RTI Connext Observability Framework

User's Manual

Version 7.3.0

Contents

Chapter 1

Copyrights and Notices

Trademarks

Copy and Use Restrictions

Third-Party Software

Notices

Deprecations and Removals

Any deprecations or removals noted in this document serve as notice under the Real-Time Innovations, Inc. Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software.

Deprecated means that the item is still supported in the release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported. By specifying that an item is deprecated in a release, RTI hereby provides customer notice that RTI reserves the right after one year from the date of such release and, with or without further notice, to immediately terminate maintenance (including without limitation, providing updates and upgrades) for the item, and no longer support the item, in a future release.

Technical Support Real-Time Innovations, Inc. 232 E. Java Drive Sunnyvale, CA 94089 Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/

Chapter 2

What is Connext Observability Framework?

RTI® Connext® Observability Framework is a holistic solution that uses telemetry data to provide deep visibility into the current and past states of your *Connext* applications. This visibility makes it easier to proactively identify and resolve potential system issues, providing a higher level of confidence in the reliable operation of the system.

Observability Framework use cases include:

- **Debugging**. Find the cause of an undesired behavior, or determine if the feature meets performance needs during development.
- **CI/CD monitoring**. Assess the performance impact of code or configuration changes.
- **Monitoring deployed applications**. Confirm that your systems are running as expected and proactively fix potential performance issues.

Important: *Observability Framework* is an experimental product that includes example configuration files for use with several third-party components (Prometheus®, Grafana Loki™, and Grafana®, NGINX®, and OpenTelemetry™ Collector). This release is an evaluation distribution; use it to explore the new observability features that support *Connext* applications. For support, you may contact support@rti.com.

Do not deploy any Observability Framework components in production. A production-ready version is expected to be available in a future *Connext* 7.3.x maintenance release.

2.1 Telemetry Data

Telemetry data can be generated at three different levels:

- **Application**. Telemetry data generated when you instrument your own applications.
- **Middleware**. Telemetry data generated by *Connext* DDS entities and infrastructure services.
- **System**. DevOps telemetry such as CPU, memory, and disk I/O usage.

In this release, *Observability Framework* supports middleware telemetry (metrics and logs) and application logs. Future releases could support application metrics and system telemetry.

Regardless of the level, telemetry data can be categorized as:

- **Metrics**. Collections of application statistics that are analyzed to understand application behavior. There are two types of metrics:
  - Counters count the number of events of a specific type; for example, the number of ACK messages sent.
  - Gauges describe the state of some part of an application as a numeric value within a specified time frame; for example, the number of samples in a queue.
- **Logs**. Events captured as text or structured data.
- **Security Events**. Events related to securing a distributed system.
  - Notification of **Security Events** in *Observability Framework* are communicated as **Logs** with a Syslog Facility of **SECURITY_EVENT**. See *Logs* for more information.
- **Traces**. A representation of a series of causally-related events that encode the end-to-end flow of a piece of information in a software system. The traces in a distributed system are called *distributed traces*.

In this release, *Observability Framework* supports metrics, logs, and security events. Future releases could support traces. See *Telemetry Data* for more information.

2.2 Distribution of Telemetry Data

Observability Framework enables you to scalably generate and forward telemetry data from individual *Connext* applications to third-party telemetry backends like Prometheus and Grafana Loki. For more information on the distribution of telemetry data see *Monitoring Library 2.0* and *Observability Collector Service*.

2.3 Flexible Storage

Observability Framework provides native integration with Prometheus as the time-series database to store *Connext* metrics and Grafana Loki as the log aggregation system to store *Connext* logs. Integration with other backends is possible through the use of OpenTelemetry and the OpenTelemetry Collector.

2.4 Visualization of Telemetry Data

In this release, *Observability Framework* provides a way to visualize the telemetry data collected from *Connext* applications using a set of Grafana dashboards. You can customize these dashboards or use them as an example to enhance and build dashboards in your preferred platform.

The *Observability Dashboards* only work with the Prometheus and Grafana Loki backends. Future releases could support other backends. For more information, see *Observability Dashboards*.

2.5 Control and Selection of Telemetry Data

Your distributed system components can produce a large amount of data, but not all of this data is required for problem detection. *Observability Framework* enables you to control the amount of telemetry data that is generated, forwarded, and stored. You can manage these settings at run-time and via an initial configuration.

See *Setting the Initial Metrics and Log Configuration* for information on the initial configuration of telemetry collection. See *Collector Service REST API Reference* for information on remote commands provided by the *Observability Collector Service* to support changing the configuration of telemetry collection at run-time. See *Change the Application Logging Verbosity* and *Change the Metric Configuration* for examples of how *Observability Framework* provides the ability to change the configuration of telemetry collection at run-time.

2.6 Security

Observability Framework provides a way to secure the telemetry data generated by the *Connext* applications and stored in the telemetry backends. Data in transit is secured by using the SECURITY PLUGINS *(RTI Security Plugins)* and BASIC-Auth over HTTPS. Data at rest is secured by the third-party telemetry backends. For more information see *Security*.

Chapter 3

Components

Connext Observability Framework consists of three RTI components:

- *RTI Monitoring Library 2.0* enables you to instrument a *Connext* application to forward telemetry data. The library also accepts remote commands to change the set of forwarded telemetry data at runtime.
- *RTI Observability Collector Service* scalably collects telemetry data from multiple *Connext* applications and stores this data in a third-party observability backend. This component can also be configured to forward telemetry data to an OpenTelemetry Collector to allow integration with other third-party observability backends.
- *RTI Observability Dashboards* enable you to visualize and alert based on the *Connext* application metrics, as well as display *Connext* log messages.

Observability Framework requires third-party components for storing and visualizing telemetry data. This release provides native integration with Prometheus for metrics storage, Grafana Loki for logs storage, and Grafana for visualization. Integration with other third-party components is also possible when using OpenTelemetry and the OpenTelemetry Collector.

Observability Dashboards are provided as a set of Grafana dashboards to be deployed on a Grafana server. These dashboards only work with the Prometheus and Grafana Loki backends. Future releases could support other backends.

Figure 3.1 shows a simple representation of how *Observability Framework* components work together.

3.1 Monitoring Library 2.0

Monitoring Library 2.0 includes the following key features:

- Collection and forwarding of *Connext* metrics and logs (including security event logs).
- Configuration using a new MONITORING QosPolicy (DDS Extension). The QoS policy can be set programmatically or via XML.
- Runtime changes to the collection and forwarding of telemetry data using remote commands from *Observability Collector Service*.

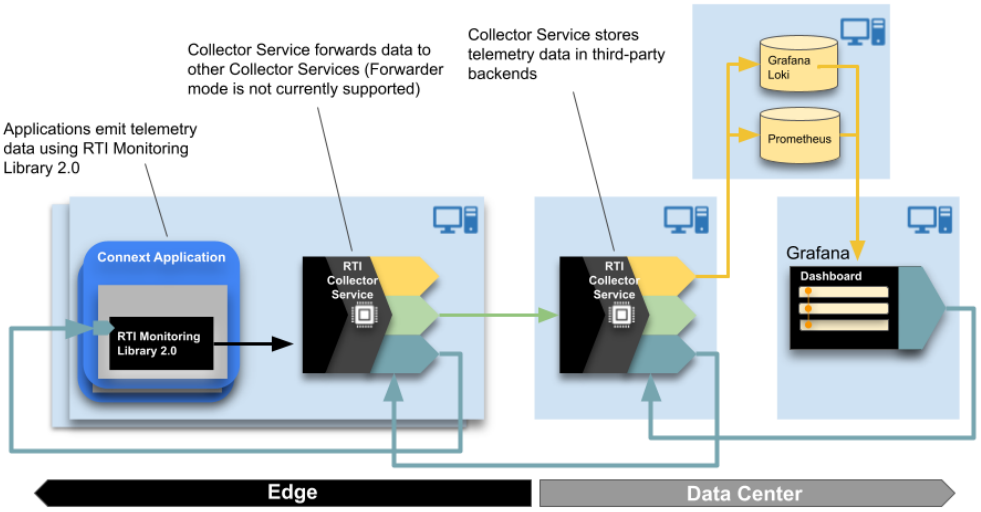


Figure 3.1: Observability Framework Components

- Ability to enable and disable use of *Monitoring Library 2.0* at runtime by changing the Monitoring QoS policy.
- Lower overhead as compared to using the RTI Monitoring Library.

For more information, see *Monitoring Library 2.0*.

3.2 Observability Collector Service

Observability Collector Service scalably collects telemetry data forwarded by *Monitoring Library 2.0* in a *Connext* application. *Collector Service* is distributed as a Docker™ image. For additional information on this image see Dockerhub. The *Observability Collector Service* is designed to work in two modes:

- **Storage**: *Collector Service* sends the telemetry data for storage to third-party observability backends. This release provides native integration with Prometheus for metrics and Grafana Loki for logs. Integration with other third-party components is also possible using OpenTelemetry and the OpenTelemetry Collector.
- **Forwarder**: *Collector Service* forwards the telemetry data from *Connext* applications to another collector instance. This mode is not supported in the current release.

Observability Collector Service includes the following key features:

- Collecting and filtering telemetry data forwarded by *Connext* applications (using *Monitoring Library 2.0*) or other collectors. This release does not provide filtering capabilities.
- Sending telemetry data for storage to Prometheus for metrics and Grafana Loki for logs.
- Ability to send telemetry data to an OpenTelemetry Collector using the OpenTelemetry protocol (OTLP). This feature enables integration with third-party observability backends other than Prometheus and

Grafana Loki.

- Remote command forwarding from *Observability Dashboards* to the *Connext* applications and other resources to which the commands are directed. Remote commands may be used to control the forwarding of log messages and metrics. For detailed information on the commands supported see *Collector Service REST API Reference*.

3.2.1 Storage Components

Observability Collector Service includes native integration with Prometheus and Grafana Loki to store metrics and logs, respectively.

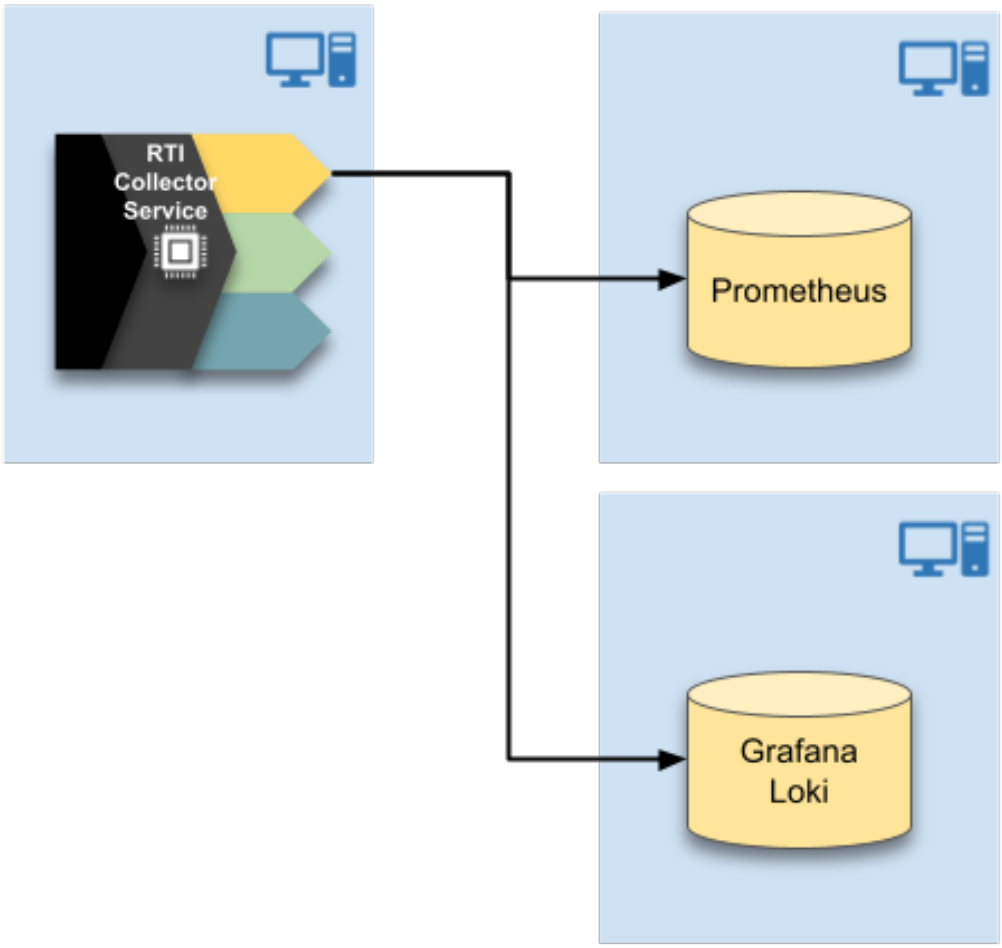


Figure 3.2: Native Integration

This release also allows integrating with other third-party storage components using OpenTelemetry and the OpenTelemetry Collector.

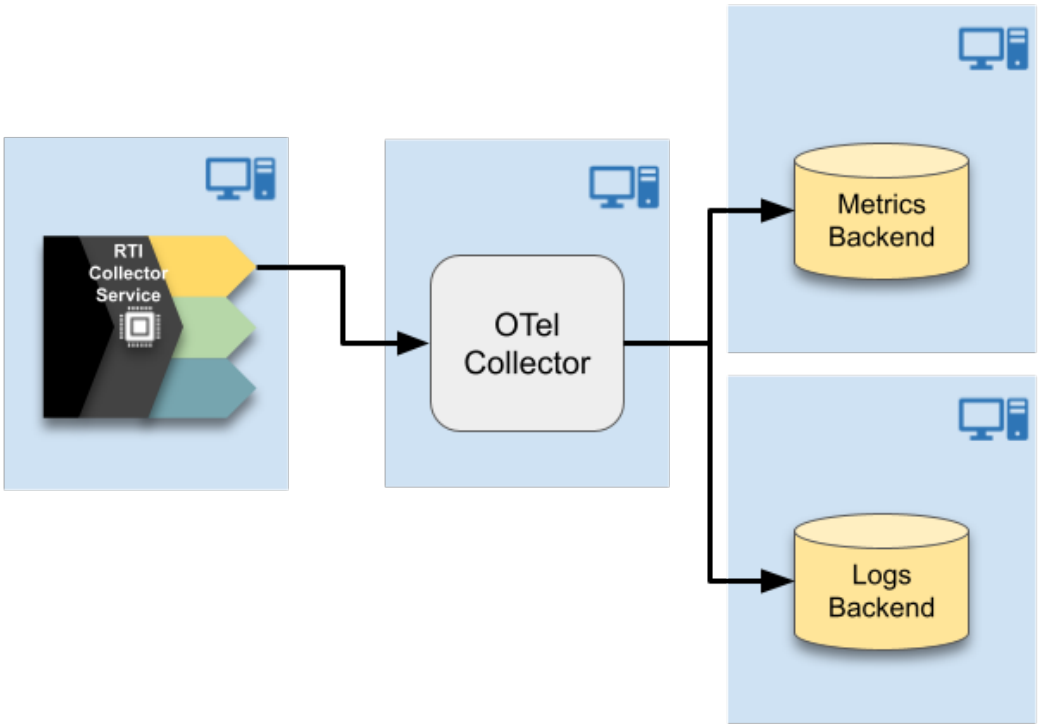


Figure 3.3: OpenTelemetry Integration

3.3 Observability Dashboards

A set of hierarchical Grafana dashboards displays alerts when a problem occurs and provides visualizations to help perform root cause analysis. The dashboards get the telemetry data from a Prometheus server and the logs from a Grafana Loki server.

See *Observability Dashboards* for more information on the *Observability Dashboards*.

Chapter 4

Observability Framework Deployments

This section describes how to deploy the components of the *Observability Framework* in the current release. Additionally, it discusses how RTI intends to introduce new deployment methods for the *Observability Framework* in future releases.

Monitoring Library 2.0 is component is included with *Connext Professional* as a shared and static library called rtimonitoring2. For details on how to use the library, refer to *Monitoring Library 2.0*. For further information on the other components, please see the following sections.

4.1 Current Release

4.1.1 Docker Compose (Prepackaged)

Collection, Storage, and Visualization Components

The *Observability Framework* package enables you to deploy and run *Observability Collector Service* and third-party components Prometheus, Grafana Loki, Grafana, OpenTelemetry Collector (optional), and NGINX (optional) using Docker Compose™ in a single Linux® host. For details, see *Supported Docker Compose Environments*.

RTI's prepackaged Docker Compose installation option facilitates initial product evaluation because it does not require you to deploy all these components individually.

Observability Framework can be deployed with or without using the OpenTelemetry Collector. Both deployment options can be configured to be secure or non-secure and to work on a LAN or WAN.

Figure 4.1 *RTI Observability Framework without OpenTelemetry Collector* shows the secure *Observability Framework* deployment without OpenTelemetry Collector. The deployment uses Prometheus and Grafana Loki to store metrics and logs, respectively.

Figure 4.2 *RTI Observability Framework with OpenTelemetry Collector* shows a secure *Observability Framework* deployment using OpenTelemetry Collector. The deployment uses OpenTelemetry Collector to store metrics and logs in Prometheus and Grafana Loki, respectively.

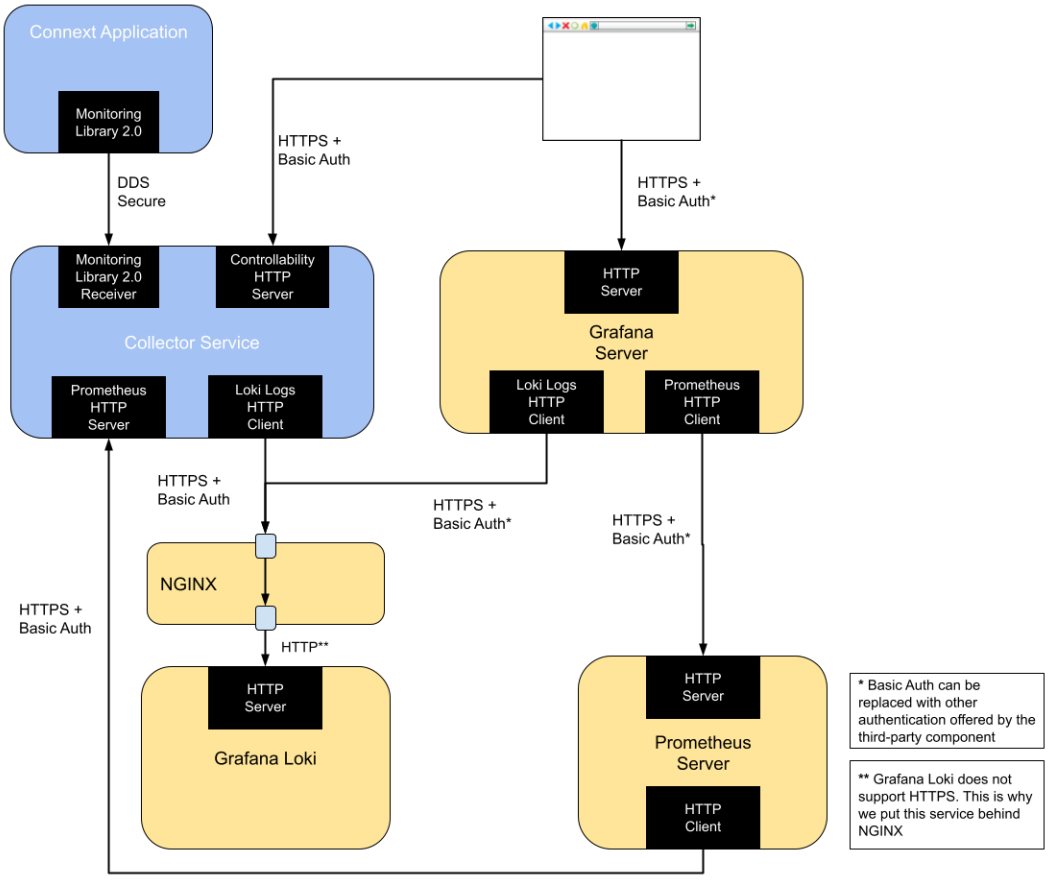


Figure 4.1: RTI Observability Framework without OpenTelemetry Collector

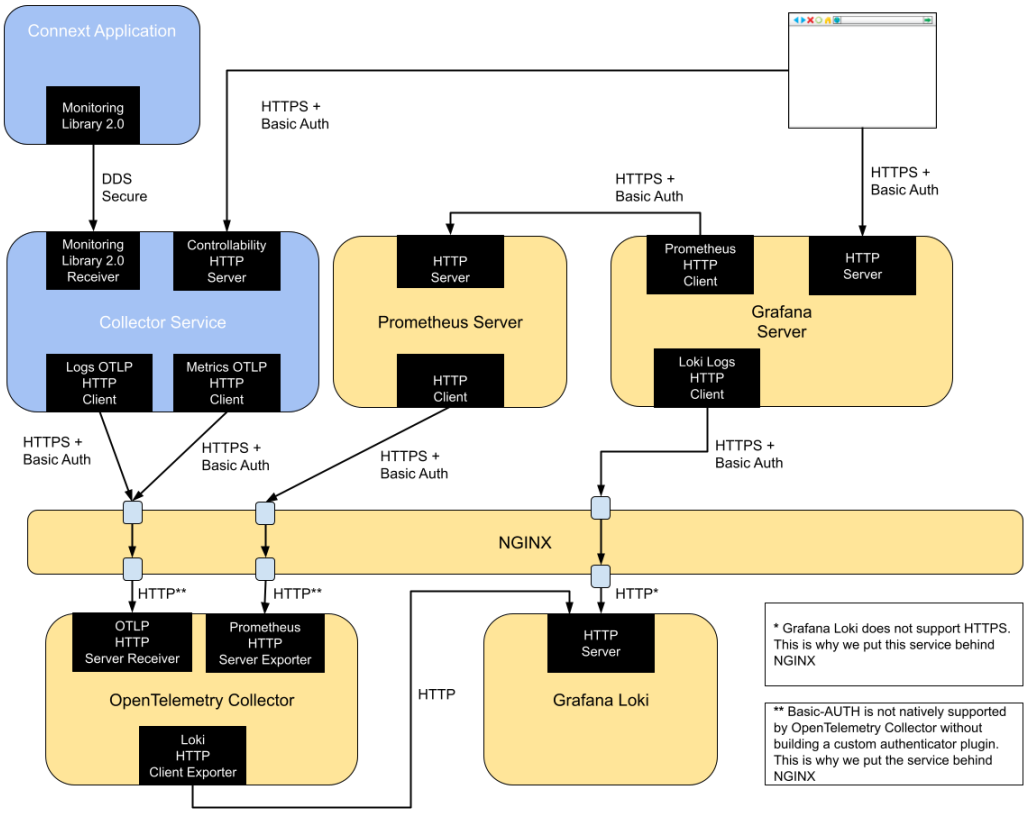


Figure 4.2: RTI Observability Framework with OpenTelemetry Collector

For additional information on how to use Docker Compose to run *Observability Framework*, see *Configuring, Running, and Removing Observability Framework Components Using Docker Compose*.

Collector Service

This release supports running *Observability Collector Service* in storage mode only. Data can be stored into Prometheus and Grafana Loki natively or into other third-party observability backends using OpenTelemetry and the OpenTelemetry Collector. The prepackaged deployment uses a single layer deployment to run only one *Collector Service* instance for the *Connext* system, as illustrated in Figure 4.3 *Single Collector Deployment*.

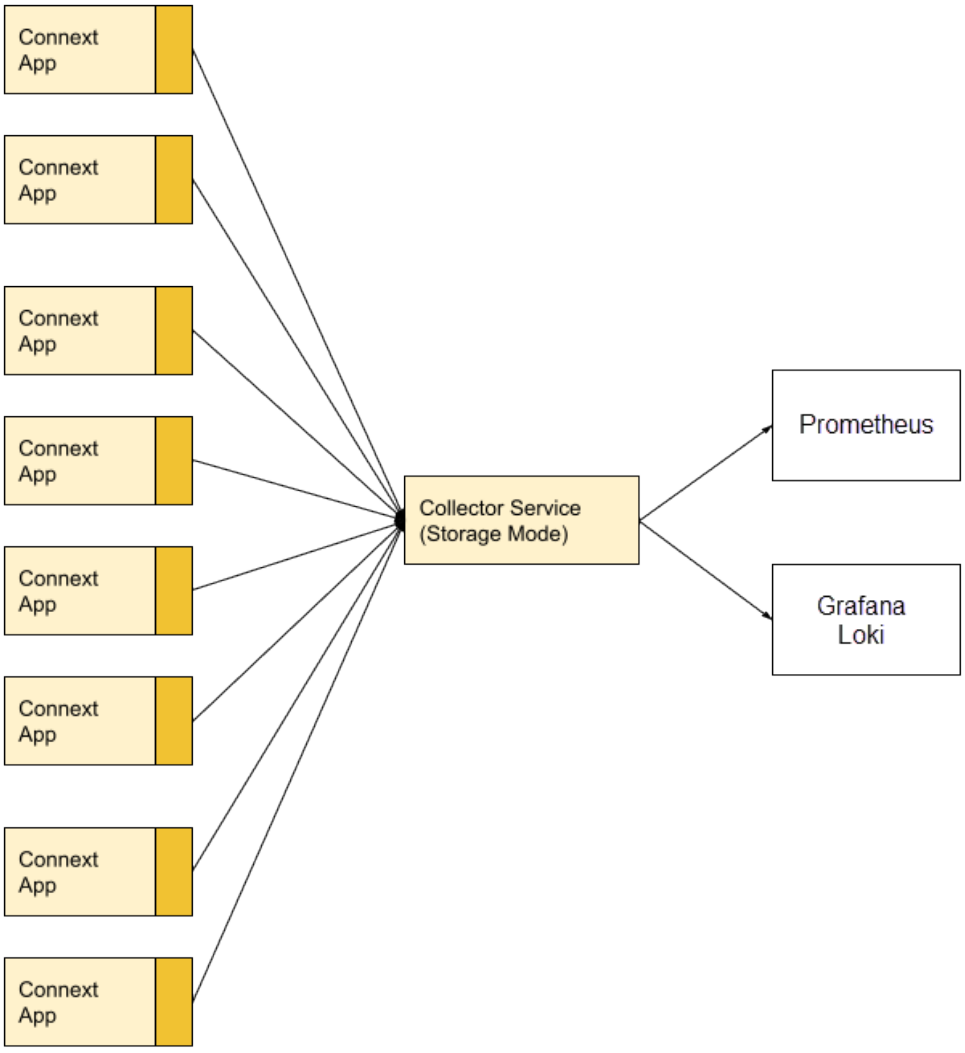


Figure 4.3: Single Collector Deployment

4.1.2 Docker (Separate Deployment)

As an alternative to the prepackaged Docker Compose deployment provided by RTI, you can also run *Observability Framework* and the third-party components (e.g, Prometheus) standalone.

The third-party components Prometheus, Grafana Loki, Grafana, OpenTelemetry Collector (optional), and NGINX (optional) are also distributed as Docker images by their respective vendors. You can use these images standalone instead of RTI's *prepackaged Docker Compose*.

Observability Collector Service is distributed as a Docker image hosted in Dockerhub. This is the same publicly available image used by the prepackaged Docker Compose installation, and it requires a valid RTI license to run.

This release supports running *Observability Collector Service* in storage mode only. Data can be stored into Prometheus and Grafana Loki natively or into other third-party observability backends using OpenTelemetry and the OpenTelemetry Collector. Because forwarding mode is not supported, you can only use a single layer of *Collector Services* per *Connext* system. This configuration is illustrated in Figure 4.4 *Single Layer Collector Deployment* and Figure 4.5 *Single Layer Collector Deployment using OpenTelemetry Collector*.

The deployments represented in Figure 4.4 *Single Layer Collector Deployment* and Figure 4.5 *Single Layer Collector Deployment using OpenTelemetry Collector* require running multiple instances of *Collector Service* where each *Connext* application configures *Monitoring Library 2.0* to connect to one of the *Collector Service* instances.

You are responsible for running the *Collector Service* instances and the third-party components for storage. For example, if you want to store telemetry data into Prometheus and Grafana Loki, you must run Prometheus and Grafana Loki instances and configure the Docker container for *Collector Service* to connect to these storage backends.

The Docker image included with *Collector Service* contains a built-in configuration that enables it to run in *storage mode* with the following operation modes:

Table 4.1: Docker Container Operation Modes

Configuration Name	Network	Data Storage	Security
NonSecureLAN	LAN	Prometheus and Grafana Loki	No
NonSecureWAN	WAN	Prometheus and Grafana Loki	No
SecureLAN	LAN	Prometheus and Grafana Loki	Yes
SecureWAN	WAN	Prometheus and Grafana Loki	Yes
NonSecureOTelLAN	LAN	Multiple through OpenTelemetry Collector	No
NonSecureOTelWAN	WAN	Multiple through OpenTelemetry Collector	No
SecureOTelLAN	LAN	Multiple through OpenTelemetry Collector	Yes
SecureOTelWAN	WAN	Multiple through OpenTelemetry Collector	Yes

For additional information on how to use the Docker image included with *Collector Service*, refer to Docker's Collector Service article.

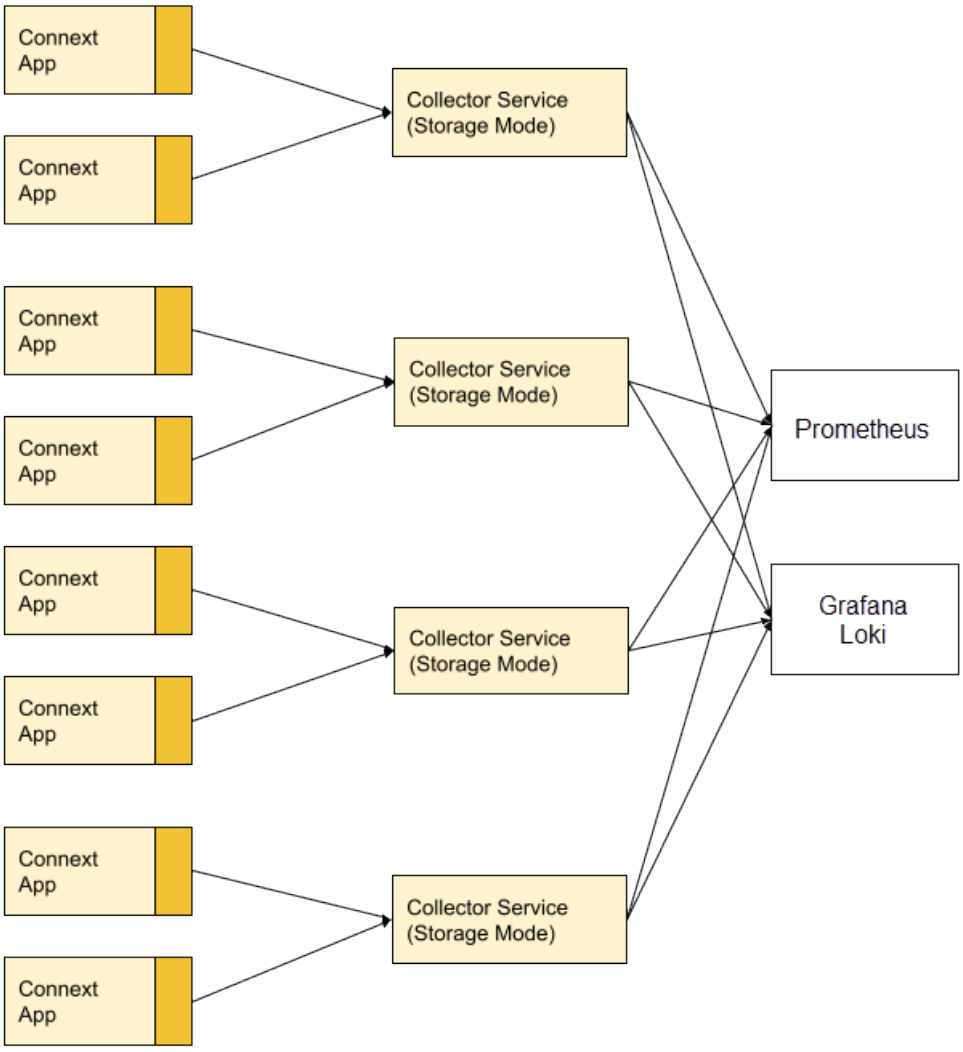


Figure 4.4: Single Layer Collector Deployment

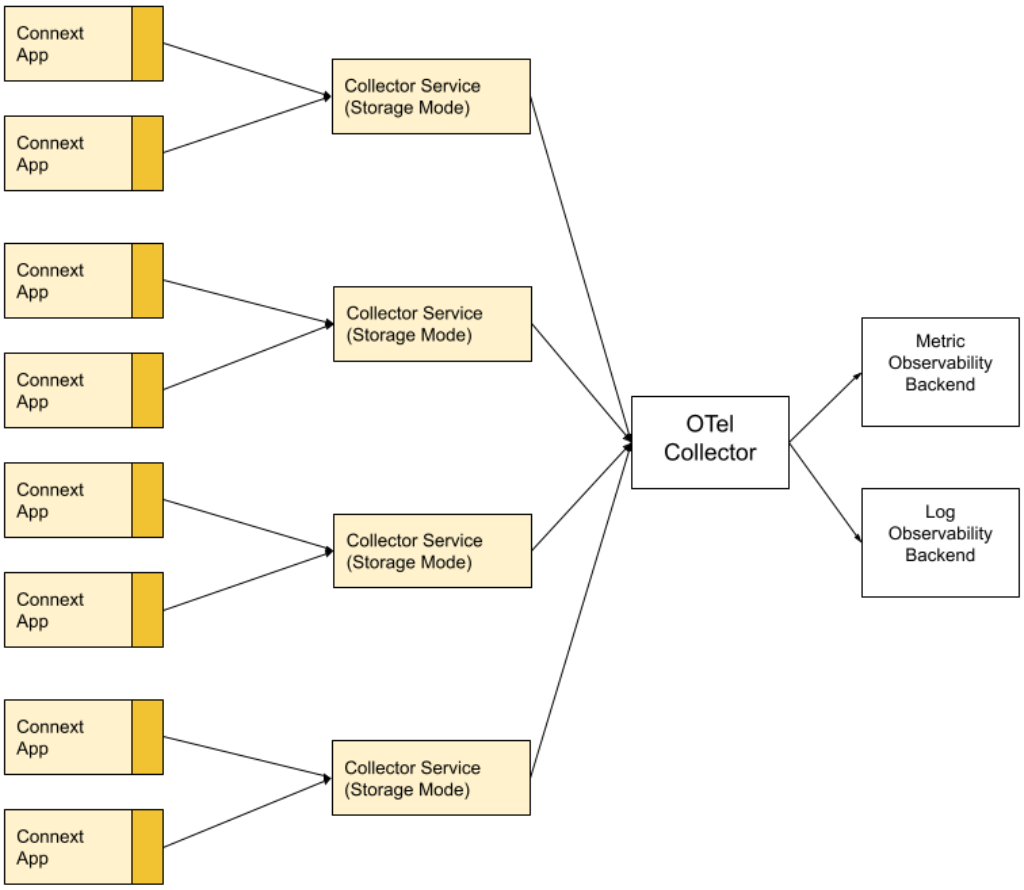


Figure 4.5: Single Layer Collector Deployment using OpenTelemetry Collector

4.2 Future releases

4.2.1 Collector Service

Executable

In future releases, *Collector Service* will be provided as a standalone executable without using Docker to deploy.

Collector Service Deployments

As you roll out telemetry data collection and distribution across all your *Connext* applications, *Observability Framework* must be deployed in a way that supports the additional load. A single layer *Collector Service* deployment, as shown in Figure 4.4 *Single Layer Collector Deployment* and Figure 4.5 *Single Layer Collector Deployment using OpenTelemetry Collector*, may not scale sufficiently.

A better deployment option would be the layered deployment depicted in Figure 4.6 *Layered Collector Deployment* and Figure 4.7 *Layered Collector Deployment Using OpenTelemetry Collector*. In this option, you have multiple layers of *Collector Service* gathering, filtering, and forwarding the telemetry data produced by the *Connext* applications. Each intermediate layer reduces the number of egress points required to send data and provides an opportunity to filter telemetry data. The last layer works as a storage layer and is responsible for storing the telemetry data into a third-party observability backend.

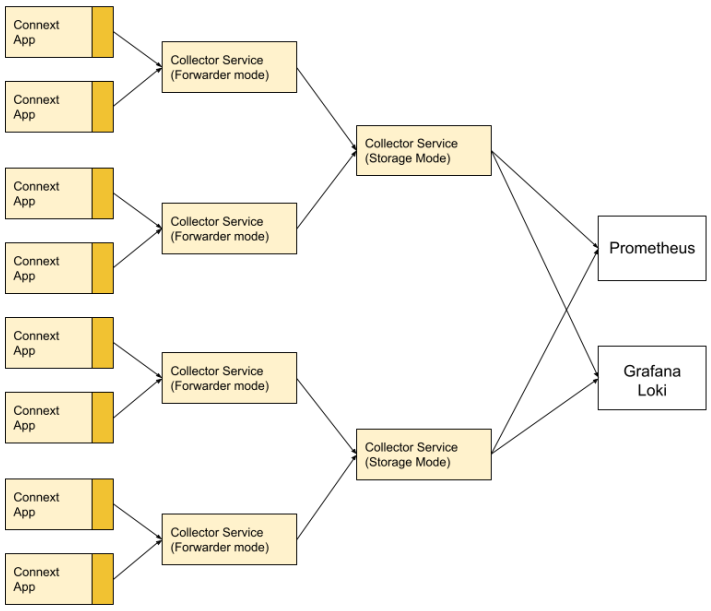


Figure 4.6: Layered Collector Deployment

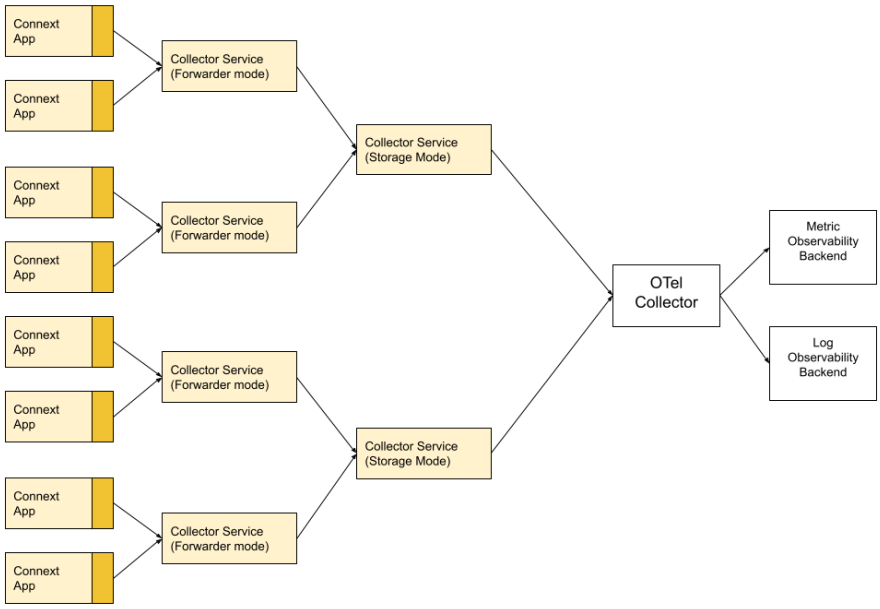


Figure 4.7: Layered Collector Deployment Using OpenTelemetry Collector

Chapter 5

Security

Observability Framework can secure the telemetry data generated by *Connext* applications and stored in the telemetry backends. Data in transit can be secured using the *RTI® Security Plugins* and BASIC-Auth over HTTPS. Data at rest is secured by the third-party telemetry backends.

Figure 5.1 shows the *Observability Framework* security architecture when *Collector Service* is configured to store the telemetry data in Prometheus and Grafana Loki.

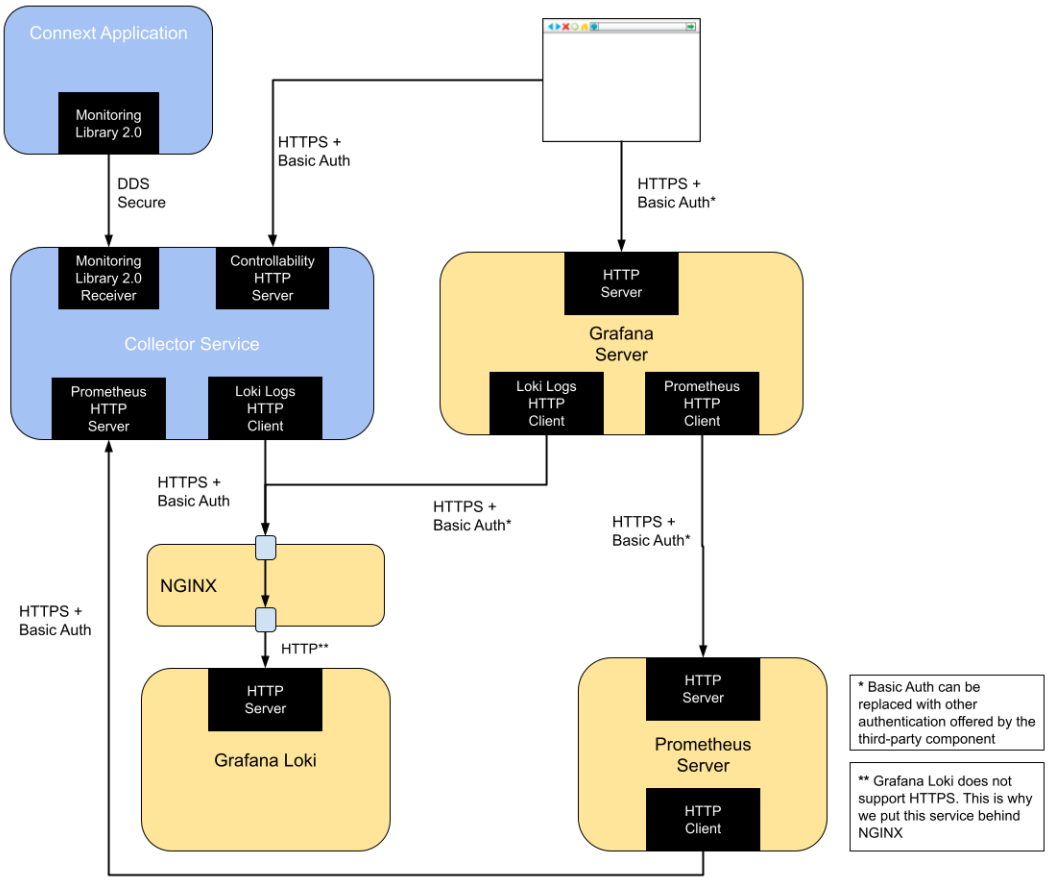


Figure 5.1: Security Architecture of RTI Observability Framework when using Prometheus and Grafana Loki

To facilitate testing and evaluation, you can install *Observability Framework* using *Docker Compose (Prepackaged)* to automatically run and deploy all the components shown in Figure 5.1 within a single host.

Figure 5.2 shows the *Observability Framework* security architecture when *Collector Service* is configured to forward the telemetry data to an OpenTelemetry Collector which itself is configured to store the telemetry into different backends for logs and metrics. Note that the *Observability Framework* only provides Grafana dashboards configured to use Grafana Loki and Prometheus backends.

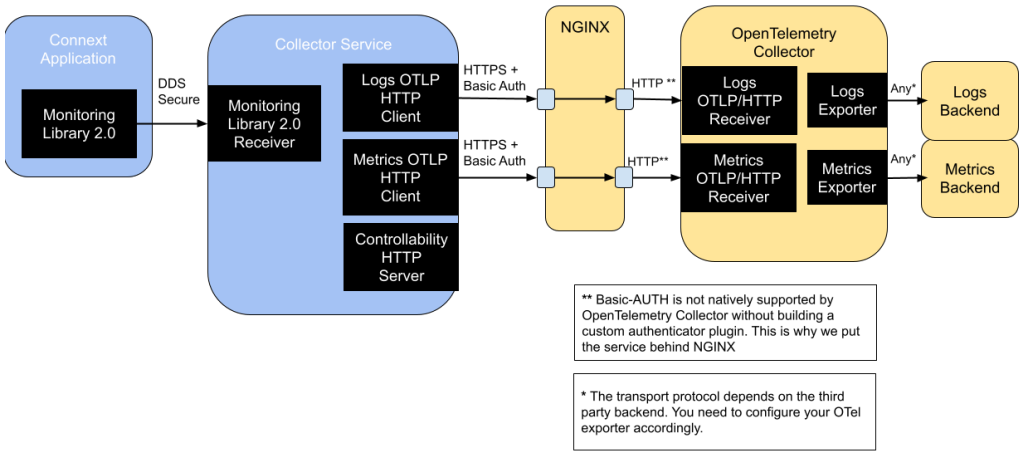


Figure 5.2: Security Architecture of RTI Observability Framework when using OpenTelemetry Collector

To facilitate testing and evaluation of securing telemetry data when using an OpenTelemetry Collector, you can run *Observability Framework* using *Docker Compose (Prepackaged)* with an OpenTelemetry Collector instance that stores the telemetry data in local Prometheus and Grafana Loki backends as shown in Figure 5.3.

5.1 Secure Communication between Connext Applications and Collector Service

The exchange of telemetry data between a *Connext* application and *Collector Service* is secured by using the Security Plugins. For additional information on how to configure the Security Plugins, see the Support for RTI Observability Framework section in the *RTI Security Plugins User's Manual*.

To configure secure communications between *Connext* applications and *Collector Service*, follow the steps for your selected deployment.

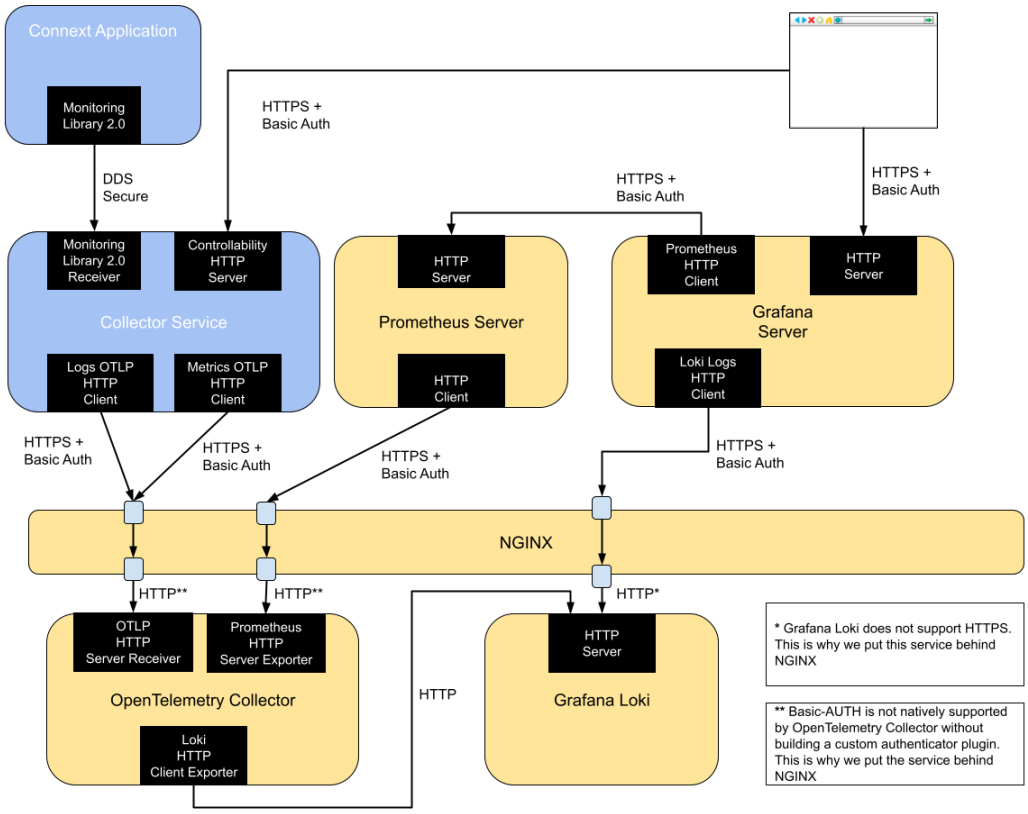


Figure 5.3: Security Architecture of the RTI Observability Framework when using OpenTelemetry Collector, Prometheus and Grafana Loki

5.1.1 Secure Communication between Connext Applications and Collector Service (Pre-Packaged Deployment)

If you install *Observability Framework* using the *Docker Compose (Prepackaged)* option, the security artifacts required to configure the SECURITY PLUGINS in *Collector Service* must be provided during the installation process. Use the highlighted parameters in your JSON configuration file:

```
{
    "securityConfig": {
        "basicAuthUsername": "yourusername",
        "basicAuthPassword": "yourpassword",
        "httpsSecurity": {
            "caCertificate":     "path/to/ca_cert.pem",
            "serverCertificate": "path/to/server_cert.pem",
            "serverKey":         "path/to/server_key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":     "path/to/identityCaCert.pem",
            "permissionsCaCertificate": "path/to/permissionsCaCert.pem",
            "identityCertificate":       "path/to/identityCert.pem",
            "identityKey":               "path/to/identityKey.pem",
            "signedPermissionsFile":     "path/to/signedPermissions.p7s",
            "signedGovernanceFile":      "path/to/signedGovernance.p7s"
        }
    }
}
```

5.1.2 Secure Communication between Connext Applications and Collector Service (Separate Deployment)

If you run *Collector Service* using the *Docker (Separate Deployment)* option, you must provide the the security artifacts required to configure the SECURITY PLUGINS in *Collector Service*. In addition, you need to set the `CFG_NAME` environment variable to one of the provided Docker image's built-in secure configurations (see *Docker (Separate Deployment)*). The security artifacts and environment variable can be provided by using the following options to the `docker run` command:

```
-v path/to/identityCaCert.pem:/rti/security/dds/identity_ca.pem
-v path/to/permissionsCaCert.pem:/rti/security/dds/permissions_ca.pem
-v path/to/identityCert.pem:/rti/security/dds/identity_certificate.pem
-v path/to/identityKey.pem:/rti/security/dds/private_key.pem
-v path/to/signedPermissions.p7s:/rti/security/dds/permissions.p7s
-v path/to/signedGovernance.p7s:/rti/security/dds/governance.p7s
-e CFG_NAME="<secure-configuration>"
```

For additional details, see the *Collector Service* docker image documentation.

For details on how to generate the security artifacts see *Generating the Observability Framework Security Artifacts*.

5.2 Secure Communication with Collector Service HTTP Servers

Collector Service can start two HTTP servers: one to receive remote commands and another one to expose the Prometheus metrics. The communication with these HTTP servers is secured using BASIC-Auth over HTTPS.

- *Collector Service* provides a REST API for receiving remote commands to control the collection and distribution of telemetry data from *Connext* Applications.
- *Collector Service* can be configured to provide a Prometheus HTTP metrics endpoint to expose telemetry data to a Prometheus backend. Prometheus collects metrics from targets by scraping HTTP metrics endpoints on these targets.

Important: The configuration of the HTTP clients initiated by third-party components is out of the scope of this documentation. Please refer to the documentation of the third-party components for additional details.

However, if you install *Observability Framework* using *Docker Compose (Prepackaged)*, the third-party components will be configured to use the security artifacts provided in the installation JSON configuration file. You can take a look into the configuration files of the third-party components located in the directory `<rti_workspace_dir>/user_config/observability` to see how the security artifacts are used.

To configure the security of the HTTP servers started by *Collector Service*, follow the steps for your selected deployment.

5.2.1 Secure Collector Service HTTP Servers (Pre-Packaged Deployment)

If you install *Observability Framework* using *Docker Compose (Prepackaged)*, use the highlighted parameters in the installation JSON configuration file:

```
{
    "securityConfig": {
        "basicAuthUsername": "yourusername",
        "basicAuthPassword": "yourpassword",
        "httpsSecurity": {
            "caCertificate":     "path/to/ca_cert.pem",
            "serverCertificate": "path/to/server_cert.pem",
            "serverKey":         "path/to/server_key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":     "path/to/identityCaCert.pem",
            "permissionsCaCertificate": "path/to/permissionsCaCert.pem",
            "identityCertificate":       "path/to/identityCert.pem",
            "identityKey":               "path/to/identityKey.pem",
            "signedPermissionsFile":     "path/to/signedPermissions.p7s",
            "signedGgovernanceFile":     "path/to/signedGovernance.p7s"
        }
    }
}
```

5.2.2 Secure Collector Service HTTP Servers (Separate Deployment)

If you run *Collector Service* using *Docker (Separate Deployment)*, you must provide the security artifacts to configure the HTTP servers running in *Collector Service*. In addition, you need to set the `CFG_NAME` environment variable to one of the provided Docker image's built-in secure configurations (see *Docker (Separate Deployment)*). The security artifacts and environment variable can be provided by using the following options to the `docker run` command:

```
-v /path/to/serverPrometheusEndpoint.pem:/rti/security/https/
↪serverPrometheusEndpoint.pem
-v /path/to/serverControl.pem:/rti/security/https/serverControl.pem
-v /path/to/htdigest:/rti/security/https/htdigest
-e CFG_NAME="<secure-configuration>"
```

- The *serverPrometheusEndpoint.pem* file must contain both a valid server certificate (*serverPrometheusEndpoint_cert.pem*) and the corresponding private key (*serverPrometheusEndpoint_key.pem*).
- The *serverControl.pem* file must contain both a valid server certificate (*serverControl_cert.pem*) and the corresponding private key (*serverControl_key.pem*).
- The *htdigest* is a password file that contains the username and password for BASIC-Auth created using Apache `htdigest`.

For additional details, see the *Collector Service* docker image documentation.

For details on how to generate the *server.pem* files and the *htdigest* file, see *Generating the Observability Framework Security Artifacts*.

5.3 Secure Communication with Third-Party Component HTTP Servers

Observability Framework can start three HTTP clients: one to send logs to Grafana Loki, one to send logs to OpenTelemetry Collector, and one to send metrics to OpenTelemetry Collector. The communication with these HTTP clients is secured using BASIC-Auth over HTTPS.

Important: The configuration of the third-party components' HTTP servers is out of the scope of this documentation. Please refer to the documentation of the third-party components for additional details.

However, if you install *Observability Framework* using *Docker Compose (Prepackaged)*, the third-party components will be configured to use the security artifacts provided in the installation JSON configuration file. You can take a look into the configuration files of the third-party components located in the directory `<rti_workspace_dir>/user_config/observability` to see how the security artifacts are used.

To configure the security of the HTTP clients started by *Collector Service*, follow the steps for your selected deployment.

5.3.1 Secure Third-Party Component HTTP Servers (Pre-Packaged Deployment)

If you install *Observability Framework* using *Docker Compose (Prepackaged)*, use the highlighted parameters in the installation JSON configuration file:

```
{
    "securityConfig": {
        "basicAuthUsername": "yourusername",
        "basicAuthPassword": "yourpassword",
        "httpsSecurity": {
            "caCertificate":     "path/to/ca_cert.pem",
            "serverCertificate": "path/to/server_cert.pem",
            "serverKey":         "path/to/server_key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":     "path/to/identityCaCert.pem",
            "permissionsCaCertificate": "path/to/permissionsCaCert.pem",
            "identityCertificate":       "path/to/identityCert.pem",
            "identityKey":               "path/to/identityKey.pem",
            "signedPermissionsFile":     "path/to/signedPermissions.p7s",
            "signedGovernanceFile":      "path/to/signedGovernance.p7s"
        }
    }
}
```

5.3.2 Secure Third-Party Component HTTP Servers (Separate Deployment)

If you run *Collector Service* using *Docker (Separate Deployment)*, you must provide the security artifacts to configure the HTTP clients running in *Collector Service*. In addition, you need to set the `CFG_NAME` environment variable to one of the provided Docker image's built-in secure configurations (see *Docker (Separate Deployment)*). The security artifacts and environment variable can be provided by using the following options to the `docker run` command:

```
-v /path/to/rootCA.crt:/rti/security/https/rootCALoki.crt
-v /path/to/rootCA.crt:/rti/security/https/rootCAOtel.crt
-e OBSERVABILITY_BASIC_AUTH_USERNAME=yourusername
-e OBSERVABILITY_BASIC_AUTH_PASSWORD=yourpassword
-e CFG_NAME="<secure-configuration>"
```

- The *rootCALoki.crt* file must contain the root certificate of the CA that signed the *server.pem* certificate used to communicate with the Grafana Loki server.
- The *rootCAOtel.crt* file must contain the root certificate of the CA that signed the *server.pem* certificate used to communicate with the OpenTelemetry Collector.

For additional details, see the *Collector Service* docker image documentation.

5.4 Generating the Observability Framework Security Artifacts

This section describes how to generate the security artifacts required to secure *Observability Framework*. For an overview of the security architecture of the *Observability Framework*, see *Security*.

There are two sets of security artifacts:

- DDS security artifacts secure the exchange of telemetry data between a *Connext* using *Monitoring Library 2.0* and *Collector Service*.
- HTTPS security artifacts secure the exchange of telemetry data between *Collector Service* and the third-party observability backends as well as to send remote commands to *Collector Service*.

5.4.1 Generating DDS Security Artifacts

The DDS security artifacts are used to secure the exchange of telemetry data between *Connext* applications and *Collector Service*.

See Support for RTI Observability Framework section in the *RTI Security Plugins User's Manual* for details about how to secure the communication between a *Connext* application and *Collector Service*.

For details on how to create/update DDS security artifacts, see Generating and Revoking Your Own Certificates Using OpenSSL in the *RTI Security Plugins Getting Started Guide*.

5.4.2 Generating HTTPS Security Artifacts

The security artifacts needed to secure the communication between *Collector Service* and the third-party observability backends are:

- A root CA certificate file
- A server certificate file
- A server key

We will start by generating a self-signed Root CA, which will issue the *Server Certificate* used to secure the various HTTP servers in *Observability Framework*. This will require us to set up a minimal security infrastructure first.

We will show an example for ECDSA as the public-key algorithm to generate the certificates. Note that you can use any public-key algorithm listed in Supported Cryptographic Algorithms in the *RTI Security Plugins User's Manual*.

Note: We will use the **OpenSSL CLI** to perform the security operations in the generation of the security artifacts. Make sure to include in the path your OpenSSL binary directory[1]. The installation process is described in the RTI Security Plugins Installation Guide.

---

[1] Read the official documentation for more information on the OpenSSL configuration files.

Preliminary Steps

Setting up a security infrastructure requires some preliminary configuration. We will cover a minimal setup here.

1. The `rti_workspace` directory containing examples and user configuration files is automatically copied into the users' home or My Documents folder when the first RTI application is launched (e.g., RTI Launcher, rtiddsgen, rtipkginstall, or rtiobservability). In your `rti_workspace` directory you should have OpenSSL configuration files named `<rti_workspace_dir>/examples/dds_security/cert/ecdsa01/ca/ecdsa01RootCa.cnf` and `<rti_workspace_dir>/examples/dds_security/cert/ecdsa01/https/ecdsa01Https01.cnf`. Make copies of these files and call them `observabilityRootCa.cnf` and `observabilityServer.cnf` respectively. To better organize your project, save these copies in a new directory called `cert/observability`:

   Linux

   ```
   $ cd <rti_workspace_dir>/examples/dds_security
   $ cp cert/ecdsa01/ca/ecdsa01RootCa.cnf cert/observability/ca/
   ↪observabilityRootCa.cnf
   $ cp cert/ecdsa01/https/ecdsa01Https01.cnf cert/observability/https/
   ↪observabilityServer.cnf
   ```

2. Modify `observabiltyRootCa.cnf` to redefine the `name` variable. Note that this configuration file uses this variable to derive some filenames, such as those used in the next section:

   ```
   ...
   # Variables defining this CA
   name = observabilityRootCa                  # Name
   desc =                                      # Description
   ...
   ```

Initialize the OpenSSL CA Database

When using a CA to perform an operation, OpenSSL relies on special database files to keep track of the issued certificates, serial numbers, revoked certificates, etc. We need to create these database files to be able to use the **`openssl x509 -req`** command:

Linux

```
$ mkdir cert/observability/ca/database
$ touch cert/observability/ca/database/observabilityRootCaIndex
$ echo 01 > cert/observability/ca/database/observabilityRootCaSerial
```

Limit the Access of the CA's Private Key

It is also a good practice to store the CA's private key in a separate directory with more restrictive access rights, so only you can sign certificates.

Linux

```
$ mkdir cert/observability/ca/private
$ chmod 700 cert/observability/ca/private
```

Generating a New Root CA

1. Modify `cert/observability/ca/observabilityRootCa.cnf` and specify the fields in the `req_distinguished_name` section. This information will be incorporated into your certificate:

   ```
   ...
   [ req_distinguished_name ]

   countryName            = US
   stateOrProvinceName    = CA
   localityName           = Santa Clara
   0.organizationName     = Observing Organization
   commonName             = Observability Root CA
   emailAddress           = rootCa@observability.com
   ...
   ```

2. Use the OpenSSL CLI to generate a self-signed certificate using the Root CA's configuration. Run the following command from the `cert/observability` directory:

   ECDSA secp256r1

   ```
   $ openssl req -nodes -x509 -days 1825 -text -sha256 -newkey ec -pkeyopt␣
   ↪ec_paramgen_curve:prime256v1 -keyout ca/private/observabilityRootCaKey.
   ↪pem -out ca/observabilityRootCaCert.pem -config ca/observabilityRootCa.
   ↪cnf
   ```

   This will produce a new private key, `observabilityRootCaKey.pem` in the `cert/observability/ca/private` directory, and a new certificate, `observabilityRootCaCert.pem`, in the `cert/observability/ca` directory. This certificate will be valid for 1825 days (5 years) starting today.

Generating Server Certificates

Server Certificates are verified against the Root CA when authenticating servers over HTTPS. Therefore, in the simplest scenario, it is the Root CA that is responsible for issuing Server Certificates.

We will create a certificate signing request (CSR) for the server localhost. Then we will use the new Root CA to issue the certificate requested by the CSR.

1. Add the information you want to include in localhost's certificate in the file `cert/observability/https/observabilityServer.cnf` that was previously created. You may want to use the following contents as a reference:

Listing 5.1: Sample contents of observabilityServer.cnf

```
prompt=no
distinguished_name         = req_distinguished_name

[ req_distinguished_name ]
countryName=US
stateOrProvinceName=CA
organizationName=Observing Organization
emailAddress=server@observability.com
commonName=localhost

[ https_cert ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = localhost
IP.1 = 127.0.0.1
```

   You are free to modify any field except `countryName`, `stateOrProvinceName`, and `organizationName`. These fields must match the ones of the Root CA; otherwise it will refuse to issue the requested certificate (note that a `commonName` is also required). These requirements are specified in `observabilityRootCa.cnf`, in the `policy_match` section.

2. Generate the new server's key and CSR. Run the following command from the `cert/observability` directory:

   ECDSA secp256r1

```
$ openssl req -nodes -new -newkey ec -pkeyopt ec_paramgen_
↪curve:prime256v1 -config https/observabilityServer.cnf -keyout https/
↪observabilityServerKey.pem -out https/observabilityServer.csr
```

   This will produce an RSA private key, `observabilityServerKey.pem`, and a CSR based on that key, `observabilityServer.csr`. Since CSRs have all the information and cryptographic material that a CA needs to issue a certificate, the server's private key must never be known to anyone but the creator.

3. Use the new Root CA's certificate and private key to issue a new Server Certificate. Run the following command from the `cert/observability` directory:

   ECDSA secp256r1

```
$ openssl x509 -req -days 730 -text -CAserial ca/database/
↪observabilityRootCaSerial -extfile https/observabilityServer.cnf -
↪extensions https_cert -CA ca/observabilityRootCaCert.pem -CAkey ca/
↪private/observabilityRootCaKey.pem -in https/observabilityServer.csr -
↪out https/observabilityServerCert.pem
```

The Root CA will issue the server's public certificate, `observabilityServerCert.pem`, which will be valid for 730 days (2 years) starting today.

4. *Collector Service* requires a server certificate file for HTTPS operation that contains both the server certificate and key. The following is an example of how to create this file using the server certificate and key generated in the previous step. Run the following command from the `<rti_workspace_dir>/examples/dds_security/cert/observability` directory:

   Linux

```
$ cp https/observabilityServerCert.pem observabilityServer.pem
$ cat https/observabilityServerKey.pem  >> observabilityServer.pem
```

BASIC-Auth Password File

The communication between *Collector Service* and the third-party observability backends is secured using BASIC-Auth over HTTPS.

The HTTP servers started by *Collector Service* require a password file that contains the username and password for BASIC-Auth. This section describes how to create this file.

Note: The creation of the equivalent password file for the third-party observability backends is out of the scope of this documentation. Please refer to the documentation of the third-party observability backends for additional details on how to create this file.

Collector Service requires an htdigest formatted password file for basic authentication. The following example uses the Apache `htdigest` command to create this file. For more information on this command see Apache - htdigest - manage user files for digest authentication

Here is an example of how to use the `htdigest` command:

Linux

```
$ htdigest -c htdigest localhost user
Adding password for user in realm localhost.
New password: <type "userpassword">
Re-type new password: <type "userpassword">
```

The example uses the following arguments for the `htdigest` command.

Table 5.1: htdigest Arguments

Parameter	Description	Value
-c	Create the passwdfile. If passwdfile already exists, it is deleted first.	-c
password-file	Name of the file to contain the username, realm, and password. If -c is given, this file is created if it does not already exist, or deleted and recreated if it does exist.	htdigest
realm	The realm name to which the user name belongs. See http://tools.ietf.org/html/rfc2617#section-3.2.1 for more details.	hostname ("localhost")
username	The user name to create or update in passwdfile. If username does not exist is this file, an entry is added. If it does exist, the password is changed.	user
password	The password to create or update in passwdfile. If username does not exist is this file, an entry is added. If it does exist, the password is changed.	user-password

This will create an htdigest file with the following content:

```
user:localhost:bbbb113a9f365f1b3787b6a944ccbc59
```

Chapter 6

Installing and Running Observability Framework

RTI Connext Observability Framework is not installed as part of *RTI Connext Professional* with the exception of *Monitoring Library 2.0* which is included in the *RTI Connext Professional* target package. *Monitoring Library 2.0* is supported in all *Connext* platforms. *Observability Framework* must be downloaded and installed separately. For information on how to obtain the *Observability Framework* package, check the RTI Customer portal, contact support@rti.com, or contact your account team.

There is one *Observability Framework* package, as outlined in Table 6.1.

Table 6.1: Observability Framework Packages

Package Name	Package Contents	Use Case	Supported Platform
rti_observability-7.3.0-host-x64Linux.tar.gz	The host package contains the files required to run the *Observability Framework* collection, storage, and visualization components using Docker and Docker Compose. This package also includes *Observability Framework* documentation.	Install this package if you need to run the collection, storage, and visualization components.	These components are only supported in Linux. The host package can be installed on a Virtual Machine (VM); for more information, see *Supported Docker Compose Environments*.

In the rest of this chapter, `<installdir>` refers to the installation directory for *Connext*.

Important: *Observability Framework* is an experimental product that includes example configuration files for use with several third-party components (Prometheus, Grafana Loki, and Grafana). This release is an evaluation distribution; use it to explore the new observability features that support *Connext* applications.

Do not deploy any *Observability Framework* components in production.

6.1 Installing the Host Package

There are two ways to install the documentation and files supporting the Docker containers used by *Observability Framework*: using *RTI Launcher* or the `rtipkginstall` command-line utility.

6.1.1 Prerequisites

The following applications must be installed before installing the experimental *Observability Framework* product.

- *Connext* 7.3.0. For installation instructions, see the RTI Connext Installation Guide.
- Docker Engine v20.10.x or higher. For installation instructions, see Docker's Engine installation overview.
- Docker Compose Plugin v2.x or higher. For installation instructions, see Docker's installation instructions.

Note: The *Observability Framework* host package has been tested on the platforms noted in *Supported Docker Compose Environments*.

6.1.2 Install from RTI Launcher

To install the *Observability Framework* host package from *RTI Launcher*:

1. Start *Launcher* from the Start menu, or from the command line using: `<installdir>/bin/rtilauncher`.
2. From the **Configuration** tab, click **Install RTI Packages**.
3. Use the plus (+) sign to add the `rti_observability-<version>-host-x64Linux.rtipkg` file.
4. Click **Install**.

6.1.3 Install from the Command Line

To install the *Observability Framework* host package from the command line, run:

```
$ <installdir>/bin/rtipkginstall /<path-to-observability-framework-file>/rti_
↪observability-<version>-host-x64Linux.rtipkg
```

6.2 Configuring, Running, and Removing Observability Framework Components Using Docker Compose

The telemetry data forwarded by *Monitoring Library 2.0* is processed, stored, and visualized using the following components:

- RTI Observability Collector Service
- Prometheus
- Grafana Loki
- Grafana
- OpenTelemetry Collector [Optional]: *Observability Framework* can be configured to launch an instance of OpenTelemetry Collector that will store the telemetry data in Prometheus and Loki instead of this being done by the *RTI Observability Collector Service*. In this configuration mode, *Observability Collector Service* sends the data to OpenTelemetry Collector.
- NGINX [Optional]: when using security the *Observability Framework* runs and instance of NGINX to secure communications with Grafana Loki and the OpenTelemetry Collector.

The files required to run these components are installed by the *Observability Framework* host package. In this release, the collection, storage, and visualization components only run in a single Linux host using Docker and Docker Compose. Future releases will offer the ability to install the components independently without using Docker.

Observability Framework can be deployed with or without using the OpenTelemetry Collector. Both deployment options can be configured to be secure or non-secure and to work on a LAN or WAN. For additional information on the deployment options, see *Docker Compose (Prepackaged)*.

Warning: *Observability Framework* uses third-party software that is subject to each product's license terms and conditions. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

6.2.1 Configuring the Docker Workspace for Observability Framework

Before creating and running the Docker containers for *Observability Framework*, the associated configuration files that comprise the Docker workspace must be created and copied to the `rti_workspace/<version>/user_config/observability` directory. This is done using the `<installdir>/bin/rtiobservability` script.

There are several optional, user-defined variables you can use to configure *Observability Framework*. These variables are specified in a JSON file.

Note: To reconfigure an existing Docker workspace you must first remove the existing workspace as described in section *Removing the Docker Workspace for Observability Framework*.

Configure the JSON File

Before creating your workspace, you will need to provide your configuration using a JSON file. This file can contain all the specific ports, names, and certificates to be used by the different services.

The following default JSON file is included in the installation folder at `<rti_installation>/resource/app/app_support/observability/default.json`. You can copy this file to another location, then modify it as needed to create the *Observability Framework* configuration for your environment. Alternately, you can create your own JSON file.

```
{
  "hostname": "localhost",
  "observabilityDomain": 2
}
```

Table 6.2 *JSON Configuration file* describes all of the JSON configuration fields and default values.

Note: All of the JSON configuration fields are optional except `hostname`. If configuration for `securityConfig` is required, then all its fields must be provided.

Table 6.2: JSON Configuration file

Field Name	Description	Type	Default Value
hostname	Hostname to be used to configure all of the services. This field is required.	String	N/A
observabilityDomain	DDS Domain to be used to exchange Observability data.	int	2
lgpStackConfig			
lgpStackConfig.grafanaPort	The Grafana server port. This is the port that the Grafana service listens to.	int	3000
lgpStackConfig.prometheusPort	The Prometheus server port. This is the port that the Prometheus service listens to.	int	9090
lgpStackConfig.lokiPort	The Loki server port. This is the port that the Loki service listens to.	int	3100
collectorConfig			
collectorConfig.prometheusExporterPort	The *Observability Collector Service* Prometheus endpoint port for exporting telemetry data to Prometheus. This is the port that the Prometheus endpoint service listens to and uses to provide telemetry data via scrapes from the Prometheus server.	int	19090
collectorConfig.controlPort	The *Observability Collector Service* server port for control commands. This is the port that the service listens to.	int	19098
collectorConfig.controlPublicHostname	The *Observability Collector Service* public server hostname for control commands.	String	*hostname*
collectorConfig.controlPublicPort	The *Observability Collector Service* public server port for control commands. This is the port exposed to the public network.	int	*collectorConfig.controlPort*
collectorConfig.rtwPublicAddress	The WAN public address used by *Real-Time WAN Transport*.	String	*hostname*
collectorConfig.rtwPort	The WAN port used by *Real-Time WAN Transport*. This is both the private port where *Real-Time WAN Transport* receives data, and the public port exposed to the public network.	int	30000
otelConfig			
otelConfig.otelHttpReceiverPort	The OpenTelemetry Collector server port. This is the port the OpenTelemetry Collector listens to for telemetry data.	int	N/A
securityConfig			
securityConfig.basicAuthUsername	Username used for HTTP Basic authentication.	String	N/A
securityConfig.basicAuthPassword	Password used for HTTP Basic authentication.	String	N/A
securityConfig.httpsSecurity			

Complete examples of both secure and non-secure configurations of the *Observability Framework* may be found in the section *Configure Observability Framework for the Appropriate Operation Mode* of the Getting started Guide.

An example of a fully-defined JSON file, with security and OpenTelemetry configured, follows. You can follow this example to create your own custom configuration.

```
{
   "hostname": "localhost",
   "observabilityDomain": 2,
   "lgpStackConfig": {
      "grafanaPort": 3000,
      "prometheusPort": 9090,
      "lokiPort": 3100
   },
   "collectorConfig": {
      "prometheusExporterPort": 19090,
      "controlPort": 19098
   },
   "otelConfig": {
      "otelHttpReceiverPort": 4318
   },
   "securityConfig": {
     "basicAuthUsername": "yourusername",
     "basicAuthPassword": "yourpassword",
     "httpsSecurity": {
         "caCertificate": "path/to/ca_cert.pem",
         "serverCertificate": "path/to/server_cert.pem",
         "serverKey": "path/to/server_key.pem"
     }
      "ddsSecurity": {
             "identityCaCertificate":     "path/to/identityCaCert.pem",
             "permissionsCaCertificate": "path/to/permissionsCaCert.pem",
             "identityCertificate":       "path/to/identityCert.pem",
             "identityKey":               "path/to/identityKey.pem",
             "signedPermissionsFile":     "path/to/signedPermissions.p7s",
             "signedGovernanceFile":      "path/to/signedGovernance.p7s"
      }
   }
}
```

Note: The tilde (~) Linux shortcut for a user home directory is not supported in the JSON configuration file.

Run the Observability script to create the Observability workspace

To configure the Docker workspace for *Observability Framework*, run the `<installdir>/bin/rtiobservability` script with the `-c <json_file>` option.

Warning: The `<installdir>/bin/rtiobservability` script requires Python3. If any Python package dependencies are missing, the script detects them and provides the command to install them. The required packages are detailed in the `<installdir>/resource/app/app_support/observability/requirements.txt` file. The following image shows the types of errors returned when running the script with a missing dependency.

```
$ rtiobservability -c NonSecureLAN.json

*******************************************************************************
*
* The Observability Docker Containers created by this script may include␣
↪images
* from third-parties, including:
*
*    Prometheus
*      (https://hub.docker.com/r/prom/prometheus)
*    Grafana Loki
*      (https://hub.docker.com/r/grafana/loki)
*    Grafana
*      (https://hub.docker.com/r/grafana/grafana-enterprise)
*    NGINX
*      (https://hub.docker.com/_/nginx)
*    OpenTelemetry Collector
*      (https://hub.docker.com/r/otel/opentelemetry-collector-contrib)
*
* Such third-party software is subject to third-party license terms and
* conditions. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-
↪PARTY
* SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND
* CONDITIONS.
*
*******************************************************************************

Do you wish to continue setting up the Connext Observability Framework[Y/
↪n]? Y

Generating configuration for the Connext Observability Framework

2023-08-03 01:36:46,017 - root - ERROR - Some requirements are missing: No␣
↪module named 'jinja2'.
2023-08-03 01:36:46,017 - root - ERROR - Please install them with the␣
↪following command:
    pip3 install -r /home/test/rti_connext_dds-7.2.0/resource/app/app_
↪support/observability/requirements.txt
```

1. Run `<installdir>/bin/rtiobservability -c <json_file>` to configure the Docker

workspace.

```
$ rtiobservability -c NonSecureLAN.json

 ␣
↪*********************************************************************************
  *
  * The Observability Docker Containers created by this script may␣
↪include images
  * from third-parties, including:
  *
  *    Prometheus
  *      (https://hub.docker.com/r/prom/prometheus)
  *    Grafana Loki
  *      (https://hub.docker.com/r/grafana/loki)
  *    Grafana
  *      (https://hub.docker.com/r/grafana/grafana-enterprise)
  *    NGINX
  *      (https://hub.docker.com/_/nginx)
  *    OpenTelemetry Collector
  *      (https://hub.docker.com/r/otel/opentelemetry-collector-contrib)
  *
  * Such third-party software is subject to third-party license terms and
  * conditions. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF␣
↪THIRD-PARTY
  * SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS␣
↪AND
  * CONDITIONS.
  *
 ␣
↪*********************************************************************************

Do you wish to continue setting up the Connext Observability Framework[Y/
↪n]?
```

2. Select `Y/y` (or simply enter) to acknowledge the license statement.

```
Do you wish to continue setting up the Connext Observability Framework[Y/
↪n]? y

Generating configuration for the Connext Observability Framework

2023-07-19 19:28:10,277 - exporter - INFO - Config: {
    "hostname": "localhost",
    "observabilityDomain": 2,
    "otelConfig": null,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
```

(continues on next page)

(continued from previous page)

```
        "controlPort": 19098,
        "rtwPort": null
    },
    "securityConfig": null
}
```

If you attempt to configure an existing Docker workspace for *Observability Framework*, you will see the following warning.

```
$ rtiobservability -c NonSecureOTelLAN.json

␣
↪*********************************************************************************
  *
  *  The Connext Observability Framework already exist in:
  *      /home/rtrentini/rti_workspace/7.2.0/user_config/
↪observability
  *
  *  Remove or rename the directory manually if you want to start␣
↪over ...
  *
␣
↪*********************************************************************************
```

6.2.2 Initialize and Run Docker Containers

Important: An RTI license is always required to run *Observability Collector Service* in a Docker container. The following table indicates the RTI licenses required based on your answers to the questions in the first two columns.

Table 6.3: License Requirements Table

Do you need to secure telemetry data exchanged between applications and *Observability Collector Service* using SECURITY PLUGINS?	Do you need to send telemetry data to *Observability Collector Service* over the WAN using *Real-Time WAN Transport*?	Required License
NO	NO	*Connext Professional*
YES	NO	*Connext Professional* and SECURITY PLUGINS
YES	YES	*Connext Professional* & SECURITY PLUGINS & *Cloud Discovery Service* & *Real-Time WAN Transport*
NO	YES	*Connext Professional* & *Cloud Discovery Service* & *Real-Time WAN Transport*

For instructions on how to install a license file, see Installing the License File in the *RTI Connext Installation Guide*.

After the Docker workspace is configured and created, run `<installdir>/bin/rtiobservability -i` to initialize and run the Docker containers for *Observability Framework*. The `-i` option calls `docker compose up -d` to create the required storage volumes and containers, then starts the containers.

```
$ rtiobservability -i

  ... using Docker version 24.0.4, build 3713ee1.
  ... using Docker Compose version v2.19.1.

Initializing and running the Connext Observability Framework

[+] Running 6/6
✓ Volume "observability_grafana_data"        Created                 ␣
 ↪                                          0.0s
✓ Volume "observability_prometheus_data"     Created                 ␣
 ↪                                          0.0s
✓ Container collector_service_observability  Started                 ␣
 ↪                                          0.2s
✓ Container prometheus_observability         Started                 ␣
 ↪                                          0.2s
✓ Container grafana_observability            Started                 ␣
 ↪                                          0.2s
✓ Container loki_observability               Started                 ␣
 ↪                                          0.2s
```

Three things happen upon running `<installdir>/bin/rtiobservability` with the `-i` option.

1. The Docker images for Grafana Loki, Prometheus, Grafana, and *Observability Collector Service* are pulled from Docker Hub to your local Docker image store. Note that this will only happen if there are no local images found.
2. The Docker data volumes are created for the Prometheus and Grafana data storage.
3. Docker containers for *Observability Framework* are started for the four components (Loki, Prometheus, Grafana, and *Observability Collector Service*).

At this point, the Docker containers used by *Observability Framework* are started and all components should be running.

6.2.3 Verify Docker Containers are Running

To verify that all Docker containers used by *Observability Framework* are running, run the command `docker ps -a`. Examine the STATUS column and verify that all containers report a status of `Up`, as shown below.

```
CONTAINER ID   IMAGE                                    COMMAND                 ␣
 ↪ CREATED         STATUS         NAMES
6651d7ed9810   prom/prometheus:v2.37.5                  "/bin/prometheus --c…"␣
 ↪ 5 minutes ago   Up 5 minutes   prometheus_observability
```

(continues on next page)

(continued from previous page)

```
25050d16b1b5  grafana/grafana-enterprise:9.2.1-ubuntu  "/run.sh"                ␣
↪ 5 minutes ago  Up 5 minutes  grafana_observability
08611ea9b255  rticom/collector-service:<version>       "/rti_connext_dds-7.…"␣
↪ 5 minutes ago  Up 5 minutes  collector_service_observability
55568de5120f  grafana/loki:2.7.0                      "/usr/bin/loki --con…"␣
↪ 5 minutes ago  Up 5 minutes  loki_observability
```

When a container does not start, the STATUS column displays `Restarting` to indicate the `prometheus-observability` container failed to start and repeatedly tried to restart.

```
CONTAINER ID  IMAGE                                       COMMAND                ␣
↪ CREATED        STATUS                            NAMES
08f75e0fadb2  prom/prometheus:v2.37.5                     "/bin/prometheus --c…"␣
↪ 5 minutes ago  Restarting (1) 27 seconds ago  prometheus_observability
9a3964b561ec  grafana/loki:2.7.0                      "/usr/bin/loki --con…"␣
↪ 5 minutes ago  Up 5 minutes                     loki_observability
b6a6ffa201f3  rticom/collector-service:<version>       "/rti_connext_dds-7.…"␣
↪ 5 minutes ago  Up 5 minutes                     collector_service_
↪observability
26658f76cfdc  grafana/grafana-enterprise:9.2.1-ubuntu  "/run.sh"                ␣
↪ 5 minutes ago  Up 5 minutes                     grafana_observability
```

If a container fails to start, refer to section *Docker Container[s] Failed to Start* for troubleshooting suggestions.

6.2.4 Configure Grafana

Initial Login

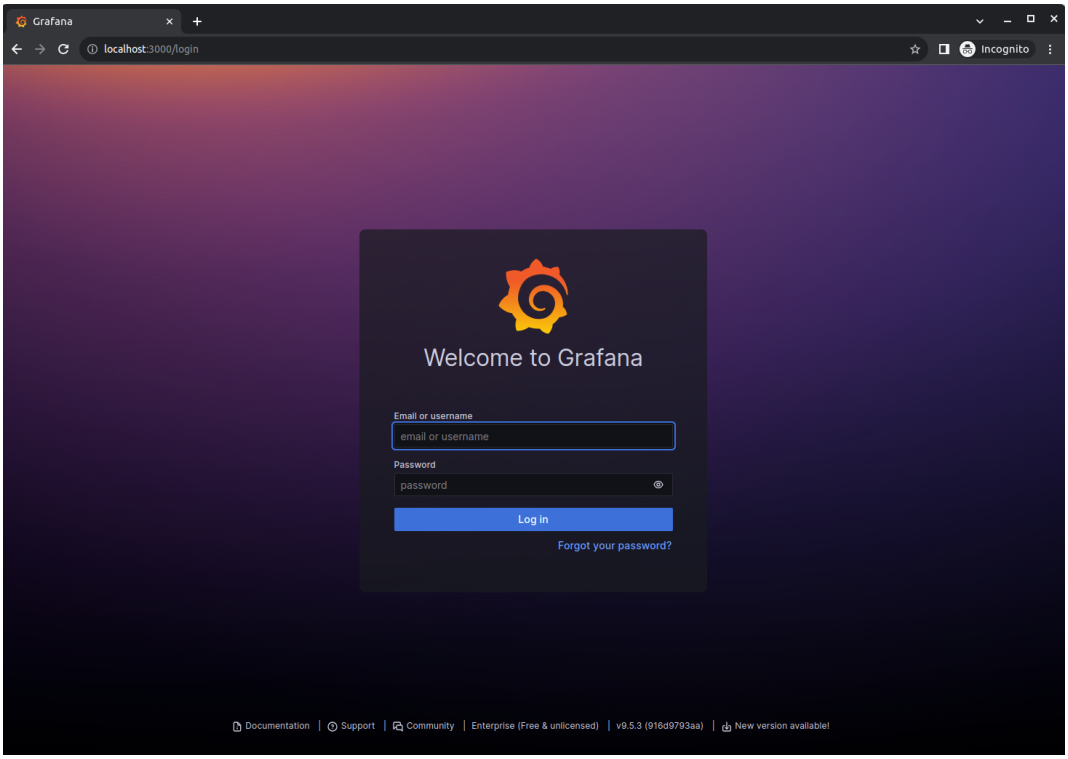
To access *Observability Dashboards*, open a new browser window and go to **http://<hostname>:<grafana-Port>** to access Grafana (3000 is the default grafanaPort). Log in using the credentials **admin : admin**, then change the password when prompted.

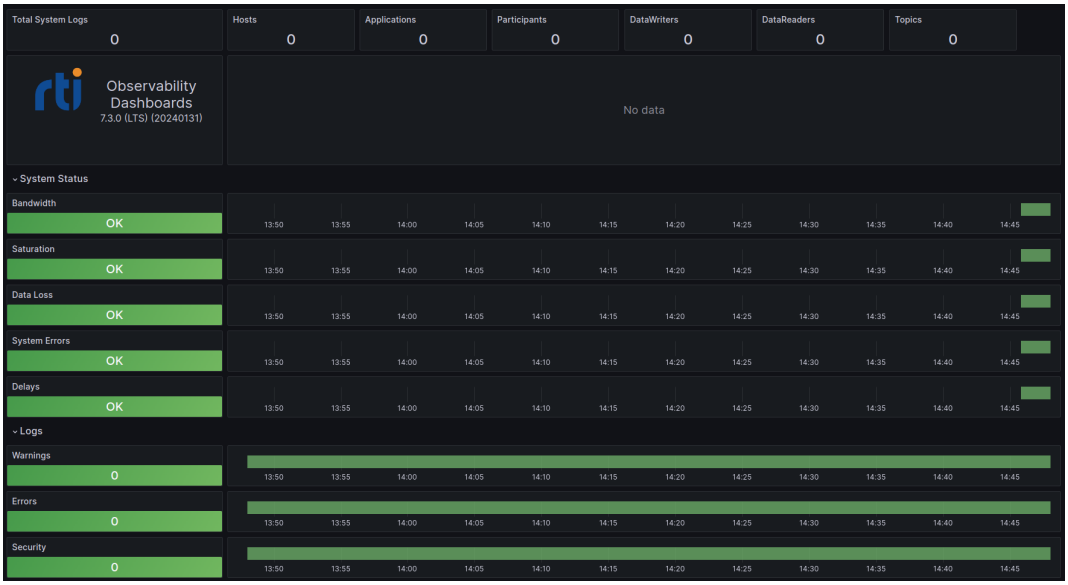
If you are using a secure configuration, the url to access Grafana will be **https://<hostname>:<grafanaPort>** and the Grafana credentials will be the values configured in the **basicAuthUsername** and **basicAuthPassword** fields in the JSON configuration.

Once you are logged in you will see the **RTI Alert Home** dashboard.

Configuration Options

You can configure the Grafana dashboard to meet your specific needs. For more information, refer to the Grafana article Use dashboards.


Grafana
localhost:3000/login
Incognito
Welcome to Grafana
Email or username
email or username
Password
password
Log in
Forgot your password?
Documentation
Support
Community
Enterprise (Free & unlicensed)
v9.5.3 (916d9793aa)
New version available!


Total System Logs
0
Hosts
0
Applications
0
Participants
0
DataWriters
0
DataReaders
0
Topics
0
Observability Dashboards
7.3.0 (LTS) (20240131)
No data
System Status
Bandwidth
OK
Saturation
OK
Data Loss
OK
System Errors
OK
Delays
OK
Logs
Warnings
0
Errors
0
Security
0
13:50
13:55
14:00
14:05
14:10
14:15
14:20
14:25
14:30
14:35
14:40
14:45

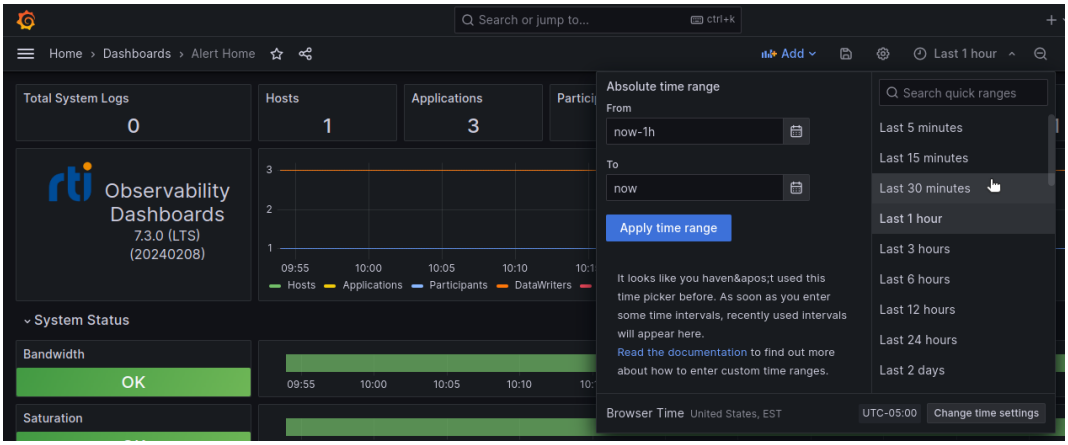
Create Accounts (Optional)

You can create additional users as needed. Refer to the Grafana article Manage Grafana Users for information about user roles and permissions.

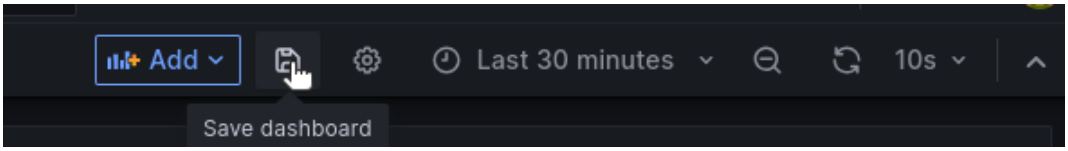
Change the Default Time Range (Optional)

The default visualization time range can be modified. The default relative time range is one hour. You may want to update the range as follows:

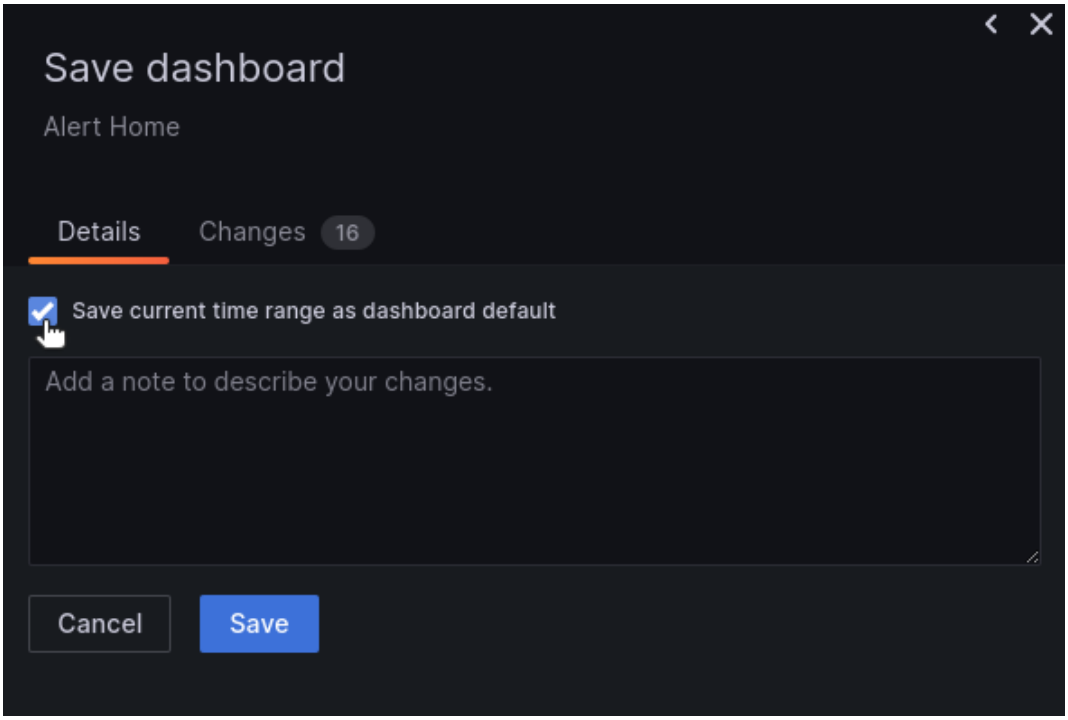
1. Go to the **Alert Home** dashboard,
2. From the toolbar, select the **time picker**.
3. Select the desired time range from the dropdown list. The dashboard refreshes to display the selected time range.



4. From the toolbar, select **Save dashboard**.



5. In the **Save dashboard** dialog, select **Save current time range as dashboard default** and then click **Save**.
6. To confirm the new time range, navigate to another dashboard and then click the Home icon at the top left to go back to the Alert Home dashboard.



6.2.5 Stop Docker Containers

Once *Observability Framework* Docker containers are running, you can stop them by running `<installdir>/bin/rtiobservability -t`. The `-t` option terminates the running Docker containers for *Observability Framework* by calling `docker compose stop`.

```
$ rtiobservability -t

  ... using Docker version 24.0.4, build 3713ee1.
  ... using Docker Compose version v2.19.1.

Terminating the running Connext Observability Framework

[+] Stopping 4/4
✓ Container collector_service_observability  Stopped                ␣
↪                                          10.1s
✓ Container prometheus_observability         Stopped                ␣
↪                                           0.1s
✓ Container grafana_observability            Stopped                ␣
↪                                           0.2s
✓ Container loki_observability               Stopped                ␣
↪                                           2.1s
```

This command stops the existing Docker containers for *Observability Framework*, but leaves associated storage volumes and configuration for a future run.

6.2.6 Start Existing Docker Containers

To restart existing Docker containers used by *Observability Framework*, run `<installdir>/bin/rtiobservability -s`. The `-s` option starts existing Docker containers for *Observability Framework* by calling `docker compose start`.

```
$ rtiobservability -s

  ... using Docker version 24.0.4, build 3713ee1.
  ... using Docker Compose version v2.19.1.

Starting the existing Connext Observability Framework

[+] Running 4/4
✓ Container prometheus_observability          Started                 ␣
 ↪                                          0.1s
✓ Container collector_service_observability   Started                 ␣
 ↪                                          0.1s
✓ Container grafana_observability             Started                 ␣
 ↪                                          0.1s
✓ Container loki_observability                Started                 ␣
 ↪                                          0.2s
```

This command starts any existing Docker containers created by *Observability Framework*.

6.2.7 Stop and Remove Docker Containers

To clean up, or stop and remove, all Docker containers and storage volumes used by *Observability Framework*, run `<installdir>/bin/rtiobservability -d`. The `-d` option stops and removes Docker containers for *Observability Framework* by calling `docker compose down`, and subsequently removes storage volumes.

Warning: Running `<installdir>/bin/rtiobservability -d` removes all Docker containers and storage volumes used by *Observability Framework*. This command removes all changes to your current *Observability Framework* Docker environment including:

- metric data in Prometheus
- log data in Loki
- all Grafana user and dashboard configurations

```
$ rtiobservability -d

  ... using Docker version 24.0.4, build 3713ee1.
  ... using Docker Compose version v2.19.1.


 ␣
 ↪********************************************************************************
```

(continues on next page)

(continued from previous page)

```
  *
  * You have requested to clean up and remove the existing Connext␣
↪Observability
  * Framework.  If you continue you will lose all changes to your current
  * environment including:
  *     - metric data in Prometheus
  *     - log data in Loki
  *     - all Grafana user and dashboard configuration
  *
 ␣
↪*********************************************************************************

Do you wish to continue cleaning and removing the existing Connext␣
↪Observability Framework[y/N]?
```

When prompted to confirm that you want to remove all Docker containers and storage volumes for *Observability Framework*:

- Select `N/n` (or simply enter) to cancel the cleanup.

```
Do you wish to continue cleaning and removing the existing Connext␣
↪Observability Framework[y/N]? n

Cleaning up and removing the existing Connext Observability Framework␣
↪canceled.
```

- Select `Y/y` to proceed with the cleanup and remove all Docker containers and storage volumes used by *Observability Framework*.

```
Do you wish to continue cleaning and removing the existing Connext␣
↪Observability Framework[y/N]? y

Cleaning up and removing the existing Connext Observability Framework

[+] Running 4/5
✓ Container prometheus_observability         Removed                          ␣
↪                                    0.1s
✓ Container grafana_observability            Removed                          ␣
↪                                    0.1s
✓ Container loki_observability               Removed                          ␣
↪                                    1.5s
✓ Container collector_service_observability  Removed                          ␣
↪                                   10.1s
observability_grafana_data
observability_prometheus_data
```

6.2.8 Removing the Docker Workspace for Observability Framework

There may be a time that you need to remove your existing Docker Workspace for *Observability Framework*. This could be because you want to change the existing configuration in some way. Things that you would want to change could include hostname, port configurations, and enabling or disabling security. The `rtiobservability` script will not overwrite an existing workspace. This prevents inadvertently corrupting or deleting an existing configuration. The following steps should be followed to remove an existing workspace to allow re-configuration.

1. You must first stop and remove any existing containers created with the current workspace configuration as detailed in section *Stop and Remove Docker Containers*.
2. Once the docker containers have been stopped and removed you must manually delete the `rti_workspace/<version>/user_config/observability` directory.

   Linux

```
$ rm -rf <path_to_workspace>/<version>/user_config/observability
```

Chapter 7

Getting Started Guide

7.1 About the Observability Example

Observability Framework includes a C++ example that you can use to evaluate the capabilities of this experimental product. The example is installed in your rti_workspace directory, in the `/examples/observability/c++` folder.

> **Attention:** The provided C++ example is not supported on VxWorks® and Android™ platforms.

This section details how the example is configured and how to run it. When you are ready to test the example, refer to the sections *Before Running the Example* and *Running the Example* for instructions.

Important: *Observability Framework* is an experimental product that includes example configuration files for use with several third-party components (Prometheus, Grafana Loki, and Grafana). This release is an evaluation distribution; use it to explore the new observability features that support *Connext* applications.

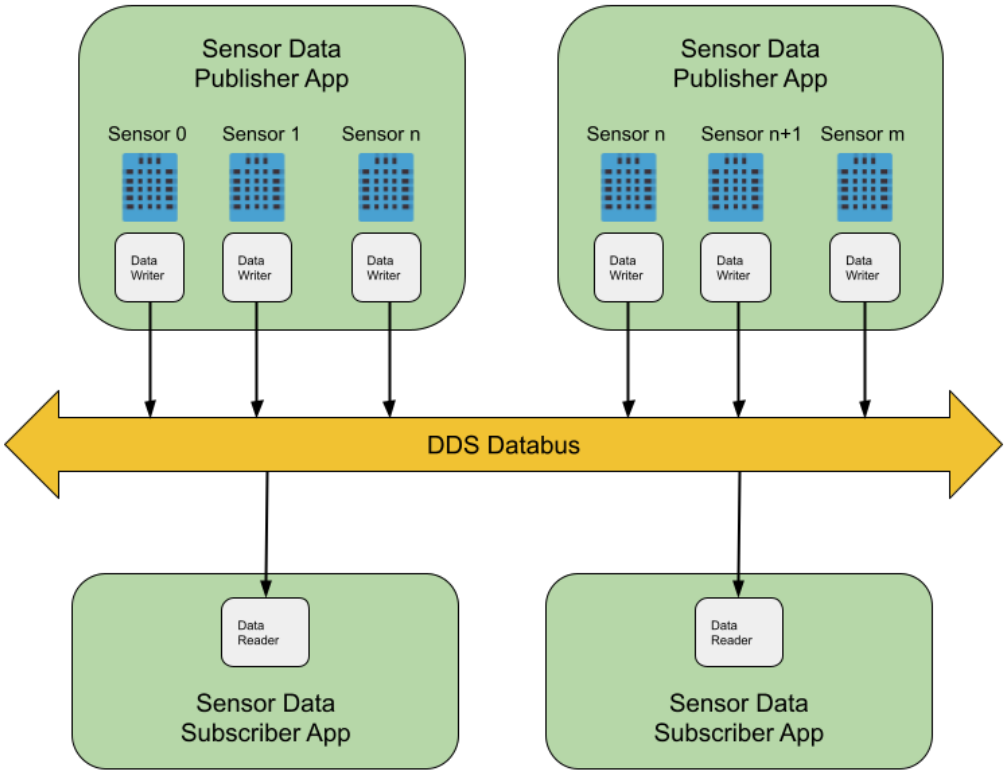
Do not deploy any *Observability Framework* components in production.

7.1.1 Applications

The example consists of two applications:

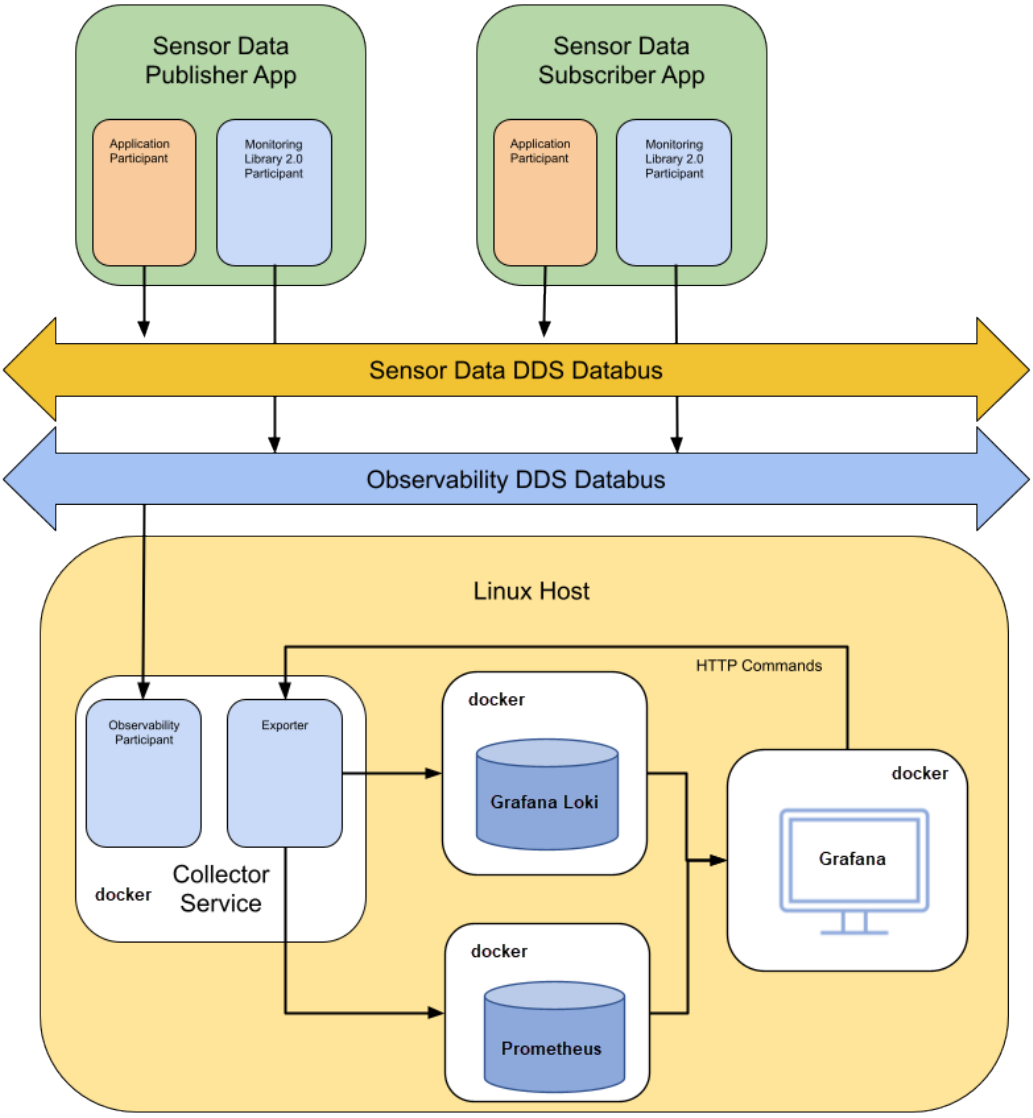
- One application publishes simulated data generated by temperature sensors.
- One application subscribes to the sensor data generated by the temperature sensors.

You can run multiple publishing and subscribing applications in the same host, or in multiple hosts, within a LAN. Each publishing application can handle multiple sensors, and each subscribing application subscribes to all sensors.


Sensor Data
Publisher App
Sensor 0
Sensor 1
Sensor n
Data
Writer
Data
Writer
Data
Writer
Sensor Data
Publisher App
Sensor n
Sensor n+1
Sensor m
Data
Writer
Data
Writer
Data
Writer
DDS Databus
Data
Reader
Sensor Data
Subscriber App
Data
Reader
Sensor Data
Subscriber App

To learn more about the publish/subscribe model, refer to Publish/Subscribe in the *RTI Connext Getting Started Guide*.

The example applications use *Monitoring Library 2.0* to forward telemetry data (logs and metrics) to *Observability Collector Service*. The collector stores this data in Prometheus (metrics) and Grafana Loki (logs) for analysis and visualization using Grafana.



7.1.2 Data Model

The DDS data model for the Temperature topic used in this example is as follows:

```
// Temperature data type
struct Temperature {
// ID of the sensor sending the temperature
@key uint32 sensor_id;
// Degrees in Celsius
int32 degrees;
};
```

Each sensor represents a different instance in the Temperature topic. For general information about data types and topics, refer to Introduction to DataWriters, DataReaders, and Topics and Data Types in the *RTI Connext Getting Started Guide*.

7.1.3 DDS Entity Mapping

The Publisher application creates one *DomainParticipant* and n-*DataWriters*, where *n* is the number of sensors published by the application. This number is configurable using the command `--sensor-count`. Each *DataWriter* publishes one instance. Refer to Keys and Instances in the *RTI Connext Getting Started Guide* for more information on instances.

The Subscriber application creates one *DomainParticipant* and a single DataReader to subscribe to all sensor data.

7.1.4 Command-Line Parameters

The following command-line switches are available when starting the Publisher and Subscriber applications included in the example. Use this information as a reference when you *run the example*.

Publishing Application

Table 7.1: Publishing Application

Parameter	Data Type	Description	Default
-n, –application-name	<str>	Application name	SensorPublisher_<init_sensor_id>
-d, –domain	<int>	Application domain ID	0
-i, –init-sensor-id	<int>	Initial sensor ID	0
-s, –sensor-count	<int>	Sensor count. Each sensor writes one instance published by a separate *DataWriter*	1
-o, –observability-domain	<int>	Domain for sending telemetry data	2
-c, –collector-peer	<str>	Collector service peer	localhost
-v, –verbosity	<int>	How much debugging output to show, range 0-3	1
-p, –protected	N/A	Enable security	disabled

The publishing applications should not publish information for the same sensor IDs. To avoid this issue, you will use the `-i` command-line parameter to specify the sensor ID to be used as the initial ID when *Running the Example*.

Subscribing Application

Table 7.2: Subscribing Application

Parameter	Data Type	Description	Default
-n, –application-name	<str>	Application name	SensorSubscriber
-d, –domain	<int>	Application domain ID	0
-o, –observability-domain	<int>	Domain for sending telemetry data	2
-c, –collector-peer	<str>	Collector service peer	localhost
-v, –verbosity	<int>	How much debugging output to show, Range 0-3	1
-p, –protected	N/A	Enable security	disabled

7.2 Before Running the Example

7.2.1 Set Up Environment Variables

Set up the environment variables for running and compiling the example:

1. Open a command prompt window.
2. Run this script:

   Linux

   ```
   $ source <installdir>/resource/scripts/rtisetenv_<architecture>.bash
   ```

   If you're using the Z shell, run this:

   ```
   $ source <installdir>/resource/scripts/rtisetenv_<architecture>.zsh
   ```

   macOS

   ```
   $ source <installdir>/resource/scripts/rtisetenv_<architecture>.bash
   ```

   If you're using the Z shell, run this:

   ```
   $ source <installdir>/resource/scripts/rtisetenv_<architecture>.zsh
   ```

   If you're using the tcsh shell, run this:

   ```
   $ source <installdir>/resource/scripts/rtisetenv_<architecture>.tcsh
   ```

   Windows

   ```
   > <installdir>/resource/scripts/rtisetenv_<architecture>.bat
   ```

`<installdir>` refers to the installation directory for *Connext*.

The rtisetenv script adds the location of the SDK libraries (`<installdir>/lib/<architecture>`) to your library path, sets the `<NDDSHOME>` environment variable to point to `<installdir>`, and puts the *RTI Code Generator* tool in your path. You may need *Code Generator* if the makefile for your architecture is not available under the *make* directory in the example.

Your architecture (such as `x64Linux3gcc7.3.0`) is the combination of processor, OS, and compiler version that you will use to build your application. For example:

```
$ source $NDDSHOME/resource/scripts/rtisetenv_x64Linux4gcc7.3.0.bash
```

7.2.2 Compile the Example

Monitoring Library 2.0 can be used in three different ways:

- **Dynamically loaded:** This method requires that the rtimonitoring2 shared library is in the library search path.
- **Dynamic linking:** The application is linked with the rtimonitoring2 shared library.
- **Static linking:** The application is linked with the rtimonitoring2 static library.

You will compile the example using *Connext* shared libraries so that *Monitoring Library 2.0* can be dynamically loaded. The example is installed in your rti_workspace directory, in the `/examples/observability/c++` folder.

Non-Windows Systems

To build this example on a non-Windows system, type the following in a command shell from the example directory:

```
$ make -f make/makefile_Temperature_<architecture> DEBUG=0
```

If there is no makefile for your architecture in the *make* directory, you can generate it using the following `rtiddsgen` command:

```
$ rtiddsgen -language C++98 -create makefiles -platform <architecture> -
↪sharedLib -sourceDir src -d ./make ./src/Temperature.idl
```

Windows Systems

To build this example on Windows, open the appropriate solution file for your version of Microsoft Visual Studio in the win32 directory. To use dynamic linking, select **Release DLL** from the dropdown menu.

7.2.3 Install Observability Framework

Before running the example, make sure that you have installed both *Monitoring Library 2.0* and the collection, storage and visualization components. Refer to the *Installing and Running Observability Framework* section for instructions.

If you want to run the example with security enabled, you must install *Observability Framework* using a secure configuration. If you did not create a secure configuration, delete the existing configuration as described in section *Removing the Docker Workspace for Observability Framework*, then update your JSON configuration file as needed. The following sections include example configuration files you can edit for your environment.

The collection, storage, and visualization components can be installed using one of two methods:

- Install the components in a Linux host on the same LAN where the applications run, or
- Install the components on a remote Linux host (for example, an AWS instance) reachable over the WAN using *Real-Time WAN Transport*.

Both methods support secure and non-secure configurations.

To facilitate testing secure configurations where all components run on the same node (docker images, test applications, and browser), artifacts are provided in your `rti_workspace` directory, in the `/examples/dds-security/` folder. The artifacts provided to secure the https connections use the hostname "localhost".

The following sections provide example JSON configurations for each of the eight operation modes supported by *Observability Framework*. These examples use the hostname "localhost", default port values, and the paths to the default security artifacts where appropriate. You can copy these examples to a local file and use them as is, or customize them with your own hostname, ports, and security artifacts. For details on how to configure *Observability Framework*, see section *Configure the JSON File*.

Configure Observability Framework for the Appropriate Operation Mode

---

Important: The provided example configurations work only if you run ALL components (docker images, test applications, and browser) on the same host machine. If you intend to run any components (test applications or browser) on a remote machine, you must update the `hostname` field in the JSON configuration file to the hostname of the machine running *Observability Framework*.

Additionally, if you run in secure mode, you will need to generate the https security artifacts and the DDS security artifacts as shown in *Generating the Observability Framework Security Artifacts*. Once you have generated your artifacts, you will need to update the `securityConfig` section in the JSON configuration file with the paths to these artifacts.

There are eight distinct operation modes you can use to configure *Observability Framework*. These modes, described below, are based on the desired security level, network environment (LAN or WAN), and use of the OpenTelemetry Collector.

1. Select the operation mode for the test you want to run, then edit your JSON configuration file with the selected content. For example, if you want to test on a LAN without security, copy the example JSON from section *Non-Secure LAN Configuration* to the `config.json` file.
2. If desired, modify the hostname, ports, or security artifact paths in the `config.json` file. For example, to use port 9091 for Prometheus, change the "prometheusPort" field in the `config.json` file from 9090 to 9091.
3. Run the rtiobservability script to apply your *Observability Framework* configuration.

   Linux

```
$ rtiobservability -c config.json
```

If you have already configured *Observability Framework* in a different operation mode than the one you want to test, you must first remove the existing workspace as described in section *Removing the Docker Workspace for Observability Framework*.

Example LAN configurations

Table 7.3 lists the four LAN configurations supported by *Observability Framework*.

Table 7.3: Docker Container LAN Operation Modes

Configuration Name	Network	Data Storage	Security
NonSecureLAN	LAN	Prometheus and Grafana Loki	No
SecureLAN	LAN	Prometheus and Grafana Loki	Yes
NonSecureOTelLAN	LAN	Multiple through OpenTelemetry Collector	No
SecureOTelLAN	LAN	Multiple through OpenTelemetry Collector	Yes

Non-Secure LAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security disabled. *Observability Collector Service* will use a LAN configuration.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098
    }
}
```

Secure LAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security enabled. *Observability Collector Service* will use a LAN configuration.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
```

(continues on next page)

(continued from previous page)

```
        "controlPort": 19098
    },
    "securityConfig": {
        "basicAuthUsername": "user",
        "basicAuthPassword": "userpassword",
        "httpsSecurity": {
        "caCertificate":     "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/ca/ecdsa01RootCaCert.pem",
        "serverCertificate": "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Cert.pem",
        "serverKey":         "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":    "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "permissionsCaCertificate": "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "identityCertificate":      "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Cert.pem",
            "identityKey":              "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Key.pem",
            "signedPermissionsFile":    "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityCollectorServicePermissions.p7s",
            "signedGovernanceFile":     "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityGovernance.p7s"
        }
    }
}
```

Non-Secure OTel LAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security disabled. *Observability Collector Service* will use a LAN configuration and the OpenTelemetry exporter. The OpenTelemetry Collector routes telemetry data from the *Observability Collector Service* OpenTelemetry exporter to Prometheus and Loki.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098
    },
```

(continues on next page)

(continued from previous page)

```
    "otelConfig": {
        "otelHttpReceiverPort": 4318
    }
}
```

Secure OTel LAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security enabled. *Observability Collector Service* will use a LAN configuration and the OpenTelemetry exporter. The OpenTelemetry Collector routes telemetry data from the *Observability Collector Service* OpenTelemetry exporter to Prometheus and Loki.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098
    },
    "otelConfig": {
        "otelHttpReceiverPort": 4318
    },
    "securityConfig": {
        "basicAuthUsername": "user",
        "basicAuthPassword": "userpassword",
        "httpsSecurity": {
        "caCertificate":     "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/ca/ecdsa01RootCaCert.pem",
        "serverCertificate": "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Cert.pem",
        "serverKey":         "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":    "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "permissionsCaCertificate": "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "identityCertificate":      "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Cert.pem",
            "identityKey":              "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Key.pem",
            "signedPermissionsFile":    "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityCollectorServicePermissions.p7s",
```

(continues on next page)

(continued from previous page)

```
            "signedGovernanceFile":     "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityGovernance.p7s"
        }
    }
}
```

Example WAN configurations

Table 7.4 lists the four WAN configurations supported by *Observability Framework*.

Table 7.4: Docker Container WAN Operation Modes

Configuration Name	Network	Data Storage	Security
NonSecureWAN	WAN	Prometheus and Grafana Loki	No
SecureWAN	WAN	Prometheus and Grafana Loki	Yes
NonSecureOTelWAN	WAN	Multiple through OpenTelemetry Collector	No
SecureOTelWAN	WAN	Multiple through OpenTelemetry Collector	Yes

Non-Secure WAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security disabled. *Observability Collector Service* will use a WAN configuration with port 30000.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098,
        "rtwPort": 30000
    }
}
```

Secure WAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security enabled *Observability Collector Service* will use a WAN configuration with port 30000.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098,
        "rtwPort": 30000
    },
    "securityConfig": {
        "basicAuthUsername": "user",
        "basicAuthPassword": "userpassword",
        "httpsSecurity": {
        "caCertificate":     "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/ca/ecdsa01RootCaCert.pem",
        "serverCertificate": "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Cert.pem",
        "serverKey":         "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":    "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "permissionsCaCertificate": "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "identityCertificate":      "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Cert.pem",
            "identityKey":              "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Key.pem",
            "signedPermissionsFile":    "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityCollectorServicePermissions.p7s",
            "signedGovernanceFile":     "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityGovernance.p7s"
        }
    }
}
```

Non-Secure OTel WAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security disabled. *Observability Collector Service* will use a WAN configuration with port 30000 and the OpenTelemetry exporter. The OpenTelemetry Collector routes telemetry data from *Observability Collector Service* Service* OpenTelemetry exporter to Prometheus and Loki.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098,
        "rtwPort": 30000
    },
    "otelConfig": {
        "otelHttpReceiverPort": 4318
    }
}
```

Secure OTel WAN Configuration

This example configures *Observability Framework* with hostname "localhost", default ports, and security enabled. *Observability Collector Service* will use a WAN configuration with port 30000 and the OpenTelemetry exporter. The OpenTelemetry Collector routes telemetry data from the *Observability Collector Service* OpenTelemetry exporter to Prometheus and Loki.

```
{
    "hostname": "localhost",
    "observabilityDomain": 2,
    "lgpStackConfig": {
        "grafanaPort": 3000,
        "prometheusPort": 9090,
        "lokiPort": 3100
    },
    "collectorConfig": {
        "prometheusExporterPort": 19090,
        "controlPort": 19098,
        "rtwPort": 30000
    },
    "otelConfig": {
        "otelHttpReceiverPort": 4318
    },
    "securityConfig": {
        "basicAuthUsername": "user",
```

(continues on next page)

(continued from previous page)

```
        "basicAuthPassword": "userpassword",
        "httpsSecurity": {
        "caCertificate":     "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/ca/ecdsa01RootCaCert.pem",
        "serverCertificate": "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Cert.pem",
        "serverKey":         "<rti_workspace_dir>/examples/dds_security/cert/
↪ecdsa01/https/ecdsa01Https01Key.pem"
        },
        "ddsSecurity": {
            "identityCaCertificate":     "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "permissionsCaCertificate": "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/ca/ecdsa01RootCaCert.pem",
            "identityCertificate":      "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Cert.pem",
            "identityKey":              "<rti_workspace_dir>/examples/dds_
↪security/cert/ecdsa01/identities/ecdsa01Peer01Key.pem",
            "signedPermissionsFile":    "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityCollectorServicePermissions.p7s",
            "signedGovernanceFile":     "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityGovernance.p7s"
        }
    }
}
```

7.2.4 Start the Collection, Storage, and Visualization Docker Containers

The Docker containers used for data collection, storage, and visualization can either be run in a Linux host on the same LAN where the applications run or they can be installed on a remote Linux host (for example, an AWS instance) reachable over the WAN using *Real-Time WAN Transport*.

There may be different licensing requirements depending on the configuration (LAN/WAN, Secure/Non-Secure) you have chosen to run. For details on the license requirements and instructions on how to run the containers, see section *Initialize and Run Docker Containers*.

7.3 Running the Example

Table 7.5 lists optional command-line parameters you can use when running the *Observability Framework* example. Choose the options appropriate for your test environment.

Table 7.5: Optional Command-Line Parameters

Parameter	Description	Default Value
`--observability-domain`	Use this command-line option if you want to overwrite the default domain ID used by *Monitoring Library 2.0* to send telemetry data to *Observability Collector Service*.	2
`--collector-peer`	If you run *Observability Collector Service* in a different host from the applications, use this command-line option to provide the address of the service. For example, 192.168.1.1 (for LAN), or udpv4_wan://10.56.78.89:16000 (for WAN).	localhost

In addition, if you run the applications in different hosts and multicast is not available, use the NDDS_DISCOVERY_PEERS environment to configure the peers where the applications run.

For simplicity, the following instructions assume that you are running the applications and the Docker containers used by *Observability Framework* on the same host using the default observability domain.

7.3.1 Start the Applications

This example assumes `x64Linux4gcc7.3.0` as the architecture. The following steps include instructions for non-secure and secure tests.

1. In a new browser window, go to **http[s]://localhost:3000** and log in using your Grafana dashboard credentials. Note the use of `https` if you are running a secure configuration.

   The default Grafana dashboard credentials are `admin:admin` for non-secure configurations, and `user:userpassword` for secure configurations (as configured in the JSON file).

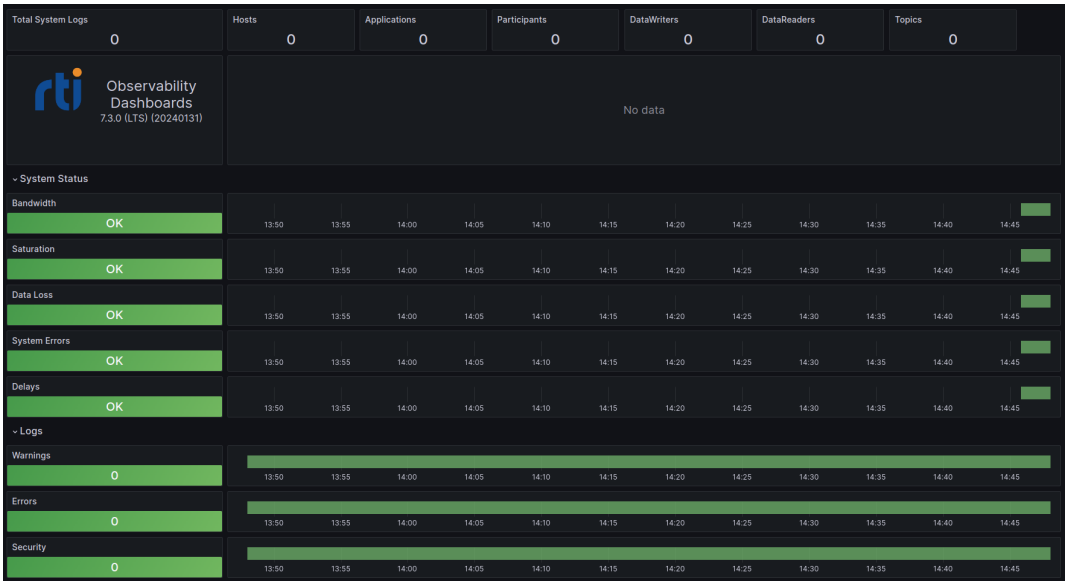
   At this point, no DDS applications are running.

2. From the example directory, open two terminals and start two instances of the application that publishes temperature sensor data. The command and resulting output for each instance are shown below.

   The `-i` parameter specifies the sensor ID that will be used. The `-n` parameter assigns a name to the application; this name will be used when sending commands in the *Change the Application Logging Verbosity* and *Change the Metric Configuration* sections of this example. The `-p` parameter enables security when using a secure configuration. See *Command-Line Parameters* for a description of all available options.

   The first instance creates two sensors.

   Non-Secure LAN




```
$ ./objs/x64Linux4gcc7.3.0/Temperature_publisher -n SensorPublisher_1 -d␣
↪57 -i 0 -s 2 -v 2
*********************************************************
******** Temperature Sensor Publisher App ***************
*********************************************************
Running with parameters:
    Application Resource Name: /applications/SensorPublisher_1
    Domain ID: 57
    Init Sensor ID: 0
    Sensor Count: 2
    Observability Domain: 2
    Collector Peer: udpv4://localhost
    Verbosity: 2
    Security: false
Running with QOS:
    Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

Secure LAN

```
$ ./objs/x64Linux4gcc7.3.0/Temperature_publisher -n SensorPublisher_1 -d␣
↪57 -i 0 -s 2 -p -v 2
*********************************************************
******** Temperature Sensor Publisher App ***************
*********************************************************
Running with parameters:
    Application Resource Name: /applications/SensorPublisher_1
    Domain ID: 57
    Init Sensor ID: 0
    Sensor Count: 2
    Observability Domain: 2
    Collector Peer: udpv4://localhost
    Verbosity: 2
```

(continues on next page)

(continued from previous page)

```
    Security: true
Running with QOS:
    Secure_Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

The second instance creates one sensor.

Note that the sensor ids used by different instances of the temperature publisher app should not overlap. The first instance used the switches *-i 0* and *-s 2*, creating two sensors with ids 0 and 1. The second instance used *-i 2* and *-s 1*, creating one sensor with id 2.

Non-Secure LAN

```
$ ./objs/x64Linux4gcc7.3.0/Temperature_publisher -n SensorPublisher_2 -d␣
↪57 -i 2 -s 1 -v 2
************************************************************
******** Temperature Sensor Publisher App ****************
************************************************************
Running with parameters:
    Application Resource Name: /applications/SensorPublisher_2
    Domain ID: 57
    Init Sensor ID: 2
    Sensor Count: 1
    Observability Domain: 2
    Collector Peer: udpv4://localhost
    Verbosity: 2
    Security: false
Running with QOS:
    Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

Secure LAN

```
$ ./objs/x64Linux4gcc7.3.0/Temperature_publisher -n SensorPublisher_2 -d␣
↪57 -i 2 -s 1 -p -v 2
************************************************************
******** Temperature Sensor Publisher App ****************
************************************************************
Running with parameters:
    Application Resource Name: /applications/SensorPublisher_1
    Domain ID: 57
    Init Sensor ID: 2
    Sensor Count: 1
    Observability Domain: 2
    Collector Peer: udpv4://localhost
    Verbosity: 2
    Security: true
Running with QOS:
    Secure_Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

3. From the example directory, open a new terminal and start one instance of the application that subscribes

to temperature sensor data.

Non-Secure LAN

```
$ ./objs/x64Linux4gcc7.3.0/Temperature_subscriber -n SensorSubscriber -d␣
↪57 -v 2
************************************************************
******** Temperature Sensor Subscriber App ****************
************************************************************
Running with parameters:
    Application Resource Name: /applications/SensorSubscriber
    Domain ID: 57
    Observability Domain: 2
    Collector Peer: udpv4://localhost
    Verbosity: 2
    Security: false
Running with QOS:
    Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

Secure LAN

```
$ ./objs/x64Linux4gcc7.3.0/Temperature_subscriber -n SensorSubscriber -d␣
↪57 -p -v 2
************************************************************
******** Temperature Sensor Subscriber App ****************
************************************************************
Running with parameters:
    Application Resource Name: /applications/SensorSubscriber
    Domain ID: 57
    Observability Domain: 2
    Collector Peer: udpv4://localhost
    Verbosity: 2
    Security: true
Running with QOS:
    Secure_Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

Note: The two Publisher applications and the Subscriber application are started with verbosity set to WARNING (-v 2). You may see any of the following warnings on the console output. These warnings are expected.

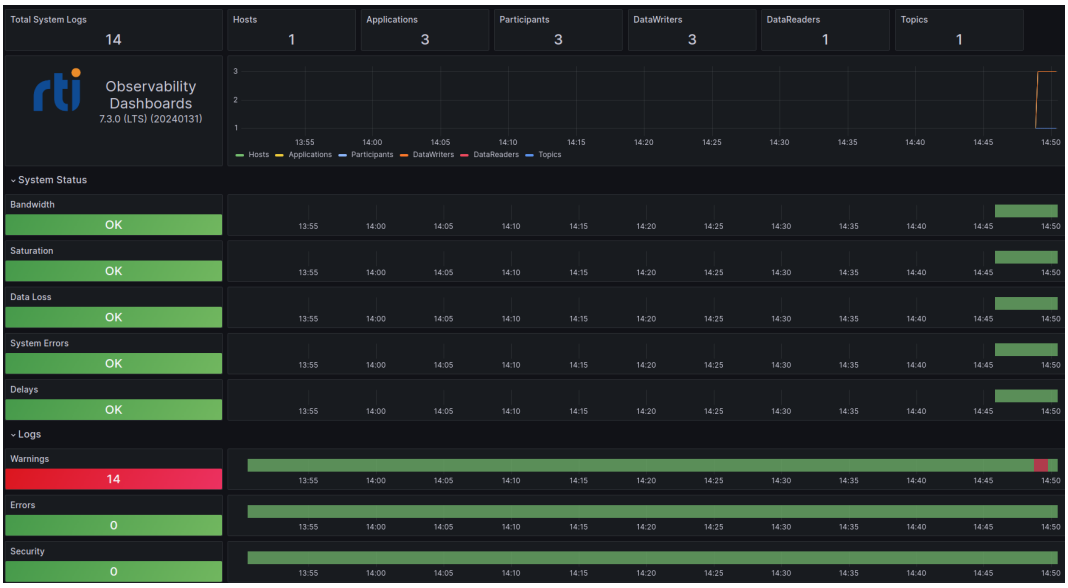
```
WARNING [0x01017774,0xFF40EEF6,0xEC566CA8:0x000001C1{Domain=2}
↪|ENABLE|LC:Discovery]NDDS_Transport_UDPv4_Socket_bind_with_ip:0X1EE6 in use
WARNING [0x01017774,0xFF40EEF6,0xEC566CA8:0x000001C1{Domain=2}
↪|ENABLE|LC:Discovery]NDDS_Transport_UDPv4_SocketFactory_create_receive_
↪socket:invalid port 7910
WARNING [0x01017774,0xFF40EEF6,0xEC566CA8:0x000001C1{Domain=2}
↪|ENABLE|LC:Discovery]NDDS_Transport_UDP_create_recvresource_rrEA:!create␣
↪socket
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature␣
↪DomainParticipant,Domain=57}|ENABLE|LC:Discovery]NDDS_Transport_UDPv4_
↪Socket_bind_with_ip:0X549C in use
```

(continues on next page)

(continued from previous page)

```
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature␣
↪DomainParticipant,Domain=57}|ENABLE|LC:Discovery]NDDS_Transport_UDPv4_
↪SocketFactory_create_receive_socket:invalid port 21660
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature␣
↪DomainParticipant,Domain=57}|ENABLE|LC:Discovery]NDDS_Transport_UDP_create_
↪recvresource_rrEA:!create socket
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature␣
↪DomainParticipant,Domain=57}|ENABLE|LC:Discovery]DDS_
↪DomainParticipantDiscovery_add_peer:no peer locators for: peer␣
↪descriptor(s) = "builtin.shmem://", transports = "", enabled_transports = ""
```

Your Grafana dashboard should now display information about the new Hosts, Applications, and DDS entities (*Participants*, *DataWriters*, and *DataReaders*). There should be 1 Host, 3 Applications, 3 *Participants*, 3 *DataWriters*, 1 *DataReader*, and 1 *Topic*.

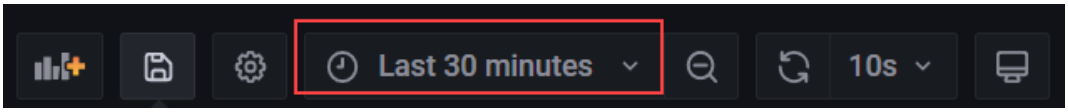


The Grafana main dashboard pictured above indicates that the system is healthy. You may see warnings in the log section related to the reservation of communication ports. These warnings are expected. You can select the Warnings panel to visualize them.

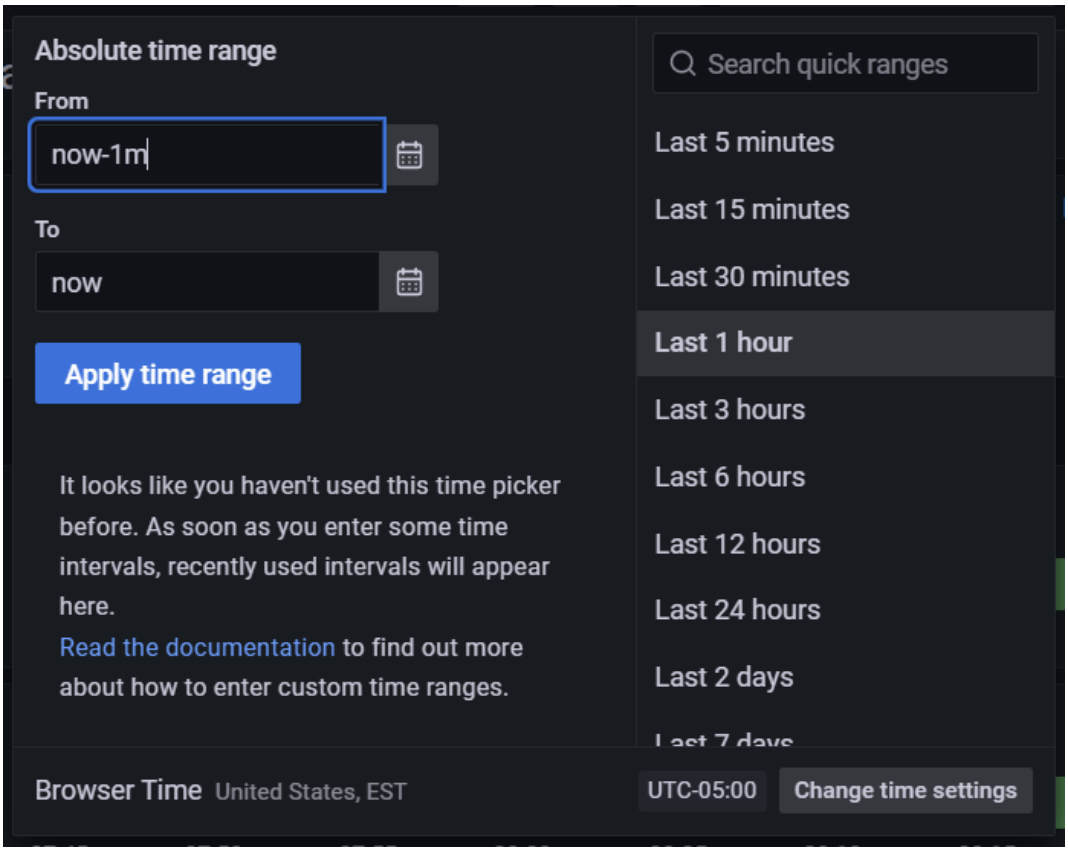
Next, you will introduce different failures that will affect the system's health.

7.3.2 Changing the Time Range in Dashboards

While running the examples, you can change the time range in the dashboards to reduce or expand the amount of history data displayed. Use the time picker dropdown at the top right to change the time range in any dashboard.



The time picker includes a predefined list of time ranges to choose from. If you want to use a custom time range, enter the desired range in the **From** field. Use the format "now-< custom time >," where < custom time > is a unit of time; Grafana supports m-minute, h-hour, and d-day time units. For example, to show a custom range of one minute, enter "now-1m" in the **From** field, then select **Apply Time Range**.



Note: The time range may be changed on any dashboard, but all changes are temporary and will reset to 1 hour when you return to the **Alert Home** dashboard. Changes to the time range made in the **Alert Home** dashboard are unique in that the selected time range will be propagated to other dashboards as you navigate through the hierarchy.

7.3.3 Simulate Sensor Failure

The *DataWriters* in each application are expected to send sensor data every second, and the *DataReader* expects to receive sensor data from each sensor every second. This QoS contract is enforced with the Deadline QoS Policy set in USER_QOS_PROFILES.xml. Refer to Deadline QoS Policy in the *RTI Connext Getting Started Guide* for basic information, or DEADLINE QoSPolicy in the *RTI Connext Core User's Manual* for detailed information.

```
<deadline>
   <period>
      <sec>1</sec>
```

(continues on next page)

(continued from previous page)

```
      <nanosec>0</nanosec>
   </period>
</deadline>
```

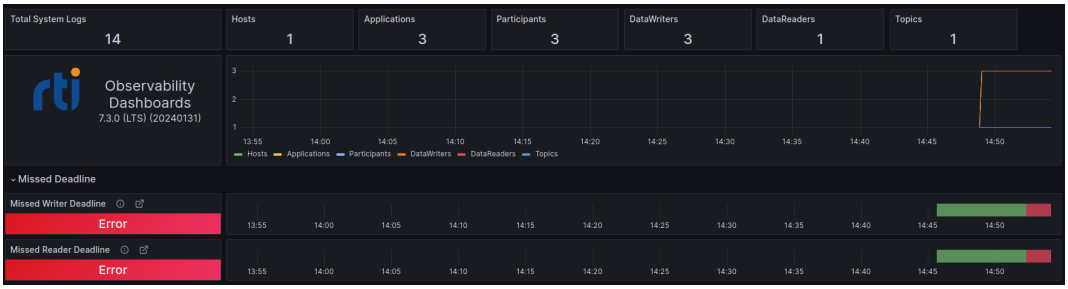
To simulate a failure in the sensor with ID 0, enter the following command in the first Temperature_publisher instance:

```
Command> stop 0
```

The Grafana dashboard updates to indicate the sensor failure. The dashboard does not update immediately; you may have to wait a few seconds to see the change reflecting the sensor failure as a Delay error. That error is expected because the deadline policy was violated when you stopped the sensor with ID 0.

The Grafana dashboards are hierarchical. Now that you know something is happening related to latency (or delays), you can get additional details to determine the root cause. Select the **Delays** panel to drill down to the next level and get more information about the error.

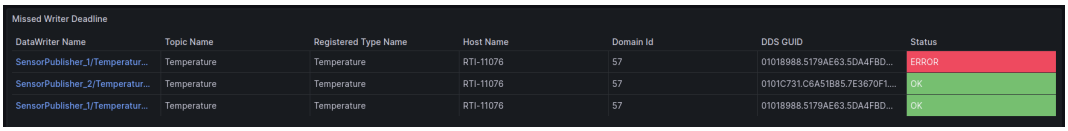
The second level of the Grafana dashboard indicates that there were deadline errors, which can be generated by both the *DataReaders* and *DataWriters* of the sensor *Topic*. Still, we do not know which sensor the problem originated from. To determine that, we have to go one level deeper; select the **Missed Writer Deadline** panel to see which *DataWriter* caused the problem.



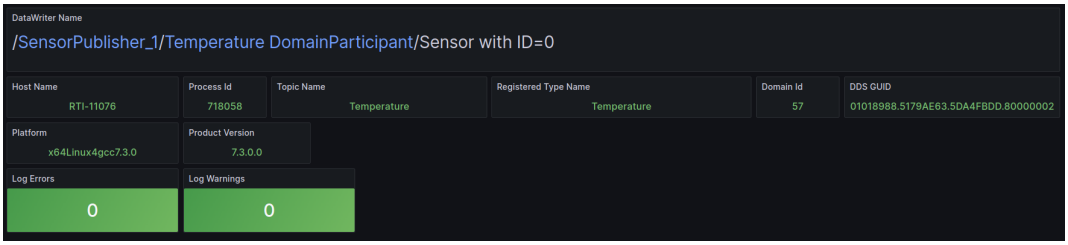
The third level of the Grafana dashboard provides a list of entities generating the deadline metric. In this case

we see three entities, or *DataWriters*, each associated with a different sensor. We see that an entity is failing, but what sensor does that entity represent?

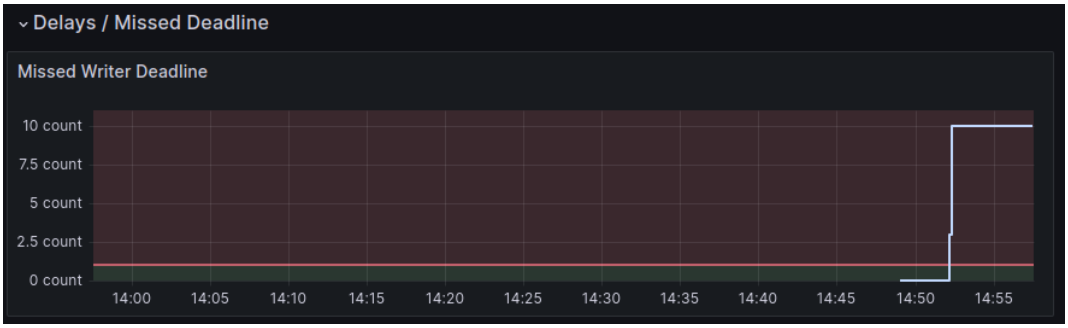
Looking at the *DataWriter* Name column, we can see that the failing sensor has the name "Sensor with ID=0". The example application set this name using the EntityName QoS Policy when creating the *DataWriter*. If you want additional information, such as the machine where the sensor *DataWriter* is located, select the **Sensor with ID-0** link in the *DataWriter* Name column.



The fourth and last level of the Grafana dashboard provides detailed information about an individual entity, including location-related information such as Host Name and Process Id.



In addition, this last level provides information about individual metrics for the entity. Scroll down to see the details of the missed deadline metric.



Next, restore the health of the failing sensor to review how the dashboard changes. Restart the first Temperature_publisher instance using the command `start 0`.

```
Command> start 0
```

Go back to the **Alert Home** dashboard to confirm that the sensor becomes healthy. After a few seconds, the **Delays** panel should indicate the sensor is healthy. Note that the status part of the Delay panel still displays an "Error" state (red) if, at anytime in the displayed time range, there was a "Delay" metric in the system that was considered to be "unhealthy" (a metric whose value exceeded configured limits).

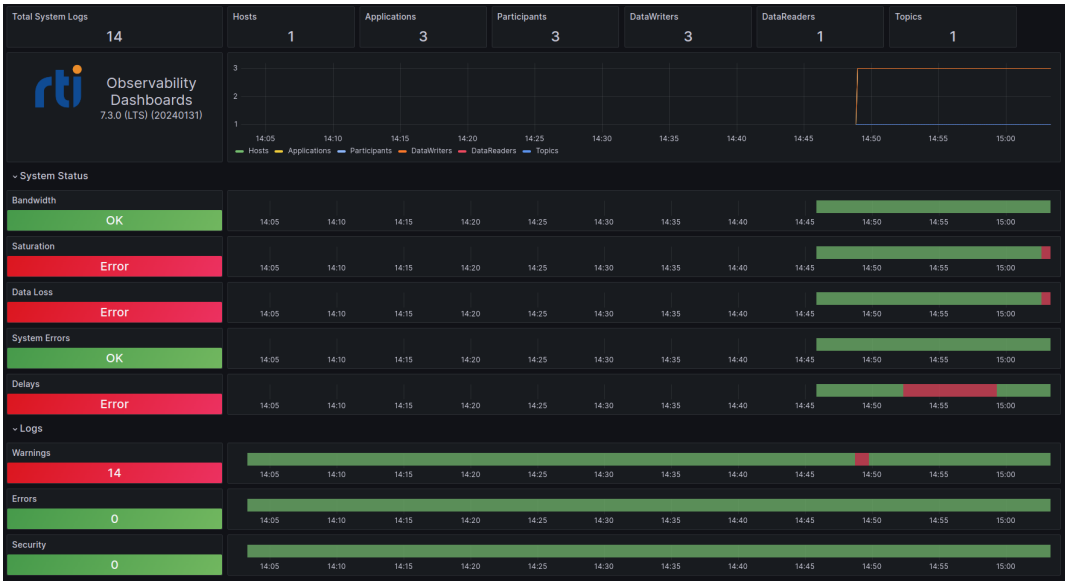
7.3.4 Simulate Slow Sensor Data Consumption

A subscribing application can be configured to consume sensor data at a lower rate than the publication rate. In a real scenario, this could occur if the subscribing application becomes CPU bound.

This scenario simulates a problem with the subscribing application; a bug in the application logic makes it slow in processing sensor data. To test this failure, enter the `slow_down` command in the Temperature_subscriber instance:

```
Command> slow_down
```

After some seconds, the Grafana dashboard displays two new system errors related to saturation and unexpected data losses. Because the *DataReader* is not able to keep up with the sensor data, the dashboard indicates that there are potential data losses. At the same time, being unable to keep up with the sensor data could be a saturation sign. For example, the subscribing application may be consuming 100% of the CPU due to an application bug.

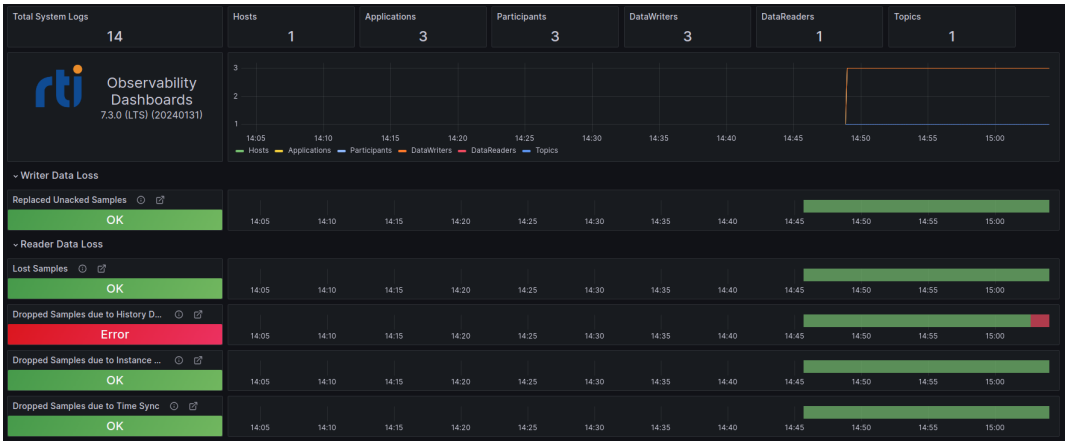


As you did when testing the sensor failure, select the displayed errors to navigate the dashboard hierarchy and determine the root cause of the problem. To go to the second level, select the **Data Loss** panel to see the reason for the losses. Because you slowed the subscriber application, the *DataReader* is not able to read fast enough. The **Dropped Samples due to History Depth** metric reveals the type of failure. Select the red errors to drill down and review further details about the problem.

After reviewing the errors, restore the health of the failing *DataReader*. In the Temperature_subscriber application, enter the `speed_up` command.

```
Command> speed_up
```

In Grafana, go back to the home dashboard and wait until the system becomes healthy again. After a few seconds, the Saturation and Data Loss panels should indicate a healthy system. Also, adjust the time window



to one minute and wait until all the system status panels are green again.

7.3.5 Simulate Time Synchronization Failures

In the example, the subscribing applications have been configured to expect all system clocks are synchronized to within 1 second. The source timestamp associated with a sensor sample by the Publisher should not be farther in the future from the reception timestamp than a configurable tolerance. This behavior is configured using the DestinationOrder QoS Policy set in USER_QOS_PROFILES.xml.

```
<destination_order>
   <kind>BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</kind>
   <source_timestamp_tolerance>
      <sec>1</sec>
      <nanosec>0</nanosec>
   </source_timestamp_tolerance>
</destination_order>
```

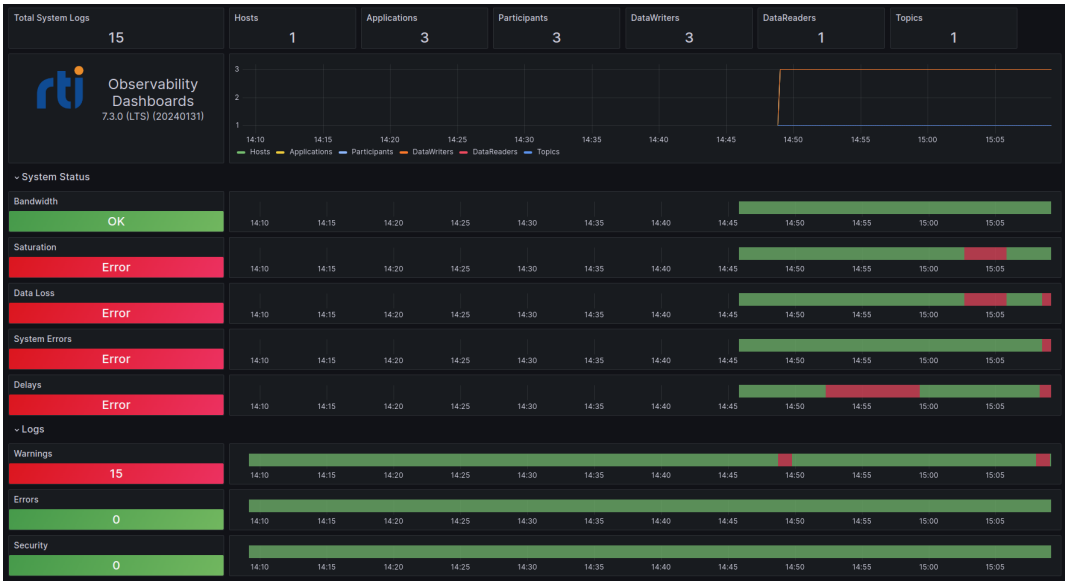
This final simulation demonstrates how to use logging information to troubleshoot problems. In this scenario, you'll create a clock synchronization issue in the first instance of Temperature_publisher. The clock will move forward in the future by a few seconds, causing the *DataReader* to drop some sensor samples from the publishing application.

To simulate this scenario, enter **clock_forward 0** in the first Temperature_publisher instance. This will cause the publishing application to artificially set the clock used for the source timestamp of the *DataWriter* named "Sensor with ID=0" by 2 seconds.

```
Command> clock_forward 0
```

After some seconds, three panels in the system status section will turn red: **Data Loss**, **System Errors**, and **Delays**. Each is affected by the same underlying problem. You can select the red errors to drill down through the dashboard hierarchy and determine the root cause of the problem.

First, select the **Data Loss** panel to see the reason for the error. The *DataReader* dropped samples coming from one or more of the *DataWriters* due to time synchronization issues.



This error indicates that the *DataReader* in the subscribing application dropped some samples, but can't yet identify the problem sensor or *DataWriter*. To determine that, select the **Dropped Samples due to Time Sync** panel.

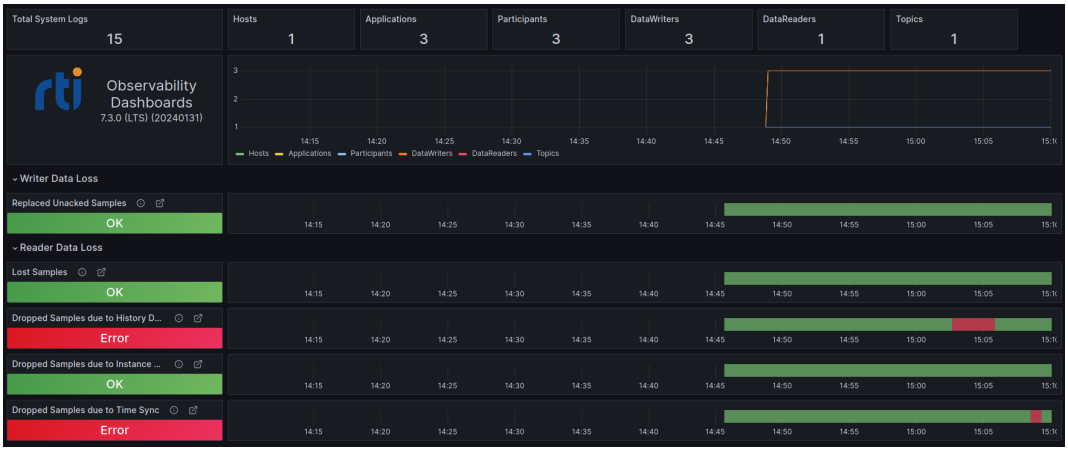
At this level, you can locate the *DataReader* reporting the error, but not the *DataWriter* causing it. Select the **TemperatureSensor** link in the DataReader Name column to go one more level down.

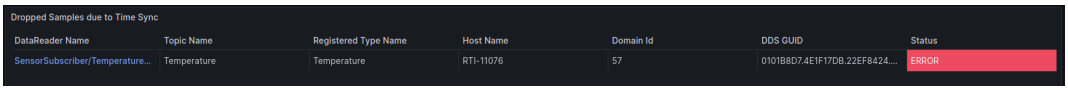
On the endpoint dashboard, there is one log warning associated with the *DataReader* reporting time synchronization issues. Select the red **Log Warning** to view the warning message logged by the *DataReader*.

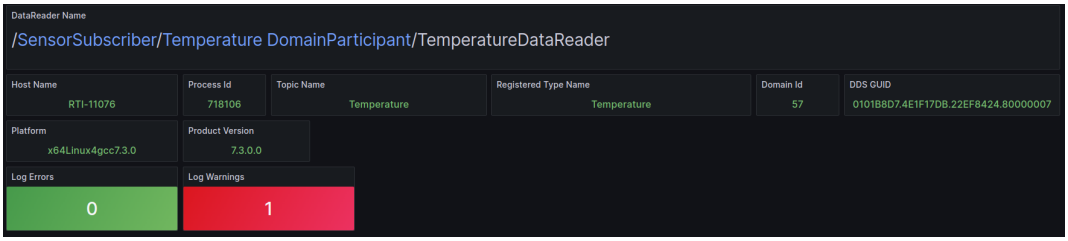
This warning message provides information about the GUID of the DataWriter that published the sensor data that was dropped due to time synchronization issues. But how do we locate the *DataWriter* from its GUID?

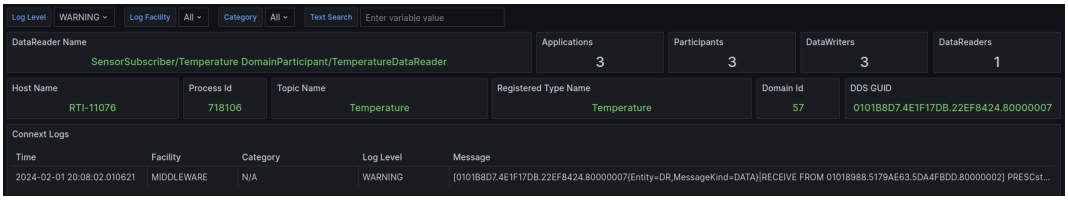
Note the highlighted RECEIVE FROM GUID in the log message. This represents the corresponding *DataWriter* that created the warning. (You can copy this GUID at this point).

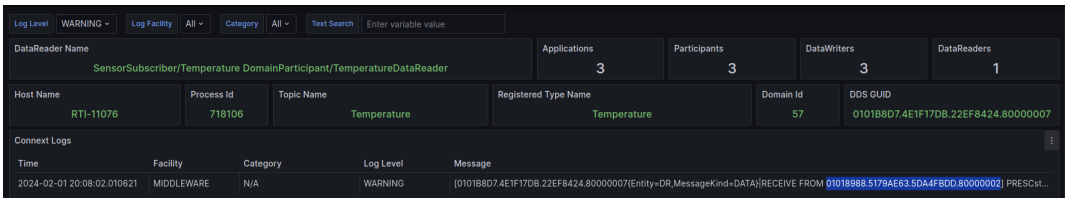
Select the **DataWriters** panel to view a list of the running *DataWriters*.


Total System Logs
15
Hosts
1
Applications
3
Participants
3
DataWriters
3
DataReaders
1
Topics
1
Observability Dashboards
7.3.0 (LTS) (20240131)
Writer Data Loss
Replaced Unacked Samples
OK
Reader Data Loss
Lost Samples
OK
Dropped Samples due to History D...
Error
Dropped Samples due to Instance ...
OK
Dropped Samples due to Time Sync
Error

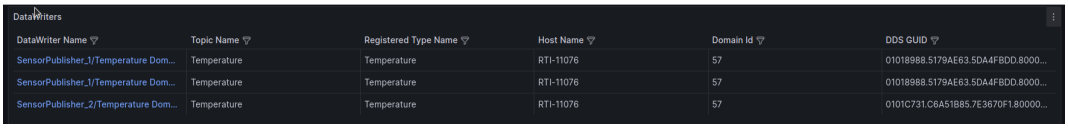

Dropped Samples due to Time Sync
DataReader Name
Topic Name
Registered Type Name
Host Name
Domain Id
DDS GUID
Status
SensorSubscriber/Temperature...
Temperature
Temperature
RTI-11076
57
0101B8D7.4E1F17DB.22EF8424....
ERROR


DataReader Name
/SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader
Host Name
RTI-11076
Process Id
718106
Topic Name
Temperature
Registered Type Name
Temperature
Domain Id
57
DDS GUID
0101B8D7.4E1F17DB.22EF8424.80000007
Platform
x64Linux4gcc7.3.0
Product Version
7.3.0.0
Log Errors
0
Log Warnings
1

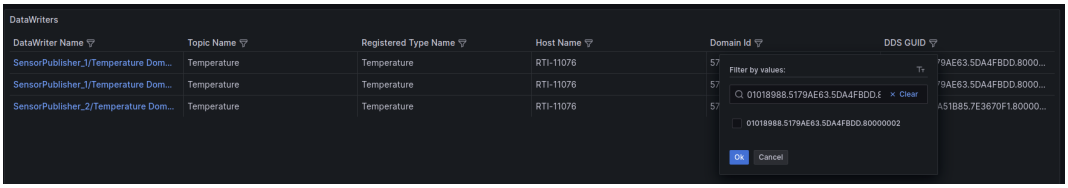

Log Level
WARNING
Log Facility
All
Category
All
Text Search
Enter variable value
DataReader Name
SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader
Applications
3
Participants
3
DataWriters
3
DataReaders
1
Host Name
RTI-11076
Process Id
718106
Topic Name
Temperature
Registered Type Name
Temperature
Domain Id
57
DDS GUID
0101B8D7.4E1F17DB.22EF8424.80000007
Connext Logs
Time
Facility
Category
Log Level
Message
2024-02-01 20:08:02.010621
MIDDLEWARE
N/A
WARNING
[0101B8D7.4E1F17DB.22EF8424.80000007{Entity=DR,MessageKind=DATA}|RECEIVE FROM 01018988.5179AE63.5DA4FBDD.80000002] PRESCst...


Log Level
WARNING
Log Facility
All
Category
All
Text Search
Enter variable value
DataReader Name
SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader
Applications
3
Participants
3
DataWriters
3
DataReaders
1
Host Name
RTI-11076
Process Id
718106
Topic Name
Temperature
Registered Type Name
Temperature
Domain Id
57
DDS GUID
0101B8D7.4E1F17DB.22EF8424.80000007
Connext Logs
Time
Facility
Category
Log Level
Message
2024-02-01 20:08:02.010621
MIDDLEWARE
N/A
WARNING
[0101B8D7.4E1F17DB.22EF8424.80000007{Entity=DR,MessageKind=DATA}|RECEIVE FROM 01018988.5179AE63.5DA4FBDD.80000002] PRESCst...

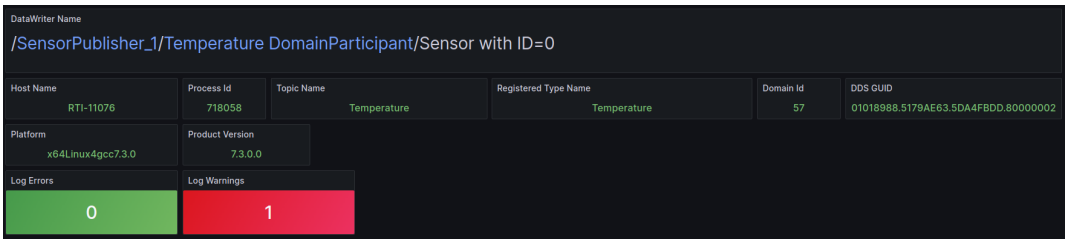
Now that we have a list of *DataWriters*, we can compare their GUIDs with the GUID in the log message to find the problem *DataWriter*. In this case the list does not have a lot of entries, so you can search manually.



However, when the number of entries is large, you can click on the funnel icon next to the **GUID** label to filter the list to the one writer with time synchronization issues by typing in the GUID or pasting the value copied from the log message.



Finally, select the problem DataWriter to learn its identity.



The problem *DataWriter* corresponds to sensor 0. You have successfully done root cause analysis by correlating metrics and logging.

7.3.6 Change the Application Logging Verbosity

Monitoring Library 2.0 has two verbosity settings.

- **Collection verbosity** controls the level of log messages an application generates.
- **Forwarding verbosity** controls the level of log messages an application forwards to the *Observability Collector Service* (making the messages visible in the dashboard).

For additional information on logging, refer to *Logs*.

By default, *Monitoring Library 2.0* only forwards error and warning log messages, even if the applications generate more verbose logging. Forwarding messages at a higher verbosity for all applications may saturate the network and the different *Observability Framework* components, such as *Observability Collector Service* and the logging aggregation backend (Grafana Loki in this example).

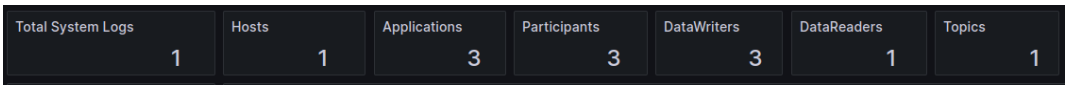
However, in some cases you may want to change the logging Collection verbosity and/or the Forwarding verbosity for specific applications to obtain additional information when doing root cause analysis.

In this section, you will increase both the Collection and Forwarding verbosity levels for the first publishing application using a remote command. To do that, you will use the application resource name generated by using the `-n` command-line option. The three applications have the following names:

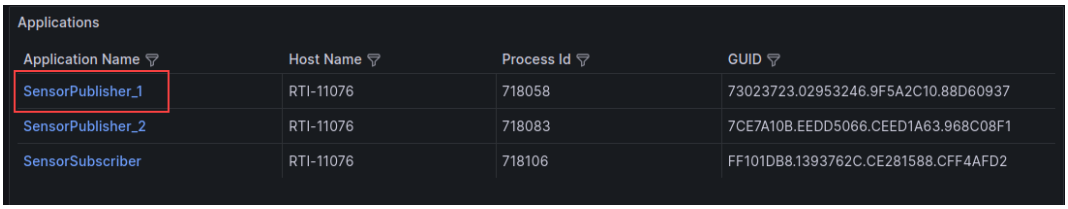
- /applications/SensorPublisher_1
- /applications/SensorPublisher_2
- /applications/SensorSubscriber

To change the Collection verbosity:

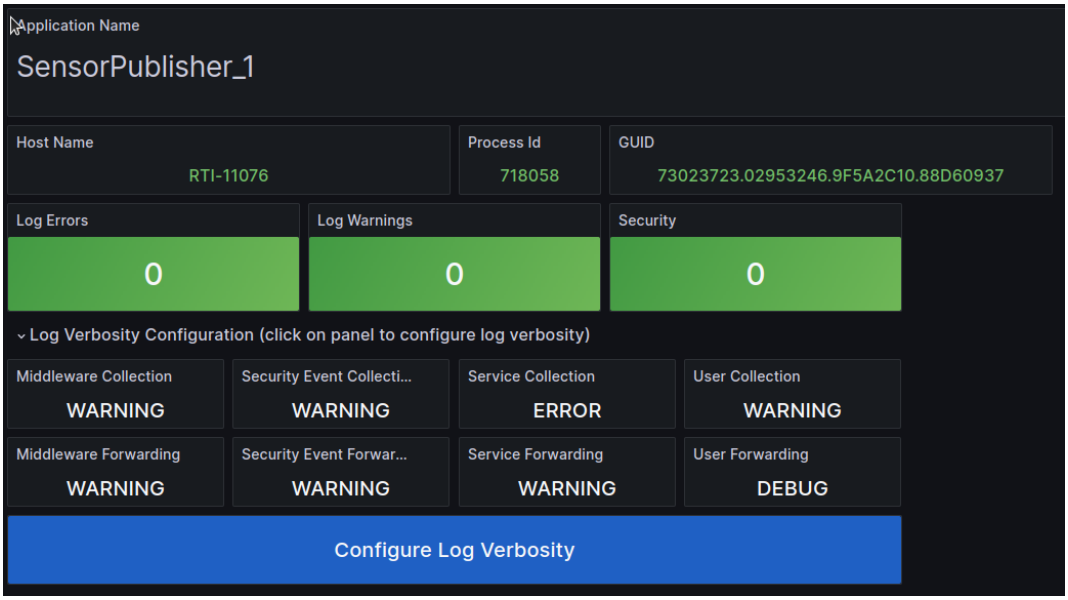
1. From the Alert Home dashboard, select the **Applications** panel to open the Application List dashboard.



2. From the Application List dashboard, select the **SensorPublisher_1** link to open the Alert Application Status dashboard.



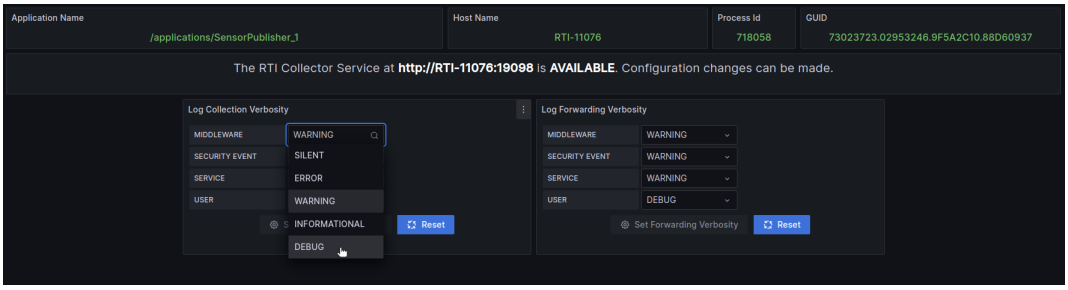
3. From the Alert Application Status dashboard, select the **Configure Log Verbosity** button to open the Log Control dashboard.

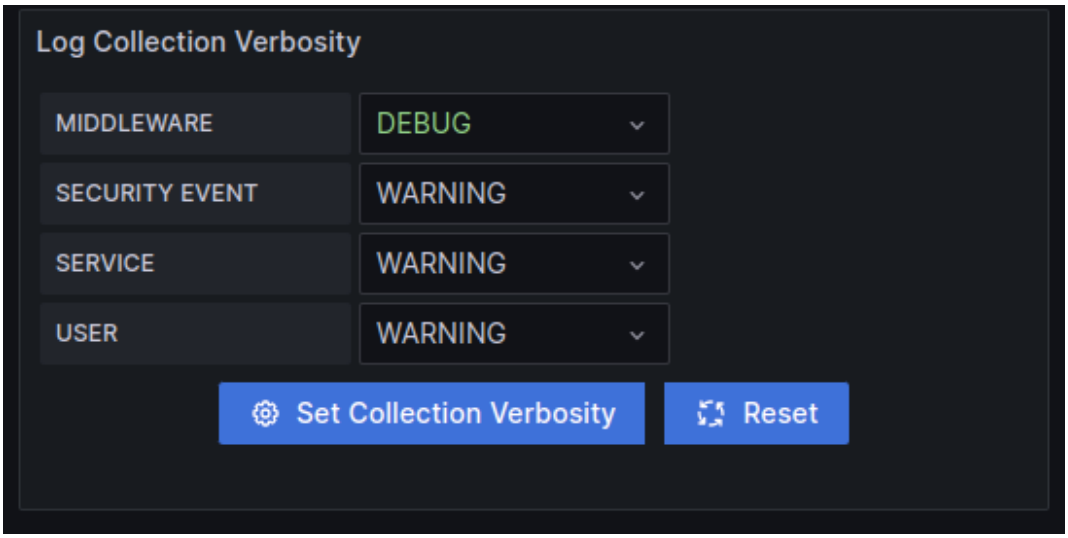


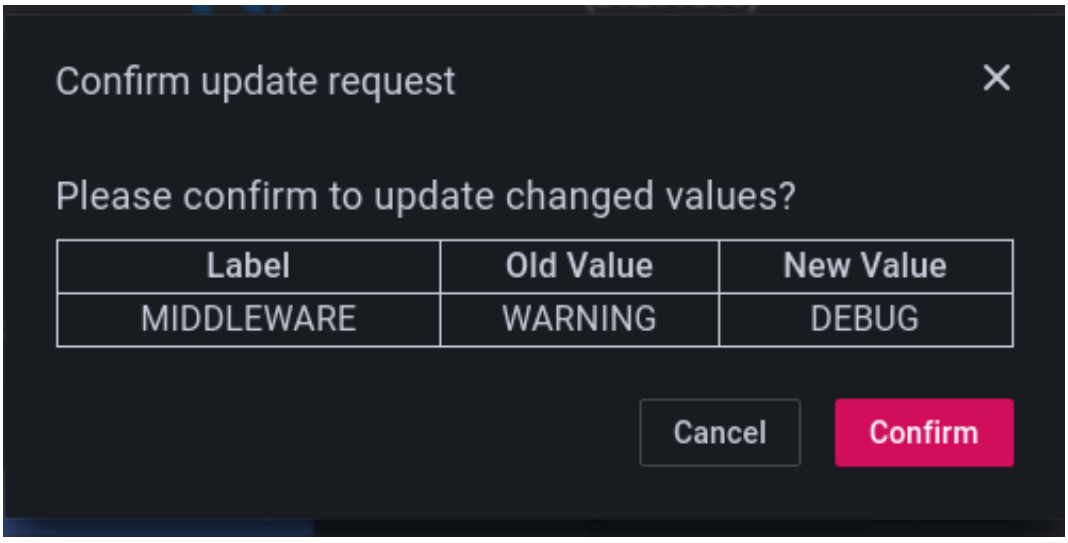
4. From the Log Control dashboard's **Log Collection Verbosity** panel, select **DEBUG** for the **MIDDLEWARE** facility.

   Note that the verbosity setting color changes to indicate the update. Also, the **Set Collection Verbosity** button becomes available.

5. Select the **Set Collection Verbosity** button. When prompted to confirm the update, select **Confirm** to set the Collection verbosity level to DEBUG at the application.


Application Name
/applications/SensorPublisher_1
Host Name
RTI-11076
Process Id
718058
GUID
73023723.02953246.9F5A2C10.88D60937
The RTI Collector Service at http://RTI-11076:19098 is AVAILABLE. Configuration changes can be made.
Log Collection Verbosity
MIDDLEWARE
WARNING
SECURITY EVENT
SILENT
SERVICE
ERROR
USER
WARNING
INFORMATIONAL
DEBUG
Reset
Log Forwarding Verbosity
MIDDLEWARE
WARNING
SECURITY EVENT
WARNING
SERVICE
WARNING
USER
DEBUG
Set Forwarding Verbosity
Reset


Log Collection Verbosity
MIDDLEWARE
DEBUG
SECURITY EVENT
WARNING
SERVICE
WARNING
USER
WARNING
Set Collection Verbosity
Reset


Confirm update request
Please confirm to update changed values?
Label
Old Value
New Value
MIDDLEWARE
WARNING
DEBUG
Cancel
Confirm

The selected application's Collection verbosity is now DEBUG. If you examine the terminal window for SensorPublisher_1, you will see messages like those in the following image.

```
DEBUG MIGInterpreter_parse:ACK from 0X1013131,0X4856CAC6
COMMENDSrWriterService_onSubmessage:[1689963305,136481175] writer oid␣
↪0x80002102 receives ACKNACK from reader 0x1013131.4856cac6.bd3f33f5.
↪80000007 for lead [(0000000000,00013981)] bitcount(0), epoch(41048),␣
↪isPureNack(0)
DEBUG COMMENDActiveFacadeReceiver_loop:rCoTemnt##02Rcv returning message loan
DEBUG NDDS_Transport_UDP_receive_rEA:rCoTemnt##02Rcv blocking on 0X549D
DEBUG NDDS_Transport_UDP_receive_rEA:rCoTemnt##02Rcv received 64 bytes from␣
↪0X100007F|40284
DEBUG RTINetioReceiver_receiveFast:rCoTemnt##02Rcv received 64 bytes
DEBUG COMMENDActiveFacadeReceiver_loop:rCoTemnt##02Rcv parsing message
DEBUG MIGInterpreter_parse:INFO_DST from 0X1013131,0X4856CAC6
DEBUG MIGInterpreter_parse:ACK from 0X1013131,0X4856CAC6
DEBUG COMMENDActiveFacadeReceiver_loop:rCoTemnt##02Rcv returning message loan
DEBUG NDDS_Transport_UDP_receive_rEA:rCoTemnt##02Rcv blocking on 0X549D
DEBUG NDDS_Transport_UDP_receive_rEA:rCoTemnt##02Rcv received 64 bytes from␣
↪0X100007F|40284
DEBUG RTINetioReceiver_receiveFast:rCoTemnt##02Rcv received 64 bytes
DEBUG COMMENDActiveFacadeReceiver_loop:rCoTemnt##02Rcv parsing message
DEBUG MIGInterpreter_parse:INFO_DST from 0X1013131,0X4856CAC6
DEBUG MIGInterpreter_parse:ACK from 0X1013131,0X4856CAC6
DEBUG RTIEventActiveGeneratorThread_loop:rCoTemnt####Evt gathering events
DEBUG RTIEventActiveGeneratorThread_loop:rCoTemnt####Evt firing events
DEBUG COMMENDActiveFacadeReceiver_loop:rCoTemnt##02Rcv returning message loan
DEBUG NDDS_Transport_UDP_receive_rEA:rCoTemnt##02Rcv blocking on 0X549D
DEBUG RTIEventActiveGeneratorThread_loop:rCoTemnt####Evt rescheduling events
DEBUG RTIEventActiveGeneratorThread_loop:rCoTemnt####Evt sleeping {00000000,
↪1B5E7420}
DEBUG NDDS_Transport_UDP_receive_rEA:rCoObsnt##00Rcv received 292 bytes from␣
↪0X100007F|46993
DEBUG RTINetioReceiver_receiveFast:rCoObsnt##00Rcv received 292 bytes
DEBUG COMMENDActiveFacadeReceiver_loop:rCoObsnt##00Rcv parsing message
DEBUG MIGInterpreter_parse:SECURE_RTPS_PREFIX from 0XDFCD91E1,0X6868BAE7
DEBUG MIGInterpreter_parse:INFO_TS from 0XDFCD91E1,0X6868BAE7
DEBUG MIGInterpreter_parse:DATA from 0XDFCD91E1,0X6868BAE7
```

At this point, the SensorPublisher_1 application is generating log messages at the DEBUG level as shown in the terminal window, but the debug messages are not being forwarded to *Observability Collector Service* because the Forwarding verbosity is still at WARNING.

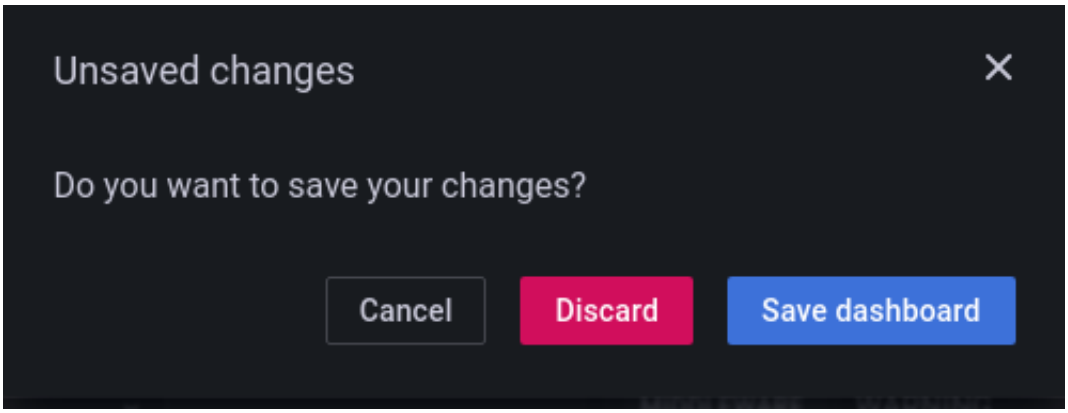
To set the Forwarding verbosity to DEBUG, repeat steps 4 and 5 above in the **Log Fowarding Verbosity** panel.

After setting both the Collection and Forwarding verbosity to DEBUG, you should see an indication that DEBUG messages are being received for the SensorPublisher_1 application by examining the **Total System Logs** panel on the **Alert Home** dashboard.

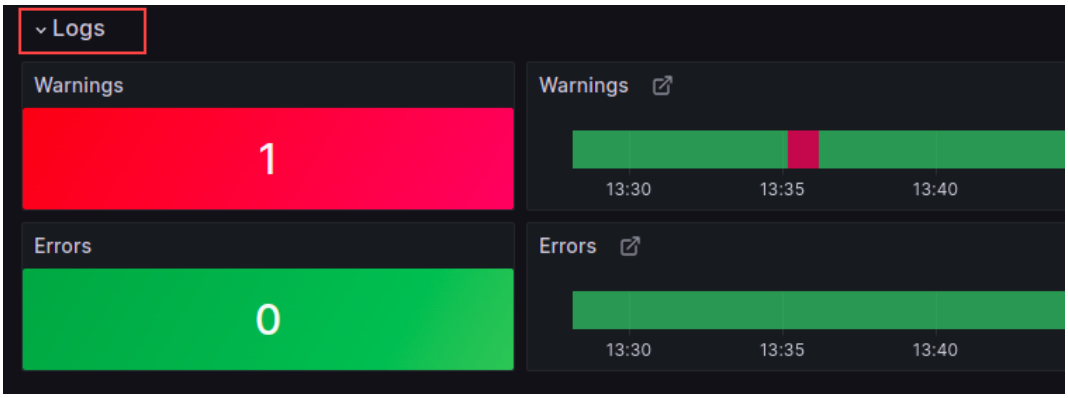
To get back to the **Alert Home** dashboard, click **Home** at the top left.

Note: If you are using the dashboards as an admin user, you will be prompted to save your changes. Select

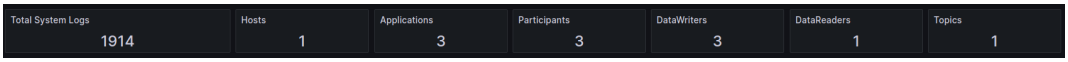
the **Discard** button; the changes to the dashboard do not need to be saved, since they are set in the application. The save prompt does not appear when logged into the dashboard as a user without admin permissions.



If the **Warnings**, **Errors**, and **Security** panels are not displayed in the **Alert Home** dashboard, select the **Logs** dashboard row. Note that there are no additional log messages indicated in the **Warnings**, **Errors**, or **Security** panels since those panels only show the number of Warning-level, Error-level, and Security-related log messages, respectively.



To verify that DEBUG messages are being collected, select the value in the **Total System Logs** panel to open the **Logs** dashboard. You will see that the total number of log messages received is increasing dramatically.



You can manipulate the Log Control settings to verify application and dashboard behavior as shown in Table 7.6.

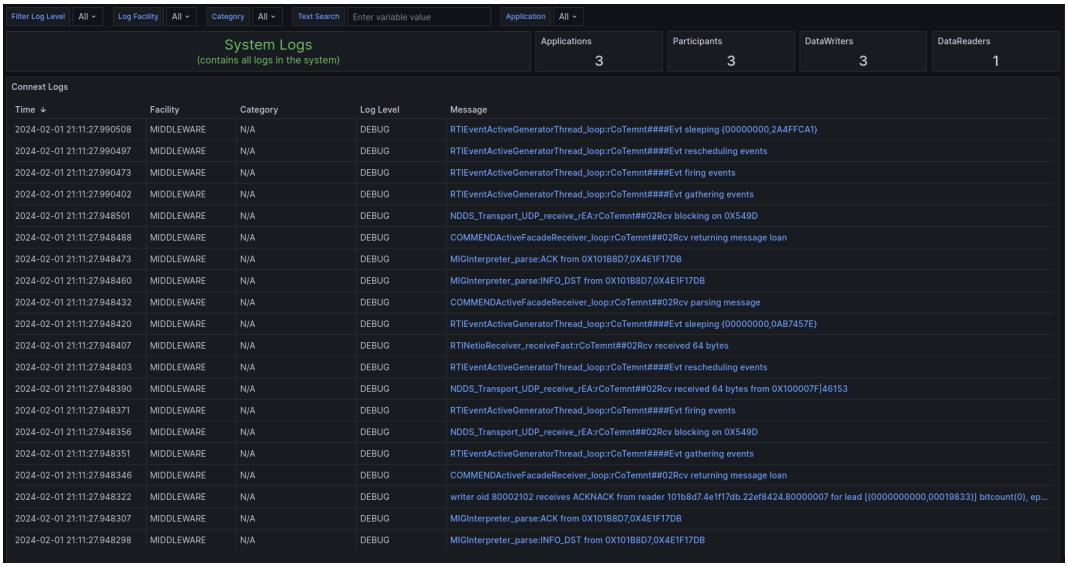


Table 7.6: Collection and Forwarding Log Verbosity DEBUG Behavior

Collection Verbosity	Forwarding Verbosity	Application DEBUG Log Output	Grafana Connext DEBUG Logs
WARNING	WARNING	NO	NO
DEBUG	WARNING	YES	NO
WARNING	DEBUG	NO	NO
DEBUG	DEBUG	YES	YES

7.3.7 Change the Metric Configuration

Metrics are the collections of counters and gauges you can use to analyze application behavior. *Observability Framework* gives you complete control of which metrics to collect, both before and during runtime. Data for your selected metrics is forwarded to *Observability Collector Service* and made available to third-party backends.

In this example, all application metrics have already been enabled, and Prometheus is used on the backend to store collected metrics.

Note: By default, *Observability Framework* does not collect metrics for any DDS entities. For details on how to enable the initial metrics to be collected, see *Setting the Initial Metrics and Log Configuration*.

The *Observability Dashboards* enable you to change the initial metric configuration for specific applications, DDS entities, or DDS entity instances during runtime, without restarting or reconfiguring your applications. You can dynamically change the metric configuration for a specific DDS resource (a single application, *Participant*, *DataReader*, *DataWriter*, or *Topic*), or all of the resources of a given type contained by another resource (for example, all *DataWriters* of an application, or all *DataReaders* of a *Participant*)

This section of the *Observability Framework* example will walk you through two scenarios:

- *Changing metrics collected for a single DataWriter resource*
- *Changing metrics collected for all DataWriters of an application*

Resources used in this example

In the *Start the Applications* section, you created and named three applications using the `-n` command-line option. Table 7.7 lists the DDS entity names for these applications and the other resources used in this example.

The DDS entity names are specified using the DDS_EntityNameQosPolicy `name` field for each entity via XML or programmatically. The entity names are used to build the **resource name**, which is the unique identifier used in the remote commands to specify each resource.

For more information, see *Resource Pattern Definitions*.

Table 7.7: Resources in this Example

Entity Name	Entity Type	Where Configured
SensorPublisher_1	Application	On start
SensorPublisher_2	Application	On start
SensorSubscriber	Application	On start
Temperature DomainParticipant	*DomainParticipant*	XML
TemperatureDataReader	*DataReader*	XML
Sensor with ID=0	*DataWriter*	Code
Sensor with ID=1	*DataWriter*	Code
Sensor with ID=2	*DataWriter*	Code

For more information about observable resource names, see *Resources*. The *Observability Framework Dashboards* build the resource names and commands based on your configuration.

The following code snippet details how to set the *DataWriter* entity name programmatically.

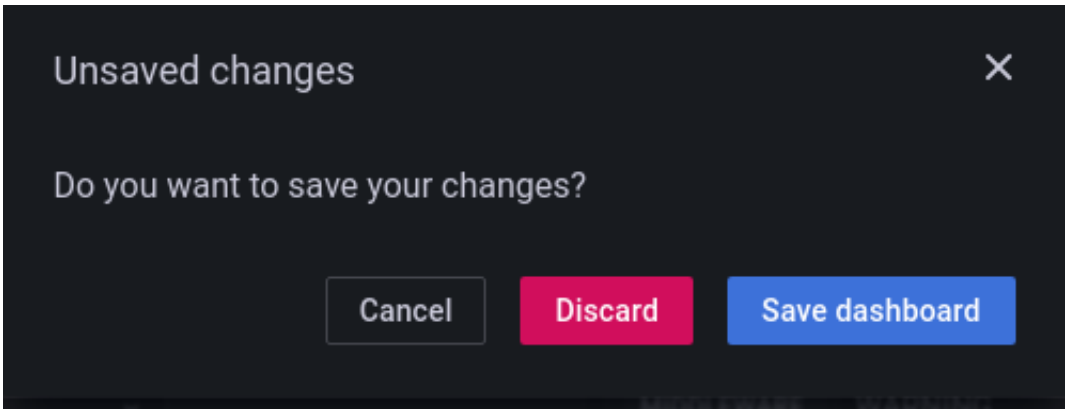
```
// create and initialize DataWriterQoS
DDS_DataWriterQos writerQos;
retcode = publisher->get_default_datawriter_qos(writerQos);
if (retcode != DDS_RETCODE_OK) {
    return shutdown_participant(
            participant,
            "get_default_datawriter_qos error",
            EXIT_FAILURE);
}

// create and initialize sensorName
char sensorName[64];
sprintf(sensorName, "Sensor with ID=%d", sensor_id);

// set the publication_name.name in DataWriterQoS
writerQos.publication_name.name = sensorName;

// create DataWriter entity with updated DataWriterQoS
untyped_writer = publisher->create_datawriter(
```

(continues on next page)

(continued from previous page)

```
        topic,
        writerQos,
        NULL /* listener */,
        DDS_STATUS_MASK_NONE);
```

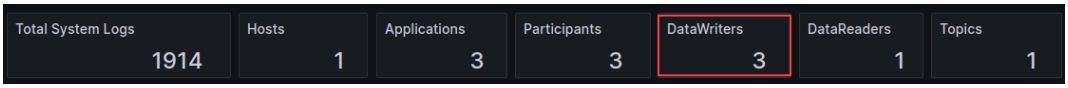
Note: In this example, you will access configuration dashboards several times. If you are using the dashboards as an admin user, you will be prompted to save your changes each time you navigate away from a configuration dashboard. When prompted to save, click the **Discard** button; changes to the dashboard do not need to be saved because they are set in the application. This prompt does not appear when logged into the dashboard as a user without admin permissions.



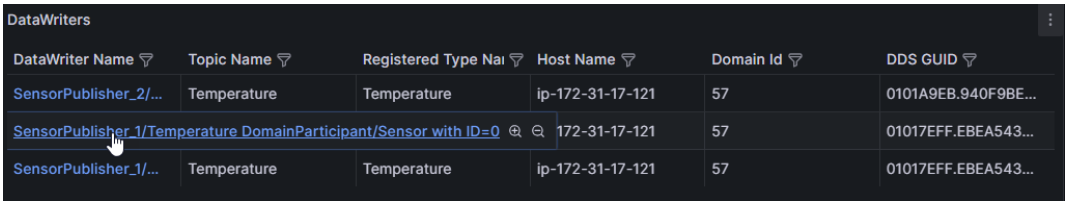
Changing metrics collected for a single DataWriter

Use the *Observability Framework Dashboards* to disable the **Pushed Sample Bytes** metric on one *DataWriter* in the **SensorPublisher_1** application.

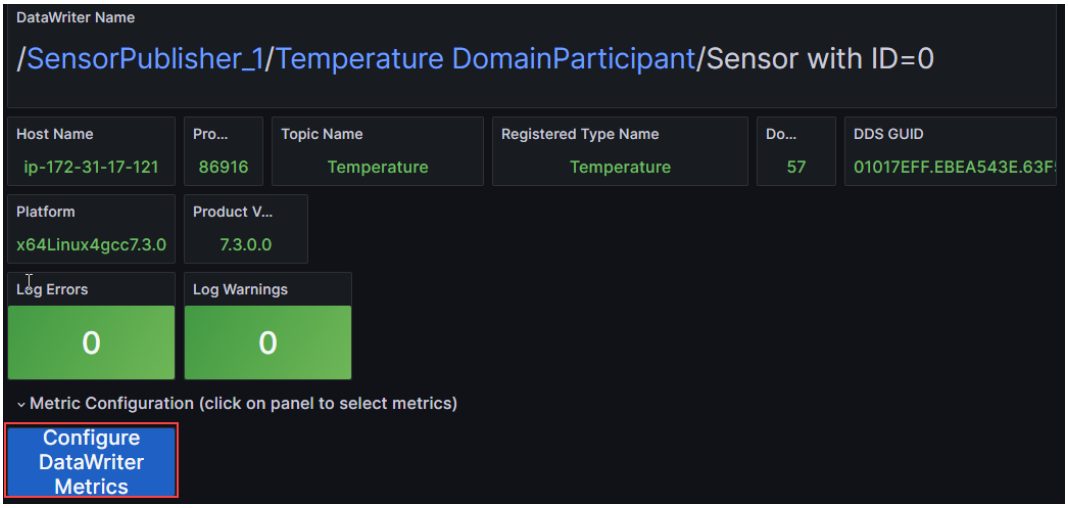
1. From the Alert Home dashboard, select the **DataWriters** panel to open the DataWriters List dashboard.



2. Select the **SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0** *DataWriter* to open the Alert DataWriter Status dashboard. You may need to hover over the **DataWriter Name** field to see the full *DataWriter* resource name.



3. Select **Configure DataWriter Metrics** to open the DataWriter Metrics dashboard.

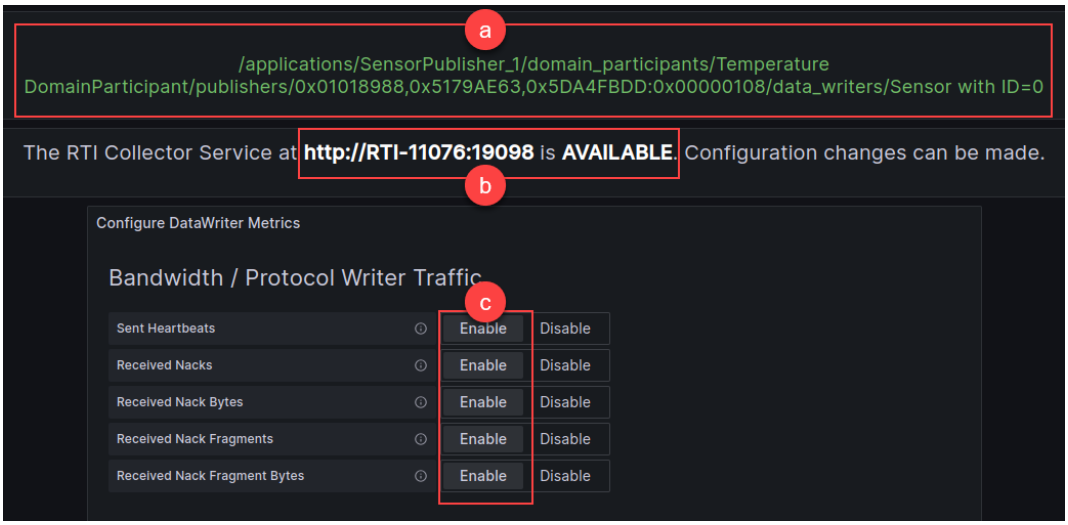


4. In the DataWriter Metrics dashboard, note the following:

   a. The fully qualified resource name is displayed at the top.

   b. The *Collector Service* Control URL and status display below the resource name. If the *Collector Service* cannot be reached, the status will be **NOT AVAILABLE** and changes will not be allowed.

   c. **Enable** is selected for all of the metrics, indicating that they are all currently active.



5. In the **Bandwidth/User Data Writer Traffic** section, select **Disable** for **Pushed Sample Bytes**. Note the text color changes to indicate the pending update.

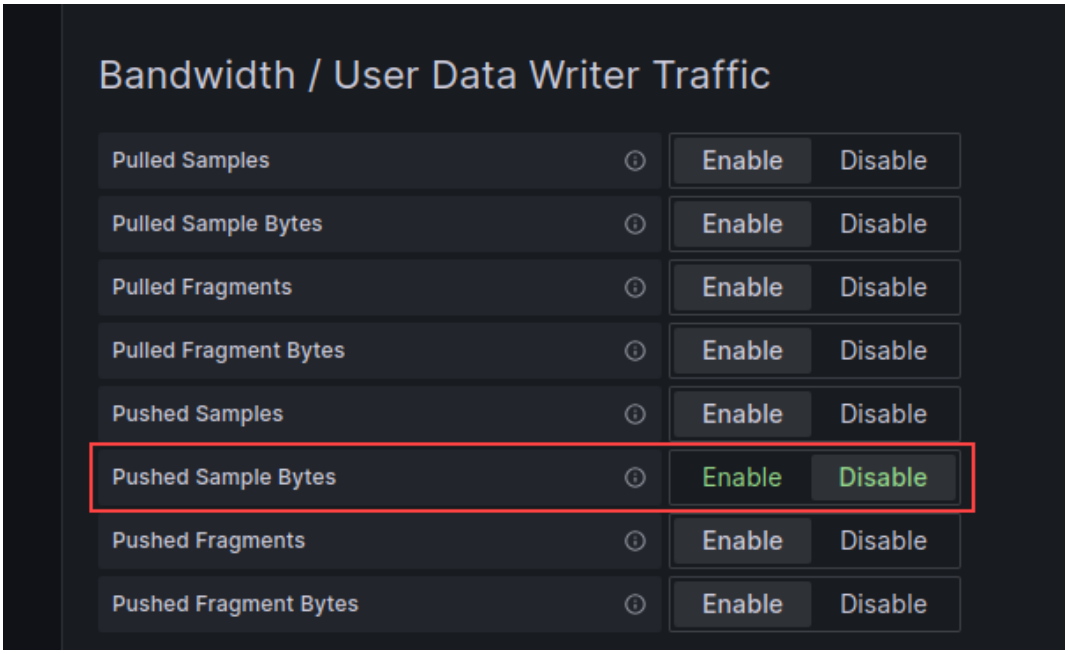
6. At the bottom of the page, select **Configure Metrics**.

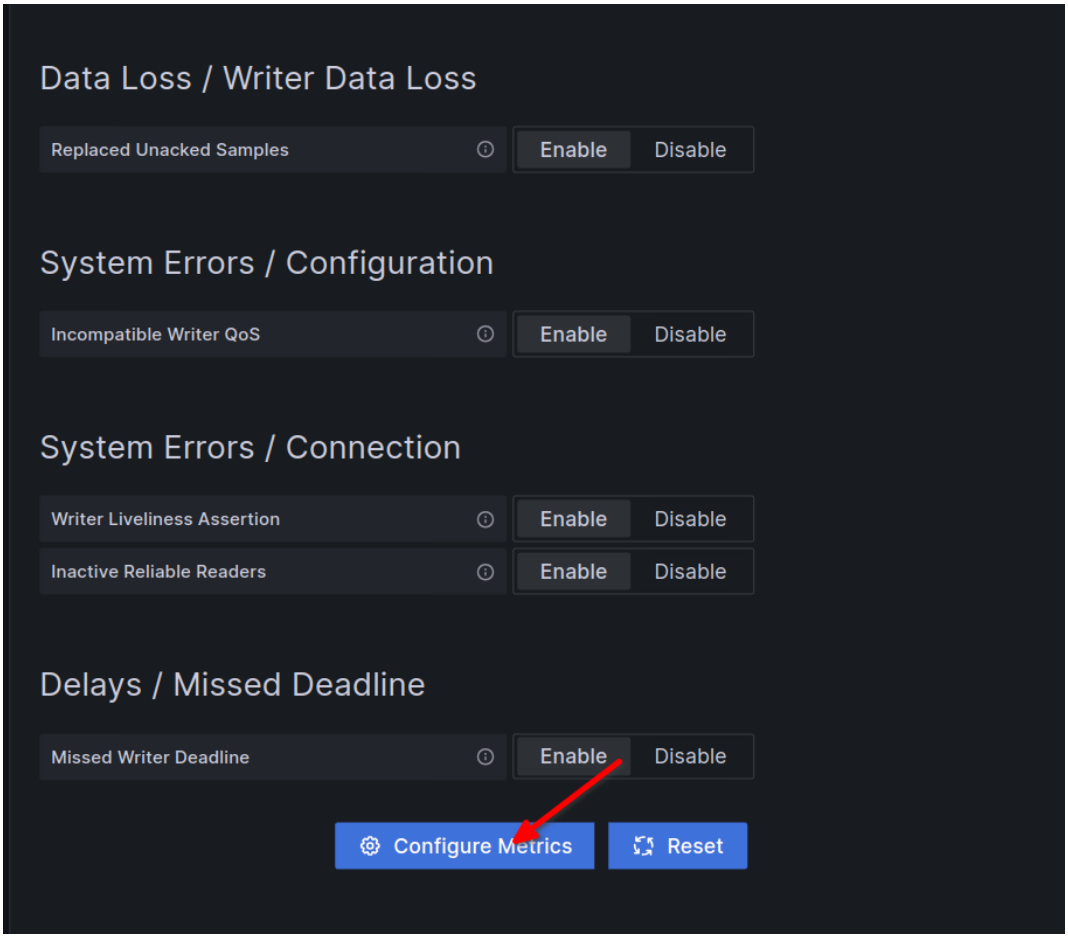
7. When prompted to confirm the change, review the updates and then select **Confirm**.

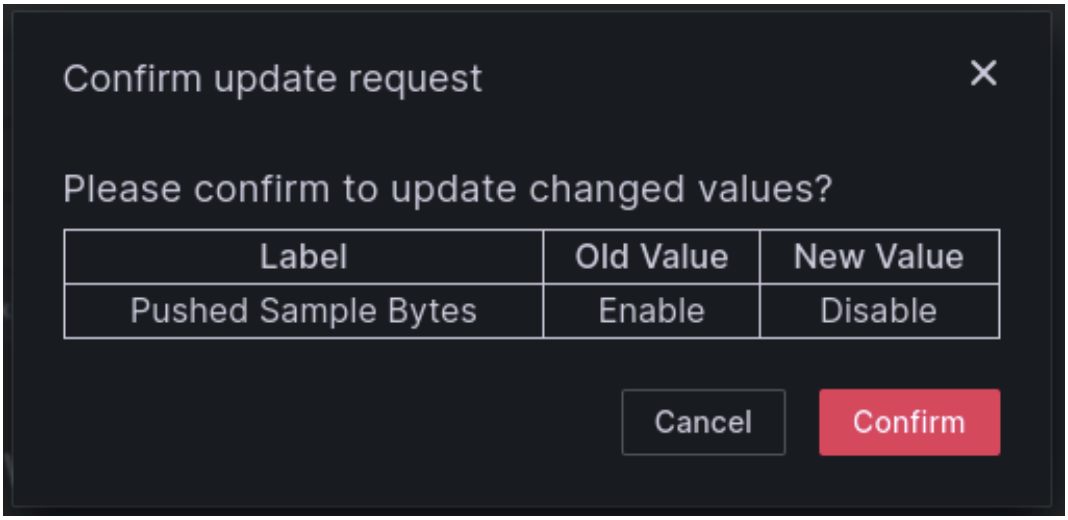
8. Verify that **Pushed Sample Bytes** is now disabled.

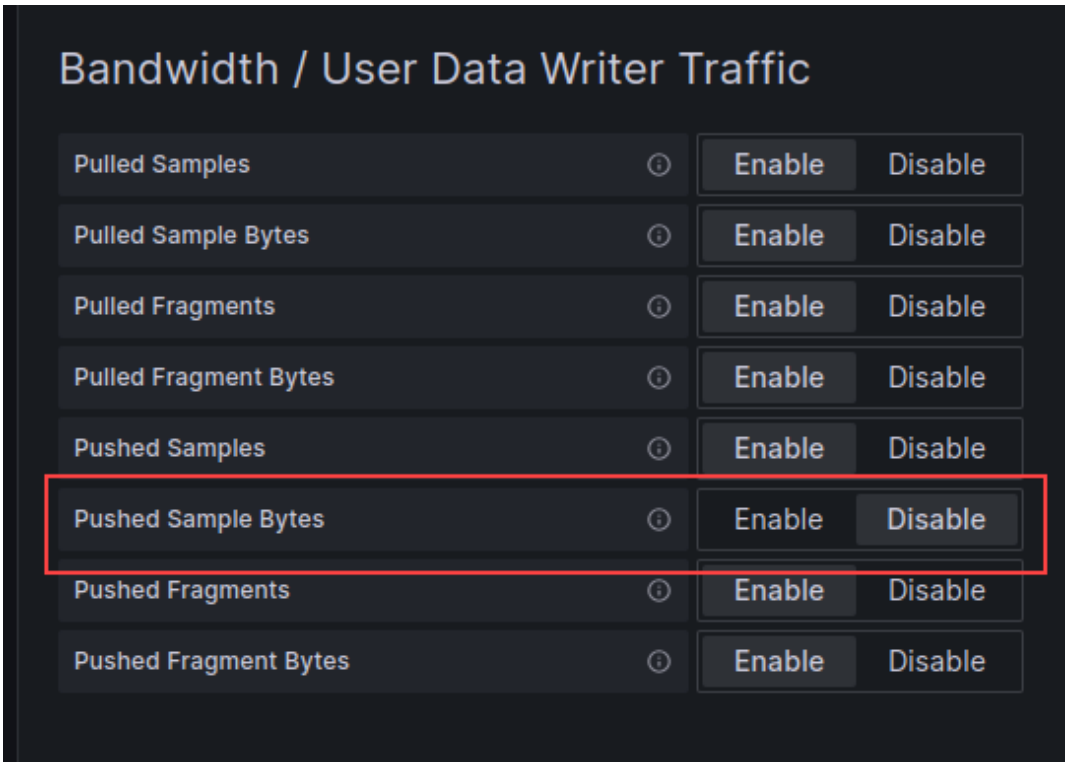
9. To confirm the **Pushed Sample Bytes** metric is no longer being collected:

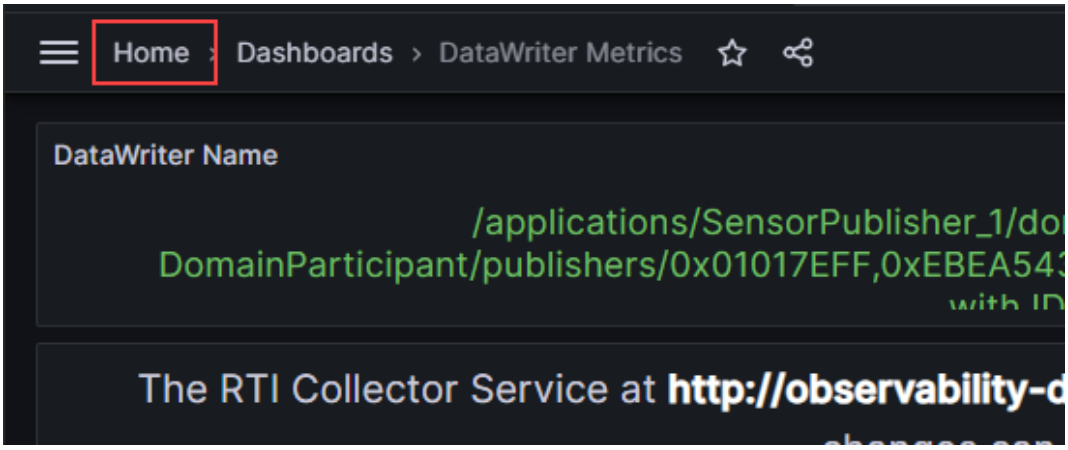
   a. At the top left, select **Home** to return to the Alert Home dashboard.


Bandwidth / User Data Writer Traffic
Pulled Samples
Enable
Disable
Pulled Sample Bytes
Enable
Disable
Pulled Fragments
Enable
Disable
Pulled Fragment Bytes
Enable
Disable
Pushed Samples
Enable
Disable
Pushed Sample Bytes
Enable
Disable
Pushed Fragments
Enable
Disable
Pushed Fragment Bytes
Enable
Disable

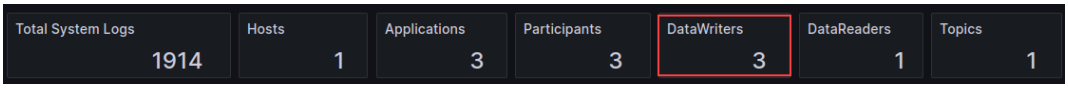

Data Loss / Writer Data Loss
Replaced Unacked Samples
Enable
Disable
System Errors / Configuration
Incompatible Writer QoS
Enable
Disable
System Errors / Connection
Writer Liveliness Assertion
Enable
Disable
Inactive Reliable Readers
Enable
Disable
Delays / Missed Deadline
Missed Writer Deadline
Enable
Disable
Configure Metrics
Reset


Confirm update request
Please confirm to update changed values?
Label
Old Value
New Value
Pushed Sample Bytes
Enable
Disable
Cancel
Confirm

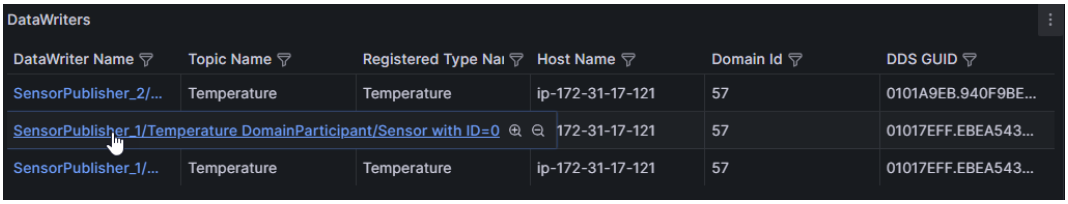

Bandwidth / User Data Writer Traffic
Pulled Samples
Pulled Sample Bytes
Pulled Fragments
Pulled Fragment Bytes
Pushed Samples
Pushed Sample Bytes
Pushed Fragments
Pushed Fragment Bytes
Enable
Disable



b. Select the **DataWriters** panel to open the DataWriters List dashboard.



c. Select the **SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0** link to open the Alert DataWriter Status dashboard.



d. Scroll down to the **Pushed Sample Bytes** graph to confirm the metric is not being collected.

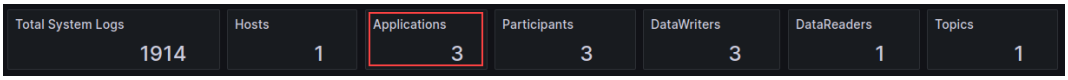
10. At the top left, select **Home** to go back to the Alert Home dashboard.

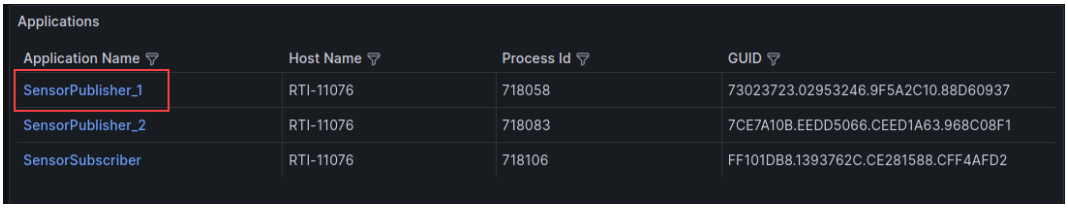
Changing metrics collected for all DataWriters of an application

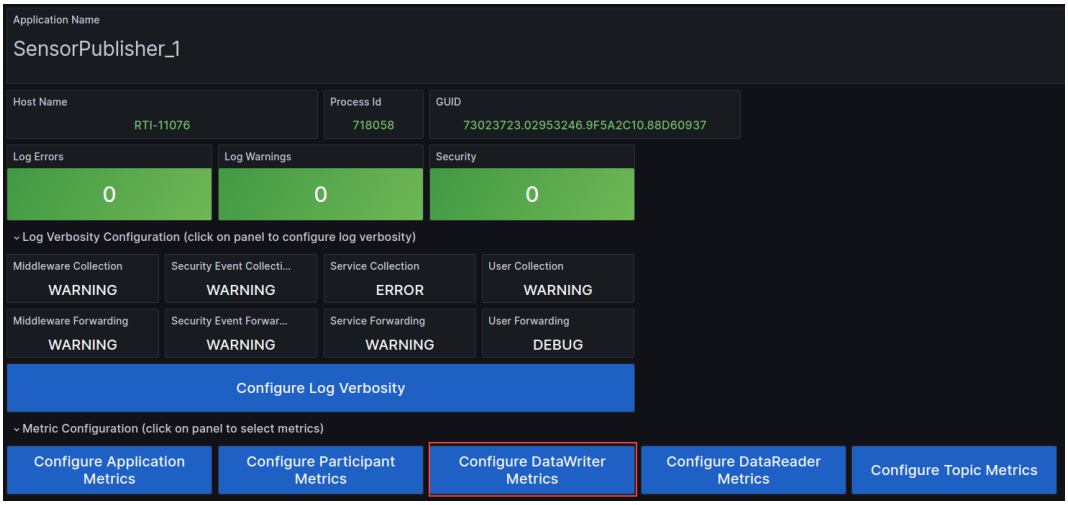
Disable the **Pushed Samples** metric on all *DataWriters* in the **SensorPublisher_1** application.

1. From the Alert Home dashboard, select the **Applications** panel to open the Application List dashboard.
2. Select **SensorPublisher_1** to open the Alert Application Status dashboard.
3. Select **Configure DataWriter Metrics** to open the DataWriter Metrics Multi dashboard.
4. In the DataWriter Metrics Multi dashboard, note the following:
   a. The Collector Service Control URL and status is displayed. If the *Collector Service* cannot be reached, the status will be **NOT AVAILABLE** and changes will not be allowed.
   b. The **Current State** panel indicates the current configuration of a metric across all *DataWriters* in the SensorPublisher_1 application.
      - **Enabled**. The metric is **enabled for all** *DataWriters* in the SensorPublisher_1 application


Bandwidth / User Data Writer Traffic
Pulled Samples
Pulled Sample Bytes
Pulled Fragments
Pulled Fragment Bytes
Pushed Samples
Pushed Sample Bytes
10 c/s
7.5 c/s
5 c/s
2.5 c/s
0 c/s
10 B/s
7.5 B/s
5 B/s
2.5 B/s
0 B/s
200 B/s
100 B/s
11:30
11:40
11:50
12:00
12:10
12:20


Total System Logs
1914
Hosts
1
Applications
3
Participants
3
DataWriters
3
DataReaders
1
Topics
1

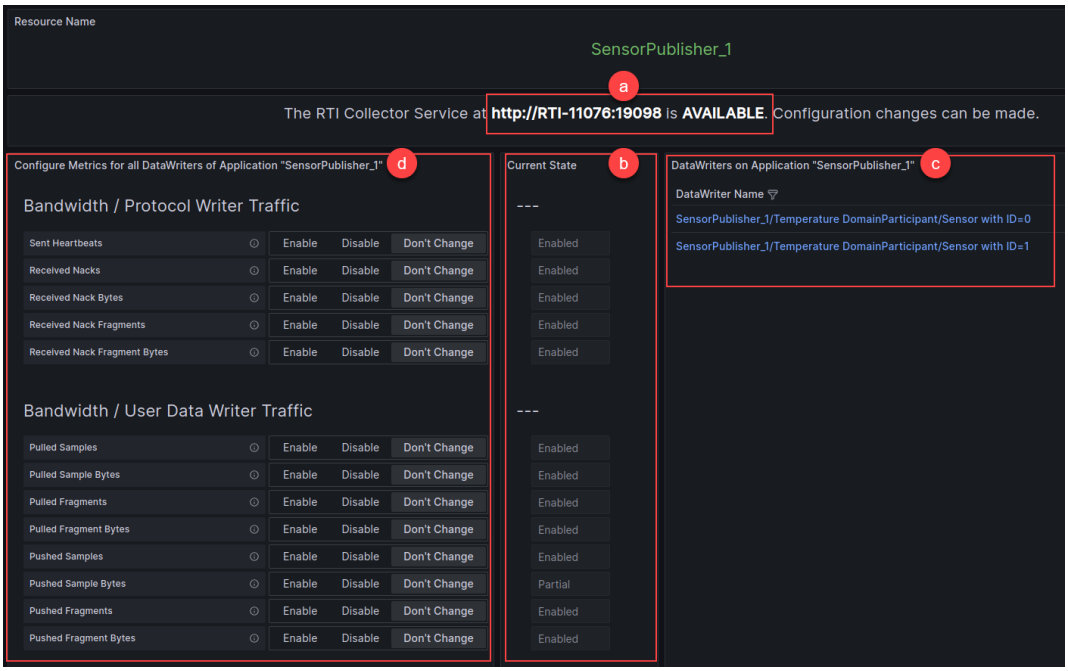

Applications
Application Name
Host Name
Process Id
GUID
SensorPublisher_1
RTI-11076
718058
73023723.02953246.9F5A2C10.88D60937
SensorPublisher_2
RTI-11076
718083
7CE7A10B.EEDD5066.CEED1A63.968C08F1
SensorSubscriber
RTI-11076
718106
FF101DB8.1393762C.CE281588.CFF4AFD2


Application Name
SensorPublisher_1
Host Name
RTI-11076
Process Id
718058
GUID
73023723.02953246.9F5A2C10.88D60937
Log Errors
0
Log Warnings
0
Security
0
Log Verbosity Configuration (click on panel to configure log verbosity)
Middleware Collection
WARNING
Security Event Collecti...
WARNING
Service Collection
ERROR
User Collection
WARNING
Middleware Forwarding
WARNING
Security Event Forwar...
WARNING
Service Forwarding
WARNING
User Forwarding
DEBUG
Configure Log Verbosity
Metric Configuration (click on panel to select metrics)
Configure Application Metrics
Configure Participant Metrics
Configure DataWriter Metrics
Configure DataReader Metrics
Configure Topic Metrics

- **Partial**. The metric is **enabled for at least one, but not all** of the *DataWriters* in the SensorPublisher_1 application
- **Disabled**. The metric is **disabled for all** DataWriters in the SensorPublisher_1 application.

c. The **DataWriters on Application "SensorPublisher_1"** panel lists the *DataWriters* in the **SensorPublisher_1** application. Use the links to access the metric control page for the selected entity.

d. The **Configure Metrics for all DataWriters of Application "SensorPublisher_1"** enables you to change the metrics configuration for all *DataWriters* of the **SensorPublisher_1** application. Changes made in this panel are used to build a command to modify the current configuration. By default, all metrics are initialized to **Don't Change**, indicating the configuration for that metric will not be changed and will remain in the state noted in the **Current State** panel. Selecting **Enable** will enable the metric for **all** *DataWriters* of the **SensorPublisher_1** application regardless of the current state. Selecting **Disable** will disable the metric for **all** *DataWriters* of the **SensorPublisher_1** application regardless of the current state.



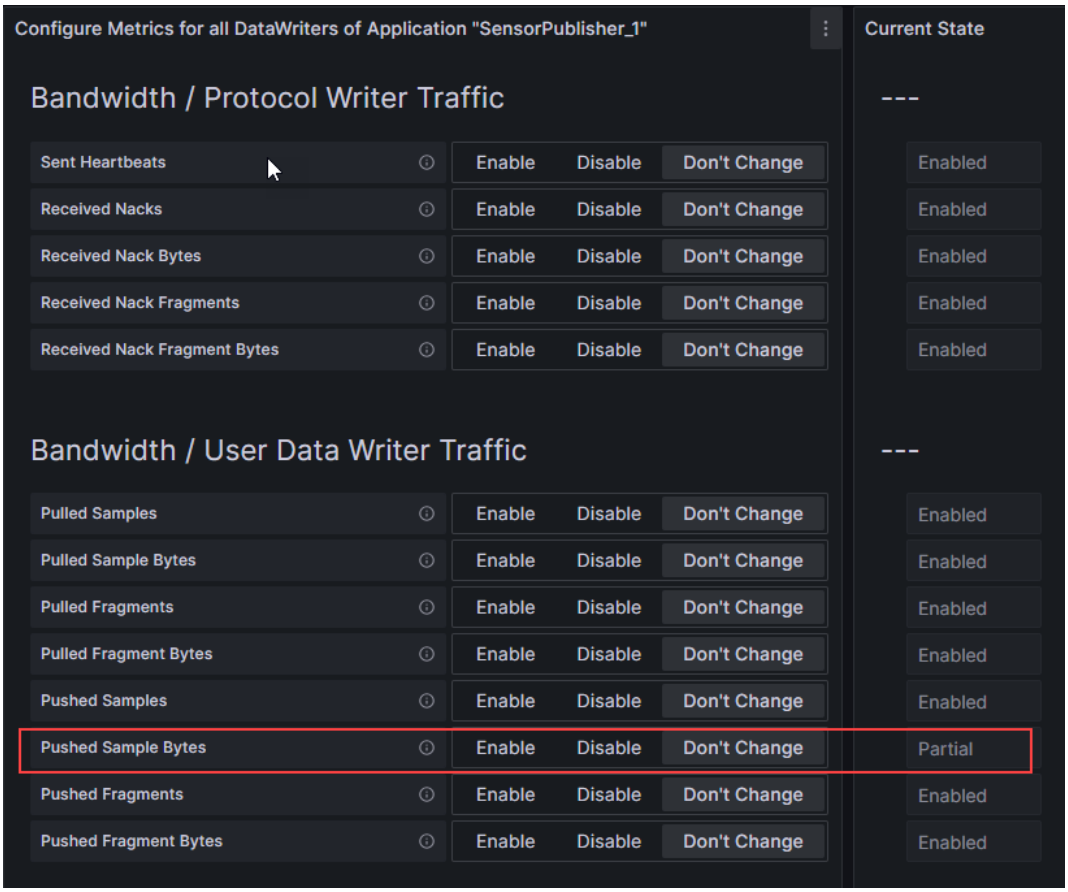
5. Verify that the **Current State** of the **Pushed Sample Bytes** metric is **Partial**. This status indicates the metric is still enabled on the *DataWriter* **SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=1** (the default setting), but disabled on **SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0** (as you configured *earlier above*).

6. Select **Disable** for **Pushed Samples**. If you do not see the **Disable** command, widen your browser window.

7. At the bottom of the page, select **Configure Metrics**.

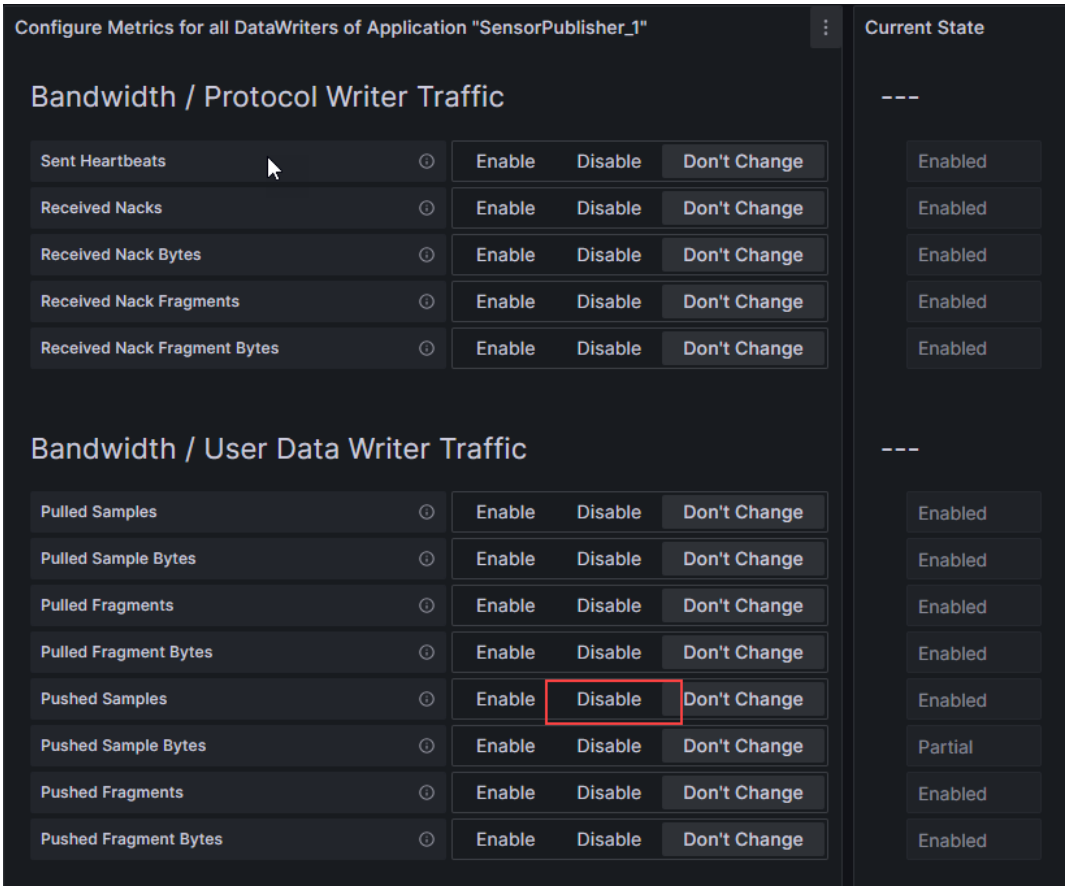
8. When prompted to confirm the change, verify the updates and then select **Confirm**.

9. Scroll up to verify the **Pushed Samples** metric is disabled. It may take a few seconds for the dashboard to refresh.


Configure Metrics for all DataWriters of Application "SensorPublisher_1"
Current State
Bandwidth / Protocol Writer Traffic

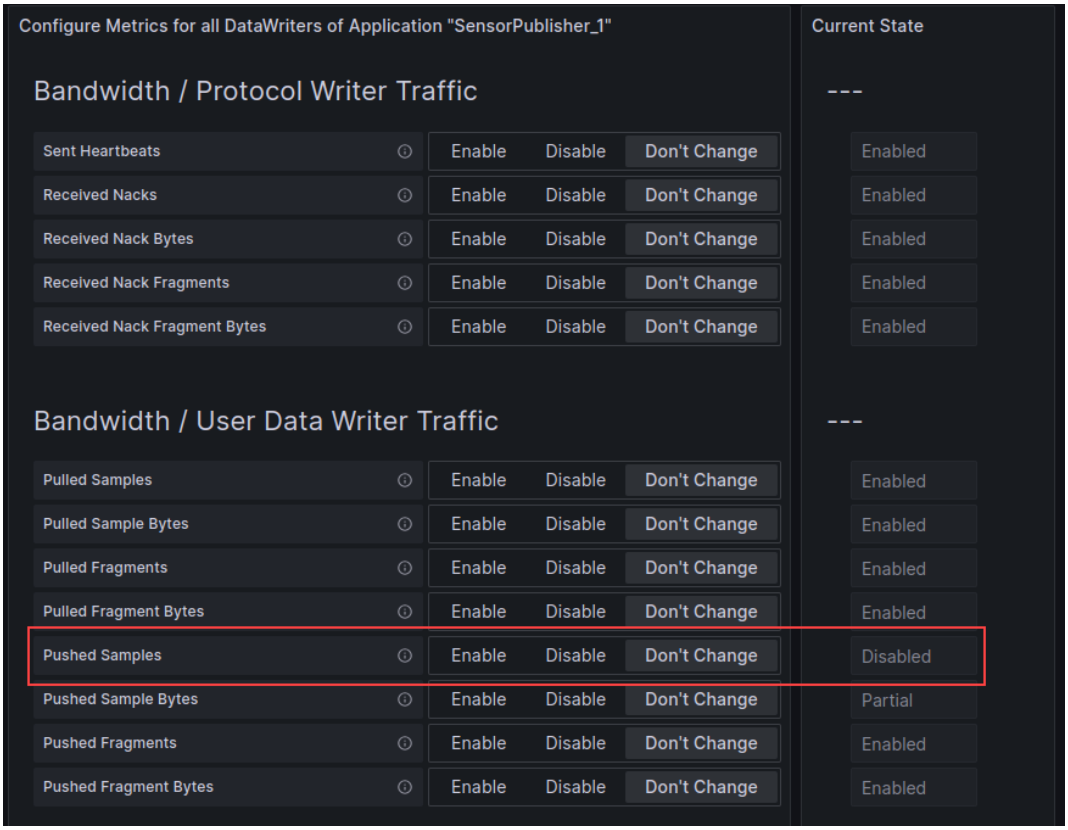
Sent Heartbeats
Enable
Disable
Don't Change
Enabled
Received Nacks
Enable
Disable
Don't Change
Enabled
Received Nack Bytes
Enable
Disable
Don't Change
Enabled
Received Nack Fragments
Enable
Disable
Don't Change
Enabled
Received Nack Fragment Bytes
Enable
Disable
Don't Change
Enabled
Bandwidth / User Data Writer Traffic

Pulled Samples
Enable
Disable
Don't Change
Enabled
Pulled Sample Bytes
Enable
Disable
Don't Change
Enabled
Pulled Fragments
Enable
Disable
Don't Change
Enabled
Pulled Fragment Bytes
Enable
Disable
Don't Change
Enabled
Pushed Samples
Enable
Disable
Don't Change
Enabled
Pushed Sample Bytes
Enable
Disable
Don't Change
Partial
Pushed Fragments
Enable
Disable
Don't Change
Enabled
Pushed Fragment Bytes
Enable
Disable
Don't Change
Enabled


Configure Metrics for all DataWriters of Application "SensorPublisher_1"
Current State
Bandwidth / Protocol Writer Traffic

Sent Heartbeats
Received Nacks
Received Nack Bytes
Received Nack Fragments
Received Nack Fragment Bytes
Enable
Disable
Don't Change
Enabled
Bandwidth / User Data Writer Traffic

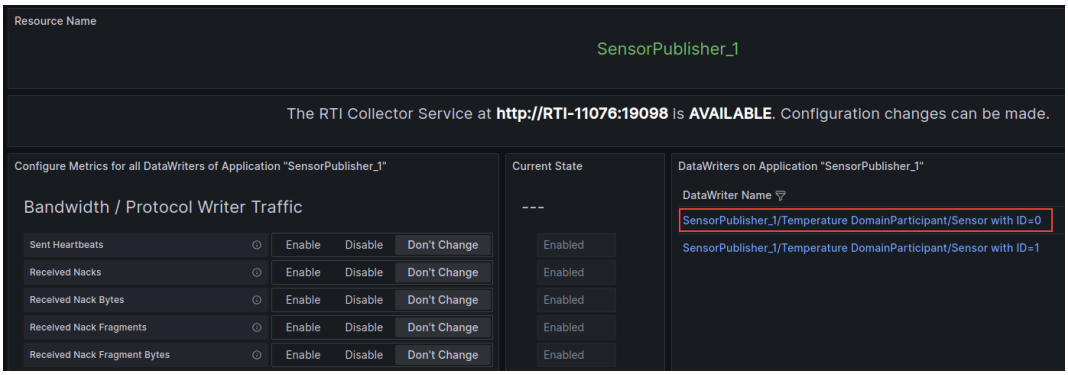
Pulled Samples
Pulled Sample Bytes
Pulled Fragments
Pulled Fragment Bytes
Pushed Samples
Pushed Sample Bytes
Pushed Fragments
Pushed Fragment Bytes
Partial


Configure Metrics for all DataWriters of Application "SensorPublisher_1"
Current State
Bandwidth / Protocol Writer Traffic

Sent Heartbeats
Received Nacks
Received Nack Bytes
Received Nack Fragments
Received Nack Fragment Bytes
Enable
Disable
Don't Change
Enabled
Bandwidth / User Data Writer Traffic

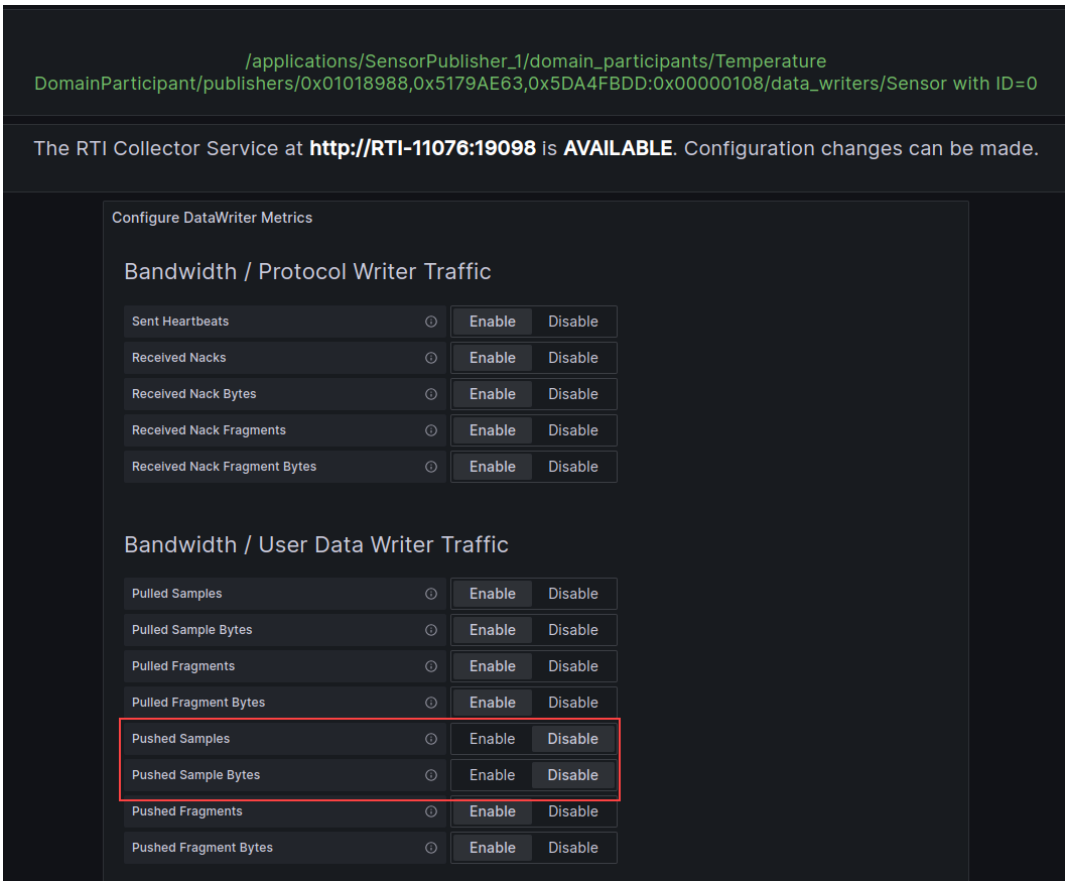
Pulled Samples
Pulled Sample Bytes
Pulled Fragments
Pulled Fragment Bytes
Pushed Samples
Pushed Sample Bytes
Pushed Fragments
Pushed Fragment Bytes
Disabled
Partial

10. In the **DataWriters on Application "SensorPublisher_1"** panel, select **SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0** to open the **DataWriter Metrics** dashboard for the **Sensor with ID=0** *DataWriter*.



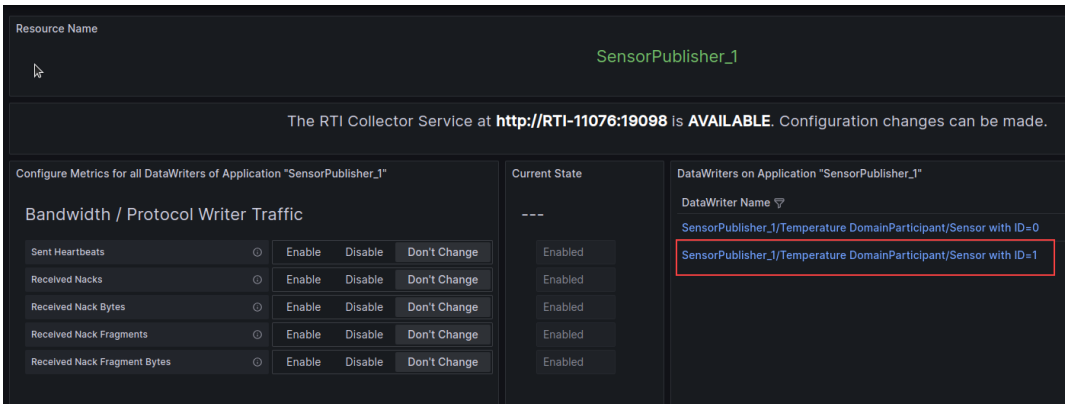
11. Verify that both the **Pushed Samples** and **Pushed Sample Bytes** metrics are disabled.

    Pushed Samples was disabled for all *DataWriters* resources on the **SensorPublisher_1** application in this section of the example. **Pushed Sample Bytes** was disabled in the *Changing metrics collected for a single DataWriter* section.

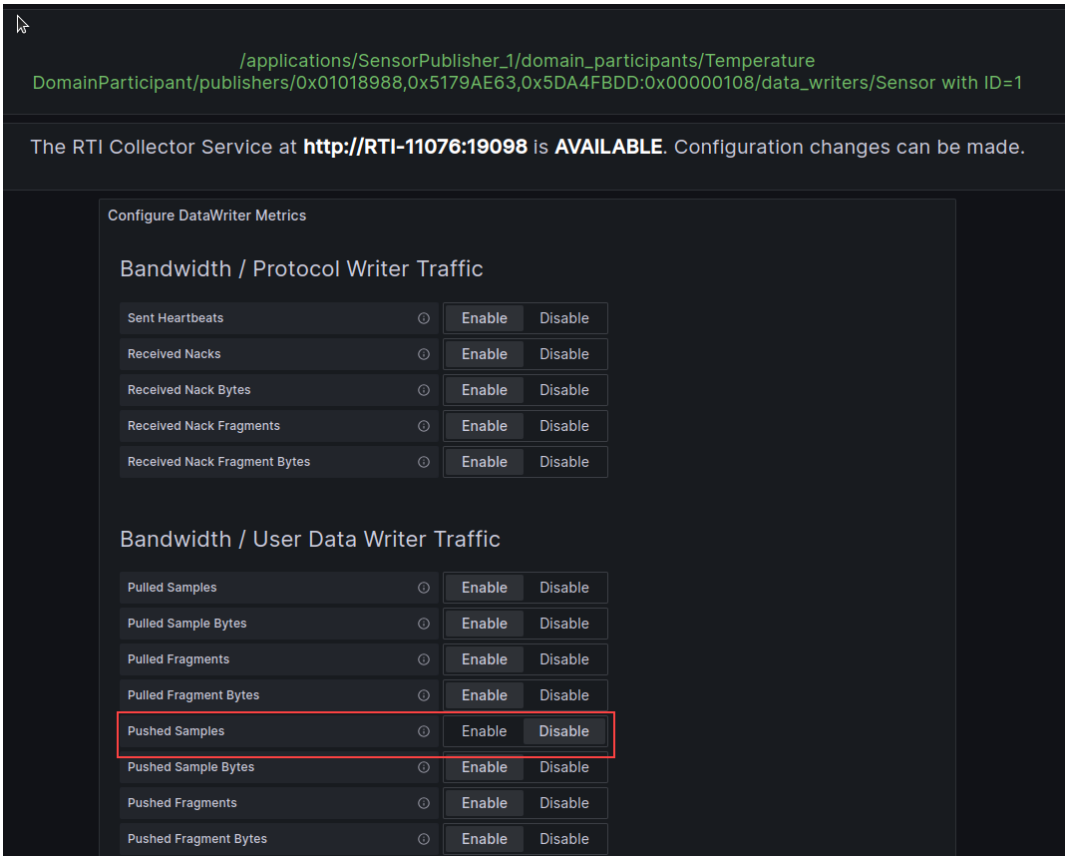


12. Select your browser's **Back** button to go back to the DataWriter Metrics Multi dashboard for the **SensorPublisher_1** application.

13. In the **DataWriters on Application "SensorPublisher_1"** panel, select **SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=1** to open the **DataWriter Metrics** dashboard for the **Sensor with ID=1** *DataWriter*.



14. Verify that only the **Pushed Samples** metric is disabled.



To re-enable the **Pushed Sample Bytes** and **Pushed Samples** metrics, repeat the above steps selecting **enable** instead of **disable**.

7.3.8 Close the Applications

When done working with the example, enter `quit` in each running application to shut it down.

Chapter 8

Telemetry Data

8.1 Introduction

Connext Observability Framework enables you to instrument your *Connext* applications to generate and forward telemetry data. This data is then collected, aggregated, and stored in third-party observability backends such as Prometheus (metrics) or Grafana Loki (logs).

You can then visualize these real-time data points using RTI's Grafana *Observability Dashboards*, or your own custom Grafana dashboards, to get a holistic view of your distributed system.

8.2 Resources

Monitoring Library 2.0 collects telemetry data associated with observable resources. In this release, the observable resources are:

- Application (one-to-one mapping to an OS process)
- *DomainParticipant*
- *Topic*
- *Publisher*
- *Subscriber*
- *DataWriter*
- *DataReader*

Each observable resource is identified by a GUID and a resource name. The GUID is automatically assigned by *Monitoring Library 2.0*, and it is globally unique across all the resources in the system (past and present). The resource GUID can be accessed using the `guid` label associated with each metric. The fully qualified resource name for each observable resource is represented by a Uniform Resource Identifier (URI). The URI strings follow REST best practices for naming. Table 8.1 details of each available resource. See *Metrics* for detailed information about the metrics available for each observable resource in this release.

Table 8.1: Observable Resource Names

Resource	Uniform Resource Identifier (URI)	Dashboard Resource Name	How to Configure
Application	/applications/<AppName>	<AppName>	To set <AppName>, configure the participant_factory_qos.monitoring.application_name QoS policy field for an application. For more information, see MONITORING QosPolicy (DDS Extension)
DomainParticipant	/applications/<AppName>/domain_participants/<ParticipantName>	<AppName>/<ParticipantName>	To set <ParticipantName>, configure the participant_qos.participant_name.name QoS policy field for a *DomainParticipant*. For more information, see ENTITY NAME QosPolicy (DDS Extension)
Topic	/applications/<AppName>/domain_participants/<ParticipantName>/topics/<TopicName>	<TopicName>	<TopicName> is the name of the DDS *Topic*. This resource cannot be configured in the Monitoring QoS.
Publisher	/applications/<AppName>/domain_participants/<ParticipantName>/publishers/<PublisherName>	Dashboards do not show information about Publishers	To set <PublisherName>, configure the publisher_qos.publisher_name.name QoS policy field for a *Publisher*. For more information, see ENTITY NAME QosPolicy (DDS Extension)
Subscriber	/applications/<AppName>/domain_participants/<ParticipantName>/subscribers/<SubscriberName>	Dashboards do not show information about Subscribers	To set <SubscriberName>, configure the publisher_qos.subscriber_name.name QoS policy field for a *Subscriber*. For more information, see ENTITY NAME QosPolicy (DDS Extension)
DataWriter	/applications/<AppName>/domain_participants/<ParticipantName>/publishers/<PublisherName>/data_writers/<DataWriterName>	<AppName>/<ParticipantName>/<DataWriterName>	To set <DataWriterName>, configure the writer_qos.publication_name.name QoS policy field for a *DataWriter*. For more information, see ENTITY NAME QosPolicy (DDS Extension)
DataReader	/applications/<AppName>/	<AppName>/<ParticipantName>/	To set <DataReaderName>, configure

The **Dashboard Resource Name** column describes how resource names appear in *RTI Connext Observability Dashboards*. To generate shorter names, *Observability Dashboards* does not show the resource class name (e.g, domain_participants).

Important: *Observability Framework* does not enforce unique resource names. You are responsible for assigning unique names. When two observable resources have the same name, the commands targeting the resource name are applied to both resources. For example, if two applications have the same name and you change the logging verbosity from *Observability Dashboards*, the change will apply to both applications. Otherwise, not having unique names should not affect functionality because each resource has a unique GUID.

8.2.1 Resource Pattern Definitions

There are two ways to configure telemetry data collection and forwarding. These are the initial configuration of the *Monitoring Library 2.0* (see *Monitoring Library 2.0*) and use of the REST API to dynamically configure the collection of telemetry data (see *Collector Service REST API Reference*) for distribution. In both cases, the Uniform Resource Identifiers (URIs) shown in Table 8.1 are used to identify the observable resources being configured by the XML or REST API. Resource selectors are used in the XML file configuration and in REST API commands to provide a pattern string to match one or more URIs.

A resource selector is an expression made up of components, with each component separated by a forward slash (/). A component is a pattern string that follows the POSIX® fnmatch syntax, or a resource GUID. In addition, you can use XPath-style matching (//) to change the path level by 0 or more components until finding a component matching the pattern following (//).

When specifying resource selectors, POSIX fnmatch pattern matching can be used. The available POSIX® fnmatch special characters are described in Table 8.2.

Table 8.2: POSIX® fnmatch Wild Card Matching

Character	Meaning
/	A / in the pattern string matches a / in the URI. It separates a sequence of mandatory substrings.
*	A * in the pattern string matches 0 or more non-special characters in the URI.
?	A ? in the pattern string matches any single non-special characters in the URI.
[charlist]	Matches any one of the characters in charlist.
[s-e]	Matches any character from [s]tart to [e]nd, inclusive.
\	Escape character for special characters.

Note: To use special characters in a resource selector string, you must escape them using a back slash (\). For example, the resource selector "myWriter[2]" will match a *DataWriter* with name "myWriter2" because of the POSIX® fnmatch processing. If the intent is to use the '[' and ']' characters in the resource selector to actually match a name, the '[' and ']' characters must be escaped as shown here "myWriter\[2\]". This resource selector would match a *DataWriter* with the name "myWriter[2]". Some example resource selectors using POSIX® fnmatch are shown below.

Table 8.3: POSIX® fnmatch Resource Selector Examples

Resource Selector	Description
/applications/SensorPublisher_1	refers to an application named "SensorPublisher_1"
/applications/SensorPublisher_?	refers to applications named "SensorPublisher_" with a single additional character (i.e. "SensorPublisher_1", "SensorPublisher_2", "SensorPublisher_a")
/app*/SensorPublisher_*	refers to applications named "SensorPublisher_" with any number of additional characters, including none (i.e. "SensorPublisher_", "SensorPublisher_1", "SensorPublisher_10", "SensorPublisher_xyz")
/applications/SensorPublisher_1/domain_participants/*/publishers/*/data_writers/Sensor with ID = [12]	refers to data_writers of application "SensorPublisher_1" named "Sensor with ID = 1" or "Sensor with ID = 2"
/applications/SensorPublisher_1/domain_participants/*/publishers/*/data_writers/Sensor with ID = [1-3]	refers to data_writers of application "SensorPublisher_1" named "Sensor with ID = 1", "Sensor with ID = 2", or "Sensor with ID = 3"

In addition to POSIX® fnmatch pattern matching, resource selectors also support the **XPath** (//). The (//) is essentially a relative path indicator that looks for the first occurrence of the text following the (//) in the resource selector. Think of the (//) as a global (*) to match any pattern before the specified text. Use of the XPath (//) can significantly shorten resource selectors. Some example resource selectors using XPath (//) and POSIX® fnmatch are shown below.

Table 8.4: XPath '//' and POSIX® fnmatch Resource Selector Examples

Resource Selector	Description
/applications/SensorPublisher_1//data_writers/Sensor with ID = [12]	refers to data_writers of application "SensorPublisher_1" named "Sensor with ID = 1" or "Sensor with ID = 2"
//data_writers/Sensor with ID = [1-3]	refers to any data_writers named "Sensor with ID = 1", "Sensor with ID = 2", or "Sensor with ID = 3"
//TemperatureDataReader	refers to any DDS entities with the name "TemperatureDataReader"

As mentioned earlier, each DDS entity is assigned a unique identifier or GUID. When creating resource selectors, an entity GUID and entity name are interchangeable. When using a GUID in a resource selector, the format is `GUID(<guid>)`. Some example resource selectors are shown below.

Table 8.5: GUID Resource Selector Examples

Resource Selector	Description
/applications/GUID(aaaaaaaa.bbbbbbbb.ccccc-ccc.dddddddd)//data_writers/Sensor with ID = [12]	refers to data_writers of application with GUID=aaaaaaaa.bbbbbbbb.cccccccc.dddddddd named "Sensor with ID = 1" or "Sensor with ID = 2"
//data_writers/GUID(bbbbbbbb.aaaaaaaa.ccccc-ccc.dddddddd)	refers to the data_writer with GUID=bbbbbbbb.aaaaaaaa.cccccccc.dddddddd
//GUID(dddddddd.cccccccc.bbbbbbbb.aaaaaaaa)	refers to the DDS resource with GUID=dddddddd.cccccccc.bbbbbbbb.aaaaaaaa regardless of the class

Note that the POSIX® fnmatch syntax may not be applied to the `GUID(<guid>)` format.

8.3 Metrics

This section details the metrics you can collect from *Connext* observable resources. Each metric has a unique name and specifies a general feature of a *Connext* observable resource. For example, a *DataWriter* is an observable resource; the metric `dds_data_writer_protocol_sent_heartbeats_total` specifies the total number of heartbeats sent by a *DataWriter*. There are two metric types:

- **Counters**. A *counter* is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart.
- **Gauges**. A *gauge* is a metric that represents a single numerical value that can arbitrarily go up and down.

Observability Framework uses a Prometheus time-series database to store collected metrics. A time series is an instantiation of a metric and represents a stream of timestamped values (measurements) belonging to the same resource as the metric. For example, we could have a time series for the metric `dds_data_writer_protocol_sent_heartbeats_total` corresponding to a *DataWriter* `DW1` identified by a resource GUID `GUID1`.

Labels (in Prometheus) or attributes (in Open Telemetry) identify each metric instantiation or time series. A label is a key/value pair that is associated with a metric. Any given combination of labels for the same metric name identifies a specific instantiation of that metric. For example, the metric `dds_data_writer_protocol_sent_heartbeats_total` for the *DataWriter* `DW1` will have the label `{guid= GUID1}`. All metrics have at least one label called `guid` that uniquely identifies a resource in a Connext system.

In *Observability Framework* there is a special kind of metric called a presence metric. Presence metrics are used to indicate the existence of a resource in a *Connext* system. For example, the `dds_domain_participant_presence` indicates the presence of a *DomainParticipant* in a *Connext* system. There will be a time series for each *DomainParticipant* ever created in the system. The labels associated with a presence metric describe the resource, and they are dependent on the type of resource. For example, a *DomainParticipant* resource has labels such as `` `domain_id` `` and `` `name` ``.

For metrics that are not presence metrics, the only label is the `guid` label identifying the resource to which the metrics apply. You can use the `guid` label to query the description labels of a resource by looking at the presence metric for the resource class.

Observability Framework provides the ability to create an initial configuration for the collection and forwarding of metrics on each observable resource, as well as the ability to dynamically change this configuration at run time. The initial configuration for the collection of metrics is set in the *Monitoring Library 2.0*, as explained in *Monitoring Library 2.0*. Dynamic metric collection configuration changes are done using the REST API as detailed in *Collector Service REST API Reference*. For an example of how to dynamically change the metric collection configuration using the *Observability Dashboards* see *Change the Metric Configuration*.

8.3.1 Metric Pattern Definitions

Observability Framework enables you to select the set of metrics collected and forwarded for a resource both before and during run time. To select metrics, you use metric selector strings. When specifying metric selector strings, POSIX® fnmatch pattern matching should be used as described in Table 8.2. The most common use case is an asterisk (*) to match 0 or more non-special characters. Some example metric selectors using POSIX® fnmatch are shown below.

Table 8.6: POSIX® fnmatch Metric Selector Examples

Metric Selector	Description
dds_application_process_memory_usage_resident_memory_bytes	refers to the metric "dds_application_process_memory_usage_resident_memory_bytes"
dds_application_process_*	refers to all metrics that begin with "dds_application_process_"
dds_*_bytes	refers to metrics that start with "dds_" and end with "_bytes"

8.3.2 Application Metrics

The following tables describe the metrics and labels generated for *Connext* applications. Only the `dds_application_presence` metric has all of the application labels listed in the table below. All other application metrics have the `guid` label only.

Table 8.7: Application Labels

Label or Attribute Name	Description
`controllability_url`	The URL and port for the control server on the *Collector Service* that forwards data for the application. This URL is used when sending remote commands to the *Collector Service* to configure the telemetry data for the application. The remote commands use the *Collector Service* REST API. See *Collector Service REST API Reference* for details on the *Collector Service* REST API.
`guid`	Application resource GUID
`hostname`	Name of the host computer for the application
`process_id`	Process ID for the application
`name`	Fully qualified resource name (/applications/<AppName>)

Table 8.8: Application Metrics

Metric Name	Description	Type
dds_application_presence	Indicates the presence of the application and provides all label values for an application instance	Gauge
dds_application_process_memory_usage_resident_memory_bytes	The application resident memory utilization	Gauge
dds_application_process_memory_usage_virtual_memory_bytes	The application virtual memory utilization	Gauge
dds_application_logging_collection_middleware_level	The middleware collection syslog logging level. See *Logs* for valid values.	Gauge
dds_application_logging_forwarding_middleware_level	The middleware forwarding syslog logging level. See *Logs* for valid values.	Gauge

8.3.3 Participant Metrics

The following tables describe the metrics and labels generated for *Connext DomainParticipants*. Only the dds_domain_participant_presence metric has all of the *DomainParticipant* labels listed in the table below. All other *DomainParticipant* metrics have the guid label only.

The *DomainParticipant* resource contains statistic variable metrics such as dds_domain_participant_udpv4_usage_in_net_pkts_count, dds_domain_participant_udpv4_usage_in_net_pkts_mean, dds_domain_participant_udpv4_usage_in_net_pkts_min, and dds_domain_participant_udpv4_usage_in_net_pkts_max.

These variables are interpreted as follows:

- The metrics with suffix _count represent the total number of packets or bytes over the last Prometheus scraping period.
- The metrics with suffix *_min* represent the minimum mean over the last Prometheus scraping period. For example, dds_domain_participant_udpv4_usage_in_net_pkts_min contains the minimum packets/sec over the last scraping period. The min mean is calculated by choosing the minimum of individual mean values reported by *Monitoring Library 2.0* every participant_factory_qos.monitoring.distribution_settings.periodic_settings.polling_period.
- The metrics with suffix _max represent the maximum mean over the last Prometheus scraping period. For example, dds_domain_participant_udpv4_usage_in_net_pkts_max contains the maximum packets/sec over the last scraping period. The max mean is calculated by choosing the maximum of individual mean values reported by *Monitoring Library 2.0* every participant_factory_qos.monitoring.distribution_settings.periodic_settings.polling_period.

- The metrics with suffix `_mean` represent the mean over the last Prometheus scraping period. For example, `dds_domain_participant_udpv4_usage_in_net_pkts_mean` contains the packets/sec over the last scraping period. If the scraping period is 30 seconds, the metric contains the packets/sec generated within the last 30 seconds. The `dds_domain_participant_udpv4_usage_in_net_pkts_mean` is calculated by averaging all individual mean metrics sent by *Monitoring Library 2.0* to *Observability Collector Service* over the last scraping period.

Table 8.9: Participant Labels

Label or Attribute Name	Description
`guid`	*DomainParticipant* resource GUID
`owner_guid`	Resource GUID of the owner entity (application)
`dds_guid`	*DomainParticipant* DDS GUID
`hostname`	Name of the host computer for the *DomainParticipant*
`process_id`	Process ID for the *DomainParticipant*
`domain_id`	DDS domain ID for the *DomainParticipant*
`platform`	*Connext* architecture as described in the RTI Architecture Abbreviation column in the Platform Notes.
`product_version`	*Connext* product version
`name`	Fully qualified resource name (/applications/<AppName> /domain_participants/<ParticipantName>)

Table 8.10: Participant Metrics

Metric Name	Description	Type
`dds_domain_participant_presence`	Indicates the presence of the *DomainParticipant* and provides all label values for a *DomainParticipant* instance	Gauge
`dds_domain_participant_udpv4_usage_in_net_pkts_count`	The UDPv4 transport in packets count over the last scraping period	Gauge
`dds_domain_participant_udpv4_usage_in_net_pkts_mean`	The UDPv4 transport in packets mean (packets/sec) over the last scraping period	Gauge
`dds_domain_participant_udpv4_usage_in_net_pkts_min`	The UDPv4 transport in packets min mean (packets/sec) over the last scraping period	Gauge
`dds_domain_participant_udpv4_usage_in_net_pkts_max`	The UDPv4 transport in packets max mean (packets/sec) over the last scraping period	Gauge
`dds_domain_participant_udpv4_usage_in_net_bytes_count`	The UDPv4 transport in bytes count over the last scraping period	Gauge
`dds_domain_participant_udpv4_usage_in_net_bytes_mean`	The UDPv4 transport in bytes mean (bytes/sec) over the last scraping period	Gauge

continues on next page

Table 8.10 – continued from previous page

Metric Name	Description	Type
dds_domain_participant_udpv4_usage_in_net_bytes_min	The UDPv4 transport in bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_in_net_bytes_max	The UDPv4 transport in bytes max mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_pkts_count	The UDPv4 transport out packets count over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_pkts_mean	The UDPv4 transport out packets mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_pkts_min	The UDPv4 transport out packets min mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_pkts_max	The UDPv4 transport out packets max mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_bytes_count	The UDPv4 transport out bytes count over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_bytes_mean	The UDPv4 transport out bytes mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_bytes_min	The UDPv4 transport out bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv4_usage_out_net_bytes_max	The UDPv4 transport out bytes max mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_pkts_count	The UDPv6 transport in packets count over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_pkts_mean	The UDPv6 transport in packets mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_pkts_min	The UDPv6 transport in packets min mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_pkts_max	The UDPv6 transport in packets max mean (packets/sec) over the last scraping period	Gauge

continues on next page

Table 8.10 – continued from previous page

Metric Name	Description	Type
dds_domain_participant_udpv6_usage_in_net_bytes_count	The UDPv6 transport in bytes count over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_bytes_mean	The UDPv6 transport in bytes mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_bytes_min	The UDPv6 transport in bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_in_net_bytes_max	The UDPv6 transport in bytes max mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_pkts_count	The UDPv6 transport out packets count over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_pkts_mean	The UDPv6 transport out packets mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_pkts_min	The UDPv6 transport out packets min mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_pkts_max	The UDPv6 transport out packets max mean (packets/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_bytes_count	The UDPv6 transport out bytes count over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_bytes_mean	The UDPv6 transport out bytes mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_bytes_min	The UDPv6 transport out bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_domain_participant_udpv6_usage_out_net_bytes_max	The UDPv6 transport out bytes max mean (bytes/sec) over the last scraping period	Gauge

8.3.4 Topic Metrics

The following tables describe the metrics and labels generated for *Connext Topics*. Only the `dds_topic_presence` metric has all of the *Topic* labels listed in the table below. All other *Topic* metrics have the `guid` label only.

Table 8.11: Topic Labels

Label or Attribute Name	Description
`guid`	*Topic* resource GUID
`owner_guid`	Resource GUID of the owner entity (*DomainParticipant*)
`dds_guid`	*Topic* DDS GUID
`hostname`	Name of the host computer for the *DomainParticipant* this *Topic* is registered with
`domain_id`	DDS domain ID for the *DomainParticipant* this *Topic* is registered with
`topic_name`	The *Topic* name
`registered_type_name`	The registered type name for this *Topic*
`name`	Fully qualified resource name (/applications/<AppName>/domain_participants /<ParticipantName>/topics/<TopicName>)

Table 8.12: Topic Metrics

Metric Name	Description	Type
`dds_topic_presence`	Indicates the presence of the *Topic* and provides all label values for a *Topic* instance	Gauge
`dds_topic_inconsistent_total`	See total_count field in the INCONSISTENT_TOPIC Status	Counter

8.3.5 DataWriter Metrics

The following tables describe the metrics and labels generated for *Connext DataWriters*. Only the `dds_data_writer_presence` metric has all of the *DataWriter* labels listed in the table below. All other *DataWriter* metrics have the `guid` label only.

Table 8.13: DataWriter Labels

Label or Attribute Name	Description
`guid`	*DataWriter* resource GUID
`owner_guid`	Resource GUID of the owner entity (publisher)
`dds_guid`	*DataWriter* DDS GUID
`hostname`	Name of the host computer for the *DomainParticipant* this *DataWriter* is registered with
`domain_id`	DDS domain ID for the *DomainParticipant* this *DataWriter* is registered with
`topic_name`	The *Topic* name for this *DataWriter*
`registered_type_name`	The registered type name for this *DataWriter*
`name`	Fully qualified resource name (/applications/<AppName>/domain_participants /<ParticipantName>/publishers/<PublisherName>/data_writers/<DataWriterName>)
`participant_guid`	Resource GUID of the *DomainParticipant* this *DataWriter* is registered with

Table 8.14: DataWriter Metrics

Metric Name	Description	Type
dds_data_writer_presence	Indicates the presence of the *DataWriter* and provides all label values for a *DataWriter* instance	Gauge
dds_data_writer_liveliness_lost_total	See total_count field in the LIVELINESS_LOST Status	Counter
dds_data_writer_deadline_missed_total	See total_count field in the OFFERED_DEADLINE_MISSED Status	Counter
dds_data_writer_incompatible_qos_total	See total_count field in the OFFERED_INCOMPATIBLE_QOS Status	Counter
dds_data_writer_reliable_cache_full_total	See full_reliable_writer_cache field in the RELIABLE_WRITER_CACHE_CHANGED Status	Counter
dds_data_writer_reliable_cache_high_watermark_total	See high_watermark_reliable_writer_cache field in the RELIABLE_WRITER_CACHE_CHANGED Status	Counter
dds_data_writer_reliable_cache_unack_samples	See unacknowledged_sample_count field in the RELIABLE_WRITER_CACHE_CHANGED Status	Gauge
dds_data_writer_reliable_cache_unack_samples_peak	See unacknowledged_sample_count_peak field in the RELIABLE_WRITER_CACHE_CHANGED Status	Gauge
dds_data_writer_reliable_cache_replaced_unack_samples_total	See replaced_unacknowledged_sample_count field in the RELIABLE_WRITER_CACHE_CHANGED Status	Counter
dds_data_writer_reliable_reader_activity_inactive_count	See inactive_count field in the RELIABLE_READER_ACTIVITY_CHANGED Status	Gauge
dds_data_writer_cache_samples_peak	See sample_count_peak field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_data_writer_cache_samples	See sample_count field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_data_writer_cache_alive_instances	See alive_instance_count field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_data_writer_cache_alive_instances_peak	See alive_instance_count_peak field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_data_writer_protocol_pushed_samples_total	See pushed_sample_count field in the DATA_WRITER_PROTOCOL_STATUS	Counter
dds_data_writer_protocol_pushed_sample_bytes_total	See pushed_sample_bytes field in the DATA_WRITER_PROTOCOL_STATUS	Counter
dds_data_writer_protocol_sent_heartbeats_total	See sent_heartbeat_count field in the DATA_WRITER_PROTOCOL_STATUS	Counter
dds_data_writer_protocol_...	See pulled_sample_count field in the	Counter

8.3.6 DataReader Metrics

The following tables describe the metrics and labels generated for *Connext DataReaders*. Only the `ddsd_datareader_presence` metric has all of the *DataReader* labels listed in the table below. All other *DataReader* metrics have the `guid` label only.

Table 8.15: DataReader Labels

Label or Attribute Name	Description
`guid`	*DataReader* resource GUID
`owner_guid`	Resource GUID of the owner entity (subscriber)
`dds_guid`	*DataReader* DDS GUID
`hostname`	Name of the host computer for the *DomainParticipant* this *DataReader* is registered with
`domain_id`	DDS domain ID for the *DomainParticipant* this *DataReader* is registered with
`topic_name`	The *Topic* name for this *DataReader*
`registered_type_name`	The registered type name for this *DataReader*
`name`	Fully qualified resource name (/applications/<AppName>/domain_participants/<ParticipantName> /subscribers/<SubscriberName>/data_readers/<DataReaderName>)
`participant_guid`	Resource GUID of the *DomainParticipant* this *DataReader* is registered with

Table 8.16: DataReader Metrics

Metric Name	Description	Type
`dds_data_reader_presence`	Indicates the presence of the *DataReader* and provides all label values for a *DataReader* instance	Gauge
`dds_data_reader_sample_rejected_total`	See total_count field in the SAMPLE_REJECTED Status	Counter
`dds_data_reader_liveliness_not_alive_count`	See not_alive_count field in the LIVELINESS_CHANGED Status	Gauge
`dds_data_reader_deadline_missed_total`	See total_count field in the REQUESTED_DEADLINE_MISSED Status	Counter
`dds_data_reader_incompatible_qos_total`	See total_count field in the REQUESTED_INCOMPATIBLE_QOS Status	Counter
`dds_data_reader_sample_lost_total`	See total_count field in the SAMPLE_LOST Status	Counter
`dds_data_reader_cache_samples_peak`	See sample_count_peak field in the DATA_READER_CACHE_STATUS	Gauge

continues on next page

Table 8.16 – continued from previous page

Metric Name	Description	Type
dds_data_reader_cache_samples	See sample_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_old_source_timestamp_samples_total	See ts_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_data_reader_cache_tolerance_source_ts_dropped_samples_total	See tolerance_source_timestamp_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_data_reader_cache_content_filter_dropped_samples_total	See content_filter_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_data_reader_cache_replaced_dropped_samples_total	See replaced_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_data_reader_cache_samples_dropped_by_instance_replaced_total	See total_samples_dropped_by_instance_replacement field in the DATA_READER_CACHE_STATUS	Counter
dds_data_reader_cache_alive_instances	See alive_instance_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_alive_instances_peak	See alive_instance_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_no_writers_instances	See no_writers_instance_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_no_writers_instances_peak	See no_writers_instance_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_disposed_instances	See disposed_instance_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_disposed_instances_peak	See disposed_instance_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
dds_data_reader_cache_compressed_samples_total	See compressed_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_data_reader_protocol_received_samples_total	See received_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_received_sample_bytes_total	See received_sample_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_duplicate_samples_total	See duplicate_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_duplicate_sample_bytes_total	See duplicate_sample_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter

continues on next page

Table 8.16 – continued from previous page

Metric Name	Description	Type
dds_data_reader_protocol_received_heartbeats_total	See received_heartbeat_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_sent_nacks_total	See sent_nack_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_sent_nack_bytes_total	See sent_nack_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_rejected_samples_total	See rejected_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_out_of_range_rejected_samples_total	See out_of_range_rejected_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_received_fragments_total	See received_fragment_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_dropped_fragments_total	See dropped_fragment_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_reassembled_samples_total	See reassembled_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_sent_nack_fragments_total	See sent_nack_fragment_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_data_reader_protocol_sent_nack_fragment_bytes_total	See sent_nack_fragment_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter

8.3.7 Derived Metrics Generated by Prometheus Recording Rules

Prometheus provides a capability called Recording Rules. The following text is an excerpt from the Prometheus documentation.

```
Recording rules allow you to precompute frequently needed or computationally
expensive expressions and save their result as a new set of time series.
Querying the precomputed result will then often be much faster than executing
the original expression every time it is needed. This is especially useful for
dashboards, which need to query the same expression repeatedly every time they
refresh.
```

A Prometheus recording rule generates a new metric time series with new values calculated at the frequency at which the rule is run. The recording rules in *Observability Framework* are run every 10 seconds, meaning

there is an evaluation and update to the associated derived metric every 10 seconds. *Observability Framework* uses Prometheus recording rules to generate three types of derived metrics.

- DDS entity proxy metrics
- raw `error` metrics
- aggregated `error` metrics.

Each of these derived metric types is discussed in detail below.

The Grafana dashboards provided with *Observability Framework* make use of the `error` metrics generated by Prometheus recording rules. The aggregated `error` metrics are used on the Alert Home dashboard, while the raw `error` metrics are used on other dashboards.

DDS Entity Proxy Metrics

The DDS entity proxy metrics are used in the recording rules for the raw `error` metrics and are always 0. The proxy metrics are used to make sure the rules evaluate to known good values in cases where the underlying metrics are not available.

Table 8.17: DDS Entity Proxy Metrics

Metric Name	Description
`dds_application_empty_metric`	A proxy for applications metrics that always provides a value of zero.
`dds_domain_participant_empty_metric`	A proxy for applications metrics that always provides a value of zero.
`dds_topic_empty_metric`	A proxy for applications metrics that always provides a value of zero.
`dds_data_writer_empty_metric`	A proxy for applications metrics that always provides a value of zero.
`dds_data_reader_empty_metric`	A proxy for applications metrics that always provides a value of zero.

Raw Error Metrics

Raw `error` metrics are derived for select metrics by doing a boolean comparison to a predefined limit. The raw `error` metrics are created by converting the monotonically increasing value of a counter metric into a rate, comparing that rate to a limit, and returning a boolean value. The returned boolean value is 1 if the limit is exceeded, otherwise 0. In the Grafana dashboards, a value of 0 indicates a healthy condition for the `error` metric, while a value of 1 indicates a fail condition.

Recording rules have been created to generate a derived raw `error` metric for all of the metrics listed in Table 8.18 and Table 8.19.

Enabled Raw Error Metrics

A set of recording rules have been created that are useful for detecting failures in all systems. These rules detect conditions that are not expected to occur in a system that is operating correctly. The rules for these "enabled" metrics test if the underlying metric has exceeded a limit of 0. Note the `>bool 0` comparison operator in each of the recording rules. A value greater than 0 in any of these metrics will result in an alert indication in the dashboards. This set of metrics is "enabled" because any increase in the underlying metric indicates an unexpected condition in DDS. Table 8.18 lists derived Raw `error` metrics that are "enabled".

Table 8.18: Raw Error Metrics (enabled)

Metric Name	Recording Rule
dds_data_reader_cache_content_filter_dropped_samples_errors	rate(dds_data_reader_cache_content_filter_dropped_samples_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_replaced_dropped_samples_errors	rate(dds_data_reader_cache_replaced_dropped_samples_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_samples_dropped_by_instance_replaced_errors	rate(dds_data_reader_cache_samples_dropped_by_instance_replaced_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_rejected_samples_errors	rate(dds_data_reader_protocol_rejected_samples_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_out_of_range_rejected_samples_errors	rate(dds_data_reader_protocol_out_of_range_rejected_samples_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_dropped_fragments_errors	rate(dds_data_reader_protocol_dropped_fragments_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_topic_inconsistent_errors	rate(dds_topic_inconsistent_total[1m]) >bool 0 or dds_topic_empty_metric
dds_data_writer_incompatible_qos_errors	rate(dds_data_writer_incompatible_qos_total[1m]) >bool 0 or dds_data_writer_empty_metric
dds_data_reader_incompatible_qos_errors	rate(dds_data_reader_incompatible_qos_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_writer_liveliness_lost_errors	rate(dds_data_writer_liveliness_lost_total[1m]) >bool 0 or dds_data_writer_empty_metric
dds_data_writer_reliable_reader_activity_inactive_count_errors	rate(dds_data_writer_reliable_reader_activity_inactive_count[1m]) >bool 0 or dds_data_writer_empty_metric
dds_data_reader_liveliness_not_alive_count_errors	rate(dds_data_reader_liveliness_not_alive_count[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_tolerance_source_ts_dropped_samples_errors	rate(dds_data_reader_cache_tolerance_source_ts_dropped_samples_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_writer_deadline_missed_errors	rate(dds_data_writer_deadline_missed_total[1m]) >bool 0 or dds_data_writer_empty_metric
dds_data_reader_deadline_missed_errors	rate(dds_data_reader_deadline_missed_total[1m]) >bool 0 or dds_data_reader_empty_metric
dds_data_writer_reliable_cache_replaced_unack_samples_errors	rate(dds_data_writer_reliable_cache_replaced_unack_samples_total[1m]) >bool 0 or dds_data_writer_empty_metric
dds_data_reader_sample_lost_errors	rate(dds_data_reader_sample_lost_total[1m]) >bool 0 or dds_data_reader_empty_metric

Disabled Raw Error Metrics

Additional recording rules have been created that by default are not useful for detecting failures since the meaningful rules depend on comparisons to values that will be dependent on actual system requirements. The rules for the "disabled" metrics test to see if the underlying metric is less than a limit of 0, ensuring that the derived raw `error` metric never indicates a failure, hence disabled. Note the `<bool 0` comparison operator in each of the recording rules. This set of metrics is "disabled" because a meaningful limit that would indicate a fail condition cannot be determined without additional knowledge of the system.

Users may modify a "disabled" rule to compare against a value that is meaningful to their system. For example, if users want to be notified when the number of repaired samples over the last minute exceeds 10, then they would modify the rule

```
rate(dds_data_writer_protocol_pulled_samples_total[1m]) <bool 0 or dds_data_
↪writer_empty_metric
```

To

```
rate(dds_data_writer_protocol_pulled_samples_total[1m]) >bool 10 or dds_data_
↪writer_empty_metric
```

For complete instructions on how to enable these metrics and display them in the dashboards, see *Enable a Raw Error Metric*.

The "disabled" rules have been created as a convenience for the user. However, only a few of these rules may be useful for any specific system. Table 8.19 lists derived raw `error` metrics that are "disabled".

Table 8.19: Raw Error Metrics (disabled)

Metric Name	Recording Rule
`dds_data_writer_protocol_sent_heartbeats_errors`	rate(dds_data_writer_protocol_sent_heartbeats_total[1m] <bool 0 or dds_data_writer_empty_metric
`dds_data_writer_protocol_received_nacks_errors`	rate(dds_data_writer_protocol_received_nacks_total[1m]) <bool 0 or dds_data_writer_empty_metric
`dds_data_writer_protocol_received_nack_bytes_errors`	rate(dds_data_writer_protocol_received_nack_bytes_total[1m]) <bool 0 or dds_data_writer_empty_metric
`dds_data_writer_protocol_received_nack_fragments_errors`	rate(dds_data_writer_protocol_received_nack_fragments_total[1m]) <bool 0 or dds_data_writer_empty_metric
`dds_data_writer_protocol_received_nack_fragment_bytes_errors`	rate(dds_data_writer_protocol_received_nack_fragment_bytes_total[1m]) <bool 0 or dds_data_writer_empty_metric
`dds_data_reader_protocol_received_heartbeats_errors`	rate(dds_data_reader_protocol_received_heartbeats_total[1m]) <bool 0 or dds_data_reader_empty_metric
`dds_data_reader_protocol_sent_nacks_errors`	rate(dds_data_reader_protocol_sent_nacks_total[1m]) <bool 0 or dds_data_reader_empty_metric

continues on next page

Table 8.19 – continued from previous page

Metric Name	Recording Rule
dds_data_reader_protocol_sent_nack_bytes_errors	rate(dds_data_reader_protocol_sent_nack_bytes_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_sent_nack_fragments_errors	rate(dds_data_reader_protocol_sent_nack_fragments_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_sent_nack_fragment_bytes_errors	rate(dds_data_reader_protocol_sent_nack_fragment_bytes_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_writer_protocol_pulled_samples_errors	rate(dds_data_writer_protocol_pulled_samples_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pulled_sample_bytes_errors	rate(dds_data_writer_protocol_pulled_sample_bytes_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pulled_fragments_errors	rate(dds_data_writer_protocol_pulled_fragments_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pulled_fragment_bytes_errors	rate(dds_data_writer_protocol_pulled_fragment_bytes_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pushed_samples_errors	rate(dds_data_writer_protocol_pushed_samples_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pushed_sample_bytes_errors	rate(dds_data_writer_protocol_pushed_sample_bytes_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pushed_fragments_errors	rate(dds_data_writer_protocol_pushed_fragments_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_pushed_fragment_bytes_errors	rate(dds_data_writer_protocol_pushed_fragment_bytes_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_reader_cache_compressed_samples_errors	rate(dds_data_reader_cache_compressed_samples_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_duplicate_samples_errors	rate(dds_data_reader_protocol_duplicate_samples_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_duplicate_sample_bytes_errors	rate(dds_data_reader_protocol_duplicate_sample_bytes_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_received_samples_errors	rate(dds_data_reader_protocol_received_samples_total[1m]) <bool 0 or dds_data_reader_empty_metric

continues on next page

Table 8.19 – continued from previous page

Metric Name	Recording Rule
dds_data_reader_protocol_received_sample_bytes_errors	rate(dds_data_reader_protocol_received_sample_bytes_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_received_fragments_errors	rate(dds_data_reader_protocol_received_fragments_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_protocol_reassembled_samples_errors	rate(dds_data_reader_protocol_reassembled_samples_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_application_process_memory_usage_resident_memory_bytes_errors	rate(dds_application_process_memory_usage_resident_memory_bytes[1m]) <bool 0 or dds_application_empty_metric
dds_application_process_memory_usage_virtual_memory_bytes_errors	rate(dds_application_process_memory_usage_virtual_memory_bytes[1m]) <bool 0 or dds_application_empty_metric
dds_domain_participant_udpv4_usage_in_net_pkts_errors	rate(dds_domain_participant_udpv4_usage_in_net_pkts_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv4_usage_in_net_bytes_errors	rate(dds_domain_participant_udpv4_usage_in_net_bytes_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv4_usage_out_net_pkts_errors	rate(dds_domain_participant_udpv4_usage_out_net_pkts_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv4_usage_out_net_bytes_errors	rate(dds_domain_participant_udpv4_usage_out_net_bytes_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv6_usage_in_net_pkts_errors	rate(dds_domain_participant_udpv6_usage_in_net_pkts_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv6_usage_in_net_bytes_errors	rate(dds_domain_participant_udpv6_usage_in_net_bytes_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv6_usage_out_net_pkts_errors	rate(dds_domain_participant_udpv6_usage_out_net_pkts_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_domain_participant_udpv6_usage_out_net_bytes_errors	rate(dds_domain_participant_udpv6_usage_out_net_bytes_mean[1m]) <bool 0 or dds_domain_participant_empty_metric
dds_data_writer_reliable_cache_full_errors	rate(dds_data_writer_reliable_cache_full_total[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_reliable_cache_high_watermark_errors	rate(dds_data_writer_reliable_cache_high_watermark_total[1m]) <bool 0 or dds_data_writer_empty_metric

continues on next page

Table 8.19 – continued from previous page

Metric Name	Recording Rule
dds_data_writer_reliable_cache_unack_samples_errors	rate(dds_data_writer_reliable_cache_unack_samples[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_reliable_cache_unack_samples_peak_errors	rate(dds_data_writer_reliable_cache_unack_samples_peak[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_protocol_send_window_size_errors	rate(dds_data_writer_protocol_send_window_size[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_cache_samples_errors	rate(dds_data_writer_cache_samples[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_cache_samples_peak_errors	rate(dds_data_writer_cache_samples_peak[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_cache_alive_instances_errors	rate(dds_data_writer_cache_alive_instances[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_writer_cache_alive_instances_peak_errors	rate(dds_data_writer_cache_alive_instances_peak[1m]) <bool 0 or dds_data_writer_empty_metric
dds_data_reader_sample_rejected_errors	rate(dds_data_reader_sample_rejected_total[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_samples_errors	rate(dds_data_reader_cache_samples[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_samples_peak_errors	rate(dds_data_reader_cache_samples_peak[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_alive_instances_errors	rate(dds_data_reader_cache_alive_instances[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_alive_instances_peak_errors	rate(dds_data_reader_cache_alive_instances_peak[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_no_writers_instances_errors	rate(dds_data_reader_cache_no_writers_instances[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_no_writers_instances_peak_errors	rate(dds_data_reader_cache_no_writers_instances_peak[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_disposed_instances_errors	rate(dds_data_reader_cache_disposed_instances[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_disposed_instances_peak_errors	rate(dds_data_reader_cache_disposed_instances_peak[1m]) <bool 0 or dds_data_reader_empty_metric
dds_data_reader_cache_old_source_ts_dropped_samples_errors	rate(dds_data_reader_cache_old_source_ts_dropped_samples_total[1m]) <bool 0 or dds_data_reader_empty_metric

Aggregated Error Metrics

The aggregated `error` metrics create a status roll-up for a group of metrics in a particular category. These aggregated `error` metrics are used in the **Alert Home** dashboard to provide a high-level view of alerts grouped by category. The categories are **Bandwidth**, **Saturation**, **Data Loss**, **System Errors**, and **Delays**. The aggregated `error` metrics are created by adding together all of the raw `error` metrics assigned to a category and clamping the values at 1, the value that indicates a failed condition. Table 8.20 shows all of the aggregated `error` metrics and the rule used to generate them. Note the use of the raw `error` metrics in the rules.

Table 8.20: Aggregate Error Metrics

Metric Name	Recording Rule
dds_excessive_bandwidth_errors	clamp_max ((sum (dds_custom_excessive_bandwidth_errors) + sum (dds_data_writer_protocol_sent_heartbeats_errors) + sum (dds_data_writer_protocol_received_nacks_errors) + sum (dds_data_writer_protocol_received_nack_bytes_errors) + sum (dds_data_writer_protocol_received_nack_fragments_errors) + sum (dds_data_writer_protocol_received_nack_fragment_bytes_errors) + sum (dds_data_reader_protocol_received_heartbeats_errors) + sum (dds_data_reader_protocol_sent_nacks_errors) + sum (dds_data_reader_protocol_sent_nack_bytes_errors) + sum (dds_data_reader_protocol_sent_nack_fragments_errors) + sum (dds_data_reader_protocol_sent_nack_fragment_bytes_errors) + sum (dds_data_writer_protocol_pulled_samples_errors) + sum (dds_data_writer_protocol_pulled_sample_bytes_errors) + sum (dds_data_writer_protocol_pulled_fragments_errors) + sum (dds_data_writer_protocol_pulled_fragment_bytes_errors) + sum (dds_data_writer_protocol_pushed_samples_errors) + sum (dds_data_writer_protocol_pushed_sample_bytes_errors) + sum (dds_data_writer_protocol_pushed_fragments_errors) + sum (dds_data_writer_protocol_pushed_fragment_bytes_errors) + sum (dds_data_reader_cache_content_filter_dropped_samples_errors) + sum (dds_data_reader_cache_compressed_samples_errors) + sum (dds_data_reader_protocol_duplicate_samples_errors) + sum (dds_data_reader_protocol_duplicate_sample_bytes_errors) + sum (dds_data_reader_protocol_received_samples_errors) + sum (dds_data_reader_protocol_received_sample_bytes_errors) + sum (dds_data_reader_protocol_received_fragments_errors) + sum (dds_data_reader_protocol_reassembled_samples_errors)), 1)
dds_saturation_errors	clamp_max ((sum (dds_custom_saturation_errors) + sum (dds_application_process_memory_usage_resident_memory_bytes_errors) + sum (dds_application_process_memory_usage_virtual_memory_bytes_errors) + sum (dds_domain_participant_udpv4_usage_in_net_pkts_errors) + sum (dds_domain_participant_udpv4_usage_in_net_bytes_errors) + sum (dds_domain_participant_udpv4_us-

Enable a Raw Error Metric

Note: The Grafana user must have Admin privileges to make any changes to the Grafana dashboards.

Use the following steps to enable any of the "disabled" metrics in your system:

1. Update the raw `error` rule to **enable** the calculation and provide a limit. See *Update the Recording Rule for the Derived Metric* below.
2. Update the Alert "Category" dashboard to update the background color of the OK/ERROR and State panels for the enabled metric. See *Update the Alert "Category" Dashboard* below.
3. Update the "Entity" status dashboard to update the query and background color in the State panel. See *Update the "Entity" Status Dashboard* below.

The example that follows uses the `dds_data_reader_cache_alive_instances_errors` metric to update/enable a rule to detect any *DataReader* that has more than 3 ALIVE instances in its cache.

Update the Recording Rule for the Derived Metric

Locate the recording rule for the `dds_data_reader_cache_alive_instances_errors` metric in the `monitoring_recording_rules.yml` file located in the `rti_workspace/<version>/observability/prometheus` directory.

```
# User Config Required
  - record: dds_data_reader_cache_alive_instances_errors
    expr: >
      rate(dds_data_reader_cache_alive_instances[1m]) <bool 0 or dds_data_
↪reader_empty_metric
```

The `dds_data_reader_cache_alive_instances` metric is a gauge metric, meaning we want to use the absolute value for our limit check rather than the rate. In the following example recording rule, we want to update the limit test so that the error will be active whenever the value is greater than `3`.

```
# User Config Required
  - record: dds_data_reader_cache_alive_instances_errors
    expr: >
      dds_data_reader_cache_alive_instances >bool 3 or dds_data_reader_empty_
↪metric
```

Important: After updating the `monitoring_recording_rules.yml` file, you must restart all Docker containers for *Observability Framework* by running `rtiobservability -t` followed by `rtiobservability -s`. The Prometheus server will read the updated file after restarting the containers.

Update the Alert "Category" Dashboard

Note: The Grafana images in this section were generated with Grafana version 9.2.1. If you are using a different version of Grafana, the interface may be slightly different.

Locate the Alert "Category" dashboard for the metric rule you are enabling. The metric in our example, `dds_data_reader_cache_alive_instances_errors`, is in the **Saturation** group (see Table 8.20), so the **Alert Saturation** dashboard is used in the following steps.

1. Go to **Dashboards > Browse** to open the list of dashboards.

2. Select the **Alert Saturation** dashboard from the list.

3. Once on the **Alert Saturation** dashboard, scroll down to the **Alive Instances** row under the **Reader Cache** section.

4. Select **Alive Instances > Edit** from the status indicator panel menu.

5. In the right panel, scroll down until you find the **Value mappings** section.

6. Click the gray color circle next to the **OK** mapping to select a new color for the panel "OK" indication.

7. Select the large green circle in the panel. The updated **OK** value should change from gray to green.

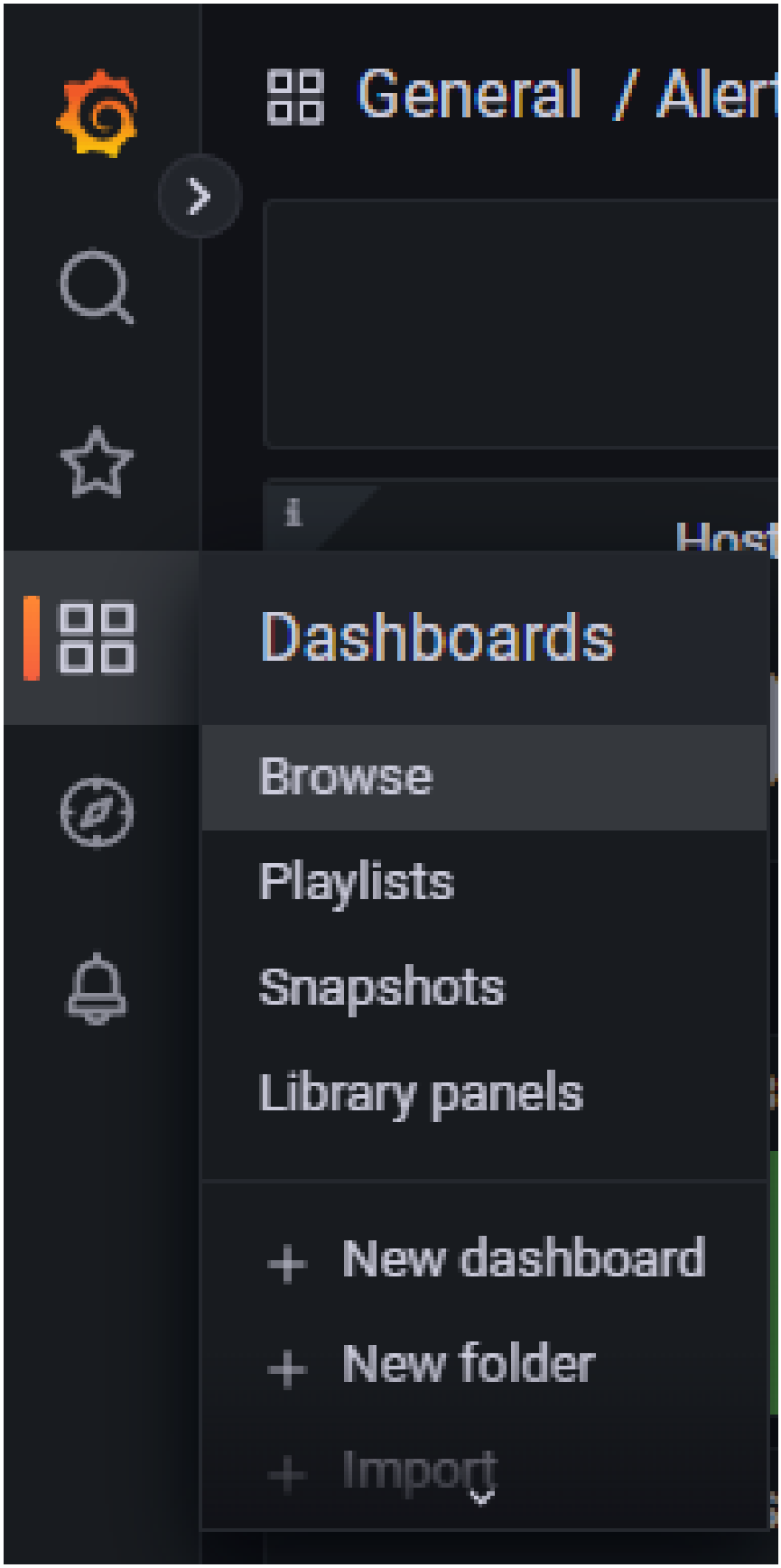
8. Select **Apply** at the top right to apply the change and return to the **Alert Saturation** dashboard.

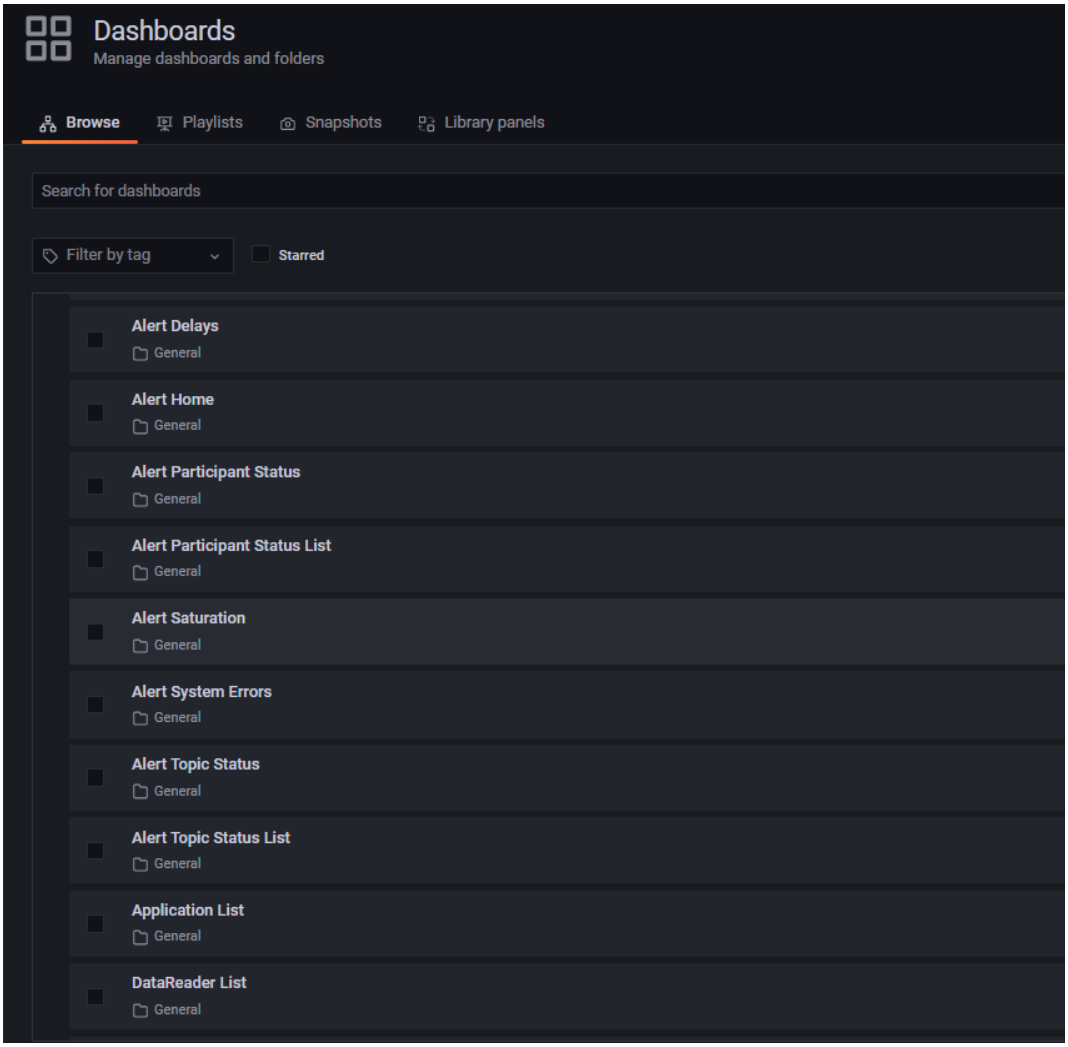
9. Select **Alive Instances > Edit** from the status indicator panel menu.

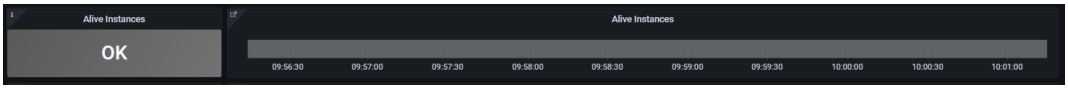
10. In the right panel, scroll down to the **Thresholds** section.

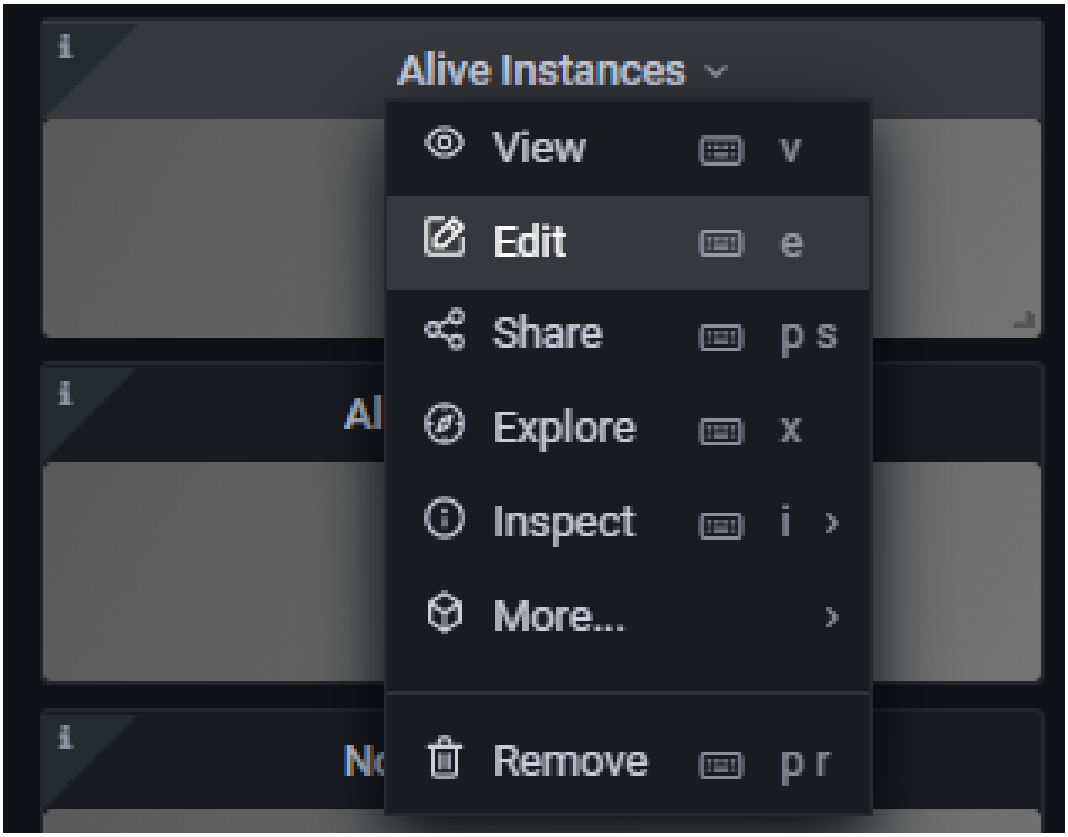
11. Click the gray circle next to **Base** to select a new base color for the **Thresholds** panel.

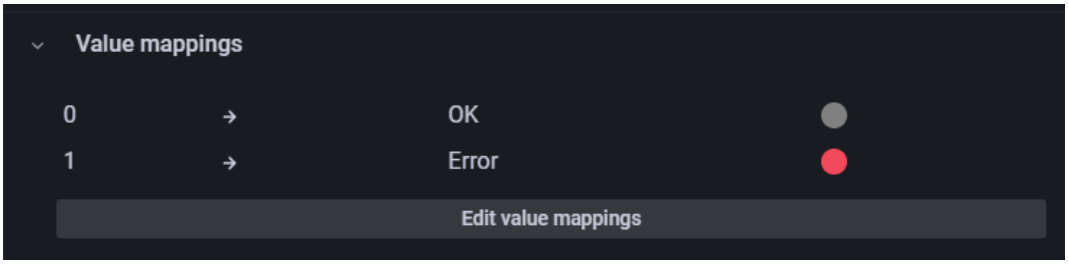
12. Select the large green circle in the panel. The updated **Threshold** base value should change to green.

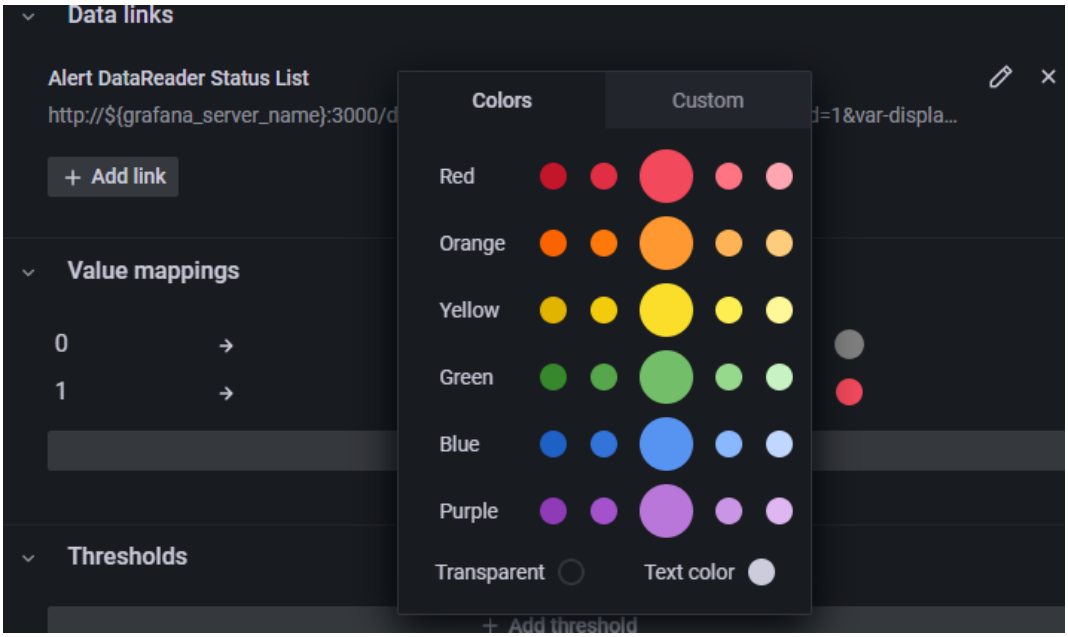

General / Alert
Dashboards
Browse
Playlists
Snapshots
Library panels
New dashboard
New folder
Import

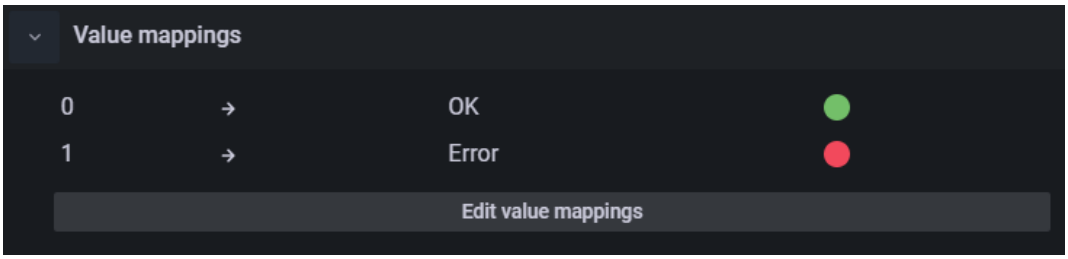

Dashboards
Manage dashboards and folders
Browse
Playlists
Snapshots
Library panels
Search for dashboards
Filter by tag
Starred
Alert Delays
General
Alert Home
General
Alert Participant Status
General
Alert Participant Status List
General
Alert Saturation
General
Alert System Errors
General
Alert Topic Status
General
Alert Topic Status List
General
Application List
General
DataReader List
General


Alive Instances
OK
Alive Instances
09:56:30
09:57:00
09:57:30
09:58:00
09:58:30
09:59:00
09:59:30
10:00:00
10:00:30
10:01:00

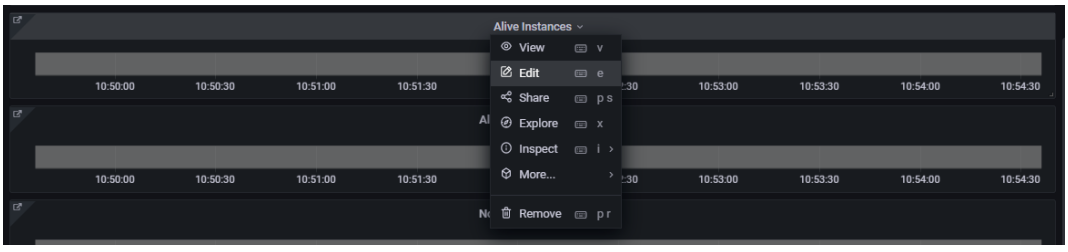

Alive Instances
View v
Edit e
Share p s
Explore x
Inspect i
More...
Remove p r

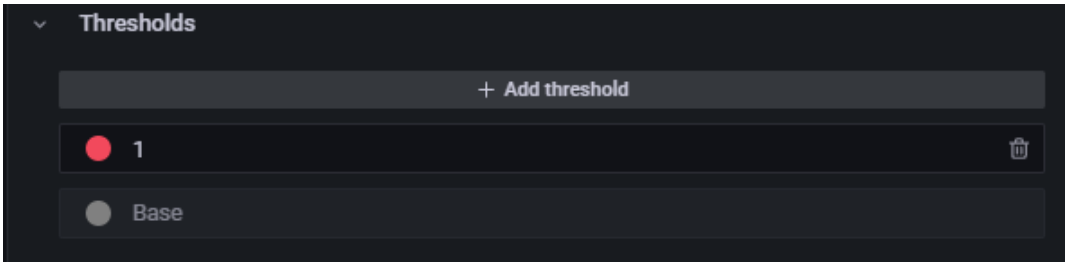

Value mappings
0 → OK
1 → Error
Edit value mappings

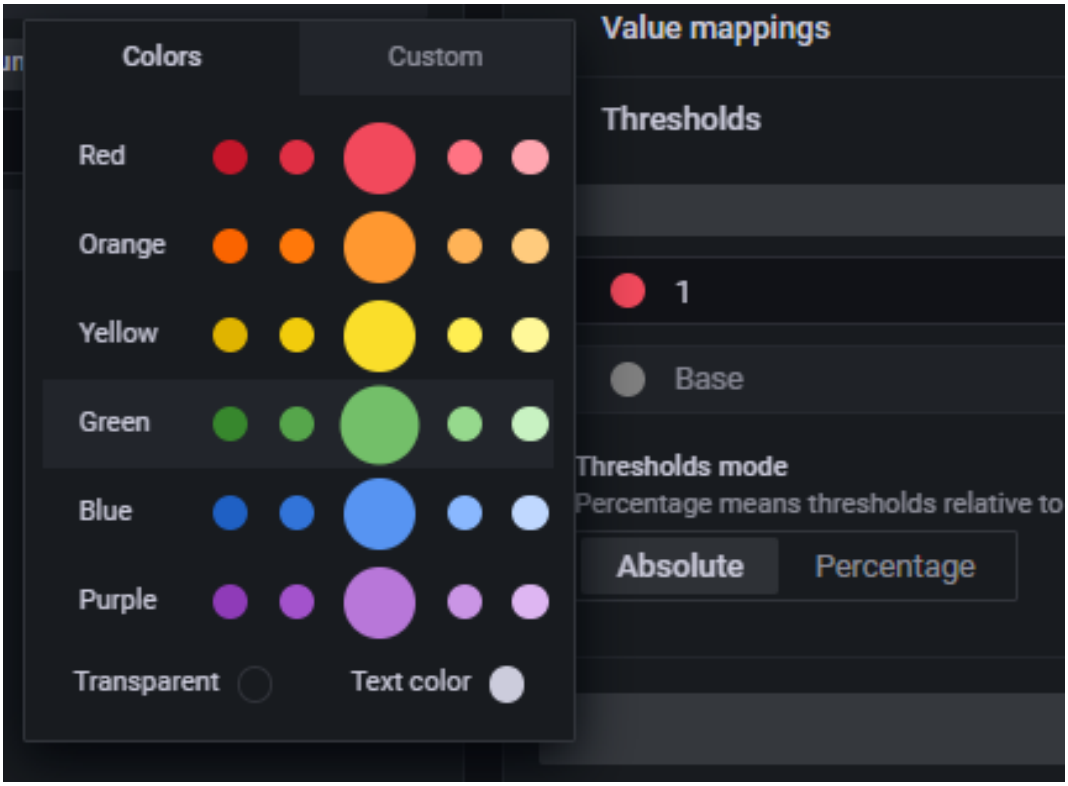

Data links
Alert DataReader Status List
http://${grafana_server_name}:3000/d
d=1&var-displa...
+ Add link
Value mappings
0
1
Thresholds
+ Add threshold
Colors
Custom
Red
Orange
Yellow
Green
Blue
Purple
Transparent
Text color

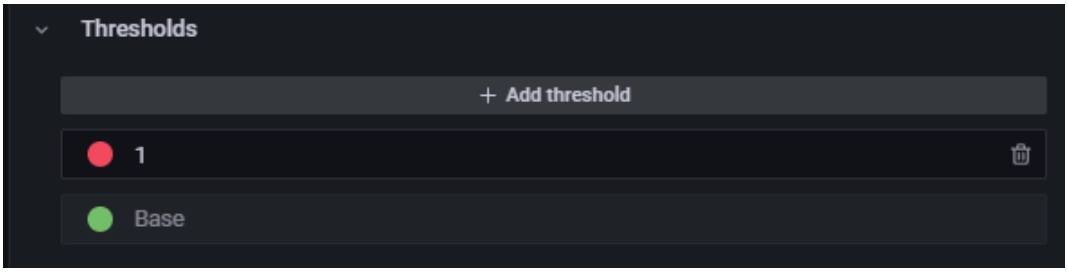

Value mappings
0
OK
1
Error
Edit value mappings


Discard
Save
Apply


Alive Instances
View
Edit
Share
Explore
Inspect
More...
Remove
10:50:00
10:50:30
10:51:00
10:51:30
10:53:00
10:53:30
10:54:00
10:54:30

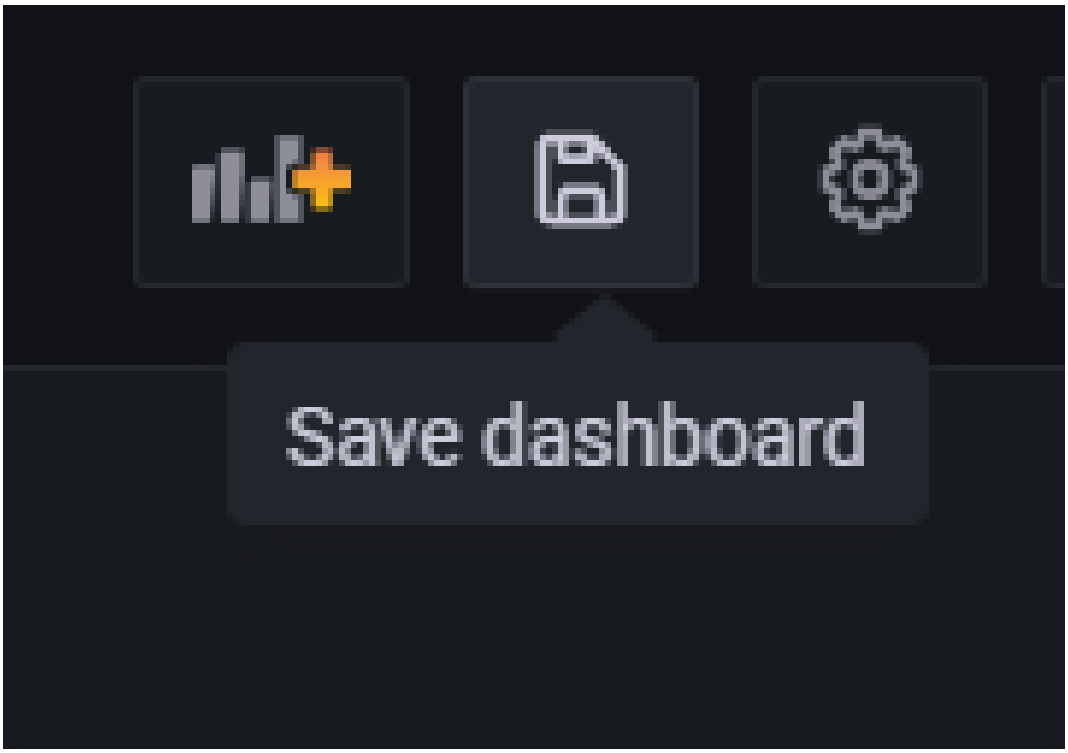

Thresholds
+ Add threshold
1
Base


Colors
Custom
Red
Orange
Yellow
Green
Blue
Purple
Transparent
Text color
Value mappings
Thresholds
1
Base
Thresholds mode
Percentage means thresholds relative to
Absolute
Percentage


Thresholds
Add threshold
1
Base

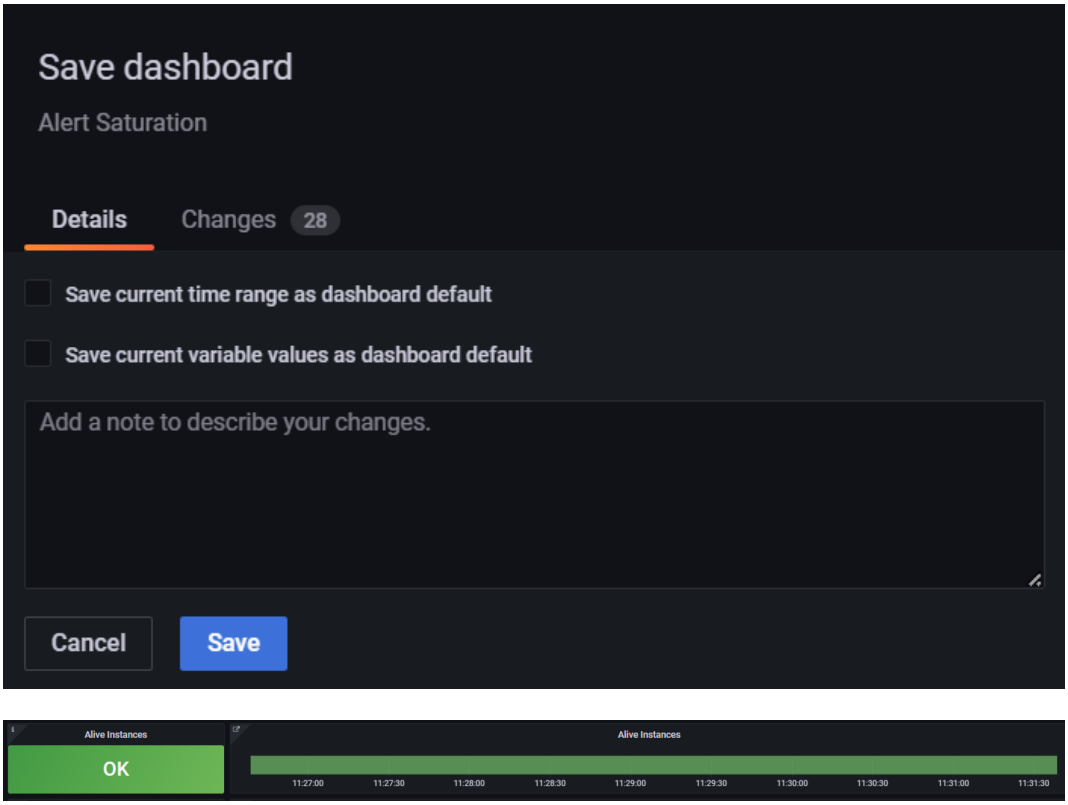
13. Select **Apply** at the top right to apply the changes and return to the **Alert Saturation** dashboard.

14. Select the **Save Dashboard** icon at the top right.



15. When prompted to confirm, select **Save**.

The **Alive Instances** row under the **Reader Cache** section should now be green, indicating it is enabled.



Update the "Entity" Status Dashboard

Locate the "Entity" status dashboard for the metric rule you are enabling. For the metric in our example, `dds_data_reader_cache_alive_instances_errors`, we need to update the **Alert DataReader Status** dashboard.

1. Go to **Dashboards > Browse** to open the list of dashboards.

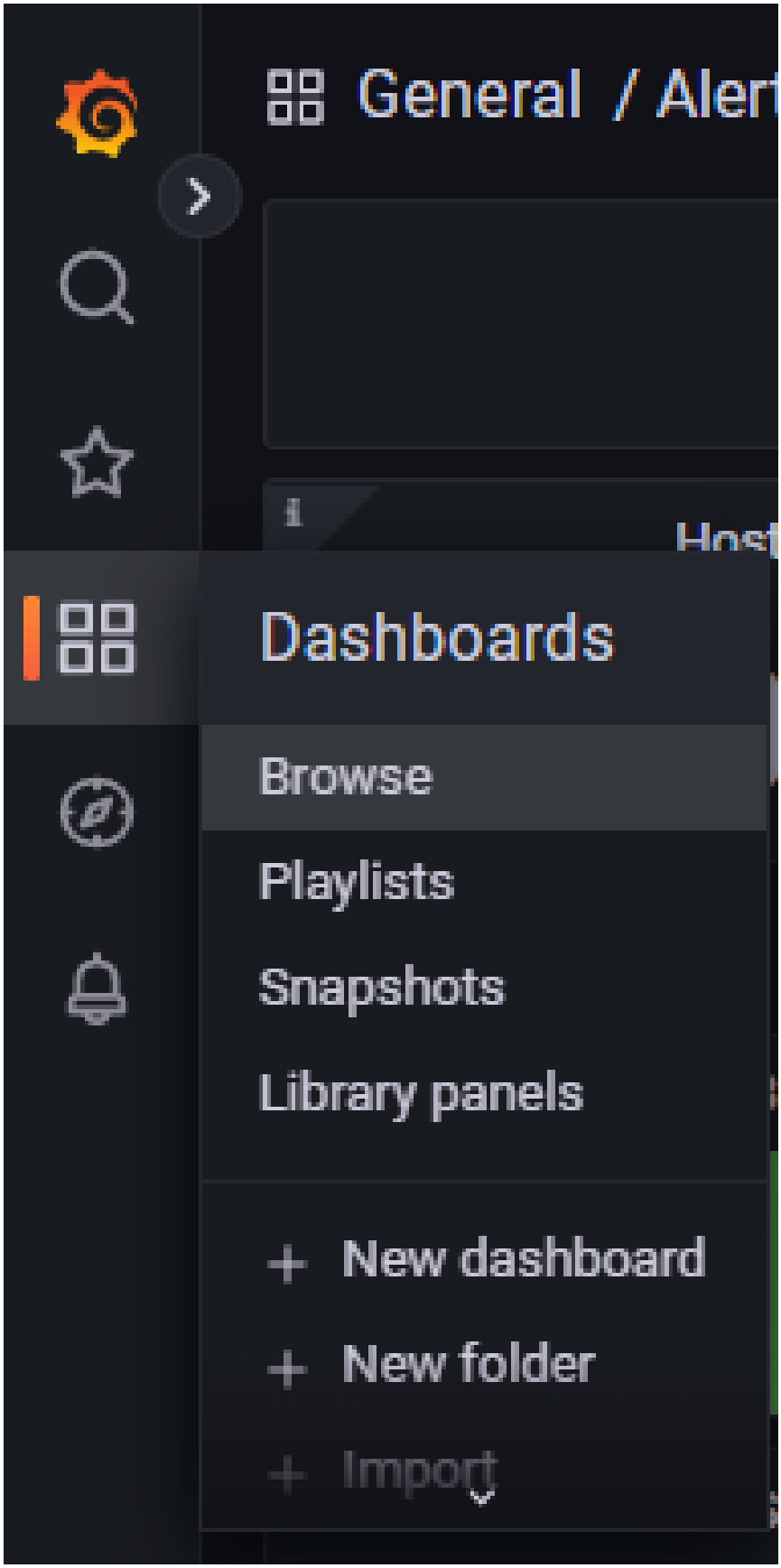
2. Select the **Alert DataReader Status** dashboard from the list.

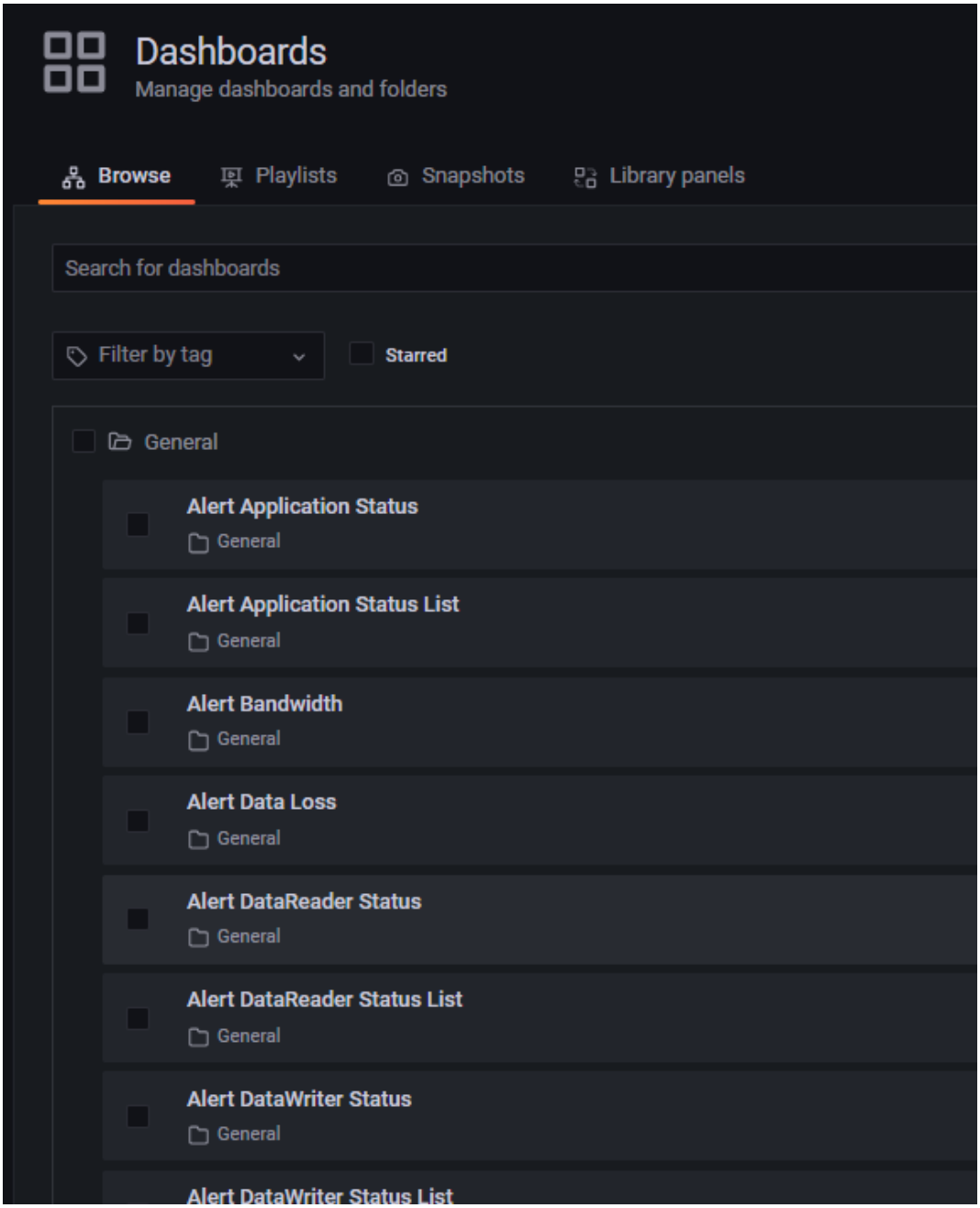
3. Once on the **Alert DataReader Status** dashboard, scroll down to the **Alive Instances** row under the **Saturation/Reader Cache** section.

4. Select **Alive Instances > Edit** from the status indicator panel menu.

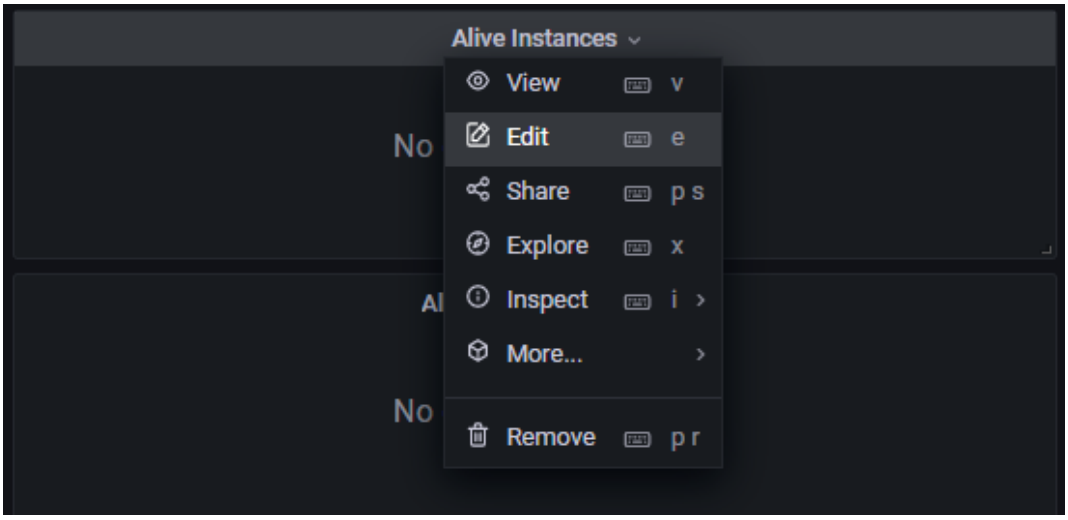
   The query for the panel is shown below.

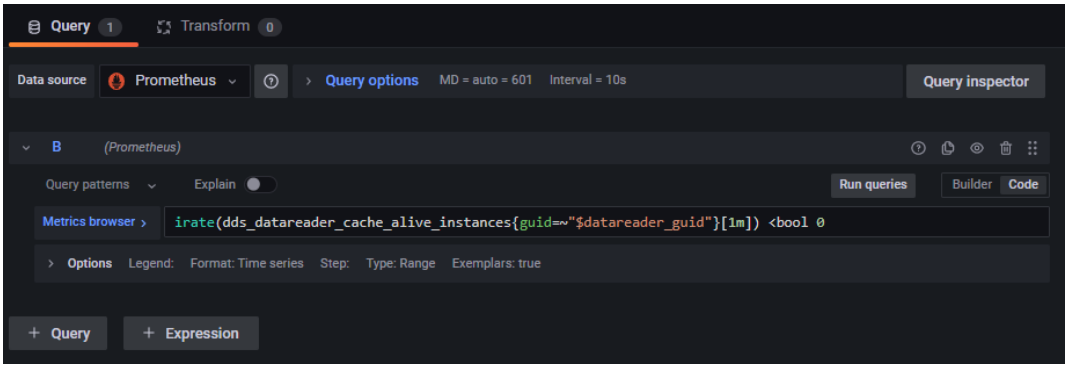
5. Edit the query to match the rule that was created for the `dds_data_reader_cache_alive_instances_errors` metric. In the **Metrics browser** field, remove the irate calculation and set the limit


General / Alert
Dashboards
Browse
Playlists
Snapshots
Library panels
New dashboard
New folder
Import

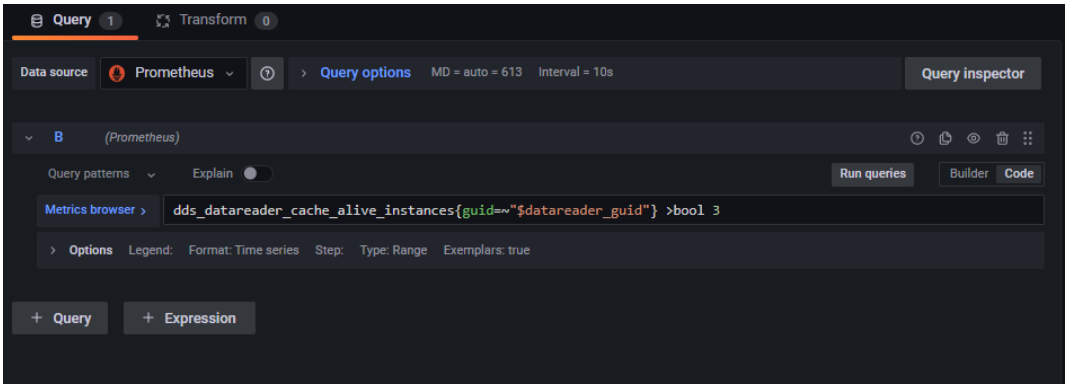

Dashboards
Manage dashboards and folders
Browse
Playlists
Snapshots
Library panels
Search for dashboards
Filter by tag
Starred
General
Alert Application Status
General
Alert Application Status List
General
Alert Bandwidth
General
Alert Data Loss
General
Alert DataReader Status
General
Alert DataReader Status List
General
Alert DataWriter Status
General
Alert DataWriter Status List


Alive Instances
No data in response
No data

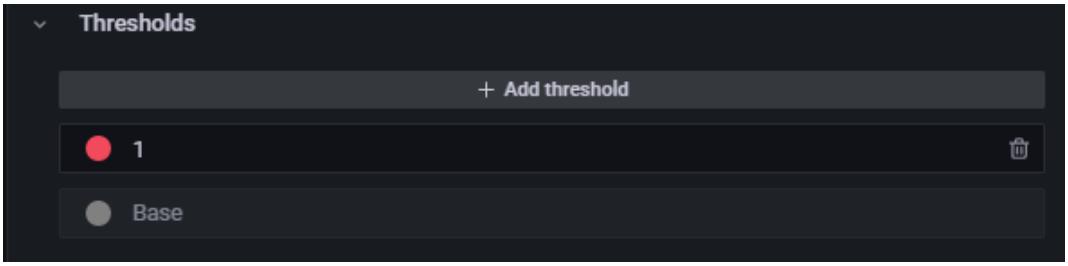

Alive Instances
View v
Edit e
Share p s
Explore x
Inspect i
More...
Remove p r


Query 1
Transform 0
Data source
Prometheus
Query options
MD = auto = 601
Interval = 10s
Query inspector
B
(Prometheus)
Query patterns
Explain
Run queries
Builder
Code
Metrics browser
irate(dds_datareader_cache_alive_instances{guid=~"$datareader_guid"}[1m]) <bool 0
Options
Legend:
Format: Time series
Step:
Type: Range
Exemplars: true
Query
Expression

check to `>bool 3`, as shown below.



6. In the right panel, scroll down to the **Thresholds** section.



7. Click the gray circle next to **Base** to select a new base color for the **Thresholds** panel.

8. Select the large green circle in the panel. The updated **Threshold** base value should change from gray to green.

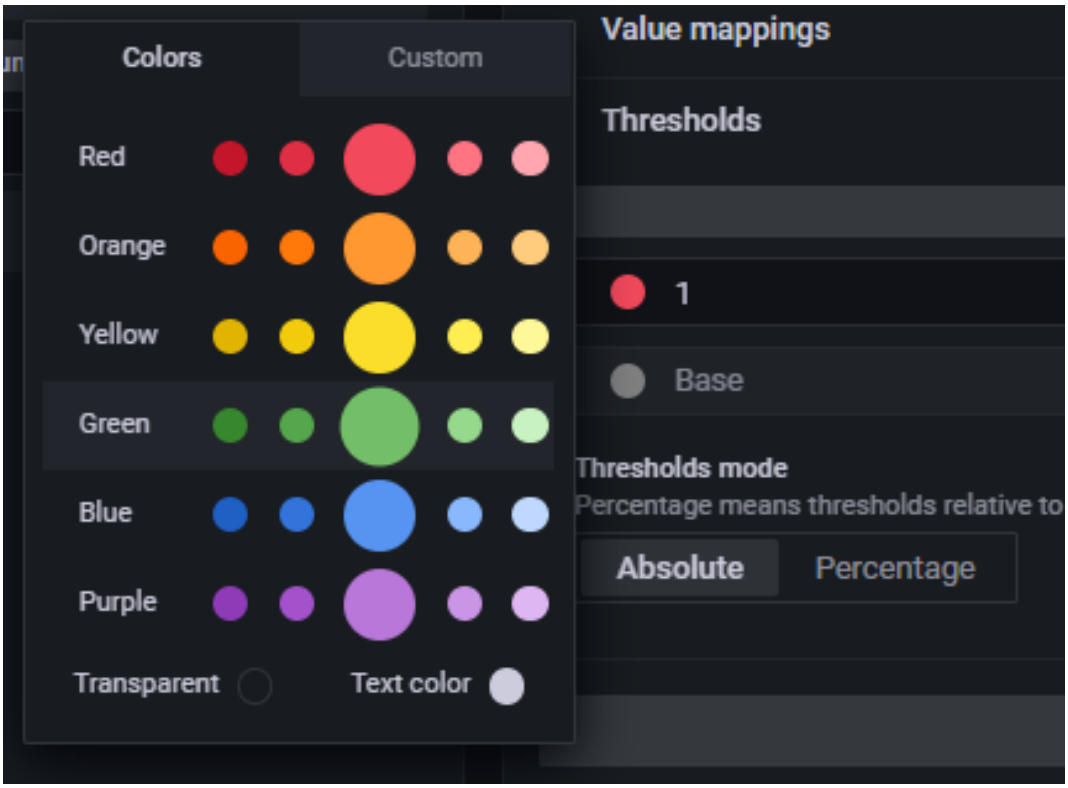
9. Select **Apply** at the top right to apply the change and return to the **Alert DataReader Status** dashboard.

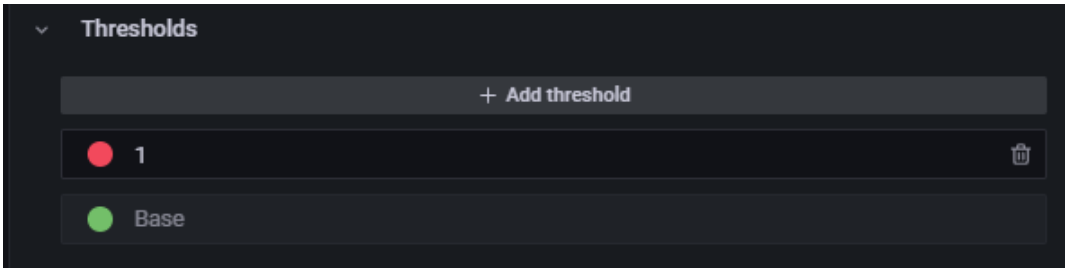
10. Select the **Save Dashboard** icon at the top right.

11. When prompted to confirm, select **Save**.

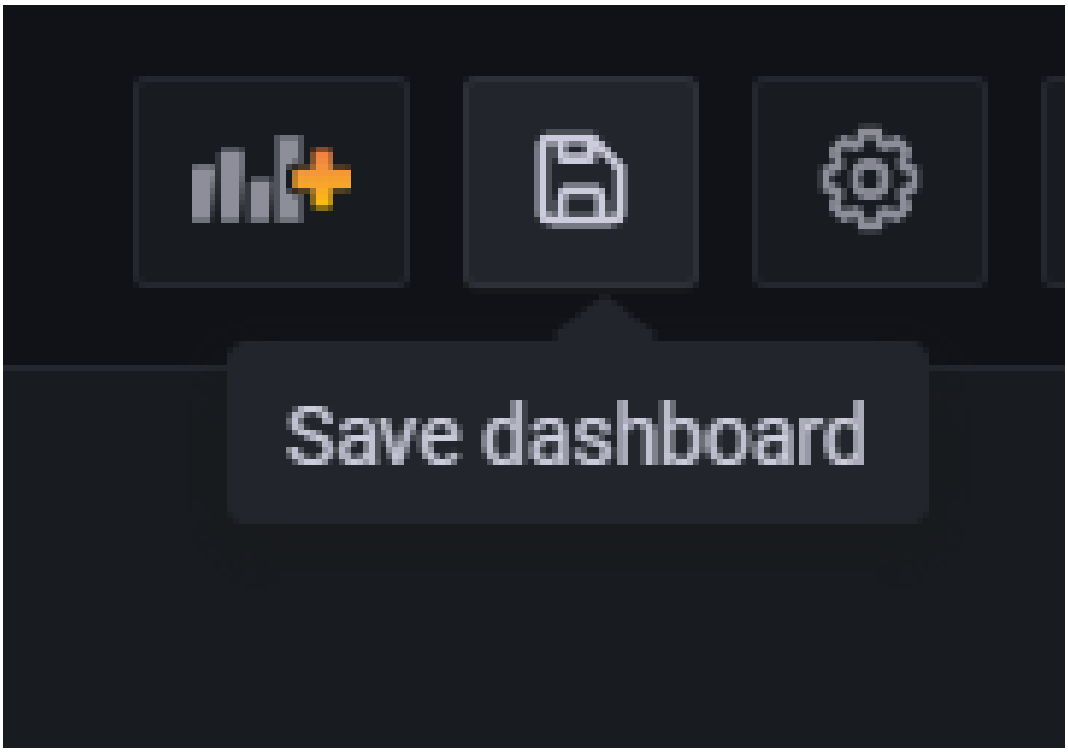
You have now enabled a rule for `dds_data_reader_cache_alive_instances` that detects any *DataReader* that has more than 3 sample instances in its queue with an instance state of ALIVE. The indication of this condition will display on all relevant dashboards.

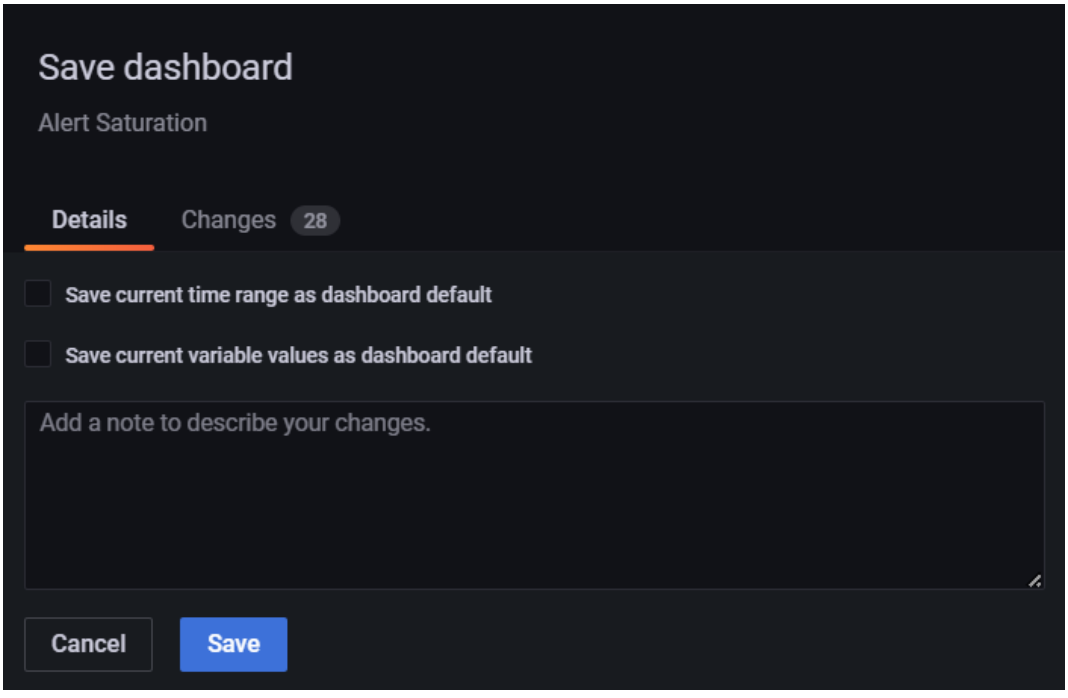
You can test this rule by running the applications as described in section *Start the Applications*. Start any combination of publishing applications with the `-s, --sensor-count` command-line arguments totaling


Colors
Custom
Red
Orange
Yellow
Green
Blue
Purple
Transparent
Text color
Value mappings
Thresholds
1
Base
Thresholds mode
Percentage means thresholds relative to
Absolute
Percentage


Thresholds
Add threshold
1
Base


Discard
Save
Apply


Save dashboard


Save dashboard
Alert Saturation
Details
Changes 28
Save current time range as dashboard default
Save current variable values as dashboard default
Add a note to describe your changes.
Cancel
Save

more than 3. Anytime this condition occurs, you will see this error indicated.

Custom Error Metrics

Table 8.21 shows metrics that are not fully implemented.

Table 8.21: Custom Error Metrics

Metric Name	Description
`dds_custom_excessive_band-width_errors`	Not fully implemented. Not to be modified or used.
`dds_custom_saturation_errors`	Not fully implemented. Not to be modified or used.
`dds_custom_errors`	Not fully implemented. Not to be modified or used.
`dds_custom_delays_errors`	Not fully implemented. Not to be modified or used.
`dds_custom_data_loss_errors`	Not fully implemented. Not to be modified or used.

8.4 Logs

Observability Framework stores the log messages generated by *Connext* applications in third-party backends (for example, Grafana Loki).

When a *Connext* application starts, it may generate log messages before *Monitoring Library 2.0* is loaded and enabled. Any such log messages are not stored by *Observability Framework*.

Each log message is divided into six parts to facilitate analysis, as illustrated in Table 8.22.

Table 8.22: Log message components

Component	Description	Example	Always present
Timestamp	The time the log message was generated.	`[2023-02-02 21:32:38.049836]`	Yes
Syslog Facility	The facility (MIDDLEWARE, SECURITY_EVENT, SERVICE, or USER) and the sequence number of the log message for the facility.	`MIDDLEWARE(sn: 123)`	Yes
Syslog Level	The severity (EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFORMATIONAL, or DEBUG).	`WARNING`	Yes
Activity Context	The DDS context in which the log was generated. See *Activity Context*.	`[010105A0.81551B17.4AA10C9B.80000007{Entity=DR, MessageKind=DATA}|RECEIVE FROM 0101C41F. 40A68B3A.C9442BC5.8000A502]`	No
Message	The message contents.	`PRESCstReaderCollator_isNewerSample:[Topic: 'Temperature', Type: 'Temperature'] Dropped sample from DataWriter (0101C41F. 40A68B3A.C9442BC5.8000A502). The source timestamp (2023-02-02 21:32:40.049765) is greater than the received timestamp (2023-02-02 21:32:38.049820) by more than the source_timestamp_tolerance. The system clocks for the DataWriter and DataReader may not be synchronized.`	Yes

8.4.1 Syslog Levels and Facilities

All the log messages generated by the *Connext* applications are members of a Syslog facility. Syslog facilities are numerical codes that represent the source of the log message, allowing the system or network administrator to categorize and filter log messages based on their origin. These facilities help organize log data, making it easier to manage and analyze.

This release supports the following Syslog facilities:

- MIDDLEWARE (23): Messages generated by the *Connext* middleware.
- SECURITY_EVENT (10): Security-related messages generated by the SECURITY PLUGINS Logging Plugin.
- SERVICE (22): Messages generated by infrastructure services, such as *Routing Service*.
- USER (1): Messages generated by the *Connext* logger APIs that log user messages.

The Syslog facility is always present in the log message, and it is followed by a sequence number that uniquely identifies the log message within the facility. The sequence number is useful for tracking the order of log messages within the same facility and for identifying missing log messages.

The available Syslog levels are: EMERGENCY, CRITICAL, ALERT, ERROR, WARNING, NOTICE, INFORMATIONAL, or DEBUG.

For SECURITY_EVENT and USER facilities, you can get messages with any Syslog level. However, for the other facilities (MIDDLEWARE and SERVICE), the Syslog level of the message is determined by translating the *Connext* builtin logging level associated with the message to the Syslog level.

The mapping between *Connext*'s builtin logging levels (NDDS_Config_LogLevel) and Syslog Levels (NDDS_Config_SysLogLevel) is as follows:

Table 8.23: Log Level Mapping

NDDS_Config_LogLevel	Syslog Level	Minimum Syslog Verbosity that lets the message pass through
NDDS_CONFIG_LOG_LEVEL_FATAL_ERROR	EMERGENCY (1)	ERROR
NDDS_CONFIG_LOG_LEVEL_ERROR	ERROR (15)	ERROR
NDDS_CONFIG_LOG_LEVEL_WARNING	WARNING (31)	WARNING
NDDS_CONFIG_LOG_LEVEL_STATUS_LOCAL	INFORMATIONAL (127)	INFORMATIONAL
NDDS_CONFIG_LOG_LEVEL_STATUS_REMOTE	INFORMATIONAL (127)	INFORMATIONAL
NDDS_CONFIG_LOG_LEVEL_DEBUG	DEBUG (255)	DEBUG

For additional information on Syslog levels and facilities, see Configuring Connext Logging in the *RTI Connext Core Libraries User's Manual*.

8.4.2 Activity Context

The Activity Context provides context for the log message associated with it. The information provided by the Activity Context includes a sequence of activities and resources to which the activities apply. Comparing the Activity Context to traces and spans in OpenTelemetry, you can think of the Activity Context as a trace and the individual activities within the Activity Context as spans within the trace. For additional information on the Activity Context, see Format of Logged Messages in the *RTI Connext Core Libraries User's Manual*.

The Activity Context is available by default in all log messages generated by a *Connext* application. However, you can disable this information by using the APIs in the C language binding: see NDDS_Config_Logger_set_print_format and NDDS_Config_Logger_set_print_format_by_log_level. The same APIs are available in other language bindings.

8.4.3 Log Labels

As with metrics, logs also have an associated set of labels. In Grafana Loki, labels are key/value pairs that act as metadata to describe a log stream. The combination of every label key and value defines a log stream. If just one label value changes, this creates a new stream. Refer to the official Grafana Loki documentation for further details about labels.

Table 8.24 describes the log labels generated by *Connext* applications.

Table 8.24: Log Labels

Loki Label Name	Description
`job`	The source of the log message. This label is useful when multiple system resources share the same Loki instance. For *Connext* applications, the `job` label is always `connext_logger`.
`resource_guid`	A GUID that identifies the specific *Connext* application that generated the log message.
`category`	The logging category of the message. • For the MIDDLEWARE and SERVICE facilities, the category logically classifies messages with the same meaning across facilities. Valid values are `Discovery`, `Security`, or a combination of both separated by a comma (for example, `Discovery,Security`). • For the SECURITY_EVENT facility, the category identifies the security plugin class that generated the message. Valid values are `Auth`, `Access`, `Crypto`, or `Logging`. • The USER facility does not support categories. If the logging category is not available for a log message of any facility, its value is `N/A`. For example, this could happen for MIDDLEWARE log messages that are not related to discovery or security.
`plugin_class`	**Only for messages with SECURITY_EVENT facility.** The standard plugin class name that originated the message, as defined in the OMG 'DDS Security' specification, version 1.2. Here, the `category` label is just a more friendly name for these standard plugin class names. Table 8.25 shows the relationship between the plugin class name and the category.

Table 8.25: Relationship between plugin_class and category

Standard plugin class names	Associated category
DDS:Auth:PKI-DH DDS:Auth:PSK RTI:Auth	Auth
DDS:Access:Permissions DDS:Access:PSK RTI:Access	Access
DDS:Crypto:AES-GCM-GMAC DDS:Crypto:PSK RTI:Crypto	Crypto
DDS:Logging:DDS_LogTopic RTI:Logging	Logging
RTI:Common	N/A

Following are a few examples of how you could use log labels in Grafana Loki:

- Use the `resource_guid` label to query all the log messages generated by a specific *Connext* application.
- Use the `category` label to query all the log messages related to discovery.
- Use the `plugin_class` label to query all the log messages related to authentication logged by the Security Plugins (`RTI:Auth` plugin class).

8.4.4 Collection and Forwarding Verbosity

Monitoring Library 2.0 has two verbosity settings:

- **Collection verbosity** controls the level of log messages an application generates.
- **Forwarding verbosity** controls the level of log messages an application forwards to the *Observability Collector Service* (making the messages visible in the dashboard).

By default, *Monitoring Library 2.0* only forwards error and warning log messages, even if the applications generate more verbose logging. Forwarding messages at a higher verbosity for all applications may saturate the network and the different *Observability Framework* components, such as *Observability Collector Service* and the logging aggregation backend (for example, Grafana Loki).

Both the collection and forwarding verbosity can be set locally by changing the configuration of a *Connext* application or remotely by sending a command.

Changing Verbosity Levels Locally

The collection level can be changed locally using the NDDS_Config_Logger_set_verbosity_by_category and NDDS_Config_Logger_set_verbosity APIs for C or equivalent in other languages. You can also use the `logging` XML tag under `participant_factory_qos`. The collection level can be changed at any time.

The forwarding level can be changed per facility using the `participant_factory_qos.monitoring.telemetry_data.logs.<facility>_forwarding_level` field in the MONITORING QosPolicy (DDS Extension). This QoS policy can be configured programmatically or

via XML. When set programmatically using QoS, the forwarding level must be changed before the *Monitoring Library 2.0* is enabled.

Changing Verbosity Levels Remotely

The collection and forwarding levels can both be changed remotely. There are two methods available to send remote commands:

- Using *Observability Dashboards*, as described in *Change the Application Logging Verbosity*
- Using the *Collector Service* REST API as described in the *Collector Service REST API Reference*.

The *Observability Dashboards* only allow setting the following Syslog levels: ERROR, WARNING, INFORMATIONAL, and DEBUG. The *Collector Service* REST API allows you to set all the Syslog levels.

Chapter 9

Monitoring Library 2.0

RTI Monitoring Library 2.0 is one component of *Connext Observability Framework*. It allows collecting and distributing telemetry data (metrics and logs) associated with the resources created by a DDS application. These observable resources are *DomainParticipants*, *Publishers*, *Subscribers*, *DataWriters*, *DataReaders*, *Topics*, and *applications* (refer to *Resources*). The library also accepts remote commands to change the set of collected and forwarded telemetry data at runtime.

The data collected by *Monitoring Library 2.0* is distributed to an *Observability Collector Service* instance. *Observability Collector Service* forwards the data to other *Observability Collector Service* instances, or stores it to a third-party observability backend such as Prometheus or Grafana Loki.

Monitoring Library 2.0 is a separate library (rtimonitoring2); applications can use it in three different modes:

- **Dynamically loaded**: This is the default mode, which does not require linking with your application. The only requirement is that the rtimonitoring2 shared library must be in the library search path. The library is loaded when the monitoring library is enabled. See *Enabling Monitoring Library 2.0*.
- **Dynamic Linking**: The application is linked with the rtimonitoring2 shared library. When the application runs, the rtimonitoring2 shared library must be in the library search path.
- **Static Linking**: The application is linked with the rtimonitoring2 static library.

The last two modes (dynamic and static linking) are only supported in C and C++ and require calling the API `RTI_Monitoring_initialize` in your application before any other *Connext* APIs. This API is defined in the header file `ndds/monitoring/monitoring_monitoringClass.h`.

Regardless of the mode, to start monitoring your application, enable monitoring as described in *Enabling Monitoring Library 2.0*.

Monitoring Library 2.0 creates a dedicated Participant and uses three different built-in *Topics* to forward telemetry data to *Observability Collector Service*:

- **Periodic**: A best-effort *Topic* for distributing periodic metric data (for example, `dds_data_writer_protocol_pushed_samples_total`). The data is sent periodically, with a configurable period.
- **Event**: A reliable *Topic* for distributing event metric data (for example, `dds_data_writer_liveliness_lost_total`). The data is sent when it changes.

- **Logging**: A reliable *Topic* for distributing log data. The data is sent when a log event occurs.

The library creates one *DomainParticipant* and three *DataWriters*, one for each *Topic* type (periodic, event, and logging). Each *DataWriter* is created within its own *Publisher*.

When *Monitoring Library 2.0* is enabled for an application (`participant_factory_qos.monitoring.enable` is TRUE), every DDS Entity created by the application will be "registered" with the library as an observable resource. *Monitoring Library 2.0* is able to monitor all DDS Entities across multiple *DomainParticipants*. You can select the telemetry data that you want collected and forwarded for an observable resource via an initial configuration, and/or change that data at runtime using remote commands. To set the initial configuration for the collection of metrics in *Monitoring Library 2.0*, see *Setting the Initial Metrics and Log Configuration*. To change metric collection configuration dynamically at runtime, use the REST API as described in *Collector Service REST API Reference*. For an example of how to dynamically change the metric collection configuration using the *Observability Dashboards*, see *Change the Metric Configuration*.

Monitoring Library 2.0 receives remote commands on the built-in ServiceRequest *Topic*. The *Monitoring Library 2.0 DomainParticipant* creates a *DataReader* for this *Topic*.

Note: You are not expected to use the built-in *Topics* directly in your applications. The builtin *Topics* are internal channels between *Monitoring Library 2.0* and *Observability Collector Service*.

To send remote commands, use the REST API (see *Collector Service REST API Reference*). This API sends configuration commands to *Observability Collector Service*, which forwards the commands to the appropriate *Monitoring Library 2.0* instance.

To access the telemetry data, connect to the third-party backends where the data is stored by *Observability Collector Service*. You can visualize the telemetry data through the reference Grafana dashboards (see *Observability Dashboards*).

9.1 Enabling Monitoring Library 2.0

To enable usage of *Monitoring Library 2.0* and to configure its behavior, you have to use the MONITORING QosPolicy (DDS Extension) on the DomainParticipantFactory and set `participant_factory_qos.monitoring.enable` to true. This QoS policy can be configured programmatically or via XML. Next, there is an example that shows how to enable *Monitoring Library 2.0* in your XML configuration file:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <!-- Enable monitoring -->
            <monitoring>
                <enable>true</enable>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

In a typical application, after enabling *Monitoring Library 2.0*, you can also configure which metrics to collect from which resources; the DDS domain ID to use for observability; a name for the application being monitored; and the locator (address), as an initial_peer, of the *Observability Collector Service* instance to which the telemetry data will be forwarded. The following XML example shows how to configure these parameters:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <monitoring>
                <!-- Enable monitoring -->
                <enable>true</enable>
                <!-- Enable all metrics -->
                <telemetry_data>
                    <metrics>
                        <element>
                            <resource_selection>//*</resource_selection>
                            <enabled_metrics_selection>
                                <element>*</element>
                            </enabled_metrics_selection>
                        </element>
                    </metrics>
                </telemetry_data>
                <!-- Change the application name -->
                <application_name>MyApplication</application_name>
                <distribution_settings>
                    <dedicated_participant>
                        <!-- Change the Observability Domain ID -->
                        <domain_id>7</domain_id>
                        <!-- Change the initial peers of the
                             Observability DomainParticipant -->
                        <collector_initial_peers>
                            <element>192.168.1.2</element>
                        </collector_initial_peers>
                    </dedicated_participant>
                </distribution_settings>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

Alternatively, you can use the snippet `BuiltinQosSnippetLib::Feature.Monitoring2.Enable` in your XML configuration file. This snippet enables *Monitoring Library 2.0* and all metrics for collection and forwarding:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <base_name>
            <element>BuiltinQosSnippetLib::Feature.Monitoring2.Enable</
↪element>
        </base_name>
```

(continues on next page)

(continued from previous page)

```
        <participant_factory_qos>
            <monitoring>
                <application_name>MyApplication</application_name>
                <distribution_settings>
                    <dedicated_participant>
                        <!-- Change the Observability Domain ID -->
                        <domain_id>7</domain_id>
                        <!-- Change the initial peers of the
                             Observability DomainParticipant -->
                        <collector_initial_peers>
                            <element>192.168.1.2</element>
                        </collector_initial_peers>
                    </dedicated_participant>
                </distribution_settings>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

The MONITORING QosPolicy (DDS Extension) is changeable at runtime. This means that you can enable or disable *Monitoring Library 2.0* at runtime.

The following sections describe in detail the most common configuration options for *Monitoring Library 2.0*. For a complete list of configuration options, refer to the MONITORING QosPolicy (DDS Extension).

9.2 Setting the Initial Metrics and Log Configuration

By default all metric collection is disabled, and all log forwarding is set to level WARNING. To configure the initial behavior of telemetry data in *Monitoring Library 2.0*, you have to use the MONITORING QosPolicy (DDS Extension) on the DomainParticipantFactory and configure the `participant_factory_qos.monitoring.telemetry_data` structure. This QoS policy can be configured programmatically or via XML. For details on how to set the `resource_selection` fields, see *Resource Pattern Definitions*. For details on how to set the `enabled_metrics_selection` and `disabled_metrics_selection` fields, see *Metric Pattern Definitions*. The following example shows how to configure the initial metric and log collection and forwarding for *Monitoring Library 2.0* in your XML configuration file:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <monitoring>
                <enable>true</enable>
                <telemetry_data>
                    <metrics>
                        <element>
                            <!-- enable all application metrics -->
                            <resource_selection>/applications/*</resource_
↪selection>
```

(continues on next page)

(continued from previous page)

```
                            <enabled_metrics_selection>
                                <element>*</element>
                            </enabled_metrics_selection>
                        </element>
                        <element>
                            <!-- enable all domain_participant metrics -->
                            <resource_selection>//domain_participants/*</
↪resource_selection>
                            <enabled_metrics_selection>
                                <element>*</element>
                            </enabled_metrics_selection>
                        </element>
                        <element>
                            <!-- enable all topic metrics -->
                            <resource_selection>//topics/*</resource_
↪selection>
                            <enabled_metrics_selection>
                                <element>*</element>
                            </enabled_metrics_selection>
                        </element>
                        <element>
                            <!-- enable all data_writer metrics except those␣
↪that end in "_bytes" -->
                            <resource_selection>//data_writers/*</resource_
↪selection>
                            <enabled_metrics_selection>
                                <element>*</element>
                            </enabled_metrics_selection>
                            <disabled_metrics_selection>
                                <element>dds_data_writer_*_bytes</element>
                            </disabled_metrics_selection>
                        </element>
                        <element>
                            <!-- enable all data_reader metrics except those␣
↪related to "protocol" -->
                            <resource_selection>//data_readers/*</resource_
↪selection>
                            <enabled_metrics_selection>
                                <element>*</element>
                            </enabled_metrics_selection>
                            <disabled_metrics_selection>
                                <element>dds_data_reader_protocol_*</element>
                            </disabled_metrics_selection>
                        </element>
                    </metrics>
                    <logs>
                        <!-- set initial MIDDLEWARE forwarding level to ERROR␣
↪-->
                        <middleware_forwarding_level>ERROR</middleware_
↪forwarding_level>
                        <!-- set initial SECURITY_EVENT forwarding level to␣
↪ERROR -->
```

(continues on next page)

(continued from previous page)

```
                        <security_event_forwarding_level>ERROR</security_
↪event_forwarding_level>
                        <!-- set initial SERVICE forwarding level to ERROR -->
                        <service_forwarding_level>ERROR</service_forwarding_
↪level>
                        <!-- set initial USER forwarding level to ERROR -->
                        <user_forwarding_level>ERROR</user_forwarding_level>
                    </logs>
                </telemetry_data>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

9.3 Setting the Application Name

To modify the application name used by *Monitoring Library 2.0*, use the `participant_factory_qos.monitoring.application_name` field. For example:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <monitoring>
                <enable>true</enable>
                <application_name>MyApplication</application_name>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

Assigning an application name is important because it helps identify the resource that represents your *Connext* application. The resource identifier representing the application will be:

`/applications/<application_name>`

This is the resource identifier that will be used to send commands to this application from the *Observability Dashboards*.

The application_name should be unique across the *Connext* system; however, *Monitoring Library 2.0* does not currently enforce uniqueness.

When application_name is not set, *Monitoring Library 2.0* will automatically assign a resource identifier with this format:

`/applications/<host_name:process_id:uuid>`

9.4 Changing the Default Observability Domain ID

To modify the domain used by *Monitoring Library 2.0*'s *DomainParticipant* to connect to *Observability Collector Service*, use the `participant_factory_qos.monitoring.distribution_settings.dedicated_participant.domain_id` field. The default value is 2.

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <monitoring>
                <enable>true</enable>
                <distribution_settings>
                    <dedicated_participant>
                        <domain_id>7</domain_id>
                    </dedicated_participant>
                </distribution_settings>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

9.5 Configuring QoS for Monitoring Library 2.0 Entities

By default, the DDS entities created by *Monitoring Library 2.0* use the built-in profile `BuiltinQosLib::Generic.Monitoring2` (as documented in `<install dir>/resource/resource/xml/BuiltinProfiles.documentationONLY.xml`) to configure their QoS. You can provide a different profile name (`MyObservabilityProfile` in the example below) for each entity by changing the Monitoring QoS Policy. It is recommended that if you provide a different profile name, you create this profile to inherit from the `BuiltinQosLib::Generic.Monitoring2` profile. For example:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyObservabilityProfile" base_name=
↪"BuiltinQosLib::Generic.Monitoring2">
        <domain_participant_qos>
            <participant_name>
                <!-- Change the name of the Observability
                     DomainParticipant
                -->
                <name>Monitoring Participant</name>
            </participant_name>
        </domain_participant_qos>

        <datawriter_qos topic_filter="DCPSEventStatusMonitoring">
            <publication_name>
                <!-- Change the name of the Observability
                     Event DataWriter
                -->
```

(continues on next page)

(continued from previous page)

```
                <name>Monitoring Event DataWriter</name>
            </publication_name>
        </datawriter>

        <datawriter_qos topic_filter="DCPSPeriodicStatusMonitoring">
            <publication_name>
                <!-- Change the name of the Observability
                     Periodic DataWriter
                -->
                <name>Monitoring Periodic DataWriter</name>
            </publication_name>
        </datawriter>

        <datawriter_qos topic_filter="DCPSLoggingStatusMonitoring">
            <publication_name>
                <!-- Change the name of the Observability
                     Logging DataWriter
                -->
                <name>Monitoring Logging DataWriter</name>
            </publication_name>
        </datawriter>
    </qos_profile>

    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <monitoring>
                <enable>true</enable>
                <distribution_settings>
                    <dedicated_participant>
                        <!-- Change the configuration of the
                             Observability DomainParticipant -->
                        <participant_qos_profile_name>
                            MyQosLibrary::MyObservabilityProfile
                        </participant_qos_profile_name>
                    </dedicated_participant>
                    <!-- Change the configuration of the
                         Observability Publishers -->
                    <publisher_qos_profile_name>
                        MyQosLibrary::MyObservabilityProfile
                    </publisher_qos_profile_name>
                    <event_settings>
                        <!-- Change the configuration of the
                             Observability Event DataWriter -->
                        <datawriter_qos_profile_name>
                            MyQosLibrary::MyObservabilityProfile
                        </datawriter_qos_profile_name>
                    </event_settings>
                    <periodic_settings>
                        <!-- Change the configuration of the
                             Observability Periodic DataWriter -->
                        <datawriter_qos_profile_name>
```

(continues on next page)

(continued from previous page)

```
                            MyQosLibrary::MyObservabilityProfile
                        </datawriter_qos_profile_name>
                    </periodic_settings>
                    <logging_settings>
                        <!-- Change the configuration of the
                             Observability Logging DataWriter -->
                        <datawriter_qos_profile_name>
                            MyQosLibrary::MyObservabilityProfile
                        </datawriter_qos_profile_name>
                    </logging_settings>
                </distribution_settings>
            </monitoring>
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

Note: The `BuiltinQosLib::Generic.Monitoring2` profile disables the use of multicast discovery by setting the `<multicast_receive_addresses/>` element for the *Monitoring Library 2.0*'s *DomainParticipant*. Using multicast may lead to multiple *Observability Collector Service* instances receiving the same data. Your applications (that is, each instance of *Monitoring Library 2.0*), should configure the address (initial_peer) of the *Observability Collector Service* that they connect to explicitly as described in *Setting Collector Service Initial Peers*.

9.6 Setting Collector Service Initial Peers

To connect *Monitoring Library 2.0* to *Observability Collector Service*, configure the library with the locator/address of the *Observability Collector Service* via the *Monitoring Library 2.0*'s *DomainParticipant* initial peers list. Set this list (usually just a single locator) using the `participant_factory_qos.monitoring.distribution_settings.dedicated_participant.collector_initial_peers` field in the *Monitoring Library 2.0* XML QoS configuration. The locator/address of the collector service uses the same format as the DISCOVERY QosPolicy (DDS Extension) `initial_peers` field.

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyApplicationProfile" is_default_participant_factory_
↪profile="true">
        <participant_factory_qos>
            <monitoring>
                <enable>true</enable>
                <distribution_settings>
                    <dedicated_participant>
                        <collector_initial_peers>
                            <element>192.168.1.2</element>
                        </collector_initial_peers>
                    </dedicated_participant>
                </distribution_settings>
            </monitoring>
```

(continues on next page)

(continued from previous page)

```
        </participant_factory_qos>
    </qos_profile>
</qos_library>
```

If `collector_initial_peers` is not specified, or if it is explicitly set to an empty list, *Monitoring Library 2.0* will use the value set in the `domain_participant_qos.discovery.initial_peers` of the QoS profile specified by `participant_factory_qos.monitoring.distribution_settings.dedicated_participant.participant_qos_profile_name` as the initial peers for the *Monitoring Library 2.0*'s *DomainParticipant*.

If both values are present, the value in `collector_initial_peers` in the Monitoring QosPolicy will be used instead of the value of `initial_peers` in the Discovery QosPolicy for the *Monitoring Library 2.0*'s *DomainParticipant*.

Chapter 10

Collector Service REST API Reference

Observability Collector Service scalably distributes telemetry data forwarded by *Monitoring Library 2.0* in a *Connext* application and sends it to configurable backends. A key feature of *Observability Collector Service* is remote command forwarding to the *Connext* applications. These commands enable you to control the amount of telemetry data forwarded by *Monitoring Library 2.0* from a *Connext* application.

This REST API reference describes the remote commands provided by *Observability Collector Service*. These commands enable you to:

- get the current logging collection and forwarding verbosity levels for applications
- dynamically change the logging collection and forwarding verbosity levels for applications
- get the current metric collection configuration for observable resources
- dynamically configure the set of metrics collected and forwarded for observable resources

10.1 Definitions

The REST API commands in the following sections share the following common fields:

application

- The **application** field is a Uniform Resource Identifier (URI) that identifies an application in responses to commands that get logging verbosity levels. For details on this Uniform Resource Identifier (URI) see the Application row in Table 8.1.

application_selector

- The **application_selector** field is a resource selector that identifies one or more applications in commands that set logging verbosity levels. For details on specifying a resource selector for **application_selector** see *Resource Pattern Definitions*.

resource_selector

- The **resource_selector** field is a resource selector that identifies one or more observable resources in a command. For details on specifying a **resource_selector** see *Resource Pattern Definitions*.

logging_settings

- The **logging_settings** field is a list of objects that specify the logging level for different facilities within an application. Each object in the list has two properties:
  - **verbosity** levels can be `SILENT`, `DEBUG`, `INFORMATIONAL`, `NOTICE`, `WARNING`, `ERROR`, `CRITICAL`, `ALERT`, `EMERGENCY`.
  - **facility** can be `MIDDLEWARE`, `SERVICE`, `SECURITY_EVENT` or `USER`.

  Note that all **verbosity** levels may not be supported in the *Observability Dashboards*. See *Logs* for details on logging in *Observability Framework*.

metrics

- The **metrics** field is a list of metric names in responses to commands that get metric subscription state. For details on metric names, see *Metrics*.

subscribe_metrics_selector

- The **subscribe_metrics_selector** is a list of metric names to subscribe to. For details on how to specify metric selectors in a **subscribe_metrics_selector** list, see *Metric Pattern Definitions*.

unsubscribe_metrics_selector

- The **unsubscribe_metrics_selector** field is a list of metric names to unsubscribe to. For details on how to specify metric selectors in a **unsubscribe_metrics_selector** list, see *Metric Pattern Definitions*.

10.2 Root endpoint (base URL)

The root endpoint for the *Observability Collector Service* REST API is the URL of the *Observability Collector Service*. It is the base URL for all the commands in this reference. For example: `https://collector_service:19080`.

The hostname and port number of the *Observability Collector Service* service can be configured as follows:

- For pre-packaged installations (see *Docker Compose (Prepackaged)*), the Host and Port information of the *Collector Service* can be configured using the following parameters in the configuration JSON file (see *Configure the JSON File*):
  - collectorConfig.controlPublicHostname
  - collectorConfig.controlPublicPort
- For standalone deployments of *Collector Service* (see *Docker (Separate Deployment)*), the Host and Port information can be configured using two environment variables in the *Collector Service* Docker image (see the Docker Collector Service Repository):
  - OBSERVABILITY_CONTROL_PUBLIC_HOSTNAME
  - OBSERVABILITY_CONTROL_PUBLIC_PORT

In addition, the root endpoint for the *Observability Collector Service* REST API is also part of the label `controllability_url`, which is associated with each application's presence metric (see *Application Metrics*).

This is useful when you have multiple *Observability Collector Service* instances storing data into a metrics backend (for example, Prometheus), and you want to dynamically discover the *Observability Collector Service* instance that is managing a particular application to send remote commands to it.

10.3 API Overview

Resource	Operation	Description
logging	*GET /rti/collector_service/rest1/logging:get_collection_level*	Get the collection logging level.
	GET /rti/collector_service/rest1/logging:get_forwarding_level	Get the forwarding logging level.
	POST /rti/collector_service/rest1/logging:set_collection_level	Set the collection logging level.
	POST /rti/collector_service/rest1/logging:set_forwarding_level	Set the forwarding logging level.
metrics	*GET /rti/collector_service/rest1/metrics:get_subscription_state*	Get the metrics subscription state.
	POST /rti/collector_service/rest1/metrics:set_subscription_state	Set the metrics subscription state.
	POST /rti/collector_service/rest1/metrics:update_subscription_state	Update the metrics subscription state.

10.4 API Reference

`GET /rti/collector_service/rest1/logging:get_collection_level`

This method gets the collection verbosity level of a given application_selector.

Request Headers

- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Query Parameters

- **`application_selector`** –
  - **Description**: The application_selector to get the collection verbosity level.
  - **Type**: `string`
  - **Required**: true
  - **Example**: //app_*

```
GET http://.../rti/collector_service/rest1/
↪logging:get_collection_level?application_selector=/
↪/app_* HTTP/1.1
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/dds-web+json`.

Status Codes

- 200 OK –
  - **Description**: If successful, this method returns the collection verbosity level for the applications that matched the application_selector.

  –Example Response Body:

  ```
  * application/dds-web+json
  ```

  ```
  [
      {
          "application": "/applications/app_1",
          "logging_settings": [
              {
                  "verbosity": "WARNING",
                  "facility": "MIDDLEWARE"
              },
              {
                  "verbosity": "ERROR",
                  "facility": "SERVICE"
              }
          ]
      },
      {
          "application": "/applications/app_2",
          "logging_settings": [
              {
                  "verbosity": "WARNING",
                  "facility": "MIDDLEWARE"
              },
              {
                  "verbosity": "ERROR",
                  "facility": "SERVICE"
              }
          ]
      }
  ]
  ```

- 400 Bad Request –
  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.

- **Example Response Body:**

  * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 404 Not Found –
  - **Description:** The application is not found or does not match any application.
  - **Example Response Body:**

    * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 500 Internal Server Error –
  - **Description:** Generic server error.
  - **Example Response Body:**

    * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

GET /rti/collector_service/rest1/logging:get_forwarding_level

This method gets the forwarding verbosity level of a given application_selector.

Request Headers

- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Query Parameters

- **application_selector** –
  - **Description**: The application_selector to get the forwarding verbosity level.
  - **Type**: string
  - **Required**: true
  - **Example**: app_1

```
GET http://.../rti/collector_service/rest1/
↪logging:get_forwarding_level?application_
↪selector=app_1 HTTP/1.1
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/dds-web+json`.

Status Codes

- 200 OK –
  - **Description**: If successful, this method returns the forwarding verbosity level for the applications that matched the application_selector.

  –Example Response Body:

  * application/dds-web+json

```
[
    {
        "application": "/applications/app_1",
        "logging_settings": [
            {
                "verbosity": "WARNING",
                "facility": "MIDDLEWARE"
            },
            {
                "verbosity": "ERROR",
                "facility": "SERVICE"
            }
        ]
    }
]
```

- 400 Bad Request –
  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.
  - **Example Response Body:**

  * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 404 Not Found –
  - **Description:** The application_selector is not found or does not match any application.

- **Example Response Body:**

  * `application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

- 500 Internal Server Error –
  - **Description:** Generic server error.
  - **Example Response Body:**

    * `application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

`POST /rti/collector_service/rest1/logging:set_collection_level`

This method sets the collection logging level of a given application_selector.

Request Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/json`, `application/dds-web+json`.
- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Example Request Body

- `application/dds-web+json`

```
[
    {
        "application_selector": "//app_1",
        "logging_settings": [
            {
                "verbosity": "WARNING",
                "facility": "MIDDLEWARE"
            },
            {
                "verbosity": "ERROR",
                "facility": "SERVICE"
            }
        ]
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "application_selector": "//GUID(AAAAAAAA.
↪BBBBBBBB.CCCCCCCC.DDDDDDDD)",
        "logging_settings": [
            {
                "verbosity": "WARNING",
                "facility": "MIDDLEWARE"
            },
            {
                "verbosity": "ERROR",
                "facility": "SERVICE"
            }
        ]
    }
]
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/dds-web+json`.

Status Codes

- 204 No Content –
  - **Description**: If successful, this method returns an empty response indicating the collection verbosity level has been set.
- 400 Bad Request –
  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.
  - **Example Response Body:**

    `* application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

- 404 Not Found –
  - **Description:** The application_selector is not found or does not match any application.
  - **Example Response Body:**

    `* application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

- 500 Internal Server Error –
  - **Description:** Generic server error.
  - **Example Response Body:**

    * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

POST /rti/collector_service/rest1/logging:set_forwarding_level

This method sets the Forwarding logging level of a given application_selector.

Request Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: application/json, application/dds-web+json.
- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Example Request Body

- application/dds-web+json

```
[
    {
        "application_selector": "//app_1",
        "logging_settings": [
            {
                "verbosity": "WARNING",
                "facility": "MIDDLEWARE"
            },
            {
                "verbosity": "ERROR",
                "facility": "SERVICE"
            }
        ]
    }
]
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: application/dds-web+json.

Status Codes

- 204 No Content –

  - **Description**: If successful, this method returns an empty response indicating the collection verbosity level has been set.

- 400 Bad Request –
  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.
  - **Example Response Body:**

    * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 404 Not Found –
  - **Description:** The application_selector is not found or does not match any application.
  - **Example Response Body:**

    * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 500 Internal Server Error –
  - **Description:** Generic server error.
  - **Example Response Body:**

    * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

GET /rti/collector_service/rest1/metrics:get_subscription_state

This method gets the metrics subscription state of a given resource_selector.

Request Headers

- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Query Parameters

- **resource_selector** –

- **Description**: The resource_selector to get the metrics subscription state.
- **Type**: `string`
- **Required**: true
- **Example**: //resource_1

```
GET http://.../rti/collector_service/rest1/
↪metrics:get_subscription_state?resource_selector=//
↪resource_1 HTTP/1.1
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/dds-web+json`.

Status Codes

- 200 OK –
  - **Description**: If successful, this method returns the metrics subscription state for the resources that matched the resource_selector. The list of metrics returned for each resource is enabled for collection.

  –Example Response Body:

  `* application/dds-web+json`

```
[
    {
        "resource": "//resource_1",
        "metrics": [
            "dds_data_writer_protocol_received_
↪nack_bytes_total",
            "dds_data_writer_protocol_pulled_
↪samples_total",
            "dds_data_writer_protocol_sent_
↪heartbeats_total"
        ]
    }
]
```

- 400 Bad Request –
  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.

  –Example Response Body:

  `* application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

- 404 Not Found –

  – **Description:** The resource_selector is not found or does not match any resource.

  –Example Response Body:

  * `application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

- 500 Internal Server Error –

  – **Description:** Generic server error.

  –Example Response Body:

  * `application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

`POST /rti/collector_service/rest1/metrics:set_subscription_state`

This method sets the metrics subscription state of a given resource_selector. The metric names provided in "subscribe_metric_selectors" are enabled for collection and distribution for the selected resource. All other metrics on the selected resource are disabled.

If no metrics match the provided selectors, no error is returned.

Request Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/json`, `application/dds-web+json`.
- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Example Request Body

- `application/dds-web+json`

```
[
    {
        "resource_selector": "//GUID(AAAAAAAA.BBBBBBBB.
↪CCCCCCCC.EEEEEEEE)",
        "subscribe_metrics_selectors": [
            "dds_data_writer_protocol_received_nack_
↪bytes_total",
            "dds_data_writer_protocol_pulled_samples_
```

(continues on next page)

(continued from previous page)

```
↪total",
            "dds_data_writer_protocol_sent_heartbeats_
↪total"
        ]
    }
]
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/dds-web+json`.

Status Codes

- 204 No Content –
  - **Description**: If successful, this method returns an empty response indicating the metrics subscription state has been set.
- 400 Bad Request –
  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.

  –Example Response Body:

  `* application/dds-web+json`

  ```
  {
      "code": "...",
      "message": "..."
  }
  ```

- 404 Not Found –
  - **Description:** The resource_selector is not found or does not match any resource.

  –Example Response Body:

  `* application/dds-web+json`

  ```
  {
      "code": "...",
      "message": "..."
  }
  ```

- 500 Internal Server Error –
  - **Description:** Generic server error.

  –Example Response Body:

  `* application/dds-web+json`

```
{
    "code": "...",
    "message": "..."
}
```

POST /rti/collector_service/rest1/metrics:update_subscription_state

This method updates the metrics subscription state of a given resource_selector. The metric names provided in "subscribe_metric_selectors" are enabled for collection and distribution for the selected resource. The metric names provided in "unsubscribe_metric_selectors" are disabled for collection and distribution for the selected resource. The "subscribe_metric_selectors" list is applied before the "unsubscribe_metric_selectors" list. It a metric is in both lists it will be disabled.

If no metrics match the provided selectors, no error is returned.

Request Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/json`, `application/dds-web+json`.
- Authorization – The authorization header is used to authenticate the user. The value of the header is the user's credentials encoded in base64. The format is "Basic <base64 encoded username:password>".

Example Request Body

- `application/dds-web+json`

```
[
    {
        "resource_selector": "//GUID(01234567.89ABCDEF.
↪01234567.89ABCDEF)",
        "subscribe_metrics_selectors": [
            "dds_data_writer_protocol_received_nack_
↪bytes_total",
            "dds_data_writer_protocol_sent_heartbeats_
↪total"
        ],
        "unsubscribe_metrics_selectors": [
            "dds_data_writer_protocol_pulled_samples_
↪total"
        ]
    }
]
```

Response Headers

- Content-Length – Transfer-length of the message-body.
- Content-Type – Valid values: `application/dds-web+json`.

Status Codes

- 204 No Content –

  - **Description**: If successful, this method returns an empty response indicating the metrics subscription state has been set.

- 400 Bad Request –

  - **Description:** In case of an invalid input, this method returns a response body with the error code and a message.

  –Example Response Body:

  * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 404 Not Found –

  - **Description:** The resource_selector is not found or does not match any resource.

  –Example Response Body:

  * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

- 500 Internal Server Error –

  - **Description:** Generic server error.

  –Example Response Body:

  * application/dds-web+json

```
{
    "code": "...",
    "message": "..."
}
```

Chapter 11

Observability Dashboards

Observability Dashboards enable you to visualize the telemetry data collected from *Connext* applications. Using a set of customized Grafana dashboards, this *Observability Framework* component provides a visual reference for the logs and metrics configured for collection in *Monitoring Library 2.0*.

This section describes the custom Grafana dashboards provided in *Observability Dashboards*. All of these dashboards are based on the current time period selected, the last hour by default.

11.1 System Status Dashboards

System Status dashboards group alerts by category to provide an overview of your system's health. These dashboards share common display elements and show related status information.

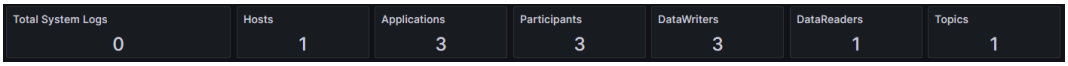
Table 11.1: System Status Dashboards

Dashboard Name	Description
Alert Home	Displays the overall system health. This dashboard displays the high-level status of the aggregated error metrics that make up the alert categories **Bandwidth**, **Saturation**, **Data Loss**, **System Errors**, and **Delays**, as well as the state of system logs.
Alert Bandwidth	Displays the state of the raw error metrics related to **Bandwidth**.
Alert Saturation	Displays the state of the raw error metrics related to **Saturation**.
Alert Data Loss	Displays the state of the raw error metrics related to **Data Loss**.
Alert System Errors	Displays the state of the raw error metrics related to detected DDS **System Errors**.
Alert Delays	Displays the state of the raw error metrics related to **Delays** in data delivery.

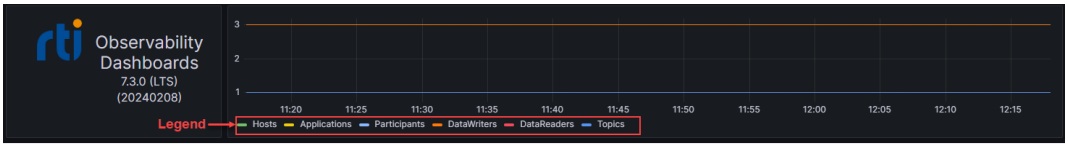
11.1.1 System Status Dashboard Common Elements

All System Status dashboards have two common display elements:

- **Status bar**. At the top of each System Status dashboard, a set of panels displays the number of DDS system logs received and the number of hosts, *Connext* applications, *DomainParticipants*, *DataReaders*, *DataWriters*, and *Topics* reported to currently exist in the system. The number in each panel indicates the number of entities known to exist at the end of the current selected time period. Each panel is a button that allows you to easily navigate to a dashboard that lists all the related entities found in the system. For example, to see a list of all existing *DataWriters*, click the **DataWriters** panel.



- **Time series chart**. Under the **status bar**, a line chart displays the history of each active DDS entity, or *observable resource*. Each resource is represented by a line in the chart showing the history of the creation and destruction of each observable resource. You can select/deselect resources in the legend to view a subset of the resources on the chart.



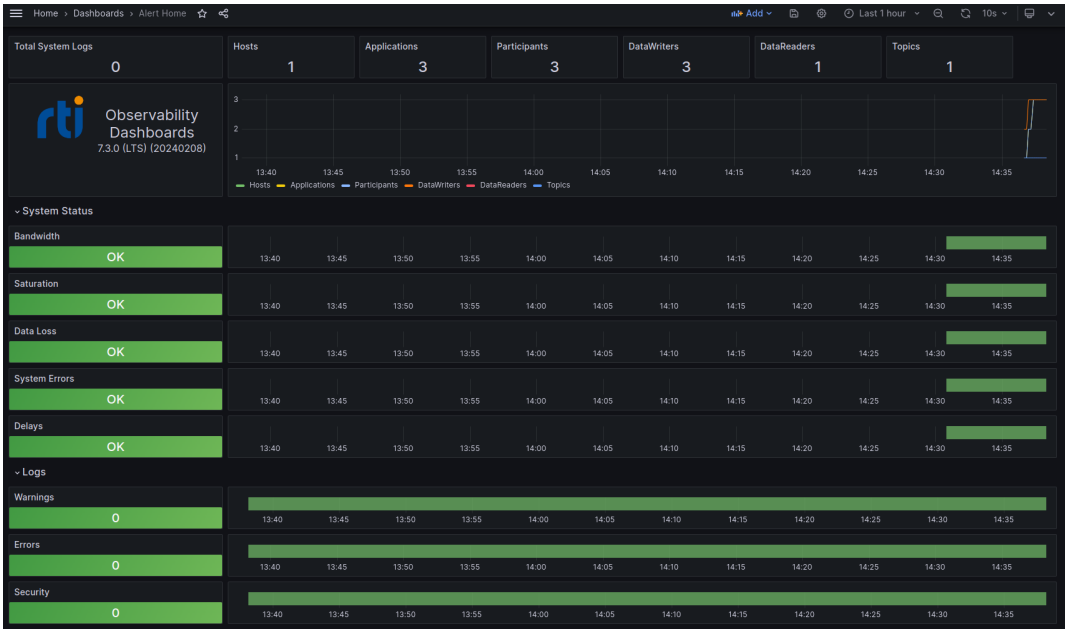
11.1.2 Alert Home Dashboard

The Alert Home dashboard is the home dashboard for visualizing system status. This dashboard shows the current status of each alert category and the system logs. For more information on the alert categories, see *Aggregated Error Metrics*.

Select the Home command at the top left to return to the Alert Home dashboard from any other dashboard.

In addition to the common display elements noted in *System Status Dashboard Common Elements*, the Alert Home dashboard includes:

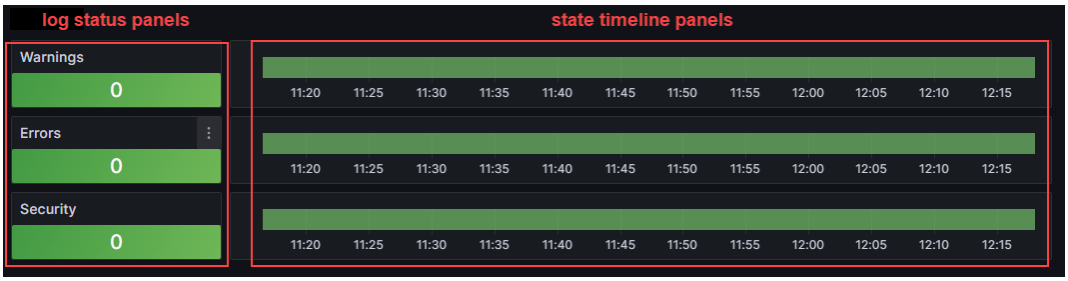
- A row for each alert category that displays the current and historical state for the selected time period. Each System Status row is made up of two panels:
  - A status panel on the left indicates the state (*OK* or *Error*) of the alert category. The panels represent a roll up of all errors that occurred over the selected time period. If a failure condition occurred during the time period, a red **Error** displays in the status panel. If no failures occurred, the panel is green and displays **OK**. For more detail about a category, select the appropriate status panel to open a dashboard for the selected category.
  - A state timeline panel that shows the historical state of the alert category. The state timeline spans the time period selected and indicates any failure conditions on the timeline



in red; otherwise the timeline is green. The timeline is aligned with the time series line chart near the top of the dashboard. This alignment makes it easier to detect a correlation between the creation and destruction of observable resources and possible error conditions.

- A row for each log message type that displays the current and historical state for the selected time period. Each System Status row is made up of two panels:
  - A status panel on the left indicates the current number of logs of the for each log type. The panels represent a roll up of the number of logs that occurred over the selected time period. If logs occurred during the time period, the number of logs displays in red; otherwise, the panel is green. For more details about a log type, select the appropriate status panel to open a dashboard for the selected log type.
  - A state timeline panel that shows the historical state of the log type. The state timeline spans the time period selected and indicates any log occurrences on the timeline in red;

otherwise the timeline is green. The timeline is aligned with the time series chart for observable resources near the top of the dashboard. This alignment makes it easier to detect a correlation between the creation and destruction of observable resources and possible logs.



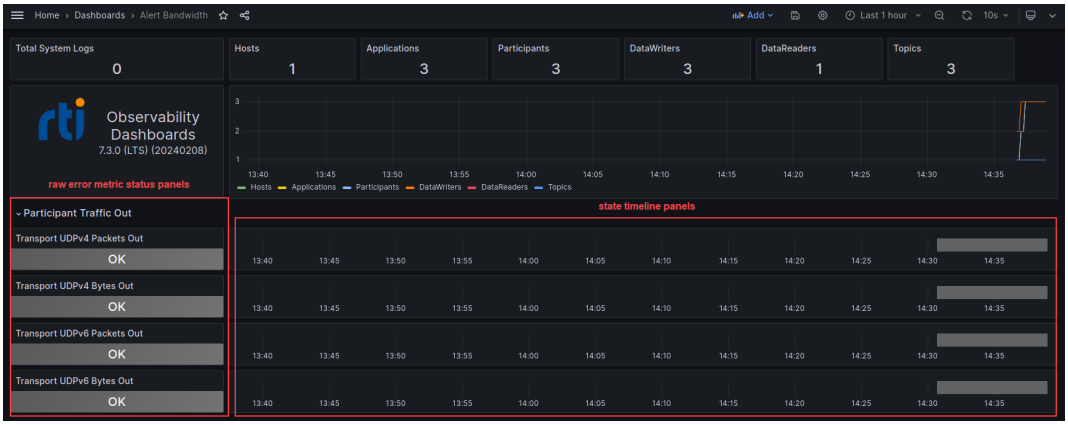
11.1.3 Alert Category Dashboards

Alert Category dashboards provide detailed status information and all associated raw error metrics for a single alert category. These dashboards show the current state of each of the raw error metrics associated with an alert category. For more information, see *Aggregated Error Metrics* and *Raw Error Metrics*.

To access, select the associated alert category status panel on the Alert Home dashboard. For example, to open the Alert Bandwidth dashboard, click the **Bandwidth** status panel.

In addition to the common display elements noted in *System Status Dashboard Common Elements*, the Alert Category dashboards include a row for each of the raw error metrics that make up the selected alert category. The rows are logically grouped by the associated *Connext* entities (application, *DomainParticipant*, *DataReader*, *DataWriter*, and *Topic*). Each row is made up of two panels:

- A status panel on the left indicates the state (*OK* or *Error*) of the raw error metric. The panels represent a roll up of all errors that occurred over the selected time period. If a failure condition occurred, a red **Error** displays in the status panel. If no failures occurred, the panel is green and displays **OK**. For more details about a raw error metric, select the appropriate status panel to open an Entity List dashboard that lists all resources containing the raw error metric.
- A state timeline panel that shows the historical state of the raw error metric. The state timeline spans the time period selected and indicates any failure conditions on the timeline in red; otherwise, the timeline is green. The timeline is aligned with the time series line chart near the top of the dashboard. This alignment makes it easier to detect a correlation between the creation and destruction of observable resources and possible error conditions.



11.2 Entity List Dashboards

Entity List dashboards provide a list of the current observable resources that match the selected entity type.

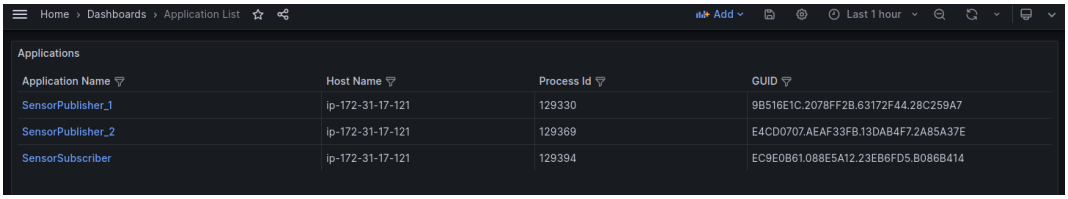
Table 11.2: Entity List Dashboards

Dashboard Name	Description
Host List	Displays the list of unique Hosts (by name) found in the system
Application List	Displays the list of *Connext* applications found in the system
Participant List	Displays the list of *DomainParticipants* found in the system
DataReader List	Displays the list of *DataReaders* found in the system
DataWriter List	Displays the list of *DataWriters* found in the system
Topic List	Displays the list of *Topics* found in the system

To access an Entity List dashboard, select the desired entity count panel on the status bar at the top of any *System Status* or *Log* dashboard.

All Entity List dashboards have the following common display elements:

- A single table panel that lists all observable resources of the selected type. The table columns display associated metadata for each resource.
- A link for each resource that opens the Entity Status dashboard for the selected resource.



11.3 Entity Status List Dashboards

Entity Status List dashboards list the observable resources that contain the selected raw error metric, and the status of that metric for each resource.

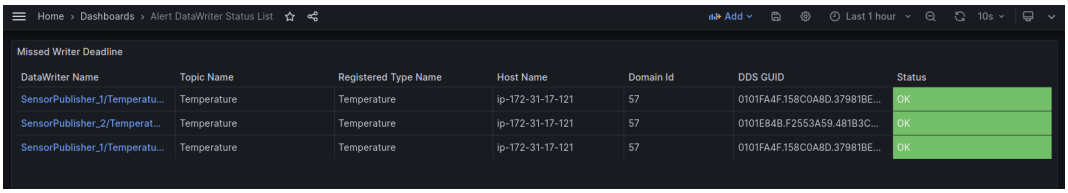
Table 11.3: Entity Status List Dashboards

Dashboard Name	Description
Alert Application Status List	Displays the list of *Connext* applications found in the system, plus the status of the associated raw error metric for the panel that sent you here.
Alert Participant Status List	Displays the list of *DomainParticipants* found in the system, plus the status of the associated raw error metric for the panel that sent you here.
Alert DataReader Status List	Displays the list of *DataReaders* found in the system, plus the status of the associated raw error metric for the panel that sent you here.
Alert DataWriter Status List	Displays the list of *DataWriters* found in the system, plus the status of the associated raw error metric for the panel that sent you here.
Alert Topic Status List	Displays the list of *Topics* found in the system, plus the status of the associated raw error metric for the panel that sent you here.

To access an Entity Status List dashboard, select a raw error metric status panel on any of the *Alert Category* dashboards. For example, select the **Pulled Samples** status panel on the Alert Bandwidth dashboard to open the Alert DataWriter Status List dashboard.

All Entity Status List dashboards have the following common display elements:

- A single table panel that lists all observable resources that contain the selected raw error metric. The table columns display associated metadata for each resource. The Status column indicates the current status (*OK* or *Error*) of each raw error metric.
- A link for each resource that opens the Entity Status dashboard for the selected resource.



11.4 Entity Status Dashboards

Entity Status dashboards provide telemetry metadata and historical charts for a single observable resource.

Table 11.4: Entity Status Dashboards

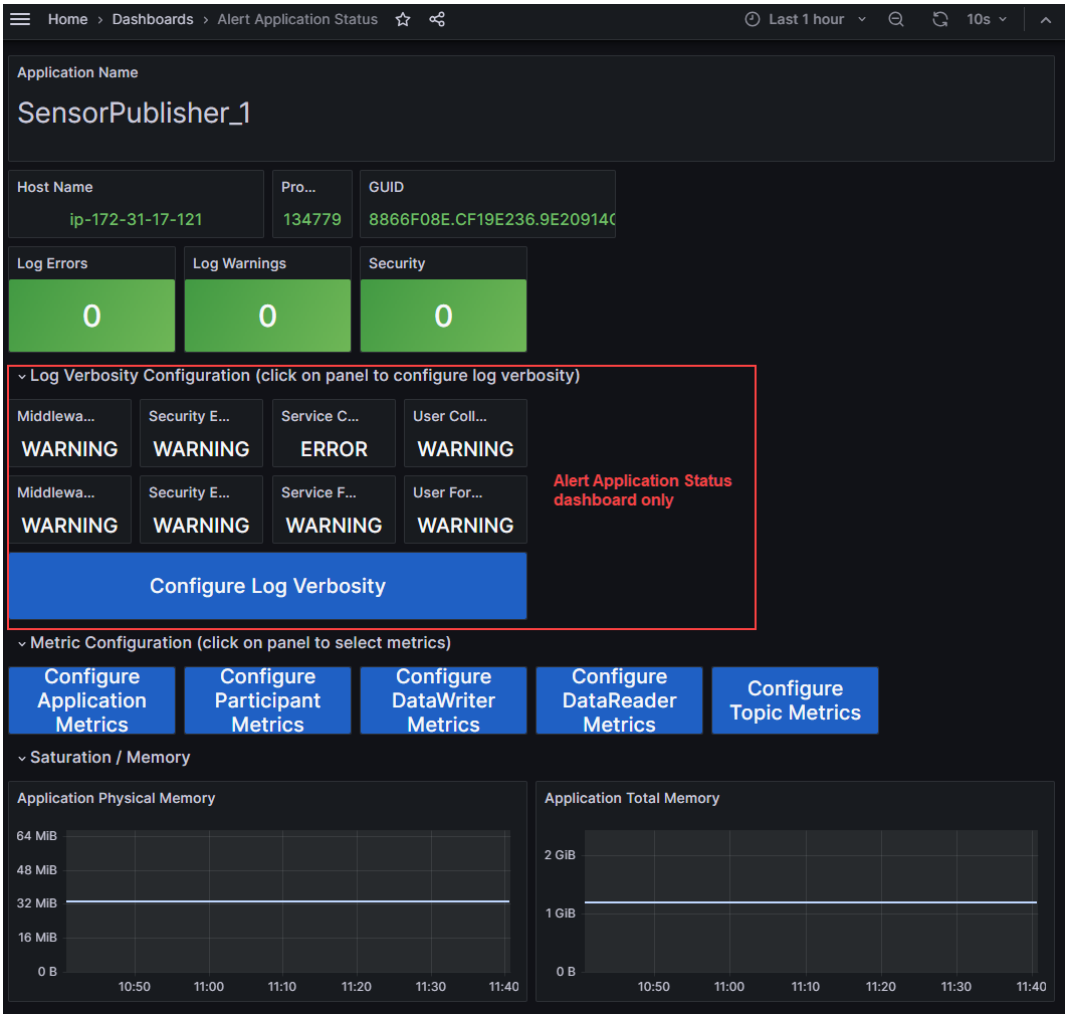
Dashboard Name	Description
Alert Application Status	Displays the metadata for a *Connext* application instance and historical charts of selected raw metrics for this observable resource.
Alert Participant Status	Displays the metadata for a *DomainParticipant* instance and historical charts of selected raw metrics for this observable resource.
Alert DataReader Status	Displays the metadata for a *DataReader* instance and historical charts of selected raw metrics for this observable resource.
Alert DataWriter Status	Displays the metadata for a *DataWriter* instance and historical charts of selected raw metrics for this observable resource.
Alert Topic Status	Displays the metadata for a *Topic* instance and historical charts of selected raw metrics for this observable resource.

To access an Entity Status dashboard, select any of the following:

- A resource link in an *Entity Status* dashboard. For example, select a DataReader Name link on the Alert DataReader Status List dashboard.
- A resource link in an *Entity List* dashboard. For example, select a DataWriter Name link on the DataWriter List dashboard.
- A resource link in the resource name of an *Entity List* dashboard. For example, select the *DomainParticipant* on an Alert DataWriter Status dashboard.
- A log message link in the Log Dashboard (will access the associated Alert Application Status dashboard).

All Entity Status dashboards have the following common display elements:

- A panel indicating the resource name.
- A group of panels displaying metadata associated with the resource.
- Panels providing the number of logs associated with the resource. These panels are buttons that allow you to navigate to the Entity Log dashboard for the current resource.
- One or more metric configuration panels that allow you to navigate to the relevant Metric Control dashboards.
- [Alert Application Status only] Panels that display the current log collection and forwarding verbosity for each log category. See *Logs* for more information on logs.
- [Alert Application Status only] A panel that allows you to navigate to the Log Control dashboard.


Home › Dashboards › Alert Application Status
Last 1 hour
10s
Application Name
SensorPublisher_1
Host Name
ip-172-31-17-121
Pro...
134779
GUID
8866F08E.CF19E236.9E20914(
Log Errors
0
Log Warnings
0
Security
0
Log Verbosity Configuration (click on panel to configure log verbosity)
Middlewa...
WARNING
Security E...
WARNING
Service C...
ERROR
User Coll...
WARNING
Middlewa...
WARNING
Security E...
WARNING
Service F...
WARNING
User For...
WARNING
Alert Application Status dashboard only
Configure Log Verbosity
Metric Configuration (click on panel to select metrics)
Configure Application Metrics
Configure Participant Metrics
Configure DataWriter Metrics
Configure DataReader Metrics
Configure Topic Metrics
Saturation / Memory
Application Physical Memory
64 MiB
48 MiB
32 MiB
16 MiB
0 B
10:50
11:00
11:10
11:20
11:30
11:40
Application Total Memory
2 GiB
1 GiB
0 B
10:50
11:00
11:10
11:20
11:30
11:40

11.5 Log Dashboards

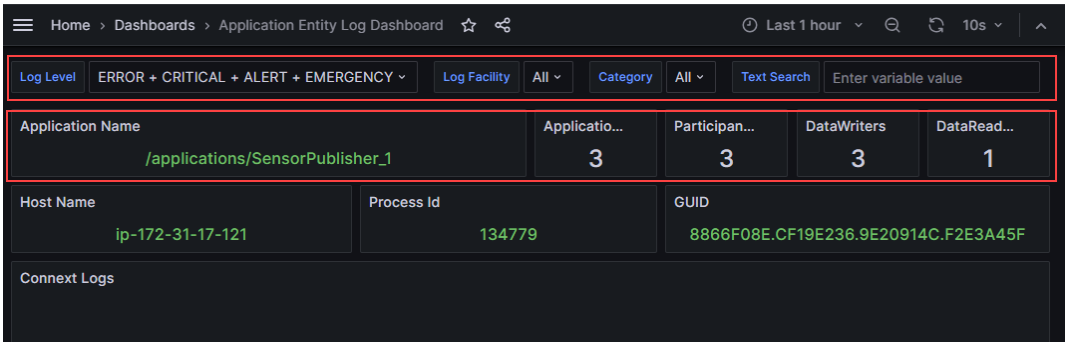
Log dashboards list the logs generated by the system.

Table 11.5: Log Dashboards

Dashboard Name	Description
Log Dashboard	Displays all logs in the system and provides filtering by log level, facility, category, application, and ad hoc text.
Application Entity Log Dashboard	Displays logs for a *Connext* application instance and provides filtering by log level, facility, category, and ad hoc text.
Participant Entity Log Dashboard	Displays logs for a *DomainParticipant* instance and provides filtering by log level, facility, category, and ad hoc text.
DataReader Entity Log Dashboard	Displays logs for a *DataReader* instance and provides filtering by log level, facility, category, and ad hoc text.
DataWriter Entity Log Dashboard	Displays logs for a *DataWriter* instance and provides filtering by log level, facility, category, and ad hoc text.

All Log dashboards have the following common display elements:

- A set of dropdown menus that enable you to select one or more filter criteria. The available filters include **Log Level**, **Log Facility**, **Category**, **Application**, and **Text Search**.
- A set of entity count panels that provide the current number of *Connext* applications, *DomainParticipants*, *DataReaders*, and *DataWriters*. Each panel displays the number of active entities at the end of the selected time period. Click any entity count panel to open an *Entity List* dashboard.



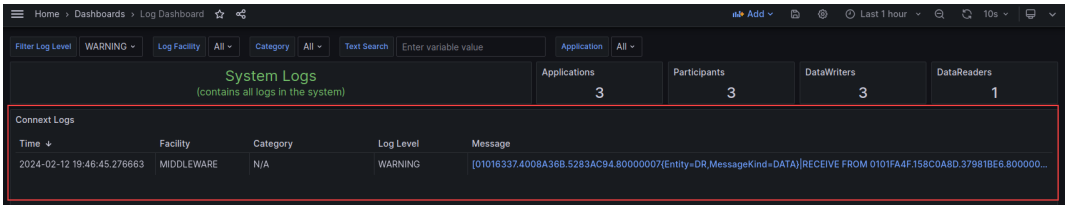
11.5.1 Log Dashboard

The Log Dashboard displays all of the log messages generated by the system.

To access the Log dashboard, select any of the log status panels (**Total System Logs**, **Warnings**, **Errors**, or **Security**) on the *Alert Home* dashboard. Log dashboard data is filtered based on how you accessed it. For example, select the **Warnings** status panel on the Alert Home dashboard to open the Log dashboard with the WARNING log level filter in place.

The Log Dashboard has the following display elements:

- A panel that displays the list of logs in the system that pass the current filter criteria.
- For each log line, several columns of associated data including Time, Facility, Category, Log Level, Plug In Class (for Security logs), and the Message. The message column is a link that navigates to the Alert Application Status dashboard for the *Connext* application that generated the message.

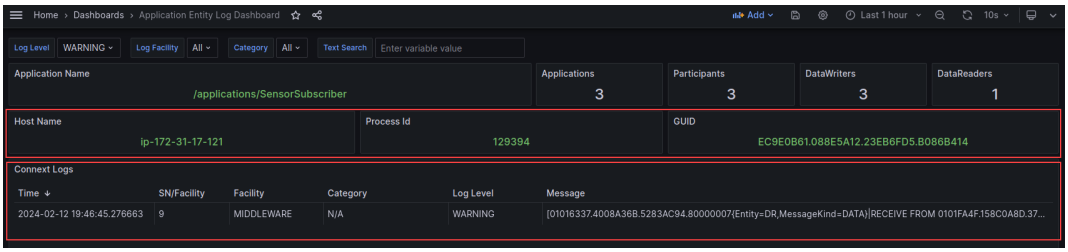


11.5.2 Entity Log Dashboards

Entity Log dashboards display all log messages generated by the system for a specific resource. To access an Entity Log Dashboard select any of the log panels on an *Entity Status* dashboard.

All Entity Log dashboards have the following common display elements:

- A group of panels displaying metadata associated with the resource.
- A panel that lists all logs in the system that pass the current filter criteria. Each log line has several columns of associated data including Time, Facility, Category, Log Level, Plug In Class (for Security logs), and the Message.



11.6 Control Dashboards

Control dashboards enable you to dynamically configure the amount of telemetry data collected and forwarded.

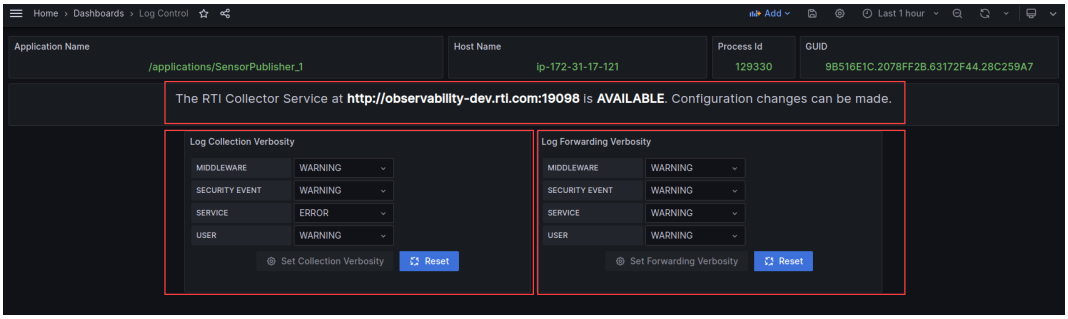
11.6.1 Log Control Dashboard

The Log Control dashboard enables you to dynamically configure the log collection and forwarding verbosity for a single *Connext* application instance. Configuring the verbosity levels of an application affects the number of logs generated and/or forwarded by that application.

To access the Log Control dashboard, select the **Configure Log Verbosity** panel on any *Alert Application Status* dashboard.

The Log Control dashboard has the following display elements:

- A status bar indicating the URL of the *Observability Collector Service* Control Server and the status (AVAILABLE/NOT AVAILABLE).
  - **AVAILABLE**. The dashboard is connected to the *Observability Collector Service* Control Server and can send metric configuration commands.
  - **NOT AVAILABLE**. The dashboard is NOT connected to the *Observability Collector Service* Control Server.
- A panel that allows you to change the log collection verbosity for each category. The collection verbosity affects the logs that the application generates and passes to the *Monitoring Library 2.0*.
- A panel that allows you to change the log forwarding verbosity for each category. The forwarding verbosity controls what logs are forwarded by the *Monitoring Library 2.0* to the *Observability Collector Service* (and subsequently stored in a backend and viewable in a dashboard).



11.6.2 Metric Control Dashboards

Metric Control dashboards enable you to configure the collection and forwarding of metric data.

Single Entity Metric Control Dashboards

Single Metric Control dashboards enable you to configure the collection and forwarding of metric data for a single observable resource. See *Change the Metric Configuration* for a usage example.

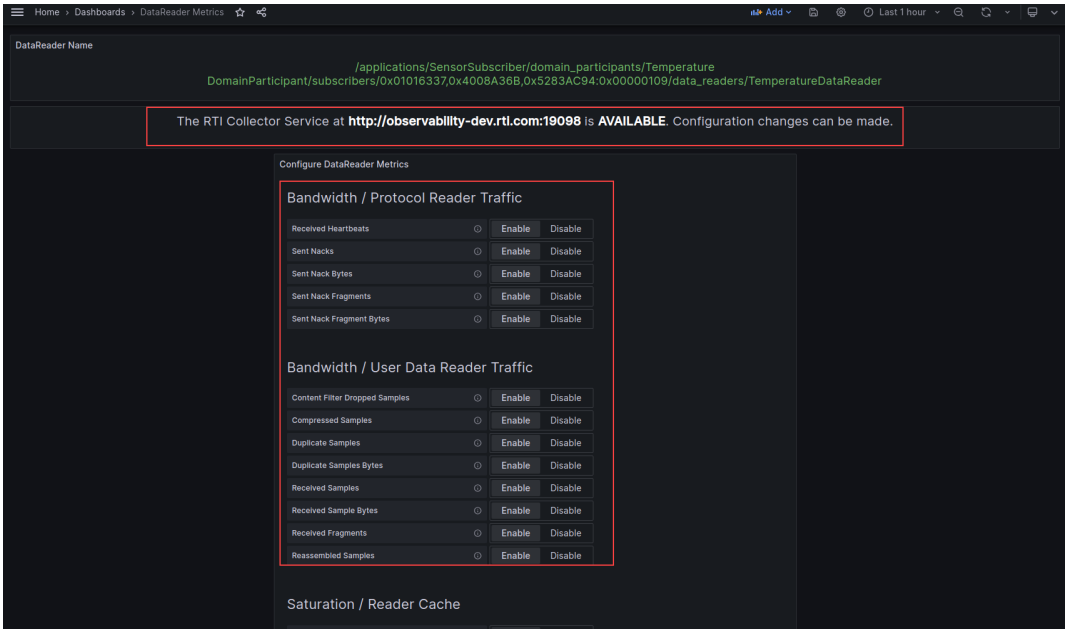
Table 11.6: Single Entity Metric Control Dashboards

Dashboard Name	Description
Application Metrics	Enables you to dynamically change the metric collection and forwarding configuration for a single *Connext* application instance.
Participant Metrics	Enables you to dynamically change the metric collection and forwarding configuration for a single *DomainParticipant* instance.
DataReader Metrics	Enables you to dynamically change the metric collection and forwarding configuration for a single *DataReader* instance.
DataWriter Metrics	Enables you to dynamically change the metric collection and forwarding configuration for a single *DataWriter* instance.
Topic Metrics	Enables you to dynamically change the metric collection and forwarding configuration for a single *Topic* instance.

To access a Single Entity Metric Control dashboard, select the **Configure [Entity] Metrics** panel on any *Entity Status* dashboard that matches the current entity type. For example, select **Configure Participant Metrics** on an Alert Participant Status dashboard.

All Single Entity Metric Control dashboards have the following common display elements:

- A status bar indicating the URL of the *Observability Collector Service* Control Server and the status (AVAILABLE/NOT AVAILABLE).
  - **AVAILABLE**. The dashboard is connected to the *Observability Collector Service* Control Server and can send metric configuration commands.
  - **NOT AVAILABLE**. The dashboard is NOT connected to the *Observability Collector Service* Control Server.
- A panel that shows the collection state (Enable/Disable) for each resource metric. Clicking Enable or Disable sends that command to the *Observability Collector Service* to enable or disable the collection state.



Multiple Entity Metric Control Dashboards

Multiple Metric Control dashboards enable you to configure the collection and forwarding of metric data for all observable resources contained by another resource (for example, all *DataReaders* of a *DomainParticipant*). See *Change the Metric Configuration* for a usage example.

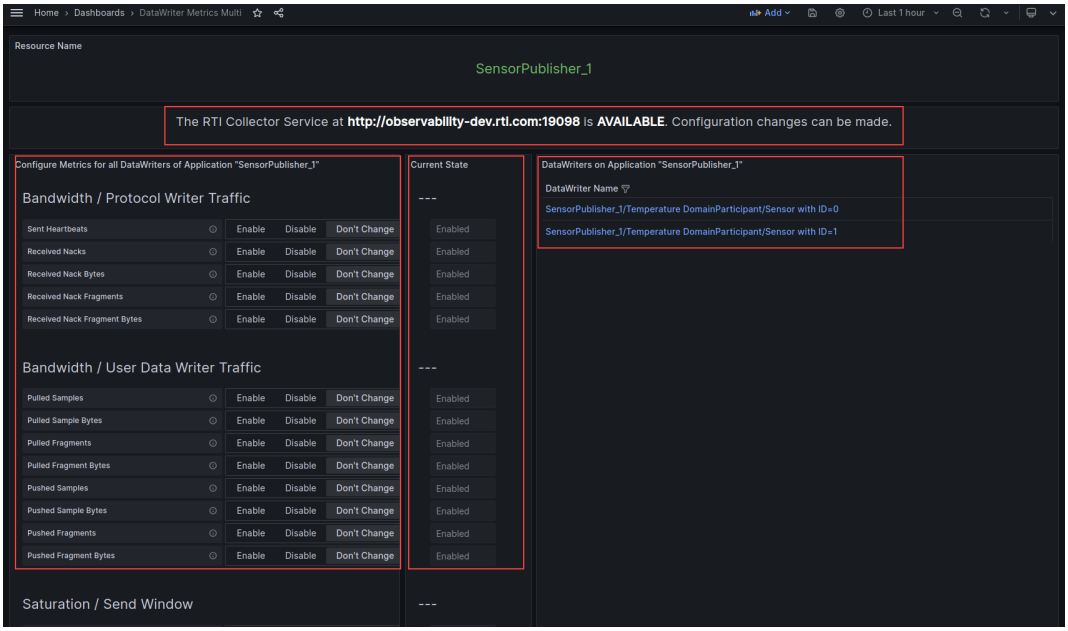
Table 11.7: Multiple Entity Metric Control Dashboards

Dashboard Name	Description
Participant Metrics Multi	Enables you to dynamically change the metric collection and forwarding configuration for all *DomainParticipant* instances of a *Connext* application.
DataReader Metrics Multi	Enables you to dynamically change the metric collection and forwarding configuration for all *DataReader* instances of the current resource. A resource can be a *Connext* application or *DomainParticipant*.
DataWriter Metrics Multi	Enables you to dynamically change the metric collection and forwarding configuration for all *DataWriter* instances of the current resource. A resource can be a *Connext* application or *DomainParticipant*.
Topic Metrics Multi	Enables you to dynamically change the metric collection and forwarding configuration for all *Topic* instances of the current resource. A resource can be a *Connext* application or *DomainParticipant*.

To access a Multiple Entity Metric Control dashboard, select the appropriate **Configure [Entity] Metrics** panel on any *Entity Status* dashboard that is a hierarchical parent of entities. For example, select **Configure DataWriter Metrics** on either an Alert Application Status dashboard or an Alert Participant Status dashboard.

All Multiple Entity Metric Control dashboards have the following display elements:

- A status bar indicating the URL of the *Observability Collector Service* Control Server and the status (AVAILABLE/NOT AVAILABLE).
  - **AVAILABLE**. The dashboard is connected to the *Observability Collector Service* Control Server and can send metric configuration commands.
  - **NOT AVAILABLE**. The dashboard is NOT connected to the *Observability Collector Service* Control Server.
- A panel that allows you to explicitly Enable or Disable the collection and forwarding of the selected metric. This selection affects all observable resources of the current resource regardless of the current collection state.
- A panel that shows the collection state (Enabled/Disabled/Partial) for each metric of the resource type.
  - **Enabled**. The metric is enabled for all resources in the container resource.
  - **Disabled**. The metric is disabled for all resources in the container resource.
  - **Partial**. Some contained resources have the metric enabled and some have it disabled.
- A panel that lists all observable resources that will be affected by configuration changes. Each entry in the list links to the Single Entity Metric Control dashboard for the selected resource.


Home › Dashboards › DataWriter Metrics Multi
Add
Last 1 hour
Resource Name
SensorPublisher_1
The RTI Collector Service at http://observability-dev.rti.com:19098 is AVAILABLE. Configuration changes can be made.
Configure Metrics for all DataWriters of Application "SensorPublisher_1"
Current State
DataWriters on Application "SensorPublisher_1"
DataWriter Name
SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0
SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=1
Bandwidth / Protocol Writer Traffic
Sent Heartbeats
Received Nacks
Received Nack Bytes
Received Nack Fragments
Received Nack Fragment Bytes
Enable
Disable
Don't Change
Enabled
Bandwidth / User Data Writer Traffic
Pulled Samples
Pulled Sample Bytes
Pulled Fragments
Pulled Fragment Bytes
Pushed Samples
Pushed Sample Bytes
Pushed Fragments
Pushed Fragment Bytes
Saturation / Send Window

Chapter 12

Troubleshooting Observability Framework

This section provides solutions for issues you may run into while evaluating *Observability Framework*.

12.1 Docker Container[s] Failed to Start

The Docker containers used by *Observability Framework* can fail to start for a variety of reasons. Two common reasons for this are port conflicts or illegal file permissions. To verify the state of these Docker containers, run the Docker command `docker ps -a`.

An example that shows all Docker containers used by *Observability Framework* have successfully started is shown below.

```
CONTAINER ID   IMAGE                                          COMMAND                  ␣
↪ CREATED         STATUS         NAMES
6651d7ed9810   prom/prometheus:v2.37.5                        "/bin/prometheus --c…"␣
↪ 5 minutes ago   Up 5 minutes   prometheus_observability
25050d16b1b5   grafana/grafana-enterprise:9.2.1-ubuntu   "/run.sh"                ␣
↪ 5 minutes ago   Up 5 minutes   grafana_observability
08611ea9b255   rticom/collector-service:<version>         "/rti_connext_dds-7.…"␣
↪ 5 minutes ago   Up 5 minutes   collector_service_observability
55568de5120f   grafana/loki:2.7.0                         "/usr/bin/loki --con…"␣
↪ 5 minutes ago   Up 5 minutes   loki_observability
```

An example that shows a container that has failed to start is shown below. The failure is indicated by the `Restarting` note in the STATUS column. In this example, the `prometheus-observability` container failed to start and repeatedly tried to restart.

```
CONTAINER ID   IMAGE                                          COMMAND                  ␣
↪ CREATED         STATUS                              NAMES
08f75e0fadb2   prom/prometheus:v2.37.5                        "/bin/prometheus --c…"␣
↪ 5 minutes ago   Restarting (1) 27 seconds ago   prometheus_observability
9a3964b561ec   grafana/loki:2.7.0                             "/usr/bin/loki --con…"␣
↪ 5 minutes ago   Up 5 minutes                        loki_observability
b6a6ffa201f3   rticom/collector-service:<version>             "/rti_connext_dds-7.…"␣
↪ 5 minutes ago   Up 5 minutes                        collector_service_
```

(continues on next page)

(continued from previous page)

```
↪observability
26658f76cfdc  grafana/grafana-enterprise:9.2.1-ubuntu  "/run.sh"                ␣
↪ 5 minutes ago  Up 5 minutes                    grafana_observability
```

To determine why a container failed, examine its log file. To generate the log, run the Docker command `docker logs <container-name>` where `<container_name>` is specified in the NAMES column, as shown above.

12.1.1 Check for Port Conflicts

Run `docker logs <container-name>` to generate the logs for the failed container, then look for a port conflict error. An example of a Prometheus port conflict is shown below.

```
ts=2023-03-14T13:12:29.275Z caller=web.go:553 level=info component=web msg=
↪"Start listening for connections" address=0.0.0.0:9090
ts=2023-03-14T13:12:29.275Z caller=main.go:786 level=error msg="Unable to␣
↪start web listener" err="listen tcp 0.0.0.0:9090: bind: address already in␣
↪use"
```

If you discover port conflicts, perform the following steps to resolve the issue.

1. Remove the existing Observability Workspace. See *Removing the Docker Workspace for Observability Framework* for details on how to remove the workspace.
2. Update the JSON configuration files to configure ports. See *Configuring the Docker Workspace for Observability Framework* for details on how to update the port configuration for the failed container.
3. Run `<installdir>/bin/rtiobservability -c <JSON config>` to recreate the Obervability Workspace with the new port configuration.
4. Run `<installdir>/bin/rtiobservability -i` to create and run the Docker containers with the new port configuration.

12.1.2 Check that You Have the Correct File Permissions

Run `docker logs <container-name>` to generate the logs for the failed container, then look for a file permissions error. An example of a file permissions problem is shown below.

```
ts=2023-03-14T22:21:47.666Z caller=main.go:450 level=error msg="Error loading␣
↪config (--config.file=/etc/prometheus/prometheus.yml)" file=/etc/prometheus/
↪prometheus.yml err="open /etc/prometheus/prometheus.yml: permission denied"
```

Docker containers for *Observability Framework* require the `other` permission to be "read/access" for directories, "read" for files. To resolve a file permission problem, ensure Linux permissions of at least:

- 755 (rwxr-xr-x) for directories
- 444 (r–r–r–) for files

12.2 No Data in Dashboards

Before proceeding, make sure all Docker containers for *Observability Framework* are running properly (see *Docker Container[s] Failed to Start*) and that you have started your applications with *Monitoring Library 2.0* enabled (see *Monitoring Library 2.0*).

12.2.1 Check that Collector Service has Discovered Your Applications

1. Run one or more applications configured with *Monitoring Library 2.0*.
2. Open a browser to `<servername>:<port>/metrics`, where `servername` is the server where *Observability Collector Service* is installed and `port` is the port number for the *Observability Collector Service* Prometheus Client port (19090 is the default).
3. Verify that you have data for the `dds_domain_participant_presence` metric for your application(s) as highlighted below.

```
# HELP exposer_transferred_bytes_total Transferred bytes to metrics services
# TYPE exposer_transferred_bytes_total counter
exposer_transferred_bytes_total 65289
# HELP exposer_scrapes_total Number of times metrics were scraped
# TYPE exposer_scrapes_total counter
exposer_scrapes_total 60
# HELP exposer_request_latencies Latencies of serving scrape requests, in␣
↪microseconds
# TYPE exposer_request_latencies summary
exposer_request_latencies_count 60
exposer_request_latencies_sum 25681
exposer_request_latencies{quantile="0.5"} 316
exposer_request_latencies{quantile="0.9"} 522
exposer_request_latencies{quantile="0.99"} 728
# TYPE dds_domain_participant_presence gauge
dds_domain_participant_presence{guid="AC462E9B.9BB5237C.DBB61B21.80B55CD8",
↪owner_guid="F8824B73.10EBC319.4ACD1E47.9ECB3033",dds_guid="010130C4.
↪C84EFC6D.973810C6.000001C1",domain_id="57",platform="x64Linux4gcc7.3.0",
↪product_version="<version>",name="/applications/SensorSubscriber/domain_
↪participants/Temperature DomainParticipant",hostname="presanella",process_
↪id="458392"} 1 1678836129957
dds_domain_participant_presence{guid="291C3B07.34755D99.608E7BF3.1F6546D9",
↪owner_guid="566D1E8D.5D7CBFD4.DD65CC20.C33D56E9",dds_guid="0101416F.
↪425D03B2.8AC75FC8.000001C1",domain_id="57",platform="x64Linux4gcc7.3.0",
↪product_version="<version>",name="/applications/SensorPublisher_2/domain_
↪participants/Temperature DomainParticipant",hostname="presanella",process_
↪id="458369"} 1 1678836129957
dds_domain_participant_presence{guid="1D5929EC.4FB3CAE4.300F0DB0.C553A54F",
↪owner_guid="D2FD6E87.D8C03AAA.EABFB1F8.E941495B",dds_guid="0101FBDA.
↪551F142B.619EE527.000001C1",domain_id="57",platform="x64Linux4gcc7.3.0",
↪product_version="<version>",name="/applications/SensorPublisher_1/domain_
↪participants/Temperature DomainParticipant",hostname="presanella",process_
↪id="458346"} 1 1678836129957
```

If there is no metric data available, you will see data as shown below with metric documentation only, but no metric data.

```
# HELP exposer_transferred_bytes_total Transferred bytes to metrics services
# TYPE exposer_transferred_bytes_total counter
exposer_transferred_bytes_total 4017
# HELP exposer_scrapes_total Number of times metrics were scraped
# TYPE exposer_scrapes_total counter
exposer_scrapes_total 4
# HELP exposer_request_latencies Latencies of serving scrape requests, in␣
↪microseconds
# TYPE exposer_request_latencies summary
exposer_request_latencies_count 4
exposer_request_latencies_sum 2510
exposer_request_latencies{quantile="0.5"} 564
exposer_request_latencies{quantile="0.9"} 621
exposer_request_latencies{quantile="0.99"} 621
# TYPE dds_domain_participant_presence gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_pkts_period_ms gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_pkts_count gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_pkts_mean gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_pkts_min gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_pkts_max gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_bytes_period_ms gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_bytes_count gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_bytes_mean gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_bytes_min gauge
# TYPE dds_domain_participant_udpv4_usage_in_net_bytes_max gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_pkts_period_ms gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_pkts_count gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_pkts_mean gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_pkts_min gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_pkts_max gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_bytes_period_ms gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_bytes_count gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_bytes_mean gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_bytes_min gauge
# TYPE dds_domain_participant_udpv4_usage_out_net_bytes_max gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_pkts_period_ms gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_pkts_count gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_pkts_mean gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_pkts_min gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_pkts_max gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_bytes_period_ms gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_bytes_count gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_bytes_mean gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_bytes_min gauge
# TYPE dds_domain_participant_udpv6_usage_in_net_bytes_max gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_pkts_period_ms gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_pkts_count gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_pkts_mean gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_pkts_min gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_pkts_max gauge
```

(continues on next page)

(continued from previous page)

```
# TYPE dds_domain_participant_udpv6_usage_out_net_bytes_period_ms gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_bytes_count gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_bytes_mean gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_bytes_min gauge
# TYPE dds_domain_participant_udpv6_usage_out_net_bytes_max gauge
```

If you see metric documentation lines only, verify that your applications are configured to use the same Observability domain as *Observability Collector Service* (2 is the default).

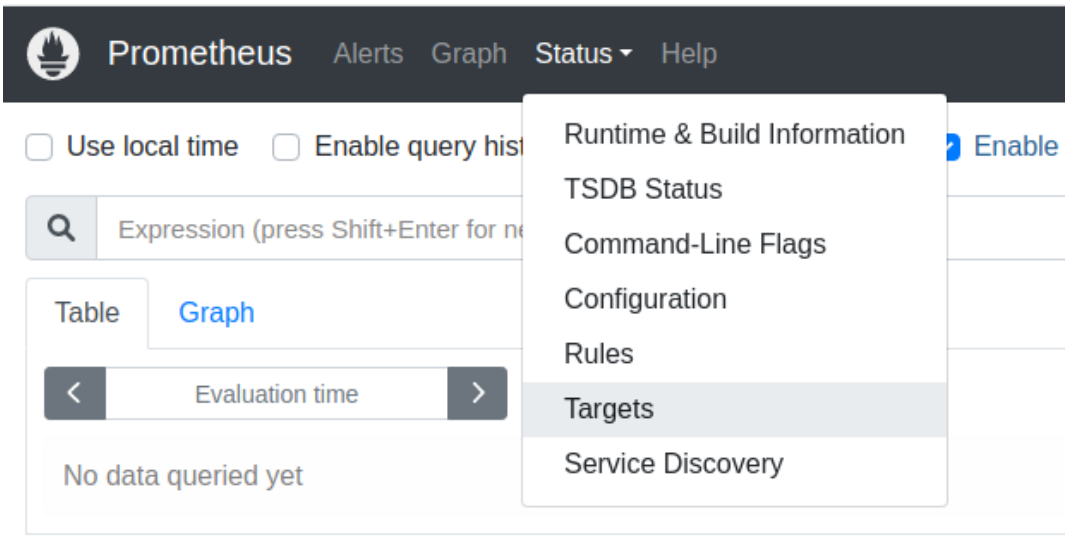
If your applications are run on a machine other than the one hosting *Observability Collector Service*, ensure that `collector_initial_peers` for the *Monitoring Library 2.0* configuration in each application is configured with the IP address where *Observability Collector Service* is running.

For more information on configuring *Monitoring Library 2.0* for your application, see *Monitoring Library 2.0*.

12.2.2 Check that Prometheus can Access Collector Service

Open a browser to `<servername>:<port>` where `servername` is the server where Prometheus is installed and `port` is the port number for the Prometheus Server (9090 is the default).

Select the **Status > Targets** menu to view configured targets as shown below.

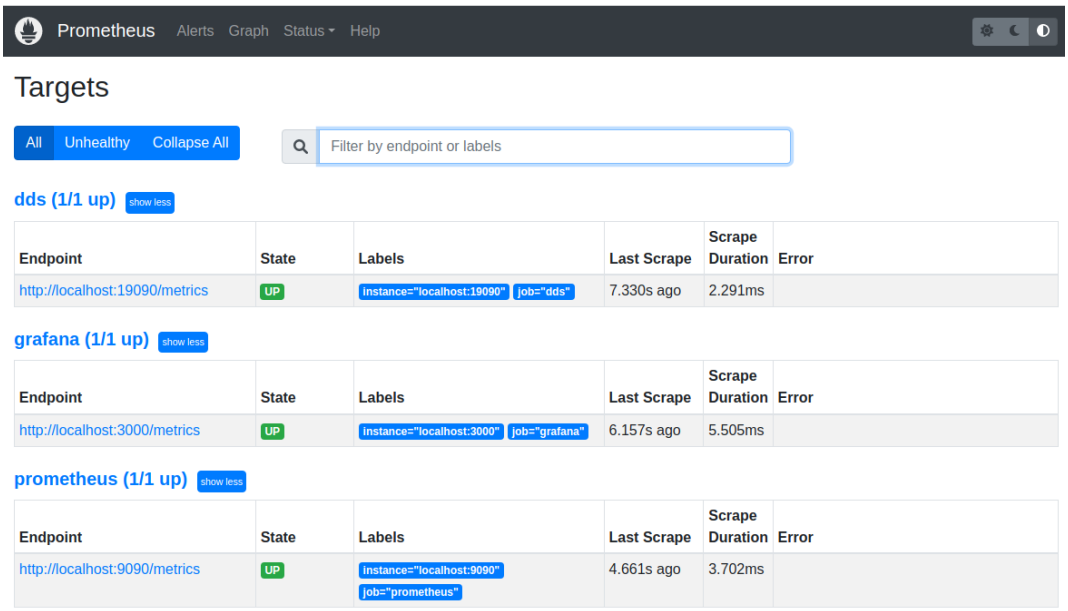


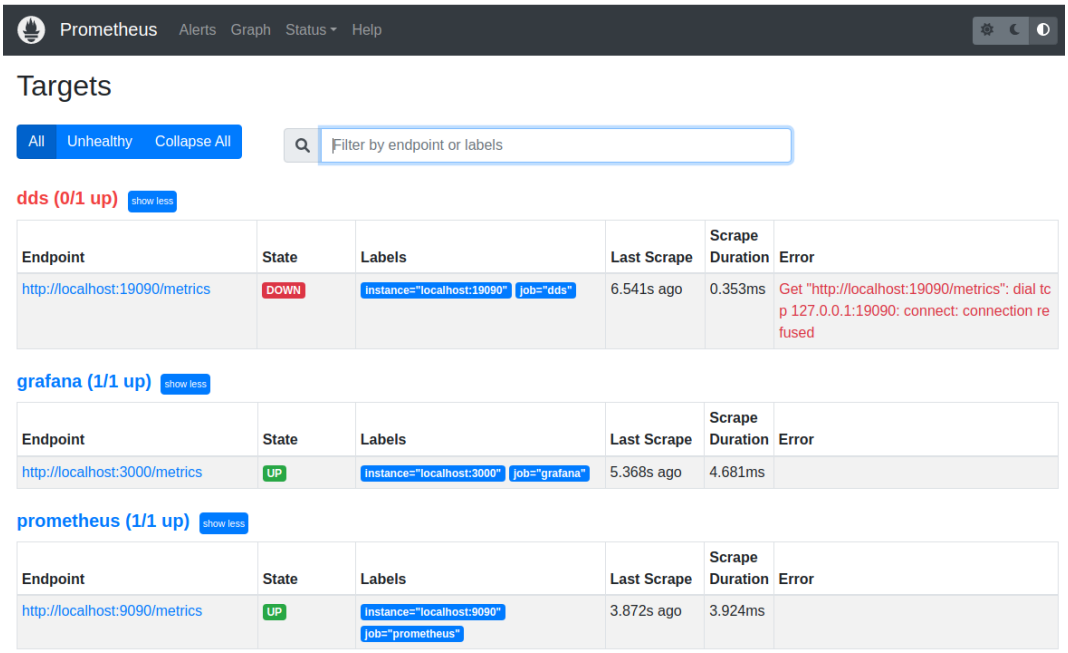
A Prometheus Server with all healthy targets is shown below.

A Prometheus Server with an unhealthy *Collector Service* is shown below. Note the `DOWN` indication for the state of the `dds` target.

If *Collector Service* is shown as `DOWN`, check the following:

- *Collector Service* is running.
- The `Endpoint` URL for *Collector Service* is correct (including port).


Prometheus
Alerts
Graph
Status
Help
Targets
All
Unhealthy
Collapse All
Filter by endpoint or labels
dds (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://localhost:19090/metrics
UP
instance="localhost:19090"
job="dds"
7.330s ago
2.291ms
grafana (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://localhost:3000/metrics
UP
instance="localhost:3000"
job="grafana"
6.157s ago
5.505ms
prometheus (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://localhost:9090/metrics
UP
instance="localhost:9090"
job="prometheus"
4.661s ago
3.702ms


Prometheus
Alerts
Graph
Status
Help
Targets
All
Unhealthy
Collapse All
Filter by endpoint or labels
dds (0/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://localhost:19090/metrics
DOWN
instance="localhost:19090"
job="dds"
6.541s ago
0.353ms
Get "http://localhost:19090/metrics": dial tcp 127.0.0.1:19090: connect: connection refused
grafana (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://localhost:3000/metrics
UP
instance="localhost:3000"
job="grafana"
5.368s ago
4.681ms
prometheus (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://localhost:9090/metrics
UP
instance="localhost:9090"
job="prometheus"
3.872s ago
3.924ms

- Examine the `Error` to see if there is another cause being reported.

12.2.3 Check that Grafana can Access Prometheus

Note: These steps can only be performed as a Grafana Admin user. The Grafana images in this section were generated with Grafana version 10.1.4. If you are using a different version of Grafana, the details might be slightly different.

In *Observability Dashboards*, click the hamburger menu and select **Connections > Data source**.

Select the "Prometheus" data source.

Scroll down and click **Test** to ensure that Grafana has connectivity with the Prometheus server.

If the test passes, the following message is displayed.

If the test fails, the following message is displayed.

If the Prometheus Data Source connectivity test fails, check the following:

- The Prometheus Server is running.
- The HTTP URL matches your Prometheus server URL (including port).
- Examine the error response to debug the connection.

12.2.4 Check that Grafana can Access Loki

Note: These steps can only be performed as a Grafana Admin user. The Grafana images in this section were generated with Grafana version 10.1.4. If you are using a different version of Grafana, the details might be slightly different.

In *Observability Dashboards*, click the hamburger menu and select **Connections > Data source**.

Select the Loki data source.

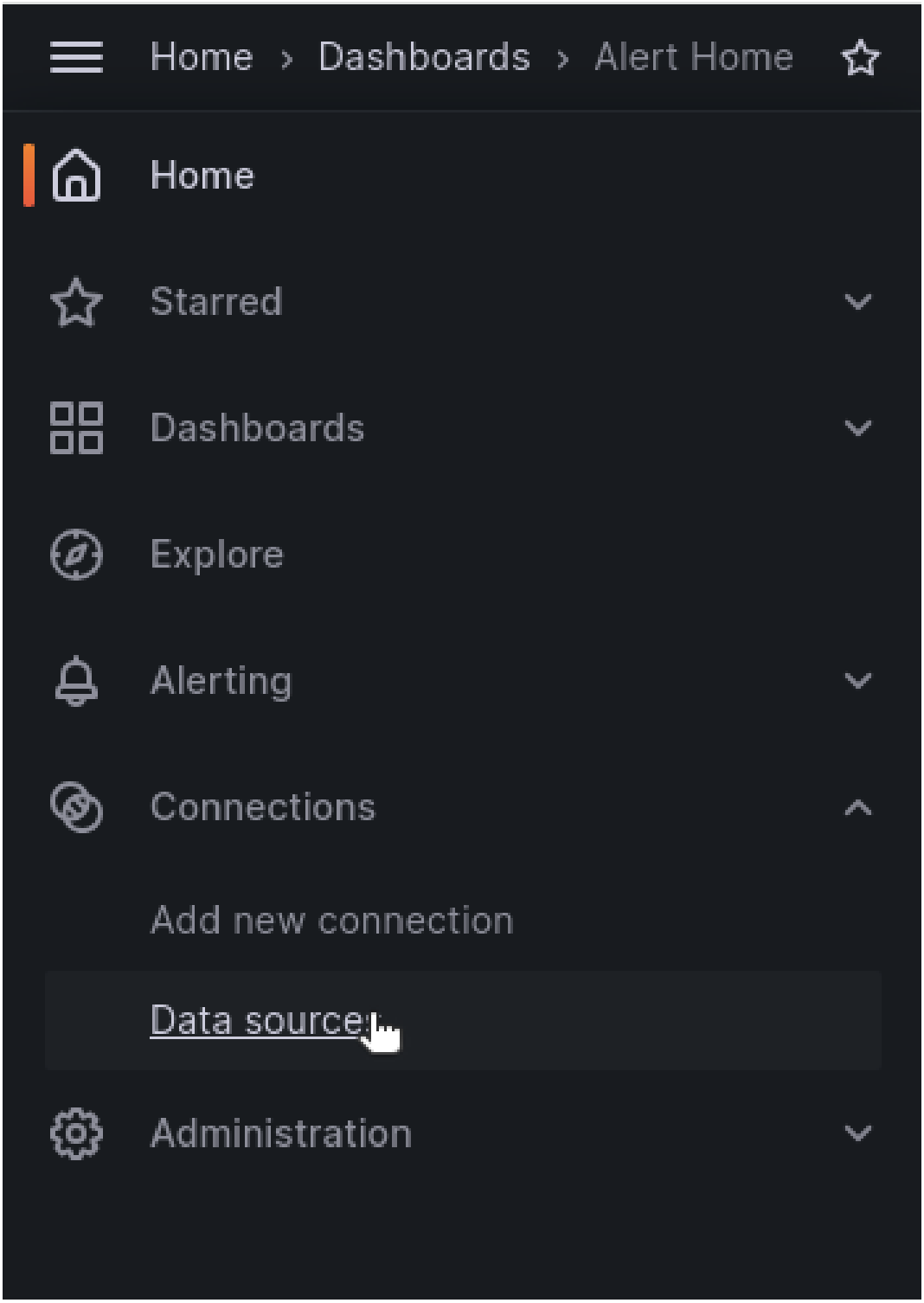
Scroll down and click **Test** to ensure that Grafana has connectivity with the Loki server.

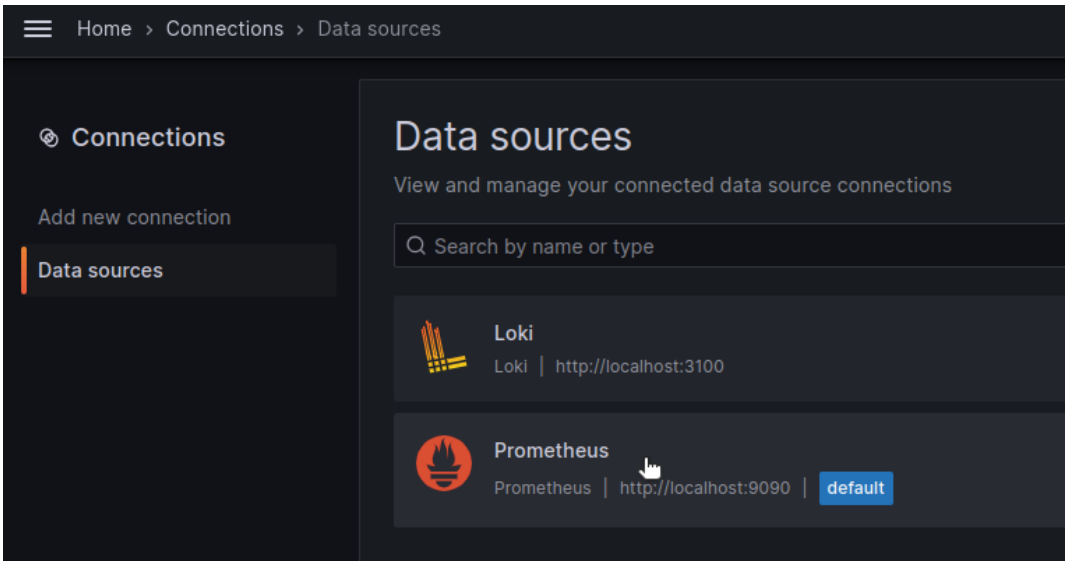
If the test passes, the following message is displayed.

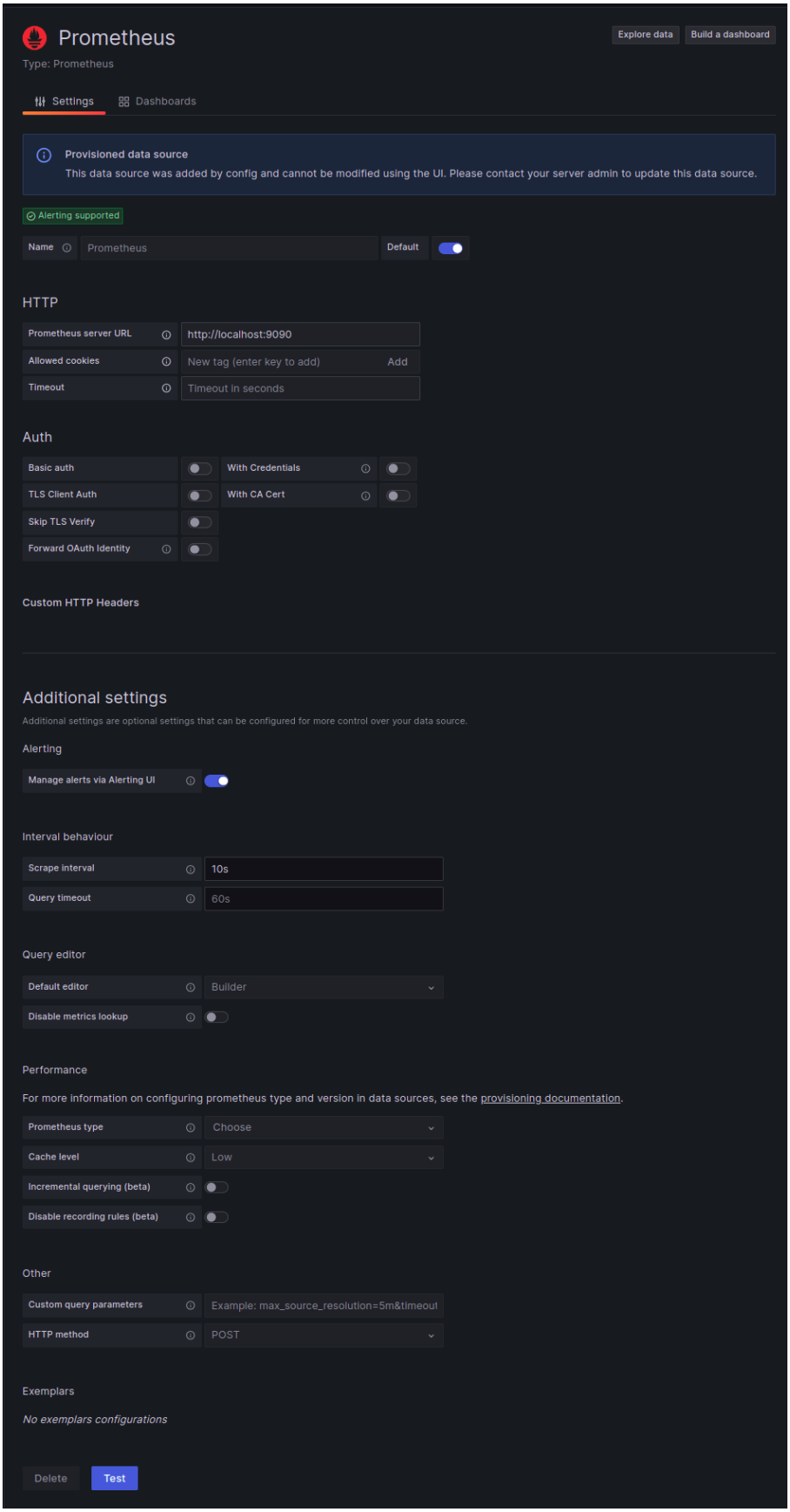
If the test fails, the following message is displayed.

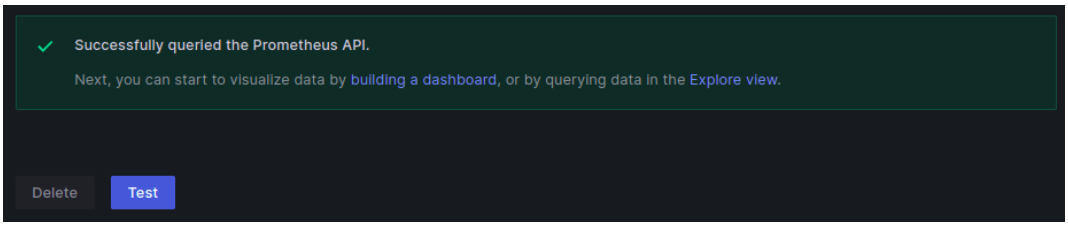
If the Loki Data Source connectivity test fails, check the following:

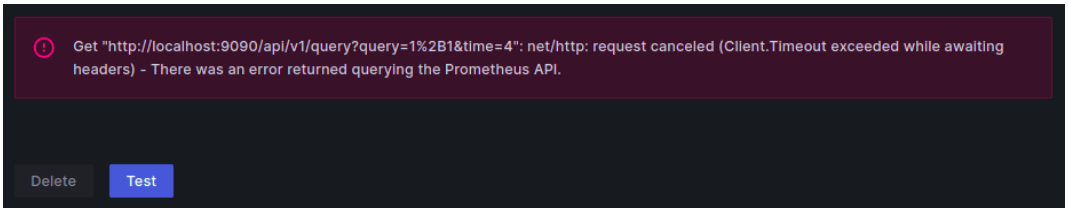
- The Loki Server is running.
- The HTTP URL matches your Loki server URL (including port).
- Examine the error response to debug the connection.

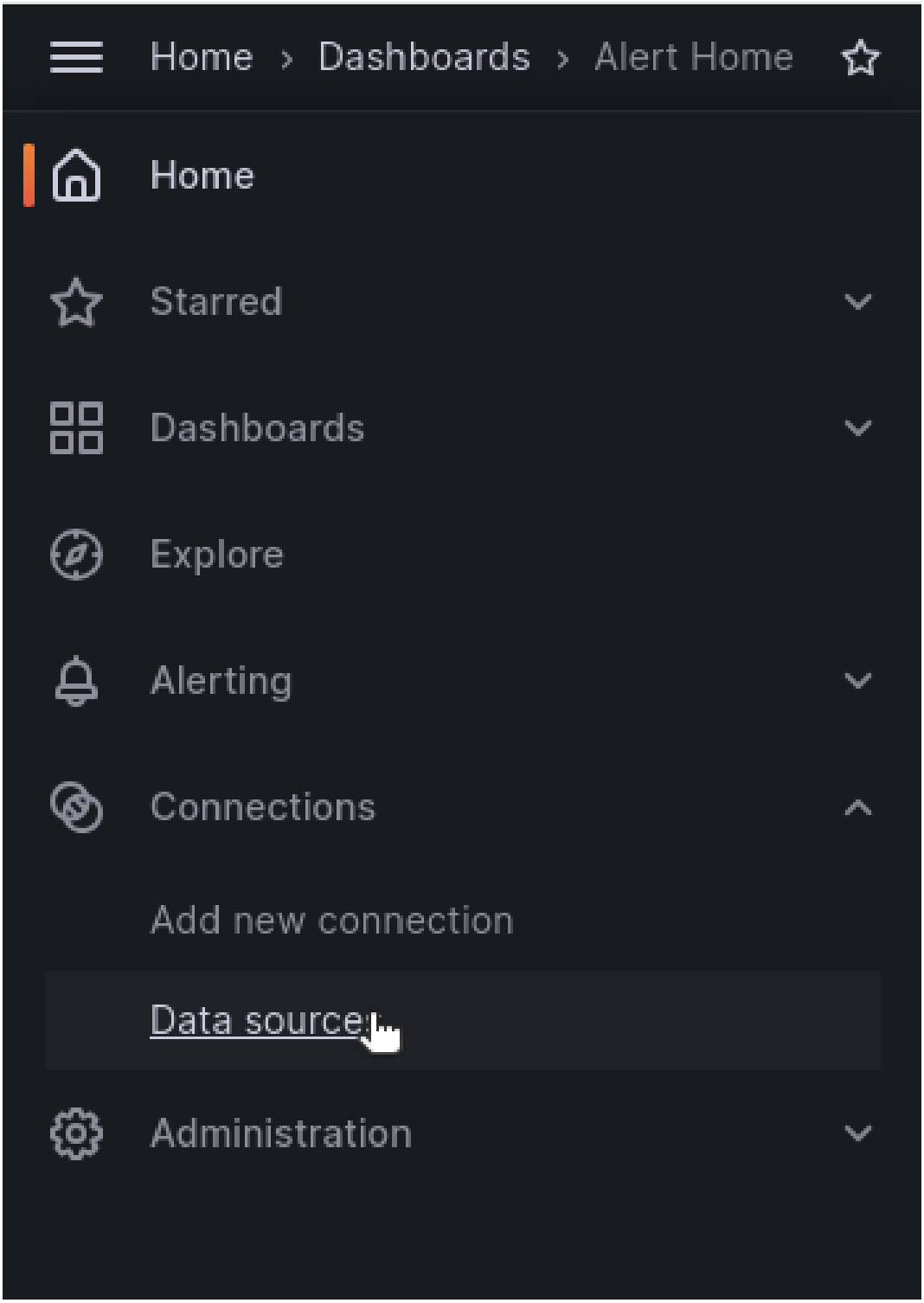

Home › Dashboards › Alert Home
Home
Starred
Dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration

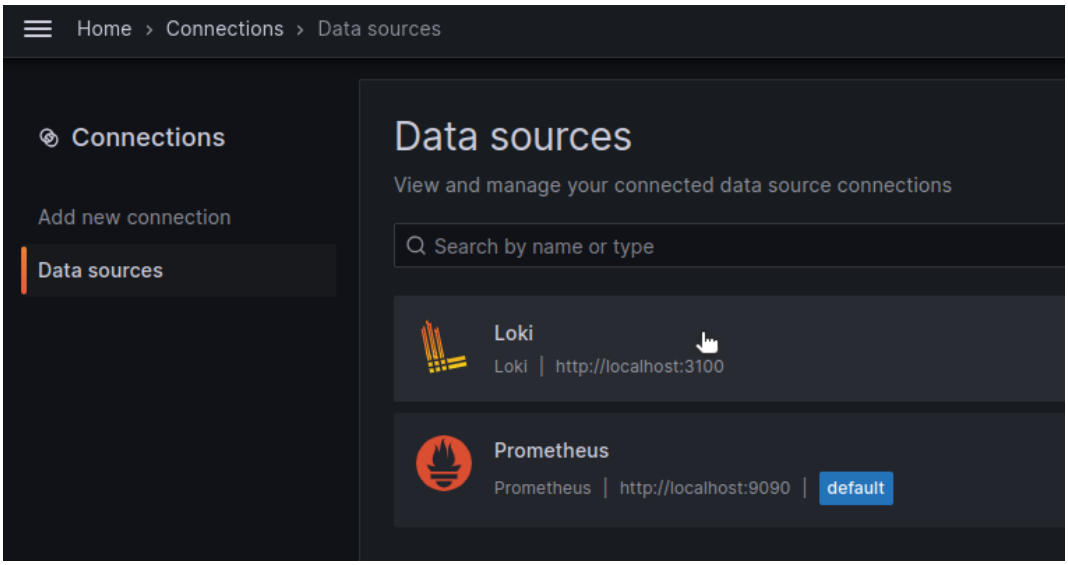

Home › Connections › Data sources
Connections
Add new connection
Data sources
Data sources
View and manage your connected data source connections
Search by name or type
Loki
Loki | http://localhost:3100
Prometheus
Prometheus | http://localhost:9090 | default

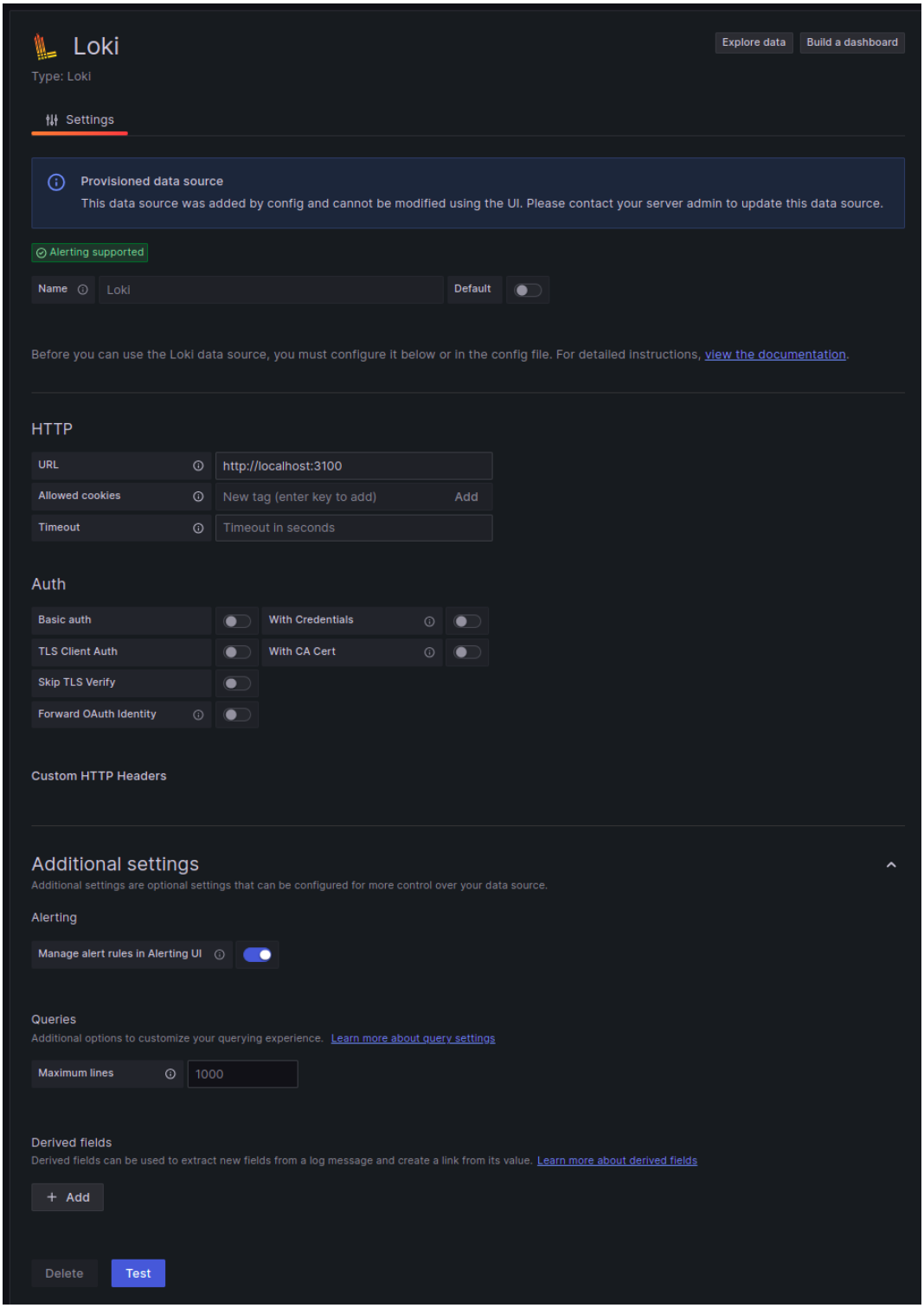

Prometheus
Explore data
Build a dashboard
Type: Prometheus
Settings
Dashboards
Provisioned data source
This data source was added by config and cannot be modified using the UI. Please contact your server admin to update this data source.
Alerting supported
Name
Prometheus
Default
HTTP
Prometheus server URL
http://localhost:9090
Allowed cookies
New tag (enter key to add)
Add
Timeout
Timeout in seconds
Auth
Basic auth
With Credentials
TLS Client Auth
With CA Cert
Skip TLS Verify
Forward OAuth Identity
Custom HTTP Headers
Additional settings
Additional settings are optional settings that can be configured for more control over your data source.
Alerting
Manage alerts via Alerting UI
Interval behaviour
Scrape interval
10s
Query timeout
60s
Query editor
Default editor
Builder
Disable metrics lookup
Performance
For more information on configuring prometheus type and version in data sources, see the provisioning documentation.
Prometheus type
Choose
Cache level
Low
Incremental querying (beta)
Disable recording rules (beta)
Other
Custom query parameters
Example: max_source_resolution=5m&timeout
HTTP method
POST
Exemplars
No exemplars configurations
Delete
Test

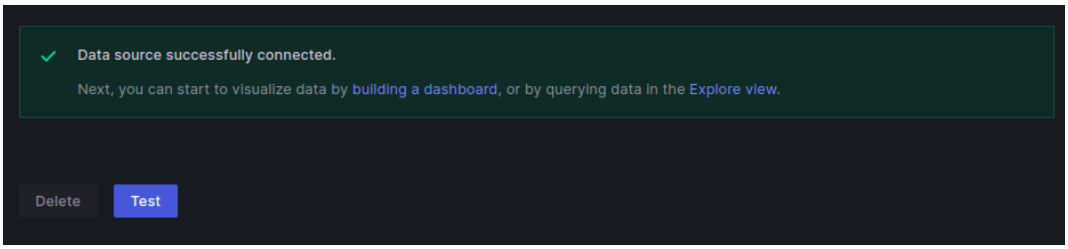

Successfully queried the Prometheus API.
Next, you can start to visualize data by building a dashboard, or by querying data in the Explore view.
Delete
Test

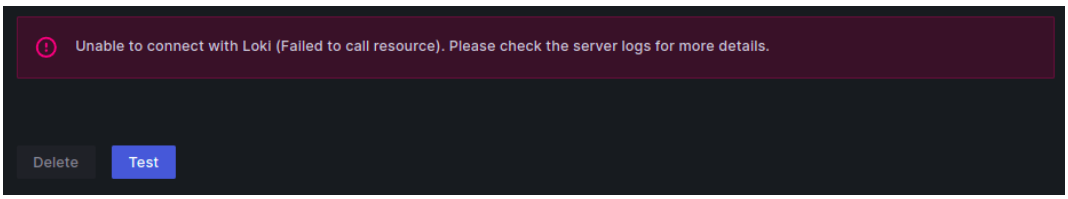

Get "http://localhost:9090/api/v1/query?query=1%2B1&time=4": net/http: request canceled (Client.Timeout exceeded while awaiting headers) - There was an error returned querying the Prometheus API.
Delete
Test


Home › Dashboards › Alert Home
Home
Starred
Dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration


Home › Connections › Data sources
Connections
Add new connection
Data sources
Data sources
View and manage your connected data source connections
Search by name or type
Loki
Loki | http://localhost:3100
Prometheus
Prometheus | http://localhost:9090 | default


Loki
Explore data
Build a dashboard
Type: Loki
Settings
Provisioned data source
This data source was added by config and cannot be modified using the UI. Please contact your server admin to update this data source.
Alerting supported
Name
Loki
Default
Before you can use the Loki data source, you must configure it below or in the config file. For detailed instructions, view the documentation.
HTTP
URL
http://localhost:3100
Allowed cookies
New tag (enter key to add)
Add
Timeout
Timeout in seconds
Auth
Basic auth
With Credentials
TLS Client Auth
With CA Cert
Skip TLS Verify
Forward OAuth Identity
Custom HTTP Headers
Additional settings
Additional settings are optional settings that can be configured for more control over your data source.
Alerting
Manage alert rules in Alerting UI
Queries
Additional options to customize your querying experience. Learn more about query settings
Maximum lines
1000
Derived fields
Derived fields can be used to extract new fields from a log message and create a link from its value. Learn more about derived fields
Add
Delete
Test


Data source successfully connected.
Next, you can start to visualize data by building a dashboard, or by querying data in the Explore view.
Delete
Test


Unable to connect with Loki (Failed to call resource). Please check the server logs for more details.
Delete
Test

Chapter 13

Glossary

Table 13.1: Observability Glossary

Term	Definition
Observability	The ability to determine a system's current state based on the telemetry data it generates, such as logs and metrics, so that you can figure out what's going on and quickly determine the root cause of problems you may not have been able to anticipate.
Application Telemetry	The automated process used to remotely collect measurements and other types of data that describe application status. The data is sent from applications to observability backends for analysis to improve system performance.
Telemetry Data	The data generated and forwarded by the Application Telemetry process, including all logs, metrics, events, and traces that are created by the applications.
OpenTelemetry	An open-source CNCF (Cloud Native Computing Foundation) project that provides a collection of tools, vendor-neutral APIs, and SDKs for capturing metrics, distributed traces, and logs from applications. Formed from the merger of the OpenCensus and OpenTracing projects, OpenTelemetry resolves the problem of integration with different observability backend technologies.

Chapter 14

Release Notes

Connext Observability Framework uses telemetry data to help identify and resolve potential issues in *Connext* applications. This product is not installed as part of a *Connext* package; it must be downloaded and installed separately, as described in *Installing and Running Observability Framework*.

Important: *Observability Framework* is an experimental product that includes example configuration files for use with several third-party components (Prometheus, Grafana Loki, and Grafana). This release is an evaluation distribution; use it to explore the new observability features that support *Connext* applications.

Do not deploy any *Observability Framework* components in production.

14.1 Supported Platforms

See Supported Platforms, in the RTI Connext Core Libraries Release Notes.

14.2 Compatibility

Connext Observability Framework is an optional product released with *RTI Connext* 7.2.0.

The current *Observability Framework* release is not compatible with *Connext* 7.1.0. This release works only with *Connext* 7.2.0 AND 7.3.0.

14.3 Supported Docker Compose Environments

The *Observability Framework* package enables you to deploy and run *Observability Collector Service* and third-party components NGINX, OpenTelemetry Collector, Prometheus, Grafana Loki, and Grafana, using Docker Compose in a single Linux host. The host can run on a Virtual Machine (VM); in this release, we have tested the following combinations:

Table 14.1: Tested VM/OS Combinations

Host Architecture/OS	VM Architecture/OS	VM
x86 Windows	x86 Ubuntu	VirtualBox
x86 Mac	x86 Ubuntu	Parallels
x86 Ubuntu	None	None

Important: The Docker Compose distribution uses "host" networking that only works on Linux hosts; it is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.

Windows virtualization technologies such as WSL, WSL2, and Hyper-V also do not support "host" networking. Running *Observability Framework* using Docker Compose in these environments will not work.

14.4 What's New in 7.3.0 LTS

Observability Framework is an experimental product included with *Connext* 7.3.0 LTS, a long-term support release that is built upon and combines all of the features in releases 7.1.0 and 7.2.0 (see *Previous Releases*). See the Connext Releases page on the RTI website for more information on RTI's software release model.

Note: For what's new in *Monitoring Library 2.0*, see the Connext Core Libraries Release Notes.

14.4.1 Enhanced control of entities distributed across various Collector Service instances

Observability Collector Service now adds a new label, `controllability_url`, to the presence metrics of the applications that it monitors; for example, `controllability_url: 'https://localhost:19098'`. This label contains the Host and Port that should be used to send commands to the applications and their resources using the *Collector Service* REST API (see the *Collector Service REST API Reference*). The label takes into account whether HTTPS is enabled in the *Collector Service* configuration.

The new label is useful when using multiple *Collector Service* instances to monitor the different applications. It allows you to send commands to the applications and their resources using the REST API without knowing the *Collector Service* instance that is monitoring the application.

For pre-packaged installations, the Host and Port information of the *Collector Service* can be configured using the following parameters in the configuration JSON file:

- collectorConfig.controlPublicHostname
- collectorConfig.controlPublicPort

For more information, see *Configure the JSON File*.

For standalone deployments, the Host and Port information can be configured using two new environment variables in the *Collector Service* Docker image:

- OBSERVABILITY_CONTROL_PUBLIC_HOSTNAME
- OBSERVABILITY_CONTROL_PUBLIC_PORT

For more information, see RTI's Docker Collector Service Repository.

14.4.2 New REST API in Collector Service to control telemetry data collection and distribution

This release introduces a REST API in Collector Service that allows controlling the telemetry data collection and distribution at run-time. The REST API includes commands to:

- get the current logging collection and forwarding verbosity levels for applications
- dynamically change the logging collection and forwarding verbosity levels for applications
- get the current metric collection configuration for observable resources
- dynamically configure the set of metrics collected and forwarded for observable resources

For more details on how to use these endpoints, see *Collector Service REST API Reference*.

14.4.3 Support for more flexible Observability Framework deployments

The *Collector Service* Docker image exposes additional environment variables and configuration parameters that allow more flexible deployments of the *Observability Framework* components.

When running the *Observability Framework* components standalone, you can now specify the following environment variables:

- `Loki hostname` to send data to a Loki server on a remote host
- `OTel hostname` to send data to an OTel collector on a remote host
- `RWT hostname` to specify the *Collector Service* public address when using *Real-Time WAN Transport*
- `Collector Service control public hostname and port` to specify the public access to the *Collector Service* control server when it is deployed behind a NAT or load balancer

Additionally, you can provide the security artifacts independently for all HTTP servers and clients created by *Collector Service*. For details on the *Collector Service* standalone deployment, see the *Docker (Separate Deployment)* section.

When running the prepackaged Docker Compose installation included in the *Observability Framework* host package, you can now specify the `RWT hostname` and `Collector Service control public`

`hostname and port` environment variables. The prepackaged installation deploys the Loki and OTel collectors on the same node, so there is no need to specify these as remote hosts. For details on the prepackaged installation, see the *Docker Compose (Prepackaged)* section.

14.4.4 Control which metrics are collected

Previously, every observable resource (*DomainParticipant*, *Publisher*, *DataReader*, etc.) was, by default, subscribed to all available metrics for that resource when it was registered.

Starting in release 7.3.0, all metric collection is disabled by default. You can control which metrics are collected using one or both of the following methods:

- Configure the initial set of metrics a resource is subscribed to using the Monitoring QosPolicy. For details and an example, see *Setting the Initial Metrics and Log Configuration*.
- Dynamically configure metric collection via *Observability Dashboards* during run time. For details and an example, see *Change the Metric Configuration*.

14.4.5 New Syslog facilities provide expanded log management

Previously, *Observability Framework* only managed log messages produced by the *Connext* Core and API libraries, in accordance with the Syslog Protocol facility `23 (middleware)`. The framework retrieved and set the collection and forwarding verbosities for this facility.

In this release, these capabilities have been implemented for three additional facilities: `10 (security_event)`, `22 (service)`, and `1 (user)`.

- The `security_event` facility applies to messages generated by *RTI Connext Security Plugins* that are defined as security events by the DDS Secure standard.
- The `service` facility applies to messages generated by Connext Infrastructure Services: *Routing Service*, *Recording Service*, *Cloud Discovery Service*, and *Web Integration Service*.
- The `user` facility applies to messages generated by the User logging API. For details, see *Logs*.

14.4.6 New logging category and plugin class labels enable more precise third-party backend queries

Starting in release 7.3.0, every log message pushed by *Observability Framework* to a logging third-party backend (for example, Grafana Loki) contains a new label representing the logging category the message belongs to. If the logging category is not available for a message , its value will be `N/A`.

Each message belonging to the `security_event` facility also includes a label indicating the standard plugin class name that originated the message, as defined in the OMG 'DDS Security' specification. For example, `DDS:Auth:PKI-DH`, `DDS:Access:Permissions`, `DDS:Crypto:AES-GCM-GMAC`, or `DDS:Logging:DDS_LogTopic`. The logging category is just a friendly name for these standard plugin class names.

The logging categories and plugin classes (when available) are also displayed in the provided Grafana Dashboards.

For details about the new labels, see *Log Labels*.

14.4.7 Updated dashboards support enhanced logging and dynamic metric control

Observability Dashboards have been updated to provide a cleaner interface and support new *Observability Framework* features. These updates include:

- Dynamic metric control
- Enhanced logging capabilities
  - Visualization of new new log metadata
    - facilities (USER, SERVICE, SECURITY_EVENT)
    - categories
    - plugin classes
  - Ability to dynamically control log generation and forwarding for all Syslog facilities
- GUI improvements
  - Entity history chart
  - Larger metric charts
  - Updated queries
  - Alerting for enabled metrics

For more information, see *Observability Dashboards*.

14.4.8 Name change for some observability metrics

This release changes the following metric name prefixes associated with *Connext* entities. For details about all available metrics, see *Metrics*.

Old Metric Name Prefix	New Metric Name Prefix
`dds_participant_*`	`dds_domain_particpant_*`
`dds_datareader_*`	`dds_data_reader_*`
`dds_datawriter_*`	`dds_data_writer_*`

14.4.9 Third-party software upgrades

Observability Collector Service

The following third-party software used by *Observability Collector Service* has been upgraded:

Table 14.2: Third-Party Software Upgrades (Collector Service)

Third-Party Software	Previous Version	Current Version
OpenTelemetry C++	1.9.1	1.13.0
OpenSSL	3.0.9	3.0.12

Docker containers for Observability Collector Service

The following third-party software used by the Docker containers created by *Observability Framework* has been upgraded:

Table 14.3: Third-Party Software Upgrades (Docker Containers)

Third-Party Software	Previous Version	Current Version
Prometheus	2.37.8	2.45.1
Grafana	9.5.3	10.1.4
Grafana Loki	2.8.2	2.8.5
OpenTelemetry Collector Contrib	0.80.0	0.91.0

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: <NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty.

Warning: All third-party software is subject to third-party license terms and conditions. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

14.5 What's Fixed in 7.3.0 LTS

This section describes bugs fixed in *Observability Framework* 7.3.0 LTS. These are fixes applied since 7.2.0. For information on what was fixed in releases 7.0.0, 7.1.0, and 7.2.0, which are also part of 7.3.0 LTS, see *Previous Releases*.

Note: For what's fixed in *Monitoring Library 2.0*, see the Connext Core Libraries Release Notes.

[Critical]: System-stopping issue, such as a crash or data loss. [Major]: Significant issue with no easy workaround. [Minor]: Issue that usually has a workaround. [Trivial]: Small issue, such as a typo in a log.

14.5.1 Crashes

[Critical] Observability Collector Service could crash when an application was discovered

When *Observability Collector Service* discovers an application, *Monitoring Library 2.0* sends a special sample with information about the discovered application, such as the logging configuration, process ID, and host name.

Normally, this information is sent in a single sample, but it could potentially be split into more than one sample. If, due to timing, the process ID or the host name was sent in a separate sample from the logging configuration, *Observability Collector Service* accessed a null pointer which led to a crash due to an invalid condition check.

[RTI Issue ID OCA-307]

14.5.2 Vulnerabilities

[Critical] Potential out of memory error when using Curl 8.1.2

Observability Collector Service had a third-party dependency on Curl 8.1.2, which is known to be affected by a number of publicly disclosed vulnerabilities. These vulnerabilities have been fixed by upgrading Curl to the latest stable version, 8.5.0.

User impact without security

This vulnerability impacts Connext 7.2.0 applications using *Observability Collector Service*, as follows:

- Exploitable by streaming an endless series of headers to the application using Curl.
- The application could run out of memory.
- CVSS Base Score: 7.5 HIGH
- CVSS v3.1 Vector: AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

User impact with security

Same as "User Impact without Security," above.

[RTI Issue ID OCA-303]

[Critical] Potential deletion of HSTS data when using Curl 8.1.2

Observability Collector Service had a third-party dependency on Curl 8.1.2, which is known to be affected by a number of publicly disclosed vulnerabilities. These vulnerabilities have been fixed by upgrading Curl to the latest stable version, 8.5.0.

User impact without security

This vulnerability impacts *Connext* 7.2.0 applications using the *Observability Collector Service*, as follows:

- When saving HSTS data to an excessively long file name, Curl could end up removing all contents.
- Making subsequent requests using that file unaware of the HSTS status they should otherwise use.
- CVSS Base Score: 5.3 MEDIUM
- CVSS v3.1 Vector: AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N

User impact with security

Same as "User Impact without Security," above.

[RTI Issue ID OCA-324]

14.6 Previous Releases

14.6.1 What's New in 7.2.0

Observability Collector Service compatible with Monitoring Library 2.0

All the *DomainParticipants* from *Collector Service* are correctly detected using *Monitoring Library 2.0* and *Observability Framework*.

To activate *Monitoring Library 2.0* in *Collector Service*, run the service from a folder with a file named USER_QOS_PROFILES.xml and the following content:

```
<?xml version="1.0"?>
<dds>
    <qos_library name="MonitoringEnabledLibrary">
        <qos_profile name="MonitoringEnabledProfile" is_default_participant_
↪factory_profile="true">
            <participant_factory_qos>
                <monitoring>
                    <enable>true</enable>
                </monitoring>
            </participant_factory_qos>
        </qos_profile>
```

(continues on next page)

(continued from previous page)

```
    </qos_library>
</dds>
```

For more information about using XML QoS profiles, see How to Load XML-Specified QoS Settings, in the RTI Connext Core Libraries User's Manual

Support for most observability backends with OpenTelemetry integration

Previously, *Observability Framework* only allowed storing metrics in a Prometheus time-series database and logs in a Grafana Loki log aggregator. This release adds support for sending telemetry data (metrics and logs) to an OpenTelemetry Collector, providing a way to store the telemetry data in other third-party observability backends.

Support for Observability Collector Service security

Starting with *RTI Connext* 7.2.0, *Collector Service* can be secured using the Security Plugins to communicate with *Monitoring Library 2.0*. (see *Secured communications between Monitoring Library 2.0 and Observability Collector Service*). *Collector Service* can also use BASIC-Auth over HTTPS to secure the telemetry data sent to the observability backends and the remote commands received from *Observability Dashboards*.

For additional details, see Support for RTI Observability Framework in the *RTI Security Plugins User's Manual*.

Name change from "RTI Observability Library" to "RTI Monitoring Library 2.0"

For details, see RTI Connext Core Libraries What's New in 7.2.0.

Name change for some Observability metrics

This release changes the name of some metrics associated with *Connext* entities.

This change applies to the following eight metrics. See the *Metrics* section of this user manual for details about all available metrics.

Table 14.4: Metric Name Changes

Old Metric Name	New Metric Name
dds_application_process_utilization_memory_usage_resident_memory_bytes	dds_application_process_memory_usage_resident_memory_bytes
dds_application_process_utilization_memory_usage_virtual_memory_bytes	dds_application_process_memory_usage_virtual_memory_bytes
dds_datawriter_reliable_cache_unacknowledged_samples	dds_datawriter_reliable_cache_unack_samples
dds_datawriter_reliable_cache_unacknowledged_samples_peak	dds_datawriter_reliable_cache_unack_samples_peak
dds_datawriter_reliable_cache_replaced_unacknowledged_samples_total	dds_datawriter_reliable_cache_replaced_unack_samples_total
dds_datareader_cache_old_source_timestamp_dropped_samples_total	dds_datareader_cache_old_source_ts_dropped_samples_total
dds_datareader_cache_tolerance_source_timestamp_dropped_samples_total	dds_datareader_cache_tolerance_source_ts_dropped_samples_total
dds_datareader_cache_samples_dropped_by_instance_replacement_total	dds_datareader_cache_samples_dropped_by_instance_replaced_total

Secured communications between Monitoring Library 2.0 and Observability Collector Service

For details, see RTI Security Plugins What's New in 7.2.0.

Ability to set initial forwarding verbosity in MONITORING QoS policy

For details, see RTI Connext Core Libraries What's New in 7.2.0.

Ability to set collector initial peers in MONITORING QoS policy

For details, see RTI Connext Core Libraries What's New in 7.2.0.

Third-Party software changes

Observability Framework

The following third-party software is now used by the scripts that configure *Observability Framework*:

Table 14.5: Observability Framework Third-Party Software Changes

Third-Party Software	Version
bcrypt	4.0.1
Jinja2	3.0.0

Observability Collector Service

The following third-party software used by *Observability Collector Service* has been upgraded:

Table 14.6: Collector Service Third-Party Software Upgrades

Third-Party Software	Previous Version	Current Version
CivetWeb	1.15	1.16
OpenTelemetry C++	1.4.1	1.9.1

Docker containers for Observability Collector Service

The following third-party software is now used by the Docker containers created by *Observability Framework*:

Table 14.7: Third-Party Software Changes (Docker Containers)

Third-Party Software	Version
NGINX	1.24.0
OpenTelemetry Collector	0.80.0

The following third-party software used by the Docker containers created by *Observability Framework* have been upgraded:

Table 14.8: Third-Party Software Upgrades (Docker Containers)

Third-Party Software	Previous Version	Current Version
Prometheus	2.37.5	2.37.8
Grafana	9.2.1	9.5.3
Grafana Loki	2.7.0	2.8.2

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: <NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty.

Warning: All third-party software is subject to third-party license terms and conditions. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

14.6.2 What's Fixed in 7.2.0

To review any fixes applied to *Monitoring Library 2.0*, see What's Fixed in 7.2.0, in the RTI Connext Core Libraries Release Notes.

Collector Service might have crashed on startup

Collector Service could have crashed on startup if something failed in the initialization of one of its components. This happened because the clean-up method called after the failure accessed invalid memory. Before the crash, error messages appeared in the `RTI_MonitoringForwarder_initialize` function.

For example, the initialization would fail if either the `event_datareader_qos`, `logging_datareader_qos`, or `periodic_datareader_qos` of the `input_connection` were configured with inconsistent QoS.

This issue is resolved.

[RTI Issue ID OCA-226]

Controllability issues on applications with same name

When multiple monitored applications shared the same application name, the exit of one of these applications could disrupt control of the remaining ones. This issue also occurred when a monitored application was closed ungracefully and then restarted. This issue has been fixed; now, the GUID of the application is also considered when an application is accessed using its name.

[RTI Issue ID OCA-224]

Unhandled exceptions may have caused segmentation fault

Observability Collector Service was not handling exceptions in the destructor; if an exception occurred, this issue may have led to a segmentation fault at the time of destruction. This issue has been fixed.

[RTI Issue ID OCA-289]

Race condition when processing remote commands led to failures and memory leaks when shutting down Collector Service

In *Observability Collector Service*, due to an internal race condition, the cleanup done after a remote administration command (for example, changing the forwarding or collection verbosity of an application) was processed could fail with the following error message:

```
ERROR DDS_AsyncWaitSetTask_detachCondition:!detach condition
```

This left one of the internal components of *Observability Collector Service* in an inconsistent state, which caused failures and memory leaks when the service was shut down:

```
ERROR DDS_AsyncWaitSet_submit_task:!wait for request completion
ERROR DDS_AsyncWaitSet_detach_condition_with_completion_token:!submit␣
↪internal task
ERROR DDS_AsyncWaitSet_detach_condition:!DDS_AsyncWaitSet_detach_condition_
↪with_completion_token
ERROR DDS_AsyncWaitSet_finalize:!detach condition
ERROR DDS_AsyncWaitSet_delete:!DDS_AsyncWaitSet_finalize
```

This race condition is fixed. The cleanup of already processed commands no longer causes unexpected failures.

[RTI Issue ID MONITOR-610]

Collector Service could discard samples when monitoring large DDS applications

In the previous release, *Observability Collector Service* could report the following error messages when collecting telemetry data from applications with a large number of DDS entities (for example, 2000 DataWriters):

```
ERROR [0x01016A0B,0x38EDDDA5,0x6C2A146D:0x00000184{Entity=DR,MessageKind=DATA}
↪|RECEIVE FROM 0x0101DC38,0xA4FD24A4,0x06193ECA:0x00000183] DDS_DataReader_
↪add_sample_to_remote_writer_queue_untypedI:add sample to remote writer queue
ERROR [0x01016A0B,0x38EDDDA5,0x6C2A146D:0x00000184{Entity=DR,MessageKind=DATA}
↪|RECEIVE FROM 0x0101DC38,0xA4FD24A4,0x06193ECA:0x00000183] RTI_
↪MonitoringForwarderEntities_addSampleToCacheReader:ADD FAILURE | Sample to␣
↪the cache reader of DCPSPeriodicStatusMonitoring
```

This problem was due to the queues of the internal Collector's *DataReaders* becoming full because of the default QoS configuration and the large amount of data received, causing new samples to be discarded and, consequently, not pushed to the *Observability Framework* backends.

This issue has been resolved. The queues for the internal *DataReaders* are now configured to have no limit, ensuring successful telemetry data collection regardless of the number of DDS entities.

Note: The example error messages above refer to the Periodic Topic, but the same messages were reported for other *Observability Framework* Topics (Events and Logging).

[RTI Issue ID MONITOR-619]

14.6.3 What's New in 7.1.0

Connext Observability Collector Service uses telemetry data to help identify and resolve potential issues in *Connext* applications. This product is not installed as part of a *Connext* package; it must be downloaded and installed separately, as described in *Installing and Running Observability Framework*.

Important: *Observability Framework* is an experimental product that includes example configuration files for use with several third-party components (Prometheus, Grafana Loki, and Grafana). This release is an evaluation distribution; use it to explore the new observability features that support *Connext* applications.

Do not deploy any *Observability Framework* components in production.

Third-Party Software

The following third-party software is used in *Observability Collector Service*.

Table 14.9: Third-Party Software (Observability Collector Service)

Third-Party Software	Version
CivetWeb	1.15
Prometheus-cpp	1.0.1
nlohmann-json	3.11.2

In addition, the Docker containers created by *Observability Framework* include the following third-party software.

Table 14.10: Third-Party Software (Docker Containers)

Third-Party Software	Version
Prometheus	2.37.5
Grafana	9.2.1
Grafana Loki	2.7.0

Warning: All third-party software is subject to third-party license terms and conditions. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

Important: *Observability Framework* is an experimental product that includes example configuration files for use with several third-party components (Prometheus, Grafana Loki, and Grafana). This release is an evaluation distribution; use it to explore the new observability features that support *Connext* applications.

Do not deploy any Observability Framework components in production. A production-ready version is expected to be available in a future *Connext* 7.3.x maintenance release.

HTTP Routing Table

/rti