NetScaler® Observability Exporter

Contents

Release notes

September 27, 2025

Release notes describe the new features and enhancements introduced in a particular build, and the issues fixed in the build.

Release 1.10.001

Fixed issues

- Fixed the issue of `ServerUrl` option for Kafka not accepting multiple bootstrap brokers. With this fix, the `ServerUrl` option now accepts a comma-separated list of bootstrap Kafka brokers. For example, `X.X.X.X:9092,Y.Y.Y.Y:9092`.

Release 1.9.001

What's new

Support for DEBUG severity level A new severity level, DEBUG, is now supported. The DEBUG severity level provides comprehensive container logging that contains fatal, error, informational, and debug messages.

Numerous logs are now supported. For the complete list of logs, see Log descriptions.

Support to export Auditlogs and Events to Kafka NetScaler® Observability Exporter can now export NetScaler Events and Auditlogs to Kafka in JSON format.

Fixed issues

- Fixed a data loss issue with Splunk export over SSL. As part of the fix, a new field `ConnectionPoolSize` is introduced. `ConnectionPoolSize` and `MaxConnections` can be used to control the rate at which data is exported. For specifics of these fields, see Description of configuration parameters.

NetScaler® Observability Exporter

September 27, 2025

NetScaler Observability Exporter is a container which collects metrics and transactions from NetScalers and transforms them to suitable formats (such as JSON, AVRO) for supported endpoints. You can export the data collected by NetScaler Observability Exporter to the desired endpoint. By analyzing the data exported to the endpoint, you can get valuable insights at a microservices level for applications proxied by NetScalers.

Supported Endpoints

NetScaler Observability Exporter currently supports the following endpoints:

- Zipkin
- Kafka
- Elasticsearch
- Prometheus
- Splunk Enterprise

Overview

Distributed tracing support with Zipkin

In a microservice architecture, a single end-user request may span across multiple microservices and tracking a transaction and fixing sources of errors is challenging. In such cases, traditional ways for performance monitoring cannot accurately pinpoint where failures occur and what is the reason behind poor performance. You need a way to capture data points specific to each microservice which is handling a request and analyze them to get meaningful insights.

Distributed tracing addresses this challenge by providing a way to track a transaction end-to-end and understand how it is being handled across multiple microservices. OpenTracing is a specification and standard set of APIs for designing and implementing distributed tracing. Distributed tracers allow you to visualize the data flow between your microservices and helps to identify the bottlenecks in your microservices architecture.

NetScaler Observability Exporter implements distributed tracing for NetScaler and currently supports Zipkin as the distributed tracer.

Currently, you can monitor performance at the application level using NetScaler. Using NetScaler Observability Exporter with NetScaler, you can get tracing data for microservices of each application proxied by your NetScaler CPX, MPX, or VPX.

Transaction collection and streaming support

NetScaler Observability Exporter supports collecting transactions and streaming them to endpoints. Currently, NetScaler Observability Exporter supports Elasticsearch and Kafka as transaction endpoints.

Time series data support

NetScaler Observability Exporter supports collecting time series data (metrics) from NetScaler instances and exports them to Prometheus. Prometheus is a monitoring solution for storing time series data like metrics. You can then add Prometheus as a data source to Grafana and graphically view the NetScaler metrics and analyze the metrics.

How does NetScaler Observability Exporter work

Distributed tracing with Zipkin using NetScaler Observability Exporter

Logstream is a Citrix-owned protocol that is used as one of the transport modes to efficiently transfer transactions from NetScaler instances. NetScaler Observability Exporter collects tracing data as Logstream records from multiple NetScalers and aggregates them. NetScaler Observability Exporter converts the data into a format understood by the tracer and then uploads to the tracer (Zipkin in this case). For Zipkin, the data is converted into JSON, with Zipkin-specific key values.

You can view the traces using the Zipkin user interface. However, you can also enhance the trace analysis by using Elasticsearch and Kibana with Zipkin. Elasticsearch provides long-term retention of the trace data and Kibana allows you to get much deeper insight into the data.

NetScaler Observability Exporter with Elasticsearch as the transaction endpoint

When Elasticsearch is specified as the transaction endpoint, NetScaler Observability Exporter converts the data to JSON format. On the Elasticsearch server, NetScaler Observability Exporter creates Elasticsearch indexes for each ADC on an hourly basis. These indexes are based on data, hour, UUID of the ADC, and the type of HTTP data (http_event or http_error). Then, NetScaler Observability Exporter uploads the data in JSON format under Elastic search indexes for each ADC. All regular transactions are placed into the http_event index and any anomalies are placed into the http_error index.

NetScaler Observability Exporter with Kafka as the endpoint

NetScaler Observability Exporter exports transactions to Kafka as Avro or JSON. Auditlogs and events are exported as JSON.

NetScaler Observability Exporter with Prometheus as the endpoint for time series data

When Prometheus is specified as the format for time series data, NetScaler Observability Exporter collects various metrics from NetScalers and converts them to appropriate Prometheus format and exports them to the Prometheus server. These metrics include counters of the virtual servers, services to which the analytics profile is bound and global counters of HTTP, TCP and so on.

NetScaler Observability Exporter with Splunk Enterprise as the endpoint

When Splunk Enterprise is specified as the transaction endpoint, NetScaler Observability Exporter collects indexes, audit logs, and events and exports to Splunk Enterprise. Splunk Enterprise captures indexes and correlates real-time data in a repository from which it can generate reports, graphs, dashboards, and visualizations. Splunk Enterprise provides a graphical representation of these data.

Deployment

You can deploy NetScaler Observability Exporter using Kubernetes YAML. To deploy NetScaler Observability Exporter using Kubernetes YAML, see Deployment. To deploy NetScaler Observability Exporter using Helm charts, see Deploy using Helm charts.

Features

Custom header logging

Custom header logging enables logging of all HTTP headers of a transaction and currently supported on the Kafka endpoint.
For more information, see Custom header logging.

Elasticsearch support enhancements

Effective with the NetScaler Observability Exporter release 1.2.001, when the NetScaler Observability Exporter sends the data to the Elasticsearch server some of the fields are available in the string format. Also, index configuration options are also added for Elasticsearch. For more information on fields which are in the string format and how to configure the Elasticsearch index, see Elasticsearch support enhancements.

Deploy NetScaler Observability Exporter

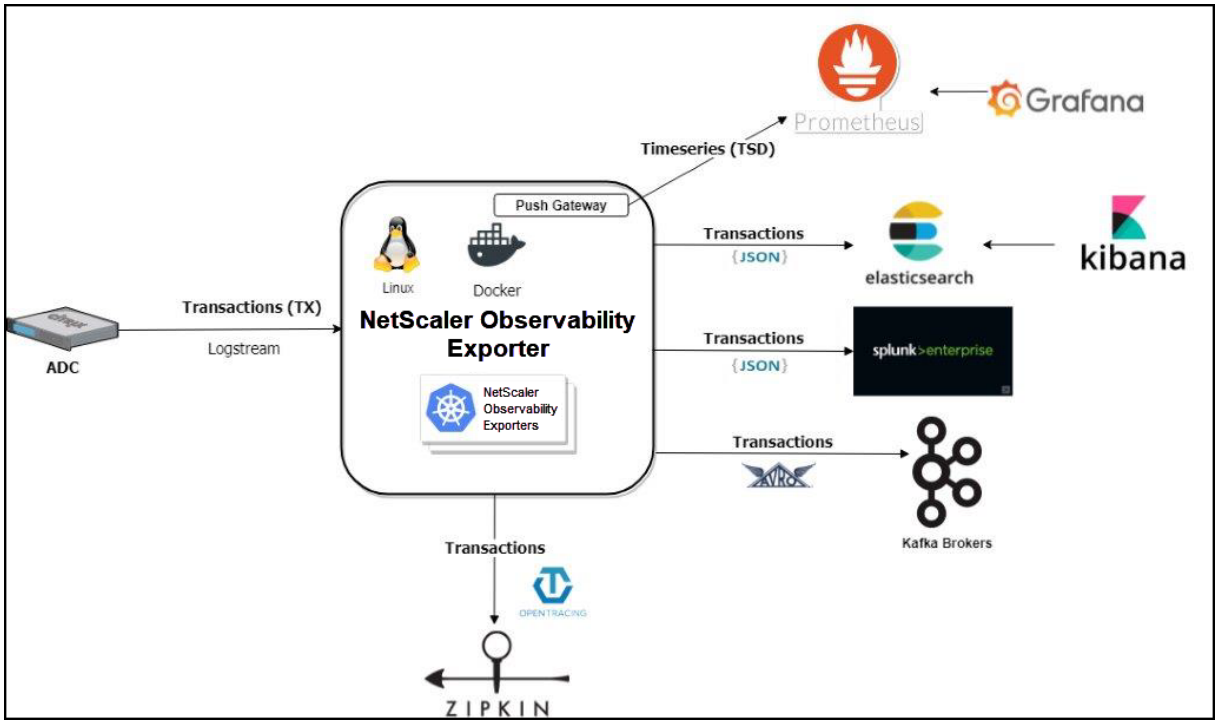
August 11, 2024

This topic provides information on how to deploy NetScaler Observability Exporter using Kubernetes YAML files.

> **Note:**
>
> You can deploy NetScaler Observability Exporter using Kubernetes YAML files or using Helm charts. You can also deploy NetScaler Observability Exporter using NetScaler Operator. For information on the steps to deploy, see Deploy NetScaler Observability Exporter using NetScaler Operator.

Based on your NetScaler deployment, you can use NetScaler Observability Exporter to export metrics and transactions from NetScaler CPX, MPX, or VPX.

The following diagram shows a deployment of NetScaler Observability Exporter with all the supported endpoints.



NetScaler Observability Exporter supports the following endpoints: Kafka, Elasticsearch, Prometheus, and Zipkin. Depending on the endpoint that you require, you can deploy NetScaler Observability Exporter with that endpoint.

You can use one of the following deployment procedures based on the endpoint that you require:

- Deploy NetScaler Observability Exporter with Zipkin
- Deploy NetScaler Observability Exporter with Prometheus
- Deploy NetScaler Observability Exporter with Elasticsearch
- Deploy NetScaler Observability Exporter with Kafka
- Deploy NetScaler Observability Exporter with Splunk

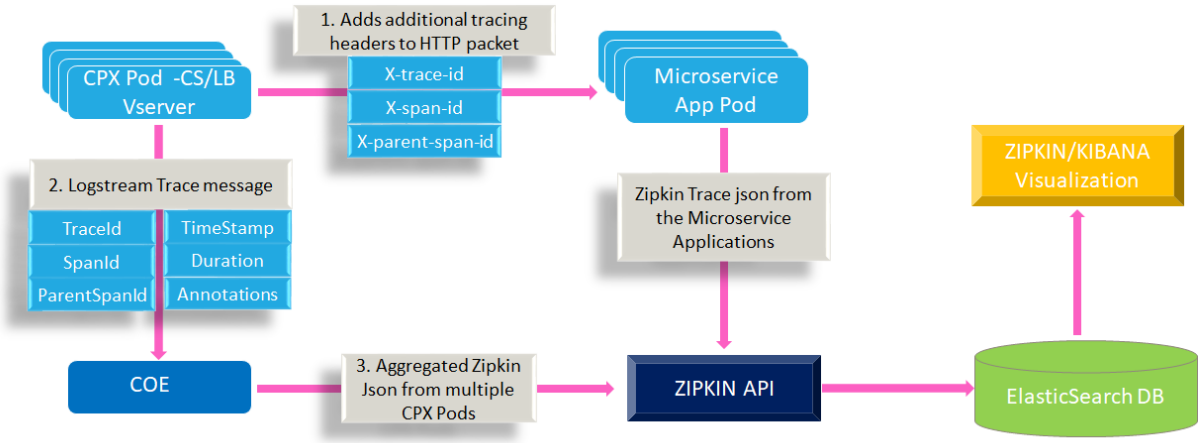
NetScaler Observability Exporter with Zipkin as endpoint

December 31, 2023

NetScaler Observability Exporter supports OpenTracing (OpenTracing is a part of OpenTelemetry now) using Zipkin as the endpoint. NetScaler Observability Exporter transforms the tracing data collected from NetScalers into supported formats suitable for OpenTracing and exports them to Zipkin. Zipkin is a distributed tracing system that helps to gather the timing data required to troubleshoot latency problems in microservice architectures. Elasticsearch is used for long-term retention of trace data and the traces can be visualized using the Zipkin UI or Kibana.

The following diagram illustrates how the Zipkin architecture works:

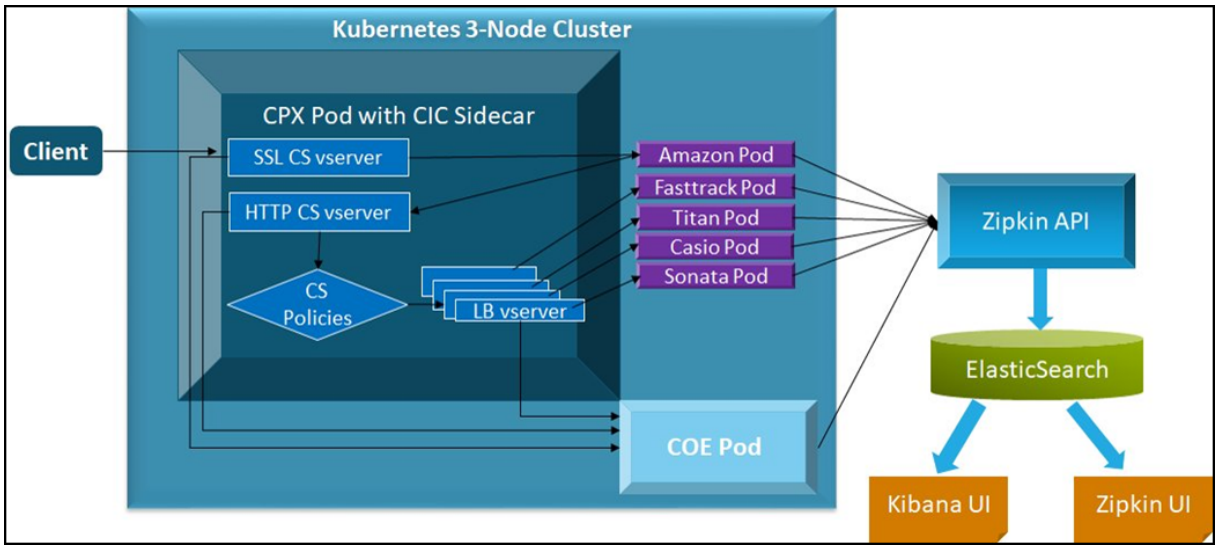
1. When the tracing is enabled, initially, it adds additional open-tracing headers: `x-trace-id`, `x-span-id`, and `x-parent-span-id` to HTTP packet, before it forwards the packet to the next microservice pod.
2. The information about this communication or transaction is pushed to NetScaler Observability Exporter. The information includes the details about the headers, the timestamp (time when this request is initiated and the entire duration of the process), and annotations (annotations include HTTP, SSL, and TCP associated with that request).
3. Then, NetScaler Observability Exporter receives multiple trace messages from all the NetScalers and aggregates them into Zipkin understandable JSON format, and push that to Zipkin through the API.
4. Similarly, if microservices are enabled with tracing, then that trace is sent to Zipkin through the API.
5. Zipkin API stores the trace data in the Elasticsearch database, and finally stitch the complete trace to the given HTTP request and visualize it in the visualization tool such as Kibana. You can view the time that the request spent on each microservices.



Deploy NetScaler Observability Exporter

Based on your NetScaler deployment, you can deploy NetScaler Observability Exporter either outside or inside Kubernetes clusters. You can deploy NetScaler Observability Exporter as a pod inside the Kubernetes cluster or enable the configuration on NetScaler MPX or VPX form factor outside the cluster. You can deploy NetScaler Observability Exporter using the Kubernetes YAML file provided by NetScaler.

The following diagram illustrates NetScaler as an ingress gateway with the NetScaler Ingress Controller as a sidecar. NetScaler Observability Exporter sends the tracing data collected from NetScalers to Zipkin API. The tracing data is, then, uploaded to the Elasticsearch server. From Elasticsearch, the data is sent to Zipkin UI or Kibana UI for visualization.



Prerequisites

- Ensure that you have a Kubernetes cluster with `kube-dns` or `CoreDNS` addon enabled.

To deploy NetScaler Observability Exporter with Zipkin, you must perform the following tasks:

1. Deploy the required application with the tracing support enabled.
2. Deploy NetScaler CPX enabled with the NetScaler Observability Exporter support.
3. Deploy Zipkin, Elasticsearch, and Kibana using the YAML files.
4. Deploy NetScaler Observability Exporter using the YAML file.

Deploy application with tracing enabled

The following is a sample application deployment with tracing enabled.

> **Note:**
>
> If you have a pre-deployed web application, skip the steps 1 and 2.

1. Create a secret ingress.crt and key ingress.key using your own certificate and key.

   In this example, a secret, called *ing* in the default namespace, is created.

   ```
   kubectl create secret tls ing --cert=ingress.crt --key=ingress.
      key
   ```

2. Access the YAML file from watches-app-tracing.yaml to deploy the application.

   ```
   kubectl create -f watches-app-tracing.yaml
   ```

3. Define the specific parameters that you must import by specifying it in the ingress annotations of the application's YAML file, using the smart annotations in the ingress.

   ```
    ingress.citrix.com/analyticsprofile: '{
  "webinsight": {
  "httpurl":"ENABLED", "httpuseragent":"ENABLED", "httpHost":"
     ENABLED","httpMethod":"ENABLED","httpContentType":"ENABLED" }
    }
  '
   ```

   Note: The parameters are predefined in the `watches-app-tracing.yaml` file.

   For more information about annotations, see Ingress annotations documentation.

Deploy NetScaler CPX with the NetScaler Observability Exporter support

You can deploy NetScaler CPX enabled with the NetScaler Observability Exporter support.

While deploying NetScaler CPX, you can modify the deployment YAML file `cpx-ingress-tracing.yaml` to include the configuration information that is required for the NetScaler Observability Exporter support.

Perform the following steps to deploy a NetScaler CPX instance with the NetScaler Observability Exporter support:

1. Download the cpx-ingress-tracing.yaml and `cic-configmap.yaml` file.
2. Create a ConfigMap with the required key-value pairs and deploy the ConfigMap. You can use the `cic-configmap.yaml` file that is available, for the specific endpoint, in the directory.
3. Modify NetScaler CPX related parameters, as required. For example, add lines under `args` in the `cpx-ingress-tracing.yaml` file as following:

```
args:
  - --configmap
    default/cic-configmap
```

4. Edit the `cic-configmap.yaml` file to specify the following variables for NetScaler Observability Exporter in the `NS_ANALYTICS_CONFIG` endpoint configuration.

```
server: 'coe-zipkin.default.svc.cluster.local' # COE service
    FQDN
```

5. Deploy NetScaler CPX with the NetScaler Observability Exporter support using the following commands:

```
kubectl create -f cpx-ingress-tracing.yaml
kubectl create -f cic-configmap.yaml
```

> **Note:**
>
> If you have used a namespace other than *default*, change `coe-zipkin.default.svc.cluster.local` to `coe-zipkin.<desired-namespace>.svc.cluster.local`. If ADC is outside the Kubernetes cluster, then you must specify IP address and Nodport address of NetScaler Observability Exporter.

Deploy Zipkin, Elasticsearch, and Kibana using YAML files

To deploy Zipkin, Elasticsearch, and Kibana using YAML, perform the following steps:

1. Download the following YAML files:
   - zipkin.yaml
   - Elasticsearch.yaml

- kibana.yaml

2. Edit the namespace definition, if you want to use a custom namespace other than the *default*.
3. Run the following commands to deploy Zipkin, Elasticsearch, and Kibana:

```
kubectl create -f zipkin.yaml
kubectl create -f elasticsearch.yaml
kubectl create -f kibana.yaml
```

> **Note:**
>
> Zipkin, Elasticsearch, and Kibana are deployed in the default namespace of the same Kubernetes cluster.

Deploy NetScaler Observability Exporter using the YAML file

You can deploy NetScaler Observability Exporter using the YAML file. Download the coe-zipkin.yaml file.

To deploy NetScaler Observability Exporter using the Kubernetes YAML, run the following command in the Elasticsearch endpoint:

```
kubectl create -f coe-zipkin.yaml
```

> **Note:**
>
> Modify the YAML file for NetScaler Observability Exporter if you have a custom namespace other than the default.

Verify the NetScaler Observability Exporter deployment

To verify the NetScaler Observability Exporter deployment, perform the following:

1. Verify the deployment by sending a request to the application using the following command.

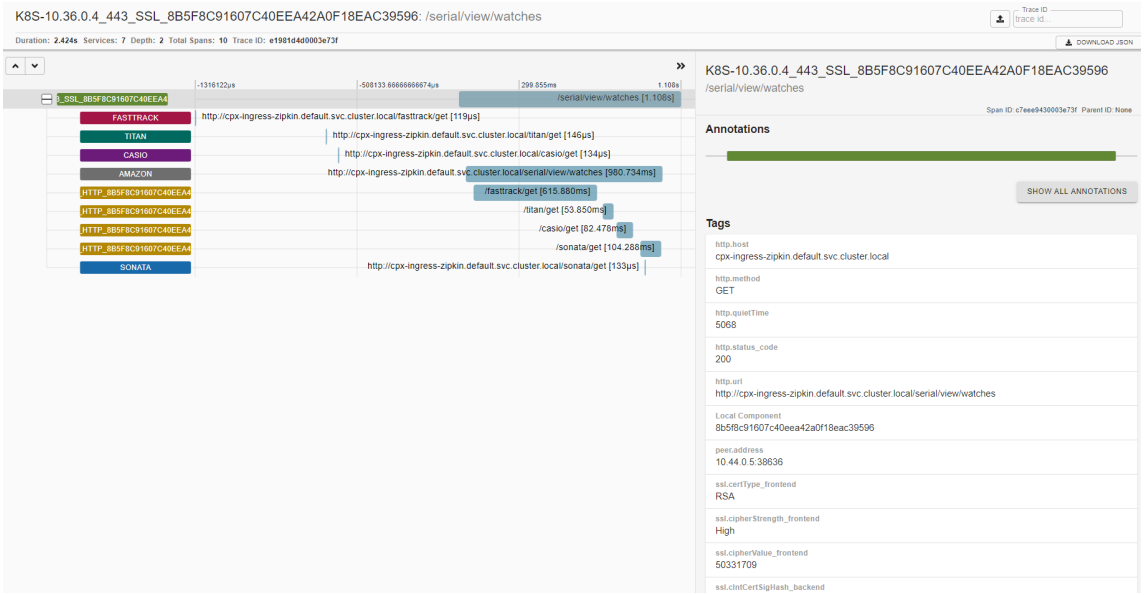
```
kubectl run -i --tty busybox --image=busybox --restart=Never --
    rm -- wget --no-check-certificate "https://cpx-ingress-zipkin
    .default.svc.cluster.local/serial/view/watches"
```

2. Open the Zipkin user interface using the Kubernetes node IP address and nodeport.

```
http://*k8-node-ip-address*:*node-port*/
```

   In the following image, you can view the traces of the *Watches* application. The *Watches* application has multiple microservices for each watches type, communicating with each other to serve the application data. The trace data shows application FASTTRACK took more time to

serve when compare to other micro services. In this way, you can identify the slow performing workloads and troubleshoot it.



You can view raw data on your Kibana dashboard too. Open Kibana using the `http://<node-ip>:<node-port>` and commence with defining a *zipkin* index pattern.

Use the `timestamp_millis` field as the timestamp field. After creating the index pattern, click the **Discover** tab and you can view the trace information collected by Zipkin.

For information on troubleshooting related to NetScaler Observability Exporter, see NetScaler CPX troubleshooting.

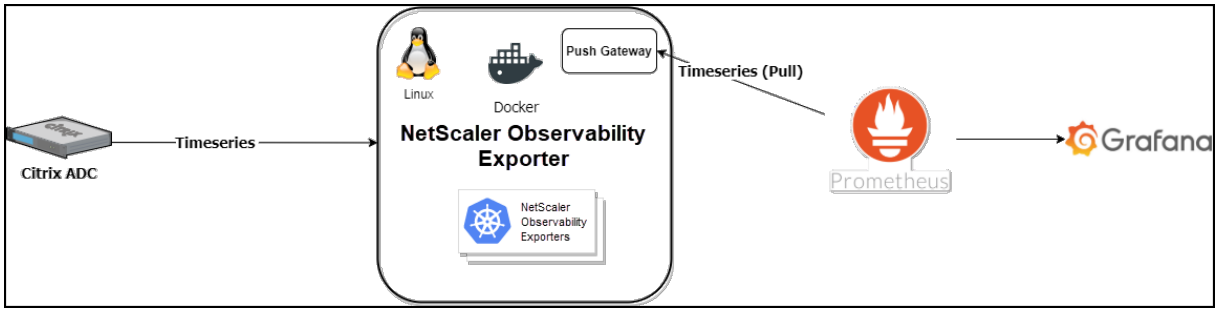
NetScaler Observability Exporter with Prometheus and Grafana

December 31, 2023

You can configure Prometheus as an endpoint to pull data from NetScaler Observability Exporter. You can also configure Grafana to visualize the same data graphically.

NetScaler Observability Exporter has a push-gateway server that listens to port 5563 to serve metrics based on pull requests from Prometheus. NetScaler Observability Exporter exports time series data

to Prometheus.



Deploy NetScaler Observability Exporter

You can deploy NetScaler Observability Exporter using the YAML file. Based on your NetScaler deployment, deploy NetScaler Observability Exporter either outside or inside Kubernetes clusters. You can deploy NetScaler Observability Exporter as a pod inside the Kubernetes cluster or on the NetScaler MPX or VPX appliance outside the cluster.

Prerequisites

- Ensure that you have a Kubernetes cluster with kube-dns or CoreDNS addon enabled.

Deploying NetScaler Observability Exporter with the Prometheus endpoint includes the following tasks:

- Deploy a sample application
- Deploy NetScaler CPX with support enabled for NetScaler Observability Exporter
- Deploy Prometheus and Grafana using YAML files
- Deploy NetScaler Observability Exporter using the YAML file
- Configure NetScaler to export metrics (optional)
- Configure Prometheus (optional) to pull telemetry data
- Configure Grafana
- Create Grafana visualization

Deploy a sample application

The following is an example procedure for deploying a sample webserver application.

> **Note:**
>
> If you have a pre-deployed web application, skip the steps from step 1 to step 3.

1. Create a secret ingress.crt and key ingress.key using your own certificate and key.

   In this example, a secret, called *ing* in the default namespace, is created.

   ```
   kubectl create secret tls ing --cert=ingress.crt --key=ingress.
      key
   ```

2. Access the YAML file from webserver-es.yaml to deploy the application.

   ```
   kubectl create -f webserver-es.yaml
   ```

3. Define the specific parameters that you must import by specifying it in the ingress annotations of the application's YAML file, using the smart annotations in the ingress.

   ```
    ingress.citrix.com/analyticsprofile: '{
  "webinsight": {
  "httpurl":"ENABLED", "httpuseragent":"ENABLED", "httpHost":"
     ENABLED","httpMethod":"ENABLED","httpContentType":"ENABLED" }
    }
  '
   ```

   > **Note:**
   >
   > The parameters are predefined in the `webserver-es.yaml` file.

   For more information about Annotations, see Ingress annotations documentation.

Deploy NetScaler CPX with support enabled for NetScaler Observability Exporter

You can deploy NetScaler CPX as a side car with the NetScaler Observability Exporter support enabled along with NetScaler Ingress Controller. You can modify the NetScaler CPX YAML file `cpx-ingress-es.yaml` to include the configuration information that is required for the NetScaler Observability Exporter support.

The following is a sample application deployment procedure.

1. Download the cpx-ingress-prometheus.yaml and cic-configmap.yaml file.
2. Create a ConfigMap with the required key-value pairs and deploy the ConfigMap. You can use the `cic-configmap.yaml` file that is available, for the specific endpoint, in the directory.
3. Modify NetScaler CPX related parameters, as required.
4. Edit the `cic-configmap.yaml` file and specify the following variables for NetScaler Observability Exporter in the `NS_ANALYTICS_CONFIG` endpoint configuration.

   ```
   server: 'coe-prometheus.default.svc.cluster.local' # COE service
       FQDN
   ```


> **Note:**
>
> If you have used a namespace other than *default*, change `coe-prometheus.default.svc.cluster.local` to `coe-prometheus.<desired-namespace>.svc.cluster.local`.

5. Deploy NetScaler CPX with the NetScaler Observability Exporter support using the following commands:

```
kubectl create -f cpx-ingress-prometheus.yaml
kubectl create -f cic-configmap.yaml
```

Deploy Prometheus and Grafana using YAML files

To deploy Prometheus and Grafana using YAML files, perform the following steps:

1. Download the `Prometheus-Grafana` YAML file from prometheus-grafana.yaml.
2. Edit the namespace definition if you want to use a different namespace other than *default*.
3. Run the following commands to deploy Prometheus and Grafana:

```
kubectl create -f prometheus-grafana.yaml
```

> **Note:**
>
> Prometheus and Grafana are deployed in the default namespace of the same Kubernetes cluster.

Deploy NetScaler Observability Exporter using the YAML file

You can deploy NetScaler Observability Exporter using the YAML file. Download the YAML file from coe-prometheus.yaml.

- For NetScaler Observability Exporter version 1.3.001 and previous versions, you can use the ConfigMap configuration provided in the `coe-prometheus.yaml` YAML file.
- For NetScaler Observability Exporter version 1.4.001, you need to modify the ConfigMap in the `coe-prometheus.yaml` file as follows before deployment.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coe-config-prometheus
data:
  lstreamd_default.conf: |
    {

```



```
 9
10         "Endpoints": {
11
12             "ZIPKIN": {
13
14                             "ServerUrl":"http://0.0.0.0:0",
15                             "RecordType":{
16    }
17   ,
18                             "PrometheusMode":"yes"
19                            }
20
21                 }
22
23         }
```

To deploy NetScaler Observability Exporter using the Kubernetes YAML, run the following command:

```
1    kubectl create -f coe-prometheus.yaml
```

> **Note:**
>
> Modify the YAML file for NetScaler Observability Exporter if you have a custom namespace.

Configure NetScaler to export metrics (optional)

> **Note:**
>
> If you do not use NetScaler Ingress Controller to configure NetScaler, then you can do the following manual configuration on your NetScaler.

You can manually configure NetScalers to export metrics to the NetScaler Observability Exporter. Specify the NetScaler Observability Exporter IP/FQDN address as an HTTP service and combine it to the default `ns_analytics_time_series_profile` analytics profile. Enable the metrics export and set the output mode to Prometheus.

The following is a sample configuration:

```
1 add server COE_instance 192.168.1.102
2 add service coe_metric_collector_svc_192.168.1.102 COE_instance HTTP
     5563
3 set analytics profile ns_analytics_time_series_profile -collector
     coe_metric_collector_svc_192.168.1.102 -Metrics ENABLED -OutputMode
     Prometheus
```


Configure Prometheus (optional) to pull telemetry data

Prometheus services are available as Docker images on Quay container registry and Docker Hub.

To launch Prometheus and expose it on port 9090, run the following command:

```
docker run -p 9090:9090 prom/prometheus
```

To manually add NetScaler Observability Exporter as scrape target, edit the `prometheus.yaml` file. Specify the NetScaler Observability Exporter IP/FQDN address and the port 5563 as the scrape target in the YAML file.

```
scrape_configs:
  - job_name: coe
    static_configs:
      - targets: ['192.168.1.102:5563']
```

Configure Grafana

In the current deployment, a Prometheus server has already been added as a data source. If you use an existing Prometheus server for the deployment, ensure to add the same as a data source on your Grafana. For more information, see Grafana support for Prometheus.

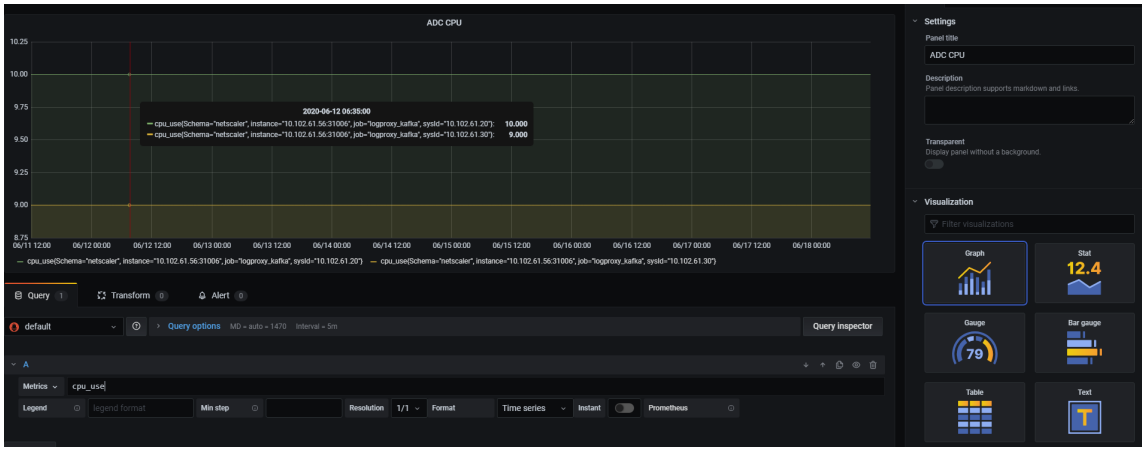
Create Grafana visualization

You can create a Grafana dashboard and select the key metrics and the visualization type that is suitable for the data.

The following procedure shows adding of the ADC CPU metric to a Grafana panel:

1. Specify the Panel Title as *ADC CPU*.
2. In the Query tab, for the query A, specify the metric as *cpu_use*.
3. In the Settings tab, select the **Visualization type**.

   You can modify the data and its representation in Grafana. For more information, see Grafana Documentation.



Import pre-built dashboards for Grafana

You can also import pre-built dashboards to Grafana. See the available Dashboards.

For information on troubleshooting related to NetScaler Observability Exporter, see NetScaler CPX troubleshooting.

NetScaler Observability Exporter with Elasticsearch as endpoint

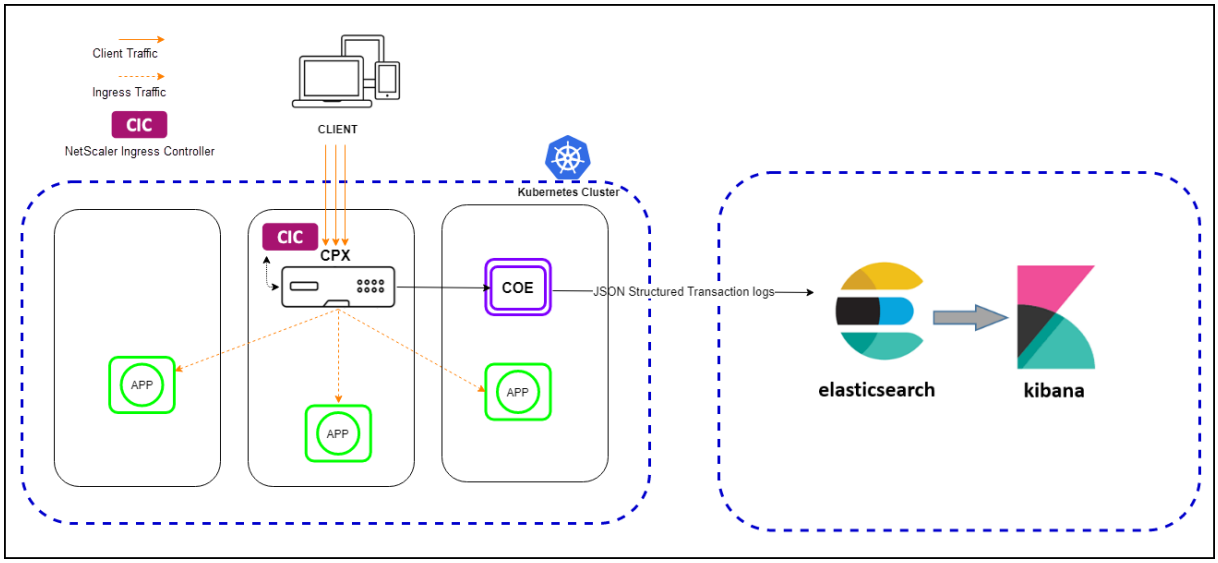
September 27, 2025

NetScaler Observability Exporter is a container that collects metrics and transactions from NetScaler. It transforms the data into the supported format (such as JSON) and exports data to Elasticsearch as an endpoint. Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable, and full-text search engine with an HTTP web interface and schema-free JSON documents.

Deploy NetScaler Observability Exporter

You can deploy NetScaler Observability Exporter using the YAML file. Based on your NetScaler deployment, you can deploy NetScaler Observability Exporter either outside or inside Kubernetes clusters. You can deploy NetScaler Observability Exporter as a pod inside the Kubernetes cluster or on NetScaler MPX or VPX appliance outside the cluster.

The following diagram illustrates a NetScaler as an Ingress Gateway with the NetScaler Ingress Controller and NetScaler Observability Exporter as sidecars. NetScaler Observability Exporter sends NetScaler application metrics and transaction data to Elasticsearch and the same data exports to Kibana. Kibana provides a graphical representation of the data.



Prerequisites

- Ensure that you have a Kubernetes cluster with `kube-dns` or `CoreDNS` addon enabled.

In the following procedure, the YAML file is used to deploy NetScaler Observability Exporter in the Kubernetes *default* namespace. If you want to deploy in a private namespace other than the *default*, edit the YAML file to specify the namespace.

The following is a sample application deployment procedure.

> **Note:**
>
> If you have a pre-deployed web application, skip the steps 1 and 2.

1. Create a secret ingress.crt and key ingress.key using your own certificate and key.

   In this example, a secret, called *ing* in the default namespace, is created.

   ```
   kubectl create secret tls ing --cert=ingress.crt --key=ingress.
      key
   ```

2. Access the YAML file from webserver-es.yaml to deploy the application.

   ```
   kubectl create -f webserver-es.yaml
   ```

3. Define the specific parameters that you must import by specifying it in the ingress annotations of the application's YAML file, using the smart annotations in the ingress.

   ```
    ingress.citrix.com/analyticsprofile: '{
   "webinsight": {
   "httpurl":"ENABLED", "httpuseragent":"ENABLED", "httpHost":"
      ENABLED","httpMethod":"ENABLED","httpContentType":"ENABLED" }
   ```



```
    }
'
```

> **Note:**
>
> The parameters are predefined in the `webserver-es.yaml` file.

For more information about Annotations, see Ingress annotations documentation.

Deploy NetScaler CPX with the NetScaler Observability Exporter support

You can deploy NetScaler CPX as a side car with the NetScaler Observability Exporter support enabled along with NetScaler Ingress Controller. You can modify the NetScaler CPX YAML file `cpx-ingress-es.yaml` to include the configuration information that is required for the NetScaler Observability Exporter support.

Perform the following steps to deploy a NetScaler CPX instance with the NetScaler Observability Exporter support:

1. Download the cpx-ingress-es.yaml and cic-configmap.yaml file.
2. Create a ConfigMap with the required key-value pairs and deploy the ConfigMap. You can use the `cic-configmap.yaml` file that is available, for the specific endpoint, in the directory.
3. Modify NetScaler CPX related parameters, as required.
4. Edit the `cic-configmap.yaml` file and specify the following variables for NetScaler Observability Exporter in the `NS_ANALYTICS_CONFIG` endpoint configuration.

```
  server: 'coe-es.default.svc.cluster.local' # COE service FQDN
```

> **Note:**
>
> If you have used a namespace other than *default*, change `coe-es.default.svc.cluster.local to` to `coe-es.<desired-namespace>.svc.cluster.local`. If ADC is outside the Kubernetes cluster, then you must specify IP address and nodport address of NetScaler Observability Exporter.

5. Deploy NetScaler CPX with the NetScaler Observability Exporter support using the following commands:

```
kubectl create -f cpx-ingress-es.yaml
kubectl create -f cic-configmap.yaml
```


Deploy Elasticsearch and Kibana using YAML files

1. Download the Elasticsearch YAML file from elasticsearch.yaml and the Kibana YAML file from kibana.yaml.
2. Edit the namespace definition, if you want to use a different namespace other than *default*.
3. Run the following commands to deploy Elasticsearch and Kibana:

```
kubectl create -f elasticsearch.yaml
kubectl create -f kibana.yaml
```

Note: Elasticsearch and Kibana are deployed in the default namespace of the same Kubernetes cluster.

Deploy NetScaler Observability Exporter using the YAML file

You can deploy NetScaler Observability Exporter using the YAML file. Download the YAML file from coe-es.yaml.

To deploy NetScaler Observability Exporter using the Kubernetes YAML, run the following command in the Elasticsearch endpoint:

```
kubectl create -f coe-es.yaml
```

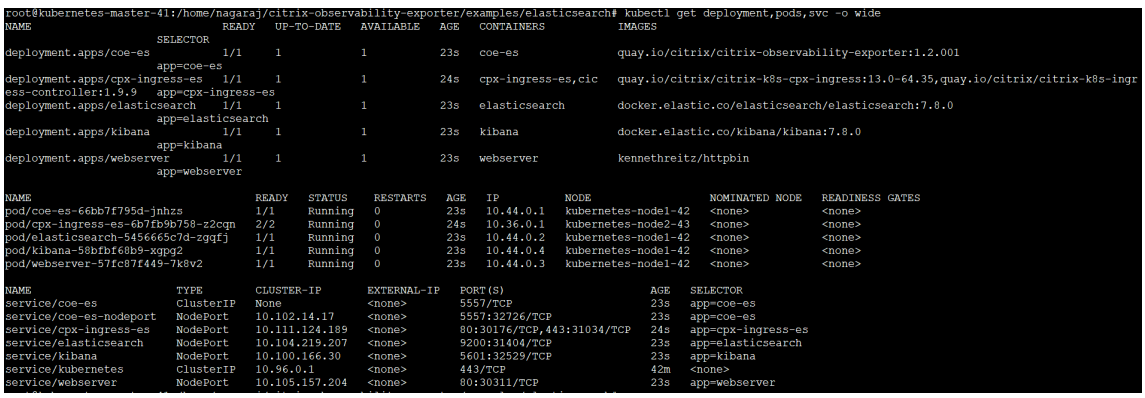
> **Note:**
>
> Modify the YAML file for NetScaler Observability Exporter if you have a custom namespace.

Verify the NetScaler Observability Exporter deployment

To verify the NetScaler Observability Exporter deployment, perform the following:

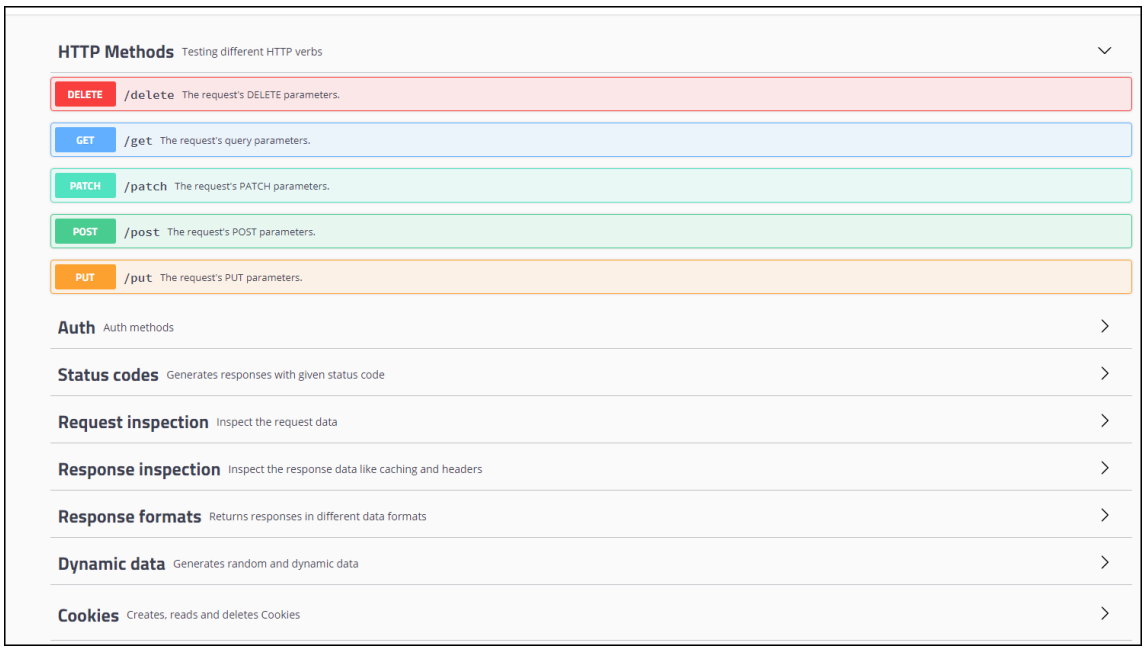
1. Verify the deployment using the following command:

```
kubectl get deployment,pods,svc -o wide
```



2. Access the application with a browser using the URL: `https://kubernetes-node-IP:cpx-ingress-es nodeport/`.

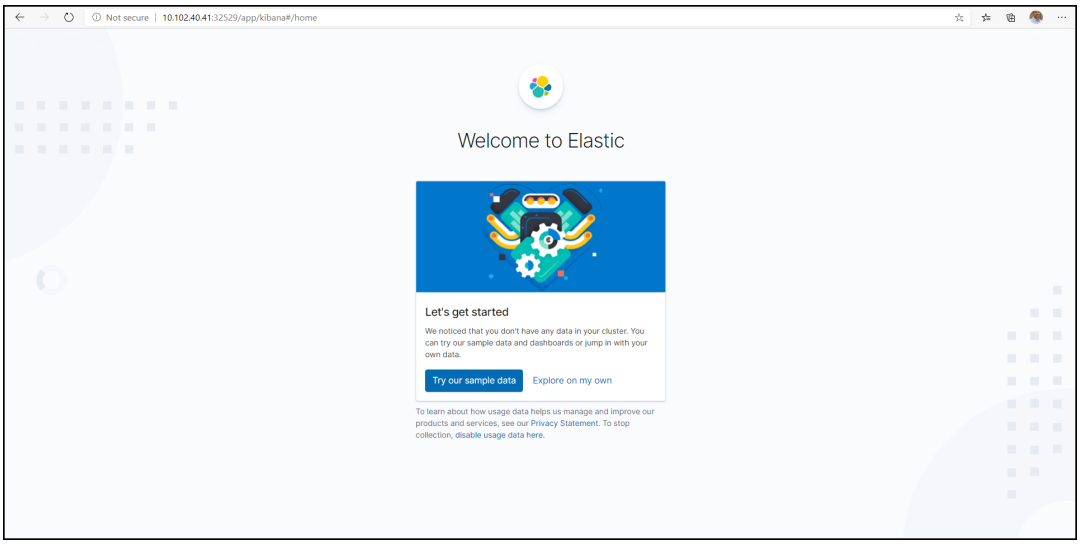
   For example, from Step 1, access http://10.102.40.41:30176/ in which, `10.102.40.41` is one of the Kubernetes node IPs.



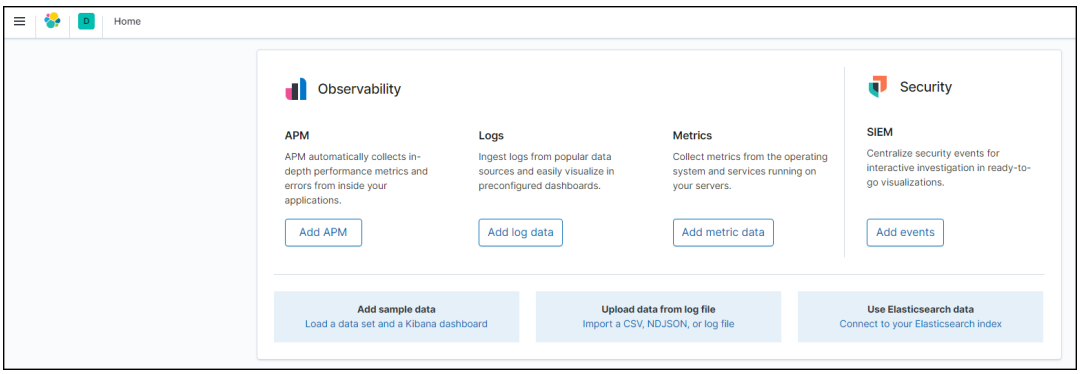
3. Access Kibana with a browser using the URL: `https://<kubernetes-node-IP>:<kibana nodeport>/`.

   For example, from step 1, access http://10.102.40.41:32529/ in which, `10.102.40.41` is one of the Kubernetes node IPs.

   a) Click **Explore on my own**.

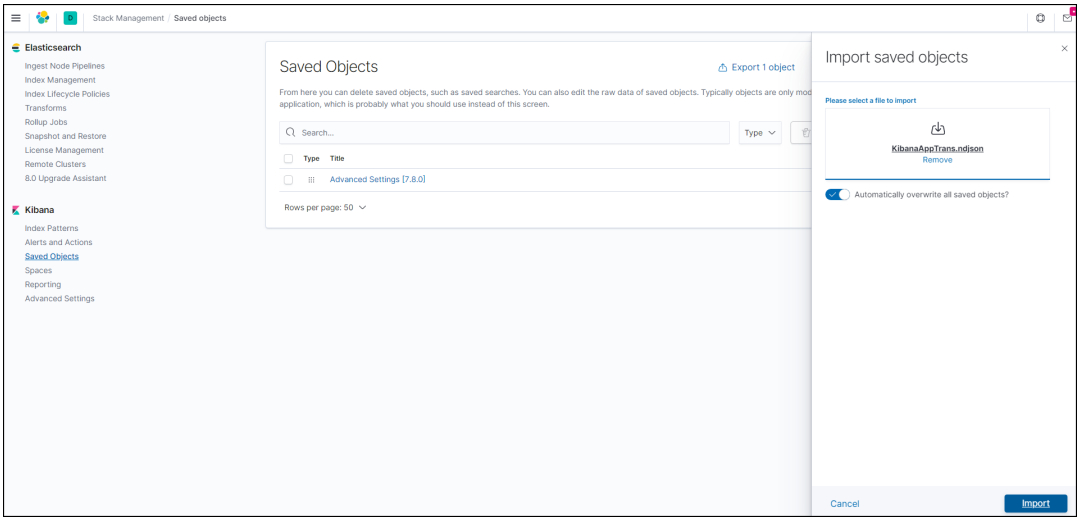


   b) Click **Connect to your Elasticsearch index**.

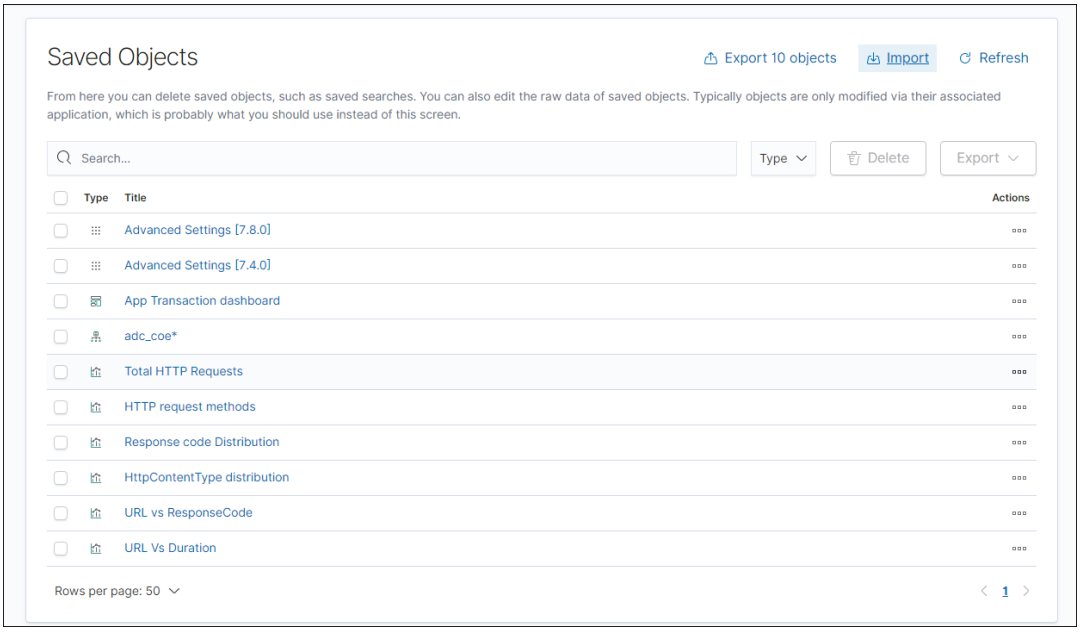


c) Click **Saved Objects**.

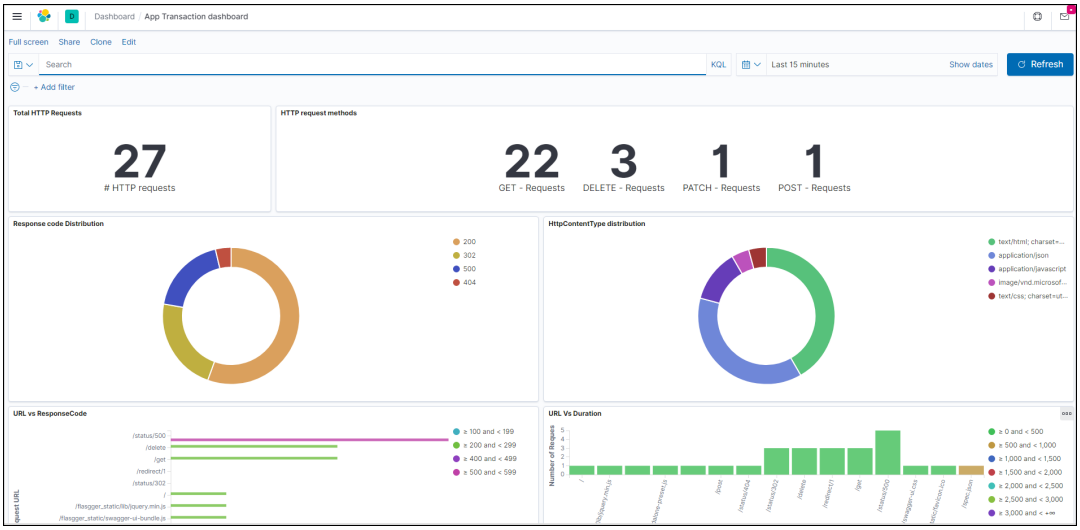
d) Download and import the Kibana Dashboard from KibanaAppTrans.ndjson.



e) Click **App Transaction dashboard**.



The dashboard appears.




```
green  open .kibana-event-log-7.8.0-000001 D-vgsLvURBSSWvSe9VHFOg 1 0      1     0   5.3kb   5.3kb
yellow open ns_coe_d22-28-m9_y2020         hbW781VkTPqpSy81sR6zKg 1 1   5953     0     3mb     3mb
green  open .apm-custom-link               D2lrHlOcTL65za3dHXHuuA 1 0      0     0    208b    208b
green  open .kibana_task_manager_1         z9DwQFW3T8O7_TX3kQS6Hg 1 0      3     0  53.8kb  53.8kb
yellow open ns_coe_d23_m9_y2020            Kb34XIXxQhOl3_zCYFozNw 1 1 333241 65472 196.9mb 196.9mb
yellow open ns_coe_h5_d23_m9_y2020         kbDaKbrPSEif6qh7rQ1LOw 1 1  33780  6812  22.2mb  22.2mb
green  open .apm-agent-configuration       XBQbte6-SC2qjH1Vyy5ddQ 1 0      0     0    208b    208b
yellow open ns_coe_y2020                   kZ-SyPOSSeGGUvoyjOtmOw 1 1 307683 80551 204.2mb 204.2mb
green  open .kibana_1                      NU8JzkSAQc-JLpSxFia4ew 1 0      4     0  31.4kb  31.4kb
```

Integrate NetScaler with multiple NetScaler Observability Exporter instances manually

You can also configure NetScaler Observability Exporter manually. We recommend deploying NetScaler Observability Exporter in an automated way with the YAML file as described in the preceding sections. You can also perform manual configuration for NetScaler in the MPX and VPX form factors.

```
enable feature appflow®
enable ns mode ULFD
add dns nameserver <KUBE-CoreDNS>
add server COEsvr <FQDN/IP>
add servicegroup COEsvcgrp LOGSTREAM  -autoScale™ DNS
bind servicegroup COEsvcgrp COEsvr <PORT>
add lb vserver COE LOGSTREAM 0.0.0.0 0
bind lb vserver COE COEsvcgrp
add analytics profile web_profile -collectors COE -type
   webinsight -httpURL ENABLED -httpHost ENABLED -
   httpMethod ENABLED -httpUserAgent ENABLED -
   httpContentType ENABLED
add analytics profile tcp_profile -collectors COE -type
   tcpinsight
bind lb vserver <WEB-VSERVER> -analyticsProfile web_profile
```



```
bind lb vserver <WEB-VSERVER> -analyticsProfile tcp_profile

# To enable metrics push to prometheus
add service metrichost_SVC <IP> HTTP <PORT>
set analyticsprofile ns_analytics_time_series_profile -
   collectors metrichost_SVC -metrics ENABLED -outputMode
   prometheus
```

Add NetScaler Observability Exporter using FQDN

```
enable feature appflow
enable ns mode ULFD
add dns nameserver <KUBE-CoreDNS>
add server COEsvr <FQDN>
add servicegroup COEsvcgrp LOGSTREAM  -autoScale DNS
bind servicegroup COEsvcgrp COEsvr <PORT>
add lb vserver COE LOGSTREAM 0.0.0.0 0
bind lb vserver COE COEsvcgrp
add analytics profile web_profile -collectors COE -type
   webinsight -httpURL ENABLED -httpHost ENABLED -
   httpMethod ENABLED -httpUserAgent ENABLED -
   httpContentType ENABLED
add analytics profile tcp_profile -collectors COE -type
   tcpinsight
bind lb vserver <WEB-VSERVER> -analyticsProfile web_profile
bind lb vserver <WEB-VSERVER> -analyticsProfile tcp_profile

# To enable metrics push to prometheus
add service metrichost_SVC <IP> HTTP <PORT>
set analyticsprofile ns_analytics_time_series_profile -
   collectors metrichost_SVC -metrics ENABLED -outputMode
   prometheus
```

To verify if NetScaler sends application data logs to NetScaler Observability Exporter:

```
    nsconmsg -g lstream_tot_trans_written -d current
```

The counter value indicates that the number of application transactions (for example, HTTP transactions) which have been sent to NetScaler Observability Exporter.

```
root@ns# nsconmsg -g trans_written -d current
Displaying performance information
NetScaler V20 Performance Data
reltime:mili second between two records Thu Oct  8 07:11:00 2020
  Index   rtime totalcount-val      delta rate/sec symbol-name&device-no
      0   63000          24142        632       90 lstream_tot_trans_written
      1    7000          43635      19493     2784 lstream_tot_trans_written
```

If the application traffic rate (for example, HTTP req/sec) that is sent to NetScaler Observability Exporter is not equal to `lstream_tot_trans_written`, you can verify the same using the following command:


```
nsconmsg -g nslstream_err_ulf_data_not_sendable -d current
```

The counter value indicates that NetScaler cannot send the data to NetScaler Observability Exporter due to network congestion, unavailability of network bandwidth, and so on. The data is stored in the available buffers.

Information about various transaction data and individual fields, and their datatype are available in the following location on the NetScaler:

```
shell/netscaler/appflow/ns_ipfix.yaml
```

```
# AppFlow Information Elements exported by NetScaler
---
0:
  1:
  - uint64      ---> Data Type
  - octetTotalCount   ---> Field Name, followed by description
  - "The number of octets since the previous report (if any) in incoming packets for this flow at the Observation Point. The number of octets includes IP header(s) and IP payload."
  2:
  - uint64
  - packetTotalCount
  - "The number of incoming packets since the previous report (if any) for this Flow at the Observation Point."
```

To verify that if application transaction records are exported from NetScaler to NetScaler Observability Exporter, use the following command:

```
nsconmsg -g appflow_tmpl -d current
```

```
root@ns# nsconmsg -g appflow_tmpl -d current
Displaying performance information
NetScaler V20 Performance Data
NetScaler NS13.0: Build 67.37.nc, Date: Sep 30 2020, 09:53:44   (64-bit)


reltime:mili second between two records Thu Oct  8 07:24:25 2020
  Index   rtime totalcount-val      delta rate/sec symbol-name&device-no
      0    7000          75597          1        0 appflow_tmpl_v4_l7_clt2ns_complete
      1       0          75597          1        0 appflow_tmpl_v4_l7_srvr2ns_complete
      2       0             10          1        0 appflow_tmpl_v46_ulfd_burst_eot
      3       0          75597          1        0 appflow_tmpl_v46_ulfd_server_eot
      4       0          75606          2        0 appflow_tmpl_v46_ulfd_client_eot
```

Location of metrics data export logs to NetScaler for time series data:

```
/var/nslog/metrics_prom.log
```

To verify Elasticsearch related counters, run the following command:

```
kubectl exec -it <cpx-pod-name> [-c <cpx-container-name>] [-n <
   namespace-name>] -- bash

tail -f /var/ulflog/counters/lstrmd\_counters\_codes.log | grep -
   iE \"\(http\_reqs\_done|elk)\"
```

Find the logs in the following location to verify that the NetScaler Observability Exporter configuration is applied correctly:


```
vi /var/logproxy/lstreamd/conf/lstreamd.conf
```

If NetScaler Observability Exporter fails, you can collect logs and files available at the following location and contact NetScaler support.

```
/var/crash/ (Loation of the coredump files, if any.)
/var/ulflog/ (Location of the `libulfd` logs and counter details
    .)
/var/log  (Location of the console logs, lstreamd logs and so on
    .)
```

For information on troubleshooting related to NetScaler Observability Exporter, see NetScaler CPX troubleshooting.

NetScaler Observability Exporter with Kafka as endpoint

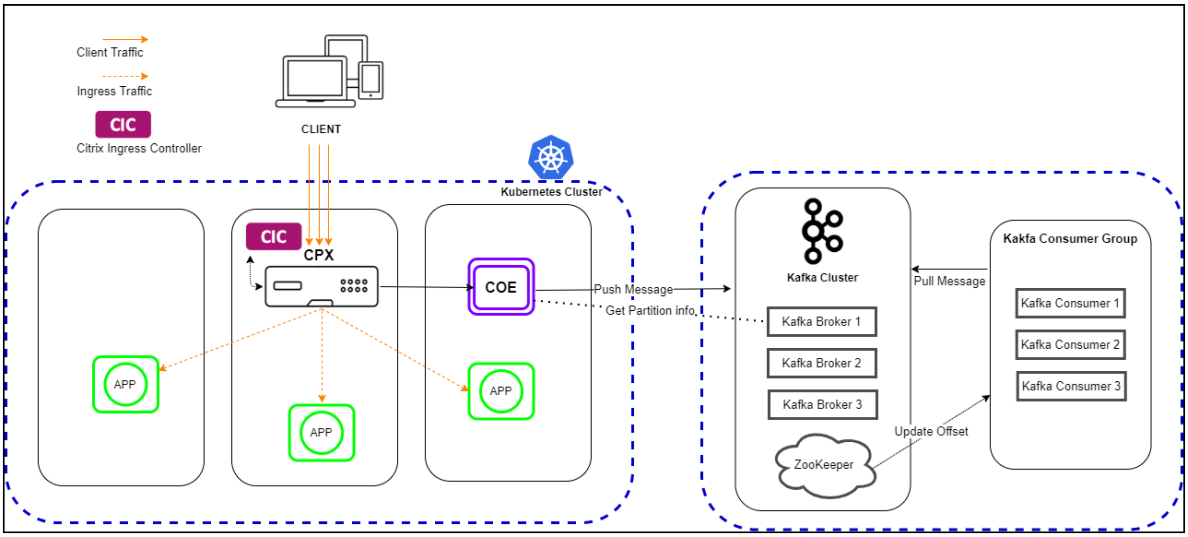
September 27, 2025

NetScaler Observability Exporter is a container that collects metrics and transactions from NetScaler. It also transforms the data into the formats (such as AVRO) that are supported in Kafka and exports the data to the endpoint. Kafka is an open-source and distributed event streaming platform for high-performance data pipelines and streaming analytics.

Deploy NetScaler Observability Exporter

You can deploy NetScaler Observability Exporter using the YAML file. Based on the NetScaler deployment, you can use NetScaler Observability Exporter to export metrics and transaction data from NetScaler. You can deploy NetScaler CPX either as a pod inside the Kubernetes cluster or on NetScaler MPX or VPX form factor outside the cluster.

The following diagram illustrates a NetScaler as an Ingress Gateway with NetScaler Observability Exporter as a sidecar. It sends NetScaler application transaction data to Kafka.



Prerequisites

- Ensure that you have a Kubernetes cluster with `kube-dns` or `CoreDNS` addon enabled.
- Ensure that the Kafka server is installed and configured.
- You must have a Kafka broker IP or FQDN address.
- You must have defined a Kafka topic *HTTP*.
- Ensure that you have Kafka Consumer to verify the data.

> **Note:**
>
> In this example scenario, the YAML file is used to deploy NetScaler Observability Exporter in the Kubernetes *default* namespace. If you want to deploy in a private Kubernetes namespace other than *default*, edit the YAML file to specify the namespace.

The following is a sample application deployment procdure.

> **Note:**
>
> If you have a pre-deployed web application, skip the step 1 and 2.

1. Create a secret ingress.crt and key ingress.key using your own certificate and key.

   In this example, a secret, called *ing* in the default namespace, is created.

   ```
   1    kubectl create secret tls ing --cert=ingress.crt --key=ingress.
           key
   ```

2. Access the YAML file from webserver-kafka.yaml to deploy a sample application.

   ```
   1    kubectl create -f webserver-kafka.yaml
   ```


3. Define the specific parameters that you must import by specifying it in the ingress annotations of the application's YAML file using the smart annotations in ingress.

```
 ingress.citrix.com/analyticsprofile: '{
"webinsight": {
"httpurl":"ENABLED", "httpuseragent":"ENABLED", "httpHost":"
   ENABLED","httpMethod":"ENABLED","httpContentType":"ENABLED" }
 }
'
```

> **Note:**
>
> The parameters are predefined in the `webserver-kafka.yaml` file.

For more information about Annotations, see Ingress annotations documentation.

Deploy NetScaler CPX with the NetScaler Observability Exporter support

You can deploy NetScaler CPX as a side car with the NetScaler Observability Exporter support. You can edit the NetScaler CPX YAML file, `cpx-ingress-kafka.yaml`, to include the configuration information that is required for NetScaler Observability Exporter support.

Perform the following steps to deploy a NetScaler CPX instance with the NetScaler Observability Exporter support:

1. Download the cpx-ingress-kafka.yaml and the cic-configmap.yaml files.
2. Create a ConfigMap with the required key-value pairs and deploy the ConfigMap. You can use the `cic-configmap.yaml` file that is available, for the specific endpoint, in the directory.
3. Modify NetScaler CPX related parameters, as required.
4. Edit the `cic-configmap.yaml` file and specify the following variables for NetScaler Observability Exporter in the `NS_ANALYTICS_CONFIG` endpoint configuration.

```
 server: 'coe-kafka.default.svc.cluster.local' # COE service FQDN
```

5. Deploy NetScaler CPX with the NetScaler Observability Exporter support using the following commands:

```
kubectl create -f cpx-ingress-kafka.yaml
kubectl create -f cic-configmap.yaml
```

> **Note:**
>
> If you have used a different namespace, other than *default*, then you must change from

> coe-kafka.**default**.svc.cluster.local to coe-kafka.<desired-namespace>.svc.cluster.local.

Deploy NetScaler Observability Exporter using YAML

You can deploy NetScaler Observability Exporter using the YAML file. Download the coe-kafka.yaml file that you can use for the NetScaler Observability Exporter deployment.

To deploy NetScaler Observability Exporter using the Kubernetes YAML, run the following command in the Kafka endpoint:

```
kubectl create -f coe-kafka.yaml
```

To edit the YAML file for the required changes, perform the following steps:

1. Edit the ConfigMap using the following YAML definition:

   > **Note:**
   >
   > Ensure that you specify the bootstrap Kafka broker addresses (separated by comma) and the desired Kafka topic.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coe-config-kafka
data:
  lstreamd\_default.conf: |
    {

        "Endpoints": {

          "KAFKA": {

            "ServerUrl": "X.X.X.X:9092,Y.Y.Y.Y:9092", #Specify the bootstrap Kafka broker addresses (separated by comma in case multiple bootstrap brokers are to be configured)
            "KafkaTopic": "HTTP", #Specify the desired kafka topic
            "RecordType": {

                "HTTP": "all",
                "TCP": "all",
                "SWG": "all",
                "VPN": "all",
                "NGS": "all",
                "ICA®": "all",
                "APPFW": "none",
                "BOT": "none",
```



```
                    "VIDEOOPT": "none",
                    "BURST_CQA": "none",
                    "SLA": "none",
                    "MONGO": "none"
                }
,
                "ProcessAlways": "yes",
                "FileSizeMax": "40",
                "ProcessYieldTimeOut": "500",
                "FileStorageLimit": "1000",
                "SkipAvro": "no",
                "AvroCompress": "yes"
               }

            }

        }
    **Note:** To export transactions in the JSON format, see [
        exporting transaction in JSON format to Kafka](#support-for-
        exporting-transactions-in-the-json-format-from-pageadc-
        observability-exporter-short-to-kafka).
```

2. Specify the host name and IP or FQDN address of the Kafka nodes. Use the following YAML definition for a three node Kafka cluster:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: coe-kafka
  labels:
    app: coe-kafka
spec:
  replicas: 1
  selector:
    matchLabels:
      app: coe-kafka
  template:
    metadata:
      name: coe-kafka
      labels:
        app: coe-kafka
    spec:
      hostAliases:
        - ip: "X.X.X.X" # Here we specify kafka node1 Ipaddress
          hostnames:
            - "kafka-node1"
        - ip: "Y.Y.Y.Y" # Here we specify kafka node2 Ipaddress
          hostnames:
            - "kafka-node2"
        - ip: "Z.Z.Z.Z" # Here we specify kafka node3 Ipaddress
          hostnames:
            - "kafka-node3"
      containers:
```



```
            - name: coe-kafka
              image: "quay.io/citrix/citrix-observability-exporter
                 :1.3.001"
              imagePullPolicy: Always
              ports:
                - containerPort: 5557
                  name: lstream
              volumeMounts:
                - name: lstreamd-config-kafka
                  mountPath: /var/logproxy/lstreamd/conf/
                     lstreamd_default.conf
                  subPath: lstreamd_default.conf
                - name: core-data
                  mountPath: /var/crash/
          volumes:
            - name: lstreamd-config-kafka
              configMap:
                name: coe-config-kafka
            - name: core-data
              emptyDir: {
    }
```

3. If necessary, edit the service configuration for exposing the NetScaler Observability Exporter port to NetScaler using the following YAML definition:

   Citrix-observability-exporter headless service:

```
    apiVersion: v1
    kind: Service
    metadata:
      name: coe-kafka
      labels:
        app: coe-kafka
    spec:
      clusterIP: None
      ports:
        - port: 5557
          protocol: TCP
      selector:
          app: coe-kafka
```

   Citrix-observability-exporter NodePort service

```
    apiVersion: v1
    kind: Service
    metadata:
      name: coe-kafka-nodeport
      labels:
        app: coe-kafka
    spec:
      type: NodePort
      ports:
        - port: 5557
```



```
            protocol: TCP
        selector:
            app: coe-kafka
```

Verify the NetScaler Observability Exporter deployment

To verify the NetScaler Observability Exporter deployment, perform the following:

1. Verify the deployment using the following command:

```
kubectl get deployment,pods,svc -o wide
```

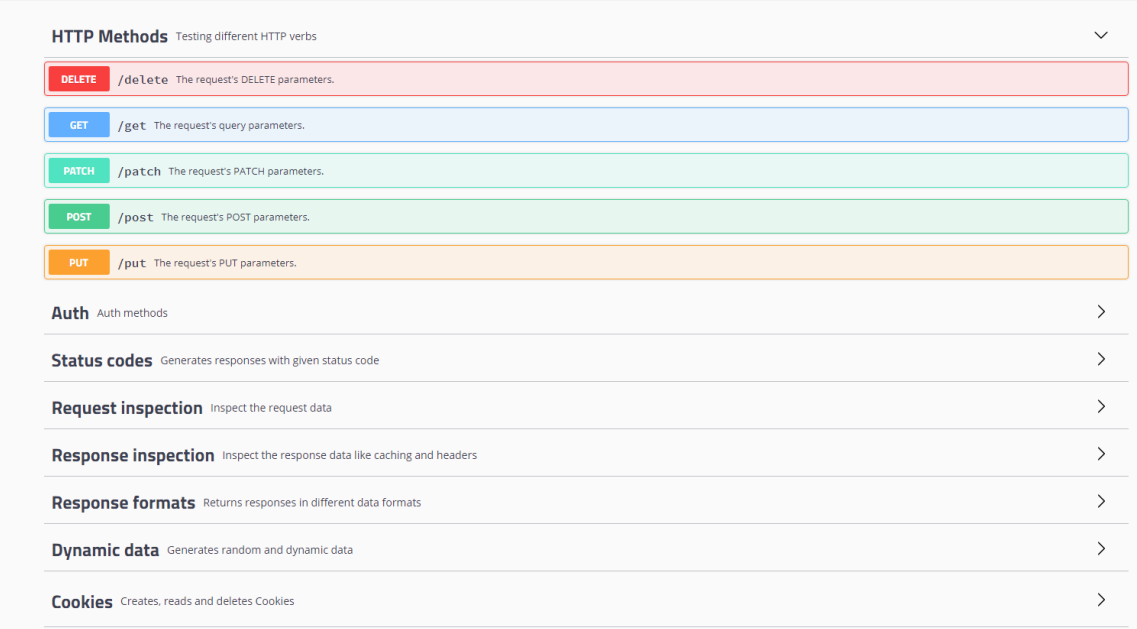
```
root@kubernetes-master:~# kubectl get deployment,pods,svc -o wide
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS              IMAGES
                                SELECTOR
deployment.apps/coe-kafka             1/1     1            1           28h   coe-kafka               nslogproxy:latest
                                app=coe-kafka
deployment.apps/cpx-ingress-kafka   1/1     1            1           28h   cpx-ingress-kafka,cic   quay.io/citrix/citrix-k8s-cpx-ingress:13.0-64.35,quay.io/citrix/citrix-k8
s-ingress-controller:1.9.9   app=cpx-ingress-kafka
deployment.apps/webserver             1/1     1            1           28h   webserver               kennethreitz/httpbin
                                app=webserver

NAME                                    READY   STATUS    RESTARTS   AGE   IP          NODE               NOMINATED NODE   READINESS GATES
pod/coe-kafka-56d96bf74-dcwdc           1/1     Running   0          28h   10.38.0.1   kubernetes-slave2   <none>           <none>
pod/cpx-ingress-kafka-668575f59b-hchkq  2/2     Running   0          28h   10.32.0.2   kubernetes-slave1   <none>           <none>
pod/webserver-7fbdf8d9c4-77xbk          1/1     Running   0          28h   10.38.0.2   kubernetes-slave2   <none>           <none>

NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                       AGE    SELECTOR
service/coe-kafka             ClusterIP   None            <none>        5557/TCP                      28h    app=coe-kafka
service/coe-kafka-nodeport    NodePort    10.105.46.33    <none>        5557:31005/TCP                28h    app=coe-kafka
service/cpx-ingress-kafka     NodePort    10.107.65.113   <none>        80:31202/TCP,443:32204/TCP    28h    app=cpx-ingress-kafka
service/kubernetes            ClusterIP   10.96.0.1       <none>        443/TCP                       125d   <none>
service/webserver             NodePort    10.110.68.1     <none>        80:31997/TCP                  28h    app=webserver
```

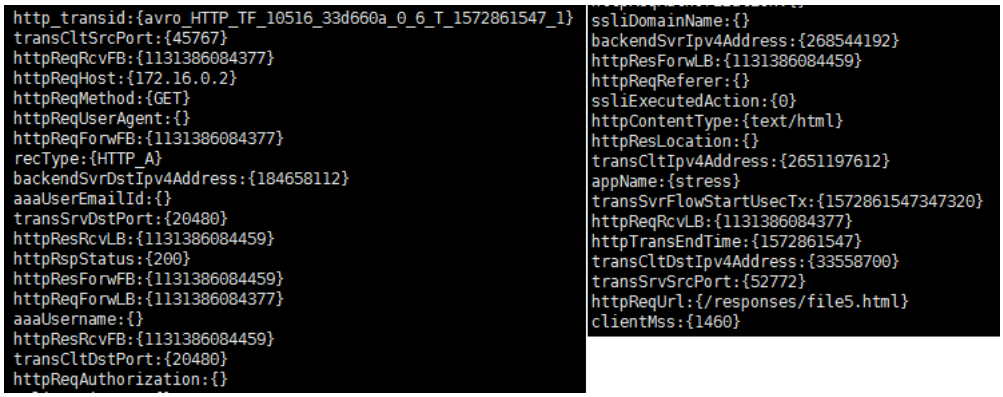
2. Access the application with a browser using the URL: `https://<kubernetes-node-IP>:<cpx-ingress-kafka nodeport>`.

```
For example, from step 1, access <http://10.102.61.56:31202/> in
    which, `10.102.61.56` is one of the Kubernetes node IPs.
```



3. Use Kafka Consumer to view the transaction data. Access kafka Consumer from PythonKafka-Consumer.

The following image shows sample data from Kafka Consumer.



Integrate NetScaler with multiple NetScaler Observability Exporter instances manually

You can configure NetScaler Observability Exporter manually in NetScaler. Manual configuration is suitable for NetScaler in MPX and VPX form factors. Citrix recommends deploying NetScaler Observability Exporter in the automated way using the YAML file as described in the preceding sections.

For information about deploying NetScaler Observability Exporter (coe-kafka.yaml) and web application (webserver-kafka.yaml), see the preceding sections.

```
enable feature appflow
enable ns mode ULFD
add service COE_svc1 <COE IP1> LOGSTREAM <COE PORT1>
add service COE_svc2 <COE IP2> LOGSTREAM <COE PORT2>
add service COE_svc3 <COE IP3> LOGSTREAM <COE PORT3>
add lb vserver COE LOGSTREAM 0.0.0.0 0
bind lb vserver COE COE_svc1
bind lb vserver COE COE_svc2
bind lb vserver COE COE_svc3
add analytics profile web_profile -collectors COE -type webinsight -
    httpURL ENABLED -httpHost ENABLED -httpMethod ENABLED -httpUserAgent
     ENABLED -httpContentType ENABLED
add analytics profile tcp_profile -collectors COE -type tcpinsight
bind lb/cs vserver <WEB-PROXY> -analyticsProfile web_profile
bind lb/cs vserver <WEB-PROXY> -analyticsProfile tcp_profile
# To enable metrics push to prometheus
add service metrichost_SVC <IP> HTTP <PORT>
set analyticsprofile ns_analytics_time_series_profile -collectors
    metrichost_SVC -metrics ENABLED -outputMode prometheus
```

Add NetScaler Observability Exporter using FQDN

```
enable feature appflow
enable ns mode ULFD
add dns nameserver <KUBE-CoreDNS>
add server COEsvr <FQDN>
```



```
add servicegroup COEsvcgrp LOGSTREAM  -autoScale DNS
bind servicegroup COEsvcgrp COEsvr <PORT>
add lb vserver COE LOGSTREAM 0.0.0.0 0
bind lb vserver COE COEsvcgrp
add analytics profile web_profile -collectors COE -type webinsight -
    httpURL ENABLED -httpHost ENABLED -httpMethod ENABLED -httpUserAgent
     ENABLED -httpContentType ENABLED
add analytics profile tcp_profile -collectors COE -type tcpinsight
bind lb vserver <WEB-VSERVER> -analyticsProfile web_profile
bind lb vserver <WEB-VSERVER> -analyticsProfile tcp_profile
 # To enable metrics push to prometheus
add service metrichost_SVC <IP> HTTP <PORT>
set analyticsprofile ns_analytics_time_series_profile -collectors
    metrichost_SVC -metrics ENABLED -outputMode prometheus
```

For information on troubleshooting related to NetScaler Observability Exporter, see NetScaler CPX troubleshooting.

Support for exporting transactions in the JSON format from NetScaler Observability Exporter to Kafka

You can now export transactions from NetScaler Observability Exporter to Kafka in the JSON format apart from the AVRO format.

A new parameter `DataFormat` is introduced in the Kafka deployment ConfigMap to support transactions in the JSON format.

This parameter can accept AVRO and JSON values. For allowing JSON based transactions, set the value of

`DataFormat` as JSON in the

coe-kafka.yaml file. The default value is AVRO.

The following example shows the YAML file with the data format configured as JSON.

```
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: coe-config-kafka
    data:
      lstreamd_default.conf: |
        {

            "Endpoints": {

              "KAFKA": {

                "DataFormat": "JSON",
                "ServerUrl": "X.X.X.X:9092,Y.Y.Y.Y:9092", #Specify the
                    bootstrap Kafka broker addresses (separated by comma
                     in case multiple bootstrap brokers are to be
                    configured)
```



```
                "KafkaTopic": "HTTP", #Specify the desired kafka topic
                "RecordType": {

                    "HTTP": "all",
                    "TCP": "all",
                    "SWG": "all",
                    "VPN": "all",
                    "NGS": "all",
                    "ICA": "all",
                    "APPFW": "none",
                    "BOT": "none",
                    "VIDEOOPT": "none",
                    "BURST_CQA": "none",
                    "SLA": "none",
                    "MONGO": "none"
                 }
,
                "TimeSeries": {

                    "EVENTS": "yes",
                    "AUDITLOGS": "yes"
                 }

               }

            }

          }
```

NetScaler Observability Exporter with Splunk Enterprise as endpoint

September 27, 2025

NetScaler Observability Exporter is a container that collects metrics and transactions from NetScaler and sends the data to various endpoints. NetScaler Observability Exporter supports Splunk Enterprise as an endpoint.

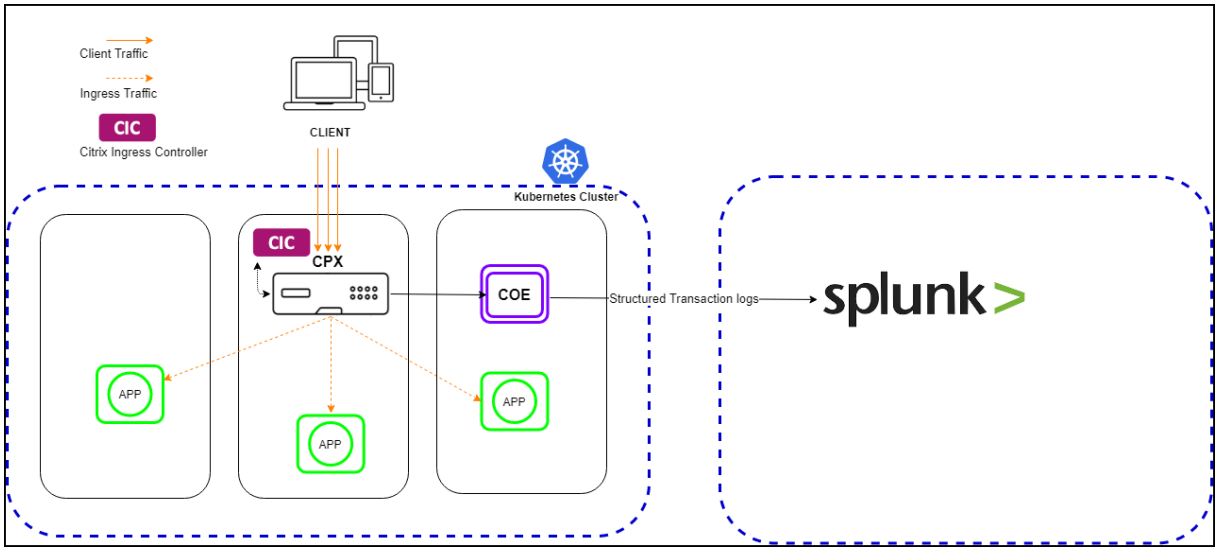
Splunk Enterprise is a data platform for searching, monitoring, and analyzing machine-generated big data. Splunk Enterprise captures indexes and correlates real-time data in a repository from which it can generate reports, graphs, dashboards, and visualizations.

You can add Splunk Enterprise as an endpoint to receive audit logs, events, and transactions from NetScaler for analysis. Splunk Enterprise provides a graphical representation of these data. You can enable or disable the type of transactions, events, and audit logs which are to be sent to Splunk Enterprise.

Deploy NetScaler Observability Exporter

You can deploy NetScaler Observability Exporter using the YAML file. Based on your NetScaler deployment, you can deploy NetScaler Observability Exporter either outside or inside Kubernetes clusters. You can deploy NetScaler Observability Exporter as a pod inside the Kubernetes cluster or on the NetScaler MPX or VPX appliance outside the cluster.

The following diagram illustrates a NetScaler as an Ingress Gateway with the NetScaler Ingress Controller and NetScaler Observability Exporter as sidecars. NetScaler Observability Exporter sends NetScaler application metrics and transaction data to Splunk Enterprise. Splunk Enterprise provides a graphical representation of the data.



Prerequisites

- Ensure that you have a Kubernetes cluster with `kube-dns` or `CoreDNS` addon enabled.

> **Note:**
>
> In the following procedure, the YAML file is used to deploy NetScaler Observability Exporter in the Kubernetes *default* namespace. If you want to deploy in a private namespace other than the *default*, edit the YAML file to specify the namespace.

Perform the following steps to deploy NetScaler Observability Exporter:

> **Note:**
>
> If you have a pre-deployed web application, skip the steps 1 and 2.

1. Create a secret ingress.crt and key ingress.key using your own certificate and key.

   In this example, a secret, called *ing* in the default namespace, is created.


```
kubectl create secret tls ing --cert=ingress.crt --key=ingress.
    key
```

2. Access the YAML file from webserver-splunk.yaml to deploy the application.

```
kubectl create -f webserver-splunk.yaml
```

3. Define the specific parameters that you must import by specifying it in the ingress annotations of the application's YAML file, using the smart annotations in the ingress.

```
ingress.citrix.com/analyticsprofile: '{
"webinsight": {
"httpurl":"ENABLED", "httpuseragent":"ENABLED", "httpHost":"
    ENABLED","httpMethod":"ENABLED","httpContentType":"ENABLED" }
}
'
```

> **Note:**
>
> The parameters are predefined in the `webserver-splunk.yaml` file.

For more information about Annotations, see Ingress annotations documentation.

Deploy NetScaler CPX with the NetScaler Observability Exporter support

You can deploy NetScaler CPX as a side car with the NetScaler Observability Exporter support enabled along with NetScaler Ingress Controller. You can modify the NetScaler CPX YAML file `cpx-ingress-splunk.yaml` to include the configuration information that is required for the NetScaler Observability Exporter support.

The following is a sample application deployment procedure.

1. Download the cpx-ingress-splunk.yaml and cic-configmap.yaml file.
2. Create a ConfigMap with the required key-value pairs and deploy the ConfigMap. You can use the `cic-configmap.yaml` file that is available, for the specific endpoint, in the directory.
3. Modify NetScaler CPX related parameters, as required.
4. Edit the `cic-configmap.yaml` file and specify the following variables for NetScaler Observability Exporter in the `NS_ANALYTICS_CONFIG` endpoint configuration.

```
server: 'coe-splunk.default.svc.cluster.local' # COE service
    FQDN
```

> **Note:**
>
> If you have used a namespace other than *default*, change `coe-splunk.default.svc`

> `.cluster.local to` to `coe-splunk.<desired-namespace>.svc.cluster.local`. If NetScaler is outside the Kubernetes cluster, then you must specify IP address and nodport address of NetScaler Observability Exporter.

5. Deploy NetScaler CPX with the NetScaler Observability Exporter support using the following commands:

```
kubectl create -f cpx-ingress-splunk.yaml
kubectl create -f cic-configmap.yaml
```

Deploy NetScaler Observability Exporter using the YAML file

You can deploy NetScaler Observability Exporter using the YAML file. Download the YAML file from coe-splunk.yaml. Ensure to specify the Splunk server address for the right namespace by editing the `coe-splunk.yaml` file.

Following is an example of how to specify the `ServerUrl` in the `lstreamd_default.conf` section in the `coe-splunk.yaml` file . Here, `ServerUrl` means the address of the Splunk server.

```
          lstreamd\_default.conf: |
          {

            "Endpoints": {

              "SPLUNK": {

                "ServerUrl": "http://10.102.34.155:8088",
                "AuthToken": "",
                "Index": "",
                "RecordType": {

                  "HTTP": "all",
                  "TCP": "all",
                  "SWG": "all",
                  "VPN": "all",
                  "NGS": "all",
                  "ICA®": "all",
                  "APPFW": "none",
                  "BOT": "all",
                  "VIDEOOPT": "none",
                  "BURST_CQA": "none",
                  "SLA": "none",
                  "MONGO": "none"
                 }
,
                "TimeSeries": {

                  "EVENTS": "yes",
                  "AUDITLOGS": "yes"
```



```
                    }
  ,
                   "ProcessAlways": "no",
                   "ProcessYieldTimeOut": "500",
                   "MaxConnections": "512",
                   "JsonFileDump": "no"
                  }

                }

              }
```

> **Note:**
>
> While deploying NetScaler Observability Exporter using the YAML file, along with the Splunk server address, you can provide the `Index` name to which the data to be sent in Splunk Enterprise. By default, this `IndexPrefix` option is empty and the data is uploaded to the default index, that is `main`, in Splunk Enterprise.

To deploy NetScaler Observability Exporter using the Kubernetes YAML, run the following command in the Splunk Enterprise endpoint:

```
kubectl create -f coe-splunk.yaml
```

> **Note:**
>
> Modify the YAML file for NetScaler Observability Exporter if you have a custom namespace.

Verify the NetScaler Observability Exporter deployment

You can verify the deployment after deploying NetScaler Observability Exporter, web application, NetScaler CPX, and NetScaler Ingress Controller.

To verify the deployment, perform the following steps:

1. Verify the deployment using the following command:

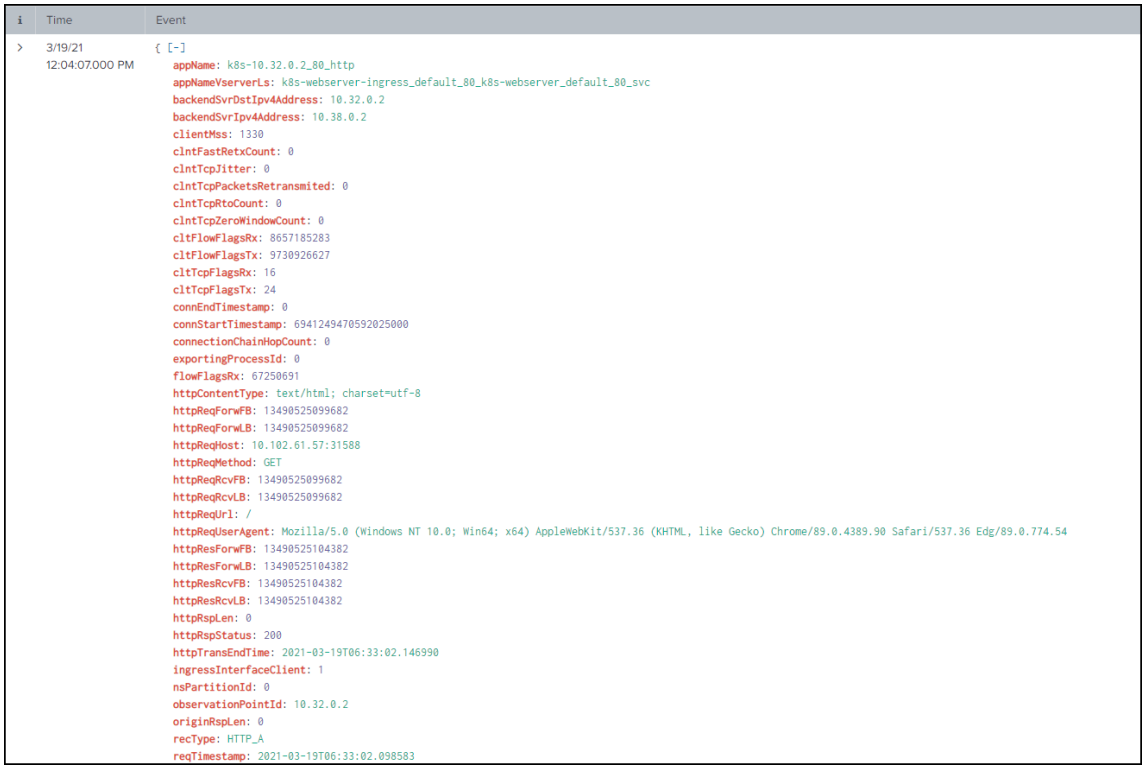
   ```
   kubectl get deployment,pods,svc -o wide
   ```

2. Access the application using a browser with the URL.
   For example:

   ```
   https://kubernetes-node-IP:cpx-ingress-splunk nodeport/
   ```

3. Access the Splunk server using a browser with the URL.
   For example:

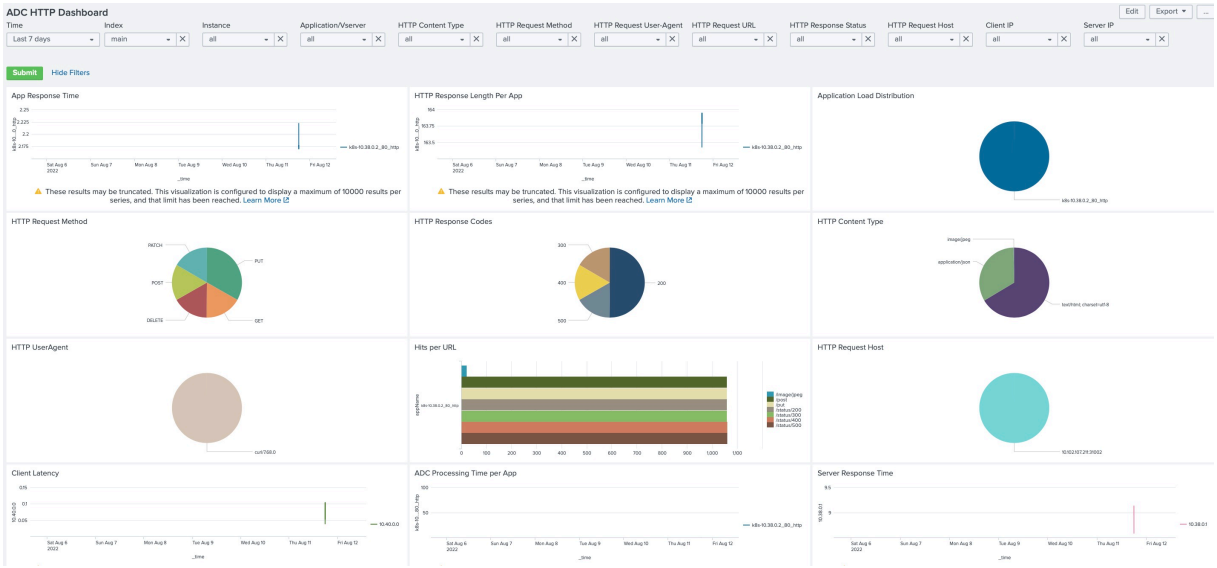
   ```
   https://splunk-node-IP:splunk nodeport/
   ```

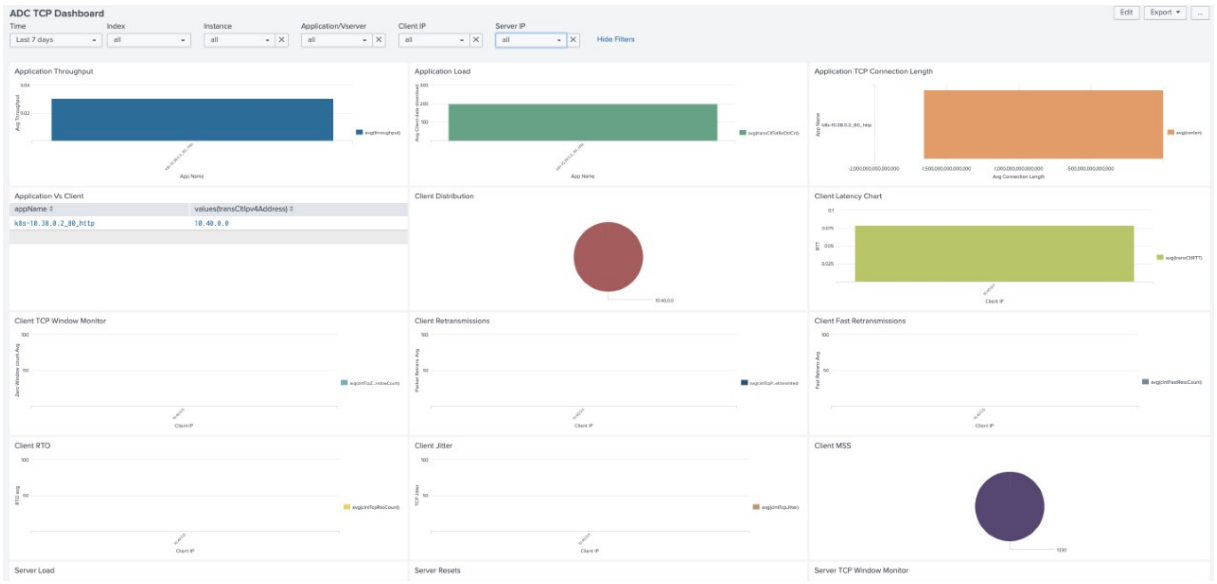
Import pre-built dashboards for Splunk

You can import pre-built Splunk dashboards provided by NetScaler. The JSON files for importing the dashboards are available at the GitHub repository. These dashboards provide you the option to filter the transactions based on parameters such as an instance IP address, application name, or client and server IP address and so on.

Following is a sample HTTP dashboard. This dashboard shows data such as HTTP header-based charts, transactional latency, response type distribution, and so on.



Following is a sample TCP dashboard for Splunk. This dashboard shows data such as bandwidth distribution for each application, TCP Jitter, client and server RTT, and so on.



NetScaler Observability Exporter troubleshooting

September 27, 2025

This document explains how to troubleshoot issues that you may encounter while using NetScaler Observability Exporter.

- How do I verify that NetScaler sends application data logs to NetScaler Observability Exporter?

Run the following command to verify that NetScaler sends application data logs to NetScaler Observability Exporter:

```
nsconmsg -g lstream_tot_trans_written -d current
```

The counter value indicates that the number of application transactions (for example, HTTP transactions) which have been sent to NetScaler Observability Exporter.

If the application traffic rate (for example, HTTP req/sec) that is sent to NetScaler Observability Exporter is not equal to `lstream_tot_trans_written`, you can verify the same using the following command:

```
nsconmsg -g nslstream_err_ulf_data_not_sendable -d current
```

The counter value indicates that NetScaler cannot send the data to NetScaler Observability Exporter due to network congestion, unavailability of network bandwidth, and so on and the data is stored in the available buffers.

Information about various transaction data and individual fields, and their datatype are available in the following location on the NetScaler:

```
shell
/netscaler/appflow/ns_ipfix.yaml
```

To verify the current record type exported from NetScaler to NetScaler Observability Exporter, use the following command:

```
nsconmsg -g appflow_tmpl -d current
```

Location of metrics data export logs to NetScaler for time series data:

```
/var/nslog/metrics_prom.log
```

To verify kafka related counters, run the following command:

```
kubectl exec -it <cpx-pod-name> [-c <cpx-container-name>] [-n <
    namespace-name>] -- bash

tail -f /var/ulflog/counters/lstrmd\_counters\_codes.log | grep -
    iE \"\(http\_reqs\_done|kafka)\"
```

Find the logs in the following location to verify that the NetScaler Observability Exporter configuration is applied correctly:

```
vi /var/logproxy/lstreamd/conf/lstreamd.conf
```

If NetScaler Observability Exporter fails, you can collect logs available at the following location and contact NetScaler support.

```
/var/crash/ (Loation of the coredump files, if any.)
```



```
/var/ulflog/ (Location of `libulfd` logs and counter details.)
/var/log  (Location of console logs, lstreamd logs and so on.)
```

For information on NetScaler CPX related troubleshooting, see NetScaler CPX Troubleshooting.

Description of configuration parameters

September 27, 2025

This topic contains descriptions of the `lstreamd_default.conf` file parameters. The `lstreamd_default.conf` parameters are used for endpoint specific configurations.

- `ServerUrl`: Specifies the address of the server.

  The protocol can be Kafka for Apache Kafka, HTTP or HTTPs for Splunk and ElasticSearch, and HTTP for Zipkin.

  The following are examples of how to specify the server URL for different endpoints:

  Kafka:

  ```
      kafka-server:9992
      1.2.3.4:10000  Splunk:

      http://splunk-server:80
      https://splunk-server:443
      http://1.2.3.4:1000
      https://5.6.7.8:2000
  ```

  ElasticSearch:

  ```
      http://elastic-server:80
      https://elastic-server:443
      http://9.8.7.1:80
      https://1.2.3.5:3000  Zipkin:

      http://zipkin-server:80
      http://1.2.3.4:80
  ```

- `KafkaTopic`:

  Specifies the topic on a Kafka cluster for sending the transaction records. The default value is HTTP.

- `DataFormat`:

  Specifies the format of the data sent over to Kafka. The values can be either JSON or AVRO. The default data format is AVRO.

- `MaxConnections`:

  Specifies the maximum parallel TCP connections to an endpoint.
  The default value is 64.

- `FileSizeMax`:

  For the Kafka endpoint, AVRO files are created and stored on the disk before they get pushed to the endpoint.
  This parameter specifies the size of each such file in Kibibyte (KiB). A file can contain multiple transaction records. The default value is 48 KiB.

- `RecordType`:

  Specifies the types of records that you want to export:

  - HTTP
  - TCP
  - SWG
  - VPN
  - ICA®
  - APPFW
  - BOT
  - VIDEOPT
  - BURST_CQA
  - SLA
  - MONGO
  - MQTT

  NetScaler Observability Exporter allows filtering of transaction records of various insights.
  By default, none of the records are exported.
  You must set these fields appropriately to export the required records.
  Examples :

  ```
  "TCP": "all",
  "SWG": "none",
  "APPFW": "all"
  ```

- `EVENTS`:

  NetScaler Observability Exporter allows exporting time series (events, and audit logs) to Splunk and Kafka.
  Set this field to `yes` to allow exporting events.
  The default value is `no`.

- `AUDITLOGS`:

You can export audit logs to Splunk and Kafka.
Set this field to `yes` to allow exporting audit logs.
The default value is `no`.

- `ConnectionPoolSize`:

  Alters the size of connection pools for Splunk.`ConnectionPoolSize` and `MaxConnections` might be used to control the rate at which data is exported (to the endpoint).

- `ElkMaxSendBuffersPerSec`:

  The maximum rate at which the data is exported to ElasticSearch.
  An ELK JSON Buffer is 32 KiB in size.
  This field configures the maximum number of ELK buffers exported every second.
  The default value is 64.

- `transRateLimitEnabled`:

  Sometimes, the incoming traffic may increase and Citrix® Observability Exporter may not be able to scale up to it.
  In such cases, export to JSON endpoints like Splunk, ElasticSearch, and Zipkin over HTTP/HTTPS might become a bottleneck.
  The memory would keep growing uncontrollably until Citrix Observability Exporter terminates.
  To avoid such scenarios, rate-limiting can be configured for JSON based endpoints including Kafka. The impact on Kafka is low as Kafka is an efficient protocol.
  Set this field to `yes` to enable rate-limiting for JSON based end points.
  The default value is `no`.

- `transQueueLimit`:

  Specifies the number of JSON buffers that can be accumulated before Citrix Observability Exporter starts discarding them.
  For Zipkin, one JSON buffer is about 64 KiB and a limit of 1000 means approximately 64 MB of JSON data.
  For Splunk and ElasticSearch, one JSON buffer is about 32 KiB and a limit of 1000 means approximately 32 MB of data.
  The default value is 1024.

- `transRateLimitWindow`:

  Specifies the recalculation window in seconds and the value must be greater than zero.
  The lower the window size, the more effective is the rate-limiting, but specifying low values may cause slight CPU overhead.
  The default value is five seconds.

- `AuthToken`:

You can use the auth token to perform the token-based authentication for Splunk.
It can also be used for password-based authentication for ElasticSearch.

Examples :

```
SPLUNK

    "AuthToken": "xxxxxxxx-xxxx-xxxx-ad58-1ce9bdeee09a"

ELASTICSEARCH

    "AuthToken": "xxxxxxxxxxxxeXBhc3MxMjM="
```

- `Index`:

  The Splunk index where the processed data is stored.
  The default value is "". That means the default index.

- `IndexPrefix`:

  Specifies the index prefix used for ElasticSearch. ElasticSearch allows you to create indexes as necessary through its APIs.
  Kibana allows the creation of index patterns and to facilitate that, this field is used.
  All index names follow this prefix followed by the time of creation of the index. The time or time format is based on the `IndexInterval`.

- `IndexInterval`:

  The interval at which the ElasticSearch indexes are rotated, following the index pattern.
  You can configure the interval as one of the following values:

  - hourly
  - 12 hours
  - daily
  - weekly
  - 2 weeks
  - monthly
  - 6 months
  - yearly

  The default value is "". That means, the index never rotates.

Additional information

Following are the guidelines while configuring the `lstreamd_default.conf` file parameters.

- Zipkin is supported in parallel to Splunk, ElasticSearch, or Kafka.

- You do not need to configure Prometheus as it is pull based. NetScaler Observability Exporter exports NetScaler metrics and its own metrics to prometheus.

  Port 5563 of the container can be scraped using insecure HTTP at the path '/metrics'.

  For example:

  ```
  1  http://coe-fqdn:5563/metrics
  ```

  Prometheus is always `ON` and metrics can be exported to it in parallel to transactions, audit logs, and events.

- Currently, you can only export time series like audit logs and events to Splunk and Kafka, but in parallel to transactions and metrics.

- You must not configure multiple endpoints of the same type in the `lstreamd_default.conf` file for one NetScaler Observability Exporter. For example, it is not possible to configure two Splunk instances, or two Kafka instances, or two ElasticSearch instances, or one Splunk and one ElasticSearch, and so on.
  For Zipkin, although you can configure it in parallel to Splunk and ElasticSearch, you may not configure multiple instances of Zipkin. For example, it is not possible to have two Zipkin instances in parallel.

- You can ignore the fields that are marked as optional as some of them may have predefined default values.

- The JSON parser used for `lstreamd_default.conf` is case-sensitive and also ensure that you do not have extra or missing commas, or anything that may make the JSON format invalid.

- Some of the `lstreamd_default.conf` file parameters are not listed in this document. Those parameters that are not listed are internal and are not meant to be altered. They have predefined default values.

Support for container logging

September 27, 2025

Now, you can enable logging on NetScaler Observability Exporter according to different severity levels. These logs help in getting information about endpoint specific configuration.

The following logging severity levels are supported and the default value is `INFO`.

- `NONE` : None of the messages are logged.
- `FATAL` : Only fatal messages are logged.
- `ERROR`: Only fatal messages and error messages are logged.

- `INFO`: Only fatal, error, and informational messages are logged.
- `DEBUG`: Only fatal, error, informational, and debug messages are logged.

For Kubernetes YAML based deployments the default value is NONE. But, for Helm and OpenShift operator deployments of NetScaler Observability Exporter logging is enabled by default and set as INFO.

You can configure logging using the environment variable `NSOE_LOG_LEVEL` while deploying NetScaler Observability Exporter for each endpoint.

The following example shows how to configure the log level in the NetScaler Observability Exporter deployment YAML:

```
      env:
        - name: NSOE_LOG_LEVEL
          value: "INFO"
```

Log descriptions

The following are the types of logs that you can get:

Error logs

Log	Severity	Description
Endpoints Missing - Invalid Config Format	Error	Field “Endpoints”is missing in lstreamd.conf.
Splunk Auth Token missing	Error	AuthToken missing for Splunk HEC.
Unknown Record Type - Invalid Configuration	Error	Unrecognized record.
Record Type Option - Invalid Configuration	Error	Unrecognized argument. Only all/none/anomalous accepted.
Unknown Timeseries - Invalid Configuration	Error	Only AUDITLOGS/EVENTS accepted.
Setting Rate Limit Enabled to false - Invalid Rate Limit Option - Invalid Configuration	Error	Invalid configuration for rate limiting.
Anamalous Config Missing	Error	Anomalous config not found.
The configuration file does not exist	Error	OE tried to read `lstreamd.conf` but did not find it.

Log	Severity	Description
Either the name of the endpoint or the address of the server is invalid (empty)	Error	The endpoint kind such as SPLUNK,KAFKA,ES,ZIPKIN is empty or its address (ServerUrl) is empty.
Invalid value- <val> configured Timeseries/<timeseries-type> export to No for <endpoint-type>	Error	Invalid value for the given type of timeseries (only “yes”/ “no” are allowed)- defaulting to “no” .
Failed to allocate Packet Engine context for a client	Error	ULFD consumer thread failed to allocate the Packet Engine context (Per NetScaler/ Packet Engine context).
Failed to allocate memory to hold the Logstream buffer- perhaps we ran of memory	Error	ULFD consumer thread ran OOM and could not allocate ~ 8 KiB memory required to hold the received Logstream Buffer.
Failed to initialize the Processor for the client <client-id>	Error	Failed to initialize the Processor for the client.

Info logs

Log	Severity	Description
lstreamd Process Initiating	Info	NSOE started.
lstreamd.conf not found - creating new and switching to agent/ adm on-prem mode	Info	lstreamd.conf was not found.
ELK Health Status DOWN	Info	ElasticSearch is down.
ELK Health Status UP	Info	ElasticSearch is up.
Splunk Health Status DOWN	Info	Splunk is down.
Splunk Health Status UP	Info	Splunk is up.
Deleting NetScaler IP: <nsip> Core: <core-id> Partition: <partition-id>	Info	Disconnected from a NetScaler.

Log	Severity	Description
Adding new NetScaler IP: <nsip> Core: <core-id> Partition: <partition-id>	Info	Connected to a new NetScaler.
New NetScaler Allocation	Info	Connected to a new NetScaler.
New NetScaler Initiating	Info	Connected to a new NetScaler.
JSON data format set for KAFKA	Info	JSON will be exported to Kafka.
AVRO data format set for Kafka	Info	AVRO will be exported to Kafka.
Enabling Traces for Zipkin	Info	Zipkin was configured.
Set the maximum number of connections to the endpoint <type> as <maxSockets>	Info	MaxConnections was either parsed or defaulted to the printed value for the printed endpoint.
Enabling Kafka Exporter	Info	Kafka was configured in lstreamd.conf and hence enabled.
Configured Kafka broker list : <brokers>	Info	Applied the configured Kafka Broker list (from the serverUrl).
Configured Kafka topic: <topic>	Info	KafkaTopic was either parsed or defaulted to the printed value.
Configured <endpoint-type> to export <all / anomalous / none> <rec-type> records	Info	Only the printed records of type <rec-type: ex HTTP_A/TCP_A/...> were configured to be exported for the printed endpoint.
Configured Timeseries/<timeseries-type> to <Yes/No> for <endpoint-type>	Info	Auditlogs/Events/Metrics were enabled/disabled for the given endpoint.
Configured Processor Yield Timeout to <val>	Info	Yield time of the processor threads was either configured or defaulted.
Prometheus Mode Enabled- Prometheus can now scrape metrics at the rest port	Info	Prometheus is authorized to perform 'GET /metrics'at the rest port.

Log	Severity	Description
Received a new client connection	Info	Received a connection from a NetScaler (Packet Engine.)
Received a disconnect from a client	Info	Received a disconnect from a NetScaler (Packet Engine).
Received a reset from a client	Info	Received a disconnect (reset) from a NetScaler (Packet Engine).
JSON transaction rate limiting enabled	Info	Enabled rate-limiting for transactions to JSON-based endpoints viz. Splunk, ElasticSearch, Zipkin, and Kafka (JSON).
Set JSON transaction rate limit to <num> Logstream buffers per second	Info	Per second rate-limit for logstream buffers.
Set JSON transaction Export Queue limit to <num> JSON buffers	Info	Limit on the JSON export queue - Beyond this threshold, the Exporter will start dropping JSON buffers.
Set JSON transaction rate limit window to <num> seconds	Info	Window of rate-limiting transactions to JSON-based endpoints. Lower values can capture spikes.

Warning logs

Log	Severity	Description
Prometheus Mode Disabled-Prometheus will not be authorized to scrape metrics at the rest port	Warning	Prometheus is unauthorized to perform 'GET /metrics'at the rest port.

Debug logs

Log	Severity	Description
Spawned Processor #1 with Thread Index: 0 , Thread ID: <id>	Debug	Created and started the first processing thread.
Spawned ULFD Receiver with Thread Index: 1 , Thread ID: <id>	Debug	Created and started the ULFD consumer thread.
Spawned Exporter with Thread Index: 2 , Thread ID: <id>	Debug	Created and started the export thread.
Spawned Processor #2 with Thread Index: 3 , Thread ID: <id>	Debug	Created and started the second processing thread.
ULFD Receiver received the signal number- <signo>	Debug	ULFD consumer thread received a signal (usually from NSULFD) - usually SIGUSR2 is used to induce counter file rotation.
ULFD Receiver received timeout event	Debug	ULFD consumer thread received a timeout event (received from NSULFD every second) - used for cleaning up CLM meta records every minute.
ULFD Receiver received an unhandleable event - ignoring it	Debug	ULFD consumer thread received an unknown event, thus ignored it.
ULFD Receiver received a Logstream Buffer	Debug	ULFD consumer thread received a Data Buffer (Logstream).
The received Logstream Buffer is corrupted - unable to parse it	Debug	ULFD consumer thread received a corrupted Data Buffer (Logstream).
Parsed the received Logstream Buffer: client-id=<>, core-id=<>, partition-id=<>, namespace=<>	Debug	ULFD consumer thread parsed the received Data Buffer.
The rate-limiter decided to drop the Logstream buffer	Debug	Buffer drop because of JSON transaction rate-limiting.
The rate-limiter decided to accept the Logstream buffer	Debug	Buffer accepted by JSON transaction rate-limiter.

Log	Severity	Description
Logstream buffer dropped because the Processor for this client is not emptying the queue fast enough	Debug	Buffer drop because the Processing thread associated with this NetScaler (Packet Engine) is not consuming the buffers fast enough.
Logstream buffer dropped because the configuration does not allow accepting Non-Anomalous buffers	Debug	Buffer drop because `Lstreamd` has been configured to process anomalous transactions only.
Relinquishing the Logstream buffer because we are done with it	Debug	ULFD consumer thread is done with the received Logstream Buffer- thus relinquishing the shared memory.
ULFD Receiver dropped a Logstream buffer- either it was corrupt or we ran out of memory	Debug	ULFD consumer thread dropped the Logstream Buffer because it was corrupt or it ran OOM.
Initialized the <AVRO/ JSON> Processor for the client `<client-id>`	Debug	Initialized the printed Processor for the client (done the first time the NetScaler / Packet Engine connects).
Pushed `<num-pushed>` Logstream Buffer(s) toward the Processor	Debug	Pushed the printed number of Logstream Buffers to the associated Processing Thread's Queue.
Timer fired for the JSON Processor with `thread-id=<thread-id>`, `client-id=<client-id>`	Debug	JSON Processing Thread's Process Timer Timed Out.
JSON Processor with thread-id=`<thread-id>`, `client-id=<client-id>` yielded the CPU	Debug	JSON Processing Thread yielded the CPU.
JSON Processing Started	Debug	Started converting the Logstream Buffers piled in the queue to JSON.

Log	Severity	Description
Processing Logstream buffer started	Debug	Started processing a logstream buffer.
Dropping a transaction because the JSON Endpoint is not healthy (set "ProcessAlways": "yes"to skip health checks)	Debug	The transaction was dropped because either the JSON endpoint- Splunk/ ElasticSearch was unhealthy or because Kafka (JSON) was not enabled
Transaction dropped: Cannot create JSON Object Queue for the Unconfigured Record Type - <rec-type>	Debug	The transaction was dropped because it belonged to a Record Type that was not configured (or configured to none).
Creating JSON Object Queue for the Record Type - <rec-type>	Debug	Created JSON Object Queue for the configured Record Type (used to hold JSON Buffers until they are pushed to the Exporter).
Transaction dropped: Cannot create JSON Object Queue for Unknown Protocol ID - <protocol-id>	Debug	The transaction was dropped because it belonged to an Unknown Record Type / Protocol ID.
Created JSON Object Queue (protocol-id=<protocol-id>)	Debug	Created the JSON Object Queue for the printed Record Type / Protocol ID.
Set Index to <index-string> for the JSON Object Queue	Debug	Set the Index for JSON Object Queue (valid for ElasticSearch/ Splunk endpoints that use indices).
Transaction dropped: Unable to create JSON Object Queue- perhaps ran out of memory	Debug	The transaction was dropped because the JSON Object Queue could not be created due to lack of memory.
Transaction processing started	Debug	Started the conversion of transaction to JSON.

Log	Severity	Description
Anomaly: The field is indexed but could not be found in the table- code=<id>	Debug	The field is indexed but could not be found in the local meta table.
Generated record type-<rec-type> for the transaction	Debug	Generated a JSON record of the printed type for the transaction
Transaction processing finished	Debug	Finished the conversion of transaction to JSON.
Failed to process the Logstream buffer because the AppFlow codename bearing meta records have not arrived yet from this client	Debug	Dropped the transaction because the Code Maps have not arrived from NetScaler (Packet Engine).
Finished processing the Logstream buffer	Debug	Finished processing the logstream buffer.
JSON processing finished	Debug	Finished converting the logstream buffers piled in the queue to JSON
Pushed the converted JSON transactions to the Processed Queue	Debug	Pushed the converted JSON transactions to the Processed Queue (common queue where the processed JSONs of all Record Types wait before being handed over to the exporter).
Pushed all the piled up JSONs in the Processed Queue to the Exporter	Debug	Handed over all the JSON transactions waiting in the Processed Queue to the Exporter.
Started exporting piled up JSON buffers to Kafka Client	Debug	Started the export of the piled up JSON buffers in the export queue to Kafka Client.
Finished exporting piled up JSON buffers to Kafka Client	Debug	Finished the export of the piled up JSON buffers in the export queue to Kafka Client.

Log	Severity	Description
Pushing the JSON buffer back to the Export Queue because of failure in exporting it to the Kafka Client	Debug	Failed to export the JSON buffer to the Kafka client, hence adding it back to the Export Queue.
The Exporter dropped <numDropped> JSON Buffers because the Export Queue hit its configured limit	Debug	Drop from the Export Queue because of Rate Limiting.
Failed to export the JSON Buffer to the Kafka Client because the Kafka Topic Manager has not been created yet	Debug	Could not export JSON Buffer to Kafka Client because the Kafka Topic Manager is not created yet.
Created Kafka Topic Manager	Debug	Constructed the Kafka Topic Manager (used to maintain Kafka Client's state).
Kafka Topic Manager is creating the topic- <topic>	Debug	Kafka Topic Manager is attempting to create the configured topic.
Kafka Topic Manager failed to create the topic- perhaps ran out of memory	Debug	Kafka Topic Manager ran OOM thus failed to create the configured topic.
Kafka Topic Manager created the the topic	Debug	Kafka Topic Manager created the configured topic
Unable to validate the configured topic- failed to acquire a meta connection to the Kafka client, perhaps ran out of memory	Debug	Kafka Topic Manager failed to validate the topic- perhaps ran OOM.
Meta Request sent to the Kafka client for topic validation	Debug	Kafka Topic Manager successfully issued Meta Request to the Kafka Client for Topic Validation.
Failed to create meta connection to the Kafka Client- perhaps ran out of memory	Debug	Failed to create meta connection to the Kafka Client - perhaps ran OOM.

Log	Severity	Description
Created meta connection to the Kafka client-connection-id=<id>	Debug	Created meta connection to the Kafka Client for the configured topic.
Reused existing meta connection to the Kafka Client-connection-id=<id>	Debug	Reused existing meta connection to the Kafka Client.
Cleaned up the stale meta connection to the Kafka Client-connection-id=<id>	Debug	Removed the erroneous meta connection to the Kafka Client from the Meta Connection Pool, for the configured topic.
Cleaning up the stale Produce Connection to the Kafka Client-connection-id=<id>	Debug	Removed the erroneous produce connection to the Kafka Client from the Produce Connection Pool.
Cleaned up the Produce Connection to the Kafka client-connection-id=<id>	Debug	Removed a valid produce connection to the Kafka Client from the Produce Connection Pool.
Reused existing Produce Connection to the Kafka Client-connection-id=<id>	Debug	Reused existing produce connection to the Kafka Client
Cleaning up Produce Connections for partition=<id>, topic=<topic>	Debug	Started the cleanup of produce connections to the Kafka Client.
Forcefully cleaned active Produce Connections	Debug	Forcefully cleaned active produce connections to the Kafka Client during partition cleanup.
Cleaned inactive Produce Connection	Debug	Cleaned inactive produce connection to the Kafka Client during partition cleanup.
Failed to push Kafka Produce Connection to the Stale Connection Pool- unable to find it in the Produce Connection Pool!-connection-id=<id>	Debug	Could not find the Kafka Produce Connection in the Kafka Produce Connection Pool.

Log	Severity	Description
Pushed Kafka Produce Connection to the Stale Connection Pool- connection-id=<id>	Debug	Pushed the Kafka Produce Connection to the Kafka Erroneous Connection Queue.
Failed to push Kafka produce connection to the Free Connection Pool- unable to find it in the Produce Connection Pool!- connection-id=<id>	Debug	Could not find the Kafka Produce Connection in the Kafka Produce Connection Pool.
Pushed Kafka Produce Connection to the Free Connection Pool- connection-id=<id>	Debug	Pushed the Kafka Produce Connection to the Kafka Erroneous Connection Queue.
Cleaned up Produce Connections for the partition	Debug	Cleaned up produce connections for the partition.
Begin cleaning up disabled partitions for the topic <topic>	Debug	Started cleaning up disabled partitions.
Cleaned up disabled partition- id=<id>	Debug	Removed the disabled partition from the Disabled Partition Queue.
Finished cleaning up disabled partitions	Debug	Finished cleaning up partitions for the topic.
Failed to retry failed JSON files to Kafka- the topic state is invalid	Debug	Failed to retry failes JSON files to Kafka because the topic state is not valid.
Anomaly: Failed to retry the Kafka JSON File	Debug	Failed to retry failes JSON files to Kafka because of some anomaly.
Retried a Kafka JSON File	Debug	Pushed the Kafka JSON file to the Kafka Client for retry.
Begin cleaning up the topics	Debug	Started cleaning up the topics (done every 7 seconds).
Cleaned up the topic- <topic>	Debug	Cleaned up the printed topic.
Finished cleaning up the topics	Debug	Finished cleanup up the topics.

Log	Severity	Description
Failed to export the JSON Buffer to the Kafka Client because the Kafka Topic has not been created yet by the Kafka Topic Manager	Debug	Could not export the JSON Buffer to Kafka Client because the Kafka Topic Manager has not yet created the topic.
Pushed the Kafka JSON File to the partition- <pid> because the topic is in valid state	Debug	Exported the Kafka JSON file to the Kafka Client.
Failed to push the Kafka JSON File to the Kafka Client because the topic is in invalid state	Debug	Could not export the JSON file to the Kafka Client because the topic is in invalid state.
Pushed the Kafka JSON Buffer to the partition- <pid> because the topic is in valid state	Debug	Exported the Kafka JSON Buffer to the Kafka Client.
Failed to push the Kafka JSON Buffer to the Kafka Client because the topic is in invalid state	Debug	Could not export the JSON Buffer to the Kafka Client because the topic is in invalid state.
Kafka JSON Buffer death because of failure to send the Produce Request to the Kafka Client- perhaps ran out of memory	Debug	Permanently lost Kafka JSON buffer because of lack of memory.
Kafka JSON Buffer death because of lack of Produce Connections- perhaps ran out of memory	Debug	Permanently lost Kafka JSON buffer because of lack of memory.
Pushed Kafka JSON Buffer to the Kafka Client	Debug	Exported the Kafka JSON buffer to the Kafka Client.
Kafka JSON File death because of failure to send the Produce Request to the Kafka Client- perhaps ran out of memory	Debug	Permanently lost Kafka JSON file because of lack of memory.
Kafka JSON File death because of lack of Produce Connections- perhaps ran out of memory	Debug	Permanently lost Kafka JSON file because of lack of memory.

Log	Severity	Description
Pushed Kafka JSON File to the Kafka Client	Debug	Exported the Kafka JSON file to the Kafka Client.
Kafka client rejected the pushed JSON buffer because it was empty	Debug	Kafka Client rejected the pushed JSON buffer because it was empty.
Kafka client failed to create a produce request for the pushed JSON buffer- perhaps ran out of memory	Debug	Kafka Client ran OOM while creating the produce request.
Kafka client failed to create Kafka File for the pushed JSON buffer- perhaps ran out of memory	Debug	Kafka Client ran OOM while creating the Kafka File.
Kafka Client dispatched JSON Buffer to Kafka	Debug	Kafka Client dispatched JSON buffer to Kafka.
Kafka client failed to create a produce request for the pushed JSON File- perhaps ran out of memory	Debug	Kafka Client ran OOM while creating the produce request.
Kafka Client dispatched JSON File to Kafka	Debug	Kafka Client dispatched JSON File to Kafka.
Kafka JSON File death-code=<errno>	Debug	Permanently lost kafka file because of the printed error code.
Anomaly: The topic this Kafka Connection belongs to does not exist!	Debug	Anomaly: This Kafka Produce connection belonged to an inexistant topic.
Anomaly: The partition this Kafka Connection belongs to does not exist!	Debug	Anomaly: This Kafka Produce connection belonged to an inexistant topic.
Kafka JSON File death-code=<errno>	Debug	Permanently lost kafka file because of the printed error code.
Anomaly: The file was sent on an inactive Produce Connection	Debug	Anomaly: The file was sent on an inactive produce connection.

Log	Severity	Description
Unsuccessful Kafka Produce Request (JSON) - code=<errno>	Debug	Unsuccessful Export of Kafka (JSON) File.
Anomaly: Kafka Topic Died- JSON File death	Debug	Anomaly: Kafka Topic Died- JSON File death.
Successful Kafka Produce Request- JSON File uploaded	Debug	Uploaded JSON File to Kafka.
Death of Kafka JSON File during retry- the configured retry rate is 0!	Debug	Dropped the new JSON File because KafkaRetryRate is 0.
Death of old Kafka JSON File during retry- Retry Queue limit reached	Debug	Dropped the old JSON File because the Retry Queue was full (upto 32K files are stored).
Pushed new Kafka JSON File to Retry Queue	Debug	Pushed new Kafka JSON File to Retry Queue.
Kafka Client failed to create Topic Metadata Request- perhaps ran out of memory	Debug	Kafka client ran OOM.
Kafka Client dispatched Topic Metadata Request to Kafka	Debug	Kafka Client dispatched Topic Metadata Request to Kafka.
Failed to create Produce Connection to the Kafka Client- perhaps ran out of memory	Debug	Kafka client ran OOM.
Created Produce Connection to the Kafka Client- connection-id=<id>	Debug	Kafka client successfuly created Produce Connection.
Anomaly: The topic this Kafka Meta Connection belongs to does not exist!	Debug	Anomaly: This Kafka Meta connection belonged to an inexistant topic.
Invalid Topic Metadata response received from Kafka- code=<errno>, empty=<true/false>, status=<sent/not sent>	Debug	Topic state invalidated.
The Topic Metadata response received from Kafka bore no topics	Debug	Topic state invalidated.

Log	Severity	Description
Kafka Client failed to create the partition- <id> for the topic-<topic>, perhaps ran out of memory	Debug	Kafka client ran OOM
Kafka Client created partition with leader=<leader>, partition-id=<id>, topic=<topic> : topic state is now validated	Debug	Kafka client created a partition.
Kafka Client failed to find a valid partition for the topic-<topic> invalidating the topic state	Debug	Kafka client failed to find a suitable partition for the topic- thus invalidated its state.
Rare condition: recreating a partition with active Produce Connections to the Kafka client- disabling the older partition and creating a new one- partition-id=<id>, topic=<topic>	Debug	The leader of a partition changed while a produce connection was actively pushing a file to that partition on Kafka- thus disabling the partition and creating a new one in its place.
Kafka Client recreated partition with leader=<leader>, partition-id=<id>, topic=<topic> : topic state is now validated	Debug	Kafka client recreated a partition.
Kafka Client setting partition-<id> as leaderless, topic=<topic>	Debug	Found no leader for the printed topic partition.
Kafka Client setting partition-<id> as having leader, topic=<topic>	Debug	Found leader for the printed topic partition.
No leader exists for any partition of the topic-<topic>, invalidating its state	Debug	Found no leader for the configured topic- invalidating its state.

Log	Severity	Description
Kafka Client received a valid Topic Metadata Response from Kafka- topic <topic> has been validated	Debug	Topic state validated.
Received a Non-POST request for auditlogs–only POST is supported, dropping it	Debug	Received a Non-POST request for auditlogs –only POST is supported for this URL, dropping it.
x-appflow-id header not present in the auditlog POST request –dropping it	Debug	x-appflow-id header not present in the POST request for auditlogs –dropping it.
The requestor is not authorized to POST auditlogs –dropping it	Debug	The requestor is not authorized to POST auditlogs –dropping it.
Received Auditlog request	Debug	Received Auditlog request.
Received a Non-Post request for events –only POST is supported, dropping it	Debug	Received a Non-Post request for events –only POST is supported for this URL, dropping it.
x-appflow-id header not present in the event POST request –dropping it	Debug	x-appflow-id header not present in the POST request for events –dropping it.
The requestor is not authorized to POST events –dropping it	Debug	The requestor is not authorized to POST events –dropping it.
Received Event Request	Debug	Received Event Request.
Death of empty auditlog/event buffer	Debug	Empty buffers are not exported.
Death of event/ auditlog buffer because of lack of memory	Debug	Ran OOM.
Processed an event/ auditlog	Debug	Processed an Event/ Auditlog.
Death of an obese event/ auditlog	Debug	One single Auditlog/ Event was too large (exceeded 32 KiB).
Death of event/auditlog buffer due to JSON parsing error	Debug	Death of event/auditlog buffer due to JSON parsing error.
Death of <num> events/auditlogs during processing	Debug	Some auditlogs/ events dies during processing.

Log	Severity	Description
Pushed <num> event/auditlog JSON buffers to the Export Queue	Debug	Pushed some events/ auditlogs to the Export Queue.