Capacity and Usage Metrics In Kubernetes

Authors:

- Phillip Wittrock (@pwittrock)
- Ebot Ndip-Agbor
- Paul Morie (@pmorie)

TLDR

- We developed a custom solution for generating high fidelity capacity and usage metrics that scales to large clusters
- We think this would be valuable to the community, and are proposing further developing the project under a kubernetes sig instrumentation project

TLDR Example

Get the 95th utilization percentile across all containers in a workload at a 1 second sample interval over the last 5 minutes

```
resources:
  "cpu": "cpu_cores" # get cpu metrics
  "memory": "memory_bytes" # get memory metrics
aggregations:
- sources:
    type: "container"
    container: [ "utilization" ] # export container utilization
  levels:
  - mask:
      name: "container"
      builtIn: # aggregate on these labels
        exported_container: true
        exported_namespace: true
        workload_name: true
        workload_kind: true
        workload_api_group: true
        workload_api_version: true
```

```
operation: "p95" # take the 95th percentile sample
```

Mission

Provide extensible, rich Kubernetes capacity and usage metrics with minimal toil.

Extensible + Rich

- High Granularity Utilization Metrics
    - Configurable sampling interval for utilization metrics
    - Default to ~1 sec
- Comprehensive Metadata
    - Join namespace, workload, node, pod metadata when exporting metrics
- Comprehensive Metrics
    - Export key metrics: un-utilized capacity on nodes, system utilization, etc

Minimal Toil

- Simple
    - Don't require writing complex promql statements to combine metrics – native support for joining metadata across resources
- Scalable
    - Support large clusters with lots of Pods
    - Optimize cardinality and storage requirements by exporting pre-aggregated results
- Performant
    - Quickly compute values without requiring tiering prometheus instances

Motivation

Having the right data is critical to managing capacity in Kubernetes clusters and tuning workloads. While there are many existing solutions for getting Kubernetes metrics, they are architected to solve use cases.

- Utilization metrics sampling intervals should be relatively low, and configurable to ~1 intervals second
- Commonly required metadata should be trivial to attach to the metric values, and should remain performant even with high cardinality joins

Non-Goals

(Don't) replace existing generic metrics solutions such as kube-state-metrics.

The capacity metrics are not intended to be generic “everything about Kubernetes” metrics, but instead focussed on providing a detailed view of capacity and usage within a cluster.

(Don’t) build an APM solution targeted at deep understanding of application specific performance and health

The capacity metrics are not intended to provide deep insight into application performance (e.g. Java, Python, Golang). There are existing solutions for these. The goal is to understand how application’s performance impacts Kubernetes capacity and make it actionable.

Example Use Cases

Get p95 utilization using 1 second sampling intervals for all containers in a workload

- Sample at 1 second interval for each Pod in a workload
- Get p95 value across all samples

Before: Not possible
- Missing granularity
- Difficult to resolve workload from container metrics

Get total utilization for a team’s namespaces broken down by priority class.

- Get pod-level utilization for all pods
- Sum utilization by pod priority class and team owning the namespace

Before: Times out
- Requires expensive and complicated promql joins + relabels + sums
- Times out as a recording rule or query

Get total requests per-node-pool aggregated by priority class.

- Get pod-level utilization for all pods
- Sum utilization by pod priority class and node

Before: Times out
- Requires expensive and complicated promql joins + relabels + sums
- Times out as a recording rule or query

Get p95 utilization using 1 second sampling intervals per-node-pool aggregated by kubepods vs node-system.

- Get cgroup-level utilization for all nodes
- Sum utilization by node-system vs kubepods cgroup

Before: Not possible
- Missing granularity
- Missing scopes

Architecture

Node Sampler

- DaemonSet run on each node and mounts cgroups filesystem
- Scrapes cgroup filesystem periodically and stores samples in a ring buffer
    - By pod UID and container ID
    - By cgroup
- Pushes periodically all samples to the collector
    - Collector resolves pod UI + container ID to pods

Prometheus Collector

- Implements prometheus Collector interface and exposes /metrics endpoint
- Receives utilization samples from the sampler
- Joins namespace, pod, node, quota, workload resource data
- Pre-aggregates metrics to reduce cardinality of data in prometheus

Prometheus + Grafana

- Run standard prometheus + grafana instances to scrape the collector

Capabilities

Operations

- Histograms
- Max
- P95
- Average
- Sum

Sources

- Containers
    - Requests
    - Limits
    - Utilization (1 second sampling rate)
- Pods
    - Utilization (1 second sampling rate)
    - Count
- Quota
    - Requests
    - Limits
    - Used
- Nodes
    - Capacity
    - Allocatable
    - Requests
    - Limits
    - Utilization (1 second sampling rate)
- Cgroups (Nodes)
    - Utilization (1 second sampling rate)
- PVs + PVCs
    -

Container / Pod Labels

- Container Name (container only)
- Pod Name
- Pod Namespaces
- Workload Name + API GVK
- Priority Class
- Node
- App
- Scheduled
- Extensions
    - Namespace Annotations / Labels
    - Pod Annotations / Labels
    - Node Annotations / Labels

Node Labels

- Node Name
- Unschedulable
- Extensions

- Node Annotations / Labels

Example Collector Configurations

mounted in the Collector as a file through a ConfigMap

Get p95 utilization (cpu and memory) using 1 second sampling intervals for all containers in each workload.

```
resources:
  "cpu": "cpu_cores" # get cpu metrics
  "memory": "memory_bytes" # get memory metrics
aggregations:
- sources:
    type: "container"
    container: [ "utilization" ] # export container utilization
  levels:
  - mask:
      name: "container"
      builtIn: # aggregate on these labels
        exported_container: true
        exported_namespace: true
        workload_name: true
        workload_kind: true
        workload_api_group: true
        workload_api_version: true
     operation: "p95" # take the 95th percentile sample
```

Result Metrics:

```
workload_p95_utilization_cpu_cores{exported_container="",exported_nam
espace="",workload_name="",
workload_kind="",workload_api_group="",workload_api_version=""}

workload_p95_utilization_memory_bytes{exported_container="",exported_
namespace="",workload_name="",
workload_kind="",workload_api_group="",workload_api_version=""}
```

Get total utilization (cpu and memory) for each team's namespaces broken down by priority class.

```
extensions:
  namespaceLabels:
```

```
  - name: team # define a team label for metrics
    annotation: team # populate from this namespace annotation
resources:
  "cpu": "cpu_cores"
  "memory": "memory_bytes"
aggregations:
- sources:
    type: "container"
    container: [ "utilization" ]
  levels:
  - mask:
      name: "container"
      builtIn:
        exported_container: true
        exported_pod: true
        exported_namespace: true
        priority_class: true
      extensions:
        team: true # keep the team label
      operation: "average" # take the average for each container
  - mask:
      name: "team"
      builtIn:
        priority_class: true # aggregate on priority_class + team
      extensions:
        team: true # aggregate on priority_class + team
      operation: "sum" # sum containers across namespaces
```

Result Metrics:

```
team_sum_utilization_cpu_cores{team="", priority_class=""}
team_sum_utilization_memory_bytes{team="", priority_class=""}
```

Get total node capacity metrics per-node-pool

```
resources:
  "cpu": "cpu_cores"
  "memory": "memory_bytes"
extensions:
  nodeLabels:
  - name: node_pool # create a metrics label called node_pool
    annotation: node_pool # get value from this node annotation
aggregations:
- sources:
```

```
    type: "node"
    node:
    - "node_capacity" # capacity of the node
    - "node_allocatable" # allocatable of the node
    - "node_requests" # requests scheduled to node
    - "node_limits" # limits scheduled to node
    - "node_allocatable_minus_requests" # remaining schedulable
  levels:
  - mask:
      name: "nodepool"
      extensions:
        node_pool: true
      Operation: sum
```

Result Metrics:

```
nodepool_sum_node_capacity_cpu_cores{node_pool=""}
nodepool_sum_node_capacity_memory_bytes{node_pool=""}

nodepool_sum_node_allocatable_cpu_cores{node_pool=""}
nodepool_sum_node_allocatable_memory_bytes{node_pool=""}

nodepool_sum_node_requests_cpu_cores{node_pool=""}
nodepool_sum_node_requests_memory_bytes{node_pool=""}

nodepool_sum_node_limits_cpu_cores{node_pool=""}
nodepool_sum_node_limits_memory_bytes{node_pool=""}

nodepool_sum_node_allocatable_minus_requests_cpu_cores{node_pool=""}
nodepool_sum_node_allocatable_minus_request_memory_bytes{node_pool=""
}
```

Get a p95 utilization using 1 second sampling intervals per-node-pool aggregated by kubepods vs node-system.

```
resources:
  "cpu": "cpu_cores"
  "memory": "memory_bytes"
extensions:
  nodeLabels:
  - name: node_pool # create a metrics label called node_pool
    annotation: node_pool # get value from this node annotation
cgroupMetrics:
  sources:
```

```
    "/": {name: "utilization"} # define source for cgroups at /
  rootSource: {name: "root_utilization"} # define total node source
aggregations:
- sources:
    type: "cgroup"
    cgroup:
    - "utilization"
    - "root_utilization"
  levels:
  - mask:
      name: "nodepool"
      builtIn:
        cgroup: true # aggregate on cgroup + node_pool
      extensions:
        node_pool: true # aggregate on cgroup + node_pool
    operation: "p95" # get 95th percentile of samples
```

Result Metrics:

```
nodepool_p95_utilization_cpu_cores{node_pool="", cgroup=""}
nodepool_p95_utilization_memory_bytes{node_pool="", cgroup=""}

nodepool_p95_root_utilization_cpu_cores{node_pool=""}
nodepool_p95_root_utilization_memory_bytes{node_pool=""}
```