## Datenmodell

### Metriktypen

Counter: monoton steigend, nur mit `rate()` / `increase()` auswerten.
Gauge: frei beweglicher Momentanwert (Queue-Laenge, Memory).
Histogram: kumulative `le`-Buckets + `_sum` / `_count` – serverseitig aggregierbar.
Summary: client-seitig berechnete Quantile – nicht ueber Instanzen aggregierbar.

### Labels, Serien, Cardinality

Jede eindeutige Label-Kombination erzeugt eine eigene Zeitserie. Serienzahl waechst multiplikativ mit den Wertemengen der Labels – Cardinality ist die zentrale Kostengroesse fuer RAM, Ingest und Query-Latenz. Labels nur fuer Dimensionen mit kleiner, stabiler Wertemenge.

### Native Histograms (stabil seit 3.8)

Exponentielle Buckets mit hoher Aufloesung in einer Serie statt Dutzender `le`-Serien. Scraping explizit aktivieren: `scrape_native_histograms: true` (global oder pro Job, Protobuf-Scrape-Format). Das alte Feature-Flag `native-histograms` ist seit 3.9 wirkungslos.

### Prometheus 3.x

UTF-8 in Metrik-/Label-Namen (Rueckfall: `metric_name_validation_scheme: legacy`).
OTLP-Receiver: `--web.enable-otlp-receiver`.
Agent-Mode: `--agent` – nur Scrape + WAL + Remote-Write, kein Querying/Alerting/lokale TSDB.
Defaults: `scrape_interval` 1m, `scrape_timeout` 10s, TSDB-Retention 15d.

```
# Cardinality inspizieren
curl -s localhost:9090/api/v1/status/tsdb | jq .
# Top-10 Metriken nach Serienzahl (PromQL)
topk(10, count by (__name__)({__name__=~".+"}))
```

## PromQL-Kern

### rate vs irate

`rate(v[5m])`: per-second-Durchschnitt uebers Fenster, extrapoliert an die Fensterraender, Counter-Resets werden korrigiert – Standard fuer Alerts und Dashboards.
`irate(v[5m])`: Momentanrate aus den letzten zwei Samples – reagiert schnell, verschluckt alles dazwischen. Nie fuer Alerts. Fenster immer gross genug fuer mehrere Samples waehlen.

### Aggregationen

`sum`, `avg`, `min`, `max`, `count`, `topk`, `quantile` – Dimensionen steuern mit `by (labels)` (behalten) oder `without (labels)` (entfernen). Regel: erst `rate()` pro Serie, dann `sum()` – nie umgekehrt.

### histogram_quantile

`histogram_quantile(0.99, ...)` berechnet das $\varphi$-Quantil aus klassischen `le`-Buckets (vorher `rate()` auf `_bucket`, `by (le)` behalten!) oder direkt aus Native Histograms. Ergebnis ist eine Schaetzung (Interpolation innerhalb des Buckets).

```
# Fehlerquote pro Service (5m-Fenster)
sum by (service)
  (rate(http_requests_total{code=~"5.."}[5m]))
/ sum by (service)
  (rate(http_requests_total[5m]))

# p99 aus klassischem Histogram
histogram_quantile(0.99, sum by (le)
  (rate(http_request_duration_seconds_bucket[5m])))

# p99 aus Native Histogram (keine _bucket-Serien)
histogram_quantile(0.99,
  sum(rate(http_request_duration_seconds[5m])))
```

## Scrape-Config & Service Discovery

### scrape_configs

Pro `job_name` entweder `static_configs` oder eine `*_sd_configs`-Discovery. Kubernetes-Rollen (`kubernetes_sd_configs`): `endpoints`, `endpointslice`, `service`, `pod`, `node`, `ingress`. Discovery liefert `__meta_*`-Labels als Rohmaterial fuers Relabeling.

### relabel vs metric_relabel

`relabel_configs`: vor dem Scrape, auf Target-Ebene – Targets filtern (`keep`/`drop`), Adresse/Pfad setzen, `__meta_*` in echte Labels ueberfuehren.
`metric_relabel_configs`: nach dem Scrape, auf jede Serie – letzte Verteidigungslinie gegen Cardinality.
Actions: `replace`, `keep`, `drop`, `labelmap`, `hashmod`, `labeldrop`, `labelkeep`.

```
scrape_configs:
- job_name: kubernetes-pods
  kubernetes_sd_configs:
  - role: pod
  relabel_configs:
  - source_labels:
    - __meta_kubernetes_pod_annotation_prometheus_io_scrape
    action: keep
    regex: "true"
  - source_labels: [__meta_kubernetes_pod_name]
    action: replace
    target_label: pod
  metric_relabel_configs:
  - source_labels: [__name__]
    action: drop
    regex: go_gc_.*
```

## Recording & Alerting Rules

### Recording Rules

Teure Ausdruecke vorberechnen, Dashboards/Alerts lesen die fertige Serie. Naming-Konvention: `level:metric:operations`, z.B. `job:http_requests:rate5m`. Auswertung pro Rule-Group sequenziell, Takt `evaluation_interval` (Default 1m).

### Alerting Rules

`expr` + `for`: Alert ist erst pending, nach Ablauf von `for` firing – entprellt Flapping. `labels` (z.B. `severity`) steuern das Routing im Alertmanager, `annotations` sind fuer Menschen (`summary`, Runbook-Link).

```
groups:
- name: api.rules
  rules:
  - record: job:http_requests:rate5m
    expr: sum by (job) (rate(http_requests_total[5m]))
  - alert: HighErrorRate
    expr: |
      sum by (job)
        (rate(http_requests_total{code=~"5.."}[5m]))
      / sum by (job)
        (rate(http_requests_total[5m])) > 0.05
    for: 10m
    labels: {severity: page}
    annotations:
      summary: >-
        Fehlerquote ueber 5% in {{ $labels.job }}
```

## Alertmanager

### Routing-Tree

Ein Wurzel-`route`-Block, Kinder matchen per `matchers`. Erster Match gewinnt (ausser `continue: true`). Die Root-Route muss alles matchen (keine Matchers) – sie ist der Default-Receiver.

### Grouping, Timing

`group_by` buendelt Alerts zu einer Notification.
`group_wait` (Default 30s): Wartezeit vor der ersten Notification einer neuen Gruppe.
`group_interval` (5m): Takt fuer Updates der Gruppe.
`repeat_interval` (4h): Wiederholung unveraenderter Alerts – als Vielfaches von `group_interval` waehlen.

### Inhibition & Silences

`inhibit_rules`: unterdruecken abhaengige Alerts (z.B. `warning`, solange dasselbe `critical` feuert) – `equal:` verhindert Querschuesse zwischen Clustern.
Silences: ad hoc via UI/`amtool`, immer mit Ablaufzeit. Geplante Wartungsfenster: `mute_time_intervals`.

```yaml
route:
  receiver: default
  group_by: [alertname, cluster]
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
  - matchers: [severity="page"]
    receiver: oncall
inhibit_rules:
- source_matchers: [severity="critical"]
  target_matchers: [severity="warning"]
  equal: [alertname, cluster]
```

## Remote-Write vs Federation

### Remote-Write

Streamt Samples (via `write_relabel_configs` gefiltert) an ein Langzeit-Backend. Remote-Write 2.0: `protobuf_message: io.prometheus.write.v2.Request` (Default bleibt `prometheus.WriteRequest`); Metadata-Versand braucht `--enable-feature=metadata-wal-records`. Receiver-Seite: `--web.enable-remote-write-receiver`.

### Federation

`/federate`-Endpoint, Auswahl per `match[]`-Selektoren. Gedacht fuer aggregierte Serien (hierarchisch) oder gezielte Cross-Service-Metriken – kein Voll-Replikat und kein Langzeit-Backup. Fuer Langzeit/HA: Remote-Write.

```yaml
remote_write:
- url: https://metrics.example.com/api/v1/push
  protobuf_message: io.prometheus.write.v2.Request
  write_relabel_configs:
  - source_labels: [__name__]
    action: drop
    regex: go_.*
```

## Langzeit & HA: Thanos / Mimir

### Thanos (CNCF Incubating)

Sidecar neben jedem Prometheus laedt TSDB-Bloecke in Object Storage; Query liefert die globale Sicht und dedupliziert HA-Paare; Store Gateway liest alte Bloecke, Compactor verdichtet und downsampled, Receive nimmt Remote-Write an. Prometheus bleibt die Scrape-Quelle.

### Grafana Mimir (AGPLv3)

Horizontal skalierbarer, nativ multi-tenanter Langzeit-Store; Ingest ausschliesslich via Remote-Write, Ablage in Object Storage (S3/GCS/Azure). Projektangabe: bis zu 1 Mrd. aktive Serien. Wahl: Thanos haelt Prometheus im Zentrum, Mimir zentralisiert.

### HA-Grundmuster

Zwei identische Prometheus-Replicas scrapen dieselben Targets; Deduplizierung passiert downstream (Thanos Query, Mimir-HA-Dedup). Beide Replicas melden an denselben Alertmanager-Cluster – der dedupliziert Notifications.

## Prometheus Operator

### CRDs statt Config-Files

ServiceMonitor (Services), PodMonitor (Pods), Probe (Blackbox/Ingress), ScrapeConfig (Targets ausserhalb des Clusters), PrometheusRule (Recording/Alerting), AlertmanagerConfig (Routing-Teilbaeume). Der Operator generiert daraus die laufende Prometheus-Konfiguration; die Prometheus-CR waehlt CRs per Label-Selektor aus.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: api
  labels: {team: platform}
spec:
  selector:
    matchLabels: {app: api}
  endpoints:
  - port: http-metrics
    interval: 30s
    path: /metrics
```

### Stolperfallen

`port` im ServiceMonitor ist der Port-Name des Service, nicht die Nummer. CR wird nicht gescraped? Label-Selektor der Prometheus-CR pruefen – nicht selektierte CRs werden still ignoriert.

## Anti-Patterns

### Was du nicht tun solltest

Unbounded Labels: `user_id`, `request_id`, volle URLs als Label-Wert – Cardinality-Explosion bis zum OOM. IDs gehoeren in Logs/Traces, nicht in Labels.
irate() in Alerts: sieht nur die letzten zwei Samples – kurze Spikes und Luecken kippen den Alert. `rate()` nehmen.
sum() vor rate(): Counter-Resets einzelner Targets gehen als Spruenge in die Summe ein – immer erst `rate()`, dann `sum()`.
Summary-Quantile mitteln: `avg()` ueber client-seitige Quantile ist statistisch bedeutungslos – Histogram + `histogram_quantile`.
Federation als Backup: `/federate` repliziert keine Historie und skaliert nicht fuer alles – Langzeit via Remote-Write.
Ein globaler Mega-Prometheus: pro Cluster/Zone scrapen, globale Sicht ueber Thanos/Mimir – nicht ueber ein einzelnes, vertikal gemaestetes Prometheus.

prometheus-cheatsheet.de

Prometheus Cheatsheet von OMNI52

## Weiterfuehren, nicht selbst neu erfinden

### Observability-Stack & Alerting-Design

OMNI52™ hilft Plattform-Teams, Prometheus fuer regulierte Enterprise-Umgebungen sauber aufzustellen.

Cardinality-Governance mit Relabeling-Strategie, HA-Design mit Thanos oder Mimir, Alert-Routing das On-Call nicht verbrennt, Operator-basiertes Rollout via GitOps. Output landet als GitOps-Repo bei euch im Cluster: Helm-Charts, Rule-Packs, Runbooks. Euer Team operiert weiter, nicht wir.

OMNI52 GmbH, Ebern · istio-consulting.de · kostenloses 30-min Initial-Assessment

### Aus der OMNI52 Cheatsheet-Fabrik

Verwandte Sheets: kubernetes-cheatsheet.de (Kubernetes Core) · istio-cheatsheet.de (Istio (Mesh-Telemetrie)) · cilium-cheatsheet.de (Cilium/Hubble) · rke2-cheatsheet.de (RKE2 (rancher-monitoring)).

### Lizenz & Weiterverteilung



### Trademarks