COLLECTING OPERATIONAL METRICS ACROSS 5,000 NAMESPACES

Using the Metering project of the Operator Framework

Rob Szumski
Product Manager, OpenShift

Chance Zibolski
Engineer/Team Lead, OpenShift

Talk Overview

- Operator-Framework overview
- Metering goals and use-cases
- Metering technical architecture
- Insights and findings from 5,000 namespaces
- Demo

OPERATOR-FRAMEWORK OVERVIEW

Operator Framework

OPERATOR SDK

Build Operators - no specialized knowledge of Kubernetes API required

OPERATOR LIFECYCLE MANAGER

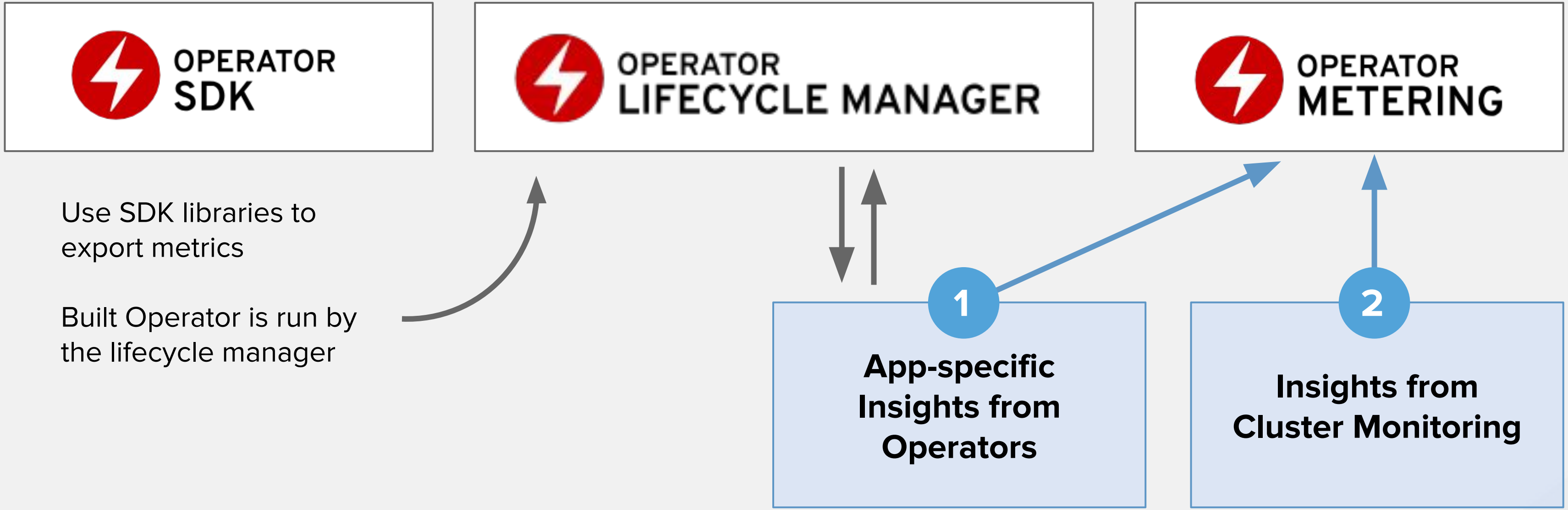
Install, update, and manage Operators and their dependencies

OPERATOR METERING

Enable usage reporting for Operators and resources within Kubernetes

https://github.com/operator-framework

Metering Goals


OPERATOR SDK
OPERATOR LIFECYCLE MANAGER
OPERATOR METERING
1
App-specific Insights from Operators
2
Insights from Cluster Monitoring

Use SDK libraries to export metrics

Built Operator is run by the lifecycle manager

GOALS AND USE-CASES

Chargeback/Showback Goals

ENCOURAGE CORRECT BEHAVIOR

Push teams towards smart resource usage

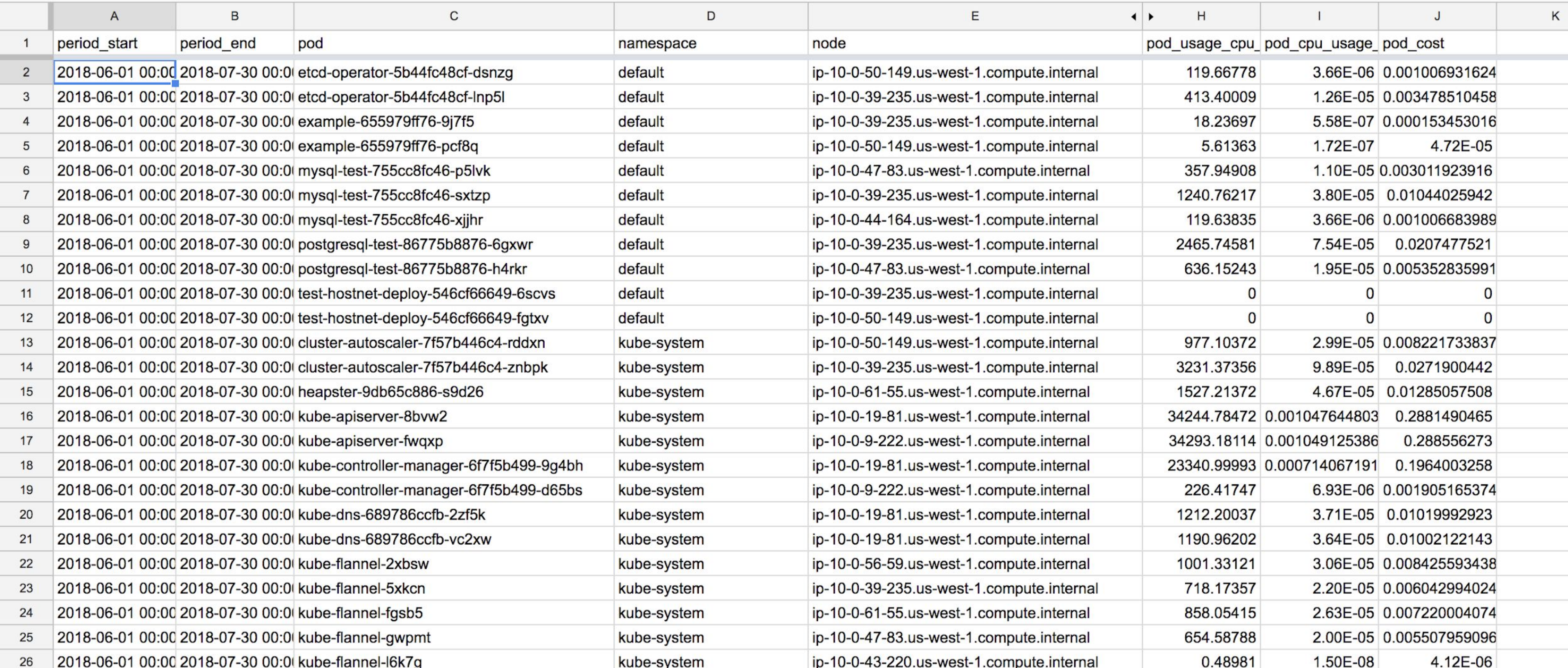
USE CLUSTER MONITORING

The cluster already tracks this information to function

POST PROCESS

Embrace further customization based on your business' needs

End Result: Usage Report



	A	B	C	D	E	H	I	J	K
1	period_start	period_end	pod	namespace	node	pod_usage_cpu_	pod_cpu_usage_	pod_cost	
2	2018-06-01 00:00	2018-07-30 00:0	etcd-operator-5b44fc48cf-dsnzg	default	ip-10-0-50-149.us-west-1.compute.internal	119.66778	3.66E-06	0.001006931624	
3	2018-06-01 00:00	2018-07-30 00:0	etcd-operator-5b44fc48cf-lnp5l	default	ip-10-0-39-235.us-west-1.compute.internal	413.40009	1.26E-05	0.003478510458	
4	2018-06-01 00:00	2018-07-30 00:0	example-655979ff76-9j7f5	default	ip-10-0-39-235.us-west-1.compute.internal	18.23697	5.58E-07	0.000153453016	
5	2018-06-01 00:00	2018-07-30 00:0	example-655979ff76-pcf8q	default	ip-10-0-50-149.us-west-1.compute.internal	5.61363	1.72E-07	4.72E-05	
6	2018-06-01 00:00	2018-07-30 00:0	mysql-test-755cc8fc46-p5lvk	default	ip-10-0-47-83.us-west-1.compute.internal	357.94908	1.10E-05	0.003011923916	
7	2018-06-01 00:00	2018-07-30 00:0	mysql-test-755cc8fc46-sxtzp	default	ip-10-0-39-235.us-west-1.compute.internal	1240.76217	3.80E-05	0.01044025942	
8	2018-06-01 00:00	2018-07-30 00:0	mysql-test-755cc8fc46-xjjhr	default	ip-10-0-44-164.us-west-1.compute.internal	119.63835	3.66E-06	0.001006683989	
9	2018-06-01 00:00	2018-07-30 00:0	postgresql-test-86775b8876-6gxwr	default	ip-10-0-39-235.us-west-1.compute.internal	2465.74581	7.54E-05	0.0207477521	
10	2018-06-01 00:00	2018-07-30 00:0	postgresql-test-86775b8876-h4rkr	default	ip-10-0-47-83.us-west-1.compute.internal	636.15243	1.95E-05	0.005352835991	
11	2018-06-01 00:00	2018-07-30 00:0	test-hostnet-deploy-546cf66649-6scvs	default	ip-10-0-39-235.us-west-1.compute.internal	0	0	0	
12	2018-06-01 00:00	2018-07-30 00:0	test-hostnet-deploy-546cf66649-fgtxv	default	ip-10-0-50-149.us-west-1.compute.internal	0	0	0	
13	2018-06-01 00:00	2018-07-30 00:0	cluster-autoscaler-7f57b446c4-rddxn	kube-system	ip-10-0-50-149.us-west-1.compute.internal	977.10372	2.99E-05	0.008221733837	
14	2018-06-01 00:00	2018-07-30 00:0	cluster-autoscaler-7f57b446c4-znbpk	kube-system	ip-10-0-39-235.us-west-1.compute.internal	3231.37356	9.89E-05	0.0271900442	
15	2018-06-01 00:00	2018-07-30 00:0	heapster-9db65c886-s9d26	kube-system	ip-10-0-61-55.us-west-1.compute.internal	1527.21372	4.67E-05	0.01285057508	
16	2018-06-01 00:00	2018-07-30 00:0	kube-apiserver-8bvw2	kube-system	ip-10-0-19-81.us-west-1.compute.internal	34244.78472	0.001047644803	0.2881490465	
17	2018-06-01 00:00	2018-07-30 00:0	kube-apiserver-fwqxp	kube-system	ip-10-0-9-222.us-west-1.compute.internal	34293.18114	0.001049125386	0.288556273	
18	2018-06-01 00:00	2018-07-30 00:0	kube-controller-manager-6f7f5b499-9g4bh	kube-system	ip-10-0-19-81.us-west-1.compute.internal	23340.99993	0.000714067191	0.1964003258	
19	2018-06-01 00:00	2018-07-30 00:0	kube-controller-manager-6f7f5b499-d65bs	kube-system	ip-10-0-9-222.us-west-1.compute.internal	226.41747	6.93E-06	0.001905165374	
20	2018-06-01 00:00	2018-07-30 00:0	kube-dns-689786ccfb-2zf5k	kube-system	ip-10-0-19-81.us-west-1.compute.internal	1212.20037	3.71E-05	0.01019992923	
21	2018-06-01 00:00	2018-07-30 00:0	kube-dns-689786ccfb-vc2xw	kube-system	ip-10-0-19-81.us-west-1.compute.internal	1190.96202	3.64E-05	0.01002122143	
22	2018-06-01 00:00	2018-07-30 00:0	kube-flannel-2xbsw	kube-system	ip-10-0-56-59.us-west-1.compute.internal	1001.33121	3.06E-05	0.008425593438	
23	2018-06-01 00:00	2018-07-30 00:0	kube-flannel-5xkcn	kube-system	ip-10-0-39-235.us-west-1.compute.internal	718.17357	2.20E-05	0.006042994024	
24	2018-06-01 00:00	2018-07-30 00:0	kube-flannel-fgsb5	kube-system	ip-10-0-61-55.us-west-1.compute.internal	858.05415	2.63E-05	0.007220004074	
25	2018-06-01 00:00	2018-07-30 00:0	kube-flannel-gwpmt	kube-system	ip-10-0-47-83.us-west-1.compute.internal	654.58788	2.00E-05	0.005507959096	
26	2018-06-01 00:00	2018-07-30 00:0	kube-flannel-l6k7g	kube-system	ip-10-0-43-220.us-west-1.compute.internal	0.48981	1.50E-08	4.12E-06	

Out of the box Reports

CPU

Memory

Storage

×

Request

Actual Usage

Utilization %

×

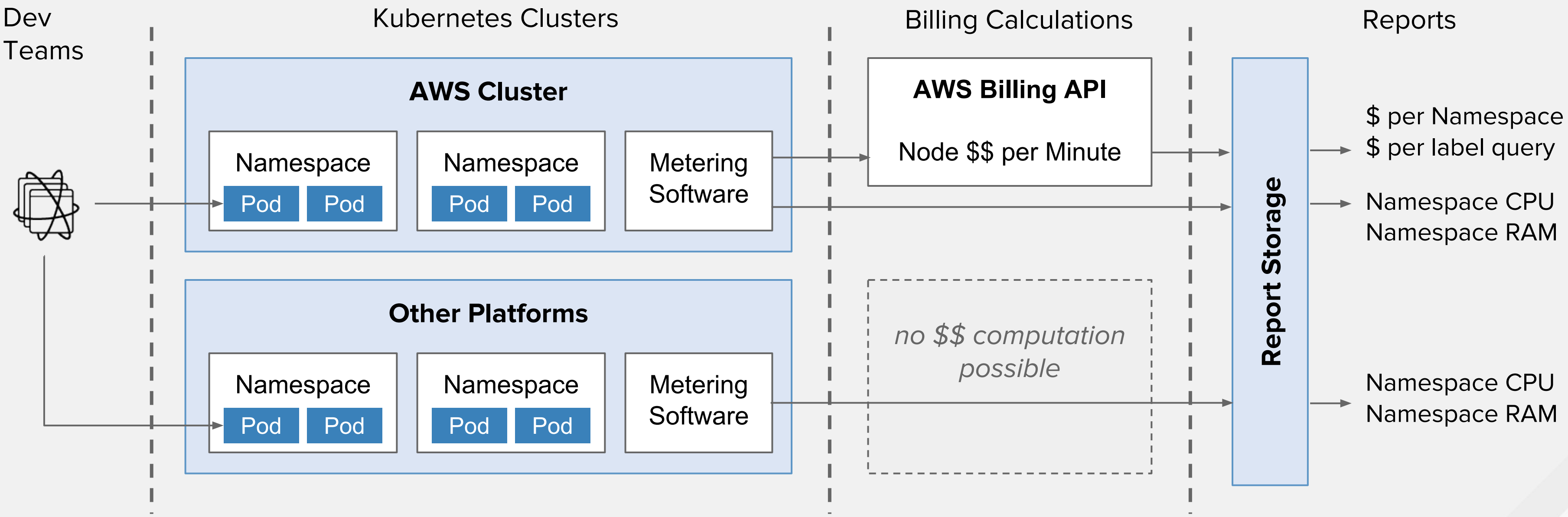
Pod

Namespace

Node

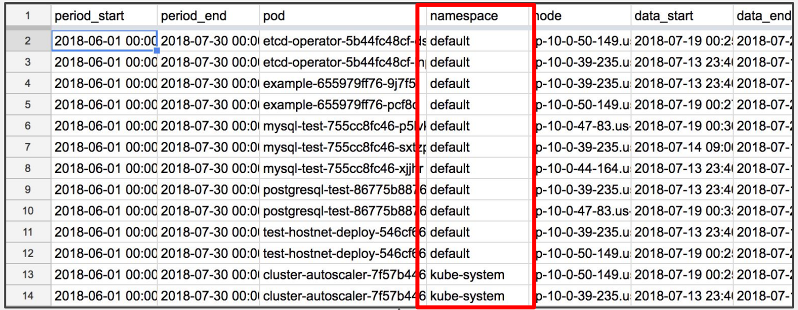
Cluster

Metering with Multiple Clusters

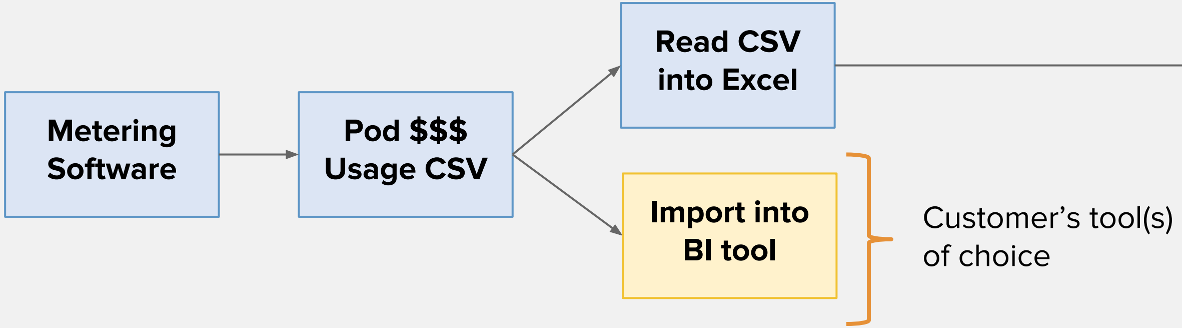


Use-Case: AWS showback by team

- Team has three projects
  - Development
  - Staging
  - Production
- Budget of $10,000 for all three



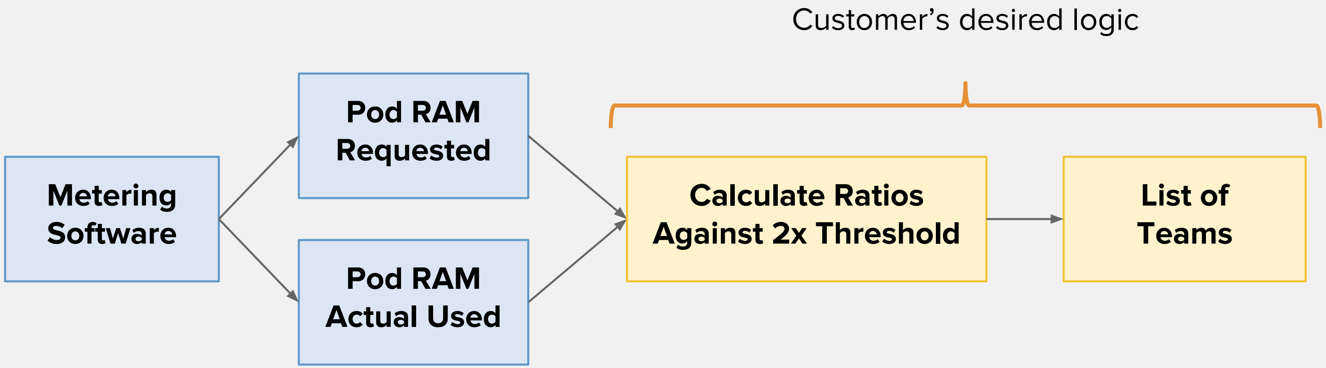
1	period_start	period_end	pod	namespace	node	data_start	data_end
2	2018-06-01 00:00	2018-07-30 00:0	etcd-operator-5b44fc48cf-d	default	p-10-0-50-149.u	2018-07-19 00:2	2018-07-2
3	2018-06-01 00:00	2018-07-30 00:0	etcd-operator-5b44fc48cf-	default	p-10-0-39-235.u	2018-07-13 23:4	2018-07-1
4	2018-06-01 00:00	2018-07-30 00:0	example-655979ff76-9j7f5	default	p-10-0-39-235.u	2018-07-13 23:4	2018-07-1
5	2018-06-01 00:00	2018-07-30 00:0	example-655979ff76-pcf8c	default	p-10-0-50-149.u	2018-07-19 00:2	2018-07-2
6	2018-06-01 00:00	2018-07-30 00:0	mysql-test-755cc8fc46-p5l	default	p-10-0-47-83.us	2018-07-19 00:3	2018-07-2
7	2018-06-01 00:00	2018-07-30 00:0	mysql-test-755cc8fc46-sxt	default	p-10-0-39-235.u	2018-07-14 09:0	2018-07-1
8	2018-06-01 00:00	2018-07-30 00:0	mysql-test-755cc8fc46-xjjh	default	p-10-0-44-164.u	2018-07-13 23:4	2018-07-1
9	2018-06-01 00:00	2018-07-30 00:0	postgresql-test-86775b8876	default	p-10-0-39-235.u	2018-07-13 23:4	2018-07-1
10	2018-06-01 00:00	2018-07-30 00:0	postgresql-test-86775b8876	default	p-10-0-47-83.us	2018-07-19 00:3	2018-07-2
11	2018-06-01 00:00	2018-07-30 00:0	test-hostnet-deploy-546cf66	default	p-10-0-39-235.u	2018-07-13 23:4	2018-07-1
12	2018-06-01 00:00	2018-07-30 00:0	test-hostnet-deploy-546cf66	default	p-10-0-50-149.u	2018-07-19 00:2	2018-07-2
13	2018-06-01 00:00	2018-07-30 00:0	cluster-autoscaler-7f57b446	kube-system	p-10-0-50-149.u	2018-07-19 00:2	2018-07-2
14	2018-06-01 00:00	2018-07-30 00:0	cluster-autoscaler-7f57b446	kube-system	p-10-0-39-235.u	2018-07-13 23:4	2018-07-1



Customer's tool(s) of choice

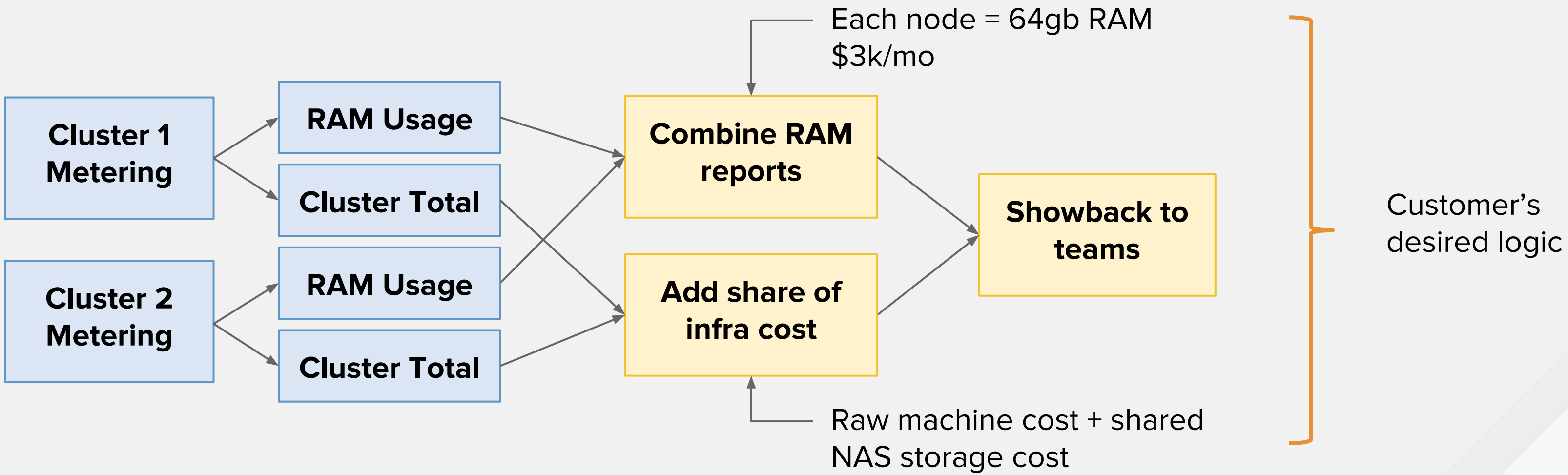
Use-Case: Shame Underutilization

- 15 teams using the cluster's finite resources
- Shame teams that are requesting over 2x what they are actually using
- Flexible granularity per cluster, per region, all clusters



Use-Case: Two clusters on bare metal

- Team is running in production across two providers
- How much RAM are we using across all clusters?
- Customer wants provide $/node and split infra node cost amongst all teams



Cluster 1 Metering

Cluster 2 Metering

RAM Usage

Cluster Total

RAM Usage

Cluster Total

Combine RAM reports

Add share of infra cost

Showback to teams

Each node = 64gb RAM
$3k/mo

Raw machine cost + shared
NAS storage cost

Customer's desired logic

Capacity Planning

- Leverage metering as long term storage of key metrics
- Compute reports over many weeks, months or years
- Out of the box “utilization” reports which indicate % of cluster capacity consumed from a pod or namespace as well as overall cluster utilization reports to allow both low-level and high-level views
- Openshift Online leverages metering to determine how much capacity is available in our Openshift clusters.

RED HAT® OPENSHIFT Online = Huge multi-tenant, 5k+ namespace clusters

Metering With Custom Metrics

- Example use-cases:
  - Telemetry
  - Licensing
  - Usage based billing
- How we use this at Red Hat
  - Reporting on Openshift upgrade telemetry metrics (version distribution, # of alerts fired during, etc)
  - Evaluating usage based billing of Red Hat products

TECHNICAL ARCHITECTURE

Tech stack

- Presto (prestodb.io)
  - Distributed SQL query engine designed for interactive analytics processing
  - Allows querying data where it lives; no need to bulk import all your data if it already lives in a database supported by Presto
  - Allows joining data from multiple datastores including Hive, Postgresql, Cassandra and more
  - Written by Facebook, used by many large internet companies like Airbnb and Dropbox
- Apache Hive
  - A data warehouse project that builds on top of Hadoop
  - Supports storing data into HDFS, S3, local file systems
  - Stores metadata about where data lives for Presto

Operators

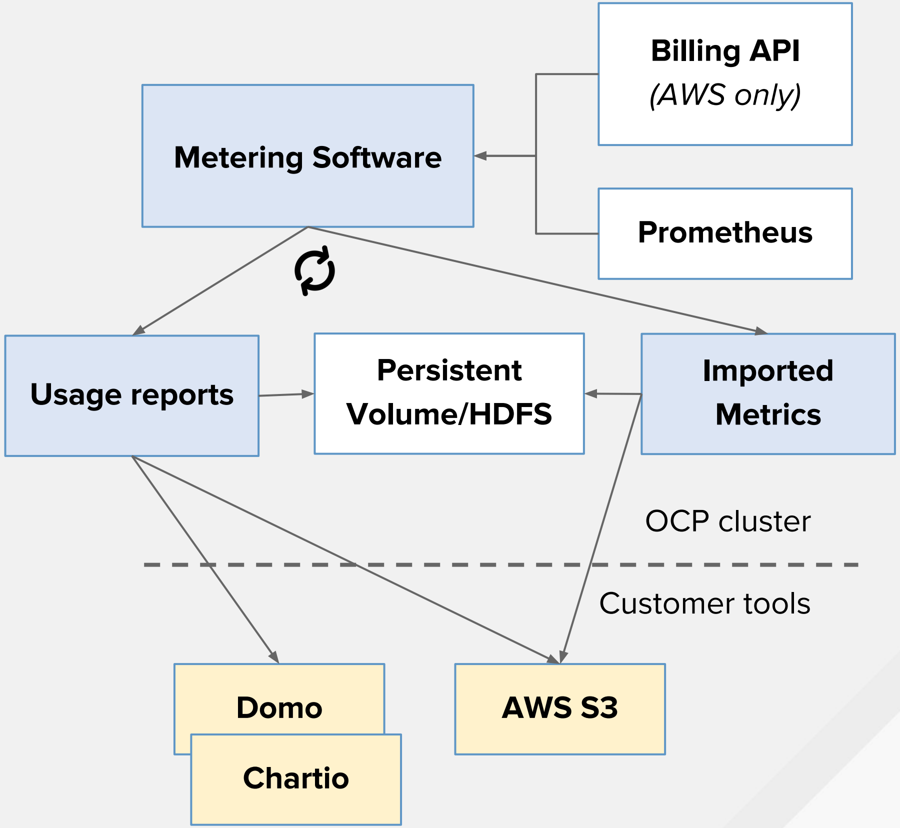
- Reporting operator
  - Interacts with Presto, Hive, Prometheus and cloud APIs to tie everything together
  - Imports metrics from Prometheus into Presto
    - Presto doesn’t have a native Prometheus Connector to expose metrics as tables natively, but this is an option we’re evaluating.
  - Executes SQL queries against Presto to produce Reports
- Metering operator
  - Manages the life cycle of all the other components including Presto, Hive, HDFS, and reporting-operator
  - Turns configuration into deployments, secrets, services and handles tasks like migrations and upgrades

Integrate with existing systems

- Use the cluster’s monitoring stack for insight into cluster
  - Installed out of the box in OpenShift 3.11+
- Do the heavy lifting calculations on the cluster
  - Doesn’t require using a SaaS
  - Works in bare metal environments
- Store scheduled reports into external storage location (PV (gluster,nfs), S3 bucket, HDFS)
  - Longer term: support storing data in Postgresql/Mysql also
- Work with pre-existing data
  - By leveraging Presto we can avoid having to do an “import” of data in many cases (eg: AWS detailed billing reports)

Metering Dataflow

- The metering components import metrics from Prometheus, storing them in S3, HDFS, or a PersistentVolume
- Metering in the background produces reports
- The report results are generated they are stored in S3, HDFS, or a PersistentVolume



Creating a report

- Creating a report is just writing a YAML file to reference your ReportGenerationQuery CRD by name:

```yaml
apiVersion: metering.openshift.io/v1alpha1
kind: Report
metadata:
  name: namespace-cpu-usage-hourly
spec:
  generationQuery: "namespace-cpu-usage"
  schedule:
    period: "hourly"
```

Example report query

```
apiVersion: metering.openshift.io/v1alpha1
kind: ReportGenerationQuery
metadata:
 name: "namespace-cpu-usage"
spec:
  columns: [ … ]
  reportQueries:
  - "pod-cpu-usage-raw"
  query: |
   SELECT
     timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}' AS period_start,
     timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS period_end,
     namespace,
     sum(pod_usage_cpu_core_seconds) as pod_usage_cpu_core_seconds
   FROM {| generationQueryViewName "pod-cpu-usage-raw" |}
   WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart | prestoTimestamp |}'
   AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}'
   AND dt >= '{| default .Report.ReportingStart .Report.Inputs.ReportingStart | prometheusMetricPartitionFormat |}'
   AND dt <= '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prometheusMetricPartitionFormat |}'
   GROUP BY namespace
   ORDER BY pod_usage_cpu_core_seconds DESC
```

USING METERING ON LARGE CLUSTERS

Capacity Planning

RED HAT OPENSHIFT Online = Huge multi-tenant, 5k+ namespace clusters

- We gate sign ups by capacity, aka Kubernetes Quotas
- Insight into capacity is important
  - Inactive Pods: wrote a Pod Descheduler
  - Inactive Users: remove after X days
- Metering helps us decide the default quota
  - Currently 1GB RAM / 2 vCPUs

Capacity Planning

What did we find?

- Top 1000 namespaces use exactly 50% of their quota

Why?

- Default Pod `resources.requests.memory` = 500mb

Conclusion

- Oversubscribe is OK on this cluster
- Most users don’t customize RAM request

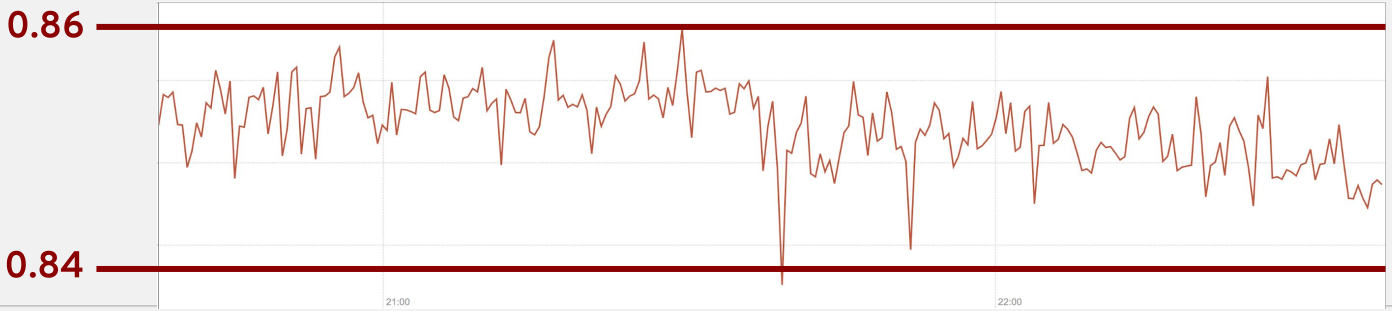
Capacity Planning

What did we find?

- Ratio of usage vs request floats around 85% of limit

Conclusion

- For “real usage”, the default request is well dialed in

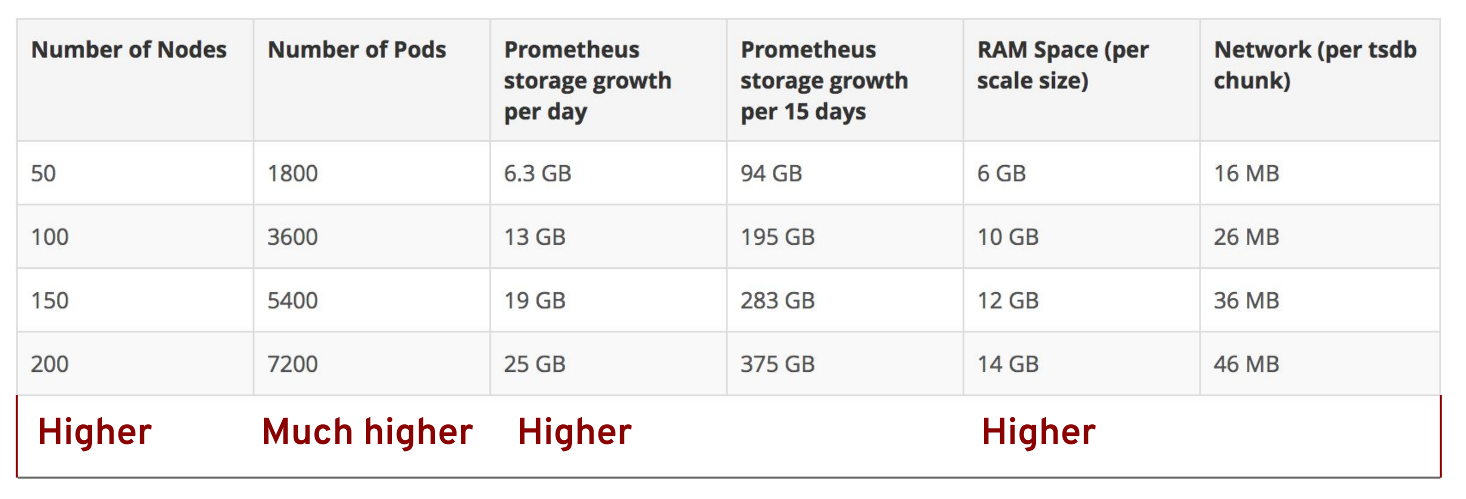


Problems Encountered With 5k Namespaces

- Importing metrics can get backed up
- High metric resolution causes huge growth in storage requirements
  - No down sampling support in Metering yet
- Prometheus cannot always handle larger queries, so we are forced to query smaller amounts
- Reports (namespace or pod reports) can get really long
  - 5k namespaces x 24 hours = 120k rows per hour for hourly report
  - Better to get TopN

Success With 5k Namespaces

- Running Prometheus and Metering at this scale does work



Number of Nodes	Number of Pods	Prometheus storage growth per day	Prometheus storage growth per 15 days	RAM Space (per scale size)	Network (per tsdb chunk)
50	1800	6.3 GB	94 GB	6 GB	16 MB
100	3600	13 GB	195 GB	10 GB	26 MB
150	5400	19 GB	283 GB	12 GB	36 MB
200	7200	25 GB	375 GB	14 GB	46 MB
Higher	**Much higher**	**Higher**		**Higher**	

https://docs.openshift.com/container-platform/3.11/scaling_performance/scaling_cluster_monitoring.html

DEMO

Create a report and view it

```
$ kubectl create -f manifests/reports/cluster-utilization.yaml
scheduledreport.metering.openshift.io/cluster-cpu-utilization-hourly created
$ kubectl proxy
$ baseURL="https://metering.apps.example.com"
$ curl "$baseURL/api/v1/scheduledreports/get?name=cluster-cpu-utilization-daily&format=tab"
```

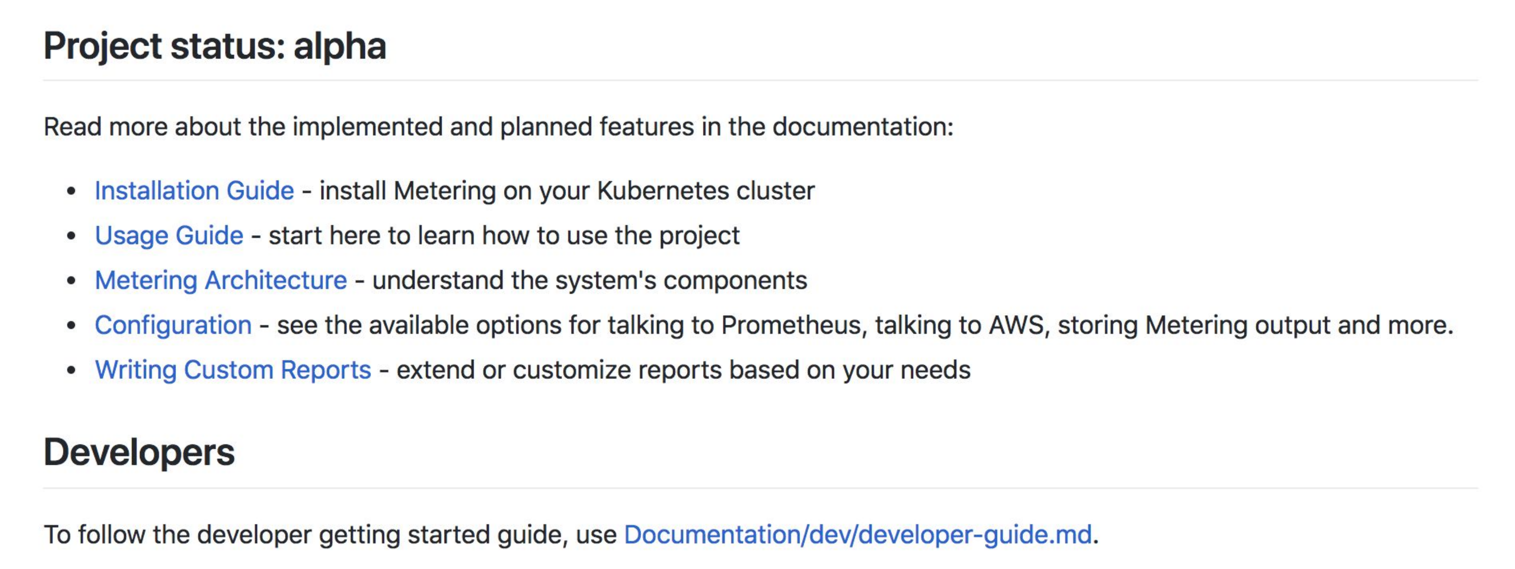
```
period_start                     period_end                       total_cluster_capacity_cpu_core_hours   total_cluster_usage_cpu_core_hours   cluster_cpu_utilization_percent
2018-11-02 00:00:00 +0000 UTC    2018-11-03 00:00:00 +0000 UTC    802.966667                              55.922713                            0.069645
2018-11-03 00:00:00 +0000 UTC    2018-11-04 00:00:00 +0000 UTC    816.000000                              51.339178                            0.062916

…

avg_cluster_capacity_cpu_cores   avg_cluster_usage_cpu_cores   avg_node_count   avg_pod_count   avg_pod_per_node_count
34.000000                        2.370608                      5.000000         91.473380       18.294676
34.000000                        2.139132                      5.000000         92.472917       18.494583
```

TRY IT OUT

https://github.com/operator-framework/operator-metering



Project status: alpha

Read more about the implemented and planned features in the documentation:

- Installation Guide - install Metering on your Kubernetes cluster
- Usage Guide - start here to learn how to use the project
- Metering Architecture - understand the system's components
- Configuration - see the available options for talking to Prometheus, talking to AWS, storing Metering output and more.
- Writing Custom Reports - extend or customize reports based on your needs

Developers

To follow the developer getting started guide, use Documentation/dev/developer-guide.md.

Metering CRDs

- Interaction using kubectl/oc and Openshift Admin Console primarily by creating CRs
- There are 4 primary CRDs users interact with:
  - Reports
  - ReportGenerationQueries
  - ReportDataSources
  - ReportPrometheusQueries
- Only need to use “Reports” if using out-of-the-box queries.

Reports

- Specify the ReportGenerationQuery to run
- How often to compute the report
  - Hourly, daily, weekly
  - Cron schedule
  - Run-once
- Specify custom “inputs” to ReportGenerationQueries to control query behavior
  - Allows reports to aggregate other reports
  - Conditionally add extra fields to the report that aren’t exposed by default
- Results are retrieved using the reporting-operator REST API
  - Supports CSV, JSON, Tabular formats

Report Queries

- ReportGenerationQueries
  - Write your own SQL queries to customize how metrics are aggregated and processed.
  - Supports using Go templates to write more flexible queries
  - Supports user-input from Reports
  - ReportGenerationQueries are used by Reports, and other ReportGenerationQueries
- ReportPrometheusQueries
  - Write your own PromQL queries to gather additional metrics from Prometheus
  - Write a ReportDataSource that uses your ReportPrometheusQuery to tell metering to begin importing metrics

Extensible Reporting queries pt. 2

```yaml
apiVersion: metering.openshift.io/v1alpha1
kind: ReportPrometheusQuery
metadata:
  name: unready-deployment-replicas
spec:
  query: |
    sum(kube_deployment_status_replicas_unavailable) by (namespace, deployment)

---

apiVersion: metering.openshift.io/v1alpha1
kind: ReportDataSource
metadata:
  name: unready-deployment-replicas
spec:
  promsum:
    query: "unready-deployment-replicas"
```

Extensible Reporting queries pt. 3

```
apiVersion: metering.openshift.io/v1alpha1
kind: ReportGenerationQuery
metadata:
  name: "unready-deployment-replicas"
spec:
  reportDataSources:
  - "unready-deployment-replicas"
  columns: [ … ]
  query: |
    SELECT
        labels['namespace'] as namespace,
        labels['deployment'] as deployment,
        sum(amount * "timeprecision") AS total_replica_unready_seconds,
        avg(amount * "timeprecision") AS avg_replica_unready_seconds
    FROM {| dataSourceTableName "unready-deployment-replicas" |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart | prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}'
    GROUP BY labels['namespace'], labels['deployment']
    ORDER BY total_replica_unready_seconds DESC, avg_replica_unready_seconds DESC, namespace ASC, deployment ASC
```