nyrvex.com

Kubernetes Observability

Hands-On

Prometheus, Alertmanager, Grafana, Thanos, Mimir, Loki, Tempo, and S3 for the old data



Interview-ready with minimal text

Kubernetes Observability

Hands-On

Prometheus, Alertmanager, Grafana, Thanos, Mimir, Loki, Tempo, and S3 for the old data

Interview-ready with minimal text

Thomas Zachmann

Kubernetes Observability Hands-On. Prometheus, Alertmanager, Grafana, Thanos, Mimir, Loki, Tempo, and S3 for the old data.

As of November 4, 2026. © 2026 Thomas Zachmann. License: CC BY-ND 4.0.

Written for 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, and 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. The commands, configurations, and sample outputs in this book have not been run on a cluster. Outputs are examples, and details can vary by version. Try everything in a lab first, never on a production cluster.

Typefaces: Charter, Fira Sans, JetBrains Mono.

Contents

How to Read This Book 5

Fundamentals **7**
1 Why Observability? 8
2 The Map 10
3 Your Lab 12
4 Labels and Cardinality 16

Prometheus **19**
5 The Prometheus Model 20
6 Metric Types 22
7 PromQL 24
8 The TSDB 27
9 Scrape and Relabeling 30
10 The Prometheus Operator 32
11 Rules 36
12 Measuring Kubernetes 39

Alerting **43**
13 Alertmanager 44
14 Good Alerts 48

Grafana **51**
15 Grafana 52
16 Dashboards That Help 55

Long-Term Metrics and S3 **57**
17 S3 for Old Data 58
18 Understanding Thanos 62
19 Operating Thanos 65
20 Understanding Mimir 70
21 Operating Mimir 73
22 Thanos or Mimir? 77

Logs **79**
23 Understanding Loki 80
24 LogQL 83
25 Alloy 85
26 Operating Loki 88

Traces and Correlation **93**
27 Tracing Fundamentals 94
28 Understanding Tempo 97
29 Operating Tempo 99
30 Connecting the Signals 103

Production and the Interview **107**
31 Meta-Monitoring 108
32 Capacity and Costs 110
33 Troubleshooting Scenarios 112
34 The Interview Questions 115
35 Exercises 120
36 Cheat Sheet 122
Index 125

How to Read This Book

You have an interview coming up and you need to know observability on Kubernetes. Not someday. Soon. This book is built for exactly that: little text, lots of small examples, and for every topic the question you will actually be asked.

The Rules

- **Type everything yourself.** Don't copy anything. Your fingers learn too.
- **One command at a time.** You type, you watch what happens.
- **Lab only.** A kind cluster is enough for almost everything, see chapter 3.
- **Understand the why first.** Interviews rarely test YAML fields. They test whether you can explain why an alert never came or why the memory blew up.

> **REMEMBER**
>
> The commands, configurations, and outputs in this book have not been run. The outputs are shortened examples. Yours will differ.

What's Changing Right Now

Many guides on the web are out of date. These points show the interviewer that you are current:

- **Promtail is dead.** It has been end of life since March 2, 2026. Its successor is Grafana Alloy.
- **Mimir 3 prefers Kafka.** The ingest storage architecture separates the write and read paths.
- **Tempo 3 has no more ingesters.** Block-Builders and Live-Stores replace them. In microservices mode, Kafka is mandatory.
- **Loki is losing a mode.** Simple Scalable goes away with Loki 4.0.
- **Native histograms are stable,** since Prometheus 3.8.
- **Charts have moved.** The charts for Grafana, Loki, and Tempo now live in the `grafana-community` repo.

The Boxes

PICTURE THIS

A comparison from everyday life. If you remember it, you can work out the topic from it.

```
kubectl get servicemonitors -A
```

Memory aid: A command to type, with a memory aid underneath.

PITFALL

The mistake almost everyone makes. And the one interviewers love to ask about.

IN THE INTERVIEW

A typical question?

An answer in two or three sentences. It really can be that short.

Your Study Plan

You have ...	Then read ...
1 evening	Chapters 1, 2, 4, 5, 7, 13, 17, 18, 20, 23, 28, and 30, then chapters 34 (the questions) and 36 (cheat sheet).
2 days	Day 1: Parts I, II, and III. Day 2: Part V, chapters 23 and 28, then chapter 34.
1 week	Days 1 and 2: Parts I through III. Day 3: Parts IV and V. Day 4: Part VI. Day 5: Part VII. Day 6: the exercises from chapter 35. Day 7: review the questions.

PART I

Fundamentals

Signals, the map, your lab, and the most important word: cardinality.

1

Why Observability?

Monitoring tells you that something is broken. Observability helps you find out why.

The Difference

Monitoring answers questions you asked in advance: Is the CPU above 90 percent? Is the Pod running? **Observability** lets you ask questions you never thought of in advance: Why are only orders from France through payment provider B slow, and only since the 2 p.m. deployment?

> **PICTURE THIS**
>
> *Monitoring is the warning light in your car. Observability is the garage that reads the fault memory, listens to the engine, and traces the problem back along its path.*

The Three Signals

Signal	Question	Form	Tool
Metrics	How much? How fast? Since when?	Numbers over time, cheap	Prometheus
Logs	What exactly happened?	Text events, expensive in bulk	Loki
Traces	Where did the time go?	A request's path through services	Tempo

A typical investigation runs in this order:

1. An **alert** from metrics says: error rate too high.
2. A **trace** shows: the `payment` service is the cause.
3. The **logs** from `payment` show: the certificate has expired.

A fourth signal is growing right now: **profiles**. They show which function is using the CPU. In the Grafana world, Pyroscope does this. This book leaves it out.

> **REMEMBER**
>
> The strength only comes from the connection. Shared labels like `namespace` and `pod` across all signals, plus the trace ID in the logs, turn three tools into one system.

Four Methods You Need to Be Able to Name

Method	Components	For whom
Golden Signals	Latency, traffic, errors, saturation	Every service
RED	Rate, Errors, Duration	Services, seen from outside
USE	Utilization, Saturation, Errors	Resources such as CPU and disk
SLI/SLO	Measure and target value	What users feel

IN THE INTERVIEW

What is the difference between monitoring and observability?

Monitoring checks known failure patterns against fixed thresholds. Observability means I can understand the inside of a system from its signals, even for failures I did not foresee. For that I need metrics, logs, and traces that are linked to each other.

IN THE INTERVIEW

Which signal do you use for what?

Metrics for alerts and trends, because they are cheap. Traces to find the guilty service. Logs for the specific detail. I go from coarse to fine.

KEY TAKEAWAYS

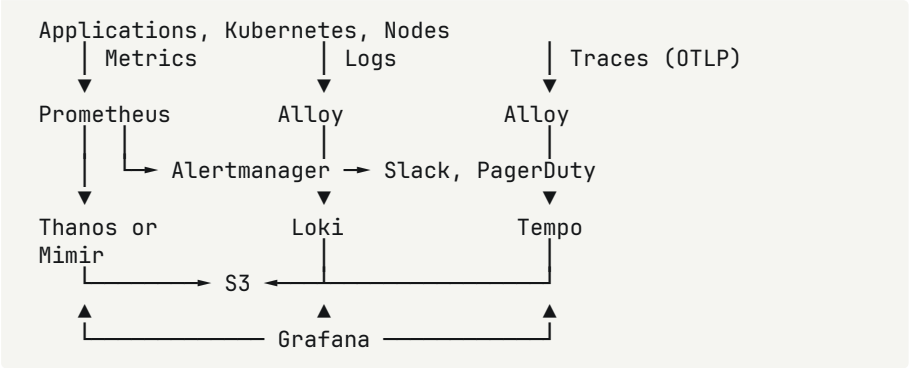
- Metrics, logs, traces: how much, what, where.
- RED for services, USE for resources.
- The value lies in the linking.

2

The Map

Eight names, one simple idea: collect, store, show, alert.

The Stack at a Glance



Who Does What?

Tool	Its job in one sentence
Prometheus	Fetches metrics, stores them briefly, evaluates rules.
Alertmanager	Groups alerts, suppresses noise, sends messages.
Grafana	Displays everything and connects the signals. Stores no measurement data itself.
Thanos	Attaches to Prometheus and puts blocks into S3 for the long term.
Mimir	A separate, multi-tenant metrics database with Remote Write and S3.
Loki	Stores logs cheaply, indexes only the labels.
Tempo	Stores traces cheaply, almost entirely in S3.
Alloy	Collector for all signals. Grafana's flavor of the OpenTelemetry Collector.
S3	Cheap, almost unlimited object storage for old data.

LGTM stands for Loki, Grafana, Tempo, Mimir. The pattern is the same for all of them:

> **PICTURE THIS**
>
> *A pub with a small bar and a big cellar. Up front (RAM, local disk) is whatever is needed right now. Everything older goes down to the cellar (S3), neatly labeled. The cellar is huge and cheap, but the trip down takes longer.*

Thanos and Mimir Are Alternatives

Many people mix this up. For long-term metrics you need **one** of the two, not both:

- **Thanos:** Prometheus stays at the center, and the blocks move to S3.
- **Mimir:** Prometheus just delivers via Remote Write. Mimir is the new center.

Chapter 22 helps you choose.

IN THE INTERVIEW

Describe an observability stack for Kubernetes.

Prometheus fetches metrics via the Operator, and alerts go to Alertmanager. Alloy collects logs and traces per Node and sends them to Loki and Tempo. Long-term metrics live in S3 via Thanos or Mimir, where Loki and Tempo store their data too. Grafana displays everything and links the signals.

KEY TAKEAWAYS

- Same pattern everywhere: hot data local, old data in S3.
- Thanos or Mimir, not both.
- Grafana only displays, the data lives elsewhere.

3

Your Lab

A kind cluster, one chart, and a small S3 store.

What You Need

`docker`, `kind`, `kubectl`, `helm`, and at least 8 GB of RAM for Docker.

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

Memory aid: kind means Kubernetes in Docker. A cluster named obs.

```
kubectl create namespace monitoring
```

Memory aid: Home to everything that does the watching.

kube-prometheus-stack

This chart installs the Prometheus Operator, Prometheus, Alertmanager, Grafana, kube-state-metrics, node-exporter, and ready-made rules and dashboards.

values-kps.yaml

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

Chapter 10 explains the three `NilUsesHelmValues` lines. They will save you a lot of grief.

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

Memory aid: kps is the release name. All object names derive from it.

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

Memory aid: Wait until everything is running.

```
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
```

Opening the UIs

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

Memory aid: Prometheus: localhost:9090.

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

Memory aid: Alertmanager: localhost:9093.

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

Memory aid: Grafana: localhost:3000, user admin.

> **PITFALL**
>
> On kind, alerts often report `etcd`, `kube-scheduler`, or `kube-controller-manager` as unreachable. There they listen only on localhost. In a lab, that is normal.

S3 in the Lab

For parts V through VII you need S3-compatible storage. The MinIO project is hardly suitable for that anymore: the free images have disappeared from the public registries, so the Mimir chart now uses a fork image. In the lab we use SeaweedFS.

s3-lab.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
```

Then create the buckets:

```
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
```

Memory aid: A throwaway Pod with the AWS CLI.

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

> **PITFALL**
>
> This is only a lab: `latest` image, no access control, data in `emptyDir`. After a restart, everything is gone.

KEY TAKEAWAYS

- `kube-prometheus-stack` ships metrics and alerts ready to go.
- The release name `kps` determines the object names.
- You will need S3 later, and SeaweedFS is enough for the lab.

4

Labels and Cardinality

The most important chapter: almost every cost problem is a label problem.

A Time Series

A time series is a metric name plus one combination of 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"}
```

Those are three time series, even though it is only one metric. **Cardinality** is the number of time series.

The Multiplication

```
  20 endpoints
×  5 methods
× 10 status codes
× 50 Pods
= 50,000 time series, in the worst case
```

Add a `user_id` label with 100,000 values and you get a number that will kill any Prometheus.

> **PICTURE THIS**
>
> *A label is like another column on an order form. Size times color gives you a few variants. But add the customer number, and suddenly every customer has their own drawer in the warehouse.*

Good and Bad Labels

Good: few, fixed values	Bad: unbounded values
`namespace`, `service`, `method`, `status`	`user_id`, `session_id`, `email`
`region`, `cluster`, `env`	`trace_id`, `request_id`
`route="/users/:id"`	`path="/users/48213"`

> **REMEMBER**
>
> This rule applies to all the tools: to Prometheus labels, to Loki stream labels, and to Tempo dimensions in derived metrics. High cardinality belongs in logs and traces, but not in labels.

Churn: The Hidden Cardinality

Every new Pod has a new name, and therefore new time series. The old ones stay in the head block until compaction. If you roll out ten times a day, you have a lot of time series even without evil labels. This is called **series churn**.

Finding What's Expensive

In Prometheus under **Status → TSDB Status**, or with PromQL:

```
# The ten metrics with the most time series
topk(10, count by (__name__) ({__name__=~".+"}))

# Which label is driving this metric?
count(count by (pod) (http_requests_total))

# How many time series does each job have?
sum by (job) (scrape_series_added)
```

> **PITFALL**
>
> The first query is expensive itself. Run it on a large system only with care, or use the TSDB status page.

Fixing It

1. Remove or normalize the label in the code. That is the best solution.
2. Throw it away at scrape time, with `metric_relabel_configs` (chapter 9).
3. Cap it at write time, with `sample_limit` or the limits in Mimir.

> **IN THE INTERVIEW**
>
> **What is cardinality, and why is it a problem?**
>
> Cardinality is the number of distinct time series, meaning every combination of metric and label values. Prometheus keeps active time series in RAM, so every new series costs memory and queries get slower. Unbounded labels like user IDs lead to an OOM kill.

IN THE INTERVIEW

Prometheus suddenly needs twice as much RAM. What do you do?

I look at the TSDB status page and compare the top metrics and labels. With `scrape_series_added` I find the job. Then I fix the application or throw away the metric via `metric_relabel_configs`, and I set a `sample_limit` as protection.

KEY TAKEAWAYS

- Time series are multiplication.
- No unbounded values in labels.
- Churn counts too: new Pods are new series.

PART II

Prometheus

Pull model, PromQL, storage, Operator, and rules.

5

The Prometheus Model

Prometheus fetches. Nobody has to deliver.

Pull Instead of Push

Applications expose their numbers as text under `/metrics`. Prometheus fetches that address every few seconds. This is called a **scrape**.

```
# HELP http_requests_total Number of requests
# TYPE http_requests_total counter
http_requests_total{method="GET",status="200"} 1027
http_requests_total{method="GET",status="500"} 3
```

> **PICTURE THIS**
>
> *The electricity meter in the basement. It just keeps counting. The meter reader comes by regularly and writes down the number. If they don't come, they know right away that something is wrong.*

Pull has two advantages: Prometheus notices immediately when a target is missing, and a slow application can't flood Prometheus.

For every target, Prometheus automatically creates the metric `up`. It is 1 if the scrape succeeded, and 0 otherwise.

Service Discovery

In Kubernetes, targets change all the time. So Prometheus asks the Kubernetes API which Pods and Endpoints exist. Nobody maintains IP addresses by hand.

Exporters

Software without `/metrics` gets a translator:

Exporter	Provides
node-exporter	CPU, RAM, disk, and network of a Node.
kube-state-metrics	State of Kubernetes objects.
cAdvisor (in the Kubelet)	CPU and RAM per container.
blackbox-exporter	Checks from outside: HTTP, TCP, DNS, ICMP.
postgres-exporter, etc.	Databases and other software.

Special Cases

- **Pushgateway:** For short batch jobs that disappear before the scrape. Only for that.
- **Agent mode:** With `--agent`, Prometheus only collects and forwards via Remote Write. Queries and rules are gone. This is ideal for edge clusters.
- **Remote Write:** Prometheus sends data onward, for example to Mimir.
- **OTLP:** With `--web.enable-otlp-receiver`, Prometheus also accepts OpenTelemetry metrics.

> **PITFALL**
>
> The Pushgateway forgets nothing. A value sent once stays there, even if the job has been dead for days. There, `up` only tells you something about the gateway itself.

IN THE INTERVIEW

Why pull and not push?

With pull, Prometheus knows which targets should exist and detects outages via `up`. Prometheus controls the load itself. For short-lived jobs or networks behind a firewall, there are the Pushgateway, agent mode, and Remote Write.

KEY TAKEAWAYS

- `/metrics` is text, and Prometheus fetches it.
- `up` is the first metric you check.
- Pushgateway for batch jobs only.

6

Metric Types

Four types, and one of them is the most common interview question.

The Four Types

Type	Behavior	Example
Counter	Only goes up. Starts at 0 after a restart.	`http_requests_total`
Gauge	Goes up and down.	`node_memory_MemAvailable_bytes`
Histogram	Counts values in buckets. Server-side quantiles.	`http_request_duration_seconds`
Summary	Calculates quantiles in the application.	`go_gc_duration_seconds`

> **REMEMBER**
>
> You never display a counter raw. You always use `rate()`. The raw value depends only on when the Pod last restarted.

Histogram vs. Summary

A histogram consists of several time series:

```
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` means **less or equal**. The buckets are cumulative: bucket 0.5 also contains everything from 0.1.

	Histogram	Summary
Quantile calculated	At query time	In the application
Can be aggregated across Pods	Yes	No
Accuracy	Depends on the buckets	Fixed in advance
Recommendation	Default	Special cases only

> **PICTURE THIS**
>
> *A summary is as if every branch store had already calculated its own average. From ten averages you can't get a true overall figure. A histogram delivers the tally sheets, which you can add up however you like.*

Native Histograms

Classic histograms create one time series per bucket, and you have to set the boundaries in advance. **Native histograms** store all the buckets in a single value and choose the boundaries automatically. That saves time series and is more accurate.

They have been stable since Prometheus 3.8. You still have to turn on scraping them explicitly:

```
global:
  scrape_native_histograms: true
```

> **PITFALL**
>
> The old flag `--enable-feature=native-histograms` has done nothing since version 3.9. Only with Prometheus 4 is scraping supposed to be active automatically.

> **IN THE INTERVIEW**
>
> **Histogram or summary?**
>
> Histogram, because only that can be aggregated across multiple Pods. A summary quantile can't be averaged in any meaningful way. Today I use native histograms where possible, because they need fewer time series and no fixed boundaries.

KEY TAKEAWAYS

- Counters only with `rate()`.
- Histograms can be aggregated, summaries can't.
- Native histograms are stable, but opt-in.

7

PromQL

The few patterns behind almost every query.

Selecting

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

An **instant vector** returns one value per time series. A **range vector** returns all the values in a time window per time series. You can't graph the latter directly, only pass it into functions like `rate()`.

rate and increase

```
rate(http_requests_total[5m])       # per second
increase(http_requests_total[1h])   # increase over the hour
irate(http_requests_total[5m])      # only the last two values
```

`rate()` detects counter resets on its own. The window should be at least four times the scrape interval. With a 30-second interval, that means at least `[2m]`.

> **PITFALL**
>
> First `rate`, then `sum`. Wrong: `rate(sum(x)[5m:])`. Right: `sum(rate(x[5m]))`. If you sum first, `rate` sees a drop after a Pod restart and misinterprets it.

Aggregating

```
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])))
```

The Most Important Recipes

Error rate:

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

p99 latency:

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

`le` has to stay in the aggregation, or it won't work. With native histograms, both `_bucket` and `le` go away.

Average latency:

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

Disk full in four hours?

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

A job is missing entirely:

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

Joining Two Metrics

When you calculate across metrics, the labels have to match. If they don't, `on` and `group_left` help:

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

> **PICTURE THIS**
>
> *`group_left` is like a VLOOKUP in Excel: many rows on the left each fetch a column from a small table on the right.*

PITFALL

`rate()` is an average over the window. Spikes shorter than the window disappear. And `histogram_quantile` only estimates: the result can never be more accurate than your buckets.

IN THE INTERVIEW

Why does `rate` sometimes return no data?

`rate` needs at least two values in the window. If the window is shorter than twice the scrape interval, or a scrape was missed, it stays empty. That's why I use at least four times the interval, and in Grafana preferably `$__rate_interval`.

IN THE INTERVIEW

What is the difference between `rate` and `irate`?

`rate` averages over the whole window and suits alerts and trends. `irate` looks only at the last two values. That shows spikes, but it flickers. For alerts I use `rate`.

KEY TAKEAWAYS

- First `rate`, then `sum`.
- Don't forget `by (le)`.
- Window at least four times the scrape interval.

8

The TSDB

Why Prometheus is so efficient and where it dies.

The Path of a Sample

```
Scrape ─► WAL (disk, for recovery)
      └─► Head Block (RAM, latest 2 hours)
             │ every 2 hours
             ▼
          Block on disk (immutable)
             │ Compaction
             ▼
          larger blocks ─► retention deletes old ones
```

Part	Job
WAL	Write-ahead log: protects the head from data loss in a crash.
Head block	The newest data in memory, queryable immediately.
Block	Directory with chunks, index, and `meta.json`. Default length two hours.
Compaction	Merges small blocks into larger ones.
Tombstones	Deletion markers. The blocks themselves don't change.

> **PICTURE THIS**
>
> *A cash ledger: every entry goes into the notebook first (WAL), then onto the desk (head). Every two hours, a finished binder goes on the shelf (block). Once on the shelf, nothing changes.*

These immutable blocks are exactly what make Thanos and Mimir possible. You can simply copy them to S3.

Doing the Math

According to the docs, Prometheus needs around 1 to 2 bytes per sample.

```
1,000,000 time series, every 30 seconds
= 33,333 samples per second
= 2.88 billion samples per day
```

```
× 1.5 bytes = about 4.3 GB per day
× 15 days   = about 65 GB
```

RAM is the bottleneck: every active time series costs a few kilobytes in the head.

Retention

Prometheus keeps data for 15 days by default. The `kube-prometheus-stack` chart sets 10 days.

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

Whichever limit is hit first applies.

High Availability

Prometheus has no built-in replication. The solution is plain: two identical instances scrape the same targets independently. Grafana or Thanos shows one of them. The values differ slightly, because the two scrape at different moments.

> **PITFALL**
>
> After an OOM, Prometheus replays the WAL at startup. That often takes even more RAM. The Pod dies again, and you get a `CrashLoop`. The cause is almost always cardinality. Avoid quick fixes that delete the WAL unless you accept the data loss.

> **PITFALL**
>
> NFS is not suitable for the TSDB. The docs recommend local, POSIX-compliant storage. On Kubernetes you use a block device via a PVC.

IN THE INTERVIEW

How does Prometheus store data?

New samples land in the WAL and in the head in RAM. Every two hours, an immutable block is created on disk. Compaction merges blocks, and retention deletes old ones. For long retention and multiple clusters I add Thanos or Mimir, because the local TSDB is not clustered.

IN THE INTERVIEW

How do you make Prometheus highly available?

Two replicas with identical configuration and different replica labels. Both scrape everything and send alerts to Alertmanager, which deduplicates. Thanos Query or Mimir's HA tracker remove duplicate data.

KEY TAKEAWAYS

- WAL, head, blocks, compaction, retention.
- Blocks are immutable, which is why they fit in S3.
- HA means: two Prometheus servers doing the same thing.

9

Scrape and Relabeling

Two steps that are among the most often confused.

A Raw Scrape Configuration

```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
```

The Difference

	`relabel_configs`	`metric_relabel_configs`
When	Before the scrape	After the scrape, before storage
Works on	Targets	Individual time series
Purpose	Choose targets, set labels	Throw away expensive metrics

> **PICTURE THIS**
>
> *`relabel_configs` is the guest list at the door. `metric_relabel_configs` is the bouncer at the buffet who decides what makes it onto the plate.*

Actions

Action	Effect
`keep`	Keeps only what matches.
`drop`	Throws away what matches.
`replace`	Sets a label, often from another one.
`labeldrop`	Removes a label from the series.
`labelmap`	Renames labels by regex.
`hashmod`	Distributes targets across multiple scrapers.

Throwing Away an Expensive Metric

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

PITFALL

Careful with `labeldrop`: if two series end up with identical labels, they collide. Prometheus then reports duplicate samples.

REMEMBER

Labels starting with `__` are internal and disappear after relabeling. That's why you have to copy `__meta_kubernetes_namespace` to `namespace` if you want to keep it.

IN THE INTERVIEW

What is the difference between `relabel_configs` and `metric_relabel_configs`?

`relabel_configs` runs before the scrape and decides which targets get scraped and which labels they receive. `metric_relabel_configs` runs afterward on every time series. I use it to throw away expensive metrics. They still cost network and parsing, though.

KEY TAKEAWAYS

- Before the scrape: targets. After the scrape: series.
- `sample_limit` is your fuse.
- `__meta_` labels need rescuing.

10

The Prometheus Operator

Prometheus as Kubernetes objects, and the most famous pitfall.

The CRDs

CRD	Meaning
`Prometheus`	A Prometheus instance.
`PrometheusAgent`	Prometheus in agent mode.
`Alertmanager`	An Alertmanager instance.
`ServiceMonitor`	Scrape targets via a Service.
`PodMonitor`	Scrape targets directly via Pods.
`Probe`	Blackbox checks.
`ScrapeConfig`	Free-form scrape configuration, also for targets outside the cluster.
`PrometheusRule`	Alerting and recording rules.
`AlertmanagerConfig`	Routing per namespace.
`ThanosRuler`	Thanos Ruler as an object.

Getting a Service Scraped

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
```

> **PICTURE THIS**
>
> *A chain of three tags: Prometheus looks for ServiceMonitors with `release: kps`. The ServiceMonitor looks for Services with `app: api`. The Service looks for Pods with `app: api`. If one link breaks, the target is invisible and nobody complains.*

> **PITFALL**
>
> This is the most famous pitfall: `port: metrics` means the **name** of the Service port, not a number. And without the matching label `release: kps`, Prometheus silently ignores the ServiceMonitor.

Why the Label Is Needed

Without further settings, the chart looks only for ServiceMonitors with the release label. With this line, Prometheus takes all of them:

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

The same goes for `podMonitorSelector…`, `ruleSelector…`, `probeSelector…`, and `scrapeConfigSelector…`.

Checking

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

Memory aid: What is Prometheus looking for?

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

Memory aid: Does it have the right label?

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

Memory aid: Does the Service have any Pods at all?

Then open **Status → Targets** in Prometheus. If the target isn't in the list, it's the label. If it's listed as `down`, it's the network or the application.

Permissions

Prometheus needs RBAC for the target namespace. The chart gives it cluster-wide read access. With self-built instances, the error `403 forbidden` in the log is common.

IN THE INTERVIEW

A ServiceMonitor isn't being picked up. How do you proceed?

I check the chain: Does the label match the `serviceMonitorSelector`, and does the namespace match the `serviceMonitorNamespaceSelector`? Does the ServiceMonitor's selector match the Service, and does the port have the right name? Does the Service have Endpoints? Finally, I look at the targets page and the Operator log.

IN THE INTERVIEW

ServiceMonitor or PodMonitor?

ServiceMonitor if there is a Service, which is the normal case. PodMonitor for workloads without a Service, such as DaemonSets or Jobs. For targets outside the cluster I use `ScrapeConfig`.

KEY TAKEAWAYS

- Selector chain: Prometheus, ServiceMonitor, Service, Pod.
- `port` is a name.
- Target missing entirely: check the labels.

11

Rules

Recording rules make things fast. Alerting rules wake you up.

Recording Rules

They precompute expensive queries at regular intervals and store the result as a new metric.

```
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]))
```

The naming convention is `level:metric:operation`.

Alerting Rules

```
    - 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: "Error rate above 5 percent"
            description: "Currently {{ $value |
humanizePercentage }}"
```

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

The Life of an Alert

```
inactive ──► pending (condition true, for is running)
         ──► firing  (for elapsed, goes to Alertmanager)
         ──► resolved
```

Field	Meaning
`for`	How long the condition must hold. Filters out brief blips.
`keep_firing_for`	How long the alert stays active after the condition ends. Prevents flapping.
`labels`	For routing. They change the alert's identity.
`annotations`	For humans. Text, links, values.

PITFALL

Don't put changing values in `labels`. Every new value creates a new alert. The current number belongs in `annotations`.

REMEMBER

When Prometheus restarts, it restores running `for` timers from the `ALERTS_FOR_STATE` series. After a long outage this only works to a limited extent, by default up to one hour.

Testing

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

Memory aid: Check the syntax.

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

Memory aid: Test rules with made-up data.

test.yaml

```
rule_files: [rules.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 }
```

IN THE INTERVIEW

What are recording rules for?

They compute expensive queries in advance. Dashboards get faster, and alerts get simpler. And with long retention I can query the precomputed series instead of reading millions of samples every time.

IN THE INTERVIEW

What does `for` do in a rule?

The alert only goes to `firing` after this time. That filters out brief blips, but it also delays the notification. For real total outages I keep it short, for slow trends longer.

KEY TAKEAWAYS

- Recording rules save compute time.
- `labels` route, `annotations` explain.
- Test rules with `promtool test rules`.

12

Measuring Kubernetes

Where the numbers come from and which ones you need to know.

Four Sources

Source	Provides	Prefix
kube-state-metrics	How the API sees the objects: desired and current state.	`kube_`
cAdvisor	How much the containers actually use.	`container_`
node-exporter	How the host is doing.	`node_`
Kubelet, API server	How the components themselves are doing.	`kubelet_`, `apiserver_`

> **PITFALL**
>
> `metrics-server` is something else. It keeps only current values in RAM and serves `kubectl top` and the HPA. It's no good for history and alerts. Prometheus doesn't replace it automatically either: for the HPA with custom metrics you need `prometheus-adapter` or KEDA.

The Most Important Queries

CPU per Pod:

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

Memory per Pod:

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

`container_memory_working_set_bytes` is the right value: the Kubelet bases evictions on it, and it is close to what the OOM killer sees.

CPU throttling:

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

Pods with restarts:

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

OOM kills:

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

Missing replicas:

```
kube_deployment_spec_replicas
  - kube_deployment_status_replicas_available > 0
```

Requests vs. actual usage:

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

> **REMEMBER**
>
> The filter `container!=""` removes the Pod's total line from cAdvisor. Without it you count everything twice.

IN THE INTERVIEW

How do you tell that a Pod is being CPU throttled?

The CPU graph looks calm because the limit caps it, but latency rises. I look at the ratio of `container_cpu_cfs_throttled_periods_total` to `container_cpu_cfs_periods_total`. If it's high, I raise the limit or remove it.

IN THE INTERVIEW

Which source do you use for what?

For object state kube-state-metrics, for usage cAdvisor, for the host node-exporter. `metrics-server` is only for autoscaling and `kubectl top`.

KEY TAKEAWAYS

- `kube_` state, `container_` usage, `node_` host.
- Working set is the right memory value.
- `container!=""` prevents double counting.

PART III

Alerting

Alertmanager, good alerts, SLOs, and error budgets.

13

Alertmanager

Prometheus decides that something is on fire. Alertmanager decides who hears about it.

What It Does

- **Grouping:** 200 Pods fail, and you get one message.
- **Inhibition:** If the Node is dead, the Pod alerts stay quiet.
- **Silencing:** For maintenance.
- **Routing:** Team A to Slack, critical to PagerDuty.
- **Deduplicating:** Two Prometheus servers report the same thing, and you hear it once.

> **PICTURE THIS**
>
> *A hospital switchboard. A hundred calls about the same accident get merged into one emergency, put through to the right ward, and during a planned drill not put through at all.*

The Configuration

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"

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]
```

The Three Timers

Field	Default	Meaning
`group_wait`	30 s	How long to collect before the first message of a new group goes out.
`group_interval`	5 min	How long to wait before reporting new alerts in an existing group.
`repeat_interval`	4 h	How long to wait before reporting unchanged alerts again.

> **REMEMBER**
>
> The route tree rule: the first matching route wins, unless it sets `continue: true`. The root route catches everything and must not have any matchers.

> **PITFALL**
>
> The old fields `match` and `match_re`, as well as `source_match` and `target_match`, are deprecated. Use `matchers`, `source_matchers`, and `target_matchers`. Old examples on the web still use them.

High Availability

Alertmanager instances form a cluster via gossip (port 9094). Prometheus sends to **all** instances, and they agree among themselves who sends.

> **PITFALL**
>
> Don't put a load balancer between Prometheus and Alertmanager. Prometheus has to know all instances directly. The Operator handles this via the Service and the Endpoints.

Per Namespace: AlertmanagerConfig

With the Operator, every team can write its own routing:

```
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
```

The Operator automatically adds a matcher on the namespace. That way a team can only route its own alerts. Which objects apply is controlled by `alertmanagerConfigSelector`.

Checking

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

Memory aid: am for Alertmanager, plus tool. Checks the file.

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

Memory aid: Where would this alert go?

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

Memory aid: Two hours of quiet, with a reason.

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

Memory aid: What's firing right now?

IN THE INTERVIEW

How do you avoid alert storms?

With `group_by` I bundle alerts, for example by alert name and namespace. With inhibition, follow-up alerts stay quiet once the cause has been reported. Add sensible timers and only actionable alerts.

IN THE INTERVIEW

An alert fires in Prometheus but never reaches Slack. How do you check that?

First I look at `/alerts` in Prometheus to see whether it's `firing`. Then in Alertmanager, whether it arrives there and whether a silence or inhibition applies. With `amtool config routes test` I check the route. Finally, I look at `alertmanager_notifications_failed_total` and the log.

IN THE INTERVIEW

Why no load balancer in front of Alertmanager?

Prometheus should reach every instance. The instances coordinate via gossip and deduplicate. If a load balancer sent to only one instance, an alert would depend on that one instance.

KEY TAKEAWAYS

- Group, inhibit, silence, route.
- First matching route wins.
- HA via gossip, Prometheus sends to all.

14

Good Alerts

Every alert must trigger an action. Otherwise it's noise.

The Rules

- Alert on **symptoms**, not causes. Users feel errors, not CPU.
- Every alert has a **runbook**.
- Every notification demands an action.
- If it can wait, it's a ticket and not a page.
- If you get woken up too often, you start ignoring alerts. This is called **alert fatigue**.

Bad: cause	Good: symptom
CPU above 90 percent	p99 above 500 ms for 10 minutes
A Pod restarted	Error rate above 1 percent
RAM above 80 percent	Queue growing for 15 minutes

SLI, SLO, Error Budget

Term	Meaning
SLI	The measure. For example: share of requests under 300 ms.
SLO	The target. For example: 99.9 percent over 30 days.
SLA	The contract. If it's broken, you get money back.
Error budget	The allowed remainder: 0.1 percent, or 43.2 minutes per 30 days.

> **PICTURE THIS**
>
> *The error budget is an allowance. As long as some is left, you may roll out risky changes. Once it's spent, it's time to save: no new features, stability first.*

Burn Rate

The burn rate tells you how fast you are using up the budget. 1 means: it lasts exactly until the end of the 30 days. 14.4 means: 2 percent of the budget is gone in one hour.

```
error rate / allowed error rate = burn rate
1.44 percent / 0.1 percent = 14.4
```

The Google SRE Workbook recommends alerts over two windows:

Long window	Short window	Burn rate	Budget used
1 h	5 min	14.4	2 percent
6 h	30 min	6	5 percent
3 days	6 h	1	10 percent

The long window proves that the problem is serious. The short one makes sure the alert ends quickly once it's fixed.

```
- 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
```

Tools like **Sloth** or **Pyrra** generate such rules from a short SLO description.

The Watchdog

What happens if the alerting system itself is dead? Then everything stays quiet. So `kube-prometheus-stack` ships the `Watchdog` alert, which **always** fires. You forward it to an external service that raises the alarm as soon as the messages **stop coming**. This is called a **dead man's switch**.

> **PICTURE THIS**
>
> *The dead man's switch in a locomotive. The driver has to press it regularly. If the signal stops, the train brakes.*

IN THE INTERVIEW

When should an alert wake a human?

Only if users are affected, or soon will be, and a human can do something. Everything else becomes a ticket or a dashboard. I alert on symptoms and use the causes for troubleshooting.

IN THE INTERVIEW

What is a multi-window burn rate?

I compare the error rate with the allowed budget, over a long and a short window. Both must be exceeded. That way the alert reacts quickly to big problems, doesn't fire on brief spikes, and ends promptly after the fix.

IN THE INTERVIEW

Who monitors the monitoring?

An always-firing Watchdog alert goes to an external dead man's switch service. If it stops arriving, I get notified. On top of that come alerts on the pipeline itself, see chapter 31.

KEY TAKEAWAYS

- Symptoms, not causes.
- Burn rate over two windows.
- Send the Watchdog outside.

PART IV

Grafana

Datasources, dashboards, and everything as code.

15

Grafana

The window onto all your data, but not the storage.

What Grafana Is

Grafana stores no measurements. It queries datasources and draws the result. All it stores itself is dashboards, users, rules, and settings.

Operations	Note
One Pod, SQLite	Default. Enough for the lab.
Multiple Pods	Needs a shared database, such as PostgreSQL.
Dashboards	Belong in Git, not just in the database.

> **PITFALL**
>
> The `grafana/grafana` chart is deprecated and moving to `grafana-community/grafana`. Old guides keep working for a while, but they won't get new versions anymore. The `kube-prometheus-stack` chart already uses the community source.

Datasources as Code

Don't click datasources together. In `kube-prometheus-stack`, a sidecar collects ConfigMaps with the label `grafana_datasource: "1"`:

`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
```

> **PITFALL**
>
> In Grafana's provisioning files, `$` is a substitution character for environment variables. If you want to write a real `$`, as in `$${__value.raw}`, double it.

The UI

- **Explore:** Free-form searching without a dashboard. Start your troubleshooting here.
- **Dashboards:** Fixed views.
- **Alerting:** Grafana's own alerts. They are an alternative to rules in Prometheus.
- **Drilldown:** Click instead of writing queries, for metrics, logs, and traces.

Grafana Alerting or Prometheus Rules?

Prometheus rules	Grafana Alerting
As code, easy in Git	Also via provisioning, but harder
Run close to the data	Depend on Grafana and the datasource
Metrics only	Across all datasources

Rule of thumb: infrastructure alerts stay in Prometheus and Alertmanager. Use Grafana Alerting for datasources that can't do rules.

> **IN THE INTERVIEW**
>
> **How do you run Grafana in a highly available way?**
>
> Multiple replicas behind a Service with a shared database such as PostgreSQL instead of SQLite. Dashboards and datasources come from Git via provisioning, so Grafana itself stays replaceable. With Grafana Alerting, the instances also have to find each other as a cluster.

KEY TAKEAWAYS

- Grafana only displays, the data lives elsewhere.
- Datasources via ConfigMap or provisioning.
- Infrastructure alerts stay in Prometheus.

16

Dashboards That Help

Fewer panels, a clear story, everything from Git.

The Layout

A good dashboard answers one question from top to bottom:

1. Top: Is the service healthy? Rate, errors, latency.
2. Middle: Who is affected? Breakdown by route and Pod.
3. Bottom: Why? CPU, RAM, dependencies.

> **PICTURE THIS**
>
> *A newspaper: headline, summary, details. If you don't know after ten seconds whether everything is fine, you have a bad dashboard.*

Variables

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

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

In the query:

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

> **REMEMBER**
>
> Use `$__rate_interval` instead of fixed windows. Grafana calculates a suitable window from the zoom and the scrape interval. For this, the scrape interval in the datasource has to be right.

Dashboards as Code

The sidecar turns every labeled ConfigMap into a dashboard:

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

Other ways:

- **Grafana Operator:** CRDs such as `GrafanaDashboard`.
- **Grafonnet:** Generate dashboards in Jsonnet.
- **Terraform:** The Grafana provider manages folders and permissions.

Staying Fast

- Expensive queries become recording rules.
- Few panels, short default time ranges.
- No regex across all series.
- With Thanos: automatic downsampling uses the resolution of the time range.

> **PITFALL**
>
> A dashboard with 40 panels that reloads every 5 seconds and is kept open by 30 people can bring a Prometheus to its knees. A dashboard is load.

IN THE INTERVIEW

How do you keep dashboards maintainable?

Dashboards live versioned in Git and are rolled out via ConfigMap or Operator. I use variables instead of copies per service and build from overview to detail. Manual changes are just an experiment until they're back in Git.

KEY TAKEAWAYS

- Top to bottom: healthy, who, why.
- `$__rate_interval` instead of fixed windows.
- Dashboards belong in Git.

PART V

Long-Term Metrics and S3

Why object storage, Thanos, Mimir, and the decision between them.

17

S3 for Old Data

Cheap, huge, slow: the cellar of the whole stack.

Why S3?

Local disks are fast but expensive, and every resize is work. Object storage has different strengths:

- You don't have to plan capacity.
- The provider takes care of redundancy.
- Pods stay almost stateless and are easy to replace.
- All replicas see the same data.

In exchange, it is slow and bills per request. That's why all the tools have caches.

Tool	Stores in S3	Writes when
Thanos	TSDB blocks	Sidecar: every 2 hours
Mimir	TSDB blocks per tenant	Ingester: every 2 hours
Loki	Chunks and TSDB index	When the chunk is full or times out
Tempo	Parquet blocks	Continuously, controlled by the block limits

> **REMEMBER**
>
> Ceph speaks S3 too, via the RADOS Gateway. So you don't need a cloud. The protocol is the standard, not Amazon.

Rules for Buckets

- **Separate buckets per tool.** Mimir won't start if blocks, ruler, and Alertmanager share the same bucket and prefix.
- **Region matching the cluster.** Otherwise you pay for transfer and wait longer.
- **Create the buckets yourself.** Mimir and Tempo don't create them.

- **Delete incomplete multipart uploads.** A lifecycle rule with one day is enough.

Who Deletes Old Data?

Always the tool, not S3. This is the most important rule in this chapter.

Tool	Setting	Default
Thanos	Compactor: `--retention.resolution-raw`, etc.	`0d`, meaning forever
Mimir	`compactor_blocks_retention_period`	never delete
Loki	`retention_period` and `retention_enabled`	never delete
Tempo	`compaction.block_retention`	336 h, meaning 14 days

> **PITFALL**
>
> An S3 lifecycle rule that simply deletes everything after 30 days destroys the tools. Index and data drift apart, and queries fail. Loki allows lifecycle rules only as a safety net: limited to chunk prefixes, longer than the retention, and never for index objects.

Storage Classes

Class	Verdict
Standard	The safe choice.
Intelligent-Tiering	Possible, as long as the archive tiers stay off.
Standard-IA	Access fee and minimum duration. Compaction reads a lot and deletes early.
Glacier	No. The tools need immediate access to the objects.

> **PICTURE THIS**
>
> *Glacier is a warehouse that takes hours to deliver. But your query is standing at the bar and waits only seconds. The cellar has to stay walkable.*

Access Without Keys

Don't put access keys in Secrets if you can avoid it. On EKS you use **IRSA** or **EKS Pod Identity**: the ServiceAccount gets an IAM role.

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

The role only needs access to its own bucket:

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

Tempo also needs `s3:GetObjectTagging` and `s3:PutObjectTagging`.

Costs

You pay for three things: stored gigabytes, requests, and data transfer. With many small objects, requests quickly get more expensive than the storage. Compaction helps, because it turns many small blocks into a few large ones.

Access from the same region needs a VPC gateway endpoint, otherwise it goes through a NAT gateway, and that costs per gigabyte.

> **PITFALL**
>
> Versioning on the bucket: the tools delete objects constantly. With versioning on, all versions stick around and you keep paying. If you need it, set a lifecycle rule for old versions.

IN THE INTERVIEW

Why object storage for observability data?

It is cheap, practically unlimited, and redundant. The data is almost immutable, so it fits objects well. This makes Pods nearly stateless. Caches and compaction absorb the downside, the latency.

IN THE INTERVIEW

May I delete old data with an S3 lifecycle rule?

No, as a rule not. Retention belongs in the tool, because the tool knows the index and blocks. A lifecycle rule deletes blindly and leaves a broken index behind. The only sensible rules are for aborted multipart uploads and old versions.

IN THE INTERVIEW

Can I move old data to Glacier?

Not if I still want to query it. The tools need immediate access. For compliance I export separately into a dedicated archive and treat that as a separate process.

KEY TAKEAWAYS

- The tool deletes, not S3.
- Separate buckets, same region, IAM role.
- No Glacier for queryable data.

18

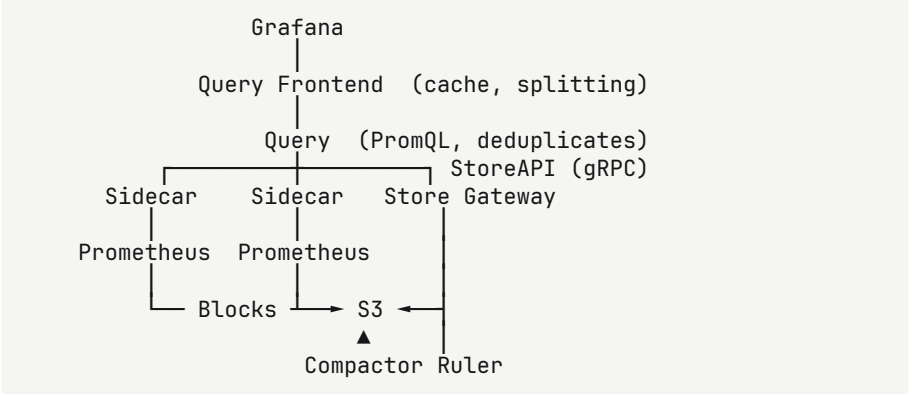
Understanding Thanos

Prometheus stays, and gets a long-term memory.

The Problems Thanos Solves

- **Long-term storage:** Blocks live in S3.
- **Global view:** One query across all clusters.
- **HA without duplicates:** Two Prometheus servers show one curve.

The Components



Component	Job
Sidecar	Runs next to Prometheus, uploads blocks, and serves current data.
Store Gateway	Makes blocks in S3 queryable.
Query	Stateless. Queries all StoreAPIs and deduplicates.
Query Frontend	Caches results and splits long queries.
Compactor	Compacts, creates downsampling, applies retention.
Ruler	Evaluates rules via Query, so globally too.
Receive	Accepts Remote Write. The alternative to the Sidecar.

> **PICTURE THIS**
>
> *Query is a librarian with a telephone. They call the branches (Sidecars) and the central archive (Store Gateway), put the results together, and cross out duplicates.*

Sidecar or Receive?

	Sidecar	Receive
Data flow	Query to Prometheus	Prometheus sends via Remote Write
Network	Query has to reach every Sidecar	Outbound from the cluster only
Operations	Simple	Stateful, hashring, replication
Good for	Few clusters, same network	Many clusters, edge, tenants

Receive distributes series across a **hashring**. The recommended algorithm is `ketama`. With `--receive.replication-factor=3`, a write succeeds as soon as two copies exist.

External Labels: Mandatory

Thanos tells blocks apart by their external labels. Every Prometheus instance needs a globally unique combination.

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

With the Operator, the labels `prometheus` and `prometheus_replica` are created automatically. You add the `cluster` label under `externalLabels`.

Query ignores the replica label when deduplicating:

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

> **PITFALL**
>
> Without the `cluster` label, blocks from two clusters look identical. Then they collide, and the Compactor can scramble the data.

IN THE INTERVIEW

How does Thanos work?

A Sidecar next to Prometheus uploads finished blocks to S3 every two hours. The Store Gateway makes them queryable. Thanos Query queries Sidecars and the Store Gateway via the StoreAPI and removes HA duplicates using the replica label. The Compactor compacts the blocks, creates downsampling, and applies retention.

IN THE INTERVIEW

When Thanos Receive instead of Sidecar?

When the clusters aren't directly reachable or a great many clusters deliver data. Then Prometheus pushes out via Remote Write. In exchange, I run a stateful hashring with replication.

KEY TAKEAWAYS

- Sidecar uploads, Store Gateway reads, Query connects.
- Compactor maintains the bucket.
- External labels must be unique.

19

Operating Thanos

Turn on the Sidecar, configure the Compactor properly, understand downsampling.

Step 1: Bucket Configuration

objstore.yml

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

With `aws_sdk_auth: true`, the credentials come from the IAM role, not from the file. In the lab you enter the lab S3 instead:

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

> **PITFALL**
>
> Mixing `access_key` from the file with `secret_key` from another source is not allowed. And `insecure: true` turns off TLS. That's for tests only.

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

Memory aid: The file goes into a Secret.

Step 2: Turning On the Sidecar

values-kps.yaml (addition)

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

The Operator sets the block duration to exactly two hours and turns off local compaction, so that the Compactor in S3 is solely responsible.

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

Memory aid: Same command, new values.

> **PITFALL**
>
> Never change the block durations of a Prometheus with a Sidecar by hand. With differing `min-block-duration` and `max-block-duration`, Prometheus compacts locally and the upload stalls.

Step 3: Query, Store, Compactor

Thanos has no official Helm chart. There are the manifests from `kube-thanos`, the Bitnami chart, or your own Deployments.

> **PITFALL**
>
> Bitnami severely restricted its free image offering in 2025. With guides that use the Bitnami chart, check where the images really come from.

The core of a Query Deployment:

```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
```

> **REMEMBER**
>
> Without `--wait`, the Compactor exits after one pass. So as a Deployment it needs `--wait`, as a CronJob it doesn't.

Downsampling

When	What
Raw data older than 40 hours	5-minute blocks are created.
5-minute blocks older than 10 days	1-hour blocks are created.

The most common misconception is: downsampling saves storage. It doesn't. Two more blocks get **added**, and if you keep all resolutions, you need up to three times the space.

The point is fast queries over long time ranges. A one-year graph doesn't have to read billions of raw values.

> **PICTURE THIS**
>
> *Downsampling is a map at several scales. The coarse map doesn't replace the city maps, it joins them. But for the drive to Rome, the coarse one is enough.*

> **PITFALL**
>
> If you set `--retention.resolution-raw` shorter than the 40 hours, 5-minute blocks never get created. For a plan like "30 days raw, 180 days coarse," all three values have to be aligned with each other. And `rate()` over a window shorter than the resolution stays empty.

Singleton

Only **one** Compactor may run per block stream in a bucket. Two at once destroy data. You can only scale through separate block streams, meaning label sharding.

The Compactor also needs disks. The docs mention around 100 GB for a medium-sized bucket. In the worst case you need room for twice the largest compaction range.

> **PITFALL**
>
> On data errors, the Compactor halts but keeps running and reports `Running`. Only the metric `thanos_compact_halted` shows it. And since retention only runs after compaction, nothing gets deleted anymore, and the bucket grows without warning.

Checking

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

Memory aid: List all blocks in the bucket.

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

Memory aid: View blocks graphically.

IN THE INTERVIEW

Does downsampling save storage?

No. Thanos creates additional blocks, so storage actually grows. The benefit is fast queries over long time ranges. I only save costs through retention, by deleting raw data earlier than the coarse resolutions.

IN THE INTERVIEW

The Thanos bucket keeps growing even though retention is set. Why?

Most likely the Compactor is no longer running cleanly. I check `thanos_compact_halted` and the log. Retention only runs after compaction and downsampling. If the Compactor halts, nothing gets deleted.

KEY TAKEAWAYS

- The Compactor does retention, and the default is forever.
- Downsampling speeds things up but saves nothing.
- Only one Compactor per block stream. Monitor `thanos_compact_halted`.

20

Understanding Mimir

One central metrics system for many clusters and tenants.

The Idea

Prometheus keeps collecting, but delivers everything to Mimir via Remote Write. Mimir stores, computes, and answers queries. It is fully PromQL-compatible and multi-tenant.

Two Architectures

Since Mimir 3.0 there are two ways. The **ingest storage architecture** is the preferred one and the default in the chart. The **classic** one continues to be supported.

	Classic	Ingest storage
Buffer	None	Kafka
Write succeeds	Quorum: 2 of 3 ingesters	Kafka has acknowledged
Replication	RF 3 between ingesters	2 ingesters per partition
Reading and writing	Meet in the ingester	Are separate

Write Path with Ingest Storage

```
Prometheus → Distributor → Kafka → Ingester → S3
             checks limits  Partition  reads along,
             picks partition           builds blocks
```

A write request counts as successful as soon as Kafka acknowledges it. So a failed ingester doesn't slow down writes. After the restart, it catches up on the backlog from Kafka.

Read Path

```
Grafana → Query Frontend → Query Scheduler → Querier
                                               ├→ Ingester
(recent)
```

```
                                                    └→ Store
Gateway (S3)
```

Component	Job
Distributor	Accepts Remote Write, checks limits, splits up series.
Ingester	Holds recent data, writes blocks every 2 hours.
Query Frontend	Splits queries by time and shards, caches.
Query Scheduler	Queue, fair per tenant.
Querier	Executes the queries.
Store Gateway	Reads blocks from S3.
Compactor	Compacts, deletes per retention. Horizontally scalable.
Ruler	Evaluates rules.
Alertmanager	Multi-tenant Alertmanager.

> **PICTURE THIS**
>
> *Kafka is a conveyor belt in a parcel depot. The front desk puts parcels on the belt and is done. The sorters grab them when they can. If one drops out, the parcel stays on the belt and doesn't get lost.*

Tenants

Every request carries the header `X-Scope-OrgID`. Each tenant gets its own TSDB, with its own limits and its own retention.

> **PITFALL**
>
> Multi-tenancy is no substitute for authentication. Mimir believes the header. Authentication belongs in a reverse proxy in front of it, which sets the header.

Deduplicating HA Pairs

Two Prometheus servers deliver the same data. The **HA tracker** picks one sender per cluster and discards the other. It recognizes the pairs by the labels `cluster` and `__replica__`. You turn this on per tenant with `accept_ha_samples`.

IN THE INTERVIEW

Why does Mimir use Kafka?

In the classic architecture, the ingester serves writes and reads at the same time. So many queries can interfere with writes. Kafka separates the two paths as a robust buffer. The distributor is done as soon as Kafka has acknowledged, and the ingesters read at their own pace.

IN THE INTERVIEW

How does Mimir protect itself from a noisy tenant?

Through per-tenant limits on ingestion, time series, and queries, plus the fair queue in the Query Scheduler. Shuffle sharding can also restrict tenants to subsets of the Pods.

KEY TAKEAWAYS

- Remote Write in, PromQL out, S3 underneath.
- Ingest storage separates writing and reading with Kafka.
- The tenant comes from `X-Scope-OrgID`, authentication is your job.

21

Operating Mimir

Chart, S3, limits, and retention.

Modes

Mimir is one binary. The `-target` parameter determines what runs. `-target=all` starts everything in one process and is the entry point. The production form is microservices. Mimir 3 removed the read-write mode.

Installation

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

Memory aid: The Mimir chart still lives in the main 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: company-mimir-blocks
    alertmanager_storage:
      s3:
        bucket_name: company-mimir-alertmanager
    ruler_storage:
      s3:
        bucket_name: company-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
```

Memory aid: The chart is called distributed because it rolls out microservices.

> **PITFALL**
>
> Without further values, the chart brings its own Kafka and an S3 substitute meant for demos only. In production you need a serious Kafka and at least as many partitions as the maximum number of ingesters. According to the chart, the topic default of 100 partitions is only a demo setting.

Connecting Prometheus

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

Grafana gets a Prometheus datasource with the path `/prometheus`:

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

The Service names depend on the release. Check them with `kubectl -n mimir get svc`.

Retention

Metrics in S3 are never deleted by default. You override this per tenant in the runtime configuration:

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

> **REMEMBER**
>
> Mimir only deletes whole blocks of a tenant. It doesn't support deleting individual series or retention per series.

Mimir also has no downsampling like Thanos. It relies on sharding and caches. If you want to speed up a one-year graph, use recording rules.

The Most Important Limits

Limit	Default	Effect
`ingestion_rate`	10,000/s	Samples per second per tenant.
`max_global_series_per_user`	150,000	Active time series per tenant.
`max_label_names_per_series`	30	Labels per series.

The time series limit is low for real clusters. Expect to raise it.

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

With its `reason` label, the metric `cortex_discarded_samples_total` shows why samples were discarded.

Notes on Querying

Since version 3, Mimir uses its own Mimir Query Engine (MQE) by default. It is more economical, but should deliver results equivalent to the Prometheus engine. If in doubt, set `-querier.query-engine=prometheus`.

> **IN THE INTERVIEW**
>
> **Data has stopped arriving in Mimir. What could be the cause?**
>
> I check the Remote Write metrics in Prometheus for backlog and errors. 429 responses point to limits, and then `cortex_discarded_samples_total` shows the reason. With 5xx I look at the distributors and Kafka. If reads return no data, I check the tenant header.

> **IN THE INTERVIEW**
>
> **How does Prometheus handle Remote Write during outages?**
>
> Prometheus buffers via the WAL and retries. If the outage lasts longer than the WAL buffer, data is lost. On 429 and 5xx it retries, on other 4xx it discards the data.

KEY TAKEAWAYS

- Chart: `grafana/mimir-distributed`, Kafka is part of the architecture.
- Retention per tenant, default forever.
- Limits are low, and `cortex_discarded_samples_total` shows why.

22

Thanos or Mimir?

The question that comes up in almost every senior interview.

The Comparison

	Thanos	Mimir
Approach	Extends Prometheus	Separate central database
Data arrives via	Sidecar or Receive	Remote Write
Queries	Distributed via StoreAPI	Central in the Mimir cluster
Tenants	Via Receive and headers, limited	Built in, with limits
Downsampling	Yes	No
Additional dependencies	Few	Kafka, caches
Operational effort	Low to medium	High
License	Apache 2.0	AGPL-3.0

Decision Guide

- **Few clusters, small team, Prometheus already running:** Thanos with Sidecar.
- **Many teams, platform as a product, limits per team:** Mimir.
- **Very many short series, long retention:** Leaning toward Mimir with sharding.
- **Yearly trends for a few metrics:** Thanos with downsampling or recording rules.

> **PICTURE THIS**
>
> *Thanos is a network of branches with a shared warehouse. Each branch stays independent. Mimir is the central warehouse: everyone delivers there, and everything is managed there.*

REMEMBER

The answer is never “X is better.” Name the requirements: number of clusters, tenants, data volume, team size, existing Kafka, and whether the networks are separated.

IN THE INTERVIEW

Thanos or Mimir?

It depends on the requirements. Thanos fits if I want to build on existing Prometheus servers and keep operations small. Mimir fits if I offer a central multi-tenant platform with hard limits and the team can carry Kafka and many components.

IN THE INTERVIEW

Why not just set Prometheus retention to two years?

Because Prometheus remains a single server: one disk fails and the data is gone. There is no global view and no deduplication. Queries over long time ranges load the same process that is supposed to be scraping.

KEY TAKEAWAYS

- Thanos extends, Mimir replaces.
- Decide by teams, clusters, and operational load.
- Only Thanos has downsampling, Mimir does tenants better.

PART VI

Logs

Loki, LogQL, Alloy, and operations with S3.

23

Understanding Loki

Prometheus thinking for logs: index only the labels, not the content.

The Basic Idea

Elasticsearch indexes every term. That is powerful but expensive. Loki indexes only the labels and stores the text compressed in S3. When searching, Loki filters the text only at query time.

> **PICTURE THIS**
>
> *An archive of labeled boxes. The labels are in the directory. Nobody knows the contents of the boxes by heart. You pick the right boxes and then leaf through them.*

Terms

Term	Meaning
Stream	All log lines with exactly the same label combination.
Chunk	Compressed package of lines from one stream, in S3.
Index	TSDB index: which streams are in which chunks?
Tenant	The tenant, identified via `X-Scope-OrgID` as with Mimir.

Write and Read Paths

```
Alloy → Distributor → Ingester (RAM, WAL) → S3
                       Replication RF 3

Grafana → Query Frontend → Querier ─┬→ Ingester (recent)
                                    └→ S3 (old)
```

Component	Job
Distributor	Checks, limits, distributes streams by hash.
Ingester	Builds chunks and writes them out.
Query Frontend	Splits queries, caches, queues them.
Querier	Reads and filters.
Index Gateway	Serves the index for queries.
Compactor	Compacts the index, applies retention. One instance.
Ruler	Rules on logs, including alerts.

Labels: Cardinality Again

This matters even more in Loki than in Prometheus. Every label combination is its own stream with its own chunks. Too many streams lead to tiny chunks and a bloated index.

As a label	Not as a label
`namespace`, `app`, `container`	`trace_id`, `user_id`
`cluster`, `env`	`pod` with very high churn
	`status`, `path`

For valuable fields that aren't labels there is **Structured Metadata**. It is stored per line, without creating a stream. That is the right place for `trace_id`.

> **PITFALL**
>
> The classic mistake is a label `level` plus `status` plus `path`. The stream count explodes, and queries get **slower**, not faster. Filters at query time usually serve you better.

IN THE INTERVIEW

How does Loki differ from Elasticsearch?

Elasticsearch builds a full-text index over the content. That makes free-form search fast, but costs a lot of storage and operational work. Loki indexes only labels and searches the text at query time. That is much cheaper, but slower for searches across huge volumes without good labels.

IN THE INTERVIEW

Why must `trace_id` not be a label?

Otherwise every request gets its own stream. That destroys the index and chunk size. I put the trace ID in Structured Metadata or filter for it in the text.

KEY TAKEAWAYS

- Labels are the index, text is read at query time.
- Stream equals label combination.
- Few labels, the rest in Structured Metadata.

24

LogQL

First pick the box, then filter, then calculate.

Structure

```
{namespace="shop", app="api"} |= "error" != "healthz" | json |
status >= 500
└──────── Selector ──────────┘ └── Line filters ──┘  └Parser┘
└Label filter┘
```

At least one selector with a non-empty value is mandatory.

Filtering

```
{app="api"} |= "timeout"          # contains
{app="api"} != "healthz"          # does not contain
{app="api"} |~ "time.?out"        # Regex
{app="api"} !~ "GET /(health|ready)"
```

Parsing

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

Metrics from Logs

```
# Errors per second
sum by (app) (rate({namespace="shop"} |= "error" [5m]))

# Lines in one hour
count_over_time({app="api"}[1h])

# p95 from a field in the log
quantile_over_time(0.95,
  {app="api"} | json | unwrap duration_ms [5m]) by (route)
```

Gaining Speed

1. Narrow selectors. They determine how many chunks Loki has to read.
2. Short time ranges.
3. Line filters before parsers. `|=` is much cheaper than `| json`.

> **PITFALL**
>
> A label from `| json` only comes into being at query time. It doesn't make the search faster, because Loki has to parse all the lines first. Only stream labels shrink the set.

Command Line

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

Memory aid: Log command line interface.

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

Memory aid: Which labels create how many streams?

IN THE INTERVIEW

How do you speed up a slow LogQL query?

First I narrow things down by labels and time, because that determines how much data gets read. Then I put line filters before the parser. For frequent questions I use recording rules in the Ruler.

KEY TAKEAWAYS

- Selector, filter, parser, calculation.
- Line filter first, then parser.
- Parsed labels speed nothing up.

25

Alloy

One collector for all signals, and the successor to Promtail.

Why Alloy?

Promtail reached its end of life on March 2, 2026. Its chart is deprecated. Alloy is Grafana's distribution of the OpenTelemetry Collector and collects metrics, logs, traces, and profiles.

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

Memory aid: Old recipe in, new one out. Check it afterward.

> **PITFALL**
>
> Alloy produces different metric names than Promtail. Dashboards and alerts with the prefix `promtail_` come up empty after the switch. Check them before migrating.

The Language

Alloy builds pipelines from components. A component has inputs and passes on results:

```
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"
  }
}
```

> **PICTURE THIS**
>
> *Lego with connectors. `forward_to` plugs the output of one brick into the input of the next. The order in the file doesn't matter, only the connections count.*

How to Collect Logs?

Component	Source	Verdict
`loki.source.kubernetes`	Kubernetes API	Simple, no host privileges, but more network and Kubelet CPU.
`loki.source.file`	`/var/log/pods` on the Node	Efficient, needs a DaemonSet and HostPath.

On large clusters you use the DaemonSet with files. For that you also set `alloy.mounts.varlog: true` in the chart.

Installation

`values-alloy.yaml`

```yaml
alloy:
  configMap:
    create: true
```

```
    content: |
      // the configuration from above
controller:
  type: daemonset
```

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

Memory aid: DaemonSet: one collector per Node.

If you want everything ready-made, use the `grafana/k8s-monitoring` chart. It rolls out Alloy instances for all signals, but hides more.

Debugging

Alloy has a web UI on port 12345. There you can see the graph of components and their state.

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

Memory aid: Open the UI and look at the component graph.

IN THE INTERVIEW

What do you use to collect logs in Kubernetes today?

Grafana Alloy as a DaemonSet, because Promtail has been end of life since March 2026. Alloy reads the files under `/var/log/pods`, adds Kubernetes labels, and sends them to Loki. An existing Promtail configuration can be carried over with `alloy convert`.

KEY TAKEAWAYS

- Promtail is dead, Alloy takes over.
- `forward_to` connects components.
- Large clusters: read files via DaemonSet.

26

Operating Loki

Mode, S3, retention, and the typical error messages.

The Modes

Mode	Volume	Verdict
Monolithic	up to about 20 GB/day	Getting started and small stacks.
Simple Scalable	up to about 1 TB/day	Deprecated. Goes away with Loki 4.0.
Microservices	above that	Recommended for production with HA.

> **PITFALL**
>
> Many guides still recommend Simple Scalable. For new installations that is a dead end. Existing installations should plan a migration to microservices or highly available Monolithic.

Installation

The chart lives in the community repo:

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

Memory aid: New home of the charts for Grafana, Loki, and 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
```

Memory aid: Community repo, not the Grafana repo.

> **REMEMBER**
>
> `schemaConfig` is empty in the chart and must be set. For new installations, `from` lies in the past. When adding to a running installation, `from` must lie in the future, calculated in UTC. Recommended are `store: tsdb` and `schema: v13`.

Retention

Without `retention_enabled: true`, logs stay in S3 forever. The Compactor deletes, and for that `index.period` must be 24 hours.

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

The highest priority wins. On a tie, the shorter period wins. Per stream, the minimum allowed is 24 hours, and `selector` only understands label matchers.

PITFALL

According to the docs, a changed retention doesn't apply retroactively to logs already ingested. So plan it from the start. The Compactor runs as a single instance, and its deletion markers need a persistent disk or a prefix in the bucket.

REMEMBER

The Table Manager is deprecated. For TSDB you always use the Compactor.

Default Limits

Limit	Default
`ingestion_rate_mb`	4 MB/s per tenant
`ingestion_burst_size_mb`	6 MB
`max_global_streams_per_user`	5,000

Typical Errors

Message	Meaning
`429 ... rate limit`	`ingestion_rate_mb` reached. Raise the limit or send less.
`stream limit exceeded`	Too many streams. Check the labels.
`entry too far behind`	Timestamp too old. `reject_old_samples` kicks in.
`max entry size`	Line too long.

With its `reason` label, the metric `loki_discarded_samples_total` shows what was discarded.

IN THE INTERVIEW

How does Loki delete old logs?

The Compactor applies retention. For that I need `retention_enabled: true`, a `delete_request_store`, and a `retention_period`, globally, per tenant, or per stream. It marks chunks and deletes them after `retention_delete_delay`. Without this setting, everything stays in S3 forever.

IN THE INTERVIEW

Which Loki mode do you use today?

Monolithic for small volumes, microservices for production. Simple Scalable is being deprecated and goes away with Loki 4.0, so I no longer start anything new with it.

KEY TAKEAWAYS

- Chart from `grafana-community`, schema v13 with TSDB.
- Retention via the Compactor, without the switch everything stays.
- Don't introduce Simple Scalable anew.

PART VII

Traces and Correlation

OpenTelemetry, Tempo, TraceQL, and connecting all the signals.

27

Tracing Fundamentals

One request, many services: where did the time go?

Terms

Term	Meaning
Trace	The entire path of a request.
Span	One work step with a start, duration, and attributes.
Trace ID	Connects all spans of a request.
Context propagation	Services pass the IDs along in a header.

```
Trace 4bf92f…
├─ frontend  GET /checkout      ████████████████  820 ms
│  ├─ cart   GET /items         ████              110 ms
│  └─ pay    POST /charge        ████████████      640 ms
│     └─ bank POST /transfer      ████████████    610 ms
```

PICTURE THIS

Package tracking. The number stays the same, the stations change. You can see right away that the package sat at customs for four days.

Context Propagation

The W3C standard uses the HTTP header `traceparent`:

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

PITFALL

If **one** service in the middle doesn't pass the header along, the trace falls apart into two halves. Often it's a proxy, an old gateway, or a queue.

OpenTelemetry

OpenTelemetry (OTel) is the vendor-neutral standard. There are SDKs, auto-instrumentation, the Collector, and the **OTLP** protocol on ports 4317 (gRPC) and 4318 (HTTP).

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

Without code changes, you can use the **OpenTelemetry Operator**: an annotation on the Pod injects the auto-instrumentation. For many languages, that's the fastest way in.

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

Storing all traces is often too expensive.

Kind	Decision	Property
Head sampling	At the start, in the SDK	Cheap but blind. Loses rare errors.
Tail sampling	At the end, in the Collector	Keeps errors and slow traces, needs RAM.

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

PITFALL

With tail sampling, all spans of a trace must arrive at the same Collector. With multiple instances you need load balancing by trace ID in front, in Alloy for example with `otelcol.exporter.loadbalancing`.

IN THE INTERVIEW

Head or tail sampling?

Head is cheap and simple, but the decision is made before it's clear whether a trace will turn out interesting. Tail decides after completion and specifically keeps errors and slow requests. In exchange, it needs memory in the Collector and routing by trace ID.

IN THE INTERVIEW

Why are my traces torn apart?

Usually the context isn't passed along, for example by a proxy or a queue, or a service isn't instrumented. Sampling that keeps only parts of a trace also causes this.

KEY TAKEAWAYS

- A trace consists of spans, and `traceparent` connects them.
- OTel is the standard, OTLP on 4317 and 4318.
- Tail sampling keeps the interesting stuff.

28

Understanding Tempo

Store only what you can find again by ID, and almost everything lives in S3.

The Idea

Tempo needs no database and no full-text index. Traces live as Parquet blocks in S3. That makes it very cheap, even with many spans.

Tempo 3 Is Different

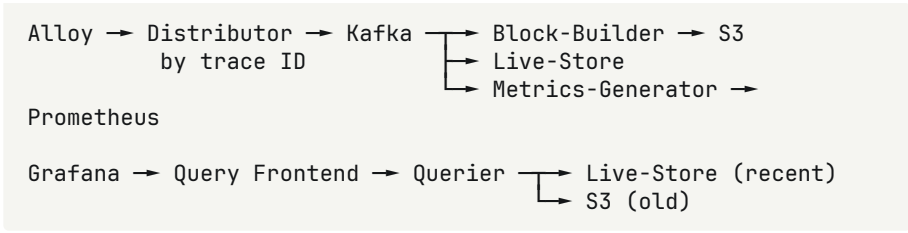
Tempo 3.0 replaced the ingester model. Many guides still show the old architecture.

Old (through 2.x)	New (from 3.0)
Ingester	Block-Builder (write) and Live-Store (read recent data)
Compactor	Backend-Scheduler and Backend-Worker
Replication between ingesters	Kafka in microservices mode
`local-blocks` processor	Removed. The Live-Store serves TraceQL metrics.

Two Modes

Mode	Meaning	Kafka
Monolithic	`-target=all`, everything in one process.	No
Microservices	Individual services, recommended for production.	Yes, mandatory

Writing and Reading



Component	Job
Distributor	Accepts spans, distributes by trace ID.
Block-Builder	Reads from Kafka, builds Parquet blocks, writes to S3.
Live-Store	Answers queries for recent data.
Query Frontend	Splits queries into jobs.
Querier	Works through the jobs.
Backend-Scheduler, -Worker	Compaction, retention, blocklist.
Metrics-Generator	Creates RED metrics and service graphs. Optional.

REMEMBER

Because Kafka guarantees durability, Tempo can work with replication factor 1 on the write path. Each block is created exactly once, and that saves storage.

PICTURE THIS

In the old days, three secretaries took down the same minutes, and a fourth weeded out the duplicates. Today there is a tape recorder (Kafka): one person transcribes the minutes, another answers questions about the latest.

IN THE INTERVIEW

How does Tempo store traces so cheaply?

Tempo puts traces as Parquet blocks into object storage and needs no expensive index. Queries by trace ID and TraceQL run directly on the blocks. In exchange, very broad searches are slower than in an indexed system.

IN THE INTERVIEW

What changed with Tempo 3?

Ingesters and Compactors are gone. In microservices mode, Kafka is mandatory, Block-Builders write to S3, Live-Stores serve recent data, and a Backend-Scheduler distributes compaction and retention. Reads and writes are separate, and TraceQL metrics are GA.

KEY TAKEAWAYS

- Parquet in S3, no index cluster.
- Tempo 3: Block-Builder and Live-Store instead of ingesters.
- Microservices need Kafka, Monolithic doesn't.

29

Operating Tempo

Installation, S3, retention, and TraceQL.

Installation

To get started, use the monolithic chart. It doesn't need 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
```

Memory aid: Tempo now lives in the community repo too.

For production there is `grafana-community/tempo-distributed` (version 3.22). It expects its own Kafka at `ingest.kafka.address`. The number of Block-Builders and Live-Stores has to match the number of partitions: one replica per partition.

> **PITFALL**
>
> `grafana/tempo` and `grafana/tempo-distributed` are deprecated and stay stuck on Tempo 2.9. If you install them, you get the old architecture.

Permissions in S3

The bucket must already exist. The role needs `PutObject`, `GetObject`, `ListBucket`, `DeleteObject`, `GetObjectTagging`, and `PutObjectTagging`.

Retention

Tempo itself keeps blocks for 336 hours by default, meaning 14 days. In the configuration, it looks like this:

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

Per tenant, there is a separate `compaction.block_retention` under `overrides`. Keep an eye on `tempodb_retention_deleted_total` as well.

> **PITFALL**
>
> The monolithic chart sets its own value with `tempo.retention`, and that is only 24 hours. So without an adjustment, traces are gone after one day.

TraceQL

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

`resource.` refers to attributes of the service, `span.` to attributes of the span. `>>` means: the right span is a descendant of the left one.

TraceQL metrics have been GA since Tempo 3.0. They calculate directly on traces:

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

Alerts on TraceQL metrics are still experimental.

Metrics-Generator

It creates ready-made metrics from spans and sends them via Remote Write to Prometheus or Mimir. In the monolithic chart you turn it on with

`tempo.metricsGenerator.enabled`, and the address goes in `remoteWriteUrl`. In the Tempo configuration itself, it looks like this:

```
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 has to accept Remote Write:

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

This produces, for example, `traces_spanmetrics_calls_total`, `traces_spanmetrics_latency_bucket`, and `traces_service_graph_request_total`. From these, Grafana draws the service map.

> **PITFALL**
>
> Span attributes as dimensions are cardinality again. A dimension `user.id` in the Metrics-Generator floods Prometheus.

IN THE INTERVIEW

How do you search Tempo for a service's slow requests?

With TraceQL: `{ resource.service.name = "api" && duration > 2s }`. For an overview I use TraceQL metrics like `quantile_over_time` or the RED metrics from the Metrics-Generator.

IN THE INTERVIEW

How do you get RED metrics without code changes?

Tempo's Metrics-Generator derives them from spans and writes them via Remote Write to Prometheus or Mimir. The `service-graphs` processor also provides the dependencies between services.

KEY TAKEAWAYS

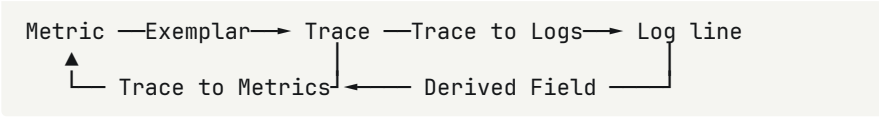
- Chart from `grafana-community`.
- Default retention: 14 days, `compaction.block_retention`.
- TraceQL metrics are GA, alerts on them still experimental.

30

Connecting the Signals

Three tools become one system: from alert to log line in three clicks.

The Tour



Shared Labels First

All signals need the same names: `cluster`, `namespace`, `pod`, `service`. Otherwise Grafana can't link anything. You take care of this during relabeling in Prometheus and Alloy.

Exemplars: From Metric to Trace

An exemplar is a trace ID attached to a measurement. Dots appear in the latency graph that lead directly to the matching trace.

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

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

The application has to deliver exemplars, and the scrape needs a format that carries them, such as OpenMetrics.

> **PICTURE THIS**
>
> *An exemplar is a bookmark in the chart: that outlier there, that was exactly this request. Turn to this page.*

From Logs to Trace

Derived Fields turn a pattern in the log line into a link:

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

From Trace to Logs and Metrics

```
- 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
```

> **REMEMBER**
>
> `datasourceUid` must match the fixed `uid` of the target datasource. So always assign your own UIDs when provisioning.

The Trace ID in the Log

Without this step, the bridge is missing. The application writes the current trace ID into every log line. OTel logging bridges often do this automatically.

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

PITFALL

The links lead nowhere if the trace ID was discarded. With head sampling at 1 percent, 99 of 100 log links point to nothing. Plan sampling and logging together.

IN THE INTERVIEW

How do you get from the alert to the cause?

The alert links to a RED dashboard. In the latency graph I click an exemplar and land on the trace. There I see the slow span and jump via Trace to Logs to the Pod's log lines. This works because all signals carry shared labels and the trace ID.

IN THE INTERVIEW

What is an exemplar?

A trace ID attached to a single metric sample. It takes me from an aggregated graph to a concrete example. Prometheus only stores it if `exemplar-storage` is active and the application delivers it.

KEY TAKEAWAYS

- First shared labels, then links.
- Exemplars lead to the trace, and so do Derived Fields.
- The trace ID has to go in the log.

PART VIII

Production and the Interview

Meta-monitoring, costs, troubleshooting, questions, and cheat sheet.

31

Meta-Monitoring

Who watches the watchmen?

The Problem

If the observability stack goes down exactly when the cluster is on fire, you are blind. So monitor it separately. For large platforms that means a small stack of its own, often in another cluster.

Loki and Mimir recommend a small separate monitoring setup for this. The Loki chart explicitly names Monolithic mode for such meta-monitoring stacks.

The Most Important Metrics

Metric	Alert if
`up == 0`	A scrape target is missing.
`prometheus_rule_evaluation_failures_total`	Rules are failing.
`prometheus_notifications_dropped_total`	Alerts are getting lost.
`prometheus_tsdb_wal_corruptions_total`	The WAL is corrupted.
`prometheus_remote_storage_samples_failed_total`	Remote Write is failing.
`alertmanager_notifications_failed_total`	Messages aren't going out.
`thanos_compact_halted`	The Compactor has halted.
`thanos_shipper_upload_failures_total`	The Sidecar isn't uploading.
`cortex_discarded_samples_total`	Mimir is discarding data.
`loki_discarded_samples_total`	Loki is discarding data.
`tempodb_retention_deleted_total`	If it stays at zero, Tempo isn't deleting anything.

Is Remote Write lagging behind?

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

Sidecar's last upload more than three hours ago?

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

Security

- Don't expose Grafana, Prometheus, and Alertmanager openly on the network. SSO belongs in front of them.
- Logs and traces often contain personal data. Filter tokens, email addresses, and passwords right in the collector.
- Separate tenants, and let only a trusted proxy set `X-Scope-OrgID`.
- GDPR applies to Loki as to any storage: retention has to fit your data protection concept.

IN THE INTERVIEW

Who monitors your monitoring?

A small separate meta-stack monitors the stack components, and the Watchdog alert runs through an external dead man's switch. I alert on ingest errors, Remote Write backlog, discarded data, and halted Compactors.

KEY TAKEAWAYS

- Don't make monitoring depend on itself.
- Discarded data is worth an alert.
- Logs contain sensitive data.

32

Capacity and Costs

The expensive question: what do we need, and what do we keep?

What Costs What?

Signal	Driver	Lever
Metrics	Active time series	Labels, `metric_relabel_configs`, longer interval
Logs	GB per day	Filter debug logs, retention per stream
Traces	Spans per second	Tail sampling, shorter retention

A Retention Strategy

Signal	Duration	Where
Raw metrics	30 days	Thanos or Mimir in S3, 2 to 3 days of that local
Coarse metrics	1 to 2 years	Thanos: 5 min and 1 h. Mimir: recording rules.
Logs	30 days, audit longer	Loki in S3 with retention per stream
Traces	7 to 14 days	Tempo in S3

> **REMEMBER**
>
> The all-important question is: who needs this data, and what for? A year of debug logs rarely helps. A year of traffic trends helps capacity planning a lot.

Doing the Math for the Interview

```
Logs:    50 GB/day × 30 days = 1.5 TB
         Loki compresses roughly 5- to 10-fold
         so around 150 to 300 GB in S3

Metrics: 2 million time series, 30 s interval
         = 66,667 samples/s
         × 86,400 s × 1.5 bytes = around 8.6 GB/day
         × 30 days = around 260 GB
```

```
S3:      About 2 cents per GB/month (Standard, varies by region)
         1 TB = around 20 euros a month
```

Storage is rarely the most expensive part. The machines for ingesters and queriers and the transfer between zones often cost more.

> **PITFALL**
>
> Transfer between Availability Zones can dominate the bill with replication and Remote Write. The pattern: lots of traffic in every zone, and everything crosses the network once.

Splitting Prometheus

If a Prometheus gets too big, you have three options:

- **Functional:** One instance per team or area.
- **Horizontal:** The Operator field `shards` distributes targets by hash across multiple instances.
- **Agent:** Prometheus as a collector only, with storage in Mimir.

> **IN THE INTERVIEW**
>
> **Observability costs are too high. Where do you start?**
>
> First I measure which metrics, namespaces, and log sources are the most expensive. Then I remove unused metrics and labels, lower log levels, split retention by value, and introduce tail sampling. After that I look at infrastructure such as cross-AZ traffic and caches.

KEY TAKEAWAYS

- Metrics: time series. Logs: GB. Traces: spans.
- Split retention by value.
- Often it's not the bytes in S3 that are expensive, but machines and network.

33

Troubleshooting Scenarios

The cases that come up as stories in interviews.

Target Missing

Symptom: No data from the new application.

1. Is it listed under **Status → Targets**? No: the chain from chapter 10.
2. Status `down`? `kubectl exec` and `curl` on the port.
3. `Connection refused`? Wrong port or path.
4. `403`? Authentication or RBAC.

Prometheus in an OOM Loop

Symptom: Pod keeps restarting, log shows WAL replay.

1. `kubectl describe pod`: reason `OOMKilled`?
2. New job or new labels in the last few days?
3. Temporarily give it more RAM so the replay can finish.
4. Then check the TSDB status and eliminate the source.
5. Set `sample_limit` so it doesn't happen again.

Gaps in the Graph

Possible causes:

- Prometheus restarted or was overloaded. `scrape_duration_seconds` close to the timeout?
- Target responds too slowly.
- With two replicas: both scrape individually, so use Thanos Query with deduplication.
- `rate` with too short a window.

Alert Didn't Arrive

1. Prometheus: Is it in the `firing` state under `/alerts`?
2. Alertmanager: Did it arrive? Is a silence or inhibition applying?
3. `amtool config routes test` for the route.

4. `alertmanager_notifications_failed_total` and the log.
5. Receiver: Are the webhook and Secret correct?

Queries over Old Time Ranges Are Empty

1. Thanos: Does `thanos tools bucket inspect` show the blocks?
2. Do Store Gateways see them? `Stores` page in Thanos Query.
3. Is the Sidecar uploading? `thanos_shipper_upload_failures_total`.
4. Compactor halted? `thanos_compact_halted`.
5. Retention wrong? Has the resolution already been deleted?

Store Gateway Needs Too Much RAM

- Index headers grow with the number of blocks: is the Compactor running?
- Very large queries: set up Query Frontend and limits.
- Distribute blocks across multiple Store Gateways by time, with `--min-time` and `--max-time`.

Loki Responds with 429

1. Alloy's log: what is the exact message?
2. `loki_discarded_samples_total` by `reason`.
3. `rate_limited`: raise limits or throttle the source.
4. `stream_limit`: find the drivers with `logcli series --analyze-labels` and remove labels.

Trace Missing

1. Sampling: was it discarded?
2. Is Alloy reachable? Ports 4317 and 4318.
3. Is it too old, meaning outside the retention?
4. Recent traces: are the Live-Stores healthy? Old ones: check the Backend-Workers and S3 permissions.
5. Trace ID and header: does every service speak `traceparent`?

IN THE INTERVIEW

It's three in the morning and the dashboard is empty. What do you do?

First I check whether the data is missing or just the display: Explore in Grafana, then Prometheus directly. Is `up` there? Are the Pods healthy? Then I walk the chain from source to display: target, scrape, storage, datasource. Only after that do I look at what changed recently.

KEY TAKEAWAYS

- Check the chain from source to display.
- Use the tools' own metrics.
- Almost every RAM case is cardinality.

34

The Interview Questions

For answering out loud. Cover up the answer.

Fundamentals

IN THE INTERVIEW

Metrics, logs, traces: when to use what?

Metrics for trends and alerts, traces for locating, logs for the detail.

IN THE INTERVIEW

What are RED and USE?

RED measures services: Rate, Errors, Duration. USE measures resources: Utilization, Saturation, Errors.

IN THE INTERVIEW

What is cardinality?

The number of distinct label combinations, meaning time series. It drives RAM, cost, and query time.

Prometheus

IN THE INTERVIEW

Pull vs. push?

Pull detects outages through `up` and protects against overload. You need push for short jobs and separated networks.

IN THE INTERVIEW

Why shouldn't you display counters directly?

They only go up and drop to zero on restart. What tells you something is the rate of change with `rate()`.

IN THE INTERVIEW

Why `rate` before `sum`?

`rate` detects counter resets only on individual series. After summing, they can no longer be detected.

IN THE INTERVIEW

Histogram or summary?

Histogram, because it can be aggregated across Pods. Summaries can't.

IN THE INTERVIEW

What is the difference between `for` and `keep_firing_for`?

`for` delays firing, `keep_firing_for` delays ending.

IN THE INTERVIEW

What is `up` worth?

It only shows whether the scrape succeeded. It says nothing about whether the service is working correctly.

IN THE INTERVIEW

How do you aggregate data across multiple clusters?

With Thanos Query, Mimir, or Remote Write to a central place. Classic federation only for a few precomputed series.

Operator and Kubernetes

IN THE INTERVIEW

What does the Prometheus Operator do?

It turns CRDs such as `ServiceMonitor` and `PrometheusRule` into the Prometheus configuration and manages the instances.

IN THE INTERVIEW

What is `kube-state-metrics`?

A service that exposes the state of Kubernetes objects as metrics. It doesn't measure usage.

IN THE INTERVIEW

Why `working_set` and not `usage`?

Because `usage` includes page cache that the kernel can free. `working_set` comes closest to OOM behavior.

Alerting

IN THE INTERVIEW

What does inhibition do?

It suppresses follow-up alerts while the cause is firing.

IN THE INTERVIEW

What does alerting on symptoms mean?

I alert on what users feel, and use causes like CPU only when investigating.

IN THE INTERVIEW

What is an error budget?

The allowed remainder before the SLO target is missed. It governs whether I may roll out risky changes.

Long-Term and S3

IN THE INTERVIEW

What does the Thanos Compactor do?

It compacts blocks, creates downsampling, applies retention, and may only run once per block stream.

IN THE INTERVIEW

When does Mimir make more sense than Thanos?

For central platforms with many tenants, hard limits, and teams that can carry Kafka.

IN THE INTERVIEW

Why does a Thanos Prometheus have to write 2-hour blocks?

So that the Sidecar can upload uncompacted blocks and the Compactor alone compacts in the bucket.

IN THE INTERVIEW

Who deletes in S3?

Always the tool via its retention, not a lifecycle rule.

Logs and Traces

IN THE INTERVIEW

Why is Loki cheap?

It indexes only labels and stores the compressed text in S3.

IN THE INTERVIEW

What replaces Promtail?

Grafana Alloy. Promtail has been end of life since March 2, 2026.

IN THE INTERVIEW

Which Loki mode is being phased out?

Simple Scalable. It goes away with Loki 4.0.

IN THE INTERVIEW

What do you need for Tempo in microservices mode?

Kafka or a compatible system, plus object storage.

IN THE INTERVIEW

How do you connect the signals?

Shared labels, exemplars, the trace ID in the log, and links in the Grafana datasources.

Architecture

IN THE INTERVIEW

Design observability for 50 clusters and 200 teams.

In each cluster, Prometheus in agent mode or kept lean, plus Alloy. Centrally, Mimir, Loki, and Tempo as a multi-tenant platform with S3 and authentication via proxy. Limits and retention per team, SLO alerts as templates, dashboards in Git, and a separate meta-stack.

IN THE INTERVIEW

A colleague wants to index all logs as labels. What do you say?

That destroys Loki. Every combination becomes a stream. I show the impact with `logcli series --analyze-labels` and propose Structured Metadata and filters at query time.

KEY TAKEAWAYS

- Answer briefly, then give an example.
- Name trade-offs, not absolutes.
- Say what you would measure.

35

Exercises

No sample solutions up front. Try it yourself first, then check.

1. **Cardinality:** A metric has 30 routes, 4 methods, 6 status codes, and 20 Pods. How many series can arise at most? What happens with a `tenant` label with 500 values?
2. **ServiceMonitor:** Write a Service and ServiceMonitor for an app on port 9100 in the `demo` namespace that `kps` will find.
3. **PromQL:** Write one query for p95 latency per route and one for the error rate.
4. **Rule:** Write an alert for an error rate above 2 percent for 10 minutes and test it with `promtool test rules`.
5. **Routing:** Build a route tree: critical to PagerDuty, team `shop` to Slack, everything else to a catch-all channel. Check it with `amtool`.
6. **SLO:** Calculate: how many minutes of outage does 99.95 percent allow over 30 days?
7. **Thanos:** Turn on the Sidecar and show a block in the bucket.
8. **Retention:** Plan 30 days raw, 180 days at 5 minutes, 2 years at 1 hour. What do you estimate for storage if the raw data costs 100 GB per month?
9. **Loki:** Install Loki with S3 and 30 days of retention, send logs with Alloy, and use LogQL to find the errors per minute.
10. **Tempo:** Send traces to Tempo via OTLP and use TraceQL to find everything over two seconds.
11. **Correlation:** Connect Loki and Tempo in both directions.

Solutions

1. $30 \times 4 \times 6 \times 20 = 14{,}400$ series. With `tenant` it would be 7.2 million.

6. 30 days are 43,200 minutes. 0.05 percent of that is 21.6 minutes.

8. The raw data stays at around 100 GB. On top of that come the 5-minute blocks over 180 days and the 1-hour blocks over 2 years. There is no fixed

formula, because the factor depends on the metrics. So measure it in the bucket. The key idea: you save through the shorter retention of the raw data, not through downsampling.

KEY TAKEAWAYS

- Typing it yourself is where the learning happens.
- Every exercise is an interview story.
- Remember what went wrong.

36

Cheat Sheet

Everything important on a few pages.

Versions and Charts

Tool	Version	Chart
Prometheus	3.15	`kube-prometheus-stack` 92.18
Alertmanager	0.34	same chart
Grafana	13.3	`grafana-community/grafana` 14.3
Thanos	0.42	no official one
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 and 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

Tool	Setting	Default
Prometheus	`--storage.tsdb.retention.time`	15 d (chart: 10 d)
Thanos	`--retention.resolution-raw`, `-5m`, `-1h`	forever
Mimir	`compactor_blocks_retention_period`	forever
Loki	`retention_enabled`, `retention_period`	forever
Tempo	`compaction.block_retention`	14 d

Numbers You Need to Know

Value	Meaning
2 h	Length of a TSDB block.
1–2 bytes	Storage per sample in Prometheus.
30 s / 5 min / 4 h	`group_wait`, `group_interval`, `repeat_interval`.
40 h / 10 d	When Thanos creates 5-minute and 1-hour blocks.
14.4 / 6 / 1	Burn rates over 1 h, 6 h, 3 days.
43.2 min	Error budget at 99.9 percent over 30 days.
4317 / 4318	OTLP over gRPC and HTTP.
9090 / 9093 / 3100 / 3200	Prometheus, Alertmanager, Loki, Tempo.
12345	Alloy UI.
9094	Alertmanager gossip.
10901	Thanos gRPC.

Never Without Thinking

- Use unbounded values as labels.
- Display a counter without `rate`.
- Point two Thanos Compactors at the same block stream.
- Delete data the tool manages with an S3 lifecycle rule.
- Use Glacier for queryable data.
- Create `trace_id` as a Loki label.
- Introduce Promtail or Simple Scalable anew.
- Put a load balancer between Prometheus and Alertmanager.
- Trust `X-Scope-OrgID` without authentication.
- Fail to send the Watchdog outside.

KEY TAKEAWAYS

- Cardinality decides the costs.
- Hot data local, old data in S3, and the tool deletes.
- Shared labels turn three signals into one system.

Index

Agent mode 21
Alert Fatigue 48
Alerting Rules 36
Alertmanager 44
Alloy 10
amtool 44
Block-Builder 97
block_retention 99
Burn Rate 48
cAdvisor 39
Cardinality 16
Chunk 80
Churn 17
Compactor 62
compactor_blocks_retention_period 73
Correlation 103
cortex_discarded_samples_total 75
Costs 110
Counter 22
Dashboards as Code 55
Datasource 52
deploymentMode 88
derivedFields 103
Downsampling 65
Error Budget 48
Exemplars 103
Exporter 20
Gauge 22
Golden Signals 9
Grafana 52
group_left 25
group_wait 44
Head Block 27
High availability 28
Histogram 22
histogram_quantile 24
Ingest Storage 70
inhibit_rules 44
IRSA 58
Kafka 70
kind 12
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 8
Metrics-Generator 97
metrics-server 39
Mimir 70
mimir-distributed 73
Monitoring 8
Native Histograms 23
node-exporter 39
Object storage 58
objstore.yml 65
Observability 8
OpenTelemetry 94
OpenTelemetry Collector 85
OTLP 94
PodMonitor 32
Profiles 9
Prometheus Operator 32
PrometheusRule 32
PromQL 24
Promtail 85
promtool 36
Pull model 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 99
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 99
Traces 8
Troubleshooting 112
TSDB 27
up 20
USE 9
WAL 27
Watchdog 48
X-Scope-OrgID 70