4.3.1.10 Telemetry Module

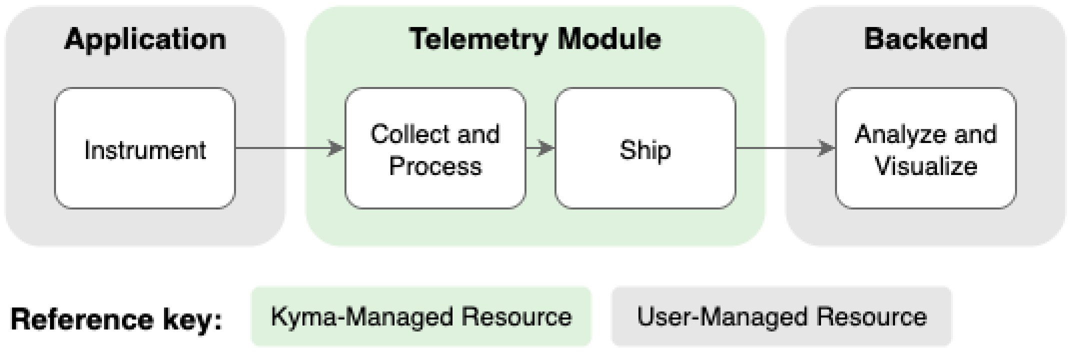
Use the Telemetry module to collect telemetry signals (logs, traces, and metrics) from your applications and send them to your preferred observability backend.

What is Telemetry?

With telemetry signals, you can understand the behavior and health of your applications and infrastructure. The Telemetry module provides a standardized way to collect these signals and send them to your observability backend, where you can analyze them and troubleshoot issues.

The Telemetry module processes three types of signals:

- Logs: Time-stamped records of events that happen over time.
- Traces: The path of a request as it travels through your application's components.
- Metrics: Aggregated numerical data about the performance or state of a component over time.



Telemetry signals flow through the following stages:

1. You instrument your application so that its components expose telemetry signals.
2. The signals are collected and enriched with infrastructural metadata.
3. You send the enriched signals to your preferred observability backend.
4. The backend stores your data, where you can analyze and visualize it.

The Telemetry module focuses on the collection, processing, and shipment stages of the observability workflow. It offers a vendor-neutral approach based on OpenTelemetry and doesn't force you into a specific backend. This means you can integrate with your existing observability platforms or choose from a wide range of available backends that best suit your operational needs.

→ Tip

Build your first telemetry pipeline with the hands-on lesson Collecting Application Logs and Shipping them to SAP Cloud Logging.

Features

To support telemetry for your applications, the Telemetry module provides the following features:

- **Consistent Telemetry Pipeline API**: Use a streamlined set of APIs based on the OTel Collector to collect, filter, and ship your logs, metrics, and traces (see Telemetry Pipeline API [page 2022]). You define a pipeline for each signal type to control how the data is processed and where it's sent. For details, see Collecting Logs [page 2027], Collecting Traces [page 2032], and Collecting Metrics [page 2036].
- **Flexible Backend Integration**: The Telemetry module is optimized for integration with SAP BTP observability services, such as SAP Cloud Logging. You can also send data to any backend that supports the OpenTelemetry protocol (OTLP), giving you the freedom to choose your preferred solution (see Integrate With Your OTLP Backend [page 2056]).

  > → Recommendation
  >
  > For production deployments, we recommend using a central telemetry solution located outside your cluster. For an example, see Integrate With SAP Cloud Logging [page 2059].
  >
  > For testing or development, in-cluster solutions may be suitable. For examples such as Dynatrace (or to learn how to collect data from applications based on the OpenTelemetry Demo App), see Integration Guides.

- **Seamless Istio Integration**: The Telemetry module seamlessly integrates with the Istio module when both are present in your cluster. For details, see Istio Integration [page 2085].
- **Automatic Data Enrichment**: The Telemetry module adds resource attributes as metadata, following OTel semantic conventions. This makes your data more consistent, meaningful, and ready for analysis in your observability backend. For details, see Automatic Data Enrichment [page 2054].
- **Instrumentation Guidance**: To generate telemetry data, you must instrument your code. Based on Open Telemetry (OTel), you get community samples on how to instrument your code using the Open Telemetry SDKs in most programming languages.

Scope

The Telemetry module focuses only on the signals of application logs, distributed traces, and metrics. Other kinds of signals are not considered. Also, audit logs are not in scope.

Supported integration scenarios are neutral to the vendor of the target system.

Architecture

The Telemetry module is built around a central controller, Telemetry Manager, which dynamically configures and deploys data collection components based on your pipeline resources.

To understand how the core components interact, see Telemetry Architecture [page 2075].

To learn how this model applies to each signal type, see:

- Logs Architecture [page 2078]
- Traces Architecture [page 2081]
- Metrics Architecture [page 2083]

API/Custom Resource Definitions

You configure the Telemetry module and its pipelines by creating and applying Kubernetes CustomResourceDefinitions (CRDs), which extend the Kubernetes API with custom additions.

To understand and configure the module's global settings, refer to the Telemetry CRD .

To define how to collect, process, and ship a specific signal, use the pipeline CRDs:

- LogPipeline CRD
- TracePipeline CRD
- MetricPipeline CRD

Resource Consumption

To learn more about the resources used by the Telemetry module, see Telemetry [page 3263].

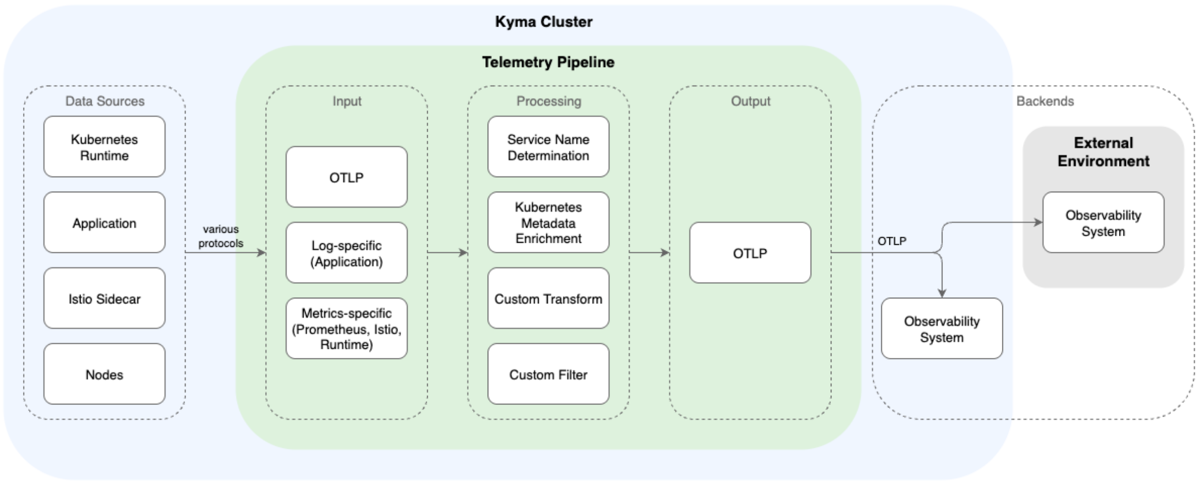
4.3.1.10.1 Telemetry Pipeline API

To collect and export telemetry data from your Kyma cluster, you define one or more pipelines for each signal type (logs, traces, metrics). You choose which data to collect and to which backend it's sent.

Pipeline Structure

You define Telemetry pipelines using three dedicated Kubernetes CRDs that extend the Kubernetes API: `LogPipeline`, `TracePipeline`, and `MetricPipeline`.

A pipeline defines how the data flows from the original data sources through the respective inputs and a series of processing steps to the backend you defined as output:



The pipelines use the OpenTelemetry Protocol (OTLP) as the primary method to ingest and export data, which gives you the flexibility to integrate with a wide range of observability backends.

While each pipeline is tailored to a specific signal, they all share a common structure:

Sample Code

```
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: <LogPipeline | TracePipeline | MetricPipeline>     # Choose pipeline
kind depending on signal type
metadata:
  name: my-observability-backend
spec:
  input:                # Enable additional inputs depending on signal type
    otlp:
      ...
  output:
    otlp:               # Integrate with your observability backend
      endpoint:
      ...
```

Pipeline Types

The `kind` attribute in the CRD specifies the type of telemetry data that the pipeline handles.

- `LogPipeline`: Collects logs from your application containers' standard output (`stdout`) and standard error (`stderr`), and from OTLP sources. It parses these logs, extracts useful information, and forwards them to your configured backend.
- `TracePipeline`: Collects trace data from OTLP sources, which show how requests flow between different components of your application. To collect Istio traces generated by applications or the service mesh, you must enable tracing within Istio using the Istio `Telemetry` CRD.
- `MetricPipeline`: Collects metrics from OTLP sources, Prometheus-annotated workloads, the Istio service mesh, and the Kubernetes runtime.

Input

In the `spec.input` section, you define the sources of your telemetry data. This section is the primary difference between the pipeline types.

All pipelines share `otlp` as the default input and can be configured with additional, signal-specific inputs:

By default, the `otlp` input is enabled for all signal types, which provisions a cluster-internal endpoint accepting OTLP data. For details, see Set Up the OTLP Input [page 2025].

Additionally, you can apply specific `input` configurations for each signal type:

- `LogPipeline`: The `application` input is enabled by default. Additionally, you can collect Istio access logs through the default `otlp` input. For both inputs, you can restrict from which Kubernetes resources you want to collect signals. For details, see Configure Application Logs [page 2029] and Configure Istio Access Logs [page 2030].
- `TracePipeline`: Tracing is a push-based model, so `otlp` is the only available input. The pipeline's OTLP endpoint receives span data pushed from your applications and Istio proxies. For Istio tracing, you can configure the sampling rate and apply individual settings to namespaces or workloads (see Configure Istio Tracing [page 2034]).
- `MetricPipeline`: You can select which metrics are collected by enabling inputs: `prometheus` (for scraping annotated workloads), `runtime` (for Kubernetes resource metrics), and `istio` (for service mesh metrics). You can filter all inputs by namespace. For details, see Collect Prometheus Metrics [page 2039], Collect Istio Metrics [page 2041], and Collect Runtime Metrics [page 2043].

Filtering and Processing

You can control the volume and focus of your telemetry data by filtering it based on Kubernetes resources like namespaces, containers, and workloads. For details, see Filtering and Processing Data [page 2045].

All pipelines automatically enrich telemetry data with Kubernetes resource attributes, such as Pod name, namespace, and labels. With this context information, you can easily identify the source of telemetry data in your backend. For details, see Automatic Data Enrichment [page 2054].

Output

In the `spec.output` section, you define the destination for your telemetry data. Each pipeline resource supports exactly one output, which sends data using OTLP.

You must specify the endpoint address of your observability backend. You can also configure the protocol (gRPC or HTTP) and the authentication details required to connect securely. For details, see Integrate With Your OTLP Backend [page 2056].

To send the same signal to multiple backends, create a separate pipeline resource for each destination. For details, see Route Specific Inputs to Different Backends [page 2026].

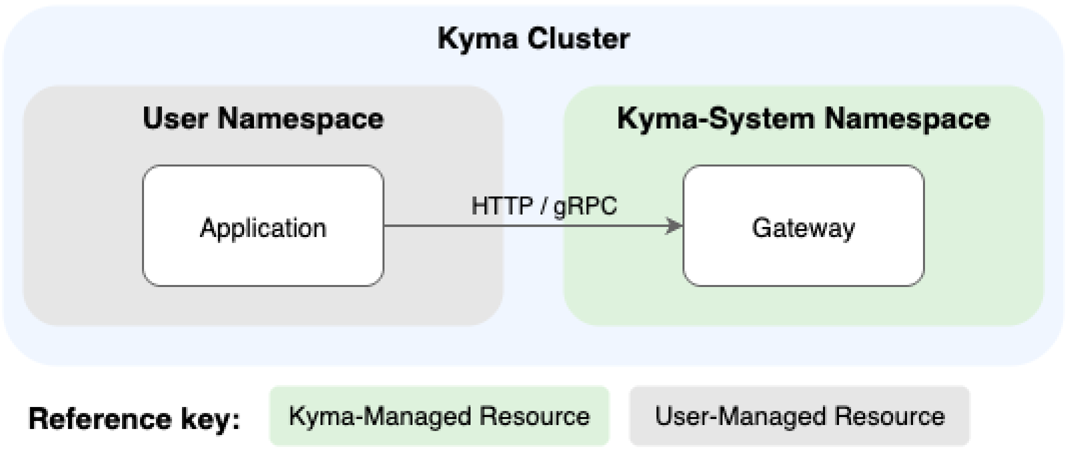
4.3.1.10.2 Set Up the OTLP Input

Use the default OTLP input to collect telemetry data from your instrumented applications and customize how that data is processed by the pipeline. You can specify which input data goes to which backend. If you're using Istio, data is collected from the service mesh.

Overview

When you create any `LogPipeline`, `TracePipeline`, or `MetricPipeline`, the Telemetry module automatically deploys the respective gateway. This opens a stable, cluster-internal OTLP endpoint for each signal type, ready to receive data from your applications.

Each endpoint listens on port `4317` for gRPC (default) and on port `4318` for HTTP.



Configure Your Application's OTLP Exporter

To send data from your application, first instrument your code using an OTel SDK for your programming language. The SDK's OTLP exporter sends the collected telemetry data to a backend.

It's recommended that you configure the exporter's destination by setting the standard environment variables in your application's deployment. This method avoids hardcoding endpoints in your application code.

Use the following environment variables to set the OTLP endpoint for each signal type:

- Traces gRPC: `export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="http://telemetry-otlp-traces.kyma-system:4317"`
- Traces HTTP: `export OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="http://telemetry-otlp-traces.kyma-system:4318/v1/traces"`
- Metrics gRPC: `export OTEL_EXPORTER_OTLP_METRICS_ENDPOINT="http://telemetry-otlp-metrics.kyma-system:4317"`

- Metrics HTTP: `export OTEL_EXPORTER_OTLP_METRICS_ENDPOINT="http://telemetry-otlp-metrics.kyma-system:4318/v1/metrics"`
- Logs gRPC: `export OTEL_EXPORTER_OTLP_LOGS_ENDPOINT="http://telemetry-otlp-logs.kyma-system:4317"`
- Logs HTTP: `export OTEL_EXPORTER_OTLP_LOGS_ENDPOINT="http://telemetry-otlp-logs.kyma-system:4318/v1/logs"`

Note

If your cluster uses Istio, communication with these endpoints is automatically secured with mTLS. For details, see Istio Integration [page 2085].

Verify the Endpoints

To see whether you've set up your gateways and their push endpoints successfully, check the status of the default `Telemetry` resource:

```
kubectl -n kyma-system get telemetries.operator.kyma-project.io default -oyaml
```

The output shows the available endpoints and the pipeline health under the `status.endpoints` section:

Output Code

```
endpoints:
  metrics:
    grpc: http://telemetry-otlp-metrics.kyma-system:4317
    http: http://telemetry-otlp-metrics.kyma-system:4318
  traces:
    grpc: http://telemetry-otlp-traces.kyma-system:4317
    http: http://telemetry-otlp-traces.kyma-system:4318
  logs:
    grpc: http://telemetry-otlp-logs.kyma-system:4317
    http: http://telemetry-otlp-logs.kyma-system:4318
```

Route Specific Inputs to Different Backends

For logs and metrics: If you have multiple pipelines sending data to different backends, you can specify which inputs are active for each pipeline. This is useful if you want one pipeline to handle only OTLP data and another to handle only data from a different source.

→ Tip

For more granular control, you can also filter incoming OTLP data by namespace. For details, see Filter Logs [page 2045] and Filter Metrics [page 2050].

For example, if you want to analyze `otlp` input data in one backend and only data from the log-specific `application` input in another backend, then disable the `otlp` input for the second backend. By default, `otlp` input is enabled.

{·} Sample Code

```
...
  input:
    application:
      enabled: true
    otlp:
      disabled: true
```

4.3.1.10.3 Collecting Logs

With the Telemetry module, you can observe and debug your applications by collecting, processing, and exporting logs. To begin collecting logs, you create a `LogPipeline` resource. It automatically collects OTLP logs and application logs from the `stdout/stderr` channel. You can also activate Istio log collection.

Overview

A `LogPipeline` is a Kubernetes custom resource (CR) that configures log collection for your cluster. When you create a `LogPipeline`, the Telemetry Manager automatically deploys the necessary components (for details, see Logs Architecture [page 2078]):

- A log gateway that provides a central OTLP endpoint for receiving logs pushed from your applications.
- A log agent that runs on each cluster node to collect logs written to `stdout` and `stderr` by your application containers.

The pipeline enriches all collected logs with Kubernetes metadata and transforms them into the OTLP format before sending them to your chosen backend.

The log collection feature is optional. If you don't create a `LogPipeline`, the log collection components are not deployed.

Prerequisites

- Before you can collect logs from a component, it must emit the logs. Typically, it uses a logger framework for the used language runtime (like Node.js) and prints them to the `stdout` or `stderr` channel (see Kubernetes: How nodes handle container logs). Alternatively, you can use the OTel SDK to use the push-based OTLP format .
- If you want to emit the logs to the `stdout/stderr` channel, use structured logs in a JSON format with a logger library like log4J. With that, the log agent can parse your log and enrich all JSON attributes as log attributes, and a backend can use that.
- If you prefer the push-based alternative with OTLP, also use a logger library like log4J. However, you must additionally instrument that logger and bridge it to the OTel SDK. For details, see OpenTelemetry: New First-Party Application Logs .

Minimal LogPipeline

For a minimal setup, you only need to create a `LogPipeline` that specifies your backend destination (see Integrate With Your OTLP Backend [page 2056]):

```
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: LogPipeline
metadata:
  name: backend
output:
    otlp:
      endpoint:
        value: http://<myEndpoint>:4317
```

By default, this minimal pipeline enables the following types of log collection:

- Application logs: Collects `stdout` and `stderr` logs from all containers running in non-system namespaces (such as `kyma-system` and `kube-system`).
- OTLP logs: Activates cluster-internal endpoints to receive logs in the OTLP format. Your applications can push logs directly to these URLs:
    - gRPC: `http://telemetry-otlp-logs.kyma-system:4317`
    - HTTP: `http://telemetry-otlp-logs.kyma-system:4318`

Configure Log Collection

You can customize your `LogPipeline` using the available parameters and attributes (see LogPipeline: Custom Resource Parameters):

- Configure or disable the collection of `application` logs from the `stdout/stderr` channel (see Configure Application Logs [page 2029]).
- Set up the collection of Istio access logs (see Configure Istio Access Logs [page 2030]).
- Choose from which specific namespaces you want to include or exclude logs (see Filter Logs [page 2045]).
- If you have more than one backend, specify which input source sends logs to which backend (see Route Specific Inputs to Different Backends [page 2026]).

Limitations

- **Throughput**:
    - When pushing OTLP logs of an average size of 2KB to the log gateway, using its default configuration (two instances), the Telemetry module can process approximately 12,000 logs per second (LPS). To ensure availability, the log gateway runs with multiple instances. For higher throughput, manually scale out the gateway by increasing the number of replicas (see Telemetry CRD). Ensure that the chosen scaling factor does not exceed the maximum throughput of the backend, as it may refuse logs if the rate is too high.
    - For example, to scale out the gateway for scenarios like a `Large` instance of SAP Cloud Logging (up to 30,000 LPS), you can raise the throughput to about 20,000 LPS by increasing the number of replicas to 4 instances.

    - The log agent, running one instance per node, handles tailing logs from `stdout` using the `runtime` input. When writing logs of an average size of 2KB to `stdout`, a single log agent instance can process approximately 9,000 LPS.
- **Load Balancing With Istio**: By design, the connections to the gateway are long-living connections (because OTLP is based on gRPC and HTTP/2). For optimal scaling of the gateway, the clients or applications must balance the connections across the available instances, which is automatically achieved if you use an Istio sidecar. If your application has no Istio sidecar, the data is always sent to one instance of the gateway.
- **Unavailability of Output**: For up to 5 minutes, a retry for data is attempted when the destination is unavailable. After that, data is dropped.
- **No Guaranteed Delivery**: The used buffers are volatile. If the gateway or agent instances crash, logs data can be lost.
- **Multiple LogPipeline Support**: The maximum amount of `LogPipeline` resources is 5.

4.3.1.10.3.1 Configure Application Logs

To collect logs that your applications write to `stdout` and `stderr`, create a `LogPipeline`. The `application` input is enabled by default and uses an agent on each node to tail container log files. You can control which namespaces and containers to include or exclude.

Prerequisites

- You have the Telemetry module in your cluster.
- You have access to Kyma dashboard. Alternatively, if you prefer CLI, you need kubectl .

Context

Use the `application` input section to restrict or specify which resources you want to include. You can define the namespaces to include in the input collection, exclude namespaces from the input collection, or choose that only system namespaces are included. For details, see LogPipeline: Custom Resource Parameters .

When you apply the `LogPipeline` resource to your Kubernetes cluster, a log agent is deployed and starts collecting the log data, transforms them to OTLP, and sends them to your backend. For details, see Transformation to OTLP Logs [page 2051].

Enable or Disable Log Collection

The `application` input is enabled by default. To create a pipeline that only accepts logs pushed with OTLP, you can disable it.

```
  ...
  input:
    application:
      enabled: false     # Default is true
```

By default, input is collected from all namespaces, except the system namespaces `kube-system`, `istio-system`, `kyma-system`, which are excluded by default.

> → Tip
>
> To select logs from specific namespaces and containers, or to include system namespaces, see Filter Logs [page 2045].

Discard the Original Log Body

By default, the log agent preserves the original JSON log message by moving it to the `attributes."log.original"` field after parsing. For details, see Transformation to OTLP Logs [page 2051].

To reduce data volume, you can disable this behavior. Set the parameter to **`false`** to discard the original JSON string after its contents are parsed into attributes.

```
...
  input:
    application:
      keepOriginalBody: false     # Default is true
```

4.3.1.10.3.2 Configure Istio Access Logs

To monitor traffic in your service mesh, configure Istio to send access logs. The `LogPipeline` automatically receives these logs through its default OTLP input.

Prerequisites

- You have the Istio module in your cluster. See Adding and Deleting a Kyma Module [page 3254].
- You have access to Kyma dashboard. Alternatively, if you prefer CLI, you need kubectl .

Context

Istio access logs help you monitor the "four golden signals" (latency, traffic, errors, and saturation) and troubleshoot anomalies.

By default, these logs are disabled because they can generate a high volume of data. To collect them, you apply an Istio `Telemetry` resource to a specific namespace, for the Istio Ingress Gateway, or for the entire mesh.

⚠ Caution

Enabling access logs, especially for the entire mesh, can significantly increase log volume and may lead to higher storage costs. Enable this feature only for the resources or components that you want to monitor.

After enabling Istio access logs, reduce data volume and costs by filtering them (see Filter Logs [page 2045]).

The Istio module provides a preconfigured extension provider called `kyma-logs`, which tells Istio to send access logs to the Telemetry module's OTLP endpoint. If your `LogPipeline` uses the legacy `http` output, you must use the `stdout-json` provider instead.

ⓘ Note

You can only have one mesh-wide Istio `Telemetry` resource. If you also plan to enable Istio tracing (see Configure Istio Tracing [page 2034]), configure both access logging and tracing in this single resource.

Enable Istio Logs for a Namespace

Apply the Istio `Telemetry` resource to a specifc namespace:

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: access-config
  namespace: $YOUR_NAMESPACE
spec:
  accessLogging:
    - providers:
      - name: kyma-logs
```

Enable Istio Logs for the Ingress Gateway

To monitor all traffic entering your mesh, enable access logs on the Istio Ingress Gateway (instead of the individual proxies of your workloads).

Apply the Istio `Telemetry` resource to the `istio-system` namespace, selecting the gateway Pods:

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
```

```
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  accessLogging:
    - providers:
      - name: kyma-logs
```

Enable Istio Logs for the Entire Mesh

To enable access logs globally for all proxies in the mesh, apply the Istio `Telemetry` resource to the `istio-system` namespace. Use this option with caution due to the high data volume.

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
spec:
  accessLogging:
    - providers:
      - name: kyma-logs
```

4.3.1.10.4 Collecting Traces

With the Telemetry module, you can collect distributed traces to understand the flow of requests through your applications and infrastructure. To begin collecting traces, you create a `TracePipeline` resource. It automatically collects OTLP traces and can be configured to collect traces from the Istio service mesh.

Overview

A `TracePipeline` is a Kubernetes custom resource (CR) that configures trace collection for your cluster. When you create a `TracePipeline`, it automatically provisions a trace gateway that provides a central OTLP endpoint receiving traces pushed from applications (for details, see Traces Architecture [page 2081]).

The pipeline enriches all collected traces with Kubernetes metadata before sending them to your chosen backend.

Trace collection is optional. If you don't create a `TracePipeline`, the trace gateway is not deployed.

Prerequisites

For the recording of a distributed trace, every involved component must propagate at least the trace context. For details, see Trace Context .

- In Kyma, all modules involved in users' requests support the W3C Trace Context protocol. The involved Kyma modules are, for example, Istio, Serverless, and Eventing.
- Your application also must propagate the W3C Trace Context for any user-related activity. This can be achieved easily using the Open Telemetry SDKs available for all common programming languages. If your application follows that guidance and is part of the Istio Service Mesh, it's already outlined with dedicated span data in the trace data collected by the Kyma telemetry setup.
- Furthermore, your application must enrich a trace with additional span data and send these data to the cluster-central telemetry services. You can achieve this with Open Telemetry SDKs .

With the default configuration, the trace gateway collects push-based OTLP traces of any container running in Kyma runtime, and the data is shipped to your backend.

Minimal TracePipeline

For a minimal setup, you only need to create a `TracePipeline` that specifies your backend destination (see Integrate With Your OTLP Backend [page 2056]):

```
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: TracePipeline
metadata:
  name: backend
output:
    otlp:
      endpoint:
        value: http://<myEndpoint>:4317
```

By default, this minimal pipeline collects push-based OTLP traces of any container running in Kyma runtime.

It activates cluster-internal endpoints to receive traces in the OTLP format. Applications can push traces directly to these URLs:

- gRPC: `http://telemetry-otlp-traces.kyma-system:4317`
- HTTP: `http://telemetry-otlp-traces.kyma-system:4318`

Configure Trace Collection

You can adjust the `TracePipeline` using runtime configuration with the available parameters and attributes (see TracePipeline: Custom Resource Parameters).

- If you use Istio, activate Istio tracing. For details, see Configure Istio Tracing [page 2034]. You can adjust which percentage of the trace data is collected.
- The Serverless module integrates the OpenTelemetry SDK by default. It automatically propagates the trace context for chained calls and reports custom spans for incoming and outgoing requests. You can add more spans within your Function's source code. For details, see Customize Function Traces .
- The Eventing module uses the CloudEvents protocol, which natively supports W3C Trace Context propagation. It ensures that the trace context is passed along but doesn't enrich a trace with more advanced span data.

Limitations

- **Throughput**: Assuming an average span with 40 attributes with 64 characters, the maximum throughput is 4200 span/sec ~= 15.000.000 spans/hour. If this limit is exceeded, spans are refused. To increase the maximum throughput, manually scale out the gateway by increasing the number of replicas for the trace gateway. For details, see Telemetry CRD .
- **Unavailability of Output**: For up to 5 minutes, a retry for data is attempted when the destination is unavailable. After that, data is dropped.
- **No Guaranteed Delivery**: The used buffers are volatile. If the OTel Collector instance crashes, trace data can be lost.
- **Multiple TracePipeline Support**: The maximum amount of `TracePipeline` resources is 5.
- **System Span Filtering**: System-related spans reported by Istio are filtered out without the opt-out option, for example:
  - Any communication of applications to the Telemetry gateways
  - Any communication from the gateways to backends

4.3.1.10.4.1 Configure Istio Tracing

To get an end-to-end view of requests in your service mesh, configure Istio to send trace data. The `TracePipeline` receives these traces through its default OTLP input. You can adjust the sampling rate and apply settings to specific namespaces or workloads.

Prerequisites

- You have the Istio module in your cluster. See Adding and Deleting a Kyma Module [page 3254].
- You have access to Kyma dashboard. Alternatively, if you prefer CLI, you need kubectl and curl .

Context

By default, Istio traces are disabled because they can generate a high volume of data. To collect them, you create an Istio `Telemetry` resource in the `istio-system` namespace. When you enable this feature, the Istio proxy sidecars automatically propagate the trace context and report spans for traffic between your services.

The Istio module provides a preconfigured extension provider called `kyma-traces` to send this data to the Telemetry module's trace gateway.

Istio plays a key role in distributed tracing. Its Ingress Gateway is typically where external requests enter your cluster. If a request doesn't have a trace context, Istio adds it. Furthermore, every component within the Istio service mesh runs an Istio proxy, which propagates the trace context and creates span data. When you enable Istio tracing, and it manages trace propagation in your application, you get a complete picture of a trace, because every component automatically contributes span data. Also, Istio tracing is preconfigured to use the vendor-neutral W3C Trace Context protocol.

⚠ Caution

Enabling Istio traces can significantly increase data volume and might quickly consume your trace storage. Start with a low sampling rate in production environments.

Enable Istio Tracing for the Entire Mesh

To enable tracing for all workloads in the service mesh, apply an Istio `Telemetry` resource to the `istio-system` namespace. Use this option to establish a baseline configuration for your mesh.

ⓘ Note

You can only have one mesh-wide Istio `Telemetry` resource in the `istio-system` namespace. If you also want to configure Istio access logs, combine both configurations into a single resource (see Configure Istio Access Logs [page 2030]).

The following example enables tracing with a default sampling rate of 1%:

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
spec:
  tracing:
  - providers:
    - name: "kyma-traces"
    randomSamplingPercentage: 1.00
```

→ Tip

After setting a mesh-wide default in the `istio-system` namespace, you can apply more specific tracing configurations for an entire namespace or for individual workloads within a namespace. This is useful for debugging a particular service by increasing its sampling rate without affecting the entire mesh. For details, see Filter Traces [page 2049].

Configure the Sampling Rate

By default, Istio samples 1% of traces to reduce data volume. To change it, set the `randomSamplingPercentage` value. The sampling decision is propagated within the trace context to ensure that either all or no spans for a given trace are reported.

ⓘ Note

For production environments, a low sampling rate (1–5%) is recommended to manage costs and performance. For development or debugging, you can set it to 100.00 to capture every trace.

```
apiVersion: telemetry.istio.io/v1
```

```
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
spec:
  tracing:
  - providers:
    - name: "kyma-traces"
    randomSamplingPercentage: 5.00 # Samples 5% of all traces
```

Propagate Trace Context Without Reporting Spans

In some cases, you may want Istio to propagate the W3C Trace Context for context-aware logging but not report any trace spans. This approach enriches your access logs with `traceId` and `spanId` without the overhead of full distributed tracing.

To achieve this, set `randomSamplingPercentage` to `0.00` in your mesh-wide configuration.

→ Tip

For guidance on Istio access logs, see Configure Istio Access Logs [page 2030].

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
spec:
  tracing:
  - providers:
    - name: "kyma-traces"
    randomSamplingPercentage: 0
```

4.3.1.10.5 Collecting Metrics

With the Telemetry module, you can collect metrics from your workloads and Kubernetes resources to monitor their health, performance, and behavior. To begin collecting metrics, you create a `MetricPipeline` resource. You can collect Prometheus, Istio, and runtime metrics.

Overview

A `MetricPipeline` is a Kubernetes custom resource (CR) that configures metric collection for your cluster. When you create a `MetricPipeline`, it automatically provisions the necessary components (for details, see Metrics Architecture [page 2083]):

- A metric gateway that provides a central OTLP endpoint for receiving metrics pushed from applications.
- A metric agent that runs on each cluster node to pull (scrape) metrics from applications and Kubernetes resources.

The pipeline enriches all collected metrics with Kubernetes metadata. It also transforms non-OTLP formats (like Prometheus) into the OTLP standard before sending them to your chosen backend.

Metric collection is optional. If you don't create a `MetricPipeline`, the metric collection components are not deployed.

Prerequisites

- Before you can collect metrics data from a component, it must expose (or instrument) the metrics. Typically, it instruments specific metrics for the used language runtime (like Node.js) and custom metrics specific to the business logic. Also, the exposure can be in different formats, like the pull-based Prometheus format or the push-based OTLP format .
- If you want to use Prometheus-based metrics, you must have instrumented your application using a library like the Prometheus client library , with a port in your workload exposed serving as a Prometheus metrics endpoint.
- If you want to scrape the metric endpoint with Istio, your Service `port` definition must define the app protocol. For details, see Scrape Metrics from Istio-enabled Workloads [page 2040].
- For the instrumentation, you typically use an SDK, namely the Prometheus client libraries or the Open Telemetry SDKs . Both libraries provide extensions to activate language-specific auto-instrumentation like for Node.js, and an API to implement custom instrumentation.

Minimal MetricPipeline

For a minimal setup, you only need to create a `MetricPipeline` that specifies your backend destination (see Integrate With Your OTLP Backend [page 2056]):

```
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: MetricPipeline
metadata:
  name: backend
output:
    otlp:
      endpoint:
        value: http://myEndpoint:4317
```

By default, this minimal pipeline collects the following types of metrics:

- OTLP Metrics: Activates cluster-internal endpoints to receive metrics in the OTLP format. Your applications can push metrics directly to these URLs:
  - gRPC: `http://telemetry-otlp-metrics.kyma-system:4317`
  - HTTP: `http://telemetry-otlp-metrics.kyma-system:4318`
- Health Metrics: Collects health and performance metrics about the Telemetry module's components. This input is always active and cannot be disabled. For details, see Monitor Pipeline Health [page 2069].

To collect metrics from Kyma modules like Istio, Eventing, or Serverless, enable additional inputs.

Configure Metrics Collection

You can adjust the `MetricPipeline` using runtime configuration with the available parameters (see MetricPipeline: Custom Resource Parameters).

- Scrape `prometheus` metrics from applications that expose a Prometheus-compatible endpoint (see Collect Prometheus Metrics [page 2039]).
- Collect `istio` service mesh metrics from Istio proxies and control plane components (see Collect Istio Metrics [page 2041]).
- Collect `runtime` resource usage and status metrics from Kubernetes components like Pods, Nodes, and Deployments (see Collect Runtime Metrics [page 2043]).
- Use diagnostic metrics to debug your `prometheus` and `istio` configuration.
- Choose from which specific namespaces you want to include or exclude metrics (see Filter Metrics [page 2050]).
- Avoid redundancy by dropping push-based OTLP metrics that are sent directly to the metric gateway (see Route Specific Inputs to Different Backends [page 2026]).

Limitations

- **Throughput**: Assuming an average metric with 20 metric data points and 10 labels, the default metric **gateway** setup has a maximum throughput of 34K metric data points/sec. If more data is sent to the gateway, it is refused. To increase the maximum throughput, manually scale out the gateway by increasing the number of replicas for the metric gateway (see Telemetry CRD).
  The metric **agent** setup has a maximum throughput of 14K metric data points/sec per instance. If more data must be ingested, it is refused. If a metric data endpoint emits more than 50.000 metric data points per scrape loop, the metric agent refuses all the data.
- **Load Balancing With Istio**: To ensure availability, the metric gateway runs with multiple instances. If you want to increase the maximum throughput, use manual scaling and enter a higher number of instances. By design, the connections to the gateway are long-living connections (because OTLP is based on gRPC and HTTP/2). For optimal scaling of the gateway, the clients or applications must balance the connections across the available instances, which is automatically achieved if you use an Istio sidecar. If your application has no Istio sidecar, the data is always sent to one instance of the gateway.
- **Unavailability of Output**: For up to 5 minutes, a retry for data is attempted when the destination is unavailable. After that, data is dropped.
- **No Guaranteed Delivery**: The used buffers are volatile. If the gateway or agent instances crash, metric data can be lost.
- **Multiple MetricPipeline Support**: The maximum amount of `MetricPipeline` resources is 5.

4.3.1.10.5.1 Collect Prometheus Metrics

To collect metrics from applications that expose a Prometheus-compatible endpoint, enable the `prometheus` input in your `MetricPipeline` and annotate your Pods or Services for discovery. You can enable diagnostic metrics and control from which namespaces metrics are collected.

Prerequisites

Instrument your application using a library like the Prometheus client library or the OTel SDK (with Prometheus exporter). Expose a port in your workload as a Prometheus metrics endpoint.

Activate Prometheus Metrics

By default, the `prometheus` input is disabled. If your applications emit Prometheus metrics, enable the collection of Prometheus-based metrics:

```
...
input:
  prometheus:
    enabled: true
```

→ Tip

To validate or debug your configuration, use diagnostic metrics.

To select metrics from specific namespaces or to include system namespaces, see Filter Metrics [page 2050].

Enable Metrics Collection With Annotations

The metric agent automatically discovers Prometheus endpoints in your cluster by looking for specific annotations on your Kubernetes Services or Pods.

To enable automatic metrics collection, apply the following annotations. If your Pod has an Istio sidecar, annotate the Service. Otherwise, annotate the Pod directly.

ⓘ Note

If your service mesh enforces STRICT mTLS, the agent scrapes the endpoint over HTTPS automatically. If you don't use STRICT mTLS, add the annotation `prometheus.io/scheme: http` to force scraping over plain HTTP.

Annotation Key	Values	Description
`prometheus.io/scrape` (mandatory)	true, false (no default value)	Set to true to enable scraping for this target.
`prometheus.io/port` (mandatory)	8080, 9100 (no default value)	Specify the port on the Pod where your application exposes metrics.
`prometheus.io/path`	/metrics (default), /custom_metrics	Set the HTTP path for the metrics endpoint.
`prometheus.io/scheme`	https-metrics (default with Istio), http (default without Istio)	Define the protocol for scraping: Either HTTPS with mTLS, or plain HTTP.
`prometheus.io/param_<name>`	Example: `format:` prometheus (no default)	Add a URL parameter to the scrape request. For example, `prometheus.io/param_format:` prometheus adds `?format=prometheus` to the URL.

For example, see the following `Service` configuration:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/port: "8080"
    prometheus.io/scrape: "true"
  name: sample
spec:
  ports:
  - name: http-metrics
    appProtocol: http
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: sample
  type: ClusterIP
```

Scrape Metrics from Istio-enabled Workloads

If your application is part of an Istio service mesh, you must consider service port naming and mutual TLS (mTLS) configuration:

- Istio must be able to identify the `appProtocol` from the Service port definition. Otherwise, Istio may block the scrape request.
  You must either prefix the port name with the protocol like in `http-metrics`, or explicitly define the `appProtocol` attribute.
- The metric agent can scrape endpoints from workloads that enforce mutual TLS (mTLS). For scraping through HTTPS, Istio must configure the workload using STRICT mTLS mode.
  If you can't use STRICT mTLS mode, you can set up scraping through plain HTTP by adding the following annotation to your Service: `prometheus.io/scheme: http`. For related troubleshooting, see Log Entry: Failed to Scrape Prometheus Endpoint [page 2073].

Collect Diagnostic Metrics

To validate or debug your scraping configuration for the `prometheus` and `istio` input, you can use diagnostic metrics. By default, they are disabled.

> ⓘ Note
>
> Unlike the `prometheus` and `istio` inputs, the `runtime` input gathers data directly from Kubernetes APIs instead of using a scraping process, so it does not generate scrape-specific diagnostic metrics.

To use diagnostic metrics, enable the `diagnosticMetrics` for the input in your `MetricPipeline`:

```
...
input:
  <istio | prometheus>:
    enabled: true
    diagnosticMetrics:
      enabled: true
```

When enabled, the metric agent generates metrics about its own scrape jobs, such as the following:

- `up`: The scraping was successful
- `scrape_duration_seconds`: Duration of the scrape
- `scrape_samples_scraped`: The number of samples the target exposed
- `scrape_samples_post_metric_relabeling`: The number of samples remaining after metric relabeling was applied
- `scrape_series_added`: The approximate number of new series in this scrape

4.3.1.10.5.2 Collect Istio Metrics

To monitor the health and performance of your service mesh, enable the `istio` input in your `MetricPipeline`. This scrapes metrics directly from Istio proxies (sidecars) and the control plane. You can enable Envoy and diagnostic metrics, and control from which namespaces metrics are collected.

Prerequisites

- The Istio module is added in your cluster. See Adding and Deleting a Kyma Module [page 3254].
- You have access to Kyma dashboard. Alternatively, if you prefer CLI, you need kubectl and curl .

> ⓘ Note
>
> This configuration collects metrics generated by the Istio proxies themselves. To collect custom metrics exposed by your own applications running in the service mesh, you must configure your application accordingly. For details, see Scrape Metrics from Istio-enabled Workloads [page 2040].

Activate Istio Metrics

By default, the `istio` input is disabled. If you are using Istio, enable the collection of Istio metrics:

```
...
input:
  istio:
    enabled: true
```

With this, the metric agent starts collecting all Istio metrics from Istio sidecars from all namespaces (including system namespaces).

→ Tip

To validate or debug your configuration, use diagnostic metrics.

To select metrics from specific namespaces, see Filter Metrics [page 2050].

Collect Envoy Metrics

By default, the metric agent collects only Istio metrics (prefixed with `istio_`) and ignores Envoy metrics (prefixed with `envoy_`).

Envoy metrics help you understand the performance and behavior of your Envoy proxy, providing details like request rates, latencies, and error counts. For details, see Envoy metrics and server metrics .

To use Envoy metrics to observe and troubleshoot service mesh traffic, enable the `envoyMetrics` section under the `istio` input:

```
...
input:
  istio:
    enabled: true
    envoyMetrics:
      enabled: true
```

Collect Diagnostic Metrics

To validate or debug your scraping configuration for the `prometheus` and `istio` input, you can use diagnostic metrics. By default, they are disabled.

ⓘ Note

Unlike the `prometheus` and `istio` inputs, the `runtime` input gathers data directly from Kubernetes APIs instead of using a scraping process, so it does not generate scrape-specific diagnostic metrics.

To use diagnostic metrics, enable the `diagnosticMetrics` for the input in your `MetricPipeline`:

```
...
```

```
  input:
    <istio | prometheus>:
      enabled: true
      diagnosticMetrics:
        enabled: true
```

When enabled, the metric agent generates metrics about its own scrape jobs, such as the following:

- `up`: The scraping was successful
- `scrape_duration_seconds`: Duration of the scrape
- `scrape_samples_scraped`: The number of samples the target exposed
- `scrape_samples_post_metric_relabeling`: The number of samples remaining after metric relabeling was applied
- `scrape_series_added`: The approximate number of new series in this scrape

4.3.1.10.5.3 Collect Runtime Metrics

To monitor the health and resource usage of your Kubernetes cluster, enable the `runtime` input in your `MetricPipeline`. This uses an agent on each node to gather metrics for resources like Pods, Nodes, and Deployments. You can choose the specific resources to monitor and control from which namespaces metrics are collected.

Activate Runtime Metrics

By default, the `runtime` input is disabled. If you want to monitor your Kubernetes resources, enable the collection of runtime metrics:

```
...
  input:
    runtime:
      enabled: true
```

With this, the metric agent starts collecting all runtime metrics from all resources (Pod, container, Node, Volume, DaemonSet, Deployment, StatefulSet, and Job).

→ Tip

To select metrics from specific namespaces or to include system namespaces, see Filter Metrics [page 2050].

Select Resource Types

By default, metrics for all supported resource types are collected. To enable or disable the collection of metrics for a specific resource, use the `resources` section in the `runtime` input.

The following example collects only DaemonSet, Deployment, StatefulSet, and Job metrics:

```
...
  input:
    runtime:
      enabled: true
      resources:
        pod:
          enabled: false
        container:
          enabled: false
        node:
          enabled: false
        volume:
          enabled: false
        daemonset:
          enabled: true
        deployment:
          enabled: true
        statefulset:
          enabled: true
        job:
          enabled: true
```

See a summary of the types of information you can gather for each resource:

Resource	Metrics Collected
`pod`	CPU, memory, filesystem, and network usage; current Pod phase
`container`	CPU/memory requests, limits, and usage; container restart count
`node`	Aggregated CPU, memory, filesystem, and network usage for the Node
`volume`	Filesystem capacity, usage, and inode statistics for persistent volumes
`deployment`	Number of desired versus available replicas
`daemonset`	Number of desired, current, and ready Nodes
`statefulset`	Number of desired, current, and ready Pods
`job`	Counts of active, successful, and failed Pods

To learn which specific metrics are collected from which source (`kubletstatsreceiver` or `k8sclusterreceiver`), see Runtime Metrics .

4.3.1.10.6 Filtering and Processing Data

When you configure the inputs for your logs, traces, and metrics, you can choose from which specific Kubernetes resources you want to include or exclude data. Your data is automatically transformed and enriched so you can analyze it in your OTLP backend.

Filtering Mechanisms

The Telemetry module supports the following mechanisms to filter the data, which apply at different stages of the collection process:

- Filtering within a pipeline (`LogPipeline`, `MetricPipeline`): You can configure the `input` section of a pipeline to select or reject data before it is processed by the agent. This is the most common method for filtering application logs, runtime metrics, and Prometheus metrics.
- Configuring the data source (Istio `Telemetry` CRD): For Istio-generated data (access logs and traces), you configure the Istio `Telemetry` resource itself. This controls which workloads generate data and at what volume (sampling rate), before it is even sent to a pipeline.

Processing Data

Application logs from containers are automatically transformed into structured OpenTelemetry (OTLP) log records.

All pipelines automatically enrich telemetry data with Kubernetes resource attributes, such as Pod name, namespace, and labels. With this context information, you can easily identify the source of telemetry data in your backend.

4.3.1.10.6.1 Filter Logs

Filter logs from the OTLP, application, and Istio input to control which data your pipeline processes. You can define filters to include or exclude logs based on their source namespace, container, and other attributes.

Overview

Source	Granularity	Behavior without `namespaces` Block	Collect from All Namespaces	Collect from Specific Namespaces
OTLP (default)	Namespace	**includes** system namespaces	This is the default, no action needed.	Use the `include` or `exclude` filter

Source	Granularity	Behavior without `namespaces` Block	Collect from All Namespaces	Collect from Specific Namespaces
Application	Namespace, Container*	**excludes** system namespaces	Set the `system` attribute to **`true`**	Use the `include` or `exclude` filter
Istio	Namespace, Workload (`selector`), Log content (`filter.expression`)	n/a	Apply the Istio `Telemetry` resource mesh-wide	Apply the Istio `Telemetry` resource to specific namespaces

* The `application` input provides an additional `containers` selector that behaves the same way as the `namespaces` selector.

Filter OTLP Logs by Namespaces

You can filter incoming OTLP logs by namespace. By default, all system namespaces are included. The `include` and `exclude` filters are mutually exclusive.

- To collect metrics from specific namespaces, use the `include` filter:

  {·} Sample Code

  ```
  ...
  input:
    otlp:
      namespaces:
        include:
          - namespaceA
          - namespaceB
  ```

- To collect OTLP logs from all namespaces except specific ones, use the `exclude` filter:

  {·} Sample Code

  ```
  ...
  input:
    otlp:
      namespaces:
        exclude:
          - namespaceA
          - namespaceB
  ```

Filter Application Logs by Namespace

You can control which namespaces to collect logs from using `include`, `exclude`, and `system` filters. The `include` and `exclude` filters are mutually exclusive.

- To collect logs from specific namespaces, use the `include` filter:

  ```
  ...
  input:
    application:
      namespaces:
        include:
          - namespaceA
          - namespaceB
  ```
- To collect logs from all namespaces except specific ones, use the `exclude` filter:

  ```
  ...
  input:
    application:
      namespaces:
        exclude:
          - namespaceC
  ```

Collect Application Logs From System Namespaces

By default, application logs from `kube-system`, `istio-system`, and `kyma-system` are excluded. To override this and collect logs from them, set the `system` attribute to true:

```
...
input:
  application:
    enabled: true
      namespaces:
        system: true
```

Filter Application Logs by Container

You can also filter logs based on the container name with `include` and `exclude` filters. These filters apply in addition to any namespace filters.

The following pipeline collects input from all namespaces excluding `kyma-system` and only from the `istio-proxy` containers:

```
...
  input:
    application:
      enabled: true
      namespaces:
        exclude:
          - myNamespace
      containers:
        exclude:
          - myContainer
    otlp:
      ...
```

Select Istio Logs From a Specific Application

To limit logging to a single application within a namespace, configure label-based selection for this workload with a selector in the Istio `Telemetry` resource.

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: access-config
  namespace: $YOUR_NAMESPACE
spec:
  selector:
    matchLabels:
      service.istio.io/canonical-name: $YOUR_LABEL
  accessLogging:
    - providers:
      - name: kyma-logs
```

Filter Istio Logs by Content

To reduce noise and cost, filter your Istio access logs. This is especially useful for filtering out traffic that doesn't use an HTTP-based protocol, as those log entries often lack useful details.

Add a `filter` expression to the same `accessLogging` block that you used to enable the logs. The expression uses Envoy attributes to define which log entries to keep.

The following example enables mesh-wide logging but only keeps logs that have a defined request protocol, effectively filtering out most non-HTTP traffic.

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
 name: mesh-default
 namespace: istio-system
spec:
 accessLogging:
 - filter:
     expression: 'has(request.protocol)'
   providers:
   - name: kyma-logs
```

4.3.1.10.6.2 Filter Traces

`TracePipeline` resources have no `input` specification. You can configure Istio trace collection by applying the Istio `Telemetry` resource to specific namespaces.

Override Tracing for a Namespace or Workload

After setting a mesh-wide default, apply more specific tracing configurations for an entire namespace or for individual workloads within a namespace. Use this to debug a particular service by increasing its sampling rate without affecting the entire mesh.

To do this, create an Istio `Telemetry` resource in the workload's namespace. To apply a tracing configuration to a specific workload within the namespace, add a `selector` that matches the workload's labels.

For example, increase the sampling rate (in this example, to `100.00`) to debug a specific application without affecting the entire mesh:

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: tracing
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: "my-app"
  tracing:
  - providers:
    - name: "kyma-traces"
    randomSamplingPercentage: 100.00
```

Disable Tracing for a Specific Workload

To completely disable Istio span reporting for a specific workload while keeping it enabled for the rest of the mesh, create a `Telemetry` resource that targets the workload and set `disableSpanReporting` to **`true`**.

```
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: $YOUR_APP_NAME-tracing-disable
  namespace: $YOUR_NAMESPACE
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: "$YOUR_APP_NAME"
  tracing:
  - providers:
    - name: "kyma-traces"
    disableSpanReporting: true
```

4.3.1.10.6.3 Filter Metrics

Filter metrics from the OTLP, Istio, Prometheus, and runtime input to control which data your pipeline processes. You can define filters to include or exclude metrics based on their source namespace and resource type.

Overview

Source	Granularity	Behavior without `namespaces` Block	Collect from All Namespaces	Collect from Specific Namespaces
OTLP (default)	Namespace	**includes** system namespaces	This is the default, no action needed.	Use the `include` or `exclude` filter
Istio	Namespace	**excludes** system namespaces	Add `namespaces: {}` to the input's configuration	Use the `include` or `exclude` filter
Prometheus	Namespace	**excludes** system namespaces	Add `namespaces: {}` to the input's configuration	Use the `include` or `exclude` filter
Runtime	Namespace, Resource Type*	**excludes** system namespaces	Add `namespaces: {}` to the input's configuration	Use the `include` or `exclude` filter

* The `runtime` input provides additional filters for Kubernetes resources such as Pods or Nodes. For details, see Select Resource Types [page 2043].

Filter Metrics by Namespace

For the all inputs (`otlp`, `prometheus`, `istio`, and `runtime`), you can filter incoming metrics by namespace. The `include` and `exclude` filters are mutually exclusive.

- To collect metrics from specific namespaces, use the `include` filter:

  Sample Code

  ```
  ...
  input:
    <otlp | prometheus | istio | runtime>:
      enabled: true
      namespaces:
        include:
          - namespaceA
          - namespaceB
  ```

- To collect metrics from all namespaces except specific ones, use the `exclude` filter:

  {·} Sample Code

  ```
  ...
  input:
    <otlp | prometheus | istio | runtime>:
      enabled: true
      namespaces:
        exclude:
          - namespaceA
          - namespaceB
  ```

Collect Metrics From System Namespaces

For `otlp` metrics, system namespaces are included by default.

To include system namespaces for `prometheus`, `istio`, and `runtime` metrics without specifying any other namespaces, explicitly configure an empty namespace object: `namespaces: {}`:

{·} Sample Code

```
...
input:
  <prometheus | istio | runtime>:
    enabled: true
    namespaces: {}
```

4.3.1.10.6.4 Transformation to OTLP Logs

Learn how the log agent processes the original container logs and transforms them into structured OpenTelemetry (OTLP) log records.

Original Log Message

The following example shows a container `myContainer` in Pod `myPod`, running in namespace `myNamespace`, logging to `stdout` with the following JSON message:

```
{
  "level": "warn",
  "message": "This is the original message",
  "tenant": "myTenant",
  "traceID": "123"
}
```

Log Tailing

The log agent reads the log message from a log file managed by the container runtime. The file name contains namespace, Pod, and container information that becomes available later as log attributes. The raw log record looks like the following example:

```
{
  "time": "2022-05-23T15:04:52.193317532Z",
  "stream": "stdout",
  "_p": "F",
  "log": "{\"level\": \"warn\",\"message\": \"This is the original
message\",\"tenant\": \"myTenant\",\"trace_id\": \"123\"}"
}
```

After the tailing, the created OTLP record looks like the following example:

```
{
  "time": "2022-05-23T15:04:52.100000000Z",
  "observedTime": "2022-05-23T15:04:52.200000000",
  "attributes": {
   "log.file.path": "/var/log/pods/myNamespace_myPod-<containerID>/myContainer/
<containerRestarts>.log",
   "log.iostream": "stdout"
  },
  "resourceAttributes": {
    "k8s.container.name": "myContainer",
    "k8s.container.restart_count": "<containerRestarts>",
    "k8s.pod.name": "myPod",
    "k8s.namespace.name": "myNamespace"
  },
  "body": "{\"level\": \"warn\",\"message\": \"This is the original
message\",\"tenant\": \"myTenant\",\"trace_id\": \"123\"}"
}
```

The log agent enriches all information identifying the log source (such as container, Pod, and namespace name) as resource attributes, following Kubernetes conventions . It enriches further metadata, like the original file name and channel, as log attributes, following log attribute conventions . The agent uses the `time` value from the container runtime's log entry as the `time` attribute in the new OTel record, as it closely matches the actual log event time. Additionally, the agent sets `observedTime` with the time it actually reads the log record, as the OTel log specification recommends. The agent moves the log payload to the OTLP `body` field.

JSON Parsing

If the `body` value is a JSON document, the agent parses the value and enriches all JSON root attributes as additional log attributes. The agent moves the original body into the `log.original` attribute (managed with the `LogPipeline` attribute `input.application.keepOriginalBody: true`).

After JSON parsing, the OTLP record looks like the following example:

```
{
  "time": "2022-05-23T15:04:52.100000000Z",
  "observedTime": "2022-05-23T15:04:52.200000000",
  "attributes": {
   "log.file.path": "/var/log/pods/myNamespace_myPod-<containerID>/myContainer/
<containerRestarts>.log",
    "log.iostream": "stdout",
    "log.original": "{\"level\": \"warn\",\"message\": \"This is the original
message\",\"tenant\": \"myTenant\",\"trace_id\": \"123\"}",
    "level": "warn",
    "tenant": "myTenant",
    "trace_id": "123",
    "message": "This is the original message"
  },
  "resourceAttributes": {
    "k8s.container.name": "myContainer",
    "k8s.container.restart_count": "<containerRestarts>",
    "k8s.pod.name": "myPod",
    "k8s.namespace.name": "myNamespace"
  },
  "body": ""
}
```

Severity Parsing

Typically, a log message includes a log level in the `level` field. Based on this, the agent parses the `level` log attribute with a severity parser. If parsing succeeds, the agent transforms the log attribute into the OTel attributes `severityText` and `severityNumber`.

Trace Parsing

OTLP natively attaches trace context to log records. If possible, the log agent parses the following log attributes according to the W3C-Tracecontext specification :

- `trace_id`
- `span_id`
- `trace_flags`
- `traceparent`

Log Body Determination

Because the original log message typically resides in the `body` attribute, the agent moves a log attribute called `message` (or `msg`) into the body.

At this point, before further enrichment, the resulting overall log record looks like the following example:

```
{
  "time": "2022-05-23T15:04:52.100000000Z",
  "observedTime": "2022-05-23T15:04:52.200000000",
  "attributes": {
    "log.file.path": "/var/log/pods/myNamespace_myPod-<containerID>/myContainer/
<containerRestarts>.log",
    "log.iostream": "stdout",
    "log.original": "{\"level\": \"warn\",\"message\": \"This is the original
message\",\"tenant\": \"myTenant\",\"trace_id\": \"123\"}",
    "tenant": "myTenant",
  },
  "resourceAttributes": {
    "k8s.container.name": "myContainer",
    "k8s.container.restart_count": "<containerRestarts>",
    "k8s.pod.name": "myPod",
    "k8s.namespace.name": "myNamespace"
  },
  "body": "This is the original message",
  "severityNumber": 13,
  "severityText": "warn",
  "trace_id": 123
}
```

This structured record is now ready to be shipped to your observability backend. For details on further enrichment, see Automatic Data Enrichment [page 2054].

4.3.1.10.6.5 Automatic Data Enrichment

The Telemetry gateways automatically enrich your data with OTel resource attributes, so you can easily identify the source of the data in your backend.

Service Name

The service name is the logical name of the service that emits the telemetry data. The gateway ensures that this attribute always has a valid value.

If you don't provide a service name, or if its value follows the pattern `unknown_service:<process.executable.name>` as described in the specification, the gateway generates it from Kubernetes metadata.

The gateway determines the service name based on the following hierarchy of labels and names:

1. `app.kubernetes.io/name`: Pod label value
2. `app`: Pod label value
3. Deployment/DaemonSet/StatefulSet/Job name
4. Pod name
5. If none of the above is available, the value is unknown_service

Kubernetes Metadata

`k8s.*` attributes encapsulate various pieces of Kubernetes metadata associated with the Pod, such as:

- `k8s.pod.name`: The Kubernetes Pod name of the Pod that emitted the data.
- `k8s.pod.uid`: The Kubernetes Pod ID of the Pod that emitted the data.
- `k8s.<workload kind>.name`: The Kubernetes workload name to which the emitting Pod belongs. Workload is either Deployment, DaemonSet, StatefulSet, Job, or CronJob.

- `k8s.namespace.name`: The Kubernetes namespace name with which the emitting Pod is associated.
- `k8s.cluster.name`: A logical identifier of the cluster, which, by default, is the API Server URL. To set a custom name, configure the `enrichments.cluster.name` field in the Telemetry CRD.
- `k8s.cluster.uid`: A unique identifier of the cluster, realized by the UID of the `kube-system` namespace.
- `k8s.node.name`: The Kubernetes node name to which the emitting Pod is scheduled.
- `k8s.node.uid`: The Kubernetes Node ID to which the emitting Pod belongs.

Pod Label Attributes

In the Telemetry CRD , you can also specify your own enrichments of telemetry data based on Pod labels.

To capture custom application metadata (for example, for filtering, grouping, or correlation), configure specific label keys or label key prefixes to include in the enrichment process. The gateway adds all matching Pod labels to the telemetry data as resource attributes, using the label key format `k8s.pod.label.<label_key>`.

The following example configuration enriches the telemetry data with Pod labels that match the specified keys or key prefixes:

- `k8s.pod.label.app.kubernetes.io/name`: The value of the exact label key `app.kubernetes.io/name` from the Pod.
- `k8s.pod.label.app.kubernetes.io.*`: All labels that start with the prefix `app.kubernetes.io` from the Pod, where `*` is replaced by the actual label key.

```
apiVersion: operator.kyma-project.io/v1alpha1
kind: Telemetry
metadata:
  name: default
  namespace: kyma-system
spec:
  enrichments:
    extractPodLabels:
    - key: "<myExactLabelKey>" # for example, "app.kubernetes.io/name"
    - keyPrefix: "<myLabelPrefix>" # for example, "app.kubernetes.io"
```

Cloud Provider Attributes

If data is available, the gateway automatically adds cloud provider attributes to the telemetry data:

- `cloud.provider`: Cloud provider name
- `cloud.region`: Region where the Node runs (from Node label `topology.kubernetes.io/region`)
- `cloud.availability_zone`: Zone where the Node runs (from Node label `topology.kubernetes.io/zone`)

Host Attributes

If data is available, the gateway automatically adds host attributes to the telemetry data:

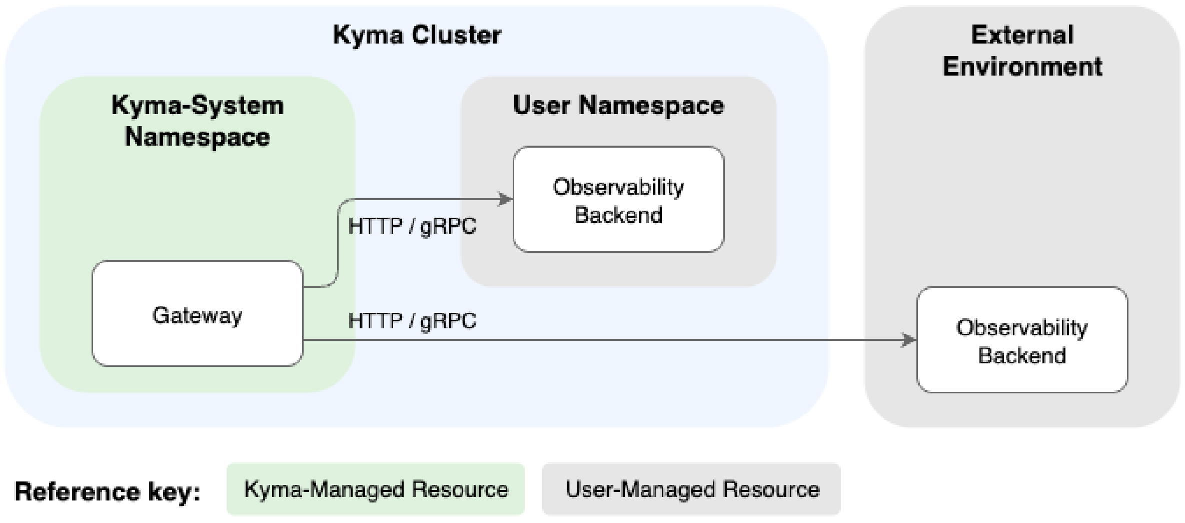
- `host.type`: Machine type of the Node (from Node label `node.kubernetes.io/instance-type`)
- `host.arch`: CPU architecture of the system the Node runs on (from Node label `kubernetes.io/arch`)

4.3.1.10.7 Integrate With Your OTLP Backend

Define the backend destination for your telemetry data by configuring the OTLP output, including the protocol and authentication method.

Overview

In every pipeline, you must configure an `output` section, which defines the destination for your telemetry data using OTLP. This is where you specify your chosen observability backend (see OpenTelemetry: Vendors).



> Note
>
> Each pipeline resource supports exactly one backend. However, you can send specific inputs to different backends by setting up designated pipelines. For details, see Route Specific Inputs to Different Backends [page 2026].

Specify the OTLP Endpoint

For the minimal pipeline configuration, you only specify an OTLP endpoint; though it's recommended that you use authentication.

```
...
  output:
    otlp:
      endpoint:
        value: https://backend.example.com:4317
```

Choose a Protocol

The default protocol for shipping the data to a backend is gRPC. If your backend requires the HTTP protocol instead, set the `protocol` attribute to **`http`**. Based on this setting, the gateway chooses the exporter: `otlp` for gRPC (the default) or `otlphttp` for HTTP.

Ensure the port in your endpoint URL is correct for the chosen protocol.

```
 ...
 output:
    otlp:
      protocol: http
      endpoint:
        value: https://backend.example.com:4318
```

Set Up Authentication

For each pipeline, add authentication details (like user names, passwords, certificates, or tokens) to connect securely to your observability backend. You can use mutual TLS (mTLS), custom headers, or Basic Authentication.

While you can choose to add your authentication details from plain text, it's recommended to store these sensitive details in a Kubernetes `Secret` and reference the Secret's keys in your pipeline configuration. When you rotate the `Secret` and update its values, Telemetry Manager detects the changes and applies the new `Secret` to your setup.

→ Tip

If you use a Secret owned by the SAP BTP Service Operator, you can configure an automated rotation policy with a specific rotation frequency and don't have to intervene manually.

- To use client certificates for mTLS, configure the `tls` section with your public certificate and private key.

  ```
  ...
  output:
    otlp:
      endpoint:
        value: https://backend.example.com/otlp:4317
      tls:
  ```

```
        cert:
          valueFrom:
            secretKeyRef:
              name: backend
              namespace: default
              key: cert
        key:
          valueFrom:
            secretKeyRef:
              name: backend
              namespace: default
              key: key
```

- To send an authentication token (such as a bearer token) in an HTTP header, configure the `headers` section.

```
  ...
  output:
    otlp:
      endpoint:
        value: https://backend.example.com:4317
      headers:
      - name: Authorization
        prefix: Bearer
        valueFrom:
          secretKeyRef:
              name: backend
              namespace: default
              key: token
```

- To use a username and password for authentication, configure the `authentication.basic` section.

```
  ...
  output:
    otlp:
      endpoint:
        valueFrom:
          secretKeyRef:
              name: backend
              namespace: default
              key: endpoint
      authentication:
        basic:
          user:
            valueFrom:
              secretKeyRef:
                name: backend
                namespace: default
                key: user
          password:
            valueFrom:
              secretKeyRef:
                name: backend
                namespace: default
                key: password
```

- If you want to configure authentication details from plain text, use the following pattern. The example shows mTLS, but you can also use Basic Authentication or custom headers:

```
  ...
  output:
    otlp:
      endpoint:
        value: https://backend.example.com/otlp:4317
      tls:
        cert:
          value: |
            -----BEGIN CERTIFICATE-----
            ...
        key:
          value: |
            -----BEGIN RSA PRIVATE KEY-----
            ...
```

ⓘ Note

If your backend is running inside the cluster and is part of the Istio service mesh, the gateways automatically secure the connection and you don't have to configure the `authentication` block. For details, see Sending Data to In-Cluster Backends [page 2086].

4.3.1.10.7.1 Integrate With SAP Cloud Logging

Configure the Telemetry module to send logs, metrics, and traces from your cluster to an SAP Cloud Logging instance. By centralizing this data in your SAP Cloud Logging instance, you can store, visualize, and analyze the observability of your applications.

Prerequisites

- Kyma as the target deployment environment, with the following modules added (see Adding and Deleting a Kyma Module [page 3254]):
    - Telemetry module
    - To collect data from your Istio service mesh: Istio module (default module)
    - SAP BTP Operator module (default module)
- An instance of SAP Cloud Logging with OpenTelemetry ingestion enabled. For details, see Ingest via OpenTelemetry API Endpoint.

  → Tip

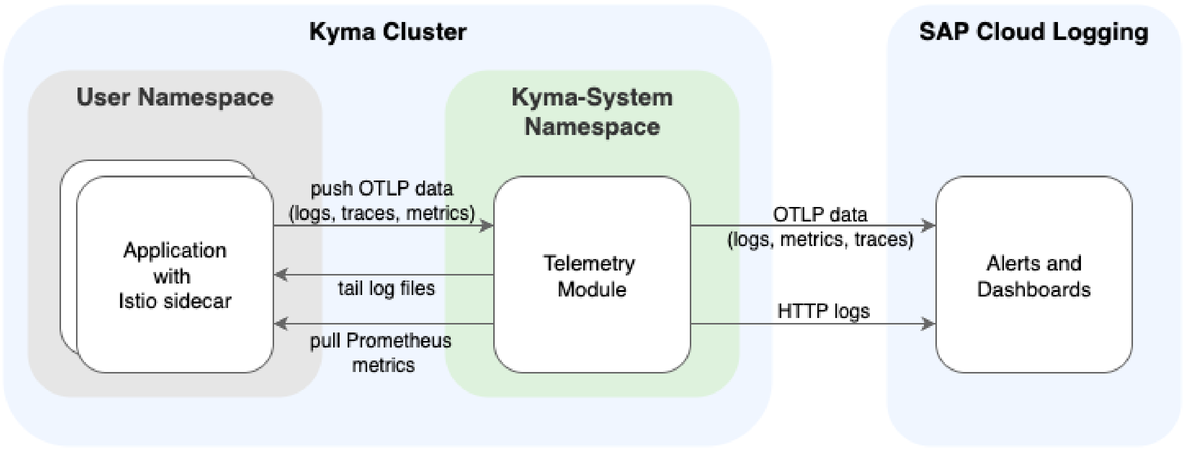
  Create the SAP Cloud Logging instance with the SAP BTP service operator (see Create an SAP Cloud Logging Instance through SAP BTP Service Operator), because it takes care of creation and rotation of the required Secret. However, you can choose any other method of creating the instance and the Secret, as long as the parameter for OTLP ingestion is enabled in the instance. For details, see Configuration Parameters.
- A Secret in the respective namespace in your Kyma cluster, holding the credentials and endpoints for the instance. It's recommended that you rotate your Secret (see SAP BTP Security Recommendation BTP-CLS-0003). In the following example, the Secret is named “sap-cloud-logging” and the namespace “sap-cloud-logging-integration”, as illustrated in the secret-example.yaml .
- Kubernetes CLI (kubectl) (see Install the Kubernetes Command Line Tool).
- UNIX shell or Windows Subsystem for Linux (WSL) to execute commands.

Context

The Telemetry module supports shipping logs and ingesting distributed traces as well as metrics from applications and the Istio service mesh to SAP Cloud Logging.

First, set up the Telemetry module to ship the logs, traces, and metrics to your backend by deploying the pipelines and other required resources. Then, you configure Kyma dashboard integration. Finally, set up SAP Cloud Logging alerts and dashboards.

SAP Cloud Logging is an instance-based and environment-agnostic observability service to store, visualize, and analyze logs, metrics, and traces.



Ship Logs To SAP Cloud Logging

The Telemetry module supports two protocols for shipping logs to your backend. The method you choose determines how you configure your `LogPipeline` and which SAP Cloud Logging dashboards you can use.

Context

Choose one of the following methods to configure shipping for application and access logs

- OpenTelemetry (recommended): Use the OTLP-native method for all new configurations. It provides a unified way to send logs, metrics, and traces.
- Fluent Bit (legacy): Use this method only if you depend on the preconfigured `Kyma_*` dashboards in SAP Cloud Logging. These dashboards were designed for the HTTP and are not compatible with the OTLP logging output.

Procedure

Set Up Application Logs

1. Deploy a `LogPipeline` for application logs:

   - For OTLP, run:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: LogPipeline
metadata:
  name: sap-cloud-logging
spec:
  input:
    application:
      enabled: true
  output:
    otlp:
      endpoint:
        valueFrom:
          secretKeyRef:
            name: sap-cloud-logging
            namespace: sap-cloud-logging-integration
            key: ingest-otlp-endpoint
      tls:
        cert:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-otlp-cert
        key:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-otlp-key
EOF
```

   - For HTTP, run:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: LogPipeline
metadata:
  name: sap-cloud-logging-application-logs
spec:
  input:
    application:
      containers:
        exclude:
          - istio-proxy
  output:
    http:
      dedot: true
      host:
        valueFrom:
          secretKeyRef:
            name: sap-cloud-logging
            namespace: sap-cloud-logging-integration
            key: ingest-mtls-endpoint
      tls:
        cert:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-mtls-cert
        key:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-mtls-key
      uri: /customindex/kyma
EOF
```

2. Verify that the `LogPipeline` is running:

```
kubectl get logpipelines
```

Set Up Istio Access Logs

By default, Istio sidecar injection and Istio access logs are disabled in Kyma. To analyze them, you must enable them:

3. Enable Istio sidecar injection for your workload (see Enabling Istio Sidecar Proxy Injection [page 1828]).
4. Depending on your log shipment protocol, configure the Istio `Telemetry` resource (see Configure Istio Access Logs [page 2030]):

    - For OTLP, set up the Istio `Telemetry` resource with the OTLP-based `kyma-logs` extension provider.
    - For HTTP:
        1. Set up the Istio `Telemetry` resource with the `stdout-json` extension provider.
        2. Deploy a `LogPipeline` for Istio access logs:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: LogPipeline
metadata:
  name: sap-cloud-logging-access-logs
spec:
  input:
    application:
      containers:
        include:
          - istio-proxy
  output:
    http:
      dedot: true
      host:
        valueFrom:
          secretKeyRef:
            name: sap-cloud-logging
            namespace: sap-cloud-logging-integration
            key: ingest-mtls-endpoint
      tls:
        cert:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-mtls-cert
        key:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-mtls-key
      uri: /customindex/istio-envoy-kyma
```

```
EOF
```

3. Verify that the `LogPipeline` for Istio access logs is running:

```
kubectl get logpipelines
```

Ship Traces To SAP Cloud Logging

You can set up ingestion of distributed traces from applications and the Istio service mesh to the OTLP endpoint of the SAP Cloud Logging service instance.

Procedure

Set Up Traces

1. Deploy a `TracePipeline`:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: TracePipeline
metadata:
  name: sap-cloud-logging
spec:
  output:
    otlp:
      endpoint:
        valueFrom:
          secretKeyRef:
            name: sap-cloud-logging
            namespace: sap-cloud-logging-integration
            key: ingest-otlp-endpoint
      tls:
        cert:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-otlp-cert
        key:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-otlp-key
EOF
```

2. Verify that the `TracePipeline` is running:

```
kubectl get tracepipelines
```

Set Up Istio Tracing

By default, Istio sidecar injection and Istio tracing are disabled in Kyma. To analyze them, you must enable them:

3. Enable Istio sidecar injection for your workload (see Enabling Istio Sidecar Proxy Injection [page 1828]).

4. Configure the Istio `Telemetry` resource to use the `kyma-traces` extension provider based on OTLP (see Configure Istio Tracing [page 2034]).

Ship Metrics To SAP Cloud Logging

You can set up ingestion of metrics from applications and the Istio service mesh to the OTLP endpoint of the SAP Cloud Logging service instance.

Procedure

1. Deploy a `MetricPipeline`:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: MetricPipeline
metadata:
  name: sap-cloud-logging
spec:
  input:
    prometheus:
      enabled: false
    istio:
      enabled: false
    runtime:
      enabled: false
  output:
    otlp:
      endpoint:
        valueFrom:
          secretKeyRef:
            name: sap-cloud-logging
            namespace: sap-cloud-logging-integration
            key: ingest-otlp-endpoint
      tls:
        cert:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-otlp-cert
        key:
          valueFrom:
            secretKeyRef:
              name: sap-cloud-logging
              namespace: sap-cloud-logging-integration
              key: ingest-otlp-key
EOF
```

   The default configuration creates a gateway to receive OTLP metrics from your applications.
2. **Optional:** To collect additional metrics, such as those from the runtime or Istio, configure the presets in the `input` section of the `MetricPipeline`. For the available options, see Collecting Metrics [page 2036].
3. Verify that the `MetricPipeline` is running:

```
kubectl get metricpipelines
```

Set Up Kyma Dashboard Integration

To add direct links from Kyma dashboard to *SAP Cloud Logging*, apply the `ConfigMap` that corresponds to your chosen log shipping method.

Context

Depending on the output you use in your `LogPipeline`, apply the `ConfigMap`. If your Secret has a different name or namespace, then download the file first and adjust the namespace and name accordingly in the `dataSources` section of the file.

Procedure

1. If your Secret has a name or namespace different from the example, download the file and edit the `dataSources` section before you apply it.
2. Apply the `ConfigMap`:
   - For OTLP, run:

     ```
     kubectl apply
     -f https://raw.githubusercontent.com/kyma-project/telemetry-manager/main/
     docs/user/integration/sap-cloud-logging/kyma-dashboard-configmap.yaml
     ```
   - For HTTP, run:

     ```
     kubectl apply
     -f https://raw.githubusercontent.com/kyma-project/telemetry-manager/main/
     docs/user/integration/sap-cloud-logging/kyma-dashboard-http-configmap.yaml
     ```

Use SAP Cloud Logging Alerts

You can import predefined alerts for SAP Cloud Logging to monitor the health of your telemetry integration.

Procedure

1. In the SAP Cloud Logging dashboard, define a “notification channel” to receive alert notifications.
2. To import a monitor, use the development tools of the SAP Cloud Logging dashboard.
3. Execute `POST _plugins/_alerting/monitors`, followed by the contents of the respective JSON file.
4. Depending on the pipelines you are using, enable some or all of the following alerts:

   The alerts are based on JSON documents defining a `Monitor` for the alerting plugin.

Monitored Component	File	Description
SAP Cloud Logging	alert-health.json	Monitors the health of the underlying OpenSearch cluster in SAP Cloud Logging using the cluster health API . Triggers if the cluster status becomes **red**.
SAP Cloud Logging	alert-rejection-in-progress.json	Monitors the `cls-rejected-*` index for new data. Triggers if new rejected data is observed.
Kyma Telemetry Integration	alert-telemetry-status.json	Monitors the status of the Telemetry module. Triggers if the module reports a non-ready state.
Kyma Telemetry Integration	alert-log-ingestion.json	For OTLP: Monitors the single `LogPipeline` used in the OTLP method. Triggers if log data stops flowing.
Kyma Telemetry Integration	alert-app-log-ingestion.json	For HTTP (legacy): Monitors the application log `LogPipeline`. Triggers if log data stops flowing.
Kyma Telemetry Integration	alert-access-log-ingestion.json	For HTTP (legacy): Monitors the Istio access log `LogPipeline`. Triggers if log data stops flowing.
Kyma Telemetry Integration	alert-trace-ingestion.json	Monitors the `TracePipeline`. Triggers if trace data stops flowing to SAP Cloud Logging.
Kyma Telemetry Integration	alert-metric-ingestion.json	Monitors the `MetricPipeline`. Triggers if metric data stops flowing to SAP Cloud Logging.

5. After importing, edit the monitor to attach your notification channel or destination and adjust thresholds as needed.
6. Verify that the new monitor definition is listed among the SAP Cloud Logging alerts.

Use SAP Cloud Logging Dashboards

You can view logs, traces, and metrics in SAP Cloud Logging dashboards. Several dashboards come with SAP Cloud Logging, and you can import additional dashboards as needed.

Context

The preconfigured `Kyma_*` dashboards in SAP Cloud Logging are compatible only with the legacy (HTTP) logging method.

Procedure

- For the status of the SAP Cloud Logging integration with the Telemetry module, import the file dashboard-status.ndjson .
- For application logs and Istio access logs using the `http` output: Use the preconfigured dashboards prefixed with `Kyma_*`.
- For traces, use the OpenSearch plugin "Observability".
- For runtime metrics, import the file dashboard-runtime.ndjson .
- For Istio Pod metrics, import the file dashboard-istio.ndjson .

4.3.1.10.7.2 Migrate Your LogPipeline From HTTP to OTLP

To use the OpenTelemetry Protocol (OTLP) for sending logs, you must migrate your `LogPipeline` from the `http` output to the `otlp` output. With OTLP, you can correlate logs with traces and metrics, collect logs pushed directly from applications, and use features available only for the OTLP-based stack.

Prerequisites

- You have an active Kyma cluster with the Telemetry module added.
- You have one or more `LogPipeline` resources that use the `http` output.
- Your observability backend has an OTLP ingestion endpoint.

Context

When you want to migrate to the `otlp` output, create a new `LogPipeline`. To prevent data loss, run it in parallel with your existing pipeline. After verifying that the new pipeline works correctly, you can delete the old one.

You can't modify an existing `LogPipeline` to change its output type. You must create a new resource.

Procedure

1. Create a new `LogPipeline` that uses the `otlp` output.

   Pay special attention to the following settings (for details, see Integrate With Your OTLP Backend [page 2056]):

   - Endpoint URL: Use the OTLP-specific ingestion endpoint from your observability backend. This URL is different from the one used for the legacy `http` output.

- Protocol: The `otlp` output defaults to the gRPC protocol. If your backend uses HTTP, you must include the protocol in the endpoint URL (for example, `https://my-otlp-http-endpoint:4318`).
- Authentication: The OTLP endpoint often uses different credentials or API permissions than your previous log ingestion endpoint. Verify that your credentials have the necessary permissions for OTLP log ingestion.

```
apiVersion: telemetry.kyma-project.io/v1alpha1
kind: LogPipeline
metadata:
  name: <my-otlp-pipeline>
spec:
  output:
    otlp:
      endpoint:
        value: "<my-backend>:4317"
```

2. **Optional:** To enrich logs with Pod labels, configure the central `Telemetry` resource (Telemetry CRD).

   In contrast to a Fluent Bit `LogPipeline`, the `otlp` output doesn't automatically add all Pod labels. To continue enriching logs with specific labels, you must explicitly enable it in the `spec.enrichments.extractPodLabels` field.

   > Note
   >
   > Enrichment with Pod annotations is no longer supported.

3. Deploy the new `LogPipeline`:

   ```
   kubectl apply -f logpipeline.yaml
   ```

4. Check that the new `LogPipeline` is healthy:

   ```
   kubectl get logpipeline <my-otlp-pipeline>
   ```

5. Check your observability backend to confirm that log data is arriving.
6. Delete the old `LogPipeline`:

   ```
   kubectl delete logpipeline <my-old-pipeline>
   ```

Results

Your cluster now sends logs exclusively through your new OTLP-based `LogPipeline`. Your enrichment logic is preserved.

4.3.1.10.8 Monitor Pipeline Health

The Telemetry module is designed to be reliable and resilient. However, there may be situations when the instances drop data or cannot handle the load, and you must take action.

Overview

The Telemetry module automatically handles temporary issues to prevent data loss and ensure that the OTel Collector instances of your pipelines are operational and healthy. For example, if your backend is temporarily unavailable, the module buffers your data and attempts to resend it when the connection is restored.

The Telemetry module continuously monitors the health of your pipelines (see Self Monitor [page 2078]). To ensure that your Telemetry pipelines operate reliably, you can monitor their health data in the following ways:

- Perform manual checks by inspecting the status conditions of your pipeline resources with `kubectl`.
- Set up continuous monitoring by using a `MetricPipeline` to export health metrics to your observability backend, where you can set up dashboards and alerts.

Check Pipeline Status

For a quick check, you can inspect the `status` of a pipeline resource directly.

1. Run `kubectl get` for the pipeline that you want to inspect:
   - For `LogPipeline`: **`kubectl get logpipeline <your-pipeline-name>`**
   - For `TracePipeline`: **`kubectl get tracepipeline <your-pipeline-name>`**
   - For `MetricPipeline`: **`kubectl get tracepipeline <your-pipeline-name>`**
2. Review the output. A healthy pipeline shows `True` for all status conditions.

   Output Code

   ```
   NAME      CONFIGURATION GENERATED   GATEWAY HEALTHY   FLOW HEALTHY
   backend   True                      True              True
   ```
3. If any condition is `False`, investigate problem and fix it.

To understand the meaning of each status condition, see the detailed reference for each pipeline type:

- LogPipeline Status
- TracePipeline Status
- MetricPipeline Status

Set Up Health Monitoring and Alerts

For production environments, set up continuous monitoring by exporting the health metrics to your observability backend, where you can create dashboards and configure alerts using alert rules. For an example, see Integrate With SAP Cloud Logging [page 2059]

⚠ Caution

Do not scrape the metrics endpoint of the OpenTelemetry Collector instances. These metrics are an internal implementation detail and are subject to breaking changes when the underlying Collector is updated. For stable health monitoring, rely on the status conditions of your `LogPipeline`, `MetricPipeline`, or `TracePipeline` custom resources.

To collect these health metrics, you must have at least one active `MetricPipeline` in your cluster. This pipeline automatically collects and exports health data for all of your pipelines, including `LogPipeline` and `TracePipeline` resources.

The Telemetry module emits the following metrics for health monitoring:

- `kyma.resource.status.conditions`: Represents the status of a specific condition on a resource. It is available for all pipelines and the main `Telemetry` resource.
  Values: `1` ("True"), `0` ("False"), or `-1` ("Unknown")
  Specific attributes:
  - `metric.attributes.type`: The type of the status condition
  - `metric.attributes.status`: The status of the condition
  - `metric.attributes.reason`: A programmatic identifier indicating the reason for the condition's last transition
- `kyma.resource.status.state`: Represents the overall state of the main `Telemetry` resource.
  Values: `1` ("Ready") or `0` ("Not Ready")
  Specific attributes: `state`: The value of the `status.state` field
- Additionally, the following attributes are attached to all health metrics to identify the source resource:
  - `k8s.resource.group`: The group of the resource
  - `k8s.resource.version`: The version of the resource
  - `k8s.resource.kind`: The kind of the resource
  - `k8s.resource.name`: The name of the resource

To create an alert, define a rule that triggers on a specific metric value. For example, to create an alert that fires if a pipeline's `TelemetryFlowHealthy` condition becomes "False" (indicating data flow issues), use the following PromQL query:

```
min by (k8s_resource_name)
((kyma_resource_status_conditions{type="TelemetryFlowHealthy",k8s_resource_kind="
metricpipelines"})) == 0
```

If there are issues with one of the pipelines, see Troubleshooting for the Telemetry Module [page 2071].

4.3.1.10.9 Troubleshooting for the Telemetry Module

Troubleshoot problems related to the Telemetry module and its pipelines.

No Data Arrive at the Backend

Symptom

- No data arrive at the backend.
- In the pipeline status, the `TelemetryFlowHealthy` condition has status `GatewayAllTelemetryDataDropped` or `AgentAllTelemetryDataDropped`.

Cause

The pipeline cannot connect to the backend and drops all data, typically because of one of the following reasons:

- Authentication Error: The credentials in your `MetricPipeline` output are incorrect.
- Network Unreachable: The backend URL is wrong, a firewall is blocking the connection, or there's a DNS issue preventing the agent or gateway from reaching the backend.
- Backend is Down: The observability backend itself is not running or is unhealthy.

Solution

1. Identify the failing component.
   - If the status is `GatewayAllTelemetryDataDropped`, the problem is with the gateway.
   - If the status is `AgentAllTelemetryDataDropped`, the problem is with the agent.
2. To check the failing component's logs, call **`kubectl logs -n kyma-system <POD_NAME>`**:
   - For the gateway, check Pod `telemetry-<log/trace/metric>-gateway`.
   - For the agent, check Pod `telemetry-<log/metric>-agent`.

   Look for errors related to authentication, connectivity, and DNS.
3. Check if the backend is up and reachable.
4. Based on the log messages, fix the `output` section of your pipeline and re-apply it.

Not All Data Arrive at the Backend

Symptom

- The backend is reachable and the connection is properly configured, but some data points are refused.
- In the pipeline status, the `TelemetryFlowHealthy` condition has status `GatewaySomeTelemetryDataDropped` or `AgentSomeTelemetryDataDropped`.

Cause

This status indicates that the telemetry gateway or agent is successfully sending data, but the backend is rejecting some of it. Common reasons are:

- Rate Limiting: Your backend is rejecting requests because you're sending too much data at once.
- Invalid Data: Your backend is rejecting specific data due to incorrect formatting, invalid labels, or other schema violations.

Solution

1. Check the error logs for the affected Pod by calling `kubectl logs -n kyma-system {POD_NAME}`:
    - For `GatewaySomeTelemetryDataDropped`, check Pod `telemetry-<log/trace/metric>-gateway`.
    - For `AgentSomeTelemetryDataDropped`, check Pod `telemetry-<log/metric>-agent`.
2. Go to your observability backend and investigate potential causes.
3. If the backend is limiting the rate by refusing data, try the following options:
    - Increase the ingestion rate of your backend (for example, by scaling out your SAP Cloud Logging instances).
    - Reduce emitted data by re-configuring the pipeline (for example, by disabling certain inputs or applying filters).
    - Reduce emitted data in your applications.
4. Otherwise, fix the issues as indicated in the logs.

Gateway Throttling

Symptom

In the pipeline status, the `TelemetryFlowHealthy` condition has status `GatewayThrottling`.

Cause

The gateway is receiving data faster than it can process and forward it.

Solution

Manually scale out the capacity by increasing the number of replicas for the affected gateway. For details, see Telemetry CRD .

Custom Spans Don’t Arrive at the Backend, but Istio Spans Do

Symptom

You see traces generated by the Istio service mesh, but traces from your own application code (custom spans) are missing.

Cause

The OpenTelemetry (OTel) SDK version used in your application is incompatible with the OTel Collector version.

Solution

1. Check which SDK version you're using for instrumentation.
2. Investigate whether it's compatible with the OTel Collector version.
3. If necessary, upgrade to a supported SDK version.

Too Few Traces Arrive at the Backend

Symptom

The observability backend shows significantly fewer traces than the number of requests your application receives.

Cause

By default, Istio samples only 1% of requests for tracing to minimize performance overhead (see Configure Istio Tracing [page 2034]).

For example, in low-traffic environments (for development or testing) or for low-traffic services, the request volume can be so low that a 1% sample rate may result in capturing zero traces.

Solution

- To see more traces in the backend, increase the percentage of requests that are sampled (see Configure the Sampling Rate [page 2035]).
- Alternatively, to trace a single request, force sampling by adding a `traceparent` HTTP header to your client request. This header contains a sampled flag that instructs the system to capture the trace, bypassing the global sampling rate (see Trace Context: Sampled Flag).

Log Entry: Failed to Scrape Prometheus Endpoint

Symptom

- Your custom Prometheus metrics don't appear in your observability backend.
- In the metric agent (OTel Collector) logs, you see entries saying `Failed to scrape Prometheus endpoint` like the following:

Output Code

```
2023-08-29T09:53:07.123Z warn internal/transaction.go:111 Failed to
scrape Prometheus endpoint {"kind": "receiver", "name": "prometheus/
app-pods", "data_type": "metrics", "scrape_timestamp": 1693302787120,
"target_labels": "{__name__=\"up\", instance=\"10.42.0.18:8080\",
job=\"app-pods\"}"}
```

Cause

There's a configuration or network issue between the metric agent and your application, such as:

- The Service that exposes your metrics port doesn't specify the application protocol.

- The workload is not configured to use `STRICT` mTLS mode, which the metric agent uses by default.
- A deny-all `NetworkPolicy` in your application's namespace prevents the agent from scraping metrics from annotated workloads.

Solution

- Define the application protocol in the Service port definition by either prefixing the port name with the protocol, or define the `appProtocol` attribute.
- If the issue is with mTLS, either configure your workload to use `STRICT` mTLS, or switch to unencrypted scraping by adding the `prometheus.io/scheme: "http"` annotation to your workload.
- Create a new `NetworkPolicy` to explicitly allow ingress traffic from the metric agent; such as the following example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-traffic-from-agent
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: "annotated-workload" # <your workload here>
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kyma-system
      podSelector:
        matchLabels:
          telemetry.kyma-project.io/metric-scrape: "true"
  policyTypes:
  - Ingress
```

Log Buffer Filling Up

Symptom

In the `LogPipeline` status, the `TelemetryFlowHealthy` condition has status `AgentBufferFillingUp`.

Cause

The backend ingestion rate is too low compared to the export rate of the log agent, causing data to accumulate in its buffer.

Solution

You can either increase the capacity of your backend or reduce the volume of log data being sent. Try one of the following options:

- Increase the ingestion rate of your backend (for example, by scaling out your SAP Cloud Logging instances).
- Reduce emitted data by re-configuring the pipeline (for example, by disabling certain inputs or applying namespace filters).
- Reduce the amount of log data generated by your applications.

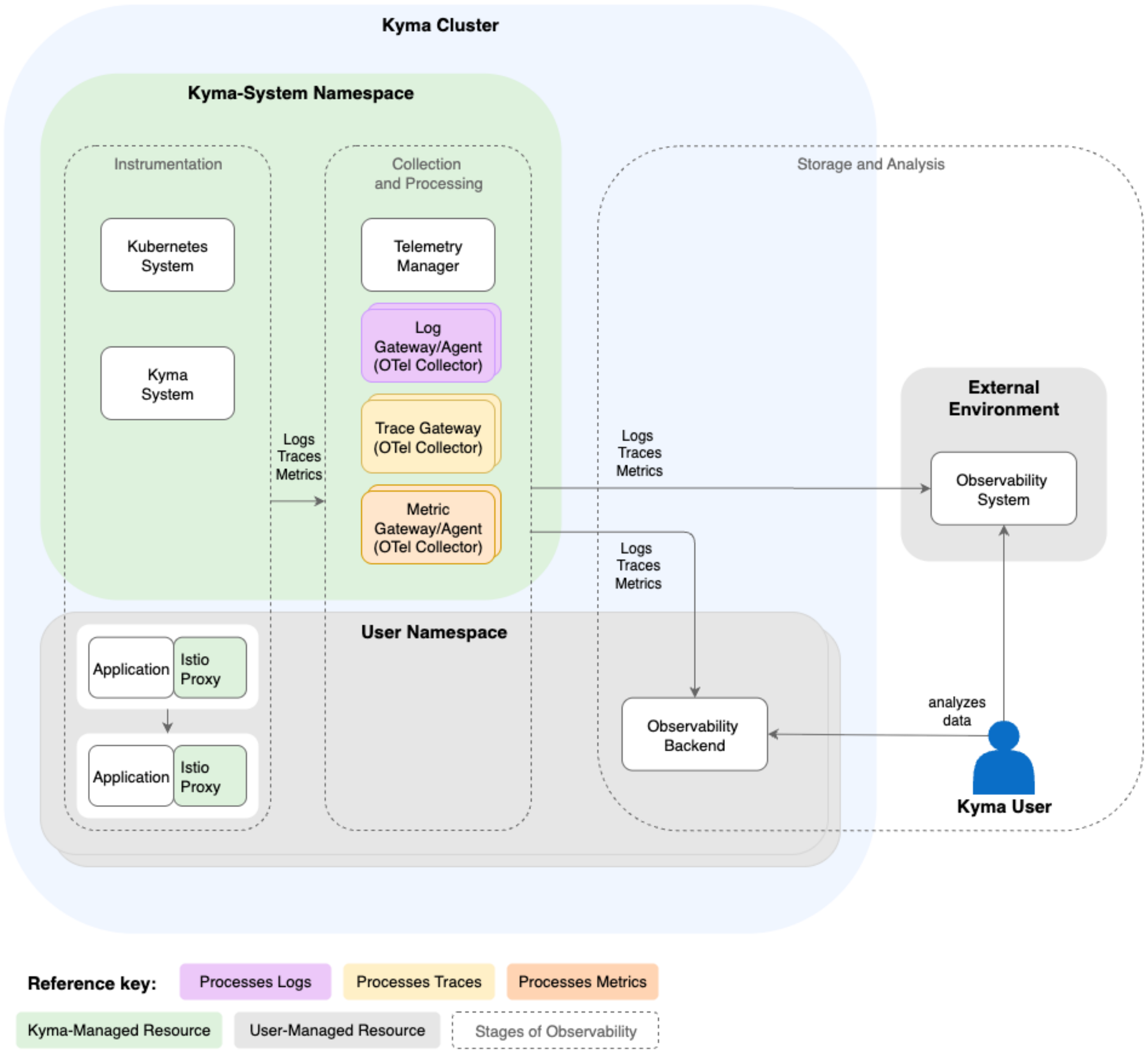
4.3.1.10.10 Telemetry Architecture

The Telemetry module consists of a manager component, which continuously watches the user-provided pipeline resources and deploys the respective OTel Collectors. Learn more about the architecture and how the components interact.

Overview

The Telemetry API provides a robust, preconfigured OpenTelemetry (OTel) Collector setup that abstracts its underlying complexities. This approach delivers several key benefits:

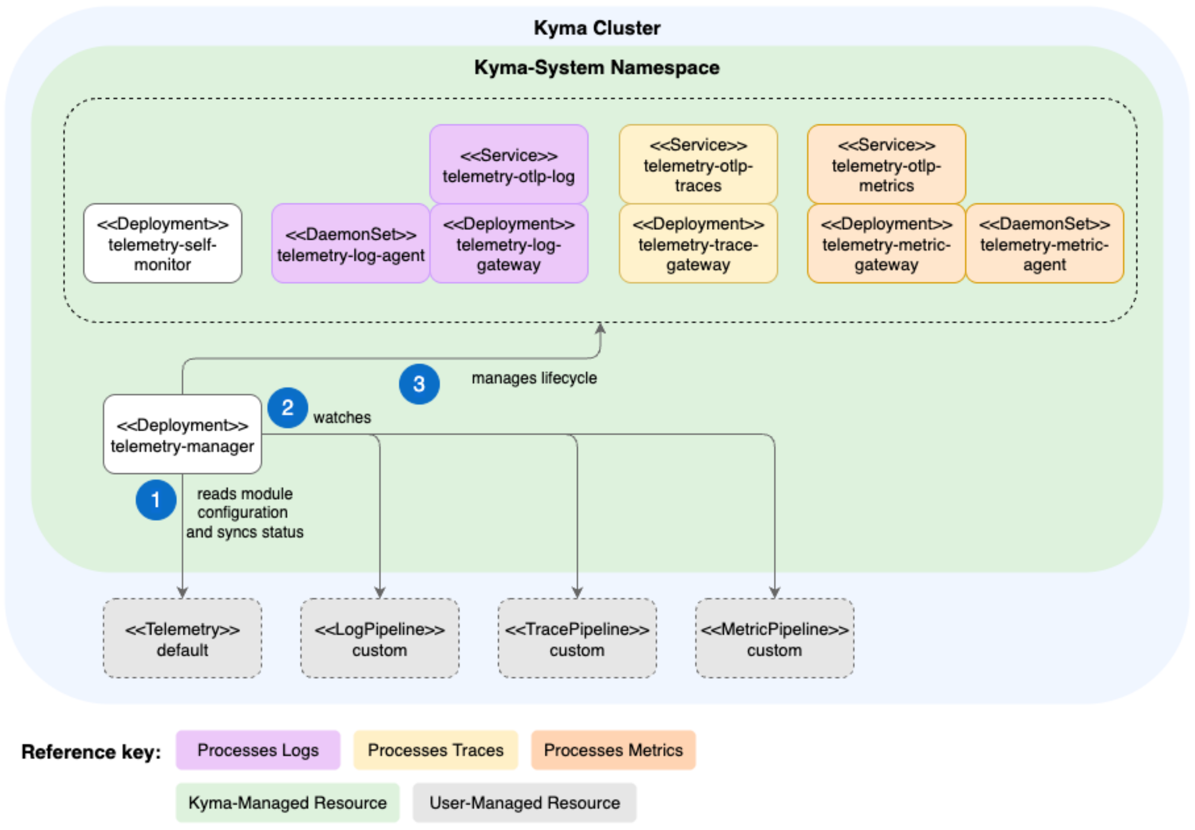
- Compatibility: Maintains stability and functionality even as underlying OTel Collector features evolve, reducing the need for constant updates on your end.
- Migratability: Facilitates smooth transitions when you switch underlying technologies or architectures.
- Native Kubernetes Support: Offers seamless integration with Secrets, for example, served by the SAP BTP Service Operator, and the Telemetry Manager automatically handles the full lifecycle of all components.
- Focus: Reduces the need to understand intricate underlying OTel Collector concepts, allowing you to focus on your application development.



Telemetry Manager

Telemetry Manager, the core component of the module, is a Kubernetes operator that implements the Kubernetes controller pattern and manages the whole lifecycle of all other Telemetry components. It performs the following tasks:

1. Watch the module configuration for changes and sync the module status to it.
2. Watch the user-created Kubernetes resources `LogPipeline`, `TracePipeline`, and `MetricPipeline`. In these resources, you specify what data of a signal type to collect and where to ship it.
3. Manage the lifecycle of the self monitor and the user-configured agents and gateways. For example, only if you defined a `LogPipeline` resource, the log gateway is deployed.


Kyma Cluster
Kyma-System Namespace
<<Service>> telemetry-otlp-log
<<Service>> telemetry-otlp-traces
<<Service>> telemetry-otlp-metrics
<<Deployment>> telemetry-self-monitor
<<DaemonSet>> telemetry-log-agent
<<Deployment>> telemetry-log-gateway
<<Deployment>> telemetry-trace-gateway
<<Deployment>> telemetry-metric-gateway
<<DaemonSet>> telemetry-metric-agent
3 manages lifecycle
<<Deployment>> telemetry-manager
2 watches
1 reads module configuration and syncs status
<<Telemetry>> default
<<LogPipeline>> custom
<<TracePipeline>> custom
<<MetricPipeline>> custom
Reference key:
Processes Logs
Processes Traces
Processes Metrics
Kyma-Managed Resource
User-Managed Resource

Gateways and Agents

Gateways and agents handle the incoming telemetry data. The Telemetry Manager deploys them based on your pipeline configuration.

The gateways are based on an OTel Collector Deployment and act as central endpoints in the cluster to which your applications push data in the OTLP format. From here, the data is enriched and filtered, and then dispatched configured in your pipeline resources.

Agents run as DaemonSet and pull data from the respective Node.

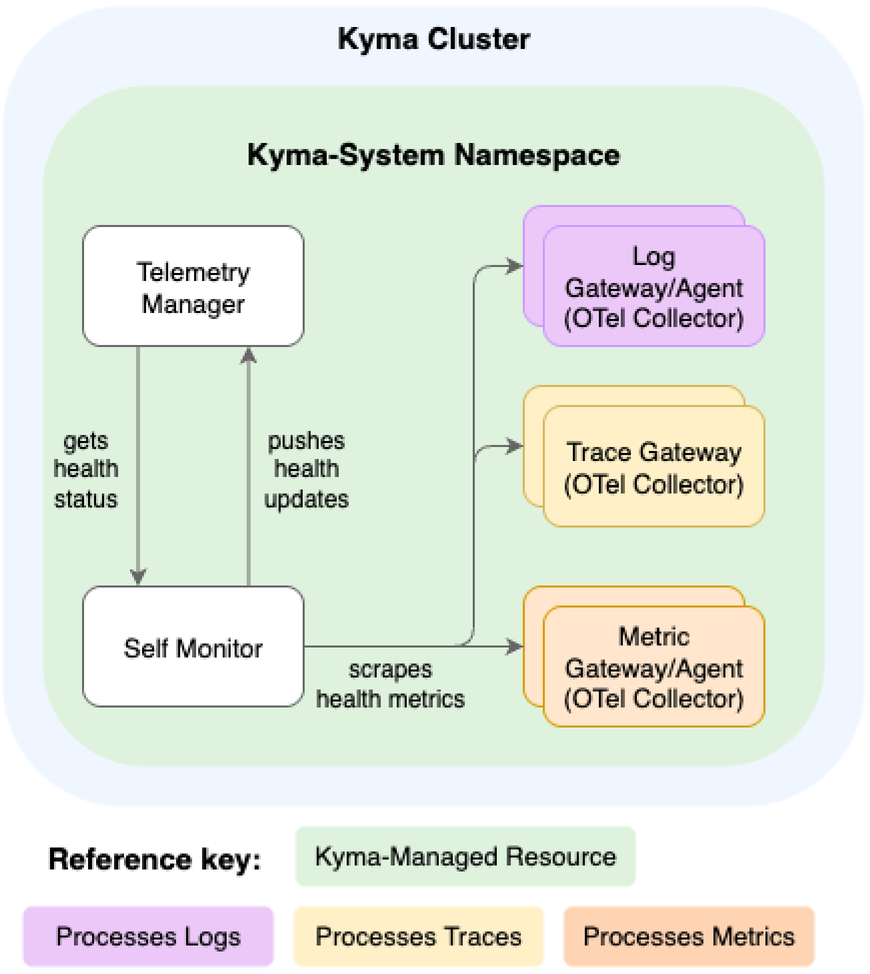
- **Log Gateway and Agent**
  The log gateway provides a central OTLP endpoint for logs. You can also enable the log agent, which collects logs from the `stdout/stderr` output of all containers on a Node. For details, see Logs Architecture [page 2078].
  As an alternative to the OTLP-based log feature, you can choose using a log agent based on a Fluent Bit installation running as a DaemonSet. It reads all containers' logs in the runtime and ships them according to your `LogPipeline` configuration. For details, see Application Logs (Fluent Bit).
- **Trace Gateway**
  The trace gateway provides a central OTLP endpoint to which your applications can push the trace signals. Kyma modules like Istio or Serverless contribute traces transparently. For details, see Traces Architecture [page 2081].
- **Metric Gateway and Agent**

The metric gateway provides a central OTLP endpoint for metrics. You can also enable the metric agent, which scrapes Prometheus-annotated workloads on each Node. For details, see Metrics Architecture [page 2083].

Self Monitor

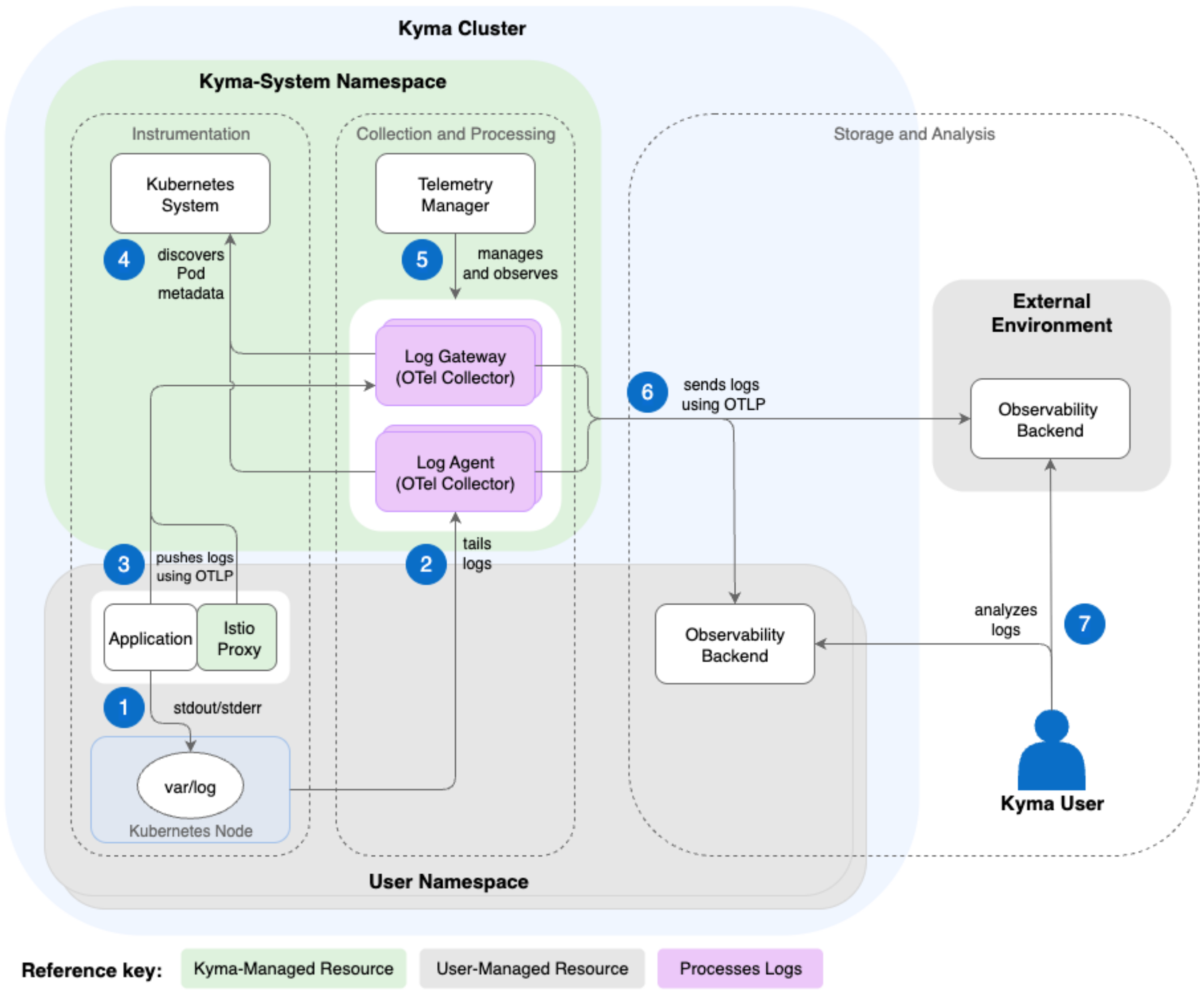
The Telemetry module includes a Prometheus -based self monitor that collects and evaluates health metrics from the gateways and agents. Telemetry Manager uses this data to report the current health status in your pipeline resources.

You can also use these health metrics in your own observability backend to set up alerts and dashboards for your telemetry pipelines. For details, see Monitor Pipeline Health [page 2069].



4.3.1.10.10.1 Logs Architecture

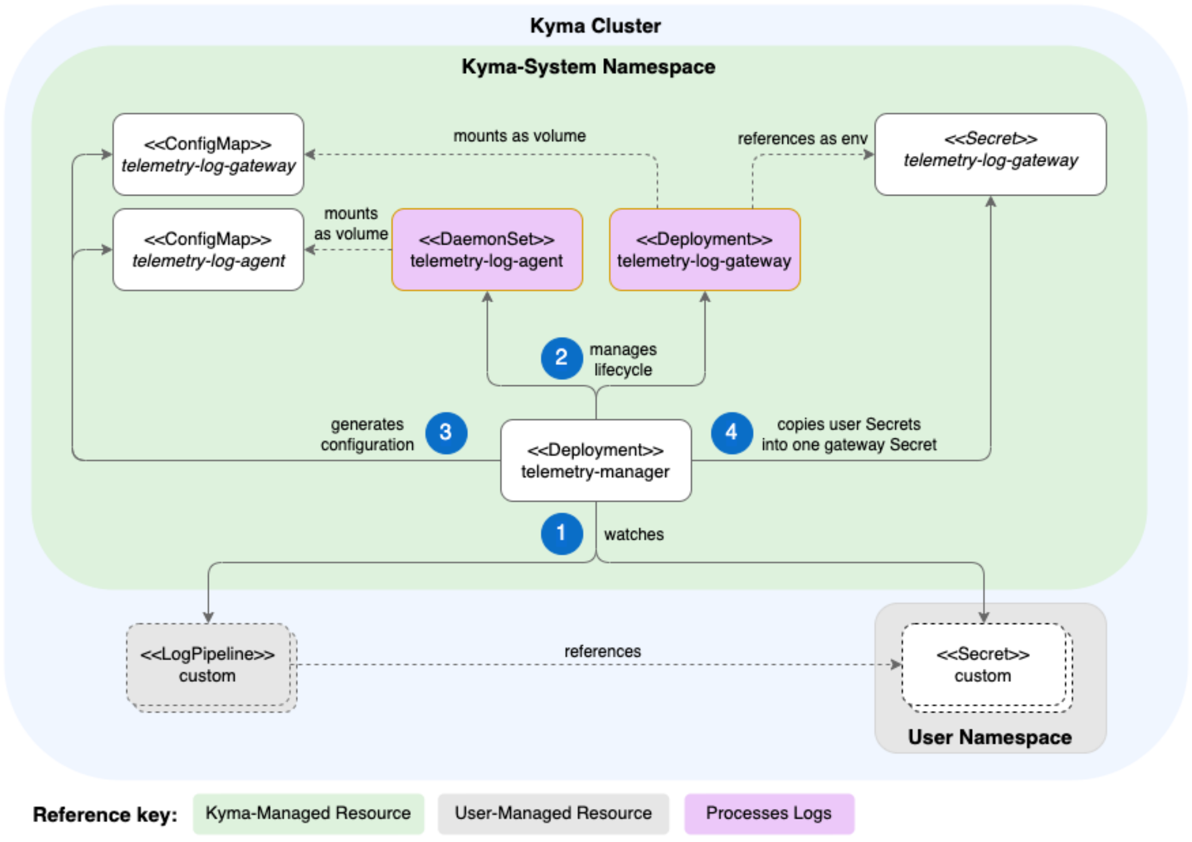
The Telemetry module provides a central Deployment of an OTel Collector acting as a gateway, and an optional DaemonSet acting as an agent. The gateway exposes endpoints that receive OTLP logs from your applications, while the agent collects container logs from each node. To control their behavior and data destination, you define a `LogPipeline`.



1. Application containers print JSON logs to the `stdout/stderr` channel and are stored by the Kubernetes container runtime under the `var/log` directory and its subdirectories at the related Node. Istio is configured to write access logs to `stdout` as well.
2. If you choose to use the agent, an OTel Collector runs as a DaemonSet (one instance per Node), detects any new log files in the folder, and tails and parses them.
3. An application (exposing logs in OTLP) sends logs to the central log gateway using the `telemetry-otlp-logs` service. Istio is configured to push access logs with OTLP as well.
4. The gateway and agent discover the metadata and enrich all received data with metadata of the source by communicating with the Kubernetes APIServer. Furthermore, they filter data according to the pipeline configuration.
5. Telemetry Manager configures the agent and gateway according to the `LogPipeline` resource specification, including the target backend. Also, it observes the logs flow to the backend and reports problems in the LogPipeline status.
6. The log agent and gateway send the data to the observability backend that's specified in your `LogPipeline` resource - either within your cluster, or, if authentication is set up, to an external observability backend.
7. You can analyze the logs data with your preferred observability backend.

Telemetry Manager

The `LogPipeline` resource is watched by Telemetry Manager, which is responsible for generating the custom parts of the OTel Collector configuration.


Kyma Cluster
Kyma-System Namespace
<<ConfigMap>> telemetry-log-gateway
<<ConfigMap>> telemetry-log-agent
mounts as volume
references as env
<<Secret>> telemetry-log-gateway
mounts as volume
<<DaemonSet>> telemetry-log-agent
<<Deployment>> telemetry-log-gateway
2 manages lifecycle
3 generates configuration
<<Deployment>> telemetry-manager
4 copies user Secrets into one gateway Secret
1 watches
<<LogPipeline>> custom
references
<<Secret>> custom
User Namespace
Reference key:
Kyma-Managed Resource
User-Managed Resource
Processes Logs

1. Telemetry Manager watches all `LogPipeline` resources and related Secrets.
2. Furthermore, Telemetry Manager takes care of the full lifecycle of the gateway Deployment and the agent DaemonSet. Only if you defined a `LogPipeline`, the gateway and agent are deployed.
3. Whenever the user configuration changes, Telemetry Manager validates it and generates a single configuration for the gateway and agent.
4. Referenced Secrets are copied into one Secret that is mounted to the gateway as well.

Log Gateway

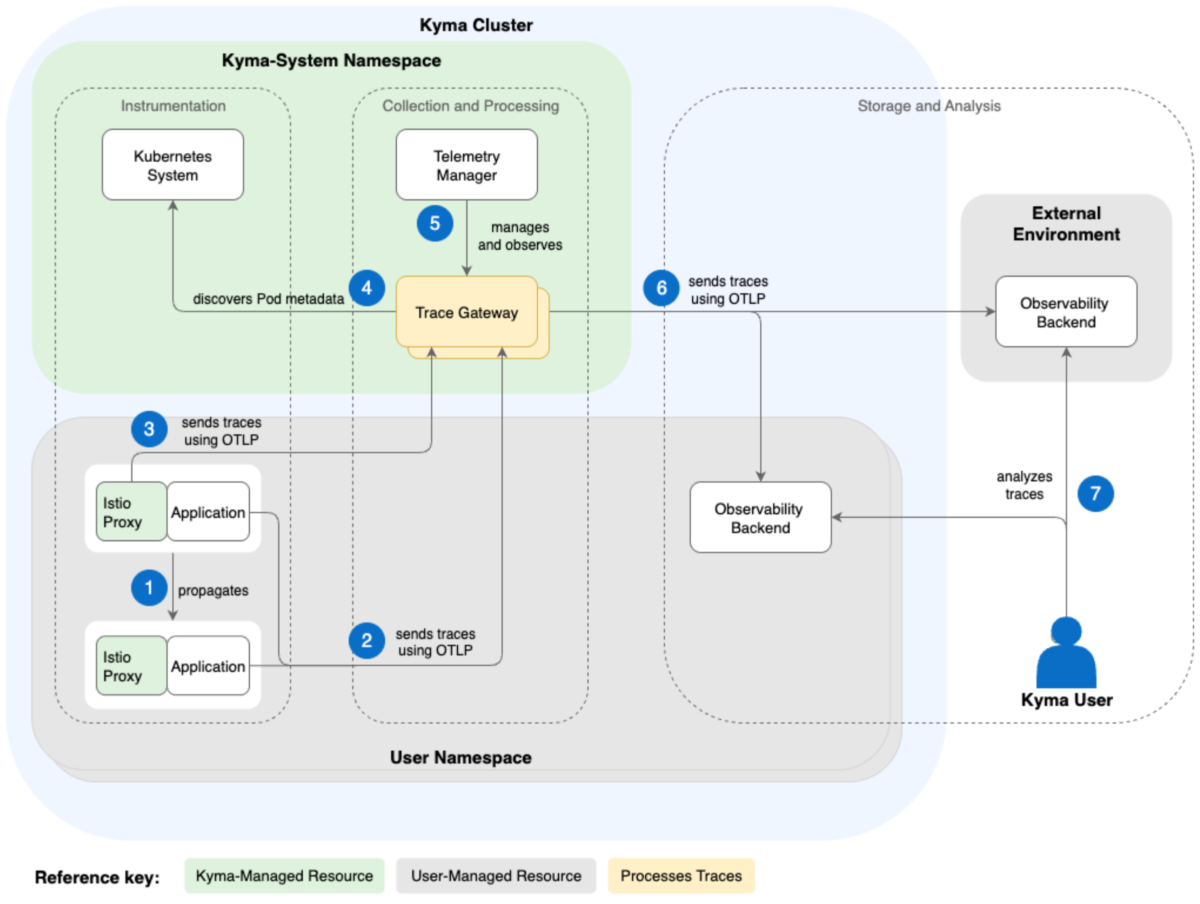
In your cluster, the log gateway is the central component to which all components can send their individual logs. The gateway collects, enriches, and dispatches the data to the configured backend. For more information, see Set Up the OTLP Input [page 2025].

Log Agent

If you configure a feature in the `input` section of your `LogPipeline`, an additional DaemonSet is deployed acting as an agent. The agent is based on an OTel Collector and encompasses the collection and conversion of logs from the container runtime. Hereby, the workload container just prints the structured log to the `stdout/stderr` channel. The agent picks them up, parses and enriches them, and sends all data in OTLP to the configured backend.

4.3.1.10.10.2 Traces Architecture

The Telemetry module provides a central Deployment of an OTel Collector acting as a gateway in the cluster. The gateway exposes endpoints that receive trace data from your applications and the service mesh. To control the gateway's behavior and data destination, you define a `TracePipeline`.

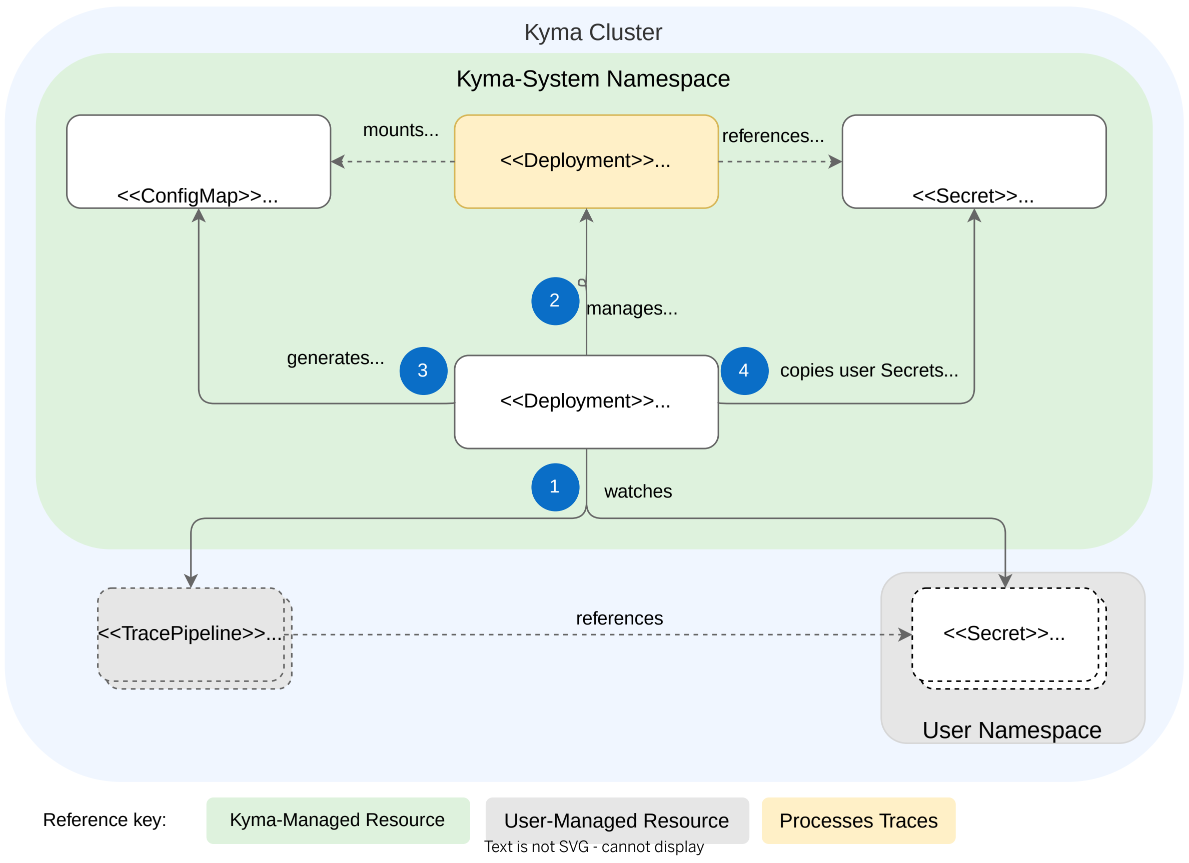


1. An end-to-end request is triggered and populated across the distributed application. Every involved component propagates the trace context using the W3C Trace Context protocol.
2. After contributing a new span to the trace, the involved components send the related span data (OTLP) to the central trace gateway using the `telemetry-otlp-traces` service.
3. Istio sends the related span data to the trace gateway as well.

4. The trace gateway discovers metadata that's typical for sources running on Kubernetes, like Pod identifiers, and then enriches the span data with that metadata.
5. Telemetry Manager configures the gateway according to the `TracePipeline` resource, including the target backend for the trace gateway. Also, it observes the trace flow to the backend and reports problems in the `TracePipeline` status.
6. The trace gateway sends the data to the observability backend that's specified in your `TracePipeline` resource - either within the cluster, or, if authentication is set up, to an external observability backend.
7. You can analyze the trace data with your preferred observability backend.

Telemetry Manager

The `TracePipeline` resource is watched by Telemetry Manager, which is responsible for generating the custom parts of the OTel Collector configuration.



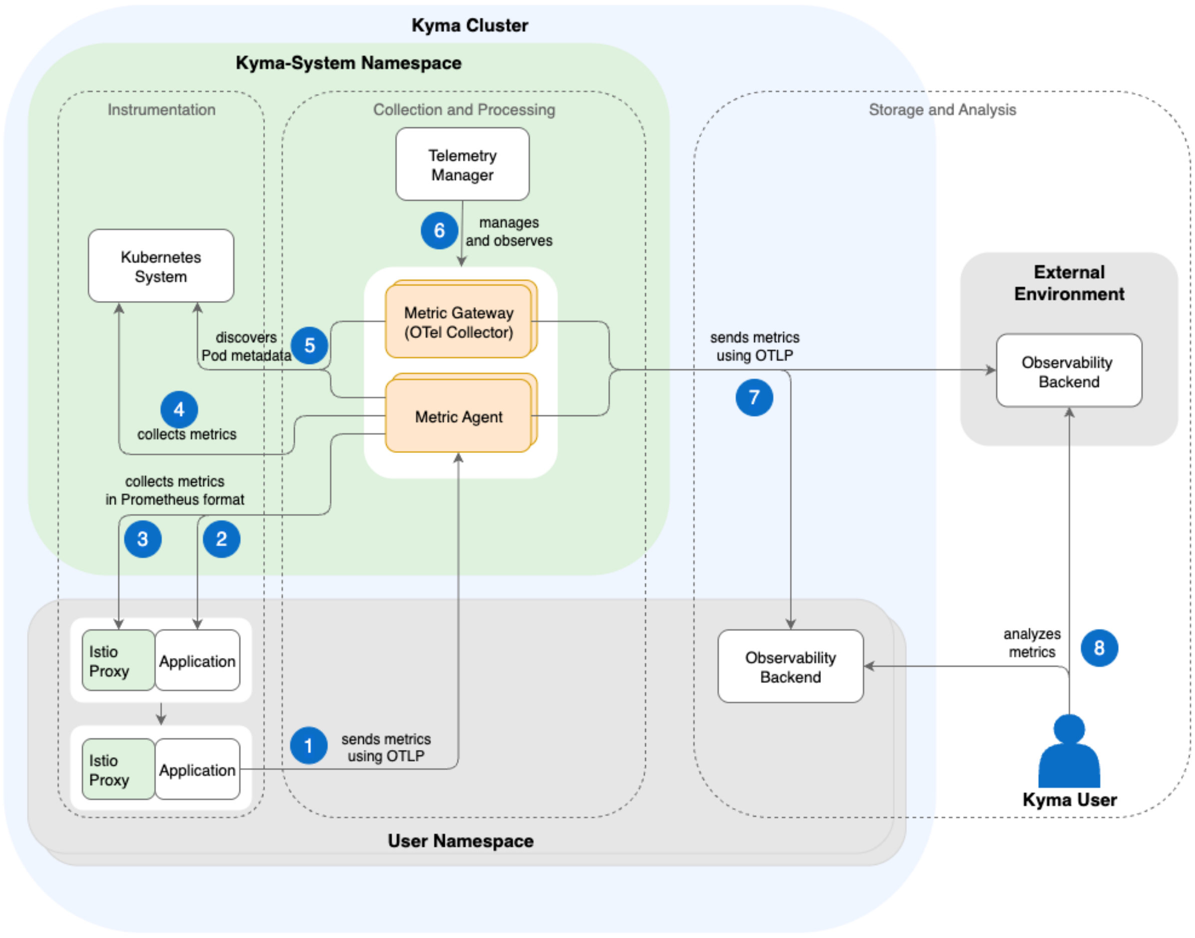
1. Telemetry Manager watches all `TracePipeline` resources and related Secrets.
2. Furthermore, Telemetry Manager takes care of the full lifecycle of the OTel Collector Deployment itself. Only if you defined a `TracePipeline`, the collector is deployed.
3. Whenever the configuration changes, it validates the configuration and generates a new configuration for OTel Collector, where a ConfigMap for the configuration is generated.
4. Referenced Secrets are copied into one Secret that is mounted to the OTel Collector as well.

Trace Gateway

In your cluster, the trace gateway is the central component to which all components can send their individual spans. The gateway collects, enriches, and dispatches the data to the configured backend. For more information, see Set Up the OTLP Input [page 2025].

4.3.1.10.10.3 Metrics Architecture

The Telemetry module provides a central Deployment of an OTel Collector acting as a gateway, and an optional DaemonSet acting as an agent. The gateway exposes endpoints that receive OTLP metrics from your applications, while the agent pulls metrics from Prometheus-annotated endpoints. To control their behavior and data destination, you define a `MetricPipeline`.



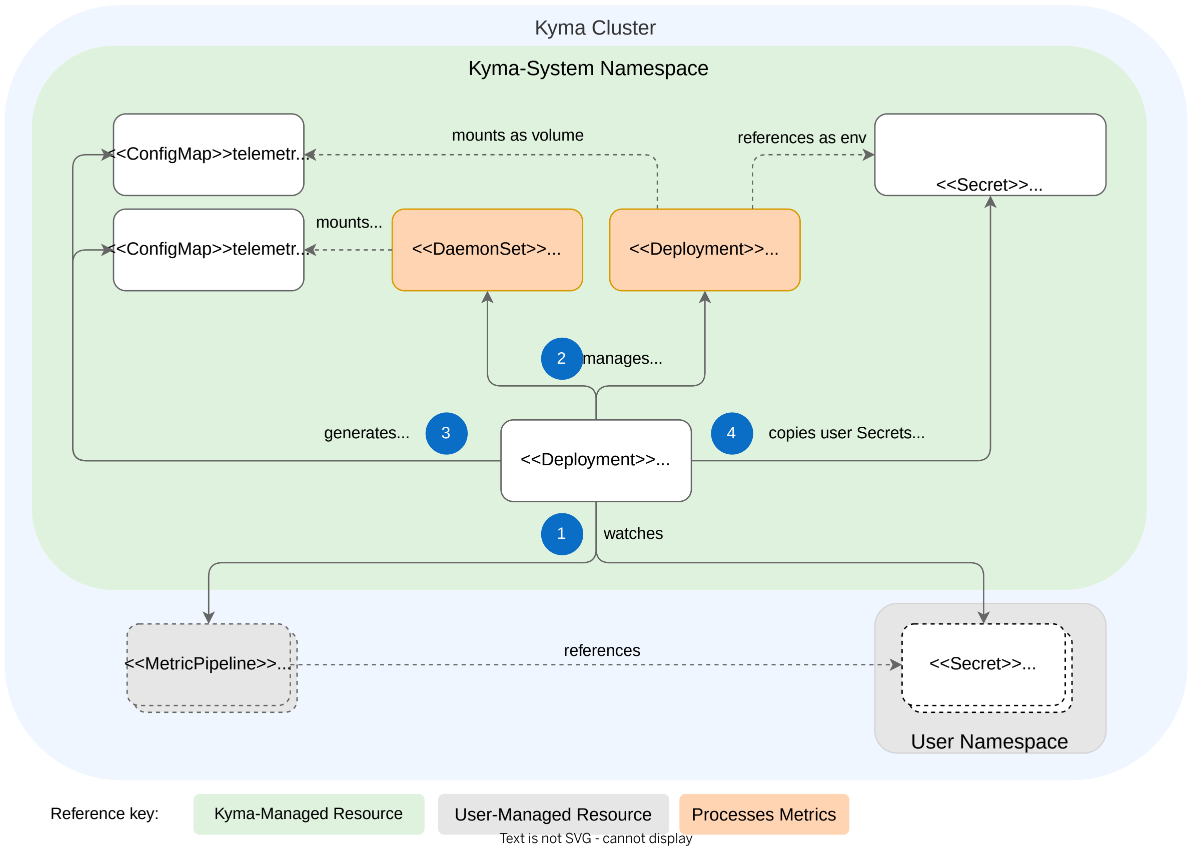
Reference key: 🟩 Kyma-Managed Resource ⬜ User-Managed Resource 🟧 Processes Metrics

1. An application (exposing metrics in OTLP) sends metrics to the central metric gateway using the `telemetry-otlp-metrics` service.
2. An application (exposing metrics in Prometheus protocol) activates the agent to scrape the metrics with an annotation-based configuration.

3. Additionally, you can activate the agent to pull metrics of each Istio sidecar.
4. The agent supports collecting metrics from the Kubelet and Kubernetes APIServer.
5. The gateway and the agent discover the metadata and enrich all received data with typical metadata of the source by communicating with the Kubernetes APIServer. Furthermore, they filter data according to the pipeline configuration.
6. Telemetry Manager configures the agent and gateway according to the `MetricPipeline` resource specification, including the target backend for the metric gateway. Also, it observes the metrics flow to the backend and reports problems in the `MetricPipeline` status.
7. The gateway and the agent send the data to the observability backend that's specified in your `MetricPipeline` resource - either within your cluster, or, if authentication is set up, to an external observability backend.
8. You can analyze the metric data with your preferred observability backend.

Telemetry Manager

The `MetricPipeline` resource is watched by Telemetry Manager, which is responsible for generating the custom parts of the OTel Collector configuration.



1. Telemetry Manager watches all `MetricPipeline` resources and related Secrets.
2. Furthermore, Telemetry Manager takes care of the full lifecycle of the gateway Deployment and the agent DaemonSet. Only if you defined a `MetricPipeline`, the gateway and agent are deployed.
3. Whenever the user configuration changes, Telemetry Manager validates it and generates a single configuration for the gateway and agent.

4. Referenced Secrets are copied into one Secret that is mounted to the gateway as well.

Metric Gateway

In your cluster, the metric gateway is the central component to which all applications can send their individual metrics. The gateway collects, enriches, and dispatches the data to the configured backend. For more information, see Set Up the OTLP Input [page 2025].

Metric Agent

If a `MetricPipeline` configures a feature in the `input` section, an additional DaemonSet is deployed acting as an agent. The agent is also based on an OTel Collector and encompasses the collection and conversion of Prometheus-based metrics. Hereby, the workload puts a `prometheus.io/scrape` annotation on the specification of the Pod or service, and the agent collects it.

4.3.1.10.10.4 Istio Integration

When you have the Istio module in your cluster, the Telemetry module automatically integrates with it. It detects the Istio installation and injects sidecars into the Telemetry components, adding them to the service mesh. This enables secure mTLS communication for your Telemetry pipelines by default.

Receiving Data from Your Applications

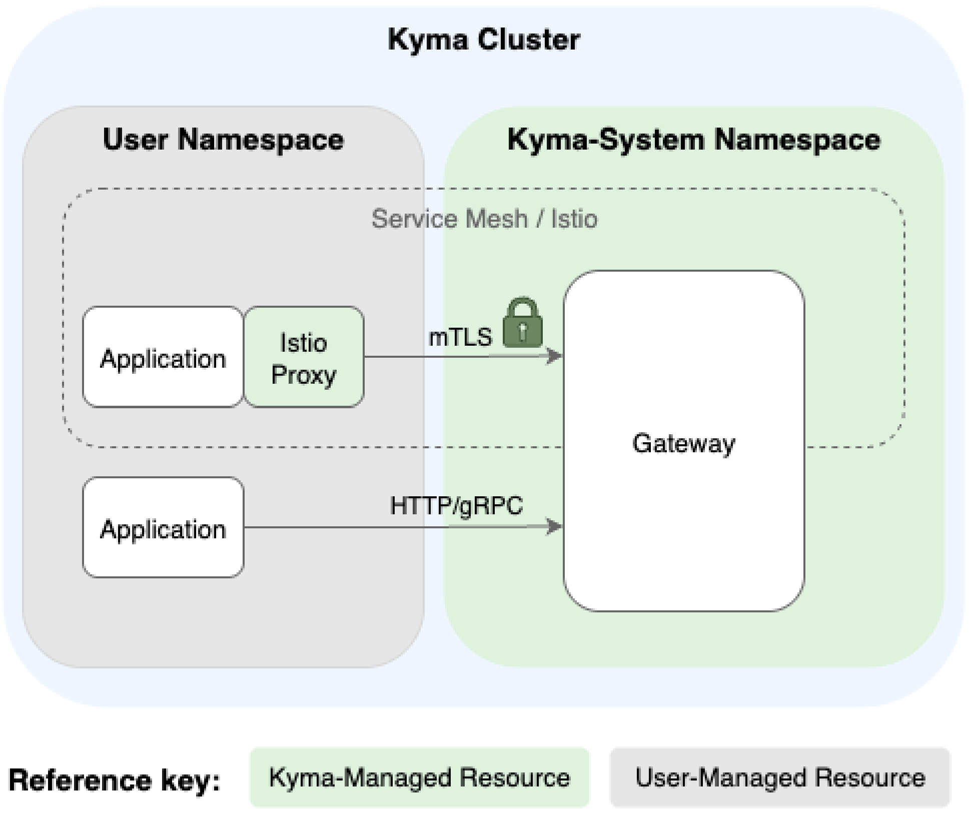
The Telemetry gateways are automatically configured to accept OTLP data from applications both inside and outside the Istio service mesh. To achieve this, the ingestion endpoints of gateways are set to Istio's permissive mode, so they accept mTLS-based communication as well as plain text.

- Applications within the mesh automatically send data to the gateways using mTLS for a secure, encrypted connection.
- Applications outside the mesh can send data to the gateway using a standard plain text connection.

→ Tip

Learn more about Istio-specific input configuration for logs, traces, and metrics:

- Configure Istio Access Logs [page 2030]
- Configure Istio Tracing [page 2034]
- Collect Istio Metrics [page 2041]

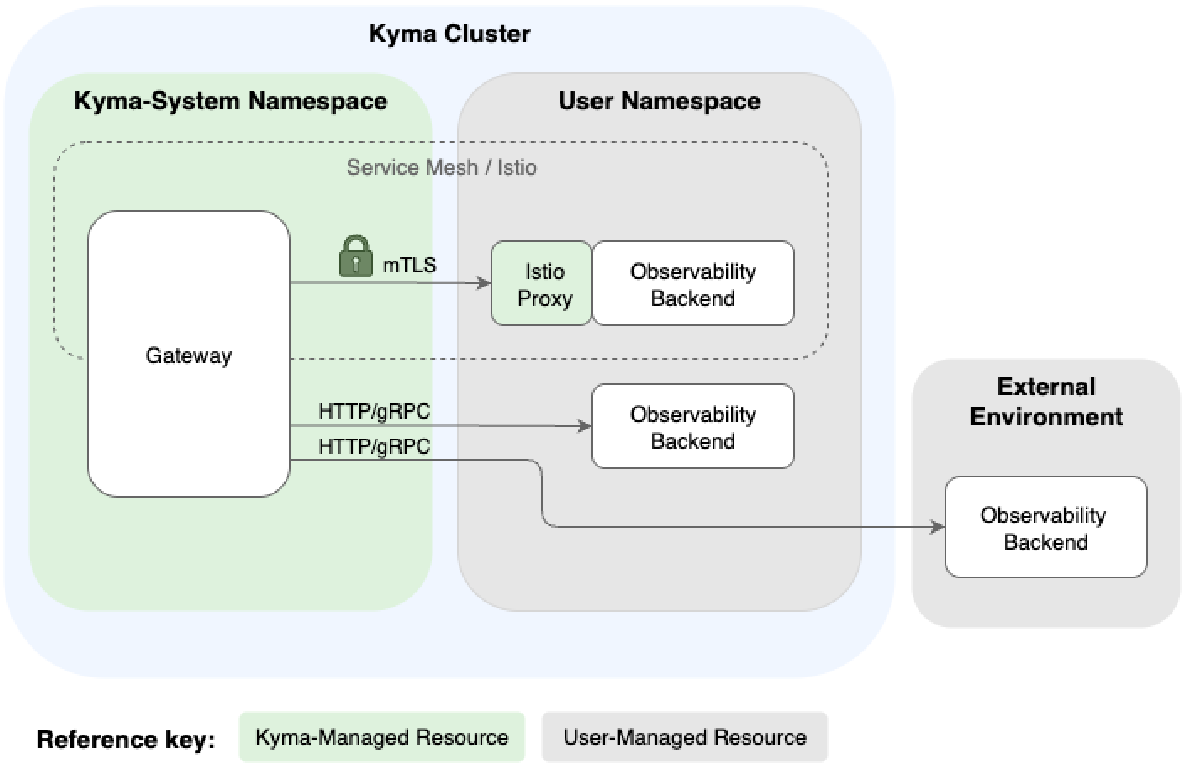


Sending Data to In-Cluster Backends

Telemetry gateways automatically secure the connection when sending data to your observability backends.

If you're using an in-cluster backend that is part of the Istio mesh, the Telemetry gateways automatically use mTLS to send data to the backend securely. You don't need any special configuration for this.

For sending data to backends outside the cluster, see Integrate With Your OTLP Backend [page 2056].



4.3.2 Access a Kyma Instance Using kubectl

As an alternative to managing Kyma with a graphical user interface, Kyma dashboard, you can also use the Kubernetes command-line tool, kubectl.

Prerequisites

You have a Kyma instance created in your subaccount of the SAP BTP cockpit.