**Prometheus Essential Metrics and sample promql query**

**# Important Kubernetes Promql metrics**

1. **Count of pods per cluster and namespace**

Having a list of how many pods your namespaces have in your cluster can be **useful for detecting an unusually high or low number of pods** on your namespaces.

Query: sum by (namespace) (kube\_pod\_info)

### Number of containers by cluster and namespace without CPU limits

With this query, you’ll get all the pods that have been restarting. This is really important since a high pod restart rate usually means CrashLoopBackOff.

**Query:**

**count by (namespace)(sum by (namespace,pod,container)(kube\_pod\_container\_info{container!=""}) unless sum by (namespace,pod,container)(kube\_pod\_container\_resource\_limits{resource="cpu"}))**

### Pod restarts by namespace

With this query, you’ll get all the pods that have been restarting. This is really important since a high pod restart rate usually means CrashLoopBackOff.

**Query**

**sum by (namespace)(changes(kube\_pod\_status\_ready{condition="true"}[5m]))**

### Pods not ready

**Query: sum by (namespace) (kube\_pod\_status\_ready{condition="false"})**

### CPU overcommit

### CPU limits over the capacity of the cluster is a scenario you need to avoid. Otherwise, you’ll end up with CPU throttling issues. You can detect CPU overcommit with the following query.

### Query:

### sum(kube\_pod\_container\_resource\_limits{resource="cpu"}) - sum(kube\_node\_status\_capacity\_cpu\_cores)

### Memory overcommit

### Memory limits over the capacity of the cluster could end up in PodEviction if a node is running out of memory. Be aware of this situation with this PromQL query.

### Query:

### sum(kube\_pod\_container\_resource\_limits{resource="memory"}) - sum(kube\_node\_status\_capacity\_memory\_bytes)

### Number of ready nodes per cluster

List the number of nodes available in each cluster.

**Query:**

**sum(kube\_node\_status\_condition{condition="Ready", status="true"}==1)**

### CPU idle by cluster

### How many CPU cores are underutilized

### Query:

### sum((rate(container\_cpu\_usage\_seconds\_total{container!="POD",container!=""}[30m]) - on (namespace,pod,container) group\_left avg by (namespace,pod,container)(kube\_pod\_container\_resource\_requests{resource="cpu"})) \* -1 >0)

1. **Memory Idle by cluster**

**Query:**

sum((container\_memory\_usage\_bytes{container!="POD",container!=""} - on (namespace,pod,container) avg by (namespace,pod,container)(kube\_pod\_container\_resource\_requests{resource="memory"})) \* -1 >0 ) / (1024\*1024\*1024).

Prometheus Alert Manager Examples

Alertmanager configuration for control plane wellness

API server is the heart of any Kubernetes cluster and it is very important to monitor the performance and take necessary actions if things don’t go as planned. Below are some of the example snippets for capturing the performance of API server using Alertmanager.

1. *# Kubernetes API server errors*
2. - alert: ApiServerErrors
3. expr: sum(rate(apiserver\_request\_total{job="apiserver",code=~"(?:5..)"}[1m])) by (instance, job) / sum(rate(apiserver\_request\_total{job="apiserver"}[1m])) by (instance, job) \* 100 > 3
4. for: 2m
5. labels:
6. severity: critical
7. annotations:
8. summary:  API server errors (instance {{ $labels.instance }})
9. description: "API server is experiencing high error rate\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
10. *# Kubernetes API client errors*
11. - alert: ApiClientErrors
12. expr: (sum(rate(rest\_client\_requests\_total{code=~"(4|5).."}[1m])) by (instance, job) / sum(rate(rest\_client\_requests\_total[1m])) by (instance, job)) \* 100 > 1
13. for: 2m
14. labels:
15. severity: critical
16. annotations:
17. summary: API client errors (instance {{ $labels.instance }})
18. description: "API client is experiencing high error rate\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
19. *# Kubernetes client certificate expiry*
20. - alert: KubernetesClientCertificateExpiresSoon
21. expr: apiserver\_client\_certificate\_expiration\_seconds\_count{job="apiserver"} > 0 and histogram\_quantile(0.01, sum by (job, le) (rate(apiserver\_client\_certificate\_expiration\_seconds\_bucket{job="apiserver"}[5m]))) < 24\*60\*60
22. for: 0m
23. labels:
24. severity: critical
25. annotations:
26. summary: Kubernetes client certificate expires soon (instance {{ $labels.instance }})
27. description: "A client certificate used to authenticate to the apiserver is expiring in less than 24.0 hours.\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
28. *# Kubernetes API server latency*
29. - alert: ApiServerLatency
30. expr: histogram\_quantile(0.99, sum(rate(apiserver\_request\_duration\_seconds\_bucket{verb!~"(?:CONNECT|WATCHLIST|WATCH|PROXY)"} [10m])) WITHOUT (subresource)) > 1
31. for: 2m
32. labels:
33. severity: warning
34. annotations:
35. summary:  API server latency (instance {{ $labels.instance }})
36. description: "API server has a 99th percentile latency of {{ $value }} seconds for {{ $labels.verb }} {{ $labels.resource }}.\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"

Alertmanager configuration for data plane wellness

It is equally important to monitor the nodes / instances where our application runs. Below is a sample snippet that monitor the health of nodes focusing on readiness, memory, disk utilization etc.

1. *# Data plane wellness*
2. *# Node not ready*
3. - alert: NodeNotReady
4. expr: kube\_node\_status\_condition{condition="Ready",status="true"} == 0
5. for: 10m
6. labels:
7. severity: critical
8. annotations:
9. summary: Node not ready (instance {{ $labels.instance }})
10. description: "Node {{ $labels.node }} has been unready for a long time\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
11. *# Memory utilization and alert configuration for worker nodes*
12. - alert: NodeMemoryPressure
13. expr: kube\_node\_status\_condition{condition="MemoryPressure",status="true"} == 1
14. for: 2m
15. labels:
16. severity: critical
17. annotations:
18. summary: Node memory pressure (instance {{ $labels.instance }})
19. description: "Node {{ $labels.node }} has MemoryPressure condition\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
20. *# Disk pressure alert config*
21. - alert: NodeDiskPressure
22. expr: kube\_node\_status\_condition{condition="DiskPressure",status="true"} == 1
23. for: 2m
24. labels:
25. severity: critical
26. annotations:
27. summary: Node disk pressure (instance {{ $labels.instance }})
28. description: "Node {{ $labels.node }} has DiskPressure condition\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
29. *# Monitoring network utilization*
30. - alert: NodeNetworkUnavailable
31. expr: kube\_node\_status\_condition{condition="NetworkUnavailable",status="true"} == 1
32. for: 2m
33. labels:
34. severity: critical
35. annotations:
36. summary: Node network unavailable (instance {{ $labels.instance }})
37. description: "Node {{ $labels.node }} has NetworkUnavailable condition\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
38. *# Capacity issues alert config*
39. - alert: NodeOutOfPodCapacity
40. expr: sum by (node) ((kube\_pod\_status\_phase{phase="Running"} == 1) + on(uid) group\_left(node) (0 \* kube\_pod\_info{pod\_template\_hash=""})) / sum by (node) (kube\_node\_status\_allocatable{resource="pods"}) \* 100 > 90
41. for: 2m
42. labels:
43. severity: warning
44. annotations:
45. summary: Node out of pod capacity (instance {{ $labels.instance }})
46. description: "Node {{ $labels.node }} is out of pod capacity\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"

Alertmanager configuration for workload wellness

In the below alertmanager examples, we will focus on the actual pod/ container’s wellness and some of the metrics you have to monitor and alert upon for the wellness of your Kubernetes cluster:

1. *# Workload wellness*
2. *# Container OOM alert*
3. - alert: ContainerOomKiller
4. expr: (kube\_pod\_container\_status\_restarts\_total - kube\_pod\_container\_status\_restarts\_total offset 10m >= 1) and ignoring (reason) min\_over\_time(kube\_pod\_container\_status\_last\_terminated\_reason{reason="OOMKilled"}[10m]) == 1
5. for: 0m
6. labels:
7. severity: warning
8. annotations:
9. summary: Container oom killer (instance {{ $labels.instance }})
10. description: "Container {{ $labels.container }} in pod {{ $labels.namespace }}/{{ $labels.pod }} has been OOMKilled {{ $value }} times in the last 10 minutes.\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
11. *# PersistentVolumeClaim in pending state*
12. - alert: PersistentvolumeclaimPending
13. expr: kube\_persistentvolumeclaim\_status\_phase{phase="Pending"} == 1
14. for: 2m
15. labels:
16. severity: warning
17. annotations:
18. summary: PersistentVolumeClaim pending (instance {{ $labels.instance }})
19. description: "PersistentVolumeClaim {{ $labels.namespace }}/{{ $labels.persistentvolumeclaim }} is pending\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
20. *# PV out of space*
21. - alert: VolumeOutOfDiskSpace
22. expr: kubelet\_volume\_stats\_available\_bytes / kubelet\_volume\_stats\_capacity\_bytes \* 100 < 10
23. for: 2m
24. labels:
25. severity: warning
26. annotations:
27. summary: Volume out of disk space (instance {{ $labels.instance }})
28. description: "Volume is almost full (< 10% left)\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
29. *# HPA unable to scale*
30. - alert: HpaScaleInability
31. expr: (kube\_horizontalpodautoscaler\_spec\_max\_replicas - kube\_horizontalpodautoscaler\_status\_desired\_replicas) \* on (horizontalpodautoscaler,namespace) (kube\_horizontalpodautoscaler\_status\_condition{condition="ScalingLimited", status="true"} == 1) == 0
32. for: 2m
33. labels:
34. severity: warning
35. annotations:
36. summary: HPA scale inability (instance {{ $labels.instance }})
37. description: "HPA {{ $labels.namespace }}/{{ $labels.horizontalpodautoscaler }} is unable to scale\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
38. *# HPA has reached the max replica count for a deployment*
39. - alert: HpaScaleMaximum
40. expr: (kube\_horizontalpodautoscaler\_status\_desired\_replicas >= kube\_horizontalpodautoscaler\_spec\_max\_replicas) and (kube\_horizontalpodautoscaler\_spec\_max\_replicas > 1) and (kube\_horizontalpodautoscaler\_spec\_min\_replicas != kube\_horizontalpodautoscaler\_spec\_max\_replicas)
41. for: 2m
42. labels:
43. severity: info
44. annotations:
45. summary: HPA scale maximum (instance {{ $labels.instance }})
46. description: "HPA {{ $labels.namespace }}/{{ $labels.horizontalpodautoscaler }} has hit maximum number of desired pods\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
47. *# Pod unhealthy*
48. - alert: PodNotHealthy
49. expr: sum by (namespace, pod) (kube\_pod\_status\_phase{phase=~"Pending|Unknown|Failed"}) > 0
50. for: 15m
51. labels:
52. severity: critical
53. annotations:
54. summary: Pod not healthy (instance {{ $labels.instance }})
55. description: "Pod {{ $labels.namespace }}/{{ $labels.pod }} has been in a non-running state for longer than 15 minutes.\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
56. *# Pod crash looping*
57. - alert: PodCrashLooping
58. expr: increase(kube\_pod\_container\_status\_restarts\_total[1m]) > 3
59. for: 2m
60. labels:
61. severity: warning
62. annotations:
63. summary: pod crash looping (instance {{ $labels.instance }})
64. description: "Pod {{ $labels.namespace }}/{{ $labels.pod }} is crash looping\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"
65. *# Deployment /daemonset rollout in stuck state*
66. - alert: DaemonsetRolloutStuck
67. expr: kube\_daemonset\_status\_number\_ready / kube\_daemonset\_status\_desired\_number\_scheduled \* 100 < 100 or kube\_daemonset\_status\_desired\_number\_scheduled - kube\_daemonset\_status\_current\_number\_scheduled > 0
68. for: 10m
69. labels:
70. severity: warning
71. annotations:
72. summary: DaemonSet rollout stuck (instance {{ $labels.instance }})
73. description: "Some Pods of DaemonSet {{ $labels.namespace }}/{{ $labels.daemonset }} are not scheduled or not ready\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}"

# EKS Observability : Essential Metrics

# Current Landscape

|Layer |Source |Tool |Installation and More info |Helm Chart |

|--- |--- |--- |--- |--- |

|Control Plane |\*api server endpoint\*/metrics |N/A - api server exposes metrics in prometheus format directly |https://docs.aws.amazon.com/eks/latest/userguide/prometheus.html |N/A|

|Cluster State |\*kube-state-metrics-http-endpoint\*:8080/metrics |kube-state-metrics |https://github.com/kubernetes/kube-state-metrics#overview |https://github.com/kubernetes/kube-state-metrics#helm-chart |

|Kube Proxy |\*kube-proxy-http\*:10249/metrics |N/A - kube proxy exposes metrics in prometheus format directly |https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ |N/A |

|VPC CNI |\*vpc-cni-metrics-helper\*/metrics |cni-metrics-helper |https://github.com/aws/amazon-vpc-cni-k8s/blob/master/cmd/cni-metrics-helper/README.md |https://github.com/aws/amazon-vpc-cni-k8s/tree/master/charts/cni-metrics-helper |

|Core DNS |\*core-dns\*:9153/metrics |N/A - core DNS exposes metrics in prometheus format directly |https://github.com/coredns/coredns/tree/master/plugin/metrics |N/A |

|Node |\*prom-node-exporter-http\*:9100/metrics |prom-node-exporter |https://github.com/prometheus/node\_exporter

https://prometheus.io/docs/guides/node-exporter/#node-exporter-metrics |https://github.com/prometheus-community/helm-charts/tree/main/charts/prometheus-node-exporter |

|Kubelet/Pod |\*kubelet\*/metrics/cadvisor |kubelet or proxied through api server |https://kubernetes.io/docs/concepts/cluster-administration/system-metrics/ |N/A |

### Agent : AWS Distro for OpenTelemetry

AWS recommends installation, configuration and operations of ADOT on your EKS cluster through the AWS EKS ADOT managed addon. This addon utilized the ADOT operator/collector custom resource model allowing you to deploy, configure and manage multiple ADOT collectors on your cluster. For detailed information on installation, advanced configuration and operations of this addon check out this [documentation](https://aws-otel.github.io/docs/getting-started/adot-eks-add-on).

Note: The AWS EKS ADOT managed addon web console can be used for [advanced configuration of the ADOT addon](https://docs.aws.amazon.com/eks/latest/userguide/deploy-collector-advanced-configuration.html).

There are two components to the ADOT collector configuration.

1. The [collector configuration](https://github.com/aws-observability/aws-otel-community/blob/master/sample-configs/operator/collector-config-amp.yaml) which includes collector deployment mode (deployment, daemonset, etc).

2. The [OpenTelemetry Pipeline configuration](https://opentelemetry.io/docs/collector/configuration/) which includes what receivers, processors, and exporters are needed for metrics collection. Example configuration snippet:

```

config: |

extensions:

sigv4auth:

region: <YOUR\_AWS\_REGION>

service: "aps"

receivers:

#

# Scrape configuration for the Prometheus Receiver

# This is the same configuration used when Prometheus is installed using the community Helm chart

#

prometheus:

config:

global:

scrape\_interval: 60s

scrape\_timeout: 10s

scrape\_configs:

- job\_name: kubernetes-apiservers

bearer\_token\_file: /var/run/secrets/kubernetes.io/serviceaccount/token

kubernetes\_sd\_configs:

- role: endpoints

relabel\_configs:

- action: keep

regex: default;kubernetes;https

source\_labels:

- \_\_meta\_kubernetes\_namespace

- \_\_meta\_kubernetes\_service\_name

- \_\_meta\_kubernetes\_endpoint\_port\_name

scheme: https

tls\_config:

ca\_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt

insecure\_skip\_verify: true

...

...

exporters:

prometheusremotewrite:

endpoint: <YOUR AMP WRITE ENDPOINT URL>

auth:

authenticator: sigv4auth

logging:

loglevel: warn

extensions:

sigv4auth:

region: <YOUR\_AWS\_REGION>

service: aps

health\_check:

pprof:

endpoint: :1888

zpages:

endpoint: :55679

processors:

batch/metrics:

timeout: 30s

send\_batch\_size: 500

service:

extensions: [pprof, zpages, health\_check, sigv4auth]

pipelines:

metrics:

receivers: [prometheus]

processors: [batch/metrics]

exporters: [logging, prometheusremotewrite]

```

A complete best practices collector configuration, ADOT pipeline configuration and Prometheus scrape configuration can be found here as [a Helm Chart in the Observability Accelerator](https://github.com/aws-observability/terraform-aws-observability-accelerator/blob/main/modules/eks-monitoring/otel-config/templates/opentelemetrycollector.yaml).

### Destination: Amazon Managed Service for Prometheus

The ADOT collector pipeline utilizes Prometheus Remote Write capabilities to export metrics to an AMP instance. Example configuration snippet, note the AMP WRITE ENDPOINT URL

```

exporters:

prometheusremotewrite:

endpoint: <YOUR AMP WRITE ENDPOINT URL>

auth:

authenticator: sigv4auth

logging:

loglevel: warn

```

A complete best practices collector configuration, ADOT pipeline configuration and Prometheus scrape configuration can be found here as [a Helm Chart in the Observability Accelerator](https://github.com/aws-observability/terraform-aws-observability-accelerator/blob/main/modules/eks-monitoring/otel-config/templates/opentelemetrycollector.yaml).

Best practices on AMP configuration and usage is [here](https://aws-observability.github.io/observability-best-practices/recipes/amp/).

# What are the relevant metrics?

Gone are the days where you have little metrics available, nowadays it is the opposite, there are hundreds of metrics available. Being able to determine what are the relevant metrics is important towards building a system with an observability first mindset.

This guide outlines the different grouping of metrics available to you and explains which ones you should focus on as you build observability into your infrastructure and applications. The list of metrics below are the list of metrics we recommend monitoring based on best practices.

The metrics listed in the following sections are in addition to the metrics highlighted in the [AWS Observability Accelerator Grafana Dashboards](https://github.com/aws-observability/terraform-aws-observability-accelerator/tree/main/modules/eks-monitoring) and [Kube Prometheus Stack Dashboards](https://monitoring.mixins.dev/).

## Control Plane Metrics

The Amazon EKS control plane is managed by AWS for you and runs in an account managed by AWS. It consists of control plane nodes that run the Kubernetes components, such as etcd and the Kubernetes API server. Kubernetes publishes various events to keep users informed of activities in the cluster, such as spinning up and tearing down pods, deployments, namespaces, and more. The Amazon EKS control plane is a critical component that you need to track to make sure the core components are able function properly and perform the fundamental activities required by your cluster.

The Control Plane API Server exposes thousands of metrics, the table below lists the essential control plane metrics that we recommend monitoring.

|Name |Metric |Description |Reason |

|--- |--- |--- |--- |

|API Server total requests |apiserver\_request\_total |Counter of apiserver requests broken out for each verb, dry run value, group, version, resource, scope, component, and HTTP response code. | |

|API Server latency |apiserver\_request\_duration\_seconds |Response latency distribution in seconds for each verb, dry run value, group, version, resource, subresource, scope, and component|

|Request latency |rest\_client\_request\_duration\_seconds |Request latency in seconds. Broken down by verb and URL. | |

|Total requests |rest\_client\_requests\_total |Number of HTTP requests, partitioned by status code, method, and host. | |

|API Server request duration |apiserver\_request\_duration\_seconds\_bucket |Measures the latency for each request to the Kubernetes API server in seconds | |

|API server request latency sum |apiserver\_request\_latencies\_sum |Cumulative Counter which tracks total time taken by the K8 API server to process requests | |

|API server registered watchers |apiserver\_registered\_watchers |The number of currently registered watchers for a given resource | |

|API server number of objects |apiserver\_storage\_object |Number of stored objects at the time of last check split by kind. | |

|Admission controller latency |apiserver\_admission\_controller\_admission\_duration\_seconds |Admission controller latency histogram in seconds, identified by name and broken out for each operation and API resource and type (validate or admit). | |

|Etcd latency |etcd\_request\_duration\_seconds |Etcd request latency in seconds for each operation and object type. | |

|Etcd DB size |apiserver\_storage\_db\_total\_size\_in\_bytes |Etcd database size. |This helps you proactively monitor etcd database usage, and avoid overrunning the limit. |

## Cluster State metrics

The Cluster State Metrics are generated by `kube-state-metrics` (KSM). KSM is a utility that runs as a pod in the cluster, listening to the Kubernetes API Server, providing you insights into your cluster state and Kubernetes objects in your cluster as Prometheus metrics. KSM will need to be [installed](https://github.com/kubernetes/kube-state-metrics) before these metrics are available. These metrics are used by Kubernetes to effectively do pod scheduling, and is focused on the health of various objects inside, such as deployments, replica sets, nodes and pods. Cluster state metrics expose pod information on status, capacity and availability. Its essential to keep track on how your cluster is performing on scheduling tasks for your cluster so you can keep track performance, get ahead of issues and monitor the health of your cluster. There are about X number of exposed Cluster State Metrics, the table below lists the essential metrics that should be tracked.

|Name |Metric |Description |

|--- |--- |--- |

|Node status |kube\_node\_status\_condition |Current health status of the node. Returns a set of node conditions and `true`, `false`, or `unknown` for each |

|Desired pods |kube\_deployment\_spec\_replicas or kube\_daemonset\_status\_desired\_number\_scheduled |Number of pods specified for a Deployment or DaemonSet |

|Current pods |kube\_deployment\_status\_replicas or kube\_daemonset\_status\_current\_number\_scheduled |Number of pods currently running in a Deployment or DaemonSet |

|Pod capacity |kube\_node\_status\_capacity\_pods |Maximum pods allowed on the node |

|Available pods |kube\_deployment\_status\_replicas\_available or kube\_daemonset\_status\_number\_available |Number of pods currently available for a Deployment or DaemonSet |

|Unavailable pods |kube\_deployment\_status\_replicas\_unavailable or kube\_daemonset\_status\_number\_unavailable |Number of pods currently not available for a Deployment or DaemonSet |

|Pod readiness |kube\_pod\_status\_ready |If a pod is ready to serve client requests |

|Pod status |kube\_pod\_status\_phase |Current status of the pod; value would be pending/running/succeeded/failed/unknown |

|Pod waiting reason |kube\_pod\_container\_status\_waiting\_reason |Reason a container is in a waiting state |

|Pod termination status |kube\_pod\_container\_status\_terminated |Whether the container is currently in a terminated state or not |

|Pods pending scheduling |pending\_pods |Number of pods awaiting node assignment |

|Pod scheduling attempts |pod\_scheduling\_attempts |Number of attempts made to schedule pods |

## Cluster Add-on Metrics

Cluster add-on is software that provides supporting operational capabilities to Kubernetes applications. This includes software like observability agents or Kubernetes drives that allow the cluster to interact with underlying AWS resources for networking, compute and storage. Add-on software is typically built and maintained by the Kubernetes community, cloud providers like AWS, or third-party vendors. Amazon EKS automatically installs self-managed add-ons such as the Amazon VPC CNI plugin for Kubernetes, `kube-proxy`, and CoreDNS for every cluster.

These Cluster add-ons provide operational support in different areas like networking, domain name resolution, etc. They provide you with insights on how the critical supporting infrastructure and components are operating. Tracking add-on metrics are important to understand your clusters operational health.

Below are the essential add-ons that you should consider monitoring along with their essential metrics.

## Amazon VPC CNI Plugin

Amazon EKS implements cluster networking through the Amazon VPC Container Network Interface (VPC CNI) plugin. The CNI plugin allows Kubernetes Pods to have the same IP address ad they do on the VPC network. More specifically, all containers inside the Pod share a network namespace, and they can communicate with each-other using local ports. The VPC CNI add-on enables you to continuously ensure the security and stability of your Amazon EKS clusters and decrease the amount of effort required to install, configure and update add-ons.

VPC CNI add-on metrics are exposed by the CNI Metrics Helper. Monitoring the IP address allocation is fundamental to ensuring a healthy cluster and avoiding IP exhaustion issues. [Here is the latest networking best practices and VPC CNI metrics to collect and monitor](https://aws.github.io/aws-eks-best-practices/networking/vpc-cni/#monitor-ip-address-inventory).

## CoreDNS Metrics

CoreDNS is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS. The CoreDNS pods provide name resolution for all pods in the cluster. Running DNS intensive workloads can sometimes experience intermittent CoreDNS failures due to DNS throttling, and this can impact applications.

Checkout the latest best practices for tracking key [CoreDNS performance metrics here](https://aws.github.io/aws-eks-best-practices/reliability/docs/dataplane/#monitor-coredns-metrics) and [Monitoring CoreDNS traffic for DNS throttling issues](https://aws.github.io/aws-eks-best-practices/networking/monitoring/)

## Pod/Container Metrics

Tracking usage in across all layers of you application is important, these includes taking a closer look at your nodes and pods running inside your cluster. Out of all the metrics available at the pod dimension, this list of metrics are of practical use for you to understand the state of the workloads running on your cluster. Tracking CPU, memory and network usage allows for diagnosing and troubleshooting application related issues. Tracking your workload metrics provide you insights into your resource utilization to right size your workloads running on EKS.

|Metric |Example PromQL Query |Dimension |

|--- |--- |--- |

|Number of running pods per namspace |count by(namespace) (kube\_pod\_info) |Per Cluster by Namespace |

|CPU usage per container per pod |sum(rate(container\_cpu\_usage\_seconds\_total{container!=""}[5m])) by (namespace, pod) |Per Cluster by Namespace by Pod |

|Memory utilization per pod |sum(container\_memory\_usage\_bytes{container!=""}) by (namespace, pod) |Per Cluster by Namespace by Pod |

|Network Received Bytes per pod |sum by(pod) (rate(container\_network\_receive\_bytes\_total[5m])) |Per Cluster by Namespace by Pod |

|Network Transmitted Bytes per pod |sum by(pod) (rate(container\_network\_transmit\_bytes\_total[5m])) |Per Cluster by Namespace by Pod |

|The number of container restarts per container |increase(kube\_pod\_container\_status\_restarts\_total[15m]) > 3 |Per Cluster by Namespace by Pod |

## Node Metrics

Kube State Metrics and Prometheus node exporter gathers metric statistics on the nodes in your cluster. Tracking your nodes status, cpu usage, memory, filesystem and traffic is important to understand your node utilization. Understanding how your nodes resources are being utilized is important for properly selecting instance types and storage to effectively the types of workloads you expect to run on your cluster. The metrics below are some of the essential metrics that you should be tracking.

|Metric |Example PromQL Query |Dimension |

|--- |--- |--- |

|Node CPU Utilization |sum(rate(container\_cpu\_usage\_seconds\_total{container!=""}[5m])) by (node) |Per Cluster by Node |

|Node Memory Utilization |sum(container\_memory\_usage\_bytes{container!=""}) by (node) |Per Cluster by Node |

|Node Network Total Bytes |sum by (instance) (rate(node\_network\_receive\_bytes\_total[3m]))+sum by (instance) (rate(node\_network\_transmit\_bytes\_total[3m])) |Per Cluster by Node |

|Node CPU Reserved Capacity |sum(kube\_node\_status\_capacity{cluster!=""}) by (node) |Per Cluster by Node |

|Number of Running Pods per Node |sum(kubelet\_running\_pods) by (instance) |Per Cluster by Node |

|Node Filesystem Usage |rate(container\_fs\_reads\_bytes\_total{job="kubelet", device=~"mmcblk.p.+|.\*nvme.+|rbd.+|sd.+|vd.+|xvd.+|dm-.+|dasd.+", container!="", cluster="", namespace!=""}[$\_\_rate\_interval]) + rate(container\_fs\_writes\_bytes\_total{job="kubelet", device=~"mmcblk.p.+|.\*nvme.+|rbd.+|sd.+|vd.+|xvd.+|dm-.+|dasd.+", container!="", cluster="", namespace!=""} |Per Cluster by Node |

|Cluster CPU Utilization |sum(rate(node\_cpu\_seconds\_total{mode!="idle",mode!="iowait",mode!="steal"}[5m])) |Per Cluster |

|Cluster Memory Utilization |1 - sum(:node\_memory\_MemAvailable\_bytes:sum{cluster=""}) / sum(node\_memory\_MemTotal\_bytes{job="node-exporter",cluster=""}) |Per Cluster |

|Cluster Network Total Bytes |sum(rate(node\_network\_receive\_bytes\_total[3m]))+sum(rate(node\_network\_transmit\_bytes\_total[3m])) |Per Cluster |

|Number of Running Pods |sum(kubelet\_running\_pod\_count{cluster=""}) |Per Cluster |

|Number of Running Containers |sum(kubelet\_running\_container\_count{cluster=""}) |Per Cluster |

|Cluster CPU Limit |sum(kube\_node\_status\_allocatable{resource="cpu"}) |Per Cluster |

|Cluster Memory Limit |sum(kube\_node\_status\_allocatable{resource="memory"}) |Per Cluster |

|Cluster Node Count |count(kube\_node\_info) OR sum(kubelet\_node\_name{cluster=""}) |Per Cluster |

# Additional Resources

## AWS Services

[https://aws-otel.github.io/](https://aws-otel.github.io/)

[https://aws.amazon.com/prometheus](https://aws.amazon.com/prometheus)

[https://aws.amazon.com/cloudwatch/features/](https://aws.amazon.com/cloudwatch/features/)

## Blogs

[https://aws.amazon.com/blogs/containers/](https://aws.amazon.com/blogs/containers/)

[https://aws.amazon.com/blogs/containers/metrics-and-traces-collection-using-amazon-eks-add-ons-for-aws-distro-for-opentelemetry/](https://aws.amazon.com/blogs/containers/metrics-and-traces-collection-using-amazon-eks-add-ons-for-aws-distro-for-opentelemetry/)

[https://aws.amazon.com/blogs/containers/](https://aws.amazon.com/blogs/containers/)

[https://aws.amazon.com/blogs/containers/introducing-amazon-cloudwatch-container-insights-for-amazon-eks-fargate-using-aws-distro-for-opentelemetry/](https://aws.amazon.com/blogs/containers/introducing-amazon-cloudwatch-container-insights-for-amazon-eks-fargate-using-aws-distro-for-opentelemetry/)

## Infrastructure as Code Resources

[https://github.com/aws-observability/terraform-aws-observability-accelerator](https://github.com/aws-observability/terraform-aws-observability-accelerator)

[https://github.com/aws-ia/terraform-aws-eks-blueprints](https://github.com/aws-ia/terraform-aws-eks-blueprints)