Understanding Sharded Time-Series Databases on Kubernetes with Prometheus and Thanos

---

Overview

As cloud-native systems scale, so does the demand for **high-resolution, long-retention observability**. The difficulty at the heart of this demand is **how to scale the input, storage, and query** of time-series data without creating silos or bottlenecks.
Although many engineers use **Prometheus** to monitor Kubernetes, they frequently run into problems when handling archives for decades or high-volume metrics. Enter a rare but powerful solution: **Sharded Time-Series Databases on Kubernetes using Prometheus and Thanos**.

This deep-dive explores this **advanced observability architecture**, combining:

- **Horizontal sharding of Prometheus**
- **HA scraping with service discovery**
- **Thanos for deduplication and long-term storage**
- **Object storage for infinite retention**
- **Query federation at scale**

Let’s unpack how this stack works, why it’s rare, and how to implement it in production environments with hundreds of microservices.

What is a Sharded Time-Series Database?

In traditional setups, Prometheus scrapes targets, stores them locally, and runs queries from its own storage. However, Prometheus isn’t horizontally scalable by default.

Sharded **time-series architecture** breaks this limitation by:

- Running **multiple Prometheus instances**, each scraping a subset of targets.
- Using a **coordinator layer (e.g., Thanos Query)** to stitch data back together.

- Writing remote blocks to **cloud object storage** (e.g., S3, GCS, Azure Blob).

Why shard?

- Scraping 1M+ metrics per second? One Prometheus won't handle it.
- Need 1+ year of retention? Prometheus has no built-in remote storage.
- High availability? Local TSDB crashes = data loss.

Sharding solves all three.

Core Components of the Stack

1. Prometheus (Sharded + HA)

Instead of one Prometheus instance:

- Deploy **N sharded Prometheus pods** using the prometheus-operator.
- Each shard is configured with a **unique scrape config**, e.g., different namespaces or label matchers.
- Add a **second replica per shard** for redundancy.

Example:

yaml

CopyEdit

```
podMonitor:
  jobLabel: "kubelet"
  selector:
    matchLabels:
      app: kubelet
  namespaceSelector:
    matchNames:
      - node-exporter
```

```
  endpoints:
    - port: metrics
      relabelings:
        - sourceLabels: [__address__]
          regex: "node(1|2|3).*"
          action: keep
```

Pro Tip: Use honor_labels: true and external labels to disambiguate time series.

---

2. Thanos Sidecar

Each Prometheus pod runs a **Thanos sidecar**, which:

- Watches the local TSDB
- Uploads time blocks to **object storage** (e.g., Amazon S3)
- Exposes a gRPC endpoint for **query federation**

Sidecar Config Example:

bash

CopyEdit

```
--tsdb.path=/prometheus \
--objstore.config-file=/etc/thanos/storage.yaml \
--prometheus.url=http://localhost:9090 \
--grpc-address=0.0.0.0:10901 \
--http-address=0.0.0.0:10902
```

Without the sidecar, Thanos Query can’t federate Prometheus shards.

---

3. Thanos Store Gateway

The **Store Gateway** reads from object storage and serves **historical blocks**.

- Supports terabytes of metrics
- Prunes chunks based on retention and compaction rules
- Enables long-term storage without overloading Prometheus

Object Storage Config (storage.yaml):

yaml

CopyEdit

```
type: S3
config:
  bucket: "thanos-metrics"
  endpoint: "s3.amazonaws.com"
  access_key: "<key>"
  secret_key: "<secret>"
```

---

4. Thanos Query

This is the **single pane of glass**.

- Talks to all sidecars + store gateways
- Deduplicates HA Prometheus replicas
- Powers Grafana or external dashboards

You can run multiple Thanos Query nodes behind a **load balancer** for availability.

---

5. Thanos Compactor (Optional)

Reduces object storage costs by:

- Compacting small blocks into large ones
- Downsampling older metrics
- Enforcing retention policies

Architecture Diagram (Text-Based)

lua

CopyEdit

```
 [ServiceMonitors]
      |
+--------------------+
|  Prometheus Shard 1| <-- Sidecar --> Object Storage
+--------------------+
      |
+--------------------+
|  Prometheus Shard 2| <-- Sidecar --> Object Storage
+--------------------+
      |
      |
    Thanos Query <--------> Store Gateway
      |
    Grafana / Alerts / API
```

Scaling Techniques

To scale this observability platform:

Horizontal Sharding

- Use labels, namespaces, or regex-based matchers
- Maintain **balanced scrape targets**

HA Setup

- Use **2+ replicas per Prometheus shard**
- Enable deduplication via external_labels

Object Storage Best Practices

- S3 lifecycle rules to archive old blocks
- Versioning to recover corruptions
- Use bucket encryption (e.g., KMS)

Cost Control

- Compact frequently
- Enable **downsampling** (e.g., 5m, 1h rollups)
- Retain full-resolution data only for 15–30 days

Real-World Use Cases

- **Kubernetes Fleet Monitoring** — Thousands of pods across clusters
- **Multi-region SLO dashboards** with 1-year lookback
- **Cost Allocation** using custom metrics and PromQL
- **Chaos Engineering** observability and feedback loops
- **Machine Learning Observability** (e.g., inference time histograms)

Benefits

Feature	Benefit
Sharded Prometheus	Infinite scraping scale

Feature	Benefit
HA Prometheus setup	Zero metric loss during restarts
Thanos Sidecar + Store	Durable, cheap long-term storage
Query Federation	Centralized dashboards across clusters
Open Source & Cloud Native	Easy to deploy, no vendor lock-in

Security Best Practices

- Use mTLS between Thanos components
- Encrypt object storage with KMS
- Run in isolated namespaces with PodSecurityPolicies
- Audit logs for query endpoints (especially external Grafana)

Testing and Validation

- Load test with **k6 + Prometheus remote_write**
- Benchmark query latency with **Thanos Bench**
- Simulate outages and observe data recovery

Common Pitfalls

Pitfall	Solution
Data duplication from HA shards	Use external_labels and Thanos dedup
Compactor overload	Set concurrency limits and shard compaction jobs

Pitfall	Solution
High query latency	Downsample old data, index efficiently, use caching
Cost spikes in object storage	Enable lifecycle rules, compact aggressively

---

Further Reading

- Thanos Docs
- Kube-Prometheus Stack
- Prometheus Scaling Deep Dive
- Grafana Best Practices
- Cloud Storage Optimization for Metrics

Conclusion

Setting up a **sharded, long-term time-series observability system** on Kubernetes isn't for the faint of heart—but it's an elite-level skill that modern cloud-native organizations need.

If you're a **DevOps engineer**, **platform architect**, or **SRE**, understanding how to scale Prometheus with Thanos in Kubernetes puts you ahead of 90% of teams still struggling with siloed monitoring setups.