KubeCon | CloudNativeCon

North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

Stateless collectors for stateful data: Scaling Prometheus as a node agent

Danny Clark, Google

About me

- Software engineer at Google working on Cloud Monitoring
- Google Cloud Managed Service for Prometheus
- Avid geek for observability tools, distributed systems, and all things cloud native.
- Twitter and Github - @pintohutch

Outline

- Background
- Google Cloud Managed Service for Prometheus
- A new Prometheus operator
- Future direction

Prerequisites

- You have run and configured Prometheus before (even if only a little bit)
- Kubernetes CRDs and the operator pattern [0]

Prometheus

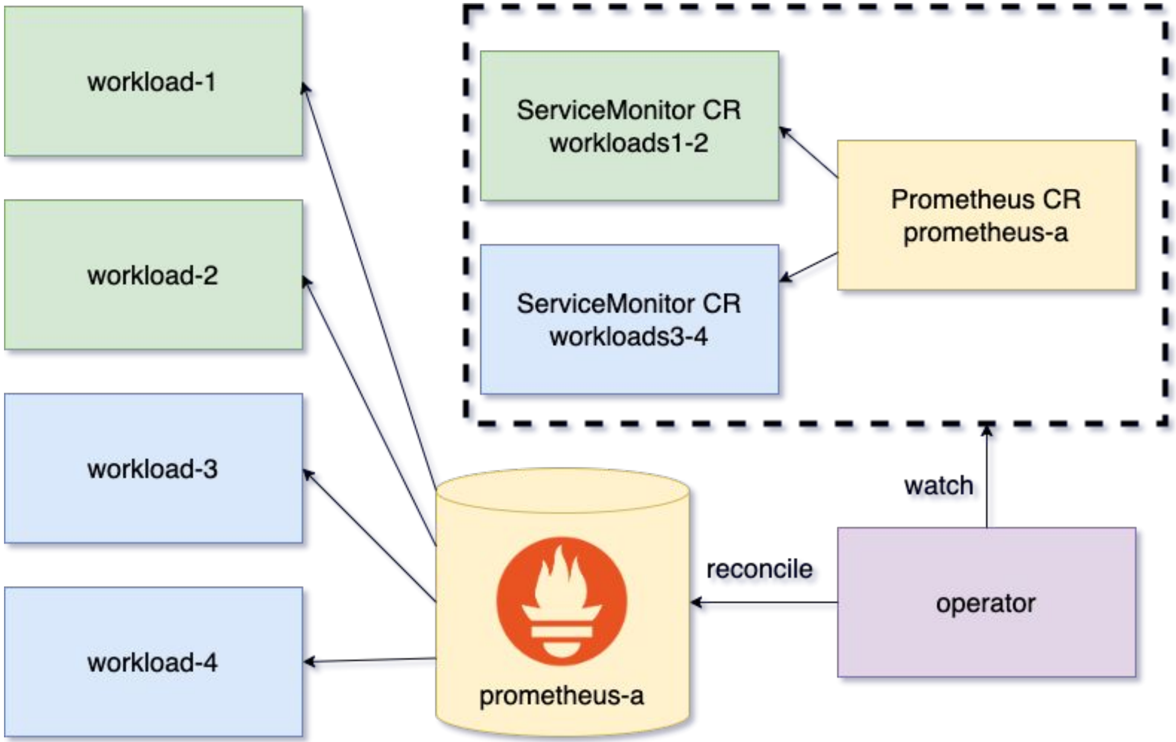
- Prometheus is a metrics-based monitoring and alerting tool
- What it's good at
  - Single process, single server running alongside workloads
  - Monitoring "live" numeric metric data - persistence of weeks
  - Larger instances: 1 million+ samples/s, millions of series, 1-2 bytes/sample [1]
- What it's not
  - Scalable, long-term storage
  - Think more like a traditional "PostgresQL" database, not a "Cassandra"
- Ubiquitous for metrics infrastructure in Kubernetes environments
  - Kubernetes components use Prometheus metrics format [2]

Prometheus on Kubernetes

- prometheus-operator de facto standard
  - Battle-tested, foundational, one of the first Kubernetes operators [3]
- Most people deploy kube-prometheus to setup metric monitoring in their Kubernetes cluster.
  - prometheus-operator
  - HA Prometheus
  - HA Alertmanager
  - Grafana
  - Cluster metrics exporters: kube-state-metrics, node-exporter, prometheus-adapter
  - Default recording and alerting rules

prometheus-operator CRDs


```
apiVersion:
monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus-a
spec:
  replicas: 1
  serviceMonitorSelector:
    matchLabels:
      key: workload
      operator: In
      values: [1,2,3,4]
```

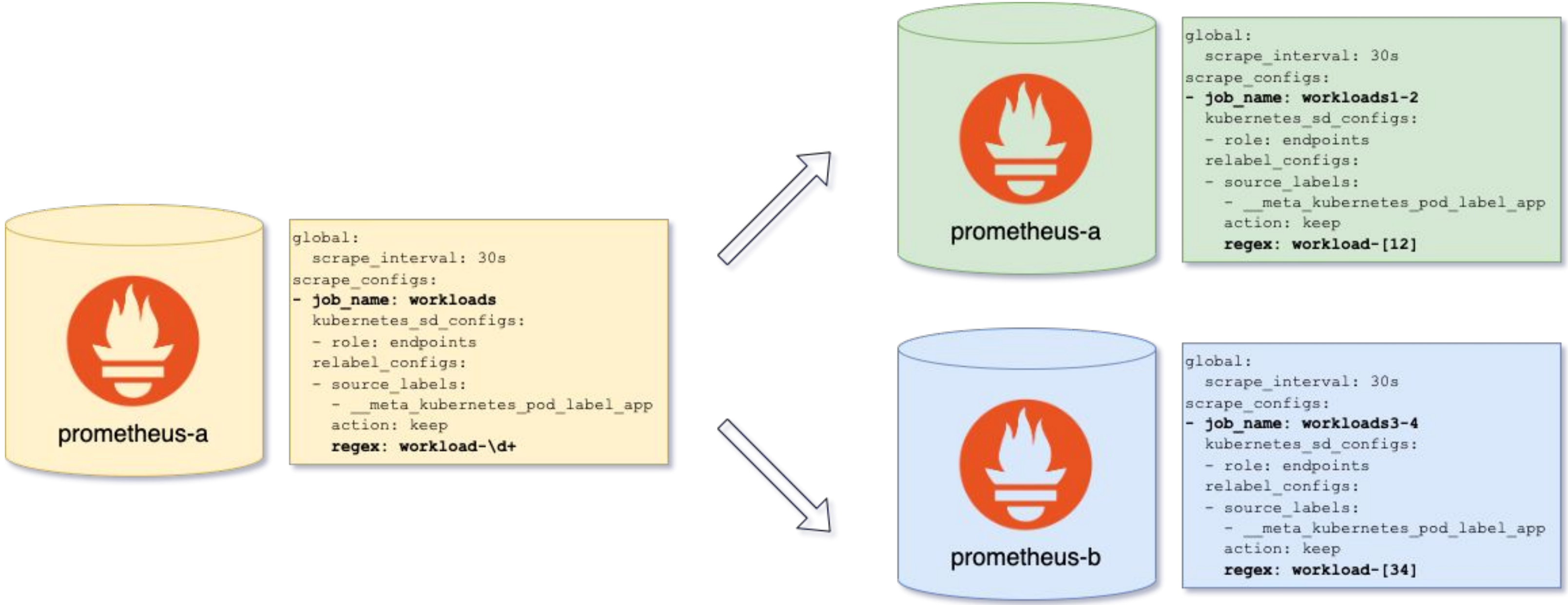


Scaling Prometheus

- Single Prometheus deployment works really great on small to mid-size clusters
    - RAM is the bottleneck for scaling Prometheus
    - Official benchmark (v2.32.1): 4M time series, 100k samples/sec, 4 cores, 25 GiB RAM [4]
        - Example: 100 node cluster, 30 pods/node, 1000 series/pod, 30s scrape interval
    - Dedicate a “monitoring” node pool for Prometheus, 32 GiB memory limit
- What happens when we need more memory?
    - Vertically scale - has practical limits, recurring problem as metric data compounds
    - Horizontally scale - how does that work?

Horizontally-scaling Prometheus

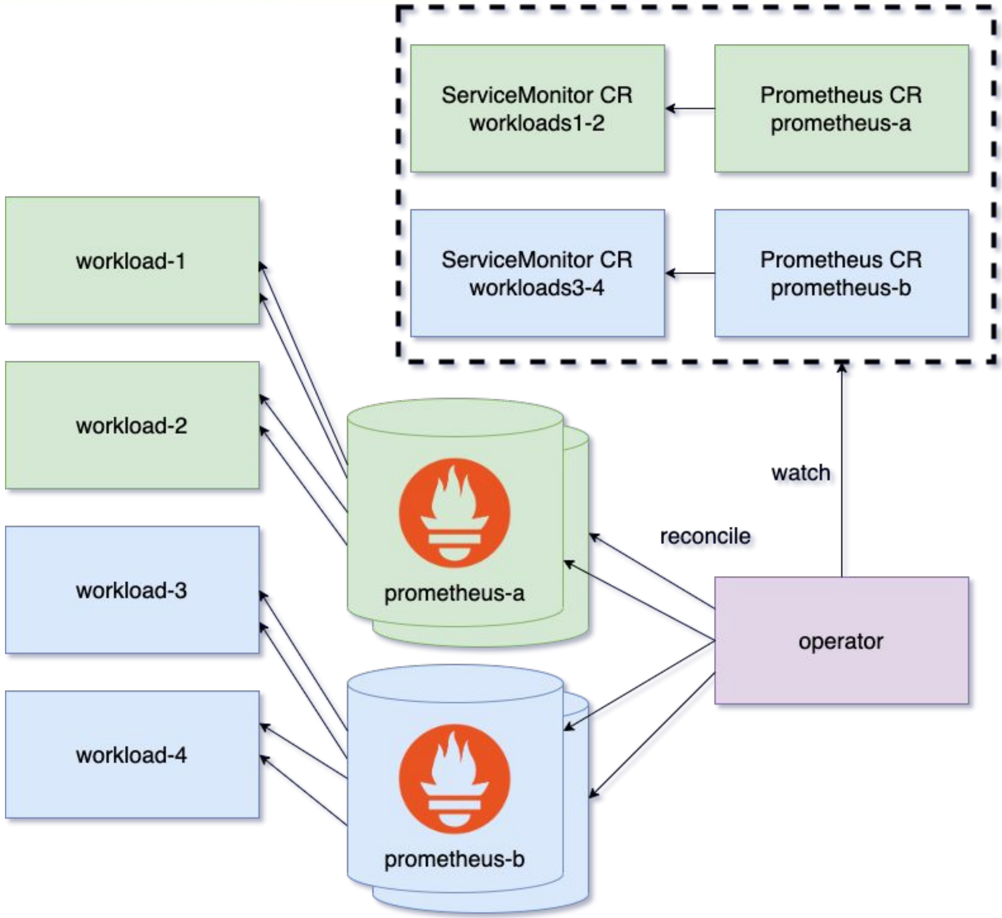
- Logically shard scraped targets by deploying multiple Prometheus servers
- Manually split `scrape_config`s to select subsets of total targets
- HA achieved through additional replicas



prometheus-operator CRDs sharded


```
apiVersion:
monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus-a
spec:
  replicas: 2
  serviceMonitorSelector:
    matchLabels:
      key: workload
      operator: In
      values: [1,2]
```

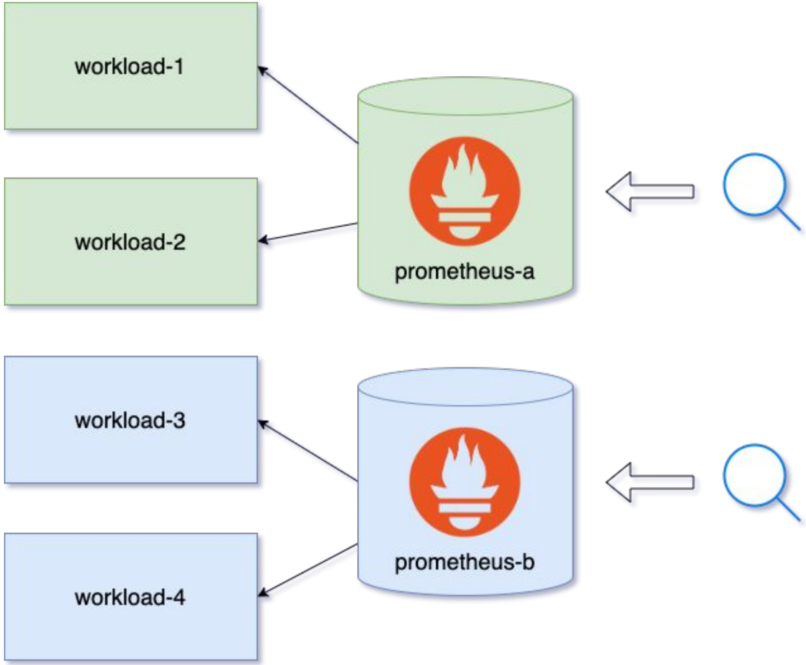
```
apiVersion:
monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus-b
spec:
  replicas: 2
  serviceMonitorSelector:
    matchLabels:
      key: workload
      operator: In
      values: [3,4]
```



Problem: "Single-pane of glass" queries

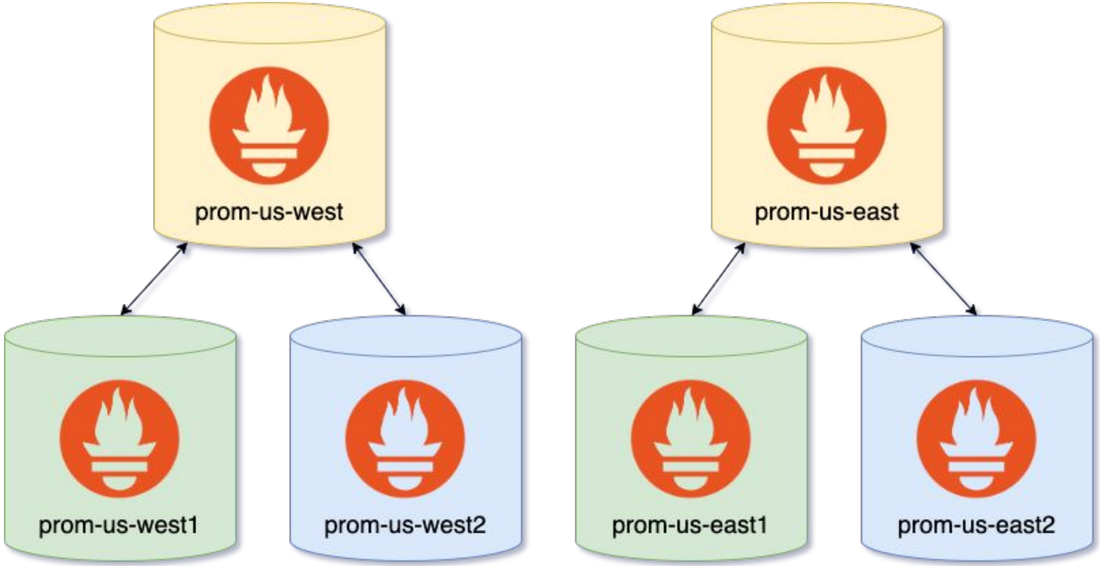
KubeCon CloudNativeCon North America 2022

- Queries can only target a subset of data, each persisted to separate Prometheus servers
- Need to know which servers have desired metrics at query time.



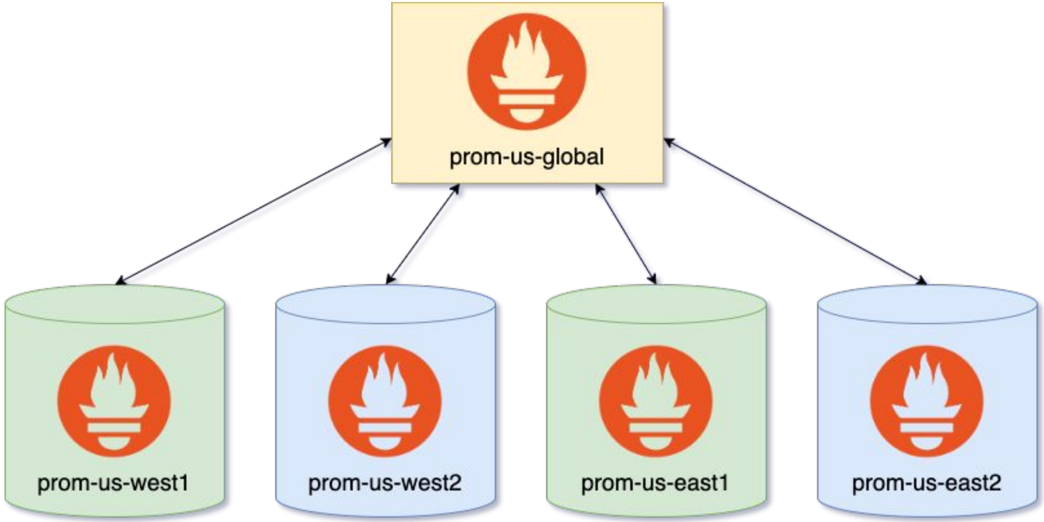
Fix: Federation

- “Parent” Prometheuses scrape from each “child” Prometheus `/federate` endpoint and aggregate
- Downside: wrangling deployments and configuration - ensuring all components are properly talking to each other, scraping and aggregating the right data
  - Fix: dedicated SREs to maintain infrastructure



Fix: Remote Read

- A central server pulls in data from remote Prometheus servers at query time without federating Prometheuses.
- Downside: currently no query pushdown [5], pulling in potentially GBs of data, susceptible to network failures, ingress policies
    - Fix: dedicated SREs to maintain network, possible client-side query aggregation layer



Fix: Thanos

- Deploy Thanos sidecar (e.g. use prometheus-operator Prometheus CR [6]) and Thanos querier components onto cluster (e.g. use kube-thanos)
- Downside: wrangling deployments and configuration
  - Fix: dedicated SREs to maintain architecture and configuration

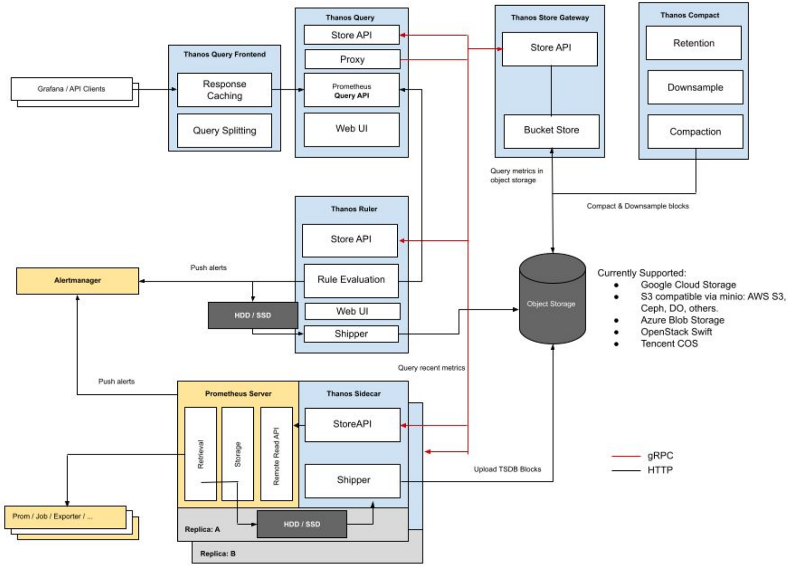
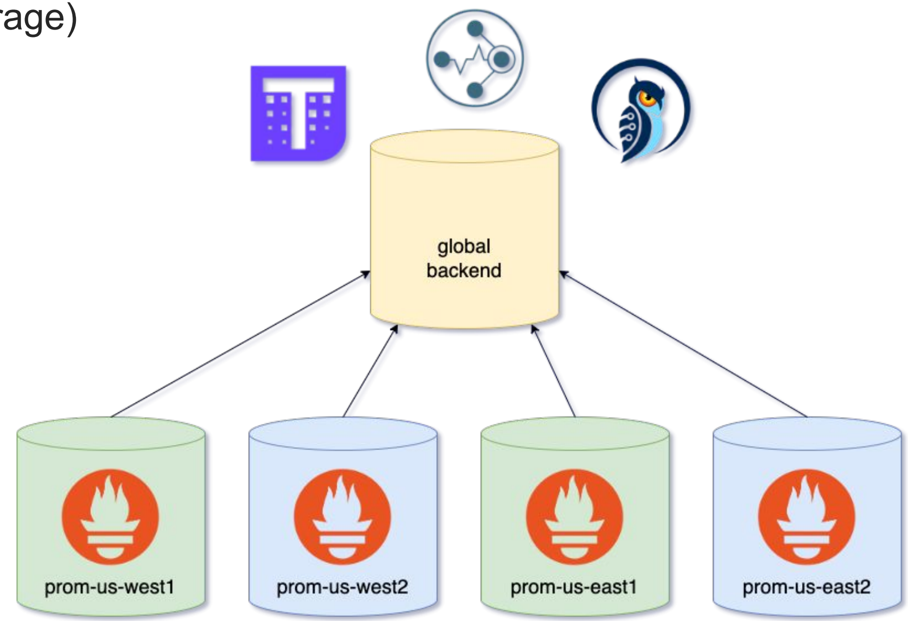


image source [7]

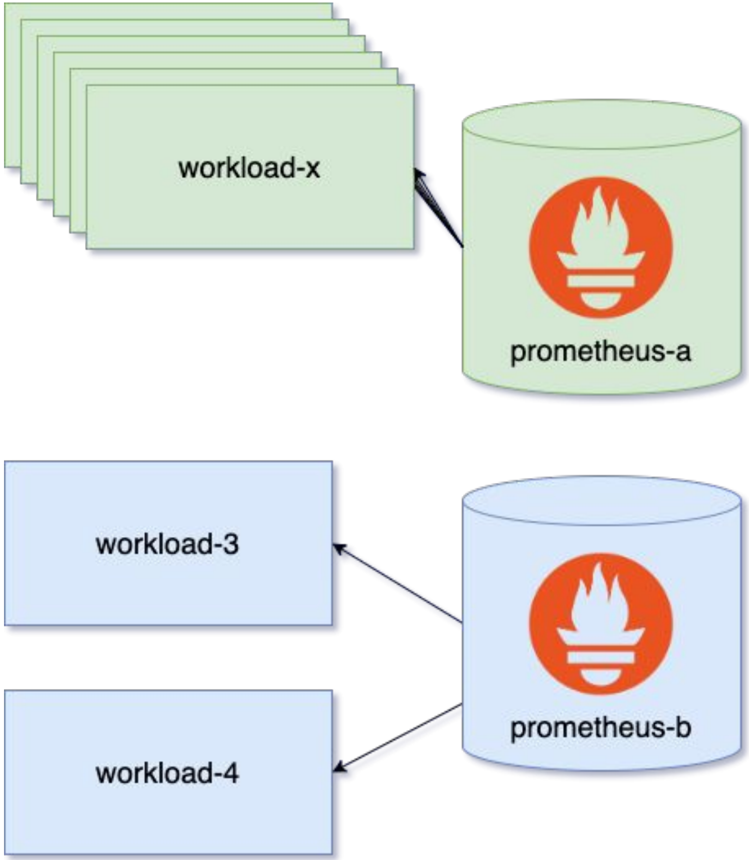
Fix: Remote Write

- Forward the data to a Remote Write-compatible backend, like Cortex, Thanos, M3, etc.
- Downside: maintaining persistent and scalable backend, Remote Write uses ~25% more memory than standard Prometheus [8]
    - Fix: Prometheus Agent Mode - consume less resources, but removing other features (e.g. querying, alerting, local storage)



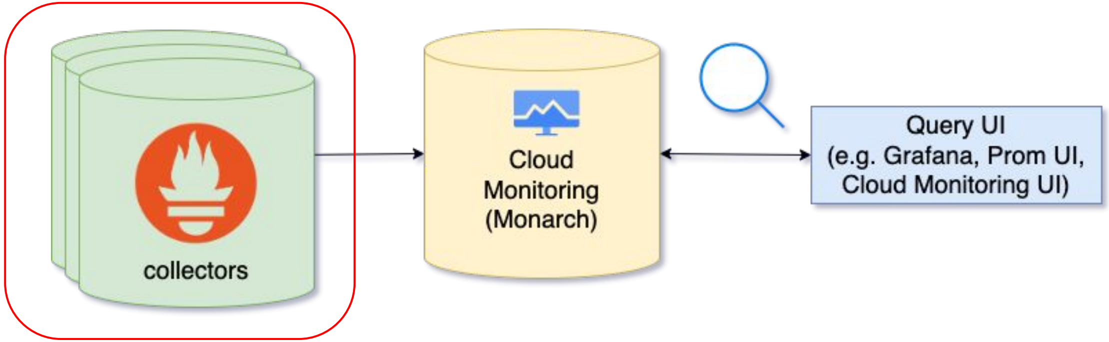
Problem: rebalance shards

- Federation and remote read
  - Once a shard contains data, it stays there
  - Scaling or rebalancing shards is a manual process and requires orchestration, potentially backfilling.
- Thanos and remote write
  - In dynamic, large-scale environments (e.g. large Kubernetes clusters), still vulnerable to a single shard becoming overwhelmed by growing targets and metrics data.



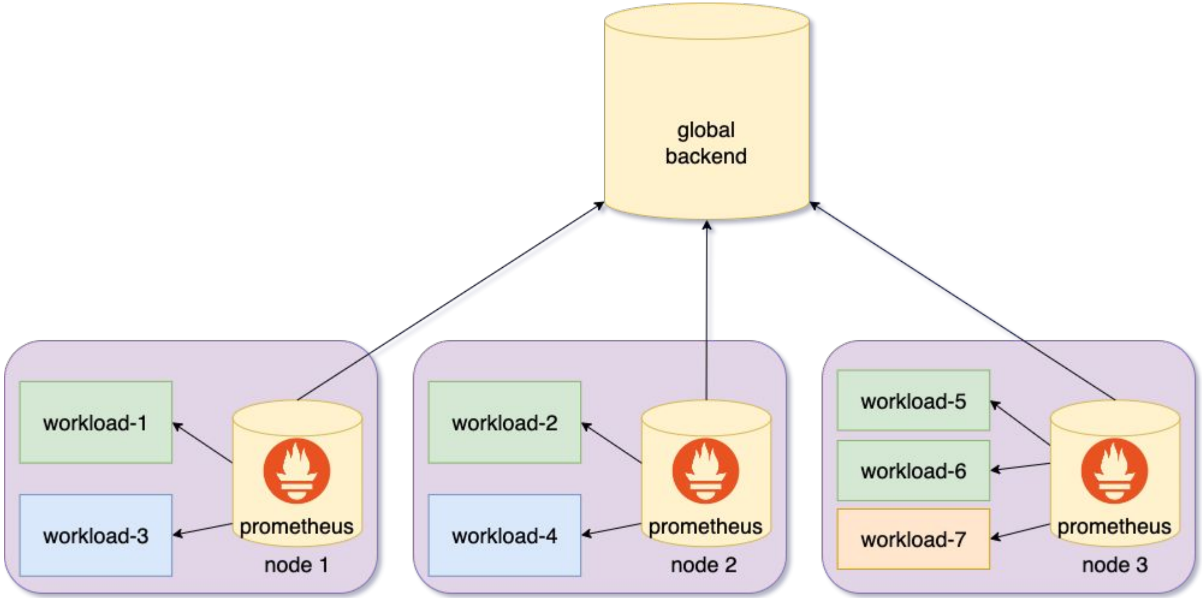
Building a managed service

- Google Cloud Managed Service for Prometheus
  - Shipping metrics off to a remote backend is appealing
    - Treat Prometheus in-cluster as “stateless”
    - Separates state management and query load from scrape concerns
    - Utilize Google’s planet-scale time series database Monarch
      - Serves over 2 trillion active time series
      - Long-term retention available - 2+ years
  - Need a stable, scalable metrics ingestion approach



Prometheus as a node agent

- Limit Prometheus to scraping co-located targets on the same node - “collectors”
  - Size of targets and metrics is naturally constrained by the capacity of the node
- “Single-pane of glass” querying - write metrics to a central remote backend
- Kubernetes provides the DaemonSet to achieve this



Kubernetes downward API

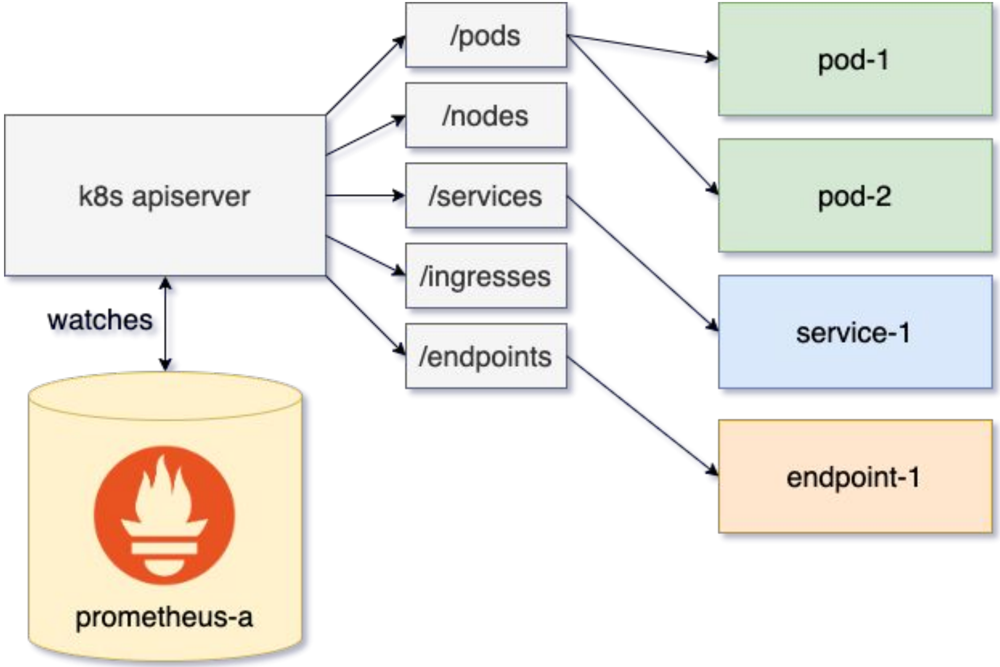
- Each Prometheus in the DaemonSet needs to know the node its on
- Leverage the Kubernetes downward API - container is exposed to information about itself
- Mount the node name as an environment variable and insert into Prometheus configuration file
- Relabeling configs are the conventional way of filtering and relabeling targets

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: prometheus
spec:
  template:
    spec:
      containers:
      - name: prometheus
        image: prom/prometheus
        env:
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: spec.nodeName
...
```

```
global:
  scrape_interval: 60s
scrape_configs:
  - job_name: 'workloads-12'
    relabel_configs:
      - source_labels:
[__meta_kubernetes_pod_node_name]
        regex: $NODE_NAME
        action: keep
    # endpoints role - most SD labels
    kubernetes_sd_configs:
      - role: endpoints
...
```

Prometheus Service Discovery

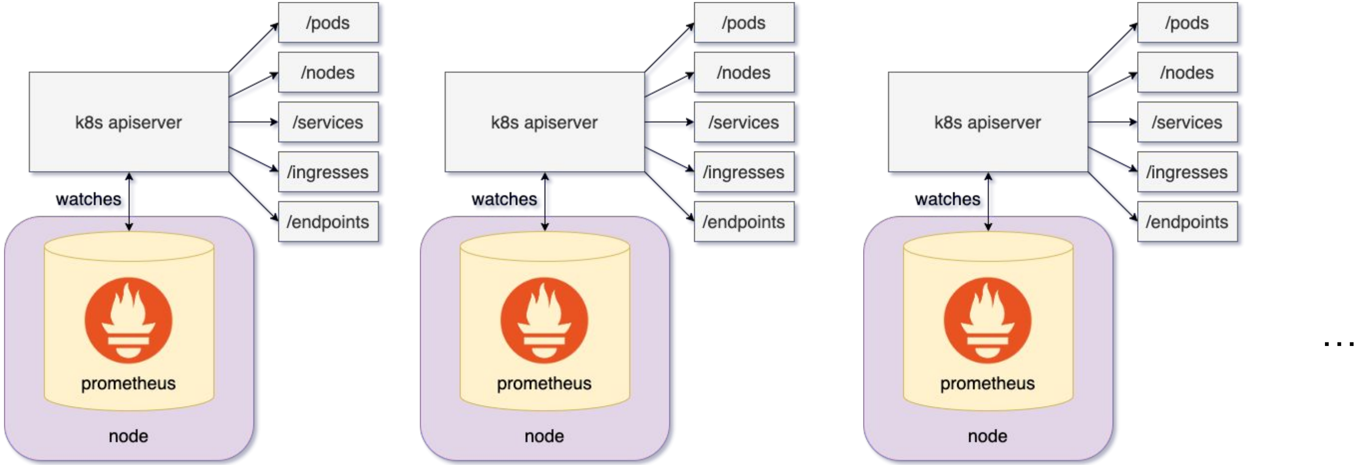
- In order to discover targets to scrape, Prometheus opens watches against the Kubernetes API server for various resources, i.e. `node`, `service`, `pod`, `endpoints`, `endpointslice`, `ingress` roles




```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
    - nodes
    - nodes/metrics
    - services
    - endpoints
    - pods
  verbs: ["get", "list", "watch"]
- apiGroups: [“networking.k8s.io”]
  resources: [“ingresses”]
  verbs: ["get", "list", "watch"]
```

DaemonSet scaling considerations

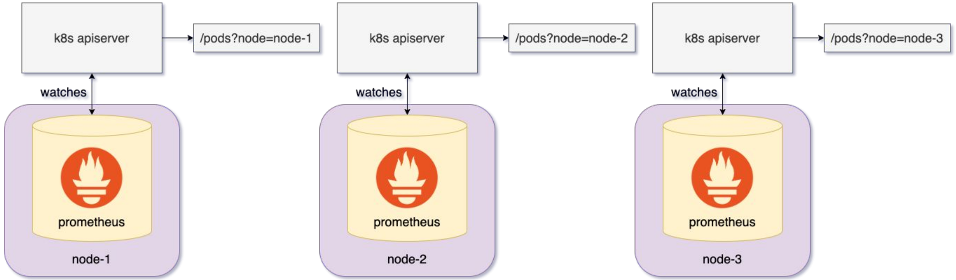
- With Prometheus running on every node in the cluster, the number of watches compounds by N
- On larger clusters of O(100) or O(1000) nodes, this puts considerable strain on the K8s apiserver
  - Each watch spawns 2 go routines [9] [10]
  - Needs to process all changes for objects of a given Kind.
  - Needs to serialize and send updates to clients for objects of a given Kind
- Reducing to node-specific targets via relabel configuration is filtering at the “last mile”, does not reduce load on K8s apiserver



DaemonSet scaling considerations

- Instead of relabel configs, use field selectors to filter targets at "discovery time"
- This works because the K8s apiserver watch cache indexes pods by node name [11]
  - Only has to process changes for pods on that node (as opposed to the whole cluster)
  - Greatly reduces the resource utilization
- kubelet [12] and kube-proxy [13] already use this pattern

```
global:
  scrape_interval: 60s
scrape_configs:
  - job_name: 'workloads-12'
    relabel_configs:
      - source_labels:
[__meta_kubernetes_pod_node_name]
        regex: $NODE_NAME
        action: keep
    # endpoints role - most SD labels
    kubernetes_sd_configs:
      - role: pod
        selectors:
          - role: pod
            field: spec.nodeName=$NODE_NAME
...
```



A new Prometheus operator

KubeCon | CloudNativeCon North America 2022

- Deployment model is fundamentally different from existing prometheus-operator CRDs
  - Though there are proposals documented [14] with a similar approach
- Enforce `pod` role and node field selectors in CRDs to handle scaling challenges
- Address some other things:
  - RBAC-enforced tenancy
  - Simpler, more opinionated configuration surface

PodMonitoring CRD

- Configure `scrape_configs` through `PodMonitoring` custom resource
- Closely modeled after prometheus-operator `ServiceMonitor` and `PodMonitor`
  - With some differences

```yaml
apiVersion: monitoring.googleapis.com/v1
kind: PodMonitoring
metadata:
  name: prom-example
  namespace: backend
  labels:
    app.kubernetes.io/name: prom-example
spec:
  selector:
    matchLabels:
      app: prom-example
  endpoints:
  - port: metrics
    interval: 30s
```

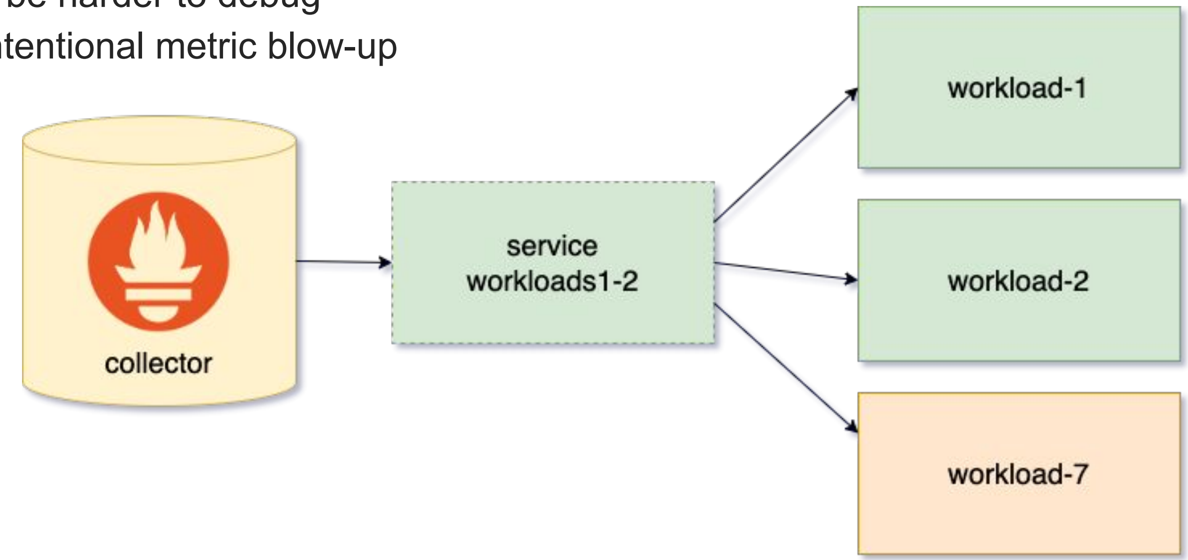
```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: prom-example
  namespace: monitoring
  labels:
    app.kubernetes.io/name: prom-example
spec:
  selector:
    matchLabels:
      app: prom-example
  endpoints:
  - port: metrics
    interval: 30s
  namespaceSelector:
    matchNames: [backend]
```

PodMonitoring tenancy

- `PodMonitoring` - namespace tenancy
  - CR in one namespace cannot scrape workloads in another
    - In contrast to `NamespaceSelector` field on `ServiceMonitor`
  - Aligns with Kubernetes RBAC enforcement around namespaces
  - Prevents accidental metric blow-up from matching targets in other namespaces
  - All ingested metrics are relabeled with `{namespace=<pm-namespace>,cluster=<cluster>}` to enforce tenancy in persisted metrics.
- `ClusterPodMonitoring` - cluster-scoped scraping has valid use-cases
  - Convenient to scrape across namespaces with less CRs
  - Support exporters where we want to preserve the `namespace` label (e.g. kube-state-metrics)
  - Dedicated Custom Resource
    - Can limit so only “trusted” authorities can use with Kubernetes RBAC
  - All ingested metrics are relabeled with `{cluster=<cluster>}` to enforce tenancy in persisted metrics.

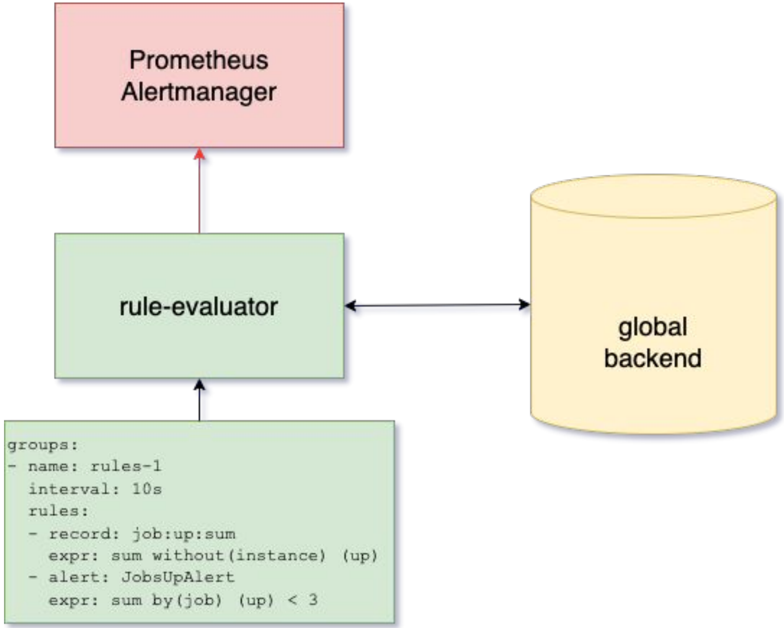
PodMonitoring CRD

- Limited to scraping pods
  - Can’t constrain node-local watches for service endpoints in a scalable way
  - `ServiceMonitor` usually used to scrape pod workloads behind a service
  - Less common to scrape pure service endpoints, or multiple services with common labels
  - Service-level spec adds a layer of indirection for target discovery
    - Can be harder to debug
    - Unintentional metric blow-up



Rule evaluation

- Prometheus running effectively "stateless" on every node won't work for recording or alerting rules
  - Metric data is centrally located in a remote backend
  - Need to query and rewrite rule data against the "global" view
- Deploy a separate workload, rule-evaluator
  - Takes Prometheus recording and alerting rules
  - Queries and writes recording-rules to remote backend
  - Sends alerts to Alertmanager
  - Similar idea to Thanos Ruler



Rules CRD

- Configure `rule_files` through `Rules` custom resource
- Closely modeled after prometheus-operator `PrometheusRule`
  - With differences

```yaml
apiVersion: monitoring.googleapis.com/v1
kind: Rules
metadata:
  name: example-rules
  namespace: backend
  labels:
    app.kubernetes.io/name: example-rules
spec:
  groups:
  - name: example
    interval: 30s
    rules:
    - record: job:up:sum
      expr: sum without(instance) (up)
    - alert: AlwaysFiring
      expr: vector(1)
```

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: example-rules
  namespace: backend
  labels:
    app.kubernetes.io/name: example-rules
spec:
  groups:
  - name: example
    interval: 30s
    rules:
    - record: job:up:sum
      expr: sum without(instance) (up)
    - alert: AlwaysFiring
      expr: vector(1)
```

Rules tenancy

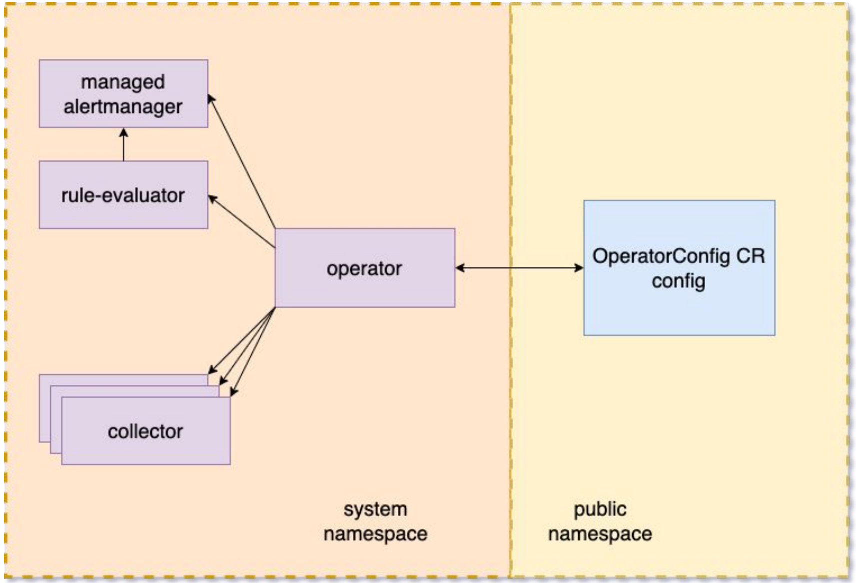
- `Rules` - namespace-scoped
  - CR in one namespace cannot query or write to time series in another
  - Prevents expensive queries matching time series in other namespaces
  - Prevents colliding with other recording rules on writes
  - All rules are relabeled with `{namespace=<rule-namespace>, cluster=<rule-cluster>}` to enforce tenancy in persisted time series.
- `ClusterRules` - cluster-scoped rules have valid use-cases
  - Convenient to query across namespaces, for a given `cluster`
  - All rules are relabeled with `{cluster=<rule-cluster>}` to enforce tenancy in persisted time series.
- `GlobalRules` - multi-cluster scoped rules have valid use-cases
  - Convenient to query across clusters
  - No relabeling
- Aligns with Kubernetes RBAC enforcement around namespaces

OperatorConfig CRD

- `OperatorConfig` - “application-level” configuration of resources controlled through a “top-level”, singleton CR
  - `CollectionSpec`
    - Configure metrics `external_labels` on persisted time series
    - Filtering, compression, when exporting to backend
  - `RuleEvaluatorSpec`
    - Configure `external_labels` on rules and alerts
    - Alertmanager endpoints to route to
  - `ManagedAlertmanagerSpec`
    - Configure out-of-the-box Alertmanager

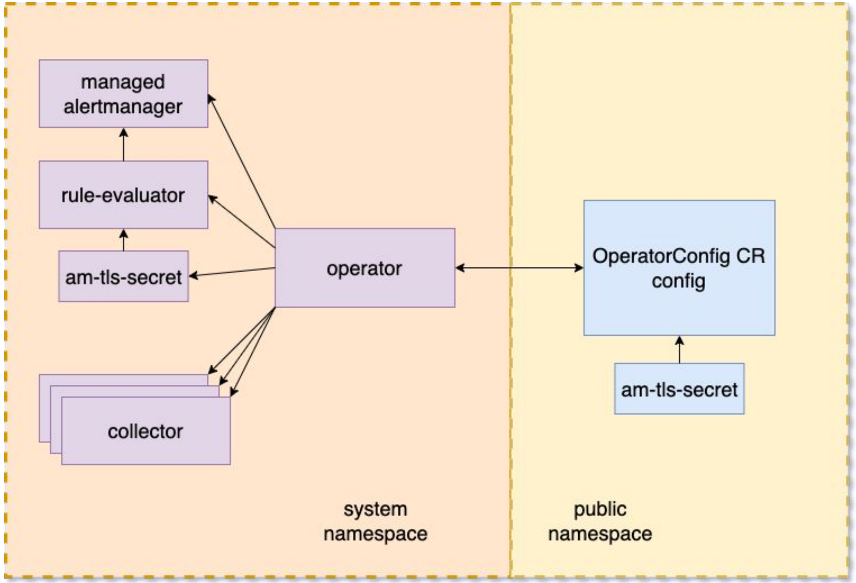
OperatorConfig CRD

- All managed components live in a “system” namespace alongside the operator
  - Users should not need to modify anything in “system”
- `OperatorConfig` singleton lives in a dedicated “public” namespace, watched by the operator
  - Users can modify resources to configure system components
- Allows more restrictive RBAC of managed collection components
  - operator only needs Secrets access in two namespaces



OperatorConfig CRD

- Example: send alerts to Alertmanager service over TLS
- Convention to use use Kubernetes `Secret` resources
  - Fetch secrets, mount to workload (i.e. rule-evaluator)
- rule-evaluator runs in “system” namespace
  - Secrets cannot be mounted across namespaces
  - Don’t want users to modify in “system” namespace
- Store secrets in designated “public” namespace alongside `OperatorConfig` singleton
  - Operator will mirror them to “system” namespace for use by managed components



Minimize configuration surface

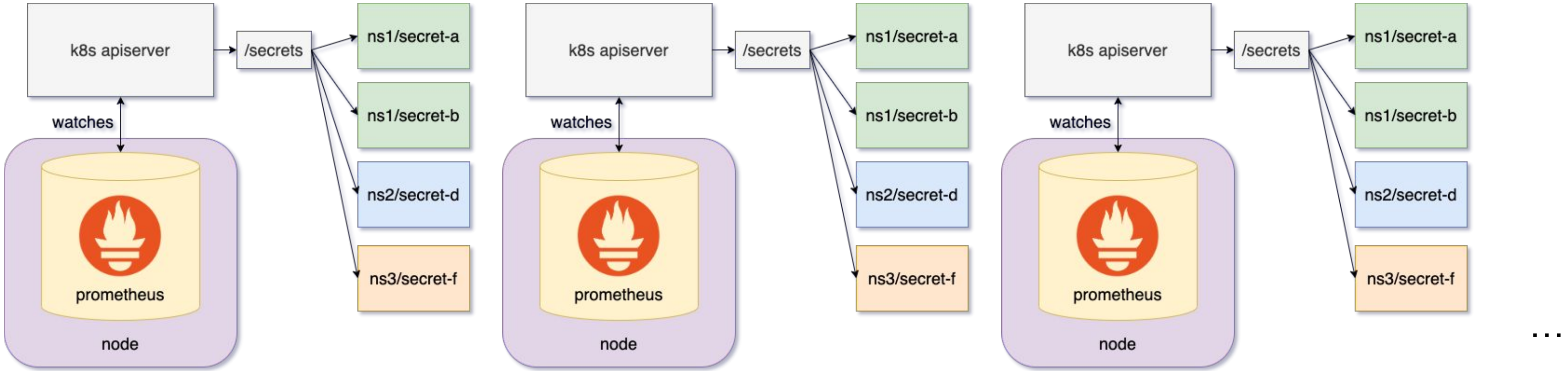
- Start out with minimum configuration to simplify user experience
    - Can always add fields based on use cases and demand
- `PodMonitoring` and `ClusterPodMonitoring` provide
    - Label selectors for target selection
    - “Basic” scrape configuration
        - Port, path, scheme, interval, timeout, etc
        - Limits - number of samples, labels
    - “Limited” relabeling capabilities
        - Allow target relabeling with conventional metadata (e.g. `container`, `node`, `pod`) and attached pod labels (e.g. `app.kubernetes.io/name`)
        - No `honor_labels`, `honor_timestamps` - primarily used for federation or Pushgateway
        - Allow `metric_relabel_configs` - can be risky, but useful for exporters that inject dozens of labels that may not be useful

Minimize configuration surface

- No `Prometheus` CRD
  - Prometheus runs as a `DaemonSet`
- No `Ruler` CRD
  - rule-evaluator runs as a `Deployment`
- Remove operator from reconcile loop
  - Instead of reconciling the entire resource, the operator reconciles configuration
    - Scrape configuration for Prometheus DaemonSet
    - Generated rules and configuration for rule-evaluator Deployment
  - Let Kubernetes reconcile managed resource at the “infrastructure level”
    - Configuration of components (e.g. Prometheus flags, CPU/RAM resource limits, volume mounts, security profiles, etc) can be done directly on the `PodTemplateSpec` at install or runtime.
    - Allows for customization of resources, while minimizing CRD configuration surface.

Future direction

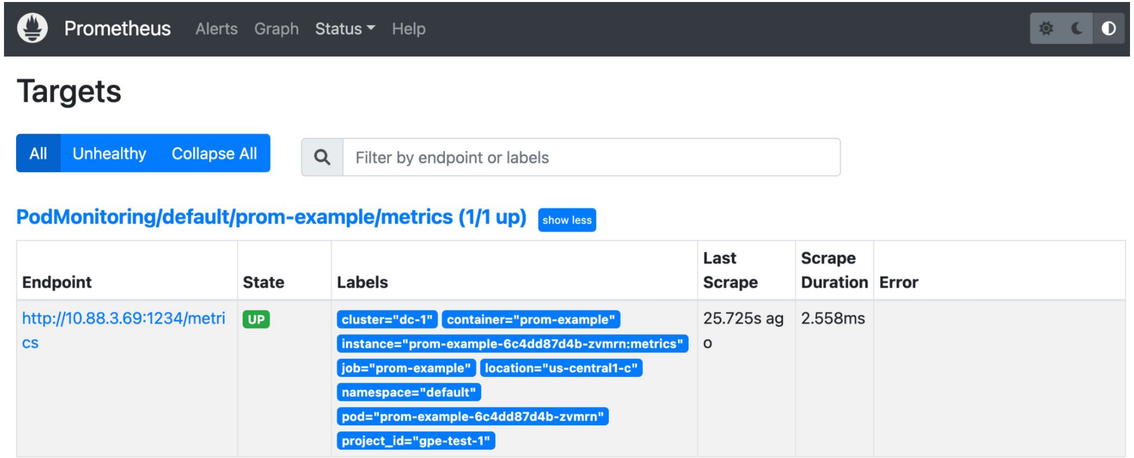
- Authenticated scraping
  - Protected metrics endpoints are supported in Prometheus
  - Collectors need access to secrets
    - Elegantly handle secrets in other namespaces
  - Again, need to be careful about scaling
    - DaemonSet opening watches against secrets could strain K8s apiserver



Future direction

KubeCon CloudNativeCon North America 2022

- Support collection-side status APIs
  - Prometheus provides “status” (i.e. non-PromQL) APIs
    - `/targets, /rules, /alerts, /alertmanagers`
    - `/status/{config,flags,buildinfo,runtimeinfo}`
  - Helpful debugging tool – sensible in single-instance Prometheus setups
  - DaemonSet, status is effectively sharded throughout the cluster
    - No centralized, convenient place to surface status information



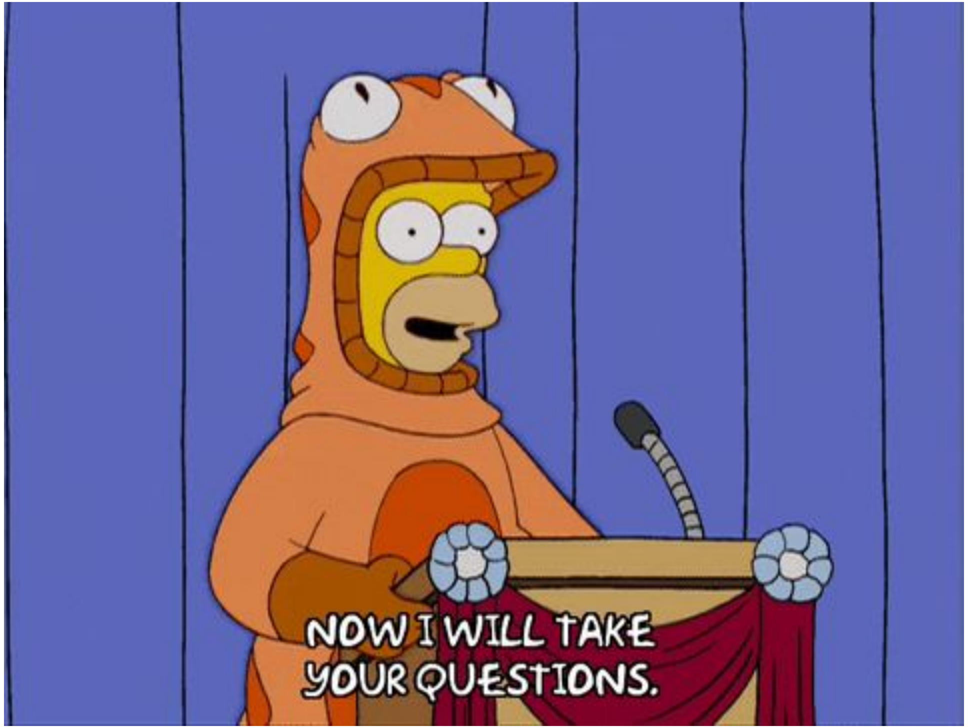
Summary

- Scaling Prometheus can be a challenge
  - Separating collection from querying can be advantageous
- Running Prometheus as a node agent
  - Fairly simple to maintain
  - Scaling can affect K8s apiserver
- New operator and custom resources
  - Emphasize tenancy around target discovery and metrics scope
    - Kubernetes RBAC best practices
  - Dedicated “system” and “public” namespaces
    - Constrain RBAC needed for monitoring infrastructure
    - Users interact with dedicated “public” namespace, not “system”
  - Simple configuration surface
    - Favor simplicity over configurability
    - Operator reconciles metrics configuration, not monitoring infrastructure

Thanks!

KubeCon CloudNativeCon North America 2022

- Check out https://github.com/GoogleCloudPlatform/prometheus-engine



Sources

0. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/
1. https://sched.co/Zex4
2. https://kubernetes.io/docs/concepts/cluster-administration/system-metrics/
3. https://web.archive.org/web/20190113040626/https://coreos.com/blog/the-prometheus-operator.html
4. http://prombench.prometheus.io/grafana/d/7gmLoNDmz/prombench?orgId=1&from=2022-01-05%2006:59:00&to=2022-01-06%2006:59:00
5. https://github.com/thanos-io/thanos/issues/305
6. https://github.com/prometheus-operator/prometheus-operator/blob/v0.59.2/pkg/apis/monitoring/v1/types.go#L472-L480
7. https://github.com/thanos-io/thanos/tree/v0.28.0#architecture-overview
8. https://prometheus.io/docs/practices/remote_write/#memory-usage
9. https://github.com/kubernetes/kubernetes/blob/v1.25.2/staging/src/k8s.io/apiserver/pkg/endpoints/handlers/get.go#L263
10. https://github.com/kubernetes/kubernetes/blob/v1.25.2/staging/src/k8s.io/apiserver/pkg/storage/cacher/cacher.go#L542
11. https://github.com/kubernetes/kubernetes/blob/v1.25.2/pkg/registry/core/pod/storage/storage.go#L90
12. https://github.com/kubernetes/kubernetes/blob/v1.25.2/pkg/kubelet/config/apiserver.go#L38
13. https://github.com/kubernetes/kubernetes/blob/v1.25.2/cmd/kube-proxy/app/server.go#L775-L780
14. https://github.com/prometheus-operator/prometheus-operator/blob/v0.59.2/Documentation/designs/prometheus-agent.md

Stateless collectors for stateful data: Scaling Prometheus as a node agent

KubeCon | CloudNativeCon
North America 2022
BUILDING FOR THE ROAD AHEAD
DETROIT 2022

October 24-28, 2021

Danny Clark
Software engineer,
Google

Please scan the QR Code above
to leave feedback on this session