Telemetry Service Private Edition Guide

5/20/2022

Table of Contents

Contents

Find links to all the topics in this guide.

Related documentation:

-

Telemetry Service is a service available with the Genesys Multicloud CX private edition offering.

The Telemetry Service is designed to act as an observability gateway to gather telemetry data, metrics, and logs for Genesys Multicloud software that has services running outside the Data Center and out of range of the Cloud Observability framework like Agent Desktop, Genesys Softphone, etc.

Once gathered by the Telemetry Service, those metrics and logs are stored in the same data sources as the standard services running inside the Data Center.

Overview

Learn more about Telemetry Service, its architecture, and how to support high availability and disaster recovery.

- About Telemetry Service
- Architecture
- High availability and disaster recovery

Configure and deploy

Find out how to configure and deploy Telemetry Service.

- Before you begin
- Configure Telemetry Service
- Provision Telemetry Service
- Deploy Telemetry Service
- Upgrade, rollback, or uninstall Telemetry Service

Operations

Learn how to monitor Telemetry Service with metrics and logging.

- Observability in Telemetry Service
- Telemetry Service metrics and alerts

About Telemetry Service

Contents

Learn about Telemetry Service and how it works in Genesys Multicloud CX private edition.

Related documentation:

-

The Telemetry Service is designed to act as an observability gateway to gather telemetry data, metrics, and logs for Genesys Multicloud software that has services running outside the data center and out of range of the Cloud Observability framework like Agent Desktop, Genesys Softphone, etc.

Once gathered by the Telemetry Service, those metrics and logs are stored in the same data sources as the standard services running inside the data center.

The microservice supports the following API:

- An endpoint to allow remote apps (e.g., applications running in customer environment) to push their **traces** for a centralized treatment.
- An endpoint to allow remote apps to push **metrics** for centralized treatment. Metrics are aggregated by Telemetry service, and available as a Prometheus-compliant data format for building dashboards and alerts.
- An endpoint allowing remote apps to push **events** for centralized treatment. Events are aggregated by Telemetry service, and available as a Prometheus-compliant data format for building dashboards and alerts.

Remote client-side applications can be browser-based interfaces like WWE, as well as executables running on customer premises like Genesys Softphone.

This Telemetry service serves two main goals:

- Proactive detection of issues:
  - Telemetry Service can aggregate information coming from client-side in forms of Metric and Events data.
  - It allows monitoring the client-side activity in any monitoring platforms for performance, active functionalities, incidents.
  - Trigger incidents in an Incident Response Platform when values are hitting some thresholds.
- Accelerate troubleshooting during incidents.

Supported Kubernetes platforms

Telemetry Service is supported on the following cloud platforms:

- Google Kubernetes Engine (GKE)

- OpenShift Container Platform (OpenShift)

See the Telemetry Service Release Notes for information about when support was introduced.

Architecture

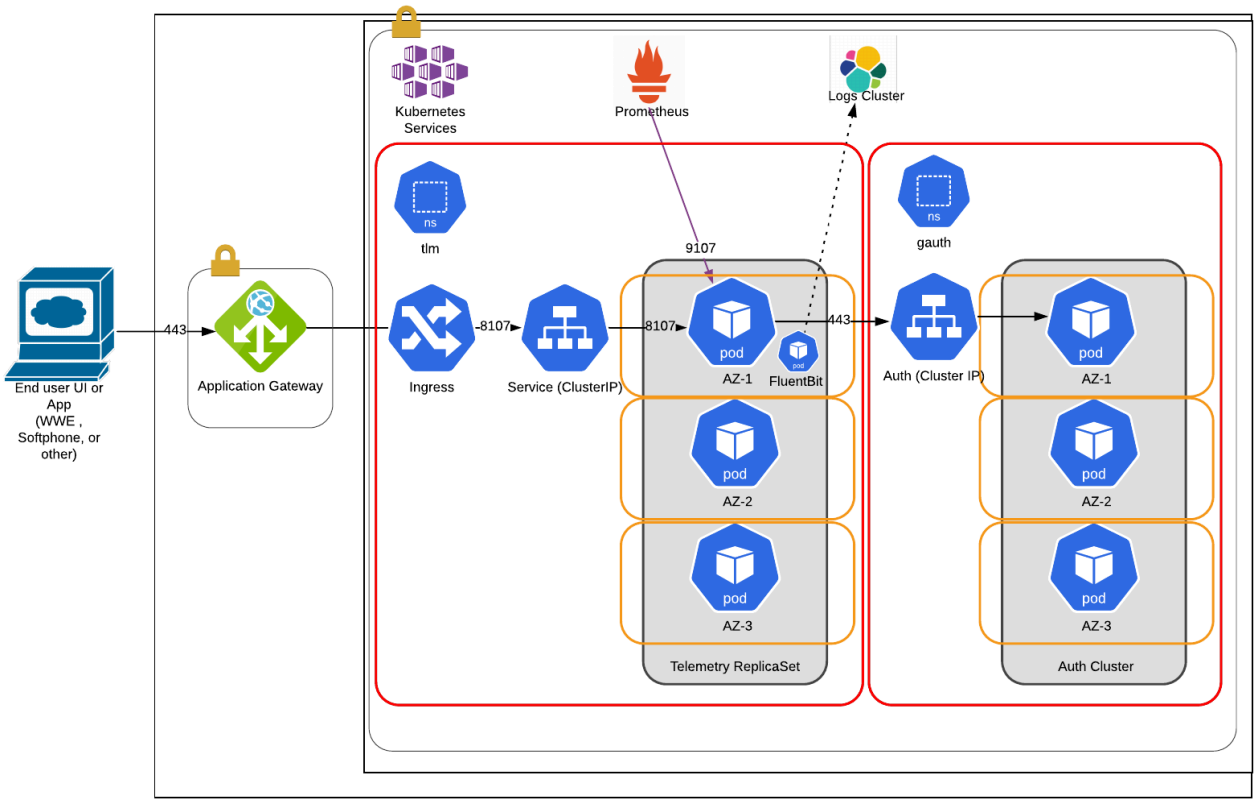
Learn about Telemetry Service's architecture.

Related documentation:

•

For a suite level, architecture, refer to Architecture.

The following diagram, which is intended purely as an example, represents a sample architecture from the Genesys SaaS offering.



High availability and disaster recovery

Find out how this service provides disaster recovery in the event the service goes down.

Related documentation:

-

Service	High Availability	Disaster Recovery	Where can you host this service?
Telemetry Service	N = N (N+1)	Active-spare	Primary or secondary unit

This information is under development: Flagged items aren't yet confirmed or have info coming soon; Checked items are valid.

See High Availability information for all services: High availability and disaster recovery

For High Availability and Load Balancing, Telemetry Service implements an N+1 architecture available behind a Load Balancer.

Telemetry Service is deployed in all Engage Multicloud region/data center so that a client application like WWE can always find a Telemetry Service region/data center where it is relocated.

Ports

Contents

Related documentation:

-

Ports and protocols for Telemetry Service

Included Service	Protocol	Port	Type of data	Comment
Telemetry Service				

Before you begin

Contents

Find out what to do before deploying Telemetry Service.

Related documentation:

-

Download the Helm charts

Telemetry Service is composed of:

- 1 Docker Container: **tlm/telemetry-service:version**
- 1 Helm Chart: **telemetry-service_version.tgz**

For additional information about overriding Helm chart values, see Overriding Helm Chart values in the *Genesys Multicloud CX Private Edition Guide*.

For information about downloading Helm charts from JFrog Edge, see Downloading your Genesys Multicloud CX containers in the *Setting up Genesys Multicloud CX Private Edition* guide.

Genesys dependencies

For any kind of Telemetry deployment, the following service must be deployed and running before deploying the Telemetry service:

- Genesys Authentication Service

For a look at the high-level deployment order, see Order of services deployment.

Configure Telemetry Service

Contents

Learn how to configure Telemetry Service.

Related documentation:

-

Configure a secret to access JFrog

If you haven't done so already, create a secret for accessing the JFrog registry:

```
kubectl create secret docker-registry  --docker-server= --docker-username= --docker-password=
--docker-email=
```

Now map the secret to the default service account:

```
kubectl secrets link default  --for=pull
```

Override Helm chart values

Parameter	Description	Default	Valid values
`TELEMETRY_SERVICES_AUTH_URL`	URL of the GWS Auth public API. This is a mandatory field.		http://gws-core-auth:8095
`TELEMETRY_AUTH_CLIENT_ID`	The Client ID that is used to authenticate with GWS Auth service.	telemetry_client	
`TELEMETRY_CORS_DOMAIN`	Domains to be supported by CORS. This can a comma separated list. **Important** Add a `\` before `.` for regex matching. eg: `\.genesyslab\.com` (another `\` should be added when using quotes).		
`TELEMETRY_TRACES_PROVIDER`	The trace provider to use can be `ElasticSearch` or `Console`	ElasticSearch	
`TELEMETRY_TRACES_CONCURRENT`	The maximum of parallel bulk request to	3	

Parameter	Description	Default	Valid values
	Elasticsearch at the same time.		
`TELEMETRY_TRACES_THRESHOLD`	The maximum buffer limit for Elasticsearch service.	`400000`	
`TELEMETRY_CONFIG_SERVICE`	The data source to fetch configuration information. Possible values : s3, azure, env, or an empty string.	none	
`TELEMETRY_CONFIG_SERVICE_CORS`	This overrides data source to fetch CORS configurations. Possible values : Same value as `TELEMETRY_CONFIG_SERVICE` or `environmentservice` for using the environment-service API (Uses the `TELEMETRY_SERVICES_ENVIRONMENT` variable).	none	
`TELEMETRY_CLOUD_PROVIDER`	Cloud provider for the service. Can be `aws`, `azure`, `gcp` or `premise`	`aws`	
`TELEMETRY_CONFIG_CONTRACTS`	Stringified JSON array to provision contracts through `env` config provider	`[]`	
`TELEMETRY_CONFIG_TENANTS`	A Stringified JSON to provision tenants through `env` config provider.	`{}`	
`TELEMETRY_SERVICES_ENVIRONMENT`	The URL of the GWS environment service API. Used only if environment service is used for configuration provisioning.	value of `TELEMETRY_SERVICES_AUTH`	http://gauth-environment-active.gauth
serviceMonitoringAnnotations.enabled	Activation of Prometheus monitoring annotations on service.	true	
podDisruptionBudget.enabled	Activation of pod disruption.	true	
enableServiceLinks	Enable service links in single namespace environment.	false	

You can modify the configuration to suit your environment by two methods:

- Specify each parameter using the --set key=value[,key=value] argument to helm install. For example,

```
helm install telemetry-service.tgz --set tlm.replicaCount 4
```

- Specify the parameters to be modified in a **values.yaml** file.

```
helm install --name tlm -f values.yaml telemetry-service.tgz
```

Prepare an environment

Create a new project namespace for Telemetry:

```
kubectl create namespace tlm
```

See Creating namespaces for a list of approved namespaces.

Download the telemetry helm charts from the JFrog repository:

```
https://pureengage.jfrog.io/artifactory/helm-staging/tlm
```

Create a **values-telemetry.yaml** file and update the following parameters:

```
TELEMETRY_AUTH_CLIENT_SECRET:
TELEMETRY_AUTH_CLIENT_ID:
TELEMETRY_SERVICES_AUTH: ""
TELEMETRY_CLOUD_PROVIDER: "GKE"
TELEMETRY_CORS_DOMAIN: ""
grafanaDashboard:
  enabled: true
```

Copy the **values-telemetry.yaml** file and the **tlm** Helm package to the installation location.

Provision Telemetry Service

Contents

- Administrator

Feature coming soon! Learn how to provision Telemetry Service.

Related documentation:

-

Each Telemetry Client application has its own way to activate the connection to the Telemetry Service. The Telemetry Service can be configured to enable some advanced functionalities like custom trace contracts, monitored events, monitored metrics, and bucketized metrics.

Currently, the Telemetry service uses a remote storage for its configuration override like AWS S3 or Azure Blob Storage. To avoid the deployment of persistent volume in Private Edition, Environment variables are used for configuration.

```
TELEMETRY_CONFIG_TENANTS="{\"2868802f-1763-4ecd-94f6-203400001200\":\"pulse\"}"
TELEMETRY_CONFIG_CONTRACTS="[{ ... }]"
```

For provisioning, the following updates can be made to the **values.yaml** file:

```
tlm:
    context:
        envs:
            TELEMETRY_CONFIG_SERVICE: "env" # This will tell Telemetry service to use
environment variables provisioning
            TELEMETRY_CONFIG_TENANTS: "{\"2868802f-1763-4ecd-94f6-203400001200\":\"pulse\"}"
            TELEMETRY_CONFIG_CONTRACTS: '[{"appName": "wwe_ui", "properties": [ "connId",
        "agentSessionId", "browserSessionId", "interactionId", "POC_override" ],
"monitoredMetrics":
        [ { "name": "http_sync_req_StartContactCenterSession", "value": 2, "type":
        "gt" }, { "name": "http_sync_req_AttachUserData", "value": 5222000, "type":
        "lt" } ], "monitoredEvents": [ "error_.*", "disaster_recovery_.*",
"performance_worker_major",
        "performance_worker_severe" ]}, {"appName": "softphone", "properties": [ "ThisDN",
        "call_id", "call_uuid", "region", "level", "msgId", "sepsessionid", "value__NUM"
        ], "apdex": { "metrics": { "sip_call_mos": { "satisfiedTH": 3.6, "toleratedTH":
        2, "isExtended": true } } }}]'
            TELEMETRY_CONFIG_CORS: '{ "33cd4384-e0e7-4860-90e7-589712c33301":
{"urls":["http://localhost:8080",
        "http://localhost:3000"], "domains":[]}, "ed79bc34-768e-4d74-a924-cf10107c1807":
        {"urls":["http://localhost:8080", "http://localhost:7000"], "domains":[]}}'
```

Tenant provisioning

The Telemetry service has the information of the Contact Center by ID and not by name. Telemetry service provides the possibility of mapping contact center ID to a name by injecting it as an environment variable. This allows displaying the name of the contact center instead of the contact enter ID in the time series labels.

To add a name to the contact center ID, add the information in the **TELEMETRY_CONFIG_TENANTS**

environment variable.

```
TELEMETRY_CONFIG_TENANTS: '{"my-contact-center-id":"my contact center"}'
```

As the JSON is parsed directly from the environment variable, use the [https://github.com/fastify/secure-json-parse JSON-SECURE-PARSE] library and [https://github.com/ajv-validator/ajv AJV] library for validating the JSON schema.

Configuring Telemetry Contracts

Each application that wants to send data to Telemetry service needs a contract to be declared and provisioned in the service. Depending on the features, the contract values can be configures. An example of a contract with all features activated:

```
[{
  "appName": "nameOfTheApplication",
  "properties": ["property1", "property2"],
  "apdex": {
    "default": {
      "satisfiedTH": 1201,
      "toleratedTH": 4801
    },
    "metrics": {
      "http_async_req_Accept": {
        "satisfiedTH": 1200,
        "toleratedTH": 4800
      },
      "call_Quality_ReversedApdex": {
        "satisfiedTH": 4,
        "toleratedTH": 3,
        "isExtended": true
      }
    }
  },
  "buckets": {
    "foo": [400, 500, 900]
  },
  "monitoredMetrics": [
    { "name": "", "value": , "type": "" }
  ],
  "monitoredEvents": [
    "error_.*",
    "disaster_recovery_.*",
    "business_attribute_option_config_issue"
  ]
}]
```

Property	Description
appName	This is the application name to be used to identify the application. for example, wwe_ui / softphone / hca / nexus_chat
properties	An array of properties to serialize when traces are pushed to Telemetry service, all other properties will be ignored.

Property	Description
apdex	Apdex calculation is done by applying Satisfied and Tolerated Threshold to a metric. This metric can be any unit as long as the threshold that have to be applied are on the same range. Once a metric needs to have an apdex calculation the looking for threshold is applied and it goes like this: 1. There is a lookup in the config file if the `metricname` is in the property **apdex.metrics** of the configuration object. 2. If the first step is not successful, there is a lookup on **apdex.default** values. 3. If none of the previous step is successful, the apdex calculation falls back to default threshold which are : SatisfiedTH: 1200 / toleratedTH: 4800. **Note**: Those default metrics are used to calculate the render speed of UI components in milliseconds.
isExtended	isExtended property is a Boolean value to activate or not the display of the raw numbers which have been used to calculate the apdex. Once it's set to true, 3 more counters will be added: • satisfied count • tolerated count • frustrated count **Note**: Apdex definition is first meant to define the lowest value as the better score. For specific concerns, Telemetry APDEX supports the reversed behavior if you switch values of satisfied and tolerated threshold.
buckets	Buckets are made with the interval declared between the thresholds; from -Infinity to +Infinity. And then those values will be exposed in buckets named with the index of the interval starting at 1 to X. For example: • Bucket definition: [400 , 500] The API receives those metrics : 200,340,350,600. This will end up with: • {metricType="Bucket", BucketId="1"} 3 • {metricType="Bucket", BucketId="2"} 0 • {metricType="Bucket", BucketId="3"} 1
monitoredMetrics	Monitored Events are declared in Telemetry contracts with plain text event or Regular Expression using the property, **monitoredEvents**.

Property	Description

	There is no default activation, definition is done per Telemetry client. Each metric received from Telemetry clients through the API endpoint **telemetry/v1/record** is evaluated by the monitored metric module. If a Monitored Metric has been configured for this metric, the received value is evaluated against the threshold configured for this monitored metric.
	• "**appName**":: the keyword representing the Telemetry client application ('wwe_ui', 'softphone', 'hca', 'nexus_chat'...).
	• "**name**":: the name of the metric as posted by the Telemetry client application at runtime.
	• "**value**":: the numeric value representing the threshold that can trigger the recording of the monitored metric. It must be consistent in unit with the value that the Telemetry client application is posting at runtime.
	• "**type**":"": the type of threshold
	  • 'gt' (default): the values greater than the threshold are recorded as 'monitored metric' records.
	  • 'lt': the values lower than the threshold are recorded as 'monitored metric' records.
	The data is collected and compiled every 10 minutes by default and can be changed with the environment variable, **TELEMETRY_EVENT_MONITOR_TIME**. After each compilation the data is sent to ElasticSearch like other traces in the index **tlm-traces-***. Those can be identified with the property **trace_type** set to **eventMonitor**

Deploy Telemetry Service

Contents

Learn how to deploy Telemetry Service.

Related documentation:

-

> **Important**
> Make sure to review Before you begin for the full list of prerequisites required to deploy Telemetry Service.

Deploy the service

To install the Telemetry Service, run the following command:

```
helm install -f values-tlm.yaml telemetry-service telemetry-service/
```

Validate the deployment

To validate the installed release, run the following command:

```
helm list –n tlm
```

Verify that details of the Telemetry Service deployment information is displayed.

To check the status of installed Helm release, execute the following command:

```
helm status telemetry-service -n tlm
```

Verify that the deployment status mentions 'STATUS: deployed'.

To verify if the objects are created and available in the Telemetry namespace

```
kubectl get all -n tlm
```

Verify that all pods, services, and config maps are displayed.

Expose ports for access

To make the Telemetry service accessible from outside the cluster, you have to create ingress files for external and internal access points and apply them to the containers.

Configuring ports for external access

- Create an ingress file named **tlm-ingress-cert.yaml** and modify it to reflect your domain configurations:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tlm-ingress
  namespace: tlm
  annotations:
    cert-manager.io/cluster-issuer:
    kubernetes.io/ingress.class:
    nginx.ingress.kubernetes.io/ssl-redirect: 'false'
    nginx.ingress.kubernetes.io/use-regex: 'true'
spec:
  tls:
    - hosts:
        - tlm.
      secretName: tlm-secret-ext
  rules:
    - host: tlm.
      http:
        paths:
          - path: /.*
            pathType: ImplementationSpecific
            backend:
              service:
                name: telemetry-service
                port:
                  number: 8107
```

- Apply the access rules:

```
kubectl apply -f tlm-ingress-cert.yaml -n tlm
```

Configuring ports for internal access

- Create an ingress file named **tlm-ingress-int-cert.yaml** and modify it to reflect your domain configurations:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tlm-ingress-int
  namespace: tlm
  annotations:
    cert-manager.io/cluster-issuer: ca-cluster-issuer
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: 'false'
    nginx.ingress.kubernetes.io/use-regex: 'true'
spec:
```

```
  tls:
    - hosts:
        - tlm.
      secretName: tlm-secret-int
  rules:
    - host: tlm.
      http:
        paths:
          - path: /metrics
            pathType: ImplementationSpecific
            backend:
              service:
                name: telemetry-service
                port:
                  number: 9107
```

- Apply the access rules:

```
kubectl apply -f tlm-ingress-int-cert.yaml -n tlm
```

Verify if the routes are created correctly:

```
kubectl get ingress -n tlm
```

Upgrade, rollback, or uninstall Telemetry Service

Contents

Learn how to upgrade, rollback, or uninstall Telemetry Service.

Related documentation:

-

Upgrade Telemetry Service

Telemetry Service supports a Rolling Update strategy to upgrade its services. To upgrade Telemetry services, first override the Helm chart values.

Next, run the following command to upgrade:

```
helm upgrade --version -f values.yaml telemetry-service ./tlm
```

Rollback Telemetry Service

To roll back Telemetry Services, use the `helm rollback` command:

```
helm rollback telemetry-service
```

To roll back to a specific version of Telemetry Services, you can use either the `helm rollback` command or the `helm upgrade` command.

- `helm rollback telemetry-service`
- `helm upgrade --version -f previous-values.yaml telemetry-service ./tlm`

Uninstall Telemetry Service

To uninstall the Telemetry service:

```
helm uninstall telemetry-service -n tlm
```

Observability in Telemetry Service

Contents

Learn about the logs, metrics, and alerts you should monitor for Telemetry Service.

Related documentation:

-

Monitoring

Private edition services expose metrics that can be scraped by Prometheus, to support monitoring operations and alerting.

- As described on Monitoring overview and approach, you can use a tool like Grafana to create dashboards that query the Prometheus metrics to visualize operational status.
- As described on Customizing Alertmanager configuration, you can configure Alertmanager to send notifications to notification providers such as PagerDuty, to notify you when an alert is triggered because a metric has exceeded a defined threshold.

The services expose a number of Genesys-defined and third-party metrics. The metrics that are defined in third-party software used by private edition services are available for you to use as long as the third-party provider still supports them. For descriptions of available Telemetry Service metrics, see:

- Telemetry Service metrics

See also System metrics.

Enable monitoring

Telemetry Service does not expose any specific metrics for monitoring. You can use standard Kubernetes metrics, as delivered by cAdvisor, of the kind that apply to any pod of the same nature.

Service	CRD or annotations?	Port	Endpoint/ Selector	Metrics update interval
Telemetry Service	n/a	n/a	n/a	n/a

Configure metrics

No additional service-level configuration is required to enable monitoring for Telemetry Service.

Alerting

Private edition services define a number of alerts based on Prometheus metrics thresholds.

> **Important**
> While you can use general third-party functionality to create rules to trigger alerts based on metrics values you specify, private edition does not enable you to create custom alerts, and Genesys does not provide support for custom alerting.

For descriptions of available Telemetry Service alerts, see:

- Telemetry Service alerts

Configure alerts

Private edition services define a number of alerts by default (for Telemetry Service, see the pages linked to above). No further configuration is required.

The alerts are defined as **PrometheusRule** objects in a **prometheus-rule.yaml** file in the Helm charts. As described above, Telemetry Service does not support customizing the alerts or defining additional **PrometheusRule** objects to create alerts based on the service-provided metrics.

Logging

Telemetry Service sends logs to **stdout**.

Telemetry Service logs are structured so that log documents can be split into two distinct indexes: one for the Core Telemetry activity and the other for the Telemetry client logs.

Telemetry Service metrics and alerts

Find the metrics Telemetry Service exposes and the alerts defined for Telemetry Service.

Contents

Service	CRD or annotations?	Port	Endpoint/Selector	Metrics update interval
Telemetry Service	n/a	n/a	n/a	n/a
	All the Telemetry Service metrics are standard Kubernetes metrics as delivered by a standard Kubernetes metrics service.			

See details about:

- Telemetry Service metrics
- Telemetry Service alerts

Metrics

Use standard Kubernetes metrics, as delivered by a standard Kubernetes metrics service (such as cAdvisor), to monitor the Telemetry Service. For information about standard system metrics to use to monitor services, see System metrics.

The following standard Kubernetes metrics are likely to be most relevant.

Metric and description	Metric details	Indicator of
container_cpu_usage_seconds_total Cumulative CPU time consumed	**Unit:** seconds **Type:** Counter **Label:** pod="podId" **Sample value:** 7000	Monitoring the CPU usage
container_fs_reads_bytes_total Cumulative count of bytes read	**Unit:** bytes **Type:** Counter **Label:** pod="podId **Sample value:** 900	Monitoring Filesystem usage
container_network_receive_bytes_total Cumulative count of bytes received	**Unit:** bytes **Type:** Counter **Label:** pod="podId" **Sample value:** 3000	Monitoring incoming network
container_network_transmit_bytes_total Cumulative count of bytes transmitted	**Unit:** bytes **Type:** Counter **Label:** pod="podId" **Sample value:** 5000	Monitoring outgoing network
kube_pod_container_status_ready Describes whether the containers readiness check succeeded.	**Unit:** integer **Type:** Gauge **Label:** pod="podId" **Sample value:** 2	Monitoring Healthy pods
kube_pod_container_status_restarts_total	**Unit:** integer	Monitoring pod restarts

Metric and description	Metric details	Indicator of
The number of container restarts per container	**Type:** Counter **Label:** pod="podId" **Sample value:** 0	

Alerts

The following alerts are defined for Telemetry Service.

Alert	Severity	Description	Based on	Threshold
Telemetry CPU Utilization is Greater Than Threshold	High	Triggered when average CPU usage is more than 60%	node_cpu_seconds_total	>60%
Telemetry Memory Usage is Greater Than Threshold	High	Triggered when average memory usage is more than 60%	container_cpu_usage_seconds_total, kube_pod_container_resource_limits_cpu_cores	>60%
Telemetry High Network Traffic	High	Triggered when network traffic exceeds 10MB/ second for 5 minutes	node_network_transmit_bytes_total, node_network_receive_bytes_total	>10MBps
Http Errors Occurrences Exceeded Threshold	High	Triggered when the number of HTTP errors exceeds 500 responses in 5 minutes	telemetry_events{eventName=~"http_error_.*", eventName!="http_error_404"}	>500 in 5 minutes
Telemetry Dependency Status	Low	Triggered when there is no connection to one of the dependent services - GAuth, Config, Prometheus	telemetry_dependency_status	
Telemetry Healthy Pod Count Alert	High	Triggered when the number of healthy pods drops to critical level	kube_pod_container_status_ready	
Telemetry GAuth Time Alert	High	Triggered when there is no connection to the GAuth service	telemetry_gws_auth_request_time	≥1000