weil sie viele kleine Blöcke zu wenigen großen macht.

Zugriffe aus derselben Region brauchen ein VPC Gateway Endpoint, sonst laufen sie über ein NAT Gateway, und das kostet pro Gigabyte.

> **STOLPERFALLE**
>
> Versionierung im Bucket: Die Werkzeuge löschen ständig Objekte. Mit aktiver Versionierung bleiben alle Versionen liegen und du zahlst weiter. Falls du sie brauchst, setze eine Lifecycle-Regel für alte Versionen.

IM INTERVIEW

Warum Objektspeicher für Observability-Daten?

Er ist billig, praktisch unbegrenzt und redundant. Die Daten sind fast unveränderlich, passen also gut zu Objekten. So werden Pods nahezu zustandslos. Den Nachteil, die Latenz, fangen Caches und Compaction ab.

IM INTERVIEW

Darf ich alte Daten mit einer S3-Lifecycle-Regel löschen?

Nein, in der Regel nicht. Die Retention gehört ins Werkzeug, denn das kennt Index und Blöcke. Eine Lifecycle-Regel löscht blind und hinterlässt einen kaputten Index. Sinnvoll sind nur Regeln für abgebrochene Multipart-Uploads und alte Versionen.

IM INTERVIEW

Kann ich alte Daten nach Glacier verschieben?

Nicht, wenn ich sie weiter abfragen will. Die Werkzeuge brauchen sofortigen Zugriff. Für Compliance exportiere ich getrennt in ein eigenes Archiv und behandle das als separaten Prozess.

DAS NIMMST DU MIT

- Das Werkzeug löscht, nicht S3.
- Eigene Buckets, gleiche Region, IAM-Rolle.
- Kein Glacier für abfragbare Daten.

18

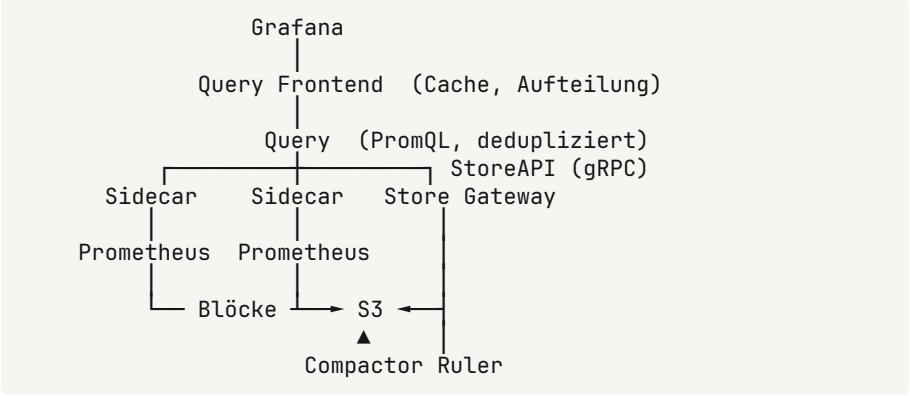
Thanos verstehen

Prometheus bleibt, und bekommt ein Gedächtnis.

Die Probleme, die Thanos löst

- **Lange Speicherung:** Blöcke liegen in S3.
- **Globale Sicht:** Eine Abfrage über alle Cluster.
- **HA ohne Doppelung:** Zwei Prometheus zeigen eine Kurve.

Die Bauteile



Bauteil	Aufgabe
Sidecar	Läuft neben Prometheus, lädt Blöcke hoch und liefert aktuelle Daten.
Store Gateway	Macht Blöcke aus S3 abfragbar.
Query	Zustandslos. Fragt alle StoreAPIs ab und dedupliziert.
Query Frontend	Cacht Ergebnisse und teilt lange Abfragen auf.
Compactor	Komprimiert, erzeugt Downsampling, wendet Retention an.
Ruler	Wertet Regeln über Query aus, also auch global.
Receive	Nimmt Remote Write an. Die Alternative zum Sidecar.

> **DAS BILD IM KOPF**
>
> *Query ist ein Bibliothekar mit einem Telefon. Er ruft die Filialen (Sidecars) und das Zentralarchiv (Store Gateway) an, legt die Ergebnisse zusammen und streicht Doppeltes.*

Sidecar oder Receive?

	Sidecar	Receive
Daten fließen	Abfrage zu Prometheus	Prometheus schickt per Remote Write
Netz	Query muss jeden Sidecar erreichen	Nur ausgehend vom Cluster
Betrieb	Einfach	Stateful, Hashring, Replikation
Gut für	Wenige Cluster, gleiches Netz	Viele Cluster, Edge, Mandanten

Receive verteilt Reihen über einen **Hashring**. Der empfohlene Algorithmus ist `ketama`. Mit `--receive.replication-factor=3` ist das Schreiben erfolgreich, sobald zwei Kopien stehen.

Externe Labels: Pflicht

Thanos unterscheidet Blöcke an ihren externen Labels. Jede Prometheus-Instanz braucht eine weltweit eindeutige Kombination.

```
global:
  external_labels:
    cluster: prod-eu
    prometheus_replica: prometheus-0
```

Mit dem Operator entstehen die Labels `prometheus` und `prometheus_replica` automatisch. Das Label `cluster` trägst du unter `externalLabels` ein.

Query ignoriert beim Deduplizieren das Replica-Label:

```
--query.replica-label=prometheus_replica
```

> **STOLPERFALLE**
>
> Fehlt das Label `cluster`, sehen die Blöcke aus zwei Clustern identisch aus. Dann kollidieren sie, und der Compactor kann die Daten durcheinanderbringen.

IM INTERVIEW

Wie funktioniert Thanos?

Ein Sidecar neben Prometheus lädt alle zwei Stunden fertige Blöcke nach S3. Das Store Gateway macht sie abfragbar. Thanos Query fragt Sidecars und Store Gateway über die StoreAPI ab und entfernt HA-Duplikate über das Replica-Label. Der Compactor komprimiert die Blöcke, erzeugt Downsampling und wendet die Retention an.

IM INTERVIEW

Wann Thanos Receive statt Sidecar?

Wenn die Cluster nicht direkt erreichbar sind oder sehr viele Cluster liefern. Dann schiebt Prometheus per Remote Write nach außen. Dafür betreibe ich einen stateful Hashring mit Replikation.

DAS NIMMST DU MIT

- Sidecar lädt hoch, Store Gateway liest, Query verbindet.
- Compactor pflegt den Bucket.
- Externe Labels müssen eindeutig sein.

19

Thanos betreiben

Sidecar einschalten, Compactor richtig einstellen, Downsampling verstehen.

Schritt 1: Bucket-Konfiguration

objstore.yml

```
type: S3
config:
  bucket: thanos
  endpoint: s3.eu-central-1.amazonaws.com
  region: eu-central-1
  aws_sdk_auth: true
```

Mit `aws_sdk_auth: true` kommen die Zugangsdaten aus der IAM-Rolle, nicht aus der Datei. Im Labor trägst du stattdessen das Labor-S3 ein:

```
type: S3
config:
  bucket: thanos
  endpoint: s3.monitoring.svc:8333
  access_key: x
  secret_key: x
  insecure: true
```

> **STOLPERFALLE**
>
> `access_key` aus der Datei und `secret_key` aus einer anderen Quelle zu mischen, ist nicht erlaubt. Und `insecure: true` schaltet TLS ab. Das ist nur für Tests.

```
kubectl -n monitoring create secret generic thanos-objstore
--from-file=objstore.yml
```

Eselsbrücke: Die Datei wandert in ein Secret.

Schritt 2: Sidecar einschalten

values-kps.yaml (Ergänzung)

```yaml
prometheus:
  thanosService:
    enabled: true
  prometheusSpec:
    replicas: 2
    retention: 3d
    externalLabels:
      cluster: labor
    thanos:
      objectStorageConfig:
        existingSecret:
          name: thanos-objstore
          key: objstore.yml
```

Der Operator setzt die Blockdauer auf genau zwei Stunden und schaltet die lokale Compaction ab, damit der Compactor in S3 allein zuständig ist.

```
helm upgrade kps oci://ghcr.io/prometheus-community/charts/kube-prometheus-stack -n monitoring --version 92.18.0 -f values-kps.yaml
```

Eselsbrücke: Gleicher Befehl, neue Werte.

> **STOLPERFALLE**
>
> Ändere nie die Blockdauern eines Prometheus mit Sidecar von Hand. Bei unterschiedlichen `min-block-duration` und `max-block-duration` komprimiert Prometheus lokal, und der Upload stockt.

Schritt 3: Query, Store, Compactor

Thanos hat kein offizielles Helm-Chart. Es gibt die Manifeste aus `kube-thanos`, das Chart von Bitnami oder eigene Deployments.

> **STOLPERFALLE**
>
> Bitnami hat sein kostenloses Image-Angebot 2025 stark eingeschränkt. Prüfe bei Anleitungen mit dem Bitnami-Chart, woher die Images wirklich kommen.

Der Kern eines Query-Deployments:

```yaml
containers:
  - name: query
    image: quay.io/thanos/thanos:v0.42.5
```

```
    args:
      - query
      - --http-address=0.0.0.0:9090
      - --grpc-address=0.0.0.0:10901
      - --query.replica-label=prometheus_replica
      - --query.auto-downsampling
      - --endpoint=dnssrv+_grpc._tcp.kps-kube-prometheus-thanos-
discovery.monitoring.svc.cluster.local
      - --endpoint=dnssrv+_grpc._tcp.thanos-
store.monitoring.svc.cluster.local
```

Store Gateway:

```
    args:
      - store
      - --data-dir=/data
      - --objstore.config-file=/etc/thanos/objstore.yml
      - --grpc-address=0.0.0.0:10901
```

Compactor:

```
    args:
      - compact
      - --wait
      - --data-dir=/data
      - --objstore.config-file=/etc/thanos/objstore.yml
      - --retention.resolution-raw=30d
      - --retention.resolution-5m=180d
      - --retention.resolution-1h=2y
```

> **MERKE**
>
> Ohne `--wait` beendet sich der Compactor nach einem Durchlauf. Als Deployment braucht er deshalb `--wait`, als CronJob nicht.

Downsampling

Wann	Was
Rohdaten älter als 40 Stunden	Es entstehen 5-Minuten-Blöcke.
5-Minuten-Blöcke älter als 10 Tage	Es entstehen 1-Stunden-Blöcke.

Die häufigste Fehleinschätzung lautet: Downsampling spart Speicher. Tut es nicht. Es kommen zwei weitere Blöcke **dazu**, und wer alle Auflösungen behält, braucht bis zu dreimal so viel Platz.

Der Sinn liegt in schnellen Abfragen über lange Zeiträume. Ein Jahres-Graph muss nicht Milliarden Rohwerte lesen.

> **DAS BILD IM KOPF**
>
> *Downsampling ist eine Landkarte in mehreren Maßstäben. Die grobe Karte ersetzt nicht die Stadtpläne, sie kommt dazu. Für die Fahrt nach Rom reicht aber die grobe.*

> **STOLPERFALLE**
>
> Setzt du `--retention.resolution-raw` kürzer als die 40 Stunden, entstehen nie 5-Minuten-Blöcke. Für einen Plan wie „30 Tage roh, 180 Tage grob„ müssen alle drei Werte aufeinander abgestimmt sein. Und `rate()` über ein Fenster kürzer als die Auflösung bleibt leer.

Singleton

Pro Blockstrom in einem Bucket darf nur **ein** Compactor laufen. Zwei gleichzeitig zerstören Daten. Skalieren kannst du nur über getrennte Blockströme, also per Label-Sharding.

Der Compactor braucht außerdem Platten. Die Doku nennt rund 100 GB für einen mittleren Bucket. Im schlimmsten Fall brauchst du Platz für zweimal den größten Compaction-Bereich.

> **STOLPERFALLE**
>
> Bei Datenfehlern hält der Compactor an, läuft aber weiter und meldet `Running`. Nur die Metrik `thanos_compact_halted` zeigt es. Und weil die Retention erst nach der Compaction läuft, wird dann nichts mehr gelöscht, und der Bucket wächst ohne Warnung.

Prüfen

```
thanos tools bucket inspect --objstore.config-file=objstore.yml
```

Eselsbrücke: Liste aller Blöcke im Bucket.

```
thanos tools bucket web --objstore.config-file=objstore.yml
```

Eselsbrücke: Blöcke grafisch ansehen.

IM INTERVIEW

Spart Downsampling Speicher?

Nein. Thanos legt zusätzliche Blöcke an, sodass der Speicher sogar wächst. Der Vorteil sind schnelle Abfragen über lange Zeiträume. Kosten spare ich erst über die Retention, also indem ich Rohdaten früher lösche als die groben Auflösungen.

IM INTERVIEW

Der Thanos-Bucket wächst, obwohl Retention gesetzt ist. Warum?

Vermutlich läuft der Compactor nicht mehr sauber. Ich prüfe `thanos_compact_halted` und das Log. Die Retention läuft erst nach Compaction und Downsampling. Hält der Compactor an, wird nichts gelöscht.

DAS NIMMST DU MIT

- Retention macht der Compactor, Standard ist unendlich.
- Downsampling beschleunigt, spart aber nichts.
- Nur ein Compactor pro Blockstrom. `thanos_compact_halted` überwachen.

20

Mimir verstehen

Ein zentrales Metrik-System für viele Cluster und Mandanten.

Die Idee

Prometheus sammelt weiter, liefert aber alles per Remote Write an Mimir. Mimir speichert, rechnet und beantwortet Abfragen. Es ist voll PromQL-kompatibel und mandantenfähig.

Zwei Architekturen

Seit Mimir 3.0 gibt es zwei Wege. Die **Ingest-Storage-Architektur** ist die bevorzugte und im Chart der Standard. Die **klassische** wird weiter unterstützt.

	Klassisch	Ingest Storage
Puffer	Keiner	Kafka
Schreiben erfolgreich	Quorum: 2 von 3 Ingestern	Kafka hat bestätigt
Replikation	RF 3 zwischen Ingestern	Pro Partition 2 Ingester
Lesen und Schreiben	Treffen sich im Ingester	Sind getrennt

Schreibpfad mit Ingest Storage

```
Prometheus → Distributor → Kafka → Ingester → S3
             prüft Limits   Partition  liest mit,
             wählt Partition           baut Blöcke
```

Eine Schreibanfrage gilt als erfolgreich, sobald Kafka sie bestätigt. Ein ausgefallener Ingester bremst das Schreiben also nicht. Nach dem Neustart holt er den Rückstand aus Kafka auf.

Lesepfad

```
Grafana → Query Frontend → Query Scheduler → Querier
                                              ├→ Ingester
(jung)
                                              └→ Store
Gateway (S3)
```

Komponente	Aufgabe
Distributor	Nimmt Remote Write an, prüft Limits, teilt Reihen auf.
Ingester	Hält junge Daten, schreibt alle 2 Stunden Blöcke.
Query Frontend	Teilt Abfragen nach Zeit und Shards, cacht.
Query Scheduler	Warteschlange, fair pro Mandant.
Querier	Führt die Abfragen aus.
Store Gateway	Liest Blöcke aus S3.
Compactor	Komprimiert, löscht nach Retention. Horizontal skalierbar.
Ruler	Wertet Regeln aus.
Alertmanager	Mandantenfähiger Alertmanager.

DAS BILD IM KOPF

Kafka ist ein Förderband in der Paketannahme. Die Annahme legt Pakete aufs Band und ist fertig. Die Sortierer greifen zu, wann sie können. Fällt einer aus, bleibt das Paket auf dem Band und geht nicht verloren.

Mandanten

Jede Anfrage trägt den Header `X-Scope-OrgID`. Pro Mandant entsteht eine eigene TSDB, mit eigenen Limits und eigener Retention.

STOLPERFALLE

Die Mandantenfähigkeit ersetzt keine Anmeldung. Mimir glaubt dem Header. Authentifizierung gehört in einen Reverse Proxy davor, der den Header setzt.

HA-Paare entdoppeln

Zwei Prometheus liefern dieselben Daten. Der **HA-Tracker** wählt pro Cluster einen Absender und verwirft den anderen. Er erkennt die Paare an den Labels `cluster` und `__replica__`. Eingeschaltet wird das pro Mandant über `accept_ha_samples`.

IM INTERVIEW

Warum nutzt Mimir Kafka?

Bei der klassischen Architektur bedient der Ingester Schreiben und Lesen zugleich. Viele Abfragen können also das Schreiben stören. Kafka trennt beide Wege als robuster Puffer. Der Distributor ist fertig, sobald Kafka bestätigt hat, und die Ingester lesen in eigenem Tempo.

IM INTERVIEW

Wie schützt Mimir sich vor einem lauten Mandanten?

Über Limits pro Mandant bei Ingestion, Zeitreihen und Abfragen, außerdem über die faire Warteschlange im Query Scheduler. Shuffle Sharding kann Mandanten zusätzlich auf Teilmengen der Pods begrenzen.

DAS NIMMST DU MIT

- Remote Write hinein, PromQL heraus, S3 darunter.
- Ingest Storage trennt Schreiben und Lesen mit Kafka.
- Mandant kommt aus `X-Scope-OrgID`, Authentifizierung ist deine Aufgabe.

21

Mimir betreiben

Chart, S3, Limits und Retention.

Modi

Mimir ist ein Binary. Der Parameter `-target` bestimmt, was läuft. `-target=all` startet alles in einem Prozess und ist der Einstieg. Die produktive Form sind Microservices. Den Read-Write-Modus hat Mimir 3 entfernt.

Installation

```
helm repo add grafana https://grafana.github.io/helm-charts
```

Eselsbrücke: Das Chart für Mimir liegt weiter im Haupt-Repo.

values-mimir.yaml

```yaml
mimir:
  structuredConfig:
    common:
      storage:
        backend: s3
        s3:
          endpoint: s3.eu-central-1.amazonaws.com
          region: eu-central-1
    blocks_storage:
      s3:
        bucket_name: firma-mimir-blocks
    alertmanager_storage:
      s3:
        bucket_name: firma-mimir-alertmanager
    ruler_storage:
      s3:
        bucket_name: firma-mimir-ruler
    limits:
      compactor_blocks_retention_period: 1y
minio:
  enabled: false
```

```
helm install mimir grafana/mimir-distributed -n mimir --
create-namespace --version 6.2.2 -f values-mimir.yaml
```

Eselsbrücke: Das Chart heißt distributed, denn es rollt Microservices aus.

> **STOLPERFALLE**
>
> Das Chart bringt ohne weitere Werte ein eigenes Kafka und einen S3-Ersatz nur für Demos mit. In Produktion brauchst du ein ernsthaftes Kafka und mindestens so viele Partitionen wie Ingester im Maximum. Der Topic-Standard von 100 Partitionen ist laut Chart nur eine Demo-Einstellung.

Prometheus anbinden

```yaml
prometheus:
  prometheusSpec:
    remoteWrite:
      - url: http://mimir-gateway.mimir.svc/api/v1/push
        headers:
          X-Scope-OrgID: team-a
```

Grafana bekommt eine Prometheus-Datenquelle mit dem Pfad `/prometheus`:

```yaml
- name: Mimir
  type: prometheus
  url: http://mimir-gateway.mimir.svc/prometheus
  jsonData:
    httpHeaderName1: X-Scope-OrgID
  secureJsonData:
    httpHeaderValue1: team-a
```

Die Service-Namen hängen vom Release ab. Prüfe sie mit `kubectl -n mimir get svc`.

Retention

Metriken in S3 werden standardmäßig nie gelöscht. Pro Mandant überschreibst du das in der Runtime- Konfiguration:

```yaml
overrides:
  team-a:
    compactor_blocks_retention_period: 2y
  team-b:
    compactor_blocks_retention_period: 90d
```

> **MERKE**
>
> Mimir löscht nur ganze Blöcke eines Mandanten. Löschen einzelner Reihen oder Retention pro Reihe unterstützt es nicht.

Mimir hat außerdem kein Downsampling wie Thanos. Es setzt auf Sharding und Caches. Wer einen Jahres-Graphen beschleunigen will, nutzt Recording Rules.

Die wichtigsten Limits

Limit	Standard	Wirkung
`ingestion_rate`	10.000/s	Samples pro Sekunde je Mandant.
`max_global_series_per_user`	150.000	Aktive Zeitreihen je Mandant.
`max_label_names_per_series`	30	Labels pro Reihe.

Das Limit für Zeitreihen ist für echte Cluster niedrig. Rechne damit, dass du es anhebst.

```
err-mimir-max-series-per-user: per-user series limit of 150000
exceeded
```

Die Metrik `cortex_discarded_samples_total` zeigt mit dem Label `reason`, warum Samples verworfen wurden.

Hinweise zum Abfragen

Mimir nutzt seit Version 3 standardmäßig seine eigene Mimir Query Engine (MQE). Sie ist sparsamer, sollte aber Ergebnisse liefern, die der Prometheus-Engine entsprechen. Bei Zweifeln stellst du `-querier.query-engine=prometheus` ein.

> **IM INTERVIEW**
>
> **Daten kommen nicht mehr in Mimir an. Woran kann das liegen?**
>
> Ich prüfe in Prometheus die Remote-Write-Metriken auf Rückstand und Fehler. 429-Antworten deuten auf Limits hin, dann zeigt `cortex_discarded_samples_total` den Grund. Bei 5xx schaue ich auf Distributoren und Kafka. Ohne Daten beim Lesen prüfe ich den Mandanten-Header.

IM INTERVIEW

Wie behandelt Prometheus Remote Write bei Ausfällen?

Prometheus puffert über das WAL und versucht es erneut. Dauert der Ausfall länger als der WAL-Puffer, gehen Daten verloren. Bei 429 und 5xx wiederholt es, bei anderen 4xx verwirft es die Daten.

DAS NIMMST DU MIT

- Chart: `grafana/mimir-distributed`, Kafka gehört zur Architektur.
- Retention pro Mandant, Standard unendlich.
- Limits sind niedrig, `cortex_discarded_samples_total` zeigt warum.

22

Thanos oder Mimir?

Die Frage, die in fast jedem Senior-Interview kommt.

Der Vergleich

	Thanos	Mimir
Ansatz	Erweitert Prometheus	Eigene zentrale Datenbank
Daten kommen	Sidecar oder Receive	Remote Write
Abfragen	Verteilt über StoreAPI	Zentral im Mimir-Cluster
Mandanten	Über Receive und Header, begrenzt	Eingebaut, mit Limits
Downsampling	Ja	Nein
Zusätzliche Abhängigkeiten	Wenig	Kafka, Caches
Betriebsaufwand	Niedrig bis mittel	Hoch
Lizenz	Apache 2.0	AGPL-3.0

Entscheidungshilfe

- **Wenige Cluster, kleines Team, Prometheus läuft schon:** Thanos mit Sidecar.
- **Viele Teams, Plattform als Produkt, Limits pro Team:** Mimir.
- **Sehr viele kurze Reihen, lange Aufbewahrung:** Eher Mimir mit Sharding.
- **Jahres-Trends für wenige Metriken:** Thanos mit Downsampling oder Recording Rules.

> **DAS BILD IM KOPF**
>
> *Thanos ist ein Netzwerk aus Filialen mit gemeinsamem Lager. Jede Filiale bleibt eigenständig. Mimir ist das Zentrallager: Alle liefern hin, und dort wird alles geregelt.*

MERKE

Die Antwort lautet nie „X ist besser„. Nenne Anforderungen: Anzahl Cluster, Mandanten, Datenmenge, Team-Größe, vorhandenes Kafka und ob die Netze getrennt sind.

IM INTERVIEW

Thanos oder Mimir?

Das hängt von den Anforderungen ab. Thanos passt, wenn ich auf vorhandene Prometheus-Server aufbauen und den Betrieb klein halten will. Mimir passt, wenn ich zentral eine mandantenfähige Plattform mit harten Limits anbiete und das Team Kafka und viele Komponenten tragen kann.

IM INTERVIEW

Warum nicht einfach die Retention von Prometheus auf zwei Jahre setzen?

Weil Prometheus ein einzelner Server bleibt: Eine Platte fällt aus, und die Daten sind weg. Es gibt keine globale Sicht und keine Duplikatentfernung. Abfragen über lange Zeiträume belasten denselben Prozess, der gerade scrapen soll.

DAS NIMMST DU MIT

- Thanos erweitert, Mimir ersetzt.
- Entscheide nach Teams, Clustern, Betriebslast.
- Downsampling hat nur Thanos, Mandanten kann Mimir besser.

TEIL VI

Logs

Loki, LogQL, Alloy und der Betrieb mit S3.

23

Loki verstehen

Prometheus-Denken für Logs: nur Labels indizieren, Inhalte nicht.

Die Grundidee

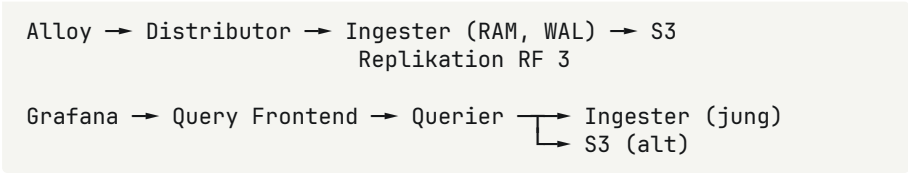
Elasticsearch indiziert jeden Begriff. Das ist mächtig, aber teuer. Loki indiziert nur die Labels und legt den Text komprimiert in S3 ab. Beim Suchen filtert Loki den Text erst zur Abfragezeit.

> **DAS BILD IM KOPF**
>
> *Ein Archiv mit beschrifteten Kartons. Die Beschriftung (Labels) ist im Verzeichnis. Den Inhalt der Kartons kennt niemand auswendig. Du wählst die richtigen Kartons und blätterst dann.*

Begriffe

Begriff	Bedeutung
Stream	Alle Logzeilen mit genau derselben Label-Kombination.
Chunk	Komprimiertes Paket aus Zeilen eines Streams, in S3.
Index	TSDB-Index: Welche Streams liegen in welchen Chunks?
Tenant	Mandant, wie bei Mimir über `X-Scope-OrgID`.

Schreib- und Lesepfad



Komponente	Aufgabe
Distributor	Prüft, begrenzt, verteilt Streams per Hash.
Ingester	Baut Chunks und schreibt sie weg.
Query Frontend	Teilt Abfragen auf, cacht, stellt in eine Warteschlange.
Querier	Liest und filtert.
Index Gateway	Bedient den Index für Abfragen.
Compactor	Komprimiert den Index, wendet Retention an. Eine Instanz.
Ruler	Regeln auf Logs, auch Alarme.

Labels: wieder die Kardinalität

Das ist in Loki noch wichtiger als in Prometheus. Jede Label-Kombination ist ein eigener Stream mit eigenen Chunks. Zu viele Streams führen zu winzigen Chunks und einem aufgeblähten Index.

Als Label	Nicht als Label
`namespace`, `app`, `container`	`trace_id`, `user_id`
`cluster`, `env`	`pod` bei sehr viel Churn
	`status`, `path`

Für wertvolle Felder ohne Label gibt es **Structured Metadata**. Sie werden pro Zeile gespeichert, ohne einen Stream zu erzeugen. Das ist der richtige Platz für `trace_id`.

> **STOLPERFALLE**
>
> Der klassische Fehler ist ein Label `level` plus `status` plus `path`. Die Stream-Zahl explodiert, und Abfragen werden **langsamer**, nicht schneller. Mit Filtern zur Abfragezeit fährst du meist besser.

IM INTERVIEW

Wie unterscheidet sich Loki von Elasticsearch?

Elasticsearch erstellt einen Volltextindex über den Inhalt. Das macht freie Suche schnell, kostet aber viel Speicher und Betrieb. Loki indiziert nur Labels und durchsucht den Text beim Abfragen. Das ist viel günstiger, aber bei Suchen über riesige Mengen ohne gute Labels langsamer.

IM INTERVIEW

Warum darf `trace_id` kein Label sein?

Sonst bekommt jede Anfrage einen eigenen Stream. Das zerstört Index und Chunk-Größe. Ich lege die Trace-ID in Structured Metadata oder filtere im Text danach.

DAS NIMMST DU MIT

- Labels sind der Index, Text wird zur Abfragezeit gelesen.
- Stream gleich Label-Kombination.
- Wenige Labels, Rest Structured Metadata.

24

LogQL

Erst den Karton wählen, dann filtern, dann rechnen.

Aufbau

```
{namespace="shop", app="api"} |= "error" != "healthz" | json |
status >= 500
└──────── Selektor ──────────┘ └── Zeilenfilter ──┘  └Parser┘
└Labelfilter┘
```

Mindestens ein Selektor mit nicht leerem Wert ist Pflicht.

Filtern

```
{app="api"} |= "timeout"          # enthält
{app="api"} != "healthz"          # enthält nicht
{app="api"} |~ "time.?out"        # Regex
{app="api"} !~ "GET /(health|ready)"
```

Parsen

```
{app="api"} | json
{app="api"} | logfmt
{app="api"} | pattern "<ip> - - <_> \"<method> <path> <_>\"
<status>"
{app="api"} | json | line_format "{{.status}} {{.path}}"
```

Metriken aus Logs

```
# Fehler pro Sekunde
sum by (app) (rate({namespace="shop"} |= "error" [5m]))

# Zeilen in einer Stunde
count_over_time({app="api"}[1h])

# p95 aus einem Feld im Log
quantile_over_time(0.95,
  {app="api"} | json | unwrap duration_ms [5m]) by (route)
```

Tempo gewinnen

1. Enge Selektoren. Sie bestimmen, wie viele Chunks Loki lesen muss.
2. Kurze Zeiträume.
3. Zeilenfilter vor Parser. `|=` ist viel günstiger als `| json`.

> **STOLPERFALLE**
>
> Ein Label aus `| json` entsteht erst beim Abfragen. Es macht die Suche nicht schneller, denn Loki muss vorher alle Zeilen parsen. Nur Stream-Labels verkleinern die Menge.

Kommandozeile

```
logcli query '{app="api"} |= "error"' --since=1h --limit=50
```

Eselsbrücke: Log command line interface.

```
logcli series '{namespace="shop"}' --analyze-labels
```

Eselsbrücke: Welche Labels erzeugen wie viele Streams?

IM INTERVIEW

Wie machst du eine langsame LogQL-Abfrage schneller?

Ich grenze zuerst über die Labels und die Zeit ein, weil das bestimmt, wie viele Daten gelesen werden. Dann setze ich Zeilenfilter vor den Parser. Für häufige Fragen nutze ich Recording Rules im Ruler.

DAS NIMMST DU MIT

- Selektor, Filter, Parser, Rechnung.
- Zeilenfilter zuerst, dann Parser.
- Geparste Labels beschleunigen nichts.

25

Alloy

Ein Sammler für alle Signale, und der Nachfolger von Promtail.

Warum Alloy?

Promtail hat am 2. März 2026 sein End of Life erreicht. Sein Chart ist abgekündigt. Alloy ist Grafanas Distribution des OpenTelemetry Collectors und sammelt Metriken, Logs, Traces und Profile.

```
alloy convert --source-format=promtail --output=config.alloy
promtail.yaml
```

Eselsbrücke: Altes Rezept rein, neues raus. Danach prüfen.

> **STOLPERFALLE**
>
> Alloy liefert andere Metriknamen als Promtail. Dashboards und Alarme mit dem Präfix `promtail_` laufen nach der Umstellung ins Leere. Prüfe sie vor der Migration.

Die Sprache

Alloy baut Pipelines aus Komponenten. Eine Komponente hat Eingänge und gibt Ergebnisse weiter:

```
discovery.kubernetes "pods" {
  role = "pod"
}

discovery.relabel "pods" {
  targets = discovery.kubernetes.pods.targets

  rule {
    source_labels = ["__meta_kubernetes_namespace"]
    target_label  = "namespace"
  }
  rule {
    source_labels = ["__meta_kubernetes_pod_label_app"]
    target_label  = "app"
  }
```

```
}

loki.source.kubernetes "pods" {
  targets    = discovery.relabel.pods.output
  forward_to = [loki.process.pods.receiver]
}

loki.process "pods" {
  stage.json {
    expressions = { trace_id = "trace_id" }
  }
  stage.structured_metadata {
    values = { trace_id = "" }
  }
  forward_to = [loki.write.default.receiver]
}

loki.write "default" {
  endpoint {
    url = "http://loki-gateway.loki.svc/loki/api/v1/push"
  }
}
```

DAS BILD IM KOPF

Lego mit Steckverbindern. `forward_to` *steckt den Ausgang des einen Bausteins in den Eingang des nächsten. Die Reihenfolge in der Datei ist egal, nur die Verbindungen zählen.*

Wie Logs abholen?

Komponente	Quelle	Bewertung
`loki.source.kubernetes`	Kubernetes-API	Einfach, ohne Host-Rechte, aber mehr Netz und Kubelet-CPU.
`loki.source.file`	`/var/log/pods` auf dem Node	Effizient, braucht DaemonSet und HostPath.

Bei großen Clustern nimmst du den DaemonSet mit Dateien. Dafür setzt du im Chart zusätzlich `alloy.mounts.varlog: true`.

Installation

values-alloy.yaml

```yaml
alloy:
  configMap:
    create: true
    content: |
      // die Konfiguration von oben
controller:
  type: daemonset
```

```
helm install alloy grafana/alloy -n monitoring --version
1.15.0 -f values-alloy.yaml
```

Eselsbrücke: DaemonSet: ein Sammler pro Node.

Wer alles fertig will, nimmt das Chart `grafana/k8s-monitoring`. Es rollt Alloy-Instanzen für alle Signale aus, verdeckt aber mehr.

Debuggen

Alloy hat eine Weboberfläche auf Port 12345. Dort siehst du den Graphen der Komponenten und ihren Zustand.

```
kubectl -n monitoring port-forward ds/alloy 12345
```

Eselsbrücke: UI öffnen und Komponenten-Graph ansehen.

IM INTERVIEW

Womit sammelst du heute Logs in Kubernetes?

Mit Grafana Alloy als DaemonSet, weil Promtail seit März 2026 End of Life ist. Alloy liest die Dateien unter `/var/log/pods`, ergänzt Kubernetes-Labels und schickt sie an Loki. Eine bestehende Promtail-Konfiguration lässt sich mit `alloy convert` übertragen.

DAS NIMMST DU MIT

- Promtail ist tot, Alloy übernimmt.
- `forward_to` verbindet Komponenten.
- Große Cluster: Dateien lesen per DaemonSet.

26

Loki betreiben

Modus, S3, Retention und die typischen Fehlermeldungen.

Die Modi

Modus	Menge	Bewertung
Monolithic	bis etwa 20 GB/Tag	Einstieg und kleine Stacks.
Simple Scalable	bis etwa 1 TB/Tag	Abgekündigt. Fällt mit Loki 4.0 weg.
Microservices	darüber	Empfehlung für Produktion mit HA.

> **STOLPERFALLE**
>
> Viele Anleitungen empfehlen noch Simple Scalable. Für neue Installationen ist das eine Sackgasse. Bestehende Installationen planen die Migration zu Microservices oder hochverfügbarem Monolithic.

Installation

Das Chart liegt im Community-Repo:

```
helm repo add grafana-community https://grafana-community.github.io/helm-charts
```

Eselsbrücke: Neues Zuhause der Charts für Grafana, Loki und Tempo.

values-loki.yaml

```yaml
deploymentMode: Monolithic
singleBinary:
  replicas: 1
loki:
  commonConfig:
    replication_factor: 1
  schemaConfig:
    configs:
      - from: 2026-01-01
        store: tsdb
        object_store: s3
        schema: v13
        index:
```

```
        prefix: index_
        period: 24h
  storage:
    type: s3
    bucketNames:
      chunks: loki
      ruler: loki
    s3:
      endpoint: http://s3.monitoring.svc:8333
      region: us-east-1
      accessKeyId: x
      secretAccessKey: x
      s3ForcePathStyle: true
      insecure: true
  compactor:
    retention_enabled: true
    delete_request_store: s3
  limits_config:
    retention_period: 720h
```

```
helm install loki grafana-community/loki -n loki --create-
namespace --version 20.4.1 -f values-loki.yaml
```

Eselsbrücke: Community-Repo, kein Grafana-Repo.

> **MERKE**
>
> `schemaConfig` ist im Chart leer und muss gesetzt werden. Für neue Installationen liegt `from` in der Vergangenheit. Beim Hinzufügen zu einer laufenden Installation muss `from` in der Zukunft liegen, gerechnet in UTC. Empfohlen sind `store: tsdb` und `schema: v13`.

Retention

Ohne `retention_enabled: true` bleiben Logs für immer in S3. Der Compactor löscht, und `index.period` muss dafür 24 Stunden betragen.

```
limits_config:
  retention_period: 744h
  retention_stream:
    - selector: '{namespace="dev"}'
      priority: 1
      period: 24h
```

Die höchste Priorität gewinnt. Bei Gleichstand gewinnt die kürzere Frist. Pro Stream sind mindestens 24 Stunden erlaubt, und `selector` kennt nur Label-Matcher.

STOLPERFALLE

Eine geänderte Retention gilt laut Doku nicht rückwirkend für bereits aufgenommene Logs. Plane sie deshalb von Anfang an. Der Compactor läuft als einzelne Instanz, und seine Löschmarken brauchen eine dauerhafte Platte oder ein Präfix im Bucket.

MERKE

Der Table Manager ist abgekündigt. Für TSDB nimmst du immer den Compactor.

Standard-Limits

Limit	Standard
`ingestion_rate_mb`	4 MB/s je Mandant
`ingestion_burst_size_mb`	6 MB
`max_global_streams_per_user`	5.000

Typische Fehler

Meldung	Bedeutung
`429 ... rate limit`	`ingestion_rate_mb` erreicht. Limit erhöhen oder weniger senden.
`stream limit exceeded`	Zu viele Streams. Labels prüfen.
`entry too far behind`	Zu alter Zeitstempel. `reject_old_samples` greift.
`max entry size`	Zeile zu lang.

Die Metrik `loki_discarded_samples_total` zeigt mit dem Label `reason`, was verworfen wurde.

IM INTERVIEW

Wie löscht Loki alte Logs?

Der Compactor wendet die Retention an. Dafür brauche ich `retention_enabled: true`, einen `delete_request_store` und eine `retention_period`, global, pro Mandant oder pro Stream. Er markiert Chunks und löscht sie nach `retention_delete_delay`. Ohne diese Einstellung bleibt alles für immer in S3.

IM INTERVIEW

Welchen Loki-Modus nimmst du heute?

Für kleine Mengen Monolithic, für Produktion Microservices. Simple Scalable wird abgekündigt und fällt mit Loki 4.0 weg, also starte ich damit nicht mehr neu.

DAS NIMMST DU MIT

- Chart aus `grafana-community`, Schema v13 mit TSDB.
- Retention über den Compactor, ohne Schalter bleibt alles.
- Simple Scalable nicht neu einführen.

TEIL VII

Traces und Korrelation

OpenTelemetry, Tempo, TraceQL und die Verbindung aller Signale.

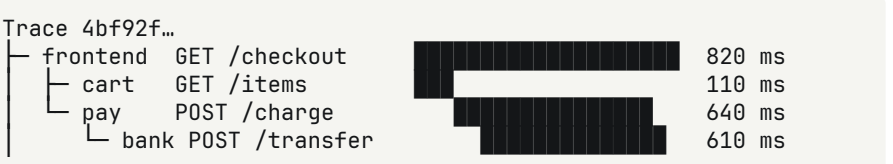
27

Tracing-Grundlagen

Eine Anfrage, viele Dienste: Wo ging die Zeit verloren?

Begriffe

Begriff	Bedeutung
Trace	Der ganze Weg einer Anfrage.
Span	Ein Arbeitsschritt mit Start, Dauer und Attributen.
Trace-ID	Verbindet alle Spans einer Anfrage.
Kontext-Weitergabe	Dienste reichen die IDs im Header weiter.



DAS BILD IM KOPF

Die Sendungsverfolgung eines Pakets. Die Nummer bleibt, die Stationen wechseln. Du siehst sofort, dass das Paket vier Tage im Zollamt lag.

Kontext-Weitergabe

Der W3C-Standard nutzt den HTTP-Header `traceparent`:

```
traceparent:
00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
   │  └── Trace-ID ──────────────────┘ └ Span-ID
└──┘ └ Flags
   └ Version
```

STOLPERFALLE

Reicht **ein** Dienst in der Mitte den Header nicht weiter, zerfällt der Trace in zwei Hälften. Oft ist es ein Proxy, ein altes Gateway oder eine Warteschlange.

OpenTelemetry

OpenTelemetry (OTel) ist der herstellerneutrale Standard. Es gibt SDKs, Auto-Instrumentierung, den Collector und das Protokoll **OTLP** auf den Ports 4317 (gRPC) und 4318 (HTTP).

```
App (SDK) ──OTLP──► Alloy oder Collector ──OTLP──► Tempo
```

Ohne Codeänderung geht es mit dem **OpenTelemetry Operator**: Eine Annotation am Pod injiziert die Auto-Instrumentierung. Für viele Sprachen ist das der schnellste Einstieg.

Pipeline in Alloy

```
otelcol.receiver.otlp "default" {
  grpc {}
  http {}
  output {
    traces = [otelcol.processor.batch.default.input]
  }
}

otelcol.processor.batch "default" {
  output {
    traces = [otelcol.exporter.otlp.tempo.input]
  }
}

otelcol.exporter.otlp "tempo" {
  client {
    endpoint = "tempo.tempo.svc:4317"
    tls { insecure = true }
  }
}
```

Sampling

Alle Traces zu speichern ist oft zu teuer.

Art	Entscheidung	Eigenschaft
Head Sampling	Am Anfang, im SDK	Billig, aber blind. Verliert seltene Fehler.
Tail Sampling	Am Ende, im Collector	Behält Fehler und langsame Traces, braucht RAM.

```
otelcol.processor.tail_sampling "default" {
  decision_wait = "10s"
  policy {
    name = "fehler"
    type = "status_code"
    status_code { status_codes = ["ERROR"] }
  }
  policy {
    name = "langsam"
    type = "latency"
    latency { threshold_ms = 1000 }
  }
  policy {
    name = "rest"
    type = "probabilistic"
    probabilistic { sampling_percentage = 5 }
  }
  output { traces = [otelcol.exporter.otlp.tempo.input] }
}
```

STOLPERFALLE

Beim Tail Sampling müssen alle Spans eines Traces beim selben Collector ankommen. Bei mehreren Instanzen brauchst du davor eine Lastverteilung nach Trace-ID, in Alloy etwa mit `otelcol.exporter.loadbalancing`.

IM INTERVIEW

Head oder Tail Sampling?

Head ist billig und einfach, aber die Entscheidung fällt, bevor klar ist, ob ein Trace interessant wird. Tail entscheidet nach Abschluss und behält gezielt Fehler und langsame Anfragen. Dafür braucht es Speicher im Collector und Routing nach Trace-ID.

IM INTERVIEW

Warum sind meine Traces zerrissen?

Meist wird der Kontext nicht weitergereicht, etwa durch einen Proxy oder eine Warteschlange, oder ein Dienst ist nicht instrumentiert. Auch Sampling, das nur Teile eines Traces behält, führt dazu.

DAS NIMMST DU MIT

- Trace besteht aus Spans, `traceparent` verbindet sie.
- OTel ist der Standard, OTLP auf 4317 und 4318.
- Tail Sampling behält das Interessante.

28

Tempo verstehen

Nur speichern, was du per ID wiederfindest, und fast alles liegt in S3.

Die Idee

Tempo braucht keine Datenbank und keinen Volltextindex. Traces liegen als Parquet-Blöcke in S3. Das macht es sehr günstig, auch bei vielen Spans.

Tempo 3 ist anders

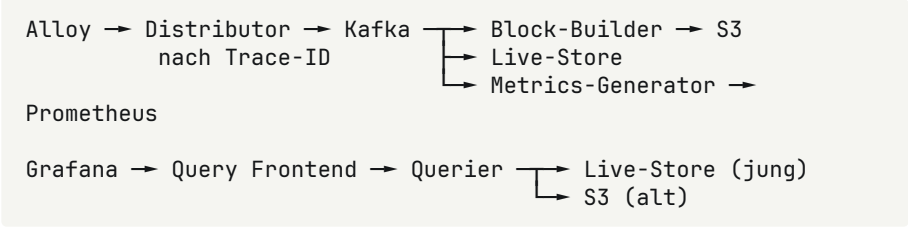
Tempo 3.0 hat das Ingester-Modell abgelöst. Viele Anleitungen zeigen noch die alte Architektur.

Alt (bis 2.x)	Neu (ab 3.0)
Ingester	Block-Builder (schreiben) und Live-Store (junge Daten lesen)
Compactor	Backend-Scheduler und Backend-Worker
Replikation zwischen Ingestern	Kafka im Microservices-Modus
Prozessor `local-blocks`	Entfernt. Der Live-Store bedient TraceQL-Metriken.

Zwei Modi

Modus	Bedeutung	Kafka
Monolithic	`-target=all`, alles in einem Prozess.	Nein
Microservices	Einzelne Dienste, für Produktion empfohlen.	Ja, Pflicht

Schreiben und Lesen



Komponente	Aufgabe
Distributor	Nimmt Spans an, verteilt nach Trace-ID.
Block-Builder	Liest aus Kafka, baut Parquet-Blöcke, schreibt nach S3.
Live-Store	Beantwortet Fragen nach jungen Daten.
Query Frontend	Teilt Abfragen in Aufträge.
Querier	Arbeitet die Aufträge ab.
Backend-Scheduler, -Worker	Compaction, Retention, Blockliste.
Metrics-Generator	Erzeugt RED-Metriken und Service-Graphen. Optional.

MERKE

Weil Kafka die Haltbarkeit sichert, kann Tempo auf dem Schreibpfad mit Replikationsfaktor 1 arbeiten. Jeder Block entsteht genau einmal, und das spart Speicher.

DAS BILD IM KOPF

Früher schrieben drei Sekretäre dasselbe Protokoll mit, und ein vierter sortierte die Dubletten aus. Heute gibt es ein Tonband (Kafka): Einer schreibt das Protokoll ab, ein anderer gibt Auskunft über das Neueste.

IM INTERVIEW

Wie speichert Tempo Traces so günstig?

Tempo legt Traces als Parquet-Blöcke in Objektspeicher ab und braucht keinen teuren Index. Abfragen nach Trace-ID und TraceQL laufen direkt auf den Blöcken. Dafür sind sehr breite Suchen langsamer als in einem indizierten System.

IM INTERVIEW

Was hat sich mit Tempo 3 geändert?

Ingester und Compactor sind weg. Im Microservices-Modus ist Kafka Pflicht, Block-Builder schreiben nach S3, Live-Stores bedienen junge Daten, und ein Backend-Scheduler verteilt Compaction und Retention. Lesen und Schreiben sind getrennt, und TraceQL-Metriken sind GA.

DAS NIMMST DU MIT

- Parquet in S3, kein Index-Cluster.
- Tempo 3: Block-Builder und Live-Store statt Ingester.
- Microservices brauchen Kafka, Monolithic nicht.

29

Tempo betreiben

Installation, S3, Retention und TraceQL.

Installation

Für den Einstieg nimmst du das monolithische Chart. Es braucht kein Kafka:

values-tempo.yaml

```yaml
tempo:
  retention: 168h
  storage:
    trace:
      backend: s3
      s3:
        bucket: tempo
        endpoint: s3.monitoring.svc:8333
        access_key: x
        secret_key: x
        insecure: true
      wal:
        path: /var/tempo/wal
```

```
helm install tempo grafana-community/tempo -n tempo --create-
namespace --version 3.2.1 -f values-tempo.yaml
```

Eselsbrücke: Auch Tempo liegt jetzt im Community-Repo.

Für Produktion gibt es `grafana-community/tempo-distributed` (Version 3.22). Es erwartet ein eigenes Kafka unter `ingest.kafka.address`. Die Zahl der Block-Builder und Live-Stores muss zur Zahl der Partitionen passen: ein Replikat pro Partition.

> **STOLPERFALLE**
>
> `grafana/tempo` und `grafana/tempo-distributed` sind abgekündigt und bleiben auf Tempo 2.9 stehen. Wer sie installiert, bekommt die alte Architektur.

Rechte in S3

Der Bucket muss schon existieren. Die Rolle braucht `PutObject`, `GetObject`, `ListBucket`, `DeleteObject`, `GetObjectTagging` und `PutObjectTagging`.

Retention

Tempo selbst behält Blöcke standardmäßig 336 Stunden, also 14 Tage. In der Konfiguration sieht das so aus:

```
compaction:
  block_retention: 336h
```

Pro Mandant gibt es unter `overrides` ein eigenes `compaction.block_retention`. Beobachte dazu `tempodb_retention_deleted_total`.

> **STOLPERFALLE**
>
> Das monolithische Chart setzt mit `tempo.retention` einen eigenen Wert, und der steht auf nur 24 Stunden. Ohne Anpassung sind Traces also nach einem Tag weg.

TraceQL

```
{ resource.service.name = "checkout" }
{ span.http.response.status_code >= 500 }
{ status = error && duration > 2s }
{ resource.service.name = "api" } >> { resource.service.name =
"db" }
```

`resource.` meint Attribute des Dienstes, `span.` Attribute des Spans. `>>` bedeutet: Der rechte Span ist ein Nachfahre des linken.

TraceQL-Metriken sind seit Tempo 3.0 GA. Sie rechnen direkt auf Traces:

```
{ resource.service.name = "api" } | rate()
{ status = error } | rate() by (resource.service.name)
{ } | quantile_over_time(duration, .99) by (span.http.route)
```

Alarme auf TraceQL-Metriken sind weiterhin experimentell.

Metrics-Generator

Er erzeugt aus Spans fertige Metriken und schickt sie per Remote Write an Prometheus oder Mimir. Im monolithischen Chart schaltest du ihn mit

`tempo.metricsGenerator.enabled` ein, die Adresse steht in `remoteWriteUrl`. In der Tempo-Konfiguration selbst sieht das so aus:

```
metrics_generator:
  storage:
    path: /var/tempo/generator/wal
    remote_write:
      - url: http://kps-kube-prometheus-prometheus.monitoring.
svc:9090/api/v1/write
overrides:
  defaults:
    metrics_generator:
      processors: [service-graphs, span-metrics]
```

Prometheus muss Remote Write annehmen:

```
prometheus:
  prometheusSpec:
    enableRemoteWriteReceiver: true
```

So entstehen zum Beispiel `traces_spanmetrics_calls_total`, `traces_spanmetrics_latency_bucket` und `traces_service_graph_request_total`. Grafana zeichnet daraus die Service Map.

STOLPERFALLE

Span-Attribute als Dimensionen sind wieder Kardinalität. Eine Dimension `user.id` im Metrics-Generator flutet Prometheus.

IM INTERVIEW

Wie suchst du in Tempo nach langsamen Anfragen eines Dienstes?

Mit TraceQL: `{ resource.service.name = "api" && duration > 2s }`. Für einen Überblick nutze ich TraceQL-Metriken wie `quantile_over_time` oder die RED-Metriken des Metrics-Generators.

IM INTERVIEW

Wie entstehen RED-Metriken ohne Code-Änderung?

Der Metrics-Generator von Tempo leitet sie aus Spans ab und schreibt sie per Remote Write in Prometheus oder Mimir. Der Prozessor `service-graphs` liefert zusätzlich die Abhängigkeiten zwischen Diensten.

DAS NIMMST DU MIT

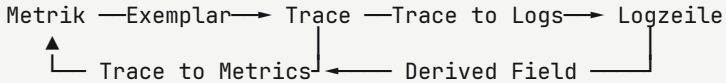
- Chart aus `grafana-community`.
- Standard-Retention: 14 Tage, `compaction.block_retention`.
- TraceQL-Metriken sind GA, Alarme darauf noch experimentell.

30

Signale verbinden

Aus drei Werkzeugen wird ein System: vom Alarm zur Logzeile in drei Klicks.

Der Rundgang



Gemeinsame Labels zuerst

Alle Signale brauchen dieselben Namen: `cluster`, `namespace`, `pod`, `service`. Sonst kann Grafana nichts verknüpfen. Darauf achtest du beim Relabeling in Prometheus und Alloy.

Exemplars: von der Metrik zum Trace

Ein Exemplar ist eine Trace-ID, die an einem Messwert hängt. Im Latenz-Graphen erscheinen Punkte, die direkt zum passenden Trace führen.

```
prometheus:
  prometheusSpec:
    enableFeatures: [exemplar-storage]
```

```
- name: Prometheus
  type: prometheus
  jsonData:
    exemplarTraceIdDestinations:
      - name: trace_id
        datasourceUid: tempo
```

Die Anwendung muss Exemplars liefern, und der Scrape braucht ein Format, das sie trägt, etwa OpenMetrics.

> **DAS BILD IM KOPF**
>
> *Ein Exemplar ist ein Lesezeichen im Diagramm: Dieser Ausreißer da, das war genau diese Anfrage. Hier blättern.*

Von den Logs zum Trace

Derived Fields machen aus einem Muster in der Logzeile einen Link:

```
- name: Loki
  type: loki
  uid: loki
  jsonData:
    derivedFields:
      - name: TraceID
        matcherRegex: '"trace_id":"(\w+)"'
        url: "$${__value.raw}"
        datasourceUid: tempo
```

Vom Trace zu Logs und Metriken

```
- name: Tempo
  type: tempo
  uid: tempo
  jsonData:
    tracesToLogsV2:
      datasourceUid: loki
      spanStartTimeShift: "-1h"
      spanEndTimeShift: "1h"
      tags:
        - { key: "k8s.namespace.name", value: "namespace" }
        - { key: "k8s.pod.name", value: "pod" }
      filterByTraceID: true
    tracesToMetrics:
      datasourceUid: prometheus
      tags:
        - { key: "service.name", value: "service" }
    serviceMap:
      datasourceUid: prometheus
    nodeGraph:
      enabled: true
```

> **MERKE**
>
> `datasourceUid` muss der festen `uid` der Zieldatenquelle entsprechen. Vergib deshalb beim Provisioning immer eigene UIDs.

Die Trace-ID ins Log

Ohne diesen Schritt fehlt die Brücke. Die Anwendung schreibt die aktuelle Trace-ID in jede Logzeile. OTel-Logging-Bridges erledigen das oft automatisch.

```
{"level":"error","msg":"charge failed","trace_id":"4bf92f35…"}
```

STOLPERFALLE

Die Links führen ins Leere, wenn die Trace-ID verworfen wurde. Bei Head Sampling von 1 Prozent zeigen 99 von 100 Log-Links auf nichts. Plane Sampling und Logging gemeinsam.

IM INTERVIEW

Wie kommst du vom Alarm zur Ursache?

Der Alarm verlinkt auf ein RED-Dashboard. Im Latenz-Graphen klicke ich auf ein Exemplar und lande beim Trace. Dort sehe ich den langsamen Span und springe über Trace to Logs zu den Logzeilen des Pods. Das funktioniert, weil alle Signale gemeinsame Labels und die Trace-ID tragen.

IM INTERVIEW

Was ist ein Exemplar?

Eine Trace-ID, die an einem einzelnen Metrik-Sample hängt. So komme ich aus einem aggregierten Graphen zu einem konkreten Beispiel. Prometheus speichert es nur, wenn `exemplar-storage` aktiv ist und die Anwendung es liefert.

DAS NIMMST DU MIT

- Erst gemeinsame Labels, dann Links.
- Exemplars führen zum Trace, Derived Fields auch.
- Die Trace-ID muss ins Log.

TEIL VIII

Produktion und Interview

Meta-Monitoring, Kosten, Fehlersuche, Fragen und Spickzettel.

31

Meta-Monitoring

Wer überwacht die Wächter?

Das Problem

Fällt der Observability-Stack genau dann aus, wenn der Cluster brennt, bist du blind. Überwache ihn deshalb getrennt. Für große Plattformen ist das ein eigener kleiner Stack, oft in einem anderen Cluster.

Loki und Mimir empfehlen dafür ein eigenes kleines Monitoring. Das Loki-Chart nennt den Monolithic- Modus ausdrücklich für solche Meta-Monitoring-Stacks.

Die wichtigsten Metriken

Metrik	Alarm, wenn
`up == 0`	Ein Scrape-Ziel fehlt.
`prometheus_rule_evaluation_failures_total`	Regeln schlagen fehl.
`prometheus_notifications_dropped_total`	Alarme gehen verloren.
`prometheus_tsdb_wal_corruptions_total`	Das WAL ist beschädigt.
`prometheus_remote_storage_samples_failed_total`	Remote Write schlägt fehl.
`alertmanager_notifications_failed_total`	Nachrichten gehen nicht raus.
`thanos_compact_halted`	Der Compactor steht.
`thanos_shipper_upload_failures_total`	Der Sidecar lädt nicht hoch.
`cortex_discarded_samples_total`	Mimir verwirft Daten.
`loki_discarded_samples_total`	Loki verwirft Daten.
`tempodb_retention_deleted_total`	Bleibt sie bei null, löscht Tempo nichts.

Hinkt Remote Write hinterher?

```
prometheus_remote_storage_highest_timestamp_in_seconds
  - ignoring(remote_name, url) group_right
prometheus_remote_storage_queue_highest_sent_timestamp_seconds
```

Letzter Upload des Sidecars vor mehr als drei Stunden?

```
time() - thanos_objstore_bucket_last_successful_upload_time > 3
* 3600
```

Sicherheit

- Grafana, Prometheus und Alertmanager nicht offen ins Netz stellen. Davor gehört SSO.
- Logs und Traces enthalten oft persönliche Daten. Filtere Tokens, E-Mail-Adressen und Passwörter schon im Sammler.
- Mandanten trennen und `X-Scope-OrgID` nur durch einen vertrauenswürdigen Proxy setzen lassen.
- Für Loki gilt die DSGVO wie für jeden Speicher: Die Retention muss zum Datenschutzkonzept passen.

IM INTERVIEW

Wer überwacht dein Monitoring?

Ein eigener kleiner Meta-Stack überwacht die Stack-Komponenten, und der Watchdog-Alarm läuft über einen externen Dead-Man's-Switch. Ich alarmiere auf Ingest-Fehler, Rückstand beim Remote Write, verworfene Daten und gestoppte Compactors.

DAS NIMMST DU MIT

- Monitoring nicht von sich selbst abhängig machen.
- Verworfene Daten sind ein Alarm wert.
- Logs enthalten sensible Daten.

32

Kapazität und Kosten

Die teure Frage: Was brauchen wir, und was behalten wir?

Was kostet was?

Signal	Treiber	Hebel
Metriken	Aktive Zeitreihen	Labels, `metric_relabel_configs`, längeres Intervall
Logs	GB pro Tag	Debug-Logs filtern, Retention pro Stream
Traces	Spans pro Sekunde	Tail Sampling, kürzere Retention

Eine Aufbewahrungsstrategie

Signal	Dauer	Wo
Metriken roh	30 Tage	Thanos oder Mimir in S3, davon 2 bis 3 Tage lokal
Metriken grob	1 bis 2 Jahre	Thanos: 5 min und 1 h. Mimir: Recording Rules.
Logs	30 Tage, Audit länger	Loki in S3 mit Retention pro Stream
Traces	7 bis 14 Tage	Tempo in S3

> **MERKE**
>
> Die alles entscheidende Frage lautet: Wer braucht diese Daten, und wofür? Ein Jahr lang Debug-Logs hilft selten. Ein Jahr Traffic-Trends hilft der Kapazitätsplanung sehr.

Rechnen fürs Interview

```
Logs:    50 GB/Tag × 30 Tage = 1,5 TB
         Loki komprimiert grob 5- bis 10-fach
         also rund 150 bis 300 GB in S3

Metriken: 2 Mio. Zeitreihen, 30 s Intervall
         = 66.667 Samples/s
         × 86.400 s × 1,5 Byte = rund 8,6 GB/Tag
```

```
          × 30 Tage = rund 260 GB

S3:       Grob 2 Cent pro GB und Monat (Standard, je nach Region)
          1 TB = rund 20 Euro im Monat
```

Der Speicher ist selten das Teuerste. Die Rechner für Ingester und Querier und die Übertragung zwischen Zonen kosten oft mehr.

> **STOLPERFALLE**
>
> Übertragung zwischen Availability Zones kann bei Replikation und bei Remote Write die Rechnung dominieren. Das Muster: In jeder Zone viel Traffic, und alles geht einmal quer durchs Netz.

Prometheus aufteilen

Wird ein Prometheus zu groß, hast du drei Wege:

- **Funktional:** Eine Instanz pro Team oder Bereich.
- **Horizontal:** Operator-Feld `shards` verteilt Ziele per Hash auf mehrere Instanzen.
- **Agent:** Prometheus nur als Sammler, Speicherung in Mimir.

> **IM INTERVIEW**
>
> **Die Observability-Kosten sind zu hoch. Wo setzt du an?**
>
> Zuerst messe ich, welche Metriken, Namespaces und Log-Quellen am teuersten sind. Dann entferne ich ungenutzte Metriken und Labels, senke Log-Level, trenne die Retention nach Wert und führe Tail Sampling ein. Danach schaue ich auf Infrastruktur wie Cross-AZ-Verkehr und Caches.

DAS NIMMST DU MIT

- Metriken: Zeitreihen. Logs: GB. Traces: Spans.
- Retention nach Wert trennen.
- Oft sind nicht die Bytes in S3 teuer, sondern Rechner und Netz.

33

Fehlersuche in Szenarien

Die Fälle, die in Interviews als Geschichte kommen.

Target fehlt

Symptom: Keine Daten von der neuen Anwendung.

1. Ist es unter **Status → Targets** gelistet? Nein: Kette aus Kapitel 10.
2. Status `down`? `kubectl exec` und `curl` auf den Port.
3. `Connection refused`? Falscher Port oder Pfad.
4. `403`? Authentifizierung oder RBAC.

Prometheus im OOM-Kreislauf

Symptom: Pod startet immer neu, Log zeigt WAL-Replay.

1. `kubectl describe pod`: Grund `OOMKilled`?
2. Neuer Job oder neue Labels in den letzten Tagen?
3. Vorübergehend mehr RAM geben, damit das Replay durchläuft.
4. Danach TSDB-Status prüfen und die Quelle beseitigen.
5. `sample_limit` setzen, damit es nicht wieder geschieht.

Lücken im Graphen

Mögliche Ursachen:

- Prometheus startete neu oder war überlastet. `scrape_duration_seconds` nah am Timeout?
- Ziel antwortet zu langsam.
- Bei zwei Replikaten: Beide scrapen einzeln, also nimm Thanos Query mit Deduplizierung.
- `rate` mit zu kurzem Fenster.

Alarm kam nicht

1. Prometheus: Ist er unter `/alerts` im Zustand `firing`?
2. Alertmanager: Ist er angekommen? Greift eine Silence oder Inhibition?
3. `amtool config routes test` für die Route.

4. `alertmanager_notifications_failed_total` und Log.
5. Empfänger: Stimmen Webhook und Secret?

Abfragen über alte Zeiträume sind leer

1. Thanos: Zeigt `thanos tools bucket inspect` die Blöcke?
2. Sehen Store Gateways sie? `Stores`-Seite in Thanos Query.
3. Sidecar lädt hoch? `thanos_shipper_upload_failures_total`.
4. Compactor angehalten? `thanos_compact_halted`.
5. Retention falsch? Wurde die Auflösung schon gelöscht?

Store Gateway braucht zu viel RAM

- Index-Header wachsen mit der Blockzahl: Läuft der Compactor?
- Sehr große Abfragen: Query Frontend und Limits setzen.
- Blöcke nach Zeit auf mehrere Store Gateways verteilen, mit `--min-time` und `--max-time`.

Loki antwortet mit 429

1. Log von Alloy: Welche Meldung genau?
2. `loki_discarded_samples_total` nach `reason`.
3. `rate_limited`: Limits anheben oder Quelle drosseln.
4. `stream_limit`: Mit `logcli series --analyze-labels` die Treiber finden und Labels entfernen.

Trace fehlt

1. Sampling: Wurde er verworfen?
2. Ist Alloy erreichbar? Ports 4317 und 4318.
3. Ist er zu alt, also außerhalb der Retention?
4. Junge Traces: Live-Stores gesund? Alte: Backend-Worker und S3-Rechte prüfen.
5. Trace-ID und Header: Spricht jeder Dienst `traceparent`?

IM INTERVIEW

Es ist drei Uhr nachts, das Dashboard ist leer. Was tust du?

Ich prüfe zuerst, ob die Daten fehlen oder nur die Anzeige: Explore in Grafana, dann Prometheus direkt. Ist `up` da? Sind die Pods gesund? Dann gehe ich die Kette von der Quelle zur Anzeige durch: Target, Scrape, Speicher, Datenquelle. Erst danach schaue ich, was sich zuletzt geändert hat.

DAS NIMMST DU MIT

- Kette von der Quelle bis zur Anzeige prüfen.
- Metriken der Werkzeuge selbst nutzen.
- Fast jeder RAM-Fall ist Kardinalität.

34

Die Interviewfragen

Zum lauten Beantworten. Decke die Antwort ab.

Grundlagen

IM INTERVIEW

Metriken, Logs, Traces: Wann was?

Metriken für Trends und Alarme, Traces zur Lokalisierung, Logs für das Detail.

IM INTERVIEW

Was sind RED und USE?

RED misst Dienste: Rate, Errors, Duration. USE misst Ressourcen: Utilization, Saturation, Errors.

IM INTERVIEW

Was ist Kardinalität?

Die Zahl der einzelnen Label-Kombinationen, also der Zeitreihen. Sie treibt RAM, Kosten und Abfragezeit.

Prometheus

IM INTERVIEW

Pull gegen Push?

Pull erkennt Ausfälle durch `up` und schützt vor Überlast. Push braucht man für kurze Jobs und getrennte Netze.

IM INTERVIEW

Warum darf man Counter nicht direkt anzeigen?

Sie steigen nur und fallen beim Neustart auf null. Aussagekräftig ist die Änderungsrate mit `rate()`.

IM INTERVIEW

Warum `rate` vor `sum`?

`rate` erkennt Counter-Resets nur an einzelnen Reihen. Nach dem Summieren sind sie nicht mehr erkennbar.

IM INTERVIEW

Histogram oder Summary?

Histogram, denn es lässt sich über Pods aggregieren. Summaries nicht.

IM INTERVIEW

Was ist der Unterschied zwischen `for` und `keep_firing_for`?

`for` verzögert das Auslösen, `keep_firing_for` verzögert das Beenden.

IM INTERVIEW

Was ist `up` wert?

Es zeigt nur, ob der Scrape klappte. Es sagt nichts darüber, ob der Dienst korrekt arbeitet.

IM INTERVIEW

Wie aggregierst du Daten über mehrere Cluster?

Mit Thanos Query, Mimir oder Remote Write zentral. Klassische Federation nur für wenige vorberechnete Reihen.

Operator und Kubernetes

IM INTERVIEW

Was tut der Prometheus Operator?

Er macht aus CRDs wie `ServiceMonitor` und `PrometheusRule` die Prometheus-Konfiguration und verwaltet die Instanzen.

IM INTERVIEW

Was ist `kube-state-metrics`?

Ein Dienst, der den Zustand der Kubernetes-Objekte als Metriken zeigt. Er misst nicht den Verbrauch.

IM INTERVIEW

Warum `working_set` und nicht `usage`?

Weil `usage` Page Cache enthält, den der Kernel freigeben kann. `working_set` kommt dem OOM-Verhalten am nächsten.

Alerting

IM INTERVIEW

Was macht Inhibition?

Sie unterdrückt Folgealarme, solange die Ursache feuert.

IM INTERVIEW

Was bedeutet Alarmieren auf Symptome?

Ich alarmiere auf das, was Nutzer spüren, und nutze Ursachen wie CPU nur beim Suchen.

IM INTERVIEW

Was ist ein Error Budget?

Der erlaubte Rest bis zum SLO-Ziel. Er steuert, ob ich riskant ausrollen darf.

Langzeit und S3

IM INTERVIEW

Was tut der Thanos Compactor?

Er komprimiert Blöcke, erzeugt Downsampling, wendet Retention an und darf pro Blockstrom nur einmal laufen.

IM INTERVIEW

Wann ist Mimir statt Thanos sinnvoll?

Bei zentralen Plattformen mit vielen Mandanten, harten Limits und Teams, die Kafka tragen können.

IM INTERVIEW

Warum muss ein Thanos-Prometheus 2-Stunden-Blöcke schreiben?

Damit der Sidecar unverdichtete Blöcke hochladen kann und allein der Compactor im Bucket verdichtet.

IM INTERVIEW

Wer löscht in S3?

Immer das Werkzeug über seine Retention, nicht eine Lifecycle-Regel.

Logs und Traces

IM INTERVIEW

Warum ist Loki günstig?

Es indiziert nur Labels und legt den komprimierten Text in S3 ab.

IM INTERVIEW

Was ersetzt Promtail?

Grafana Alloy. Promtail ist seit dem 2. März 2026 End of Life.

IM INTERVIEW

Welcher Loki-Modus läuft aus?

Simple Scalable. Er fällt mit Loki 4.0 weg.

IM INTERVIEW

Was brauchst du für Tempo in Microservices?

Kafka oder ein kompatibles System sowie Objektspeicher.

IM INTERVIEW

Wie verbindest du die Signale?

Gemeinsame Labels, Exemplars, Trace-ID im Log und Links in den Grafana-Datenquellen.

Architektur

IM INTERVIEW

Entwirf Observability für 50 Cluster und 200 Teams.

In jedem Cluster Prometheus im Agent-Modus oder schlank, dazu Alloy. Zentral Mimir, Loki und Tempo als Mandantenplattform mit S3 und Authentifizierung per Proxy. Limits und Retention pro Team, SLO-Alarme als Vorlagen, Dashboards in Git und ein separater Meta-Stack.

IM INTERVIEW

Ein Kollege will alle Logs als Label indizieren. Was sagst du?

Das zerstört Loki. Jede Kombination wird ein Stream. Ich zeige die Auswirkung mit `logcli series --analyze-labels` und schlage Structured Metadata und Filter zur Abfragezeit vor.

DAS NIMMST DU MIT

- Antworte kurz, dann ein Beispiel.
- Nenne Abwägungen und keine Absolutheiten.
- Sag, was du messen würdest.

35

Aufgaben

Ohne Musterlösung vorn. Erst selbst, dann nachschauen.

1. **Kardinalität:** Eine Metrik hat 30 Routen, 4 Methoden, 6 Statuscodes und 20 Pods. Wie viele Reihen entstehen höchstens? Was passiert mit dem Label `tenant` mit 500 Werten?
2. **ServiceMonitor:** Schreibe Service und ServiceMonitor für eine App auf Port 9100 im Namespace `demo`, der von `kps` gefunden wird.
3. **PromQL:** Schreibe eine Abfrage für die p95-Latenz pro Route und eine für die Fehlerquote.
4. **Regel:** Schreibe einen Alarm für eine Fehlerquote über 2 Prozent seit 10 Minuten und teste ihn mit `promtool test rules`.
5. **Routing:** Baue einen Route-Baum: kritisch an PagerDuty, Team `shop` nach Slack, alles andere in einen Sammelkanal. Prüfe mit `amtool`.
6. **SLO:** Rechne: Wie viele Minuten Ausfall erlauben 99,95 Prozent in 30 Tagen?
7. **Thanos:** Schalte den Sidecar ein und zeige einen Block im Bucket.
8. **Retention:** Plane 30 Tage roh, 180 Tage in 5 Minuten, 2 Jahre in 1 Stunde. Was schätzt du für den Speicher, wenn die Rohdaten 100 GB pro Monat kosten?
9. **Loki:** Installiere Loki mit S3 und 30 Tagen Retention, schicke Logs mit Alloy, und finde mit LogQL die Fehler pro Minute.
10. **Tempo:** Schicke Traces per OTLP an Tempo und suche mit TraceQL alles über zwei Sekunden.
11. **Korrelation:** Verbinde Loki und Tempo in beide Richtungen.

Lösungen

1. $30 \times 4 \times 6 \times 20 = 14.400$ Reihen. Mit `tenant` wären es 7,2 Millionen.

6. 30 Tage sind 43.200 Minuten. 0,05 Prozent davon sind 21,6 Minuten.

8. Die Rohdaten bleiben bei rund 100 GB. Dazu kommen die 5-Minuten-Blöcke über 180 Tage und die 1-Stunden-Blöcke über 2 Jahre. Eine feste Formel gibt es nicht, denn der Faktor hängt von den Metriken ab. Miss deshalb im Bucket nach. Wichtig ist der Gedanke: Du sparst durch die kürzere Retention der Rohdaten, nicht durch das Downsampling.

DAS NIMMST DU MIT

- Selbst tippen ist der Lerneffekt.
- Jede Aufgabe ist eine Interview-Geschichte.
- Merke dir, was schiefging.

36

Spickzettel

Alles Wichtige auf ein paar Seiten.

Versionen und Charts

Werkzeug	Version	Chart
Prometheus	3.15	`kube-prometheus-stack` 92.18
Alertmanager	0.34	gleiches Chart
Grafana	13.3	`grafana-community/grafana` 14.3
Thanos	0.42	kein offizielles
Mimir	3.2	`grafana/mimir-distributed` 6.2
Loki	3.7	`grafana-community/loki` 20.4
Tempo	3.1	`grafana-community/tempo` 3.2, `tempo-distributed` 3.22
Alloy	1.21	`grafana/alloy` 1.15

PromQL

```
sum(rate(x_total[5m]))
sum by (le) (rate(x_bucket[5m]))
histogram_quantile(0.99, sum by (le) (rate(x_bucket[5m])))
sum(rate(x_total{status=~"5.."}[5m])) / sum(rate(x_total[5m]))
predict_linear(node_filesystem_avail_bytes[6h], 4 * 3600) < 0
absent(up{job="api"})
topk(10, count by (__name__) ({__name__=~".+"}))
```

LogQL und TraceQL

```
{app="api"} |= "error" | json | status >= 500
sum by (app) (rate({namespace="shop"} |= "error" [5m]))
```

```
{ resource.service.name = "api" && duration > 2s }
{ status = error } | rate() by (resource.service.name)
```

Retention

Werkzeug	Einstellung	Standard
Prometheus	`--storage.tsdb.retention.time`	15 d (Chart: 10 d)
Thanos	`--retention.resolution-raw`, `-5m`, `-1h`	unendlich
Mimir	`compactor_blocks_retention_period`	unendlich
Loki	`retention_enabled`, `retention_period`	unendlich
Tempo	`compaction.block_retention`	14 d

Zahlen, die du kennen musst

Wert	Bedeutung
2 h	Länge eines TSDB-Blocks.
1–2 Byte	Speicher pro Sample in Prometheus.
30 s / 5 min / 4 h	`group_wait`, `group_interval`, `repeat_interval`.
40 h / 10 d	Wann Thanos 5-Minuten- und 1-Stunden-Blöcke erzeugt.
14,4 / 6 / 1	Burn Rates über 1 h, 6 h, 3 Tage.
43,2 min	Fehlerbudget bei 99,9 Prozent in 30 Tagen.
4317 / 4318	OTLP über gRPC und HTTP.
9090 / 9093 / 3100 / 3200	Prometheus, Alertmanager, Loki, Tempo.
12345	Alloy-Oberfläche.
9094	Alertmanager-Gossip.
10901	Thanos gRPC.

Niemals ohne Nachdenken

- Unbegrenzte Werte als Label verwenden.
- Einen Counter ohne `rate` anzeigen.
- Zwei Thanos-Compactors auf denselben Blockstrom setzen.
- Mit einer S3-Lifecycle-Regel Daten löschen, die das Werkzeug verwaltet.
- Glacier für abfragbare Daten nutzen.
- `trace_id` als Loki-Label anlegen.
- Promtail oder Simple Scalable neu einführen.
- Einen Load Balancer zwischen Prometheus und Alertmanager stellen.
- `X-Scope-OrgID` ohne Authentifizierung vertrauen.
- Den Watchdog nicht nach außen führen.

DAS NIMMST DU MIT

- Kardinalität entscheidet über Kosten.
- Heiße Daten lokal, alte Daten in S3, das Werkzeug löscht.
- Gemeinsame Labels machen aus drei Signalen ein System.

Stichwörter

Agent-Modus 21
Alert Fatigue 48
Alerting Rules 36
Alertmanager 44
Alloy 10
amtool 44
Block-Builder 97
block_retention 100
Burn Rate 48
cAdvisor 39
Chunk 80
Churn 17
Compactor 62
compactor_blocks_retention_period 73
cortex_discarded_samples_total 75
Counter 22
Dashboards as Code 55
Datasource 52
deploymentMode 88
derivedFields 104
Downsampling 65
Error Budget 48
Exemplars 104
Exporter 20
Gauge 22
Golden Signals 9
Grafana 52
group_left 25
group_wait 44
Head Block 27
Histogram 22
histogram_quantile 24
Hochverfügbarkeit 28
Ingest Storage 70
inhibit_rules 44
IRSA 58
Kafka 70
Kardinalität 16
kind 12
Korrelation 104
Kosten 110
kube-prometheus-stack 12
kube-state-metrics 39
Labels 16
LGTM 10
Lifecycle 58
Live-Store 97
logcli 83
LogQL 83
Logs 8
Loki 80
loki.source.kubernetes 85
loki_discarded_samples_total 90
Meta-Monitoring 108
metric_relabel_configs 30
Metrics-Generator 97
metrics-server 39
Metriken 8
Mimir 70
mimir-distributed 73
Monitoring 8
Native Histograms 23
node-exporter 39
Objektspeicher 58
objstore.yml 65
Observability 8
OpenTelemetry 94
OpenTelemetry Collector 85
OTLP 94
PodMonitor 32
Profile 9
Prometheus Operator 32
PrometheusRule 32
PromQL 24
Promtail 85
promtool 36
Pull-Modell 20
Pushgateway 21
rate 24
Receive 62
Recording Rules 36
RED 9
relabel_configs 30
Remote Write 21
Retention 88
S3 58
Sampling 94
schema_config 88
Service Graph 100
ServiceMonitor 32

Sidecar 62
Silence 44
SLI 48
SLO 48
Span 94
Store Gateway 62
Stream 80
Structured Metadata 80
Tempo 97
Tenant 70
Thanos 62
Thanos Query 62
Trace 94
traceparent 94
TraceQL 100
Traces 8
Troubleshooting 112
TSDB 27
up 20
USE 9
WAL 27
Watchdog 48
X-Scope-OrgID 70