



KubeCon

CloudNativeCon

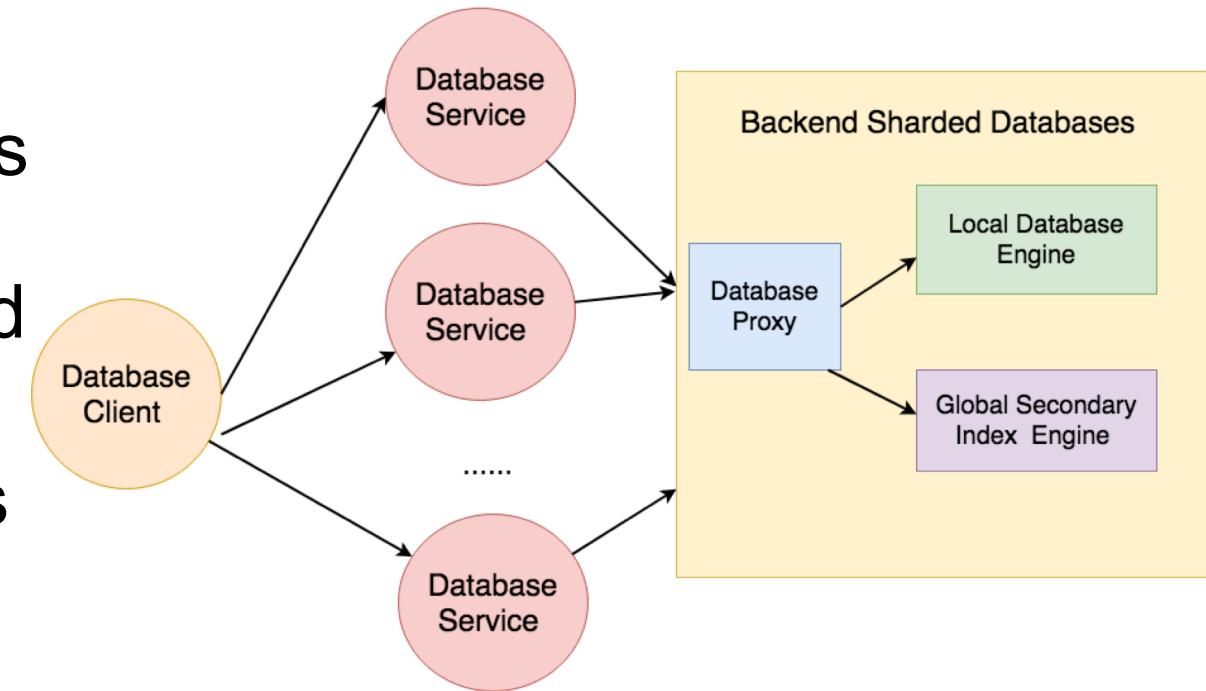
— North America 2018 —

# Sharded And Federated Prometheus Clusters to Monitor Distributed Databases

Jun Li and Viswa Vutharkar, eBay Inc.

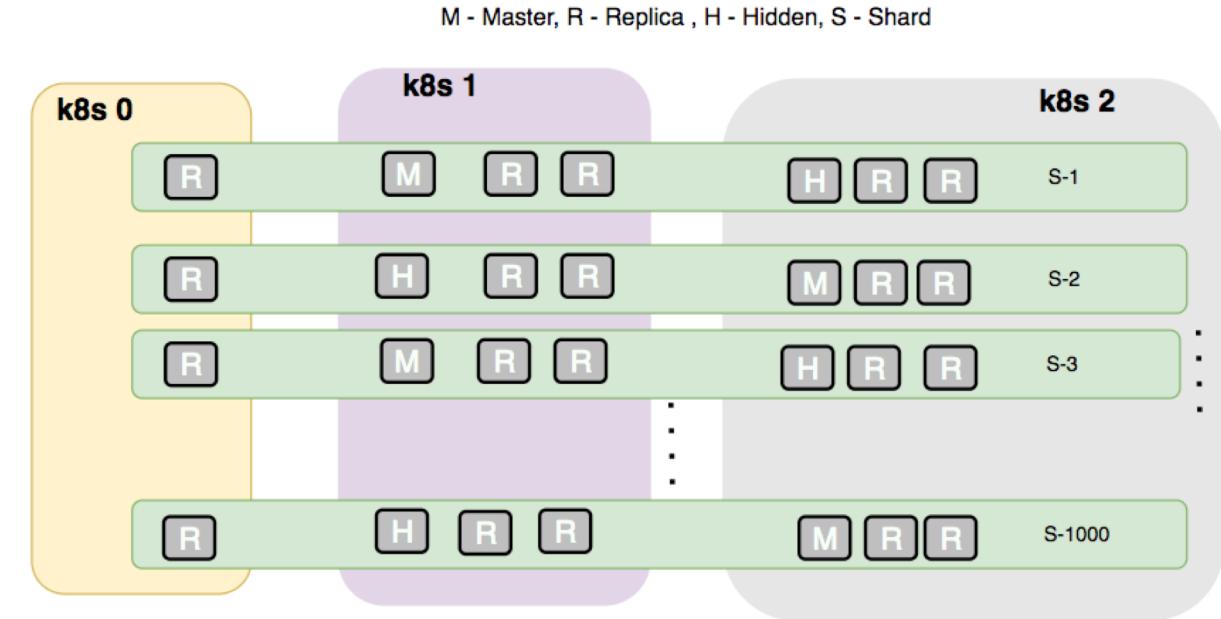
# Monitoring a Geo-Distributed Database

- NuData: A geo-distributed database developed at eBay
- Deployment: Thousands of pods across datacenters in eBay internal Kubernetes based cloud infrastructure
- Metrics monitoring: Prometheus 2.3
- Real-time monitoring for:
  - System operation
  - System development



# The Sharded Distributed Database

- The entire database framework supports many keyspaces
  - A keyspace consists of multiple shards
  - Each shard consists of multiple replicas (master, secondary, hidden)
  - Replicas in each shard are provisioned across datacenters



# Outline

- 
- Metrics capturing and aggregation
  - The Sharded and Federated Prometheus cluster
  - Query routing & UI integration
  - Monitoring experiences
  - Conclusions

# Metrics Being Monitored

- Metrics captured:
  - Throughputs
  - Latencies
  - Errors
  - Saturation (queuing)
  - State (master, replica, instance up/down)
- OS metrics (from Kubernetes Kubelet) and JVM metrics
- Custom metrics export: disk IO metrics from iostat
- Total metrics captured: **20M** metrics/scrape interval/per DC
  - Current scrape interval at 1 minute
  - Total storage size accumulated per day per DC: **195 GB**
  - Currently retain only **7** days of metrics data

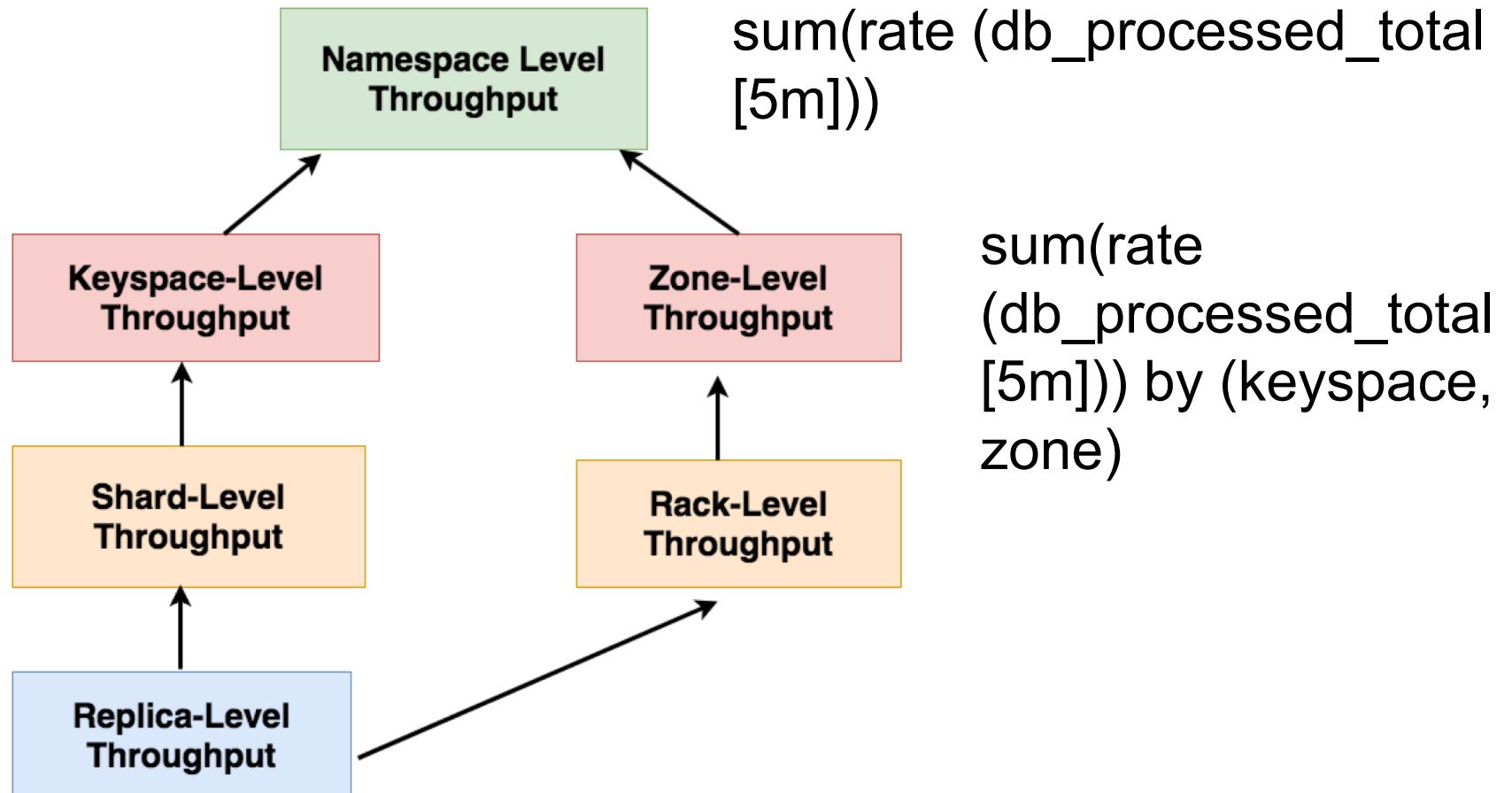
# Metric Labels & Labeling Hierarchy

- Prometheus on Kubernetes provides:
  - Dynamic discovery of target
  - Automatic label injection
  - Target filtering by auto-discovered labels
  - Re-labeling and label injection
- Metric labels:
  - Labels due to physical datacenter hierarchy:
    - pod, host, rack, datacenter
  - Labels due to logical database hierarchy:
    - replica, shard, keyspace
  - All of these labels are automatically injected by Prometheus from pod spec.

# Hierarchical Multi-Label Metrics Aggregation

`sum(rate(db_processed_total[5m]) by (keyspace))`

`sum(rate(db_processed_total[5m]) by (keyspace, shard))`



`sum(rate(db_processed_total[5m]))`

`sum(rate(db_processed_total[5m]) by (keyspace, zone))`

`db_processed_total (zone="dc1", host="...", rack="...", pod="...", keyspace="marketing", shard="78654", replica="21345", type="read")`

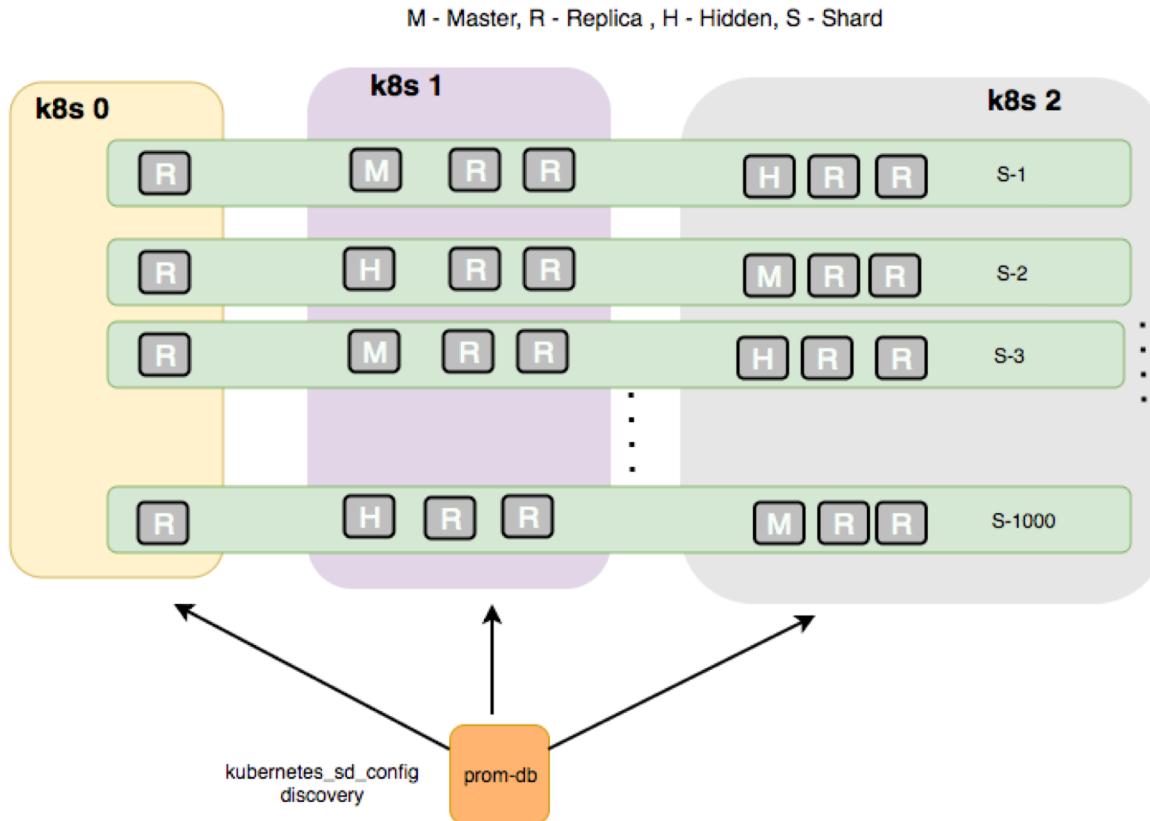
# The Need for a Scalable Cluster

- To collect 20M metric samples/minute from a single Prometheus becomes prohibitive
- In addition, >1200 recording rules in total to support real-time alerting and dashboards
  - One metric can be tied to multiple dashboards with different hierarchical aggregations
- The CPU consumption in Prometheus devoted to recording rules evaluation is much more significant compared to metrics scraping

# Outline

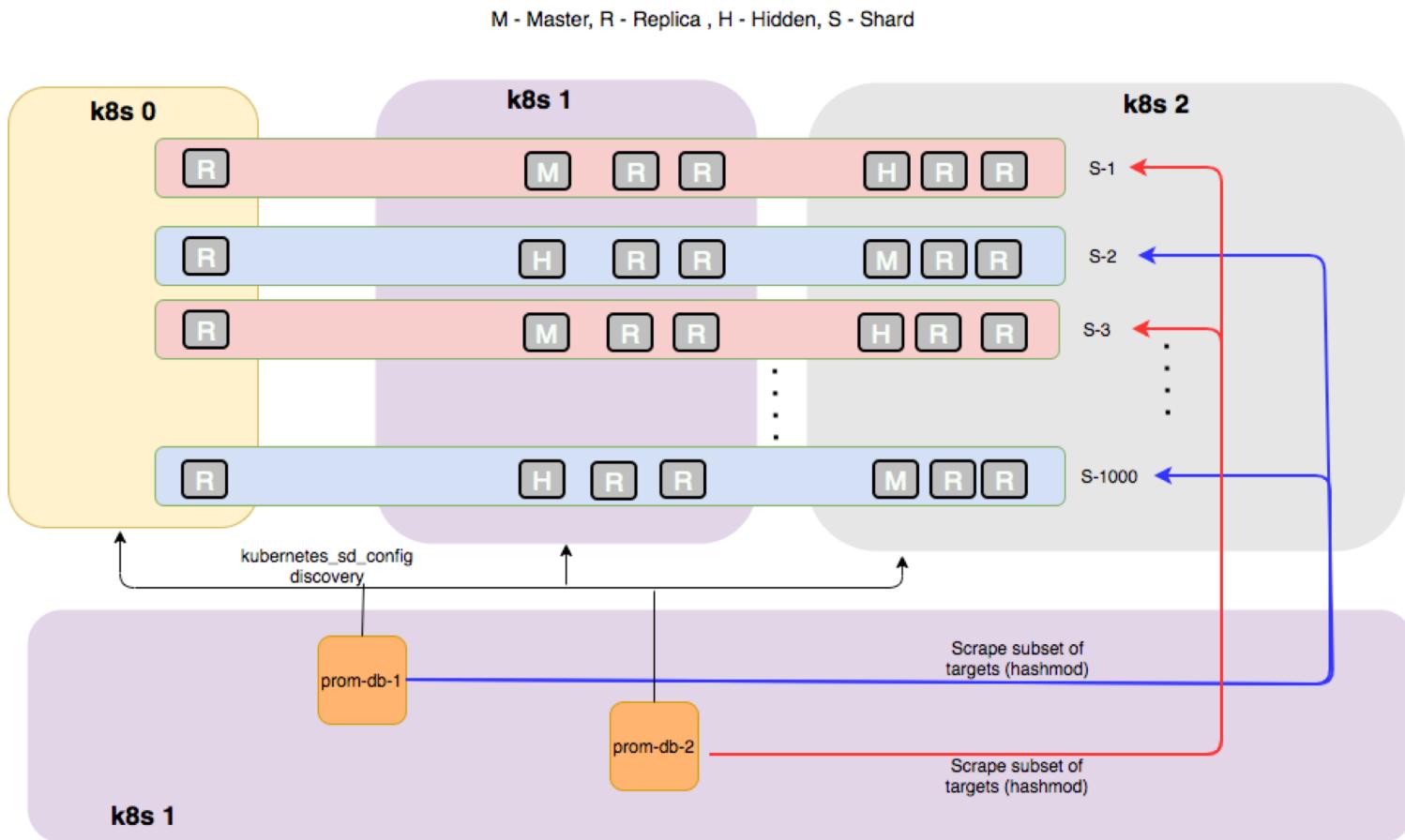
- Metrics capturing and aggregation
- The Sharded and Federated Prometheus cluster 
- Query routing & UI integration
- Monitoring experiences
- Conclusions

# The Distributed Database Being Monitored



- Highly available distributed database across three DCs
- Need to have a Prometheus setup to linearly scale with the targets being scraped

# Sharded Prometheus



- Sharded Prometheus setup
  - 2 prometheus shards illustrated
  - “Even” numbered data shards scraped by prom-db-1
  - And “odd” numbered ones scraped by prom-db-2
- Generalization: hash and modulus using Prometheus hashmod relabel config
  - Hashing done on data shard ID (& keyspace)

# Hashmod Relabel Config

```
...  
- source_labels: [__meta_kubernetes_pod_label_keyspace, __meta_kubernetes_pod_label_shard]  
  action: hashmod  
  modulus: __MODULUS__  
  target_label: hashmod  
- source_labels: [hashmod]  
  regex: __SHARD__  
  action: keep  
...
```

...

**prom-db-1.yaml**

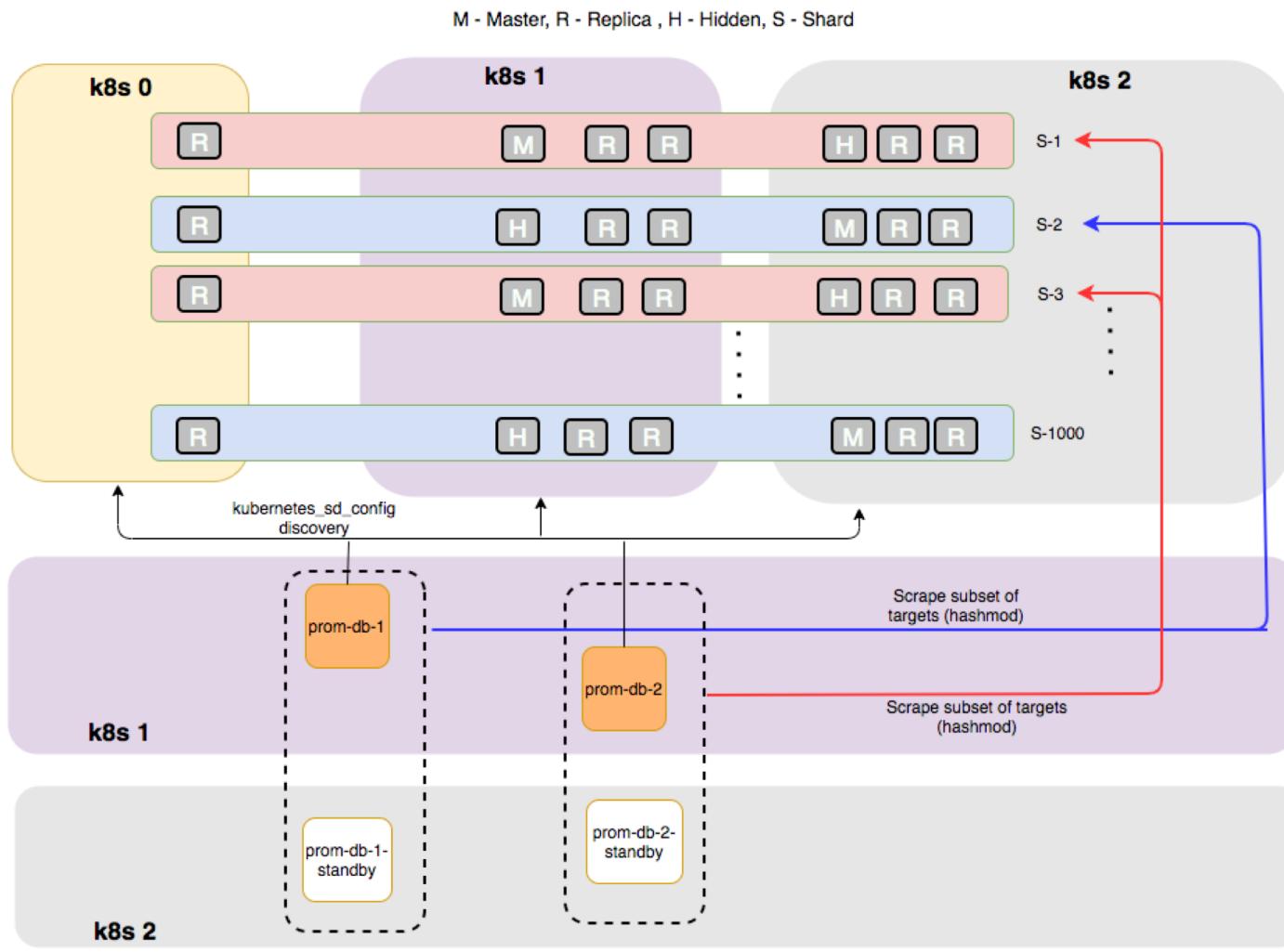
```
- source_labels: [__meta_kubernetes_pod_label_keyspace, __meta_kubernetes_pod_label_shard]  
  action: hashmod  
modulus: 2  
  target_label: hashmod  
- source_labels: [hashmod]  
regex: 0  
  action: keep  
...
```

**prom-db-2.yaml**

```
- source_labels: [__meta_kubernetes_pod_label_keyspace, __meta_kubernetes_pod_label_shard]  
  action: hashmod  
modulus: 2  
  target_label: hashmod  
- source_labels: [hashmod]  
regex: 1  
  action: keep  
...
```

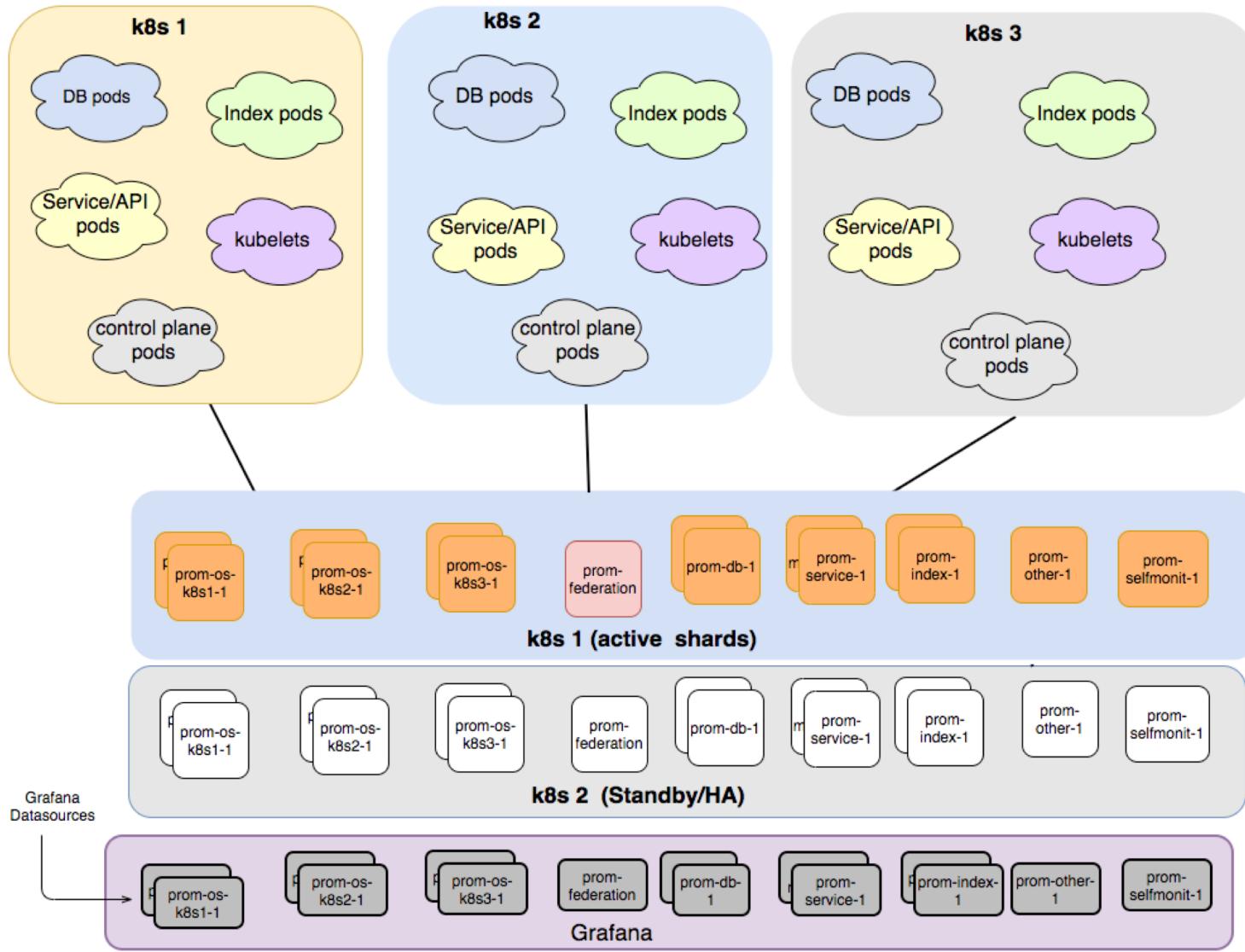
- Our deployment scripts takes yaml template as input and generates prometheus yaml files
- “**Keyspace + shardID**” is the input to the hashmod function
- A nice side effect: **all replicas of a data shard** are scraped by the same Prometheus server

# High Availability of Sharded Prometheus



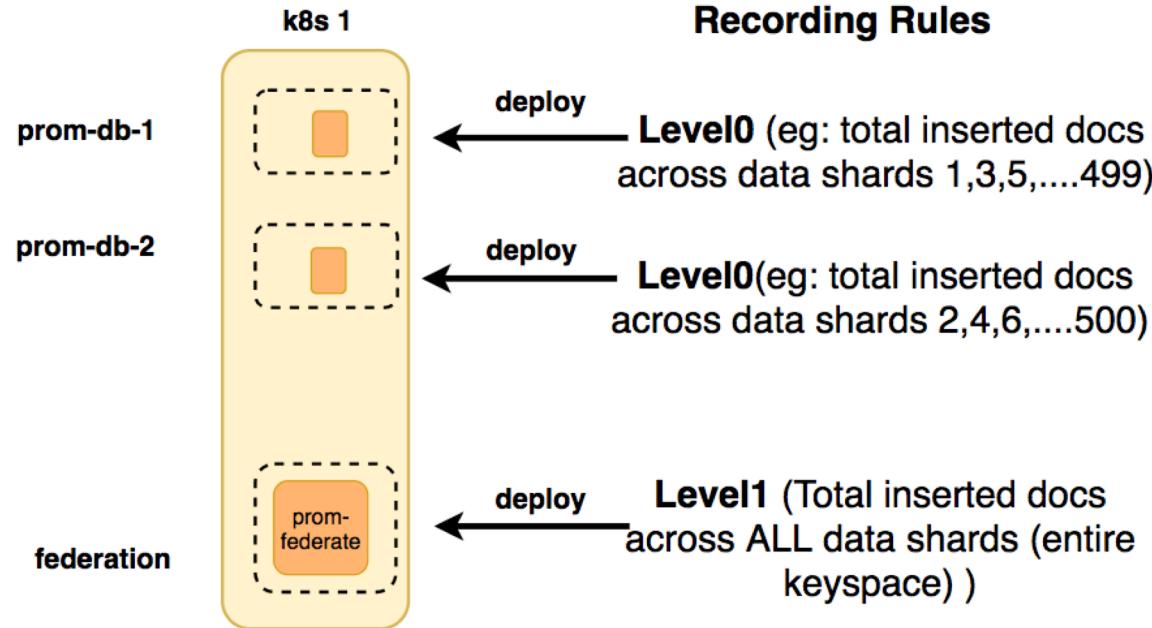
- High availability: deploying the same set of Prometheus servers (mirrored config) in two clusters
- Active/standby configuration for each Prometheus server pair
- The paired Prometheus servers share same config and scrape the same targets

# Sharded Setup with Multiple Categories



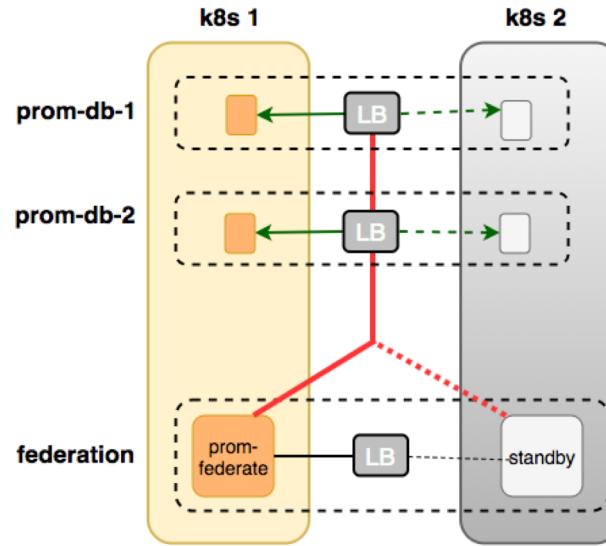
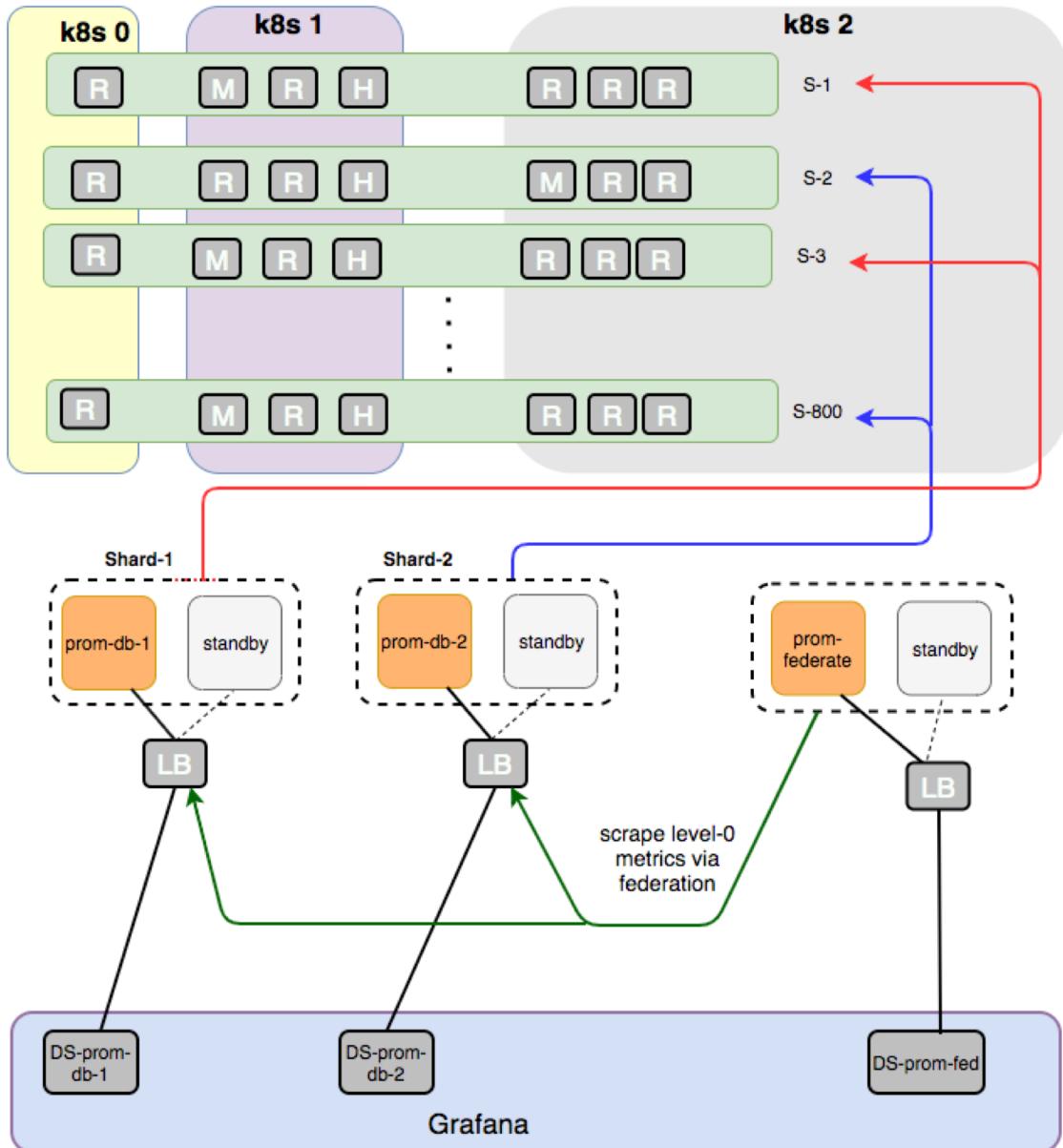
- Multiple categories: DB Service, DB Proxy, DB Engine, Indexing Engine, OS metrics
- A Prometheus cluster dedicated to each category
- Each Prometheus cluster has multiple shards
- Each Prometheus cluster is mirrored in a remote DC for HA

# Federation: Level 0 & 1 Recording Rules



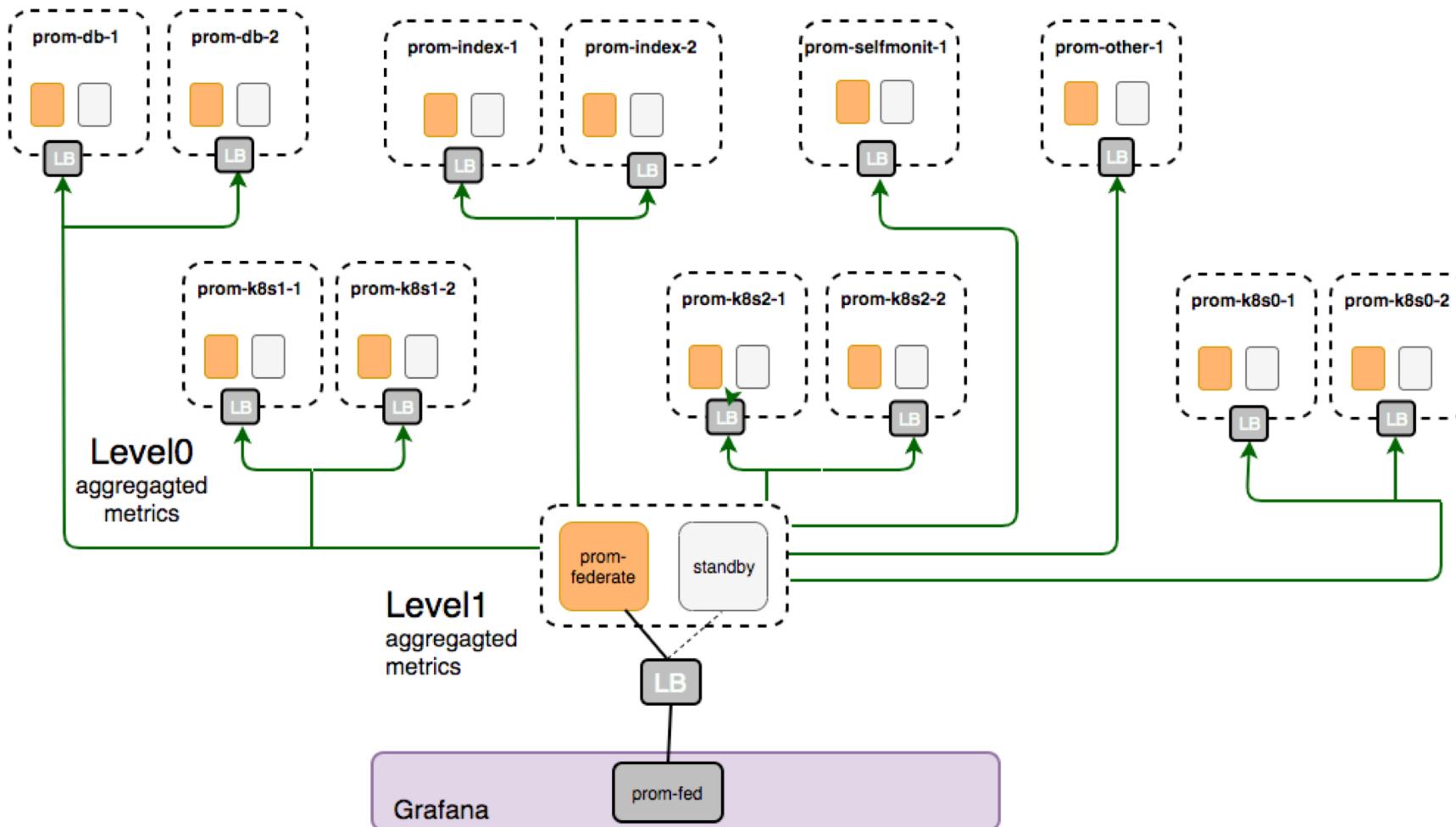
- record: level0:inserted\_document\_at\_keyspace\_level:rate5m  
expr: sum(rate(document\_total{state="inserted"}[5m])) BY(keysphere)
- record: level1:inserted\_document\_at\_keyspace\_level:rate5m  
expr: sum(level0:inserted\_document\_at\_keyspace\_level:rate5m) BY (keysphere)

# Highly Available Federated Setup



- Highly available federation server pair
- Highly available sharded Prometheus servers
- Each federation server scrapes the sharded Prometheus cluster via LB VIPs to provide HA transparently
- Grafana points to LB VIPs

# Complete Picture: Federated and Sharded Setup



- Automation scripts developed to deploy the full setup illustrated above

# Outline

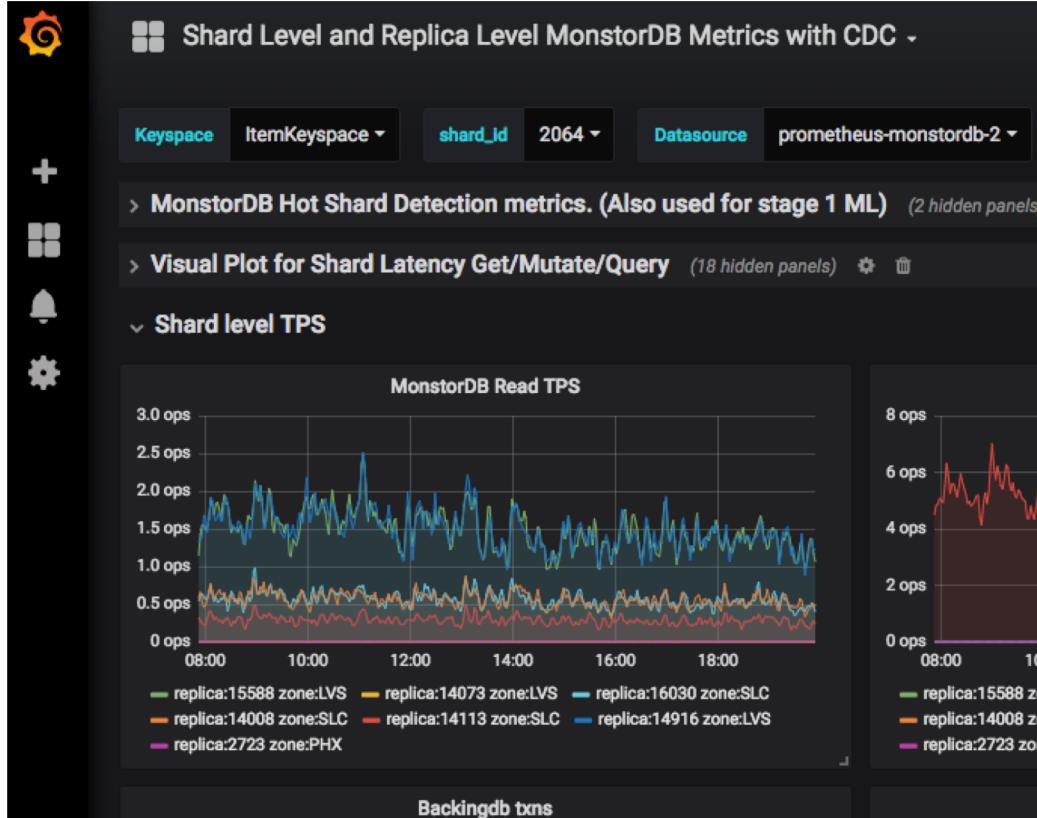
- Metrics capturing and aggregation
- The Sharded and Federated Prometheus cluster
- Query routing & UI integration
- Monitoring experiences
- Conclusions



# Query Routing

- The **hashmod** function determines which Prometheus sharded server should scrape (and store) metrics for a particular data shard
- A visualization framework (such as Grafana) requires auto-selection of the Prometheus data source to query metrics
- The solution:
  - A Federated Lookup/Routing table in our Prometheus cluster setup and
  - Templated variables and templated datasource in Grafana
  - **No changes needed to Prometheus and Grafana**

# Auto-Populated Datasource in Grafana



- Example: View metrics at shard-level.
  - The metrics are labeled with {replica id, shard id, keyspace} hierarchy
- {keyspace, shard} are chosen from the first two drop-downs
- The Datasource (a templated data source) is automatically populated

# Federated Lookup Routing Map

- A special recording rule (timeseries): **level0:routing\_map\_prom\_keyspace\_shard**
  - Based on the default ‘**up**’ for every scraped target
  - Deployed to each Prometheus shard
  - Scrapped by the Federation server
- **A Time-based Global Lookup Table:** mapping of {scraped targets, Prometheus shard} over time.

```
record: level0:routing_map_prom_keyspace_shard
expr: count(up{job=~"monstordb-*"})
BY (keyspace, shard, zone)
```

## Instant Timeseries Vector on prom-shard-1

```
level0:routing_map_prom_keyspace_shard
{keyspace="KS1",shard="1", zone="PHX"}
level0:routing_map_prom_keyspace_shard
{keyspace="KS2",shard="3", zone="PHX"}
```

## Instant Timeseries Vector on prom-shard-2

```
level0:routing_map_prom_keyspace_shard
{keyspace="KS1",shard="2", zone="LVS"}
level0:routing_map_prom_keyspace_shard
{keyspace="KS2",shard="4", zone="LVS"}
```

## Instant Timeseries Vector on prom-federation (with injected external label **ds\_name**)

```
level0:routing_map_prom_keyspace_shard {keyspace="KS1",shard="1", zone="PHX", ds_name="prom-shard-1"}
level0:routing_map_prom_keyspace_shard {keyspace="KS1",shard="2", zone="LVS", ds_name="prom-shard-2"}
level0:routing_map_prom_keyspace_shard {keyspace="KS2",shard="3", zone="PHX", ds_name="prom-shard-1"}
level0:routing_map_prom_keyspace_shard {keyspace="KS2",shard="4", zone="LVS", ds_name="prom-shard-2"}
```

# Template Variables based on Routing Map

Variables > Edit

General

Name	Keyspace
Type	Query
Label	optional display name

Query Options

Data source	prometheus-federation-0
Refresh	On Dashboard Load
Query	label_values(level0:routing_map_prom_keyspace_shard, keyspace)
Regex	/.*-(\*).*/
Sort	Alphabetical (asc)

The `label_values()` function is applied to `level0:routing_map_prom_keyspace_shard` at the Federation server

List of **keyspaces** =  
`label_values(level0:routing_map_prom_keyspace_shard, keyspace) = { KS1, KS2 }`

List of **shards** of KS1 =  
`label_values(level0:routing_map_prom_keyspace_shard {keyspace=KS1}, shard) = {1, 2 }`

List of **shards** of KS2 =  
`label_values(level0:routing_map_prom_keyspace_shard {keyspace=KS2}, shard) = {3, 4 }`

Variables > Edit

General

Name	shard_id
Type	Query
Label	optional display name

Query Options

Data source	prometheus-federation-0
Refresh	On Dashboard Load
Query	label_values(level0:routing_map_prom_keyspace_shard {keyspace="\$Keyspace"}, shard)
Regex	/.*-(\*).*/
Sort	Numerical (asc)

# Template Variables based on Routing Map

Variables > Edit

General

Name	datasource_hint	Type	Query
Label	optional display name	Hide	Variable

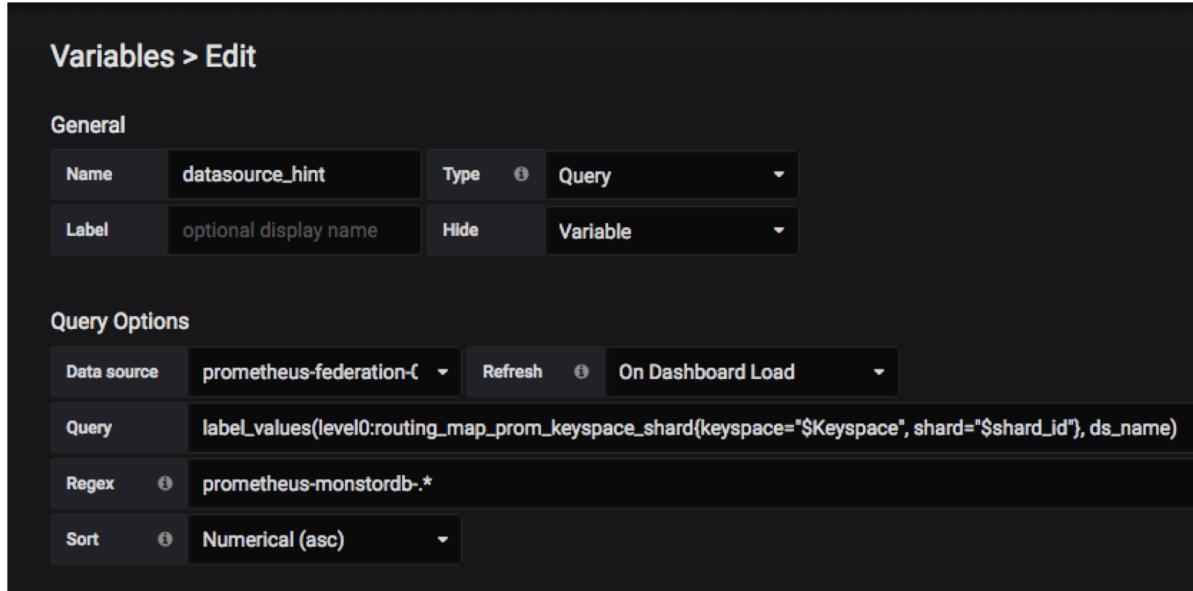
Query Options

Data source: prometheus-federation-1 Refresh: On Dashboard Load

Query: `label_values(level0:routing_map_prom_keyspace_shard(keyspace="$Keyspace", shard="$shard_id"), ds_name)`

Regex: `prometheus-monstordb-*`

Sort: Numerical (asc)



Variables > Edit

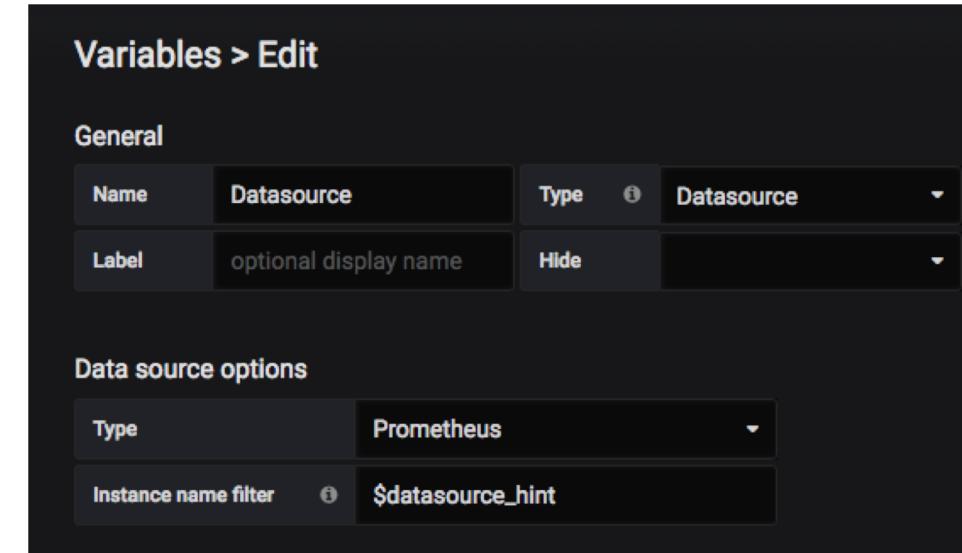
General

Name	Datasource	Type	Datasource
Label	optional display name	Hide	

Data source options

Type: Prometheus

Instance name filter: `$datasource_hint`



- Upon a keyspace and shard selection, retrieve the Datasource name by making query to Federated Prometheus server:
  - datasource\_hint =**  
`label_values(level0:routing_map_prom_keyspace_shard {keyspace=KS1, shard=1}, ds_name) = {prom-shard-1}`
  - The label values of `ds_name` match the datasource names we define in Grafana
- Grafana `datasource` type template variable cannot be directly of 'query' value type
- Hence the **hidden** variable `datasource_hint` is introduced to hold query value in the transient/hidden variable

# Outline

- Metrics capturing and aggregation
- The Sharded and Federated Prometheus cluster
- Query Routing & UI Integration
- Monitoring experiences
- Conclusions



# Self Monitoring (Monitoring of Monitoring)

- All Prometheus Instances are scraped by a **Selfmonit** Prometheus instance
  - The whole monitoring infrastructure metrics captured at one place
  - Easy comparison of metrics among Prometheus shards (e.g., load distribution is even or not)
- Self Monitoring instance is also HA

# Self Monitoring (Monitoring of Monitoring)



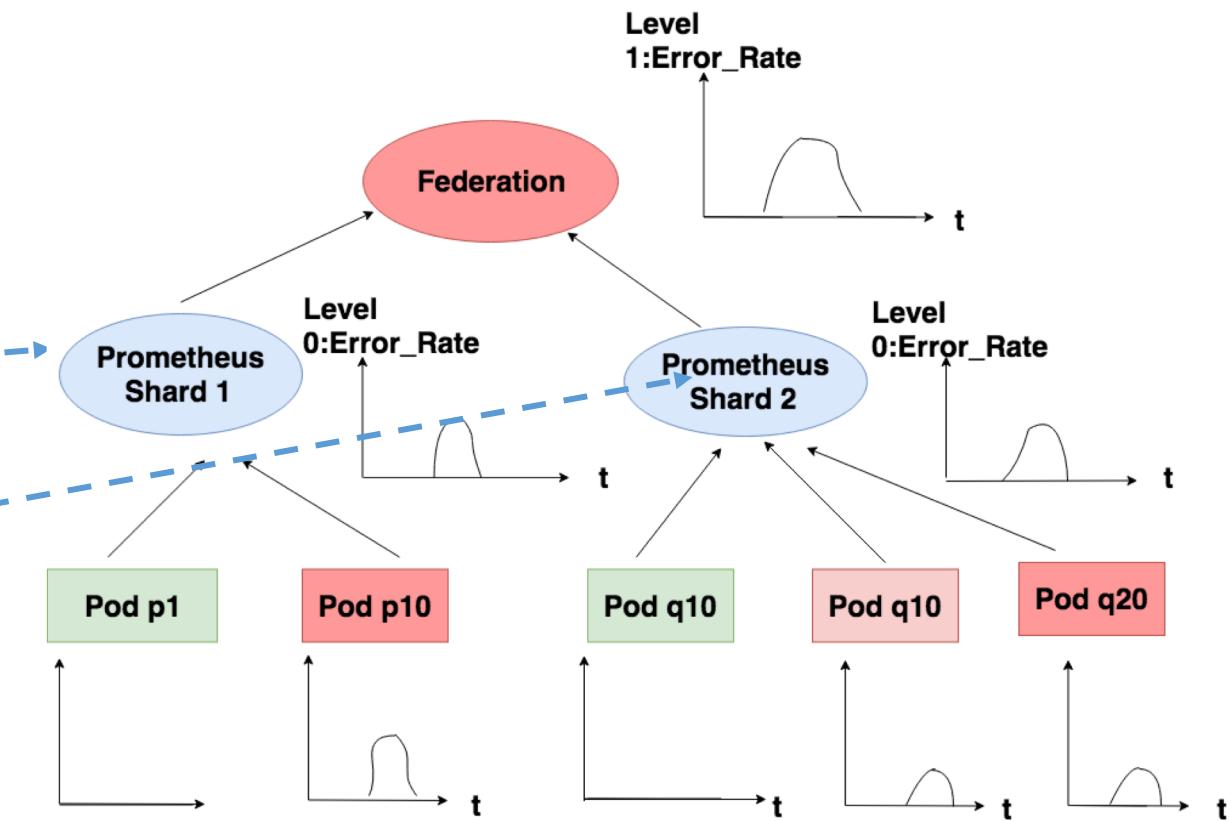
# Self Monitoring

- Example queries:
  - *Relative deviation in time series appended: sum*  
`(rate(prometheus_tsdb_head_samples_appended_total{container_name="prometheus"}[5m]) / rate(prometheus_tsdb_head_samples_appended_total{container_name="prometheus"}[5m] offset 30m)) by (instance, pod_name, prom_shard, prom_type, tess_cluster, tess_namespace)`
  - *Rule evaluation duration percentile :*  
`prometheus_rule_evaluation_duration_seconds{prom_type=~"prometheus.*", quantile="0.99"}`
- Example of alert rules:
  - Down Prometheus instances
  - Abrupt drop in time series appended
  - Abrupt drop in targets discovered
  - VIP endpoint reachability

# Pinpoint Troublesome Runtime Entities

- Example 1: Read error rate to a keyspace is now going up, what are the worst service pods that we need to investigate?

```
topk(5, sum(increase  
  (failed_responses_total{keyspace="K1",  
   method_name="READ"})[30m])) by  
  (pod_name)
```

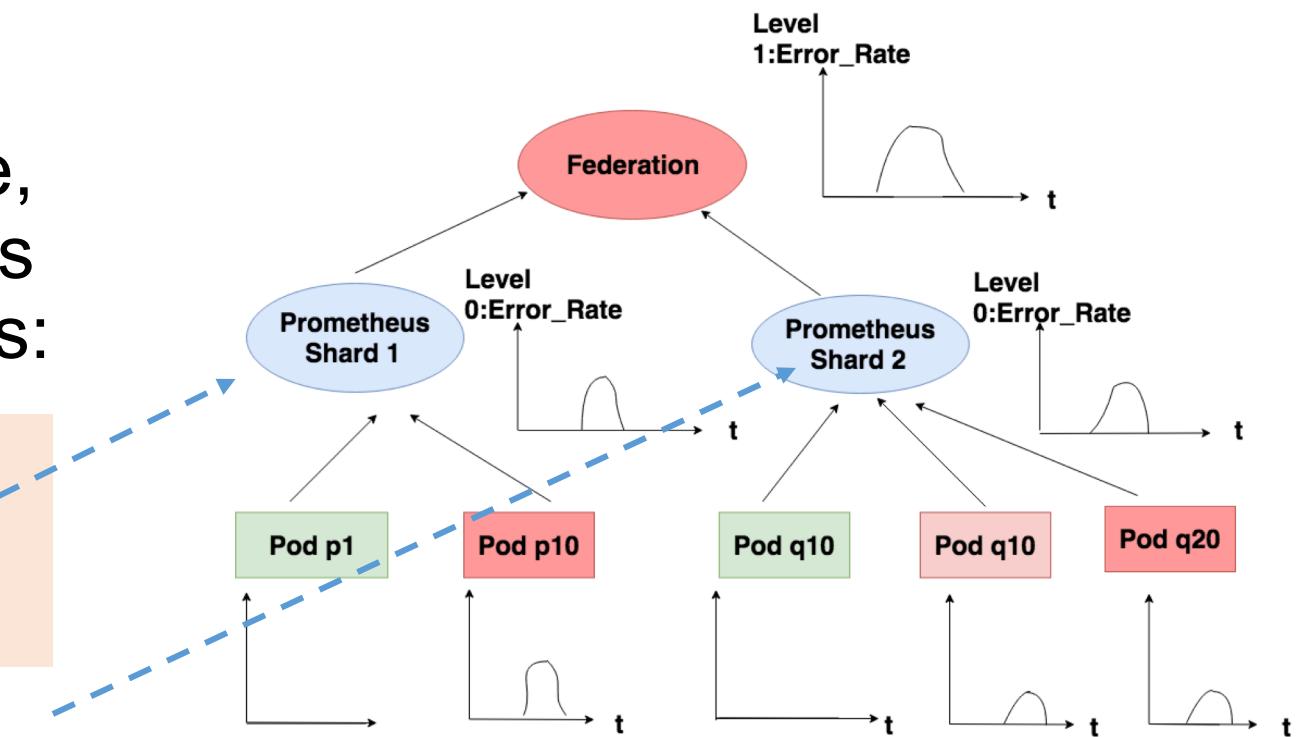


- Query over each sharded server and combine the top-k results

# Pinpoint Troublesome Runtime Entities (2)

- Alternatively, to have the following query to be plotted over the specified time range, on each sharded Prometheus server and inspect the results:

```
sum(rate  
(failed_responses_total{keyspace="K1",  
method_name="READ"})[5m]) by  
(pod_name)
```



- Plotting can be done in the Prometheus web console:
  - It can handle hundreds of time-series plots easily

# Special OS Metrics Aggregation

- A database pod has pod spec. to track the logical hierarchy {keyspace, shard, replica}
- Kublet exposes OS pod level metrics, but without labels from the application's pod spec attached

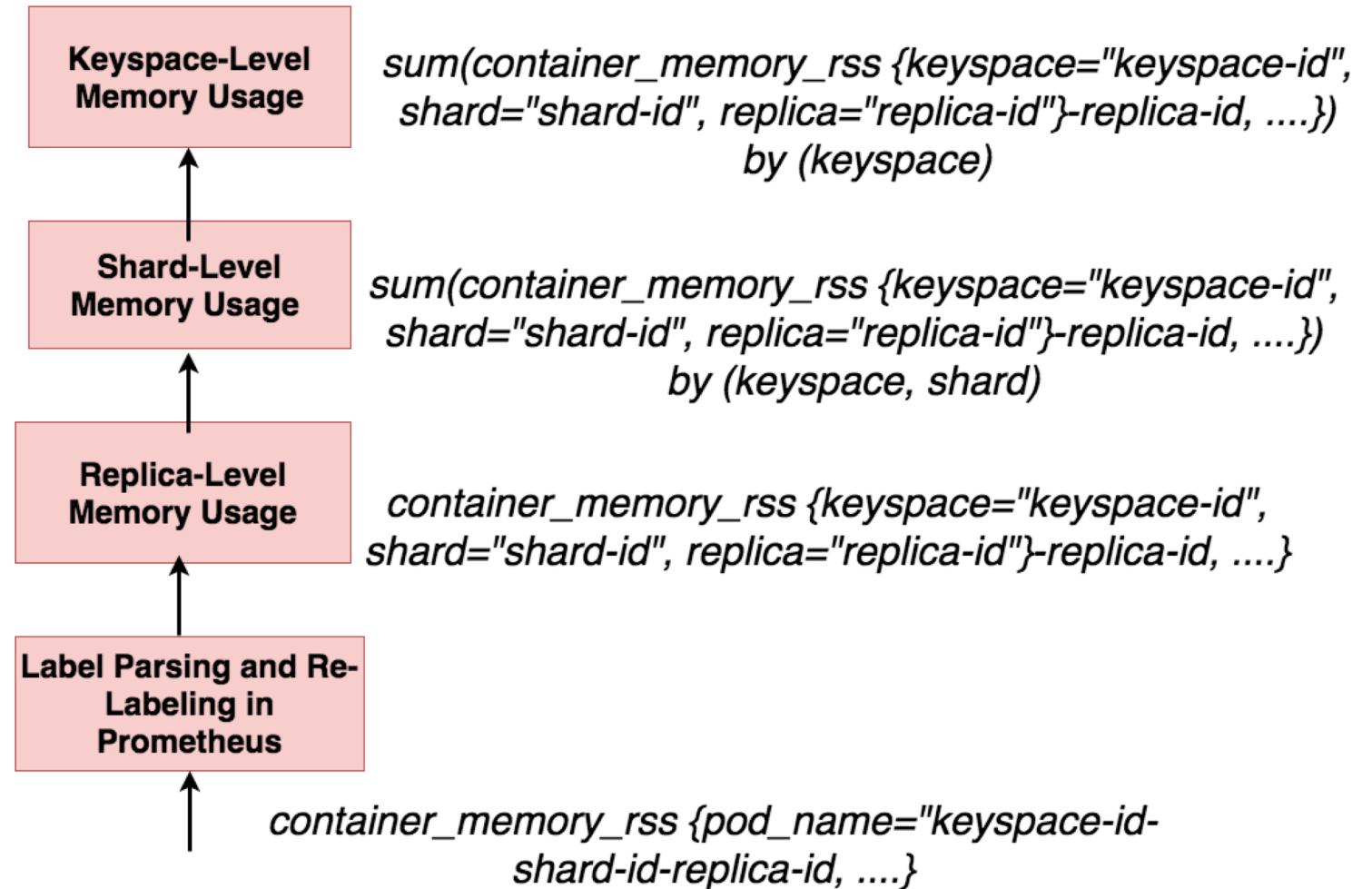
```
container_memory_rss (zone="...", host="...", rack="...", pod=pod_name) =  
150000
```

- Thus CPU/memory aggregation over logical hierarchy is not available
- Solution: label extraction and label injection, by leveraging the naming convention that we follow for database pods:

```
pod_name = keyspace-id + shard-id + replica-id + other information
```

# Special OS Metrics Aggregation

- Label extraction: to extract keyspace id, shard id and replica id from pod name
- Label injection: to inject labels: {keyspace, shard, and replica} into the OS metrics
- Thus OS aggregation over logical aggregation is now available



# Alert Summarization

- A Prometheus alert has the labels from the recording rule evaluated
- Summarization on alerts:
  - Over severity {critical, high, warning}
  - Over logical hierarchy
  - Over physical hierarchy
- For both historical alerts and active firing (not resolved) alerts
- Solution: to store and index the received alerts into Elasticsearch

# Alert Dashboard in Kibana

The screenshot displays the Kibana Alert Dashboard interface, which includes the following components:

- Search Bar:** Located at the top left, with a placeholder "Search... (e.g. status:200 AND extension:PHP)".
- Filter Buttons:** CRITICAL, OPEN, PROD, and Add a filter +.
- Time Range:** Last 1 hour, with options for Full screen, Share, Clone, Edit, Auto-refresh, Options, and Actions.
- [Alerts] Hierarchy Control Dropdown:** A complex dropdown menu for filtering alerts based on Component, Tess\_namespace, Tess\_Host, Pod, Replica, Shard, Keyspace, Alertname, Severity, and State. It features "Select..." dropdowns and "Apply changes" buttons.
- [Alerts] Top 20 Alerts Piechart:** A pie chart showing the distribution of alert types. The legend includes:
  - MonstorDBDBService...
  - MongoDBNodeDown
  - MonstorDBNodeDown...
  - MonstorDBOverallDB...
  - MonstorDBOverallDB...
  - MonstorDBOverallDB...
  - MonstorDBOverallDB...
  - MonstorDBOverallDB...
  - MonstorDBOverallDB...
  - ServiceNodeProxyMa...
- [Alerts] Top 20 Alerts:** A bar chart showing the count of alerts for each type. The legend indicates Count values: 0 - 50 (green), 50 - 75 (yellow), and 75 - 100 (red). The data shows a single large green bar reaching approximately 600.
- [Alerts] Alert count:** A gauge chart showing the total alert count as 771.
- [Alerts] Hierarchy group by alert:** A table showing the count of alerts grouped by monitor\_db: Component. The data is as follows:

monitor_db: Component	Count
MonstorDBDBServiceCallLatencyInMutateTooLarge	615
MonstorDBNodeDownByPrometheusScan	49
MonstorDBOverallDBTransactionTimeTooLarge	31
MonstorDBOverallDBCommitTransactionTimeTooLarge	8
MonstorDBOverallDBTimeTooLarge	8

Annotations with arrows point to specific features:

- An orange arrow points to the search bar and filter buttons, labeled **Resolved/Active Firing**.
- A purple arrow points to the time range and hierarchy dropdown, labeled **Historical or Real-time**.
- A green arrow points to the pie chart and bar chart, labeled **Logical/Physical Hierarchies**.
- A red arrow points to the gauge chart, labeled **Alert Severity**.

# Outline

- Metrics capturing and aggregation
- The sharded and federated Prometheus cluster
- Query routing & UI integration
- Monitoring experiences
- Conclusions



# Conclusions

- Prometheus itself is deployed as a standalone single process
- We have developed a horizontally scalable, sharded, and federated Prometheus monitoring cluster from Prometheus binary distribution with full automation scripts, without modifying its source code
- The scalable monitoring cluster allows us to have real-time dashboards and real-time alerts over the hierarchically aggregated metrics

Thank You !

Q & A



**KubeCon**

---



**CloudNativeCon**

---

North America 2018

