

RTI Connex Observability Framework

User's Manual

Version 7.2.0



Your systems.
Working as one.

Contents

1	What is Connex Observability Framework?	1
1.1	Telemetry Data	1
1.2	Distribution of Telemetry Data	2
1.3	Flexible Storage	2
1.4	Visualization of Telemetry Data	2
1.5	Control and Selection of Telemetry Data	3
1.6	Security	3
2	Components	4
2.1	Monitoring Library 2.0	4
2.2	Observability Collector Service	5
2.2.1	Storage Components	6
2.3	Observability Dashboards	7
2.4	How We Provide the Components	8
2.4.1	Monitoring Library 2.0	8
2.4.2	Collection, Storage, and Visualization Components	8
	Current Release	8
	Future releases	11
3	Deployments	12
3.1	Collector Service Deployments	12
3.1.1	Current Release	12
3.1.2	Future Releases	16
4	Security	18
4.1	Secure Communication between Connex Applications and Collector Service	19
4.2	Secure Communication with Collector Service's HTTP Servers	21
4.3	Secure Communication with Third-party Components' HTTP Servers	23
4.4	Generating the Observability Framework Security Artifacts	24
4.4.1	Generating DDS Security Artifacts	24
4.4.2	Generating HTTPS Security Artifacts	24
	Preliminary Steps	25
	Generating a New Root CA	26
	Generating Server Certificates	27
	BASIC-Auth Password File	28
5	Installing and Running Observability Framework	30
5.1	Installing the Target Package	31

5.1.1	Install from RTI Launcher	31
5.1.2	Install from the Command Line	31
5.2	Installing the Host Package	31
5.2.1	Prerequisites	32
5.2.2	Install from RTI Launcher	32
5.2.3	Install from the Command Line	32
5.3	Configuring, Running, and Removing Observability Framework Components Using Docker Compose	32
5.3.1	Configuring the Docker Workspace for Observability Framework	33
	Configure the JSON File	33
	Run the Observability script to create the Observability workspace	37
5.3.2	Initialize and Run Docker Containers	39
5.3.3	Verify Docker Containers are Running	40
5.3.4	Configure Grafana	41
	Initial Login	41
	Configuration Options	41
5.3.5	Stop Docker Containers	43
5.3.6	Start Existing Docker Containers	44
5.3.7	Stop and Remove Docker Containers	45
5.3.8	Removing the Docker Workspace for Observability Framework	46
6	Getting Started Guide	48
6.1	About the Observability Example	48
6.1.1	Applications	48
6.1.2	Data Model	50
6.1.3	DDS Entity Mapping	51
6.1.4	Command Line Parameters	51
	Publishing Application	51
	Subscribing Application	52
6.2	Before Running the Example	52
6.2.1	Set Up Environment Variables	52
6.2.2	Compile the Example	53
	Non-Windows Systems	53
	Windows Systems	54
6.2.3	Install Observability Framework	54
	Configure Observability Framework for the Appropriate Operation Mode	54
6.2.4	Start the Collection, Storage, and Visualization Docker Containers	61
6.3	Running the Example	62
6.3.1	Start the Applications	62
6.3.2	Changing the Time Range in Dashboards	66
6.3.3	Simulate Sensor Failure	67
6.3.4	Simulate Slow Sensor Data Consumption	69
6.3.5	Simulate Time Synchronization Failures	71
6.3.6	Change the Application Logging Verbosity	73
6.3.7	Close the Applications	79
7	Telemetry Data	80
7.1	Introduction	80

7.2	Resources	80
7.3	Metrics	82
7.3.1	Application Metrics	83
7.3.2	Participant Metrics	83
7.3.3	Topic Metrics	86
7.3.4	DataWriter Metrics	87
7.3.5	DataReader Metrics	89
7.3.6	Derived Metrics Generated by Prometheus Recording Rules	91
	DDS Entity Proxy Metrics	92
	Raw Error Metrics	92
	Aggregated Error Metrics	99
	Enable a Raw Error Metric	101
	Custom Error Metrics	116
7.4	Logs	116
7.4.1	Collection and Forwarding Verbosity	117
8	Monitoring Library 2.0	119
8.1	Enabling Monitoring Library 2.0	120
8.2	Setting the Application Name	121
8.3	Changing the Default Observability Domain ID	121
8.4	Configuring QoS for Monitoring Library 2.0 Entities	122
8.5	Setting Collector Service Initial Peers	124
9	Troubleshooting Observability Framework	126
9.1	Docker Container[s] Failed to Start	126
9.1.1	Check for Port Conflicts	127
9.1.2	Check that You Have the Correct File Permissions	127
9.2	No Data in Dashboards	128
9.2.1	Check that Collector Service has Discovered Your Applications	128
9.2.2	Check that Prometheus can Access Collector Service	130
9.2.3	Check that Grafana can Access Prometheus	132
10	Glossary	137
11	Release Notes	138
11.1	Supported Platforms	138
11.2	Compatibility	138
11.3	Supported Docker Compose Environments	139
11.4	What's New in 7.2.0	139
11.4.1	Observability Collector Service compatible with Monitoring Library 2.0	139
11.4.2	Support for most observability backends with OpenTelemetry integration	140
11.4.3	Support for RTI Observability Collector Service security	140
11.4.4	Name change from "RTI Observability Library" to "RTI Monitoring Library 2.0"	140
11.4.5	Name change for some Observability metrics	140
11.4.6	Secured communications between RTI Monitoring Library 2.0 and RTI Observability Collector Service	141
11.4.7	Ability to set initial forwarding verbosity in MONITORING QoS policy	141
11.4.8	Ability to set collector initial peers in MONITORING QoS policy	141

11.4.9	Third-Party software changes	142
	Observability Framework	142
	Observability Collector Service	142
	Docker containers for Observability Framework	142
11.5	What's Fixed in 7.2.0	143
11.5.1	Collector Service might have crashed on startup	143
11.5.2	Controllability issues on applications with same name	143
11.5.3	Unhandled exceptions may have caused segmentation fault	144
11.5.4	Race condition when processing remote commands led to failures and memory leaks when shutting down Collector Service	144
11.5.5	Observability Collector Service could discard samples when monitoring large DDS applications	144
11.6	Previous Release	145
11.6.1	What's New in 7.1.0	145
	Third-Party Software	145
12	Copyrights and Notices	147

Chapter 1

What is Connex Observability Framework?

RTI® Connex® Observability Framework™ is a holistic solution that uses telemetry data to provide deep visibility into the current and past states of your *Connex* 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 *Connex* applications. For support, you may contact support@rti.com.

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

1.1 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 *Connex* 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. Future releases could support application 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.
- **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* only supports metrics and logs. Future releases could support security events and traces.

1.2 Distribution of Telemetry Data

Observability Framework enables you to scalably generate and forward telemetry data from individual *Connex* applications to third-party telemetry backends like Prometheus and Grafana Loki.

1.3 Flexible Storage

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

1.4 Visualization of Telemetry Data

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

The reference dashboards only work with the Prometheus and Grafana Loki backends. Future releases could support other backends.

1.5 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.

This release of *Observability Framework* provides a way to control the amount of logs that are generated and forwarded by *Connex* applications. Future releases could support control of metrics.

1.6 Security

Observability Framework provides a way to secure the telemetry data generated by the *Connex* applications and stored in the telemetry backends. Data in transit is secured by using the *RTI Security Plugins* and BASIC-Auth over HTTPS. Data at rest is secured by the third-party telemetry backends.

Chapter 2

Components

Connex Observability Framework consists of three RTI components:

- *RTI Monitoring Library 2.0* enables you to instrument a *Connex* 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 *Connex* 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 *Connex* application metrics, as well as display *Connex* 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](#).

RTI 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 2.1 shows a simple representation of how *Observability Framework* components work together.

2.1 Monitoring Library 2.0

Monitoring Library 2.0 includes the following key features:

- Collection and forwarding of *Connex* metrics and logs. Logs for DDS Security events will be supported in future releases.
- Configuration using a new [MONITORING QosPolicy \(DDS Extension\)](#). The QoS policy can be set programmatically or via XML.

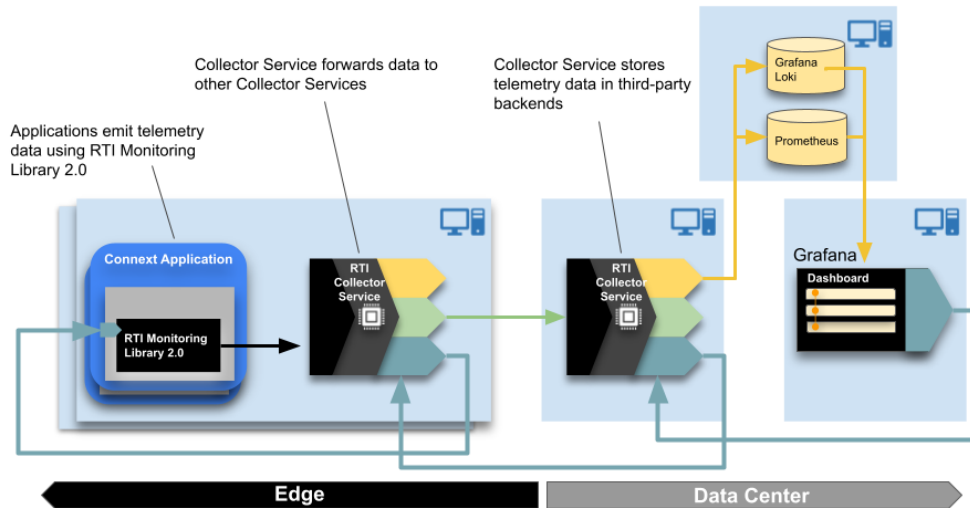


Figure 2.1: Observability Framework Components

- Runtime changes to the set of generated and forwarded telemetry data using remote commands from *Observability Collector Service*. In this release, the library only allows you to change the forwarding and generation log verbosity levels for a *Connex* application.
- 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](#).

2.2 Observability Collector Service

Observability Collector Service scalably collects telemetry data forwarded by *Monitoring Library 2.0* in a *Connex* application. *Collector Service* is distributed as a Docker™ image and can 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 a *Connex* application to other collector instances. This mode is not supported in the current release.

Observability Collector Service includes the following key features:

- Collecting and filtering telemetry data forwarded by *Connex* 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 *Connex* applications and other resources to which the commands are directed. This release only allows forwarding commands that change the logging verbosity of *Connex* applications. Future releases will support additional commands.

2.2.1 Storage Components

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

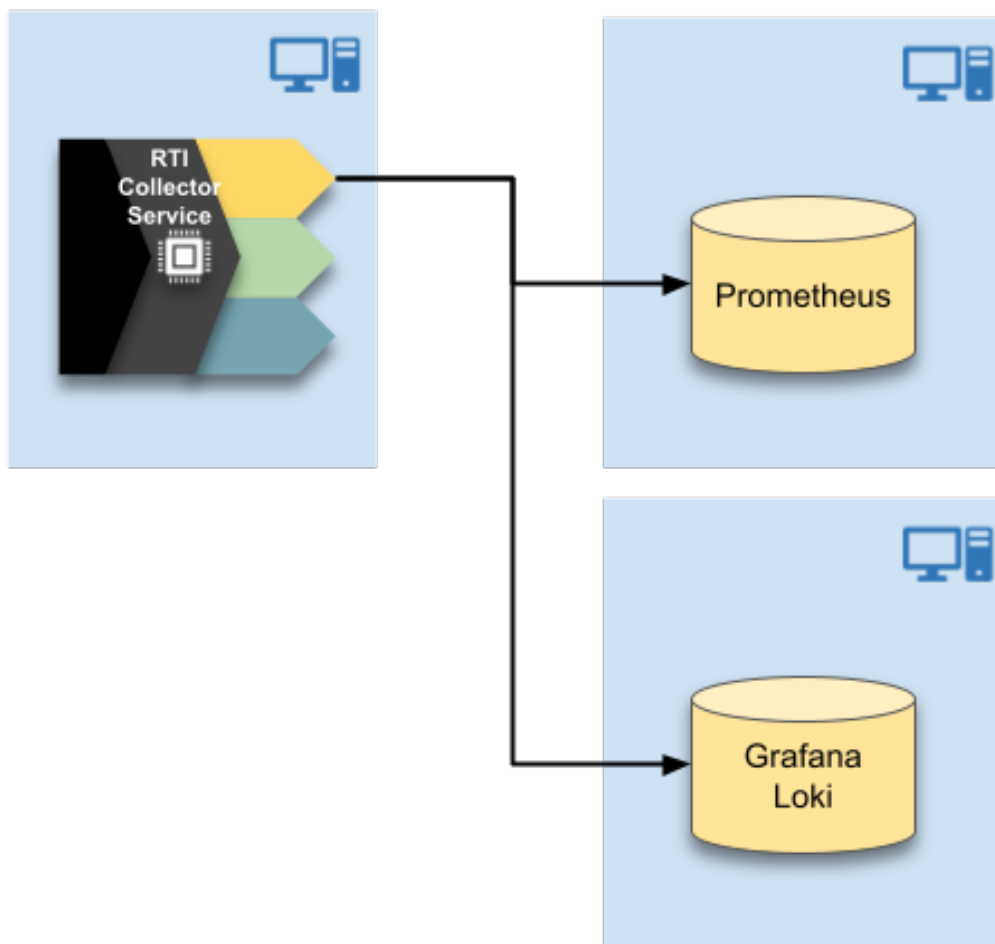


Figure 2.2: Native Integration

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

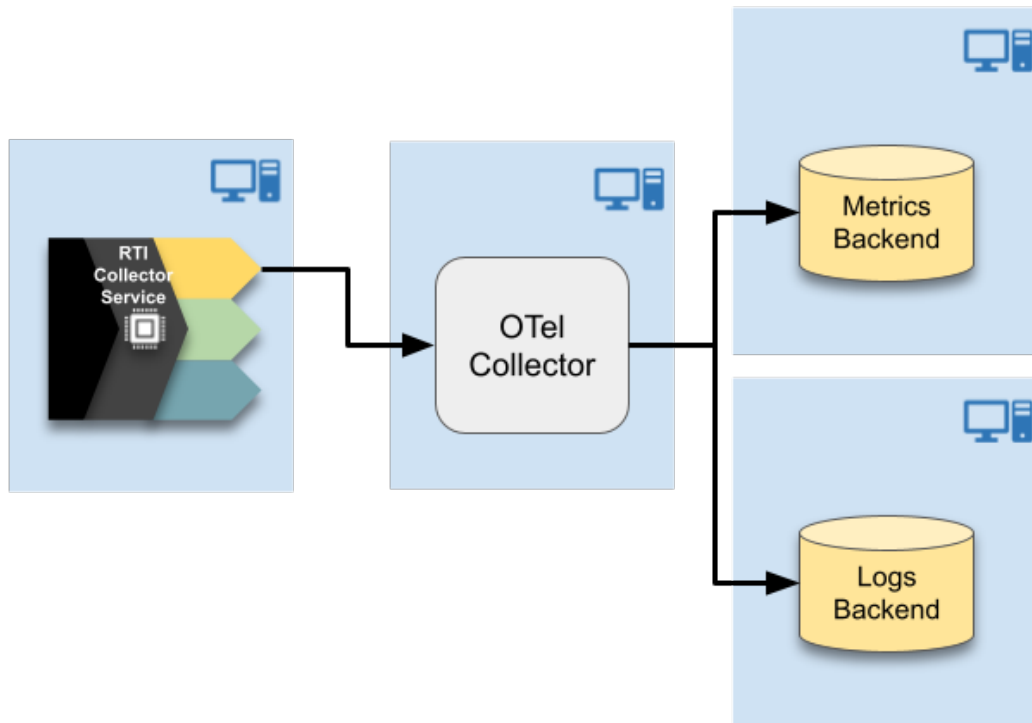


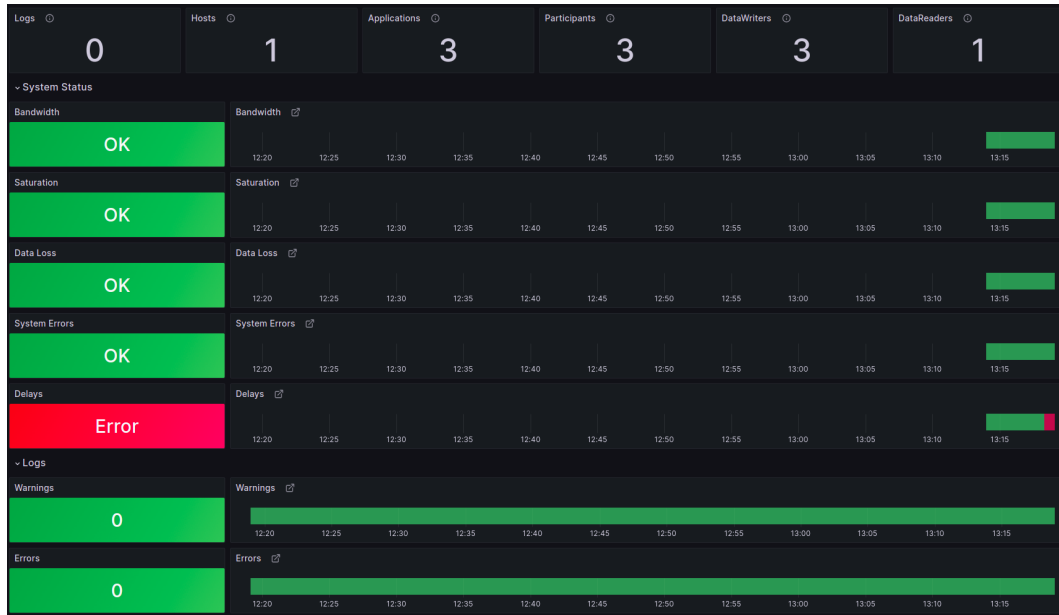
Figure 2.3: OpenTelemetry Integration

2.3 Observability Dashboards

A set of hierarchical Grafana dashboards sends 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.

The first layer of the Grafana dashboards provides a health status summary focused on five golden signals: Bandwidth, Saturation, Data Loss, System Errors, and Delays.

The top-level dashboard also provides access to the system logs and indicates the number of entities running in the system. To get additional details on error conditions, select any of the golden signals displaying an error.



2.4 How We Provide the Components

This section describes how *Observability Framework* components are provided in the current release and how RTI will provide them in future releases.

2.4.1 Monitoring Library 2.0

Monitoring Library 2.0 is provided as a shared and static library called *rtimonitoring2*. For details on how to use the library, refer to *Monitoring Library 2.0*.

2.4.2 Collection, Storage, and Visualization Components

Current Release

Docker Compose (Prepackaged)

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 NG-INTX (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 2.4 shows the secure *Observability Framework* deployment without OpenTelemetry Collector. The deployment uses Prometheus and Grafana Loki to store metrics and logs, respectively.

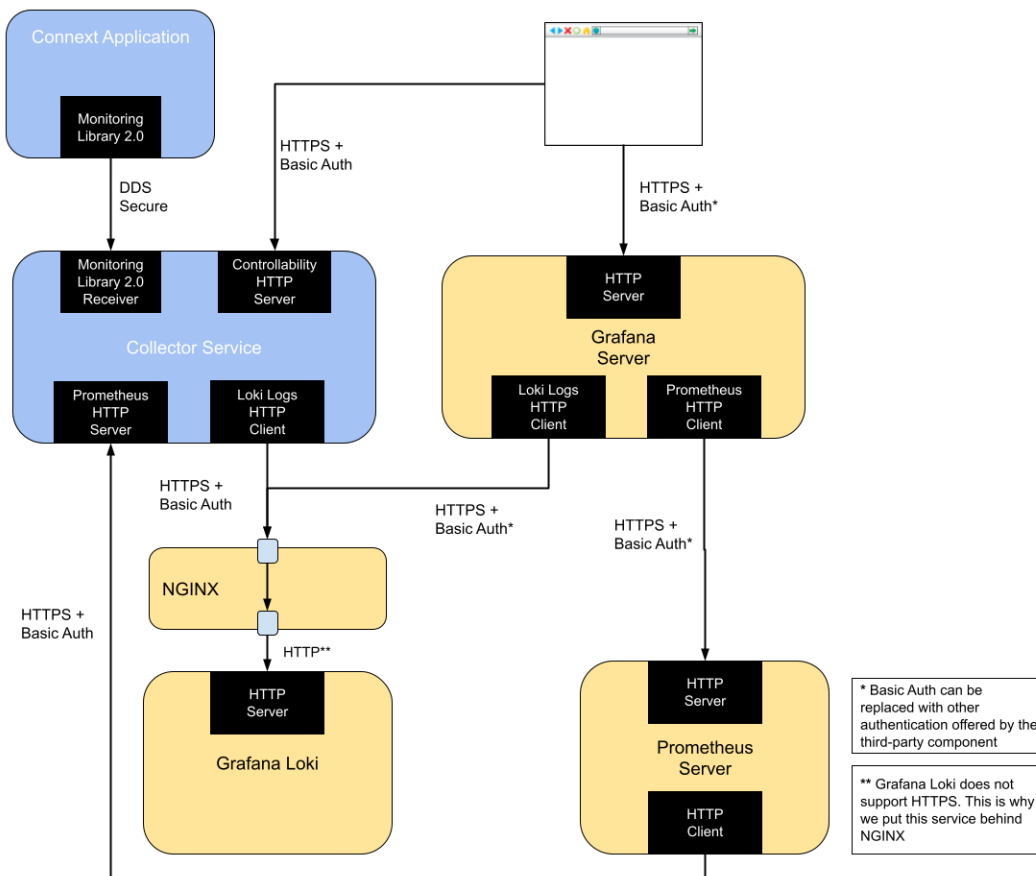


Figure 2.4: RTI Observability Framework without OpenTelemetry Collector

Figure 2.5 shows the secure *Observability Framework* deployment using OpenTelemetry Collector. The deployment uses OpenTelemetry Collector to store metrics and logs in Prometheus and Grafana Loki to store metrics and logs, respectively.

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

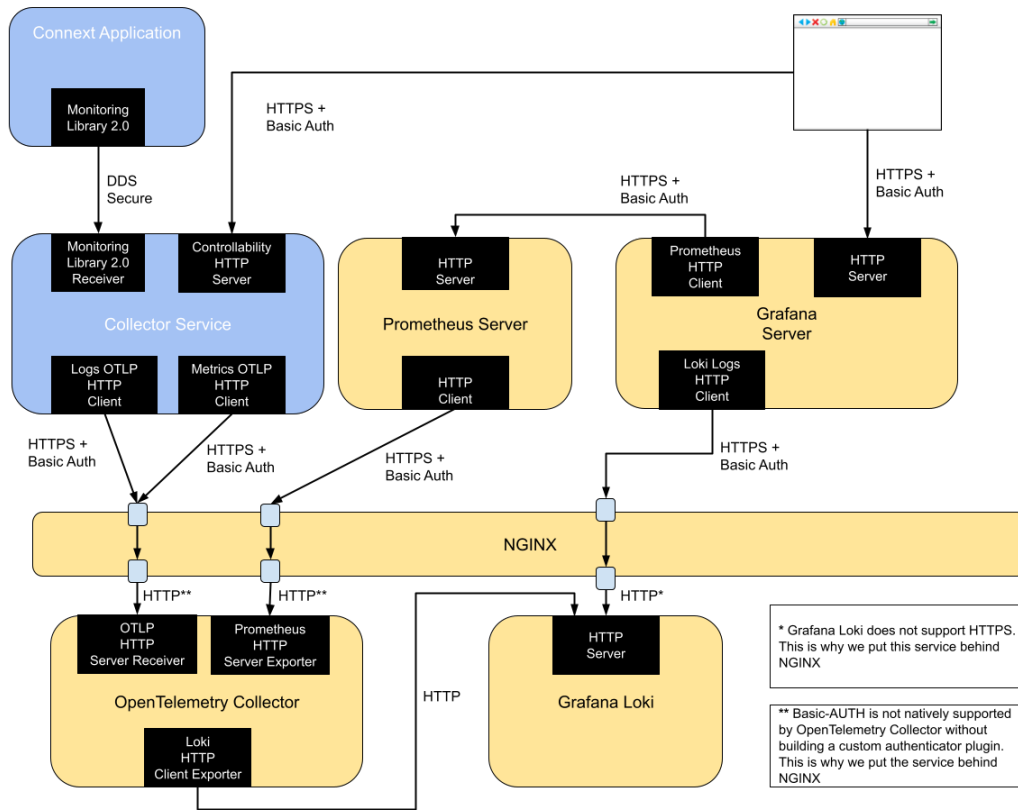


Figure 2.5: RTI Observability Framework with OpenTelemetry Collector

Docker (Separate Deployment)

As an alternative to the prepackaged Docker Compose provided by RTI, you can also run *Observability Framework* components standalone.

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.

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 2.1: 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](#).

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*.

Future releases

Executable

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

Kubernetes

In future releases, the Docker images for *Collector Service*, Prometheus, Grafana Loki, and Grafana can be deployed on an orchestrated platform such as Kubernetes. RTI will provide example deployment configurations for these deployments when they are made available.

Chapter 3

Deployments

This section describes *Observability Framework* deployment options.

3.1 Collector Service Deployments

3.1.1 Current Release

This release supports running *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 *Connex* system. This configuration is illustrated in Figure 3.1 and Figure 3.2

The deployments represented in Figure 3.1 and Figure 3.2 require running multiple instances of *Collector Service* where each *Connex* application configures *Monitoring Library 2.0* to connect to one of the *Collector Service* instances. This release distributes *Collector Service* as a Docker image; for details, refer to the *Docker (Separate Deployment)* section.

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.

Alternately, you can use a single layer deployment to run only one *Collector Service* instance for the *Connex* system, as illustrated in Figure 3.3. To deploy *Observability Framework* using a single collector (including the third-party components Prometheus and Grafana), use Docker Compose as described in the *Docker Compose (Prepackaged)* section.

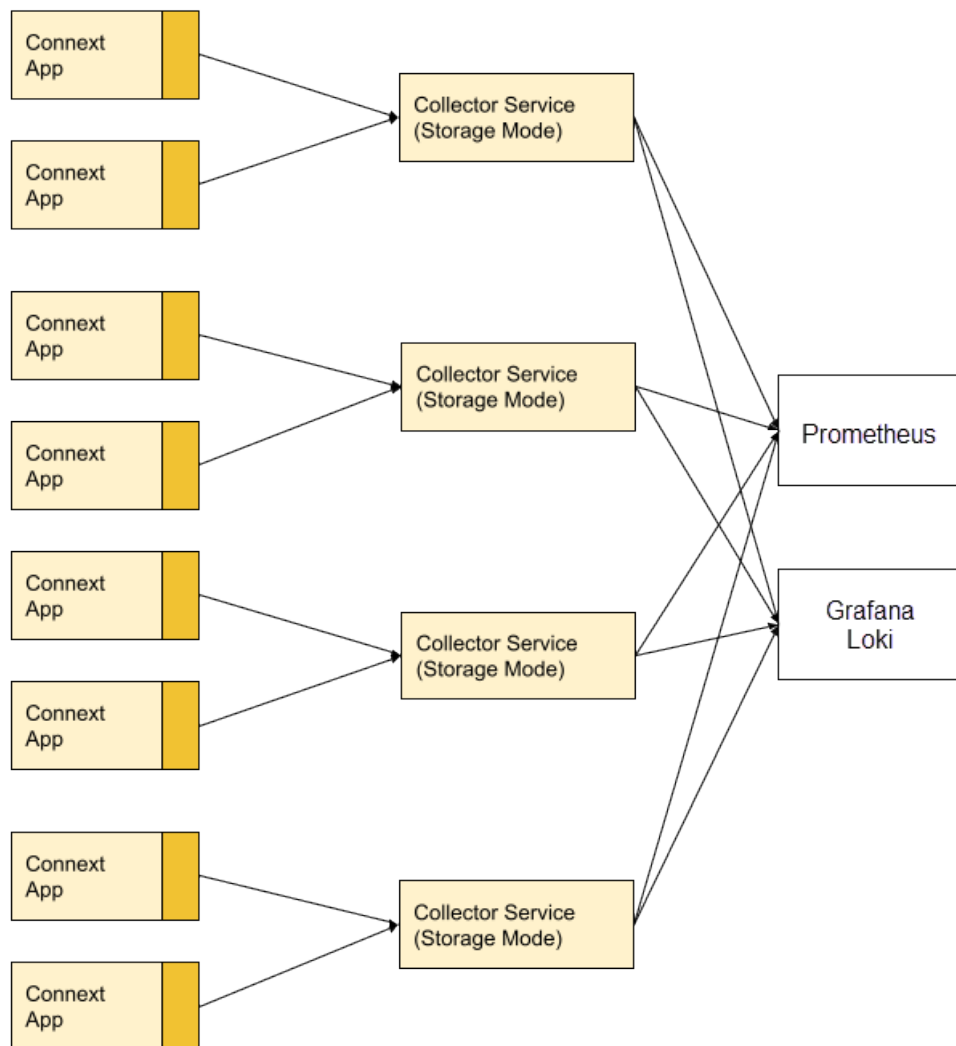


Figure 3.1: Single Layer Collector Deployment

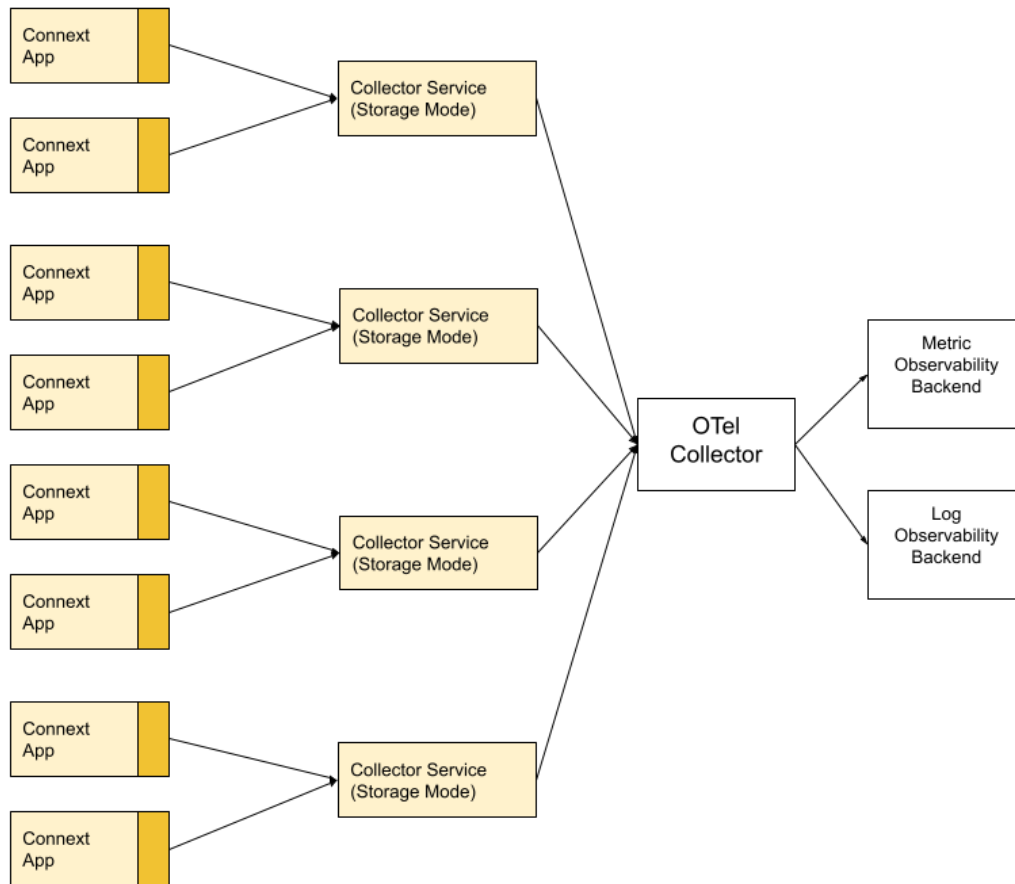


Figure 3.2: Single Layer Collector Deployment using OpenTelemetry Collector

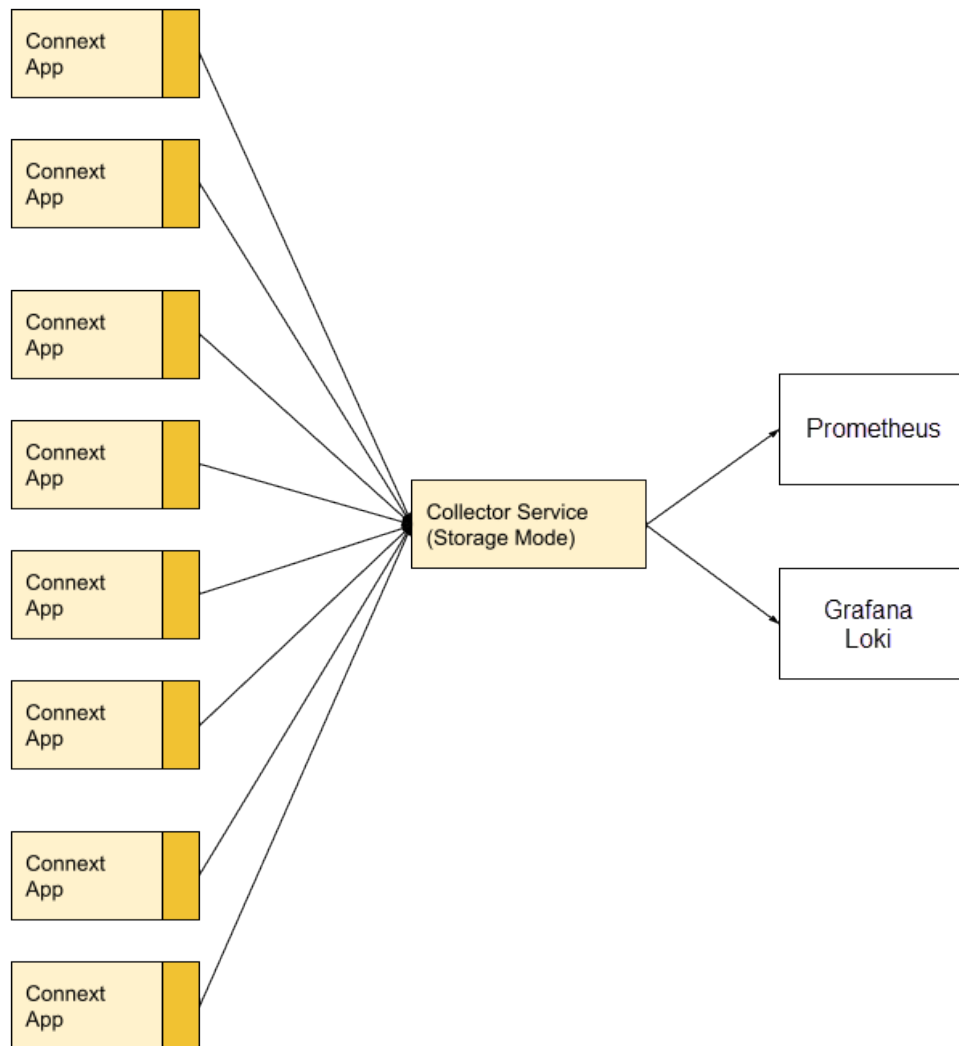


Figure 3.3: Single Collector Deployment

3.1.2 Future Releases

As you roll out telemetry data collection and distribution across all your *Connex* applications, *Observability Framework* must be deployed in a way that supports the additional load. A single layer *Collector Service* deployment, as shown in *Current Release*, may not scale sufficiently.

A better deployment option would be the layered deployment depicted in Figure 3.4 and Figure 3.5. In this option, you have multiple layers of *Collector Service* gathering, filtering, and forwarding the telemetry data produced by the *Connex* 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.

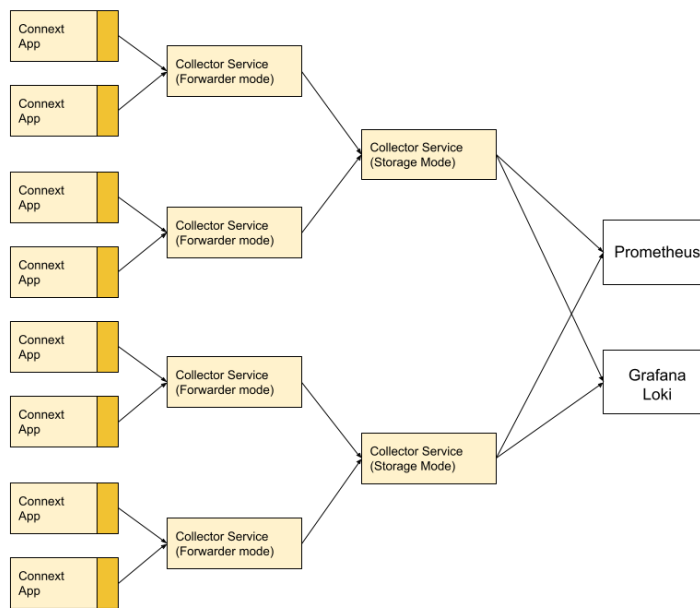


Figure 3.4: Layered Collector Deployment

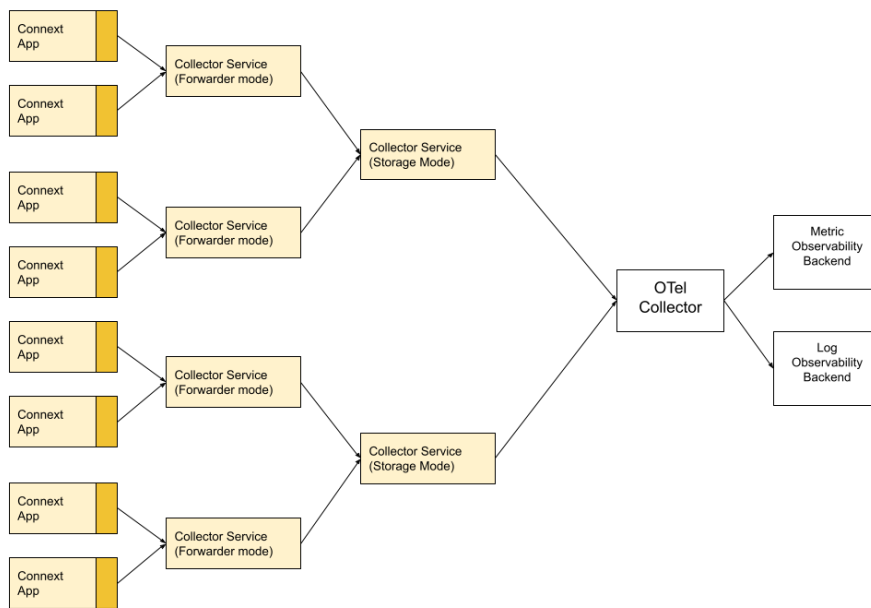


Figure 3.5: Layered Collector Deployment Using OpenTelemetry Collector

Chapter 4

Security

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

Figure 4.1 shows the *Observability Framework* security architecture when *Collector Service* is configured to store the telemetry data in Prometheus and Grafana Loki.

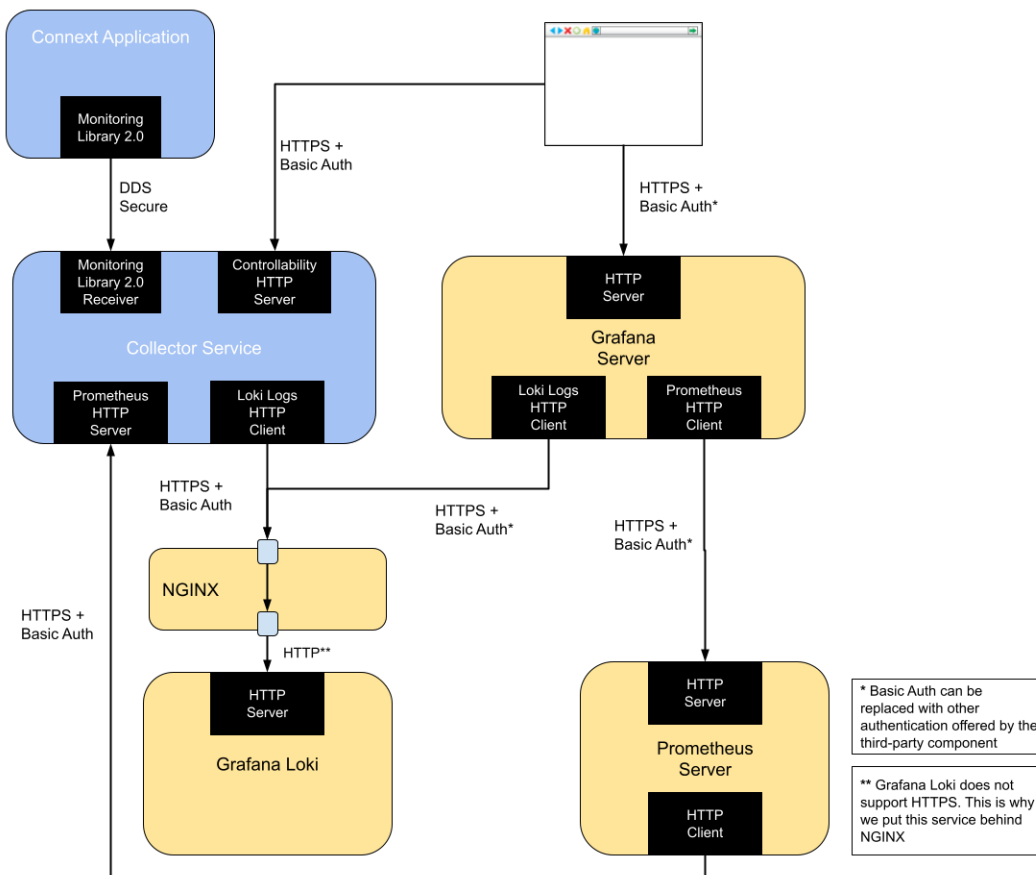


Figure 4.1: Security Architecture of RTI Observability Framework when using Prometheus and Grafana Loki

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 4.1 within a single host.

Figure 4.2 shows the *Observability Framework* security architecture when *Collector Service* is configured to forward the telemetry data to an OpenTelemetry Collector. This diagram does not show the Grafana server because *Observability Dashboards* are only provided for the Prometheus and Grafana Loki telemetry backends.

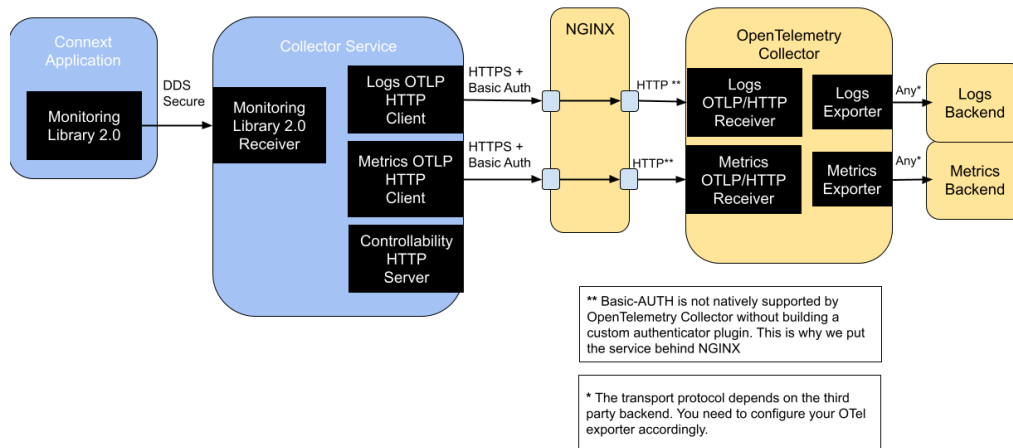


Figure 4.2: Security Architecture of RTI Observability Framework when using OpenTelemetry Collector

To facilitate testing and evaluation, you can install *Observability Framework* using *Docker Compose (Prepackaged)* to automatically run and deploy an OpenTelemetry Collector instance that stores the telemetry data in a local Prometheus and Grafana Loki as shown in Figure 4.3.

4.1 Secure Communication between Connex Applications and Collector Service

The exchange of telemetry data between a *Connex* application and *Collector Service* is secured by using the *RTI 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](#).

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

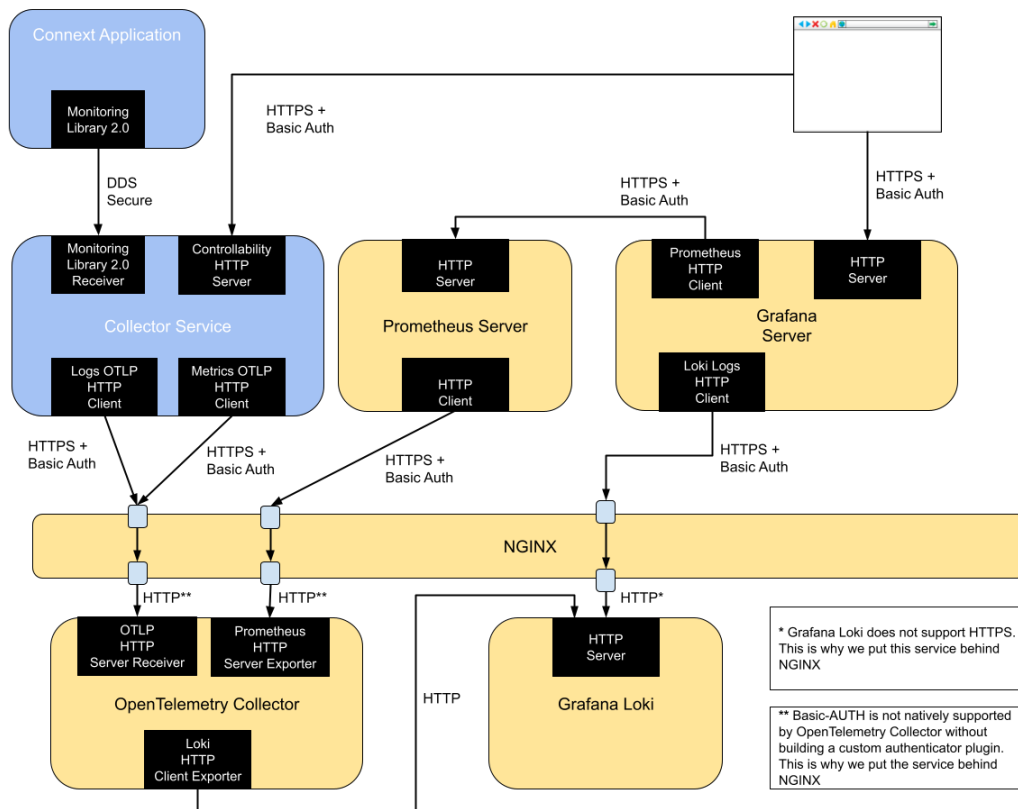


Figure 4.3: Security Architecture of the RTI Observability Framework when using OpenTelemetry Collector, Prometheus and Grafana Loki

```

{
  "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"
    }
  }
}

```

If you run *Collector Service* using the *Docker (Separate Deployment)* option, the security artifacts required to configure the *RTI Security Plugins* in *Collector Service* can be bind-mounted by providing 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

```

In addition, you need to set the `CFG_NAME` environment variable to one of the provided Docker image's built-in configurations that enable the *Security Plugins* in *Collector Service* (see *Docker (Separate Deployment)*).

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*.

4.2 Secure Communication with Collector Service's 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 *Connex Applications*. In this release, the REST API can only be used from *Observability Dashboards*.

Collector Service can be configured to store the telemetry data in a Prometheus backend. Prometheus collects metrics from targets by scraping metrics HTTP endpoints on these targets.

To configure the security of the HTTP servers started by *Collector Service*, you can follow these steps.

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"
    }
  }
}
```

If you run *Collector Service* using *Docker (Separate Deployment)*, the security artifacts to configure the HTTP servers running in *Collector Service* can be bind-mounted by providing the following options to the `docker run` command:

```
-v path/to/server.pem:/rti/security/https/server.pem
-v path/to/htdigest:/rti/security/https/htdigest
```

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 `server.pem` file must contain both a valid server certificate (`server_cert.pem`) and the corresponding private key (`server_key.pem`). The `htdigest` is a password file that contains the username and password for BASIC-Auth. The `htdigest` file is a password file that contains the Apache `htdigest`.

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

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.

4.3 Secure Communication with Third-party Components' 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.

To configure the security of the HTTP clients started by *Collector Service*:

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"
    }
  }
}
```

If you run *Collector Service* using *Docker (Separate Deployment)*, the security artifacts to configure the HTTP clients running in *Collector Service* can be provided by using the following options to the `docker run` command:

```
-e OBSERVABILITY_BASIC_AUTH_USERNAME=yourusername
-e OBSERVABILITY_BASIC_AUTH_PASSWORD=yourpassword
-v path/to/ca_cert.pem:/rti/security/https/rootCA.crt
```

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)*).

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.

4.4 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 with an application using *Monitoring Library 2.0*.
- 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*.

4.4.1 Generating DDS Security Artifacts

The DDS security artifacts are used to secure the exchange of telemetry data between a *Connex* application 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 *Connex* application and *Collector Service*.

For details on how to create/update DDS security artifacts, see [Generating and Revoking Your Own Certificates Using OpenSSL in Security Plugins Getting Started](#).

4.4.2 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 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 directory¹. 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 `observabilityRootCa.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 4.1: 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 4.1: htdigest Arguments

Parameter	Description	Value
-c	Create the passwdfile. If passwdfile already exists, it is deleted first.	-c
passwdfile	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.	host-name ("local-host")
username	The user name to create or update in passwdfile. If username does not exist in 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 in 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 5

Installing and Running Observability Framework

RTI Connex Observability Framework is not installed as part of *RTI Connex Professional Edition*. *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 are two *Observability Framework* packages, as outlined in Table 5.1.

Table 5.1: Observability Framework Packages

Package Name	Package Contents	Use Case	Supported Platform
rti_observability-7.2.0-target-platform>.rtipkg	This target package installs <i>Monitoring Library 2.0</i> in your target platform(s).	Install this package if you need to forward telemetry data generated by DDS applications on the target platform.	<i>Monitoring Library 2.0</i> is supported in all <i>Connex</i> platforms.
rti_observability-7.2.0-host-x86_64-linux-pkg	The host package contains the files required to run the <i>Observability Framework</i> collection, storage, and visualization components using Docker and Docker Compose. This package also includes <i>Observability Framework</i> 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 <i>Supported Docker Compose Environments</i> .

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

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 *Connex* applications.

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

5.1 Installing the Target Package

There are two ways to install *Monitoring Library 2.0*: using *RTI Launcher* or the `rtipkginstall` command-line utility.

Note: For detailed information about *Monitoring Library 2.0*, see the [MONITORING QosPolicy \(DDS Extension\)](#).

5.1.1 Install from RTI Launcher

To install the *Observability Framework* target 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>-target-<your platform>.rtipkg` file.
4. Click **Install**.

5.1.2 Install from the Command Line

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

```
$ <installdir>/bin/rtipkginstall /<path-to-observability-framework-file>/rti_
↪observability-<version>-target-<your platform>.rtipkg
```

5.2 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.

5.2.1 Prerequisites

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

- *Connex* 7.2.0. For installation instructions, see the [RTI Connex 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*.

5.2.2 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**.

5.2.3 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
```

5.3 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](#)

5.3. Configuring, Running, and Removing Observability Framework Components Using32 Docker Compose

- [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, RTI Observability Collector Service sends the data to OpenTelemetry Collector.
- [NGINX](#) [Optional]: when using security the Observability Framework runs an 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.

5.3.1 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.

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

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

The following table describes all the fields you can configure:

Table 5.2: 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 <i>Observability</i> data.	int	2
lgpStackConfig			
lgpStackConfig.grafanaPort	The Grafana server port.	int	3000
lgpStackConfig.prometheusPort	The Prometheus server port.	int	9090
lgpStackConfig.lokiPort	The Loki server port.	int	3100
collectorConfig			
collectorConfig.prometheusExporterPort	The Collector Service client port for Prometheus data.	int	19090
collectorConfig.controlPort	The Collector Service server port for control commands.	int	19098
collectorConfig.rtwPort	WAN port used by <i>RTI Real-Time Wan Transport</i> .	int	30000
otelConfig			
otelConfig.otelHttpReceiverPort	Port used to configure OpenTelemetry http receiver port.	int	
securityConfig			
securityConfig.basicAuthUsername	Username used for HTTP Basic authentication.	String	
securityConfig.basicAuthPassword	Password used for HTTP Basic authentication.	String	
securityConfig.httpsSecurity			
securityConfig.httpsSecurity.caCertificate	Certificate authority (CA) used to verify the SSL certificates signed by this CA.	String	
securityConfig.httpsSecurity.serverCertificate	Server Certificate used with HTTPs.	String	
securityConfig.httpsSecurity.serverKey	Server Key for the Server Certificate.	String	
securityConfig.ddsSecurity			
securityConfig.ddsSecurity.identityCaCertificate	File to be passed as <code>dds.sec.auth.identity_ca</code>	String	
securityConfig.ddsSecurity.permissionsCaCertificate	File to be passed as <code>dds.sec.access.permissions_ca</code>	String	
securityConfig.ddsSecurity.identityCertificate	File to be passed as <code>dds.sec.auth.identity_certificate</code>	String	
securityConfig.ddsSecurity.identityKey	File to be passed as <code>dds.sec.auth.private_key</code>	String	
securityConfig.ddsSecurity.signedPermissionsFile	File to be passed as <code>dds.sec.access.permissions</code>	String	
securityConfig.ddsSecurity.signedGovernanceFile	File to be passed as <code>dds.sec.access.governance</code>	String	

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

All the fields are optional except `hostname`. If `securityConfig` is provided, then all its fields must be provided as well.

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 Connex Observability Framework[Y/
n]? Y

Generating configuration for the Connex 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_connex_dds-7.2.0/resource/app/app_
support/observability/requirements.txt
```

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

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

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 Connex Observability Framework[Y/
↳ n]?
```

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

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

Generating configuration for the Connex Observability Framework

2023-07-19 19:28:10,277 - exporter - INFO - Config: {
  "hostname": "localhost",
  "observabilityDomain": 2,
  "otelConfig": null,
  "lppStackConfig": {
    "grafanaPort": 3000,
    "prometheusPort": 9090,
    "lokiPort": 3100
  },
  "collectorConfig": {
    "prometheusExporterPort": 19090,
    "controlPort": 19098,
    "rtwPort": null
  }
}
```

(continues on next page)

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

(continued from previous page)

```

    },
    "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 Connex 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 ...
*
*****

```

5.3.2 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 5.3: License Requirements Table

Do you need to secure telemetry data exchanged between applications and <i>Collector Service</i> using <i>RTI Security Plugins</i> ?	Do you need to send telemetry data to <i>Collector Service</i> over the WAN using <i>RTI Real-Time WAN Transport</i> ?	Required License
NO	NO	Connex Professional
YES	NO	Connex Secure
YES	YES	Connex Anywhere
NO	YES	Connex Anywhere

For instructions on how to install a license file, see [Installing the License File](#) in the *RTI Connex 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.

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

```

$ rtiobservability -i

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

Initializing and running the Connex 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.

5.3.3 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	STATUS	NAMES	COMMAND
6651d7ed9810	prom/prometheus:v2.37.5	Up 5 minutes	prometheus_observability	"/bin/prometheus --c..."
25050d16b1b5	grafana/grafana-enterprise:9.2.1-ubuntu	Up 5 minutes	grafana_observability	"/run.sh"
08611ea9b255	rticom/collector-service:<version>	Up 5 minutes	collector_service_observability	"/rti_connex_dds-7..."
55568de5120f	grafana/loki:2.7.0	Up 5 minutes	loki_observability	"/usr/bin/loki --con..."

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_connex_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.

5.3.4 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.

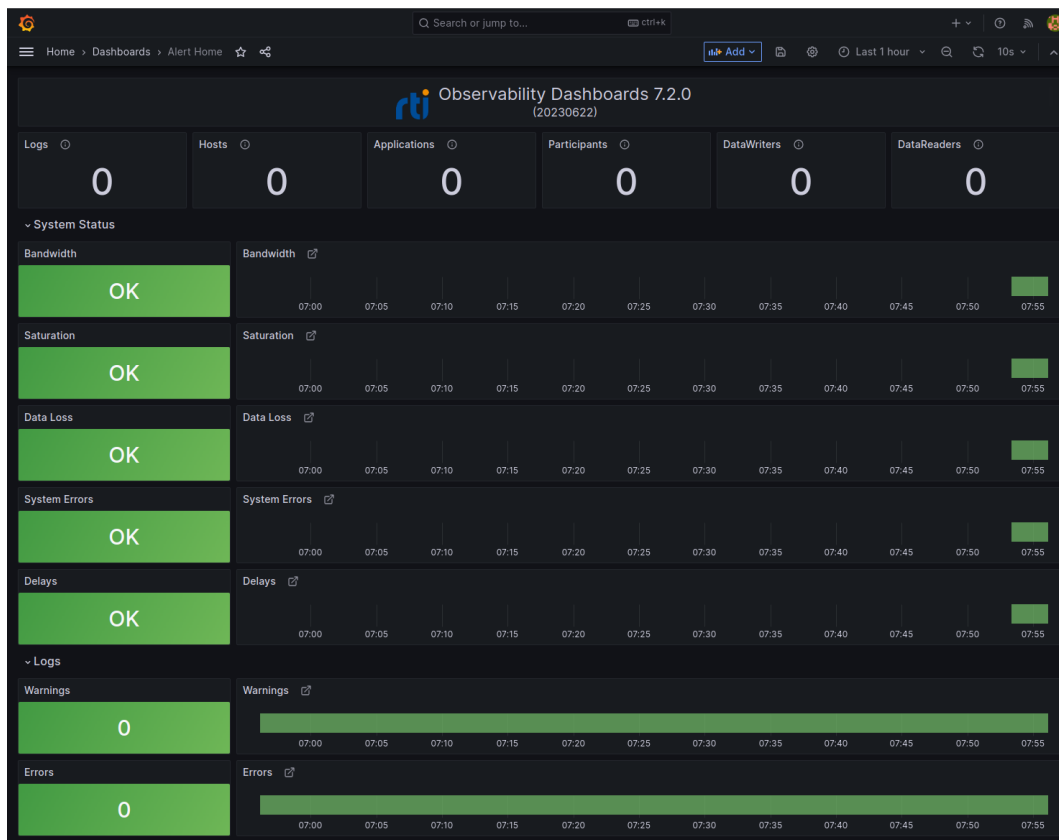
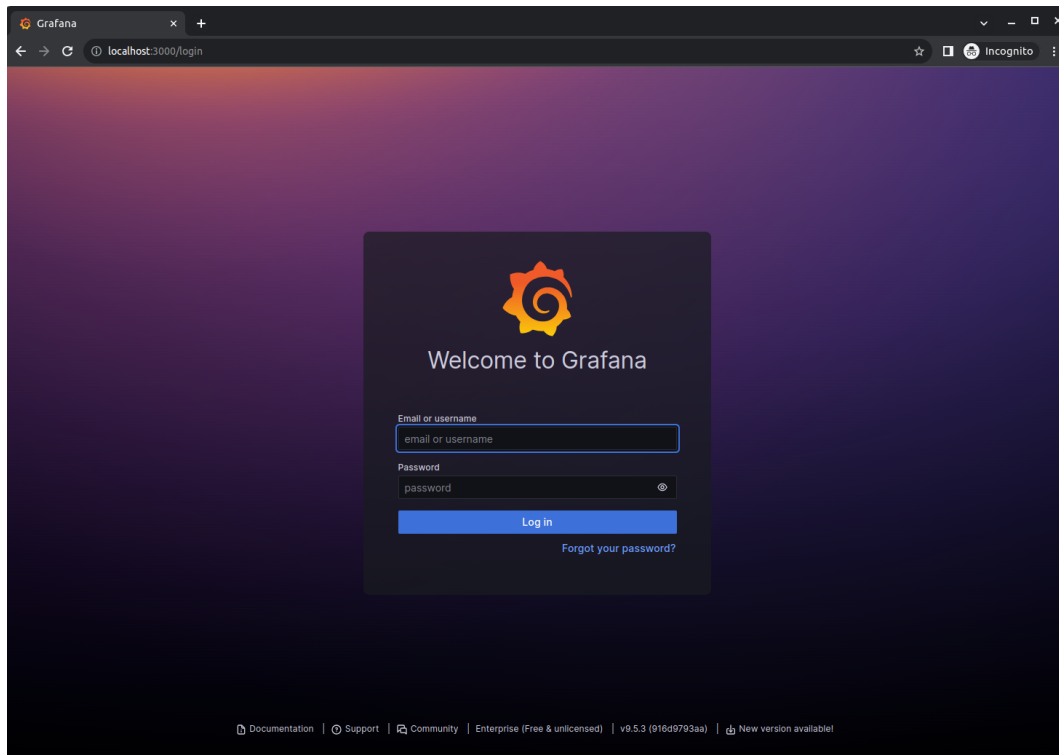
Once you are logged in you will see the **RTI Alert Home** dashboard.

Configuration Options

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

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.

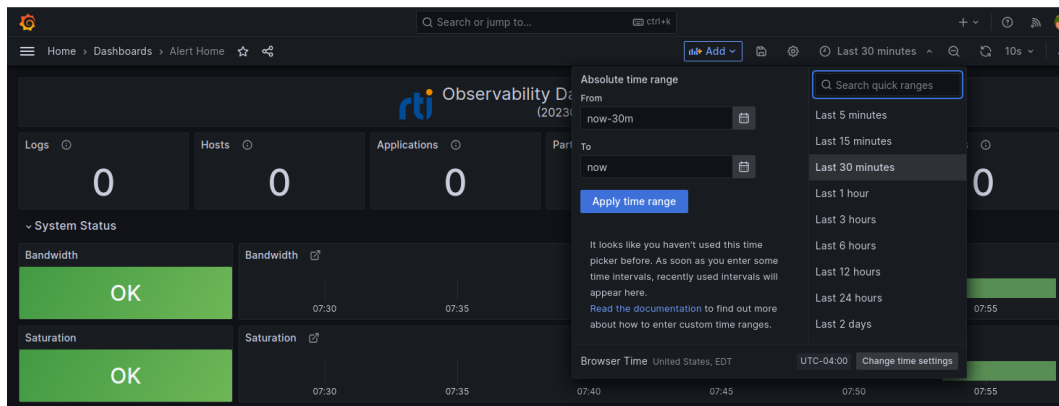


5.3. Configuring, Running, and Removing Observability Framework Components Using42 Docker Compose

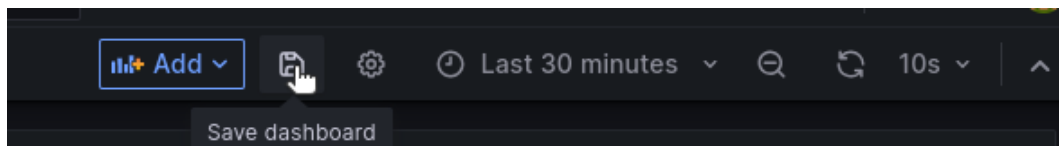
Change the Default Time Range (Optional)

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.

5.3.5 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`.

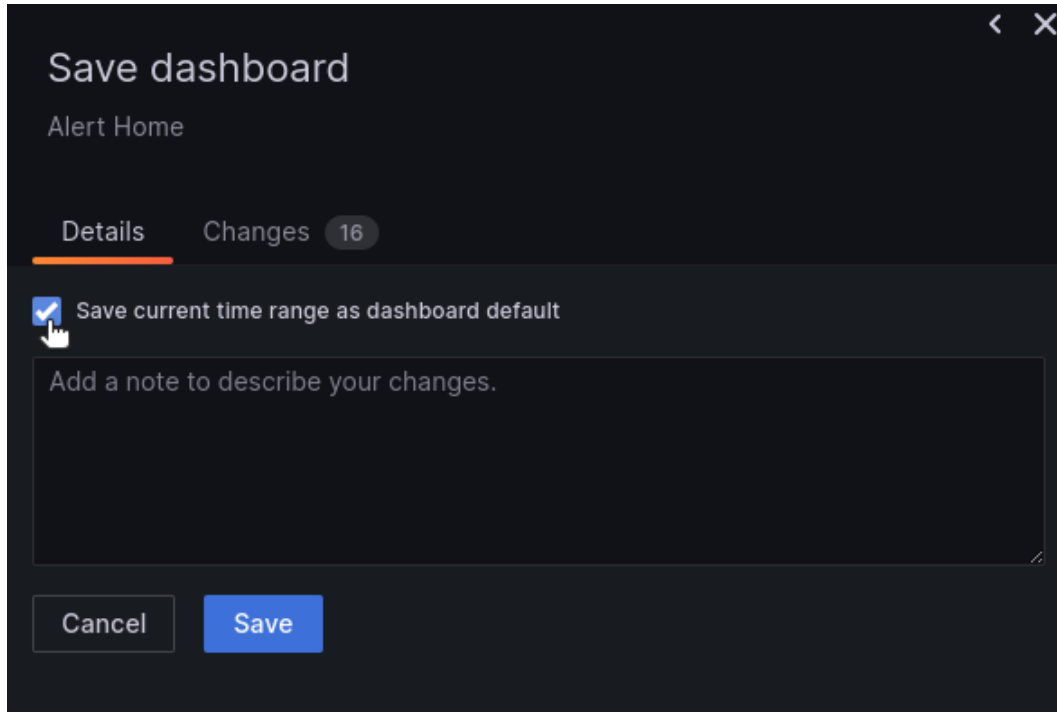
```
$ rtiobservability -t

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

Terminating the running Connex Observability Framework
```

(continues on next page)

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose



(continued from previous page)

```
[+] 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.

5.3.6 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 Connex Observability Framework
```

(continues on next page)

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

(continued from previous page)

```
[+] 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*.

5.3.7 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.

*****
*
* You have requested to clean up and remove the existing Connex_
↪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 Connex_
↪Observability Framework[y/N]?
```

(continues on next page)

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

(continued from previous page)

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 Connex
↳Observability Framework[y/N]? n

Cleaning up and removing the existing Connex 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 Connex
↳Observability Framework[y/N]? y

Cleaning up and removing the existing Connex 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
```

5.3.8 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

5.3. Configuring, Running, and Removing Observability Framework Components Using Docker Compose

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

Chapter 6

Getting Started Guide

6.1 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.

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 *Connex* applications.

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

6.1.1 Applications

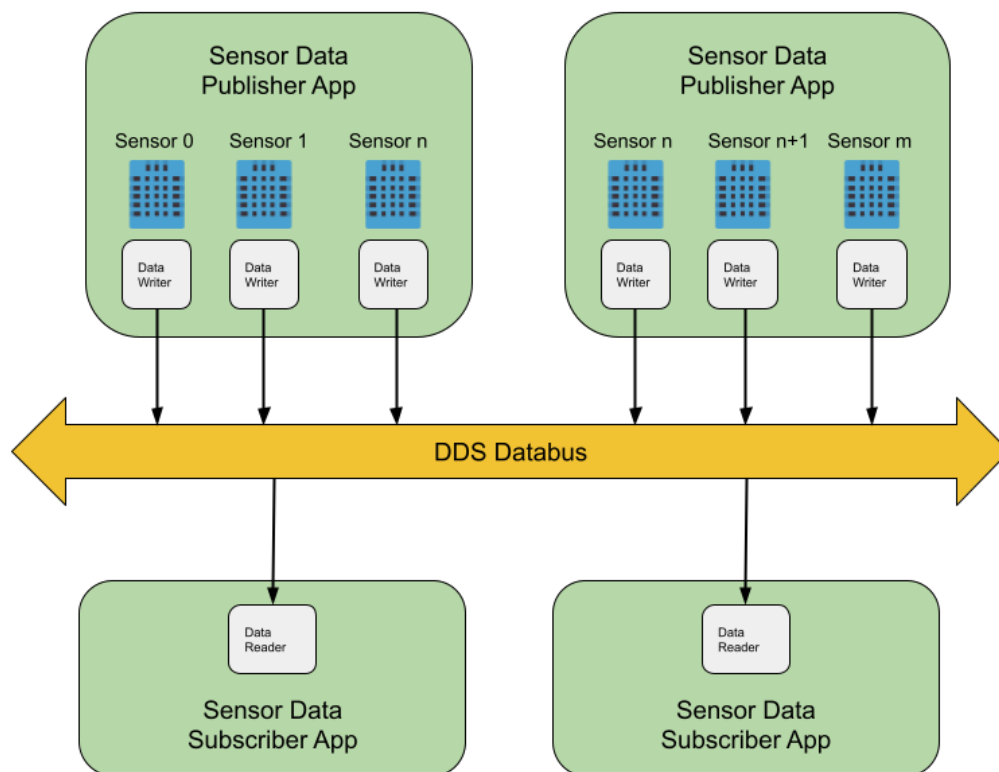
The example consists of two applications:

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

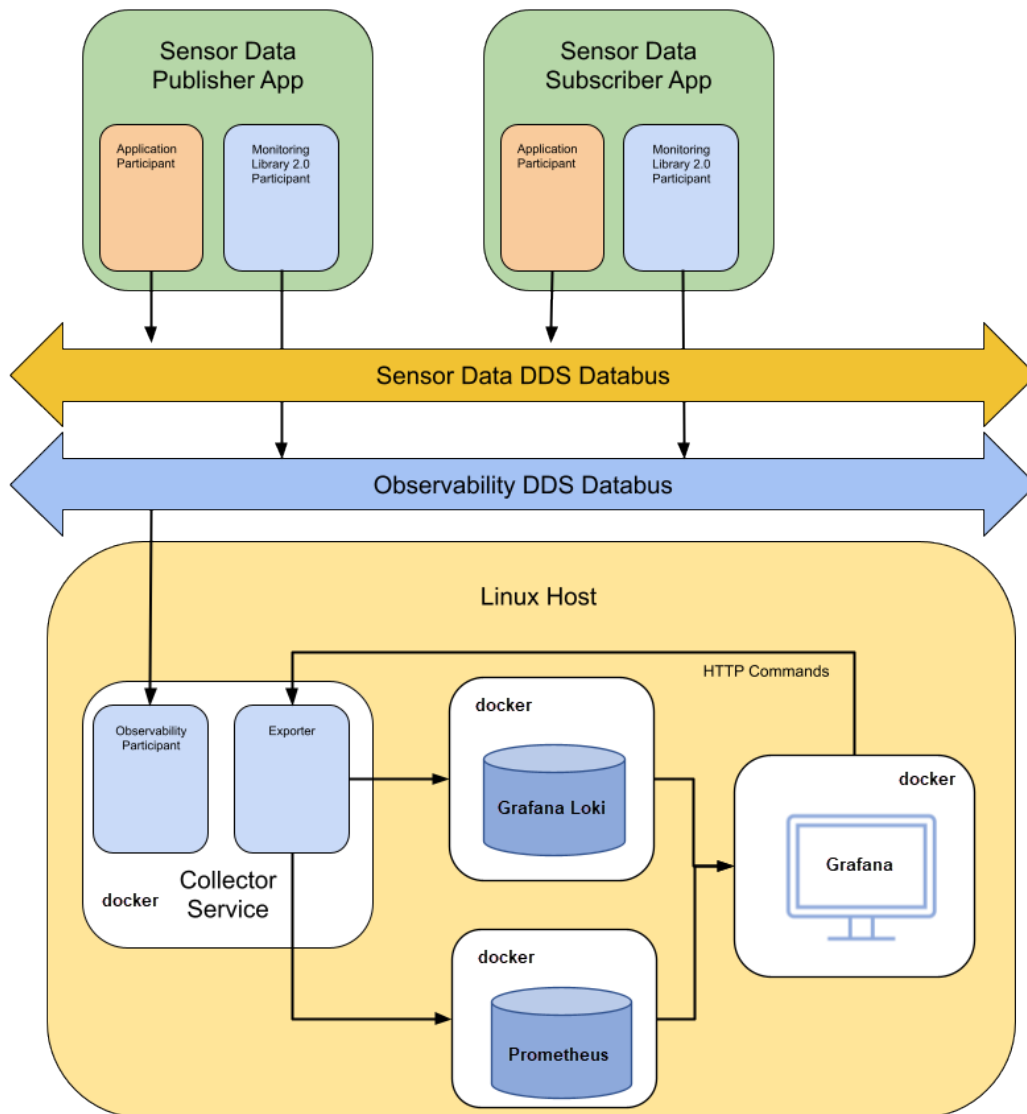
You can run multiple publishing and subscribing applications in the same host, or in multiple hosts, within a LAN. Each publishing application can handle multiple sensors, and each subscribing application subscribes to all sensors.

To learn more about the publish/subscribe model, refer to [Publish/Subscribe](#) in the *RTI Connex 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.



6.1.2 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 Connex Getting Started Guide*.

6.1.3 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 Connex Getting Started Guide* for more information on instances.

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

6.1.4 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 6.1: Publishing Application

Parameter	Data Type	Description	De- fault
-n, --application-name	<str>	Application name	Sensor-Publisher- <code><init_sensor_id></code>
-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	<int>	Collector service peer	local-host
-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 6.2: Subscribing Application

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

6.2 Before Running the Example

6.2.1 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
```

Windows

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

<installdir> refers to the installation directory for *Connex*.

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
```

6.2.2 Compile the Example

Monitoring Library 2.0 can be used in three different ways:

- **Dynamically loaded:** This method requires that the `rtmonitoring2` shared library is in the library search path.
- **Dynamic linking:** The application is linked with the `rtmonitoring2` shared library.
- **Static linking:** The application is linked with the `rtmonitoring2` static library.

You will compile the example using *Connex* 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.

6.2.3 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 *RTI Real-Time WAN Transport*.

In either of these methods, you can use a secure or non-secure configuration.

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 configurations supported by *Observability Framework*. These configurations 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 and network environment (LAN or WAN).

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 6.3 lists the four LAN configurations supported by *Observability Framework*.

Table 6.3: 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,
    "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
  },
  "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": {
```

(continues on next page)

(continued from previous page)

```

    "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"
  }
}

```

Example WAN configurations

Table 6.4 lists the four WAN configurations supported by *Observability Framework*.

Table 6.4: Docker Container WAN Operation Modes

Configuration Name	Network	Data Storage	Secu- rity
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,

```

(continues on next page)

(continued from previous page)

```

        "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",

```

(continues on next page)

(continued from previous page)

```

        "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* 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": {

```

(continues on next page)

(continued from previous page)

```

        "prometheusExporterPort": 19090,
        "controlPort": 19098,
        "rtwPort": 30000
    },
    "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",
            "signedGovernanceFile": "<rti_workspace_dir>/examples/dds_
↪security/xml/signed/signed_ObservabilityGovernance.p7s"
        }
    }
}

```

6.2.4 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 *RTI 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*.

6.3 Running the Example

Before running the example, select the appropriate values for the following deployment command-line options as needed:

Table 6.5: Optional Command-Line Parameters

Parameter	Description	Default Value
<code>--observability-domain</code>	Use this command-line option if you want to overwrite the default domain ID used by <i>Monitoring Library 2.0</i> to send telemetry data to <i>Observability Collector Service</i> .	2
<code>--collector-peer</code>	If you run <i>Observability Collector Service</i> 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 <code>udp4_wan://10.56.78.89:16000</code> (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 instructions in this section assume that you are running the applications and the Docker containers used by *Observability Framework* on the same host with the default observability domain.

6.3.1 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.

Note that at this point, no DDS applications are running.

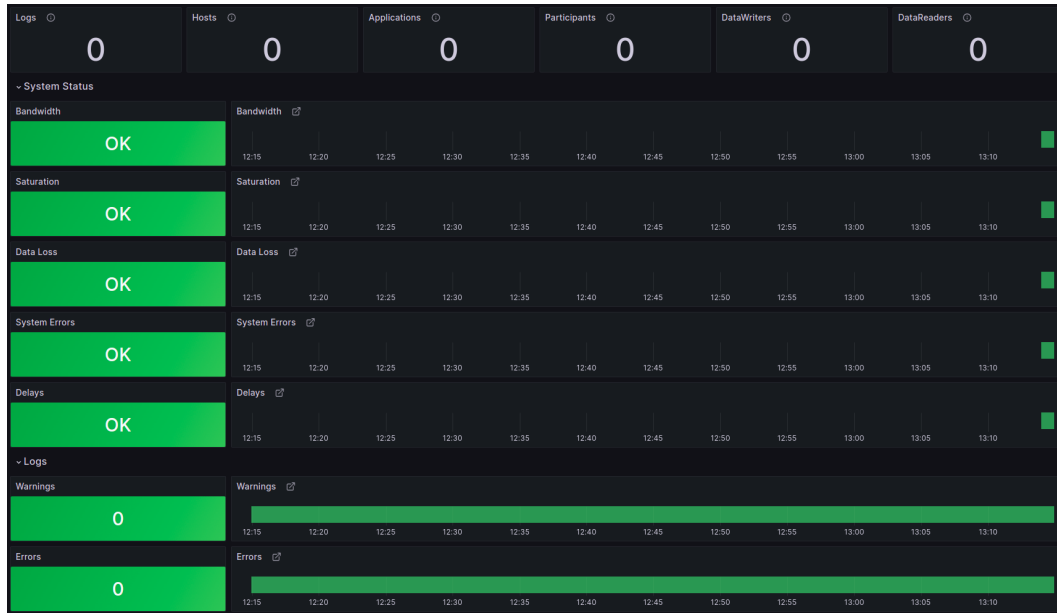
2. From the example directory, open two terminals and start two instances of the application that publishes temperature sensor data. The command and resulting output for each instance are shown below. The `-i` parameter specifies the sensor ID that will be used. The `-n` parameter assigns a name to the application. This name will be used when sending the commands in *Change the Application Logging Verbosity*. Note the addition of the `-p` parameter to enable security for secure configurations.

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 *****
```

(continues on next page)



(continued from previous page)

```
*****
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
  Security: true
Running with QOS:
```

(continues on next page)

(continued from previous page)

```
Secure_Temperature_Profile_With_Monitoring2_Over_LAN
Command>
```

The second instance creates one sensor.

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
```

(continues on next page)

(continued from previous page)

```

*****
***** 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:DISC]NDDS_Transport_UDPv4_Socket_bind_with_ip:0X1EE6 in use
WARNING [0x01017774,0xFF40EEF6,0xEC566CA8:0x000001C1{Domain=2}
→|ENABLE|LC:DISC]NDDS_Transport_UDPv4_SocketFactory_create_receive_
→socket:invalid port 7910
WARNING [0x01017774,0xFF40EEF6,0xEC566CA8:0x000001C1{Domain=2}
→|ENABLE|LC:DISC]NDDS_Transport_UDP_create_rcvresource_rrEA:!create socket
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature_
→DomainParticipant,Domain=57}|ENABLE|LC:DISC]NDDS_Transport_UDPv4_Socket_
→bind_with_ip:0X549C in use
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature_
→DomainParticipant,Domain=57}|ENABLE|LC:DISC]NDDS_Transport_UDPv4_
→SocketFactory_create_receive_socket:invalid port 21660
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature_
→DomainParticipant,Domain=57}|ENABLE|LC:DISC]NDDS_Transport_UDP_create_
→rcvresource_rrEA:!create socket

```

(continues on next page)

(continued from previous page)

```
WARNING [0x010175D0,0x7A41F985,0xF3813392:0x000001C1{Name=Temperature_
↳DomainParticipant,Domain=57}|ENABLE|LC:DISC]DDS_DomainParticipantDiscovery_
↳add_peer:no peer locators for: peer descriptor(s) = "builtin.shmem://",
↳transports = "", enabled_transports = ""
```

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

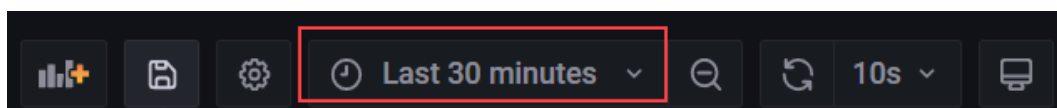


The Grafana main dashboard pictured above indicates that the system is healthy. You may see warnings in the log section related to the reservation of communication ports. These warnings are expected. You can select the Warnings panel to visualize them.

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

6.3.2 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**.

Absolute time range

From

To

[Apply time range](#)

It looks like you haven't used this time picker before. As soon as you enter some time intervals, recently used intervals will appear here.

[Read the documentation](#) to find out more about how to enter custom time ranges.

Search quick ranges

- Last 5 minutes
- Last 15 minutes
- Last 30 minutes
- Last 1 hour**
- Last 3 hours
- Last 6 hours
- Last 12 hours
- Last 24 hours
- Last 2 days
- Last 7 days

Browser Time United States, EST UTC-05:00 [Change time settings](#)

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.

6.3.3 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 Connex Getting Started Guide* for basic information, or [DEADLINE QoSPolicy](#) in the *RTI Connex Core User's Manual* for detailed information.

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

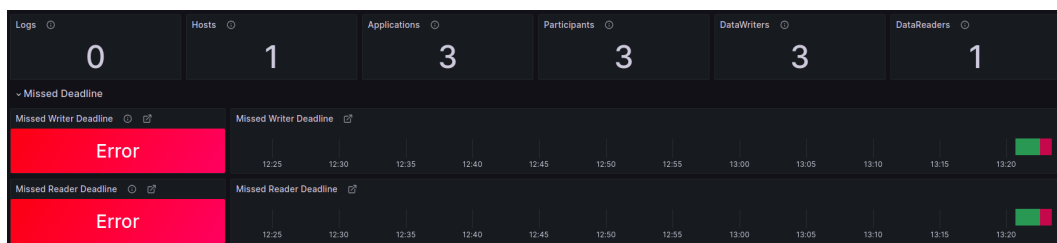
```
Command> stop 0
```

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 were generated by both the DataWriters associated with the sensors and the DataReader expecting sensor data at a specific rate. 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 providing deadline information. In this case we see three entities, or DataWriters, each associated with a sensor. We also see the first 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”. This name is set using the [EntityName QoS Policy](#) when creating a 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.

Missed Writer Deadline						
DataRow Name	Topic Name	Registered Type Name	Host Name	Domain Id	DDS GUID	Status
SensorPublisher_1/Temperatur...	Temperature	Temperature	sugarloaf	57	0101BD17.3B55DC3B.7AE004C...	OK
SensorPublisher_1/Temperatur...	Temperature	Temperature	sugarloaf	57	0101BD17.3B55DC3B.7AE004C...	ERROR
SensorPublisher_2/Temperatur...	Temperature	Temperature	sugarloaf	57	0101FF9B.23468324.ED78EEB...	OK

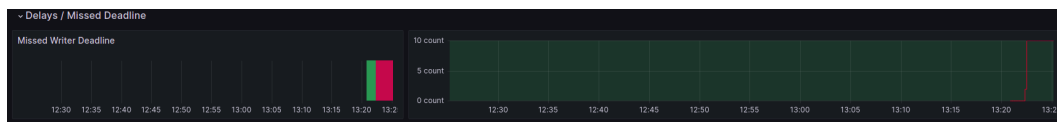
The fourth and last level of the Grafana dashboard provides detailed information about an individual entity, including location-related information such as Host Name and Process Id.

Links

/SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0

Participant Name SensorPublisher_1/Temperature DomainParticipant		Host Name sugarloaf	Process Id 887546	Domain Id 57	Platform x64Linux4gcc7.3.0	Product Version 7.2.0.0
DataRow Name SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0			Topic Name Temperature	Registered Type Name Temperature	DDS GUID 0101BD17.3B55DC3B.7AE004C2.80000002	
Log Errors 0	Log Warnings 0					

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. However, part of the Delay panel is still red. Depending on the selected dashboard time range, the sensor was likely unhealthy for part of the time displayed..

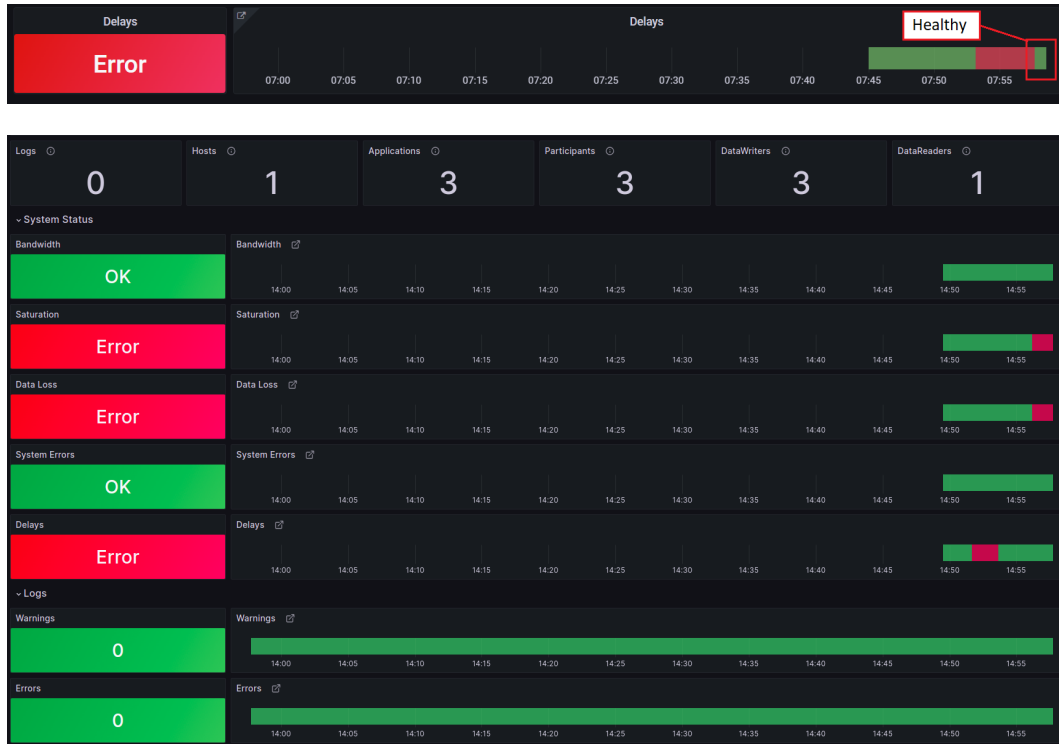
6.3.4 Simulate Slow Sensor Data Consumption

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.

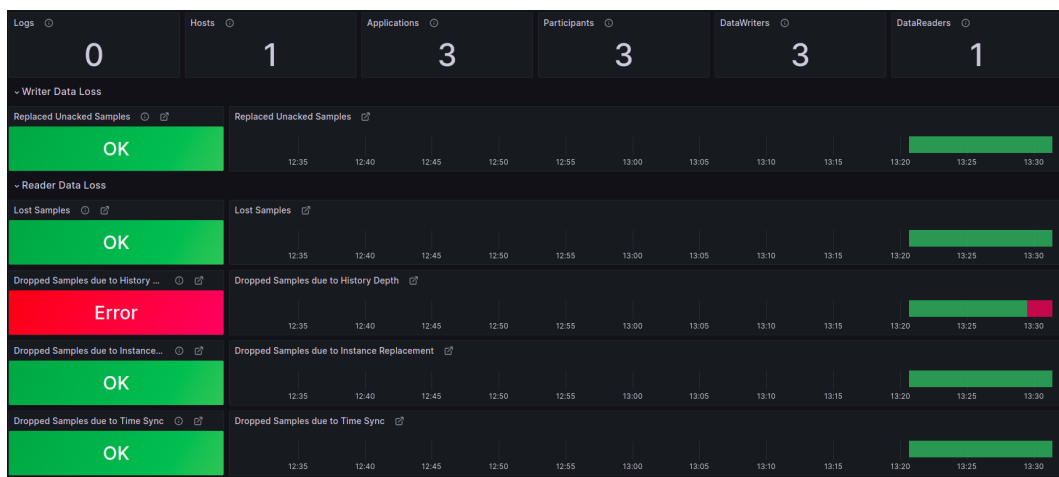
This scenario simulates a problem with the subscribing application; a bug in the application logic makes it slow in processing sensor data. To test this failure, enter the `slow_down` command in the Temperature_subscriber instance:

```
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 DataReader may consume 100% of the CPU due to the 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.



6.3.5 Simulate Time Synchronization Failures

The DataReaders created by the subscribing applications expect that clocks are synchronized. 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>
```

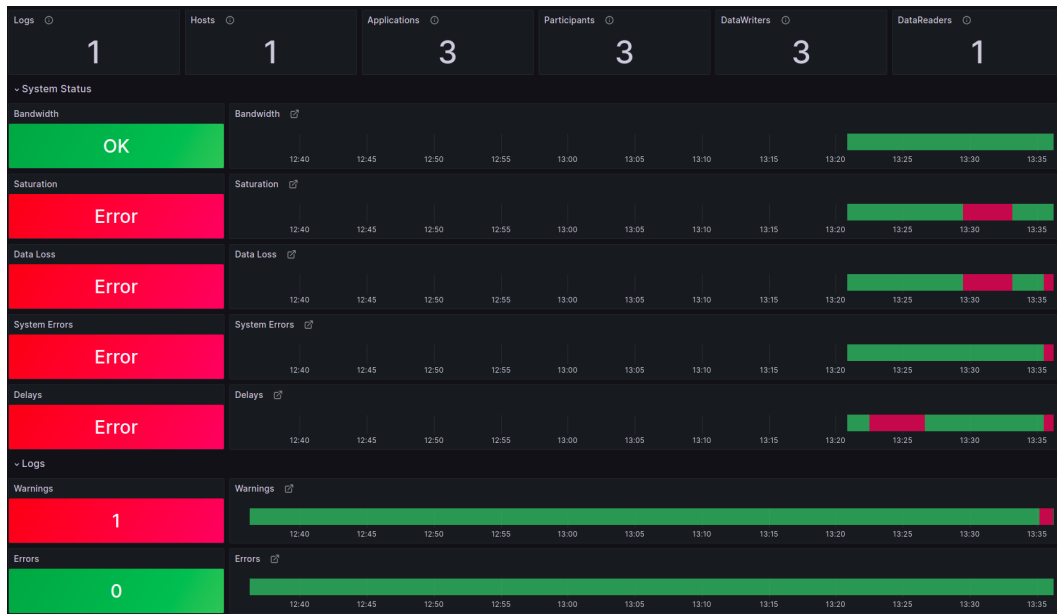
This final simulation demonstrates how to use logging information to troubleshoot problems. In this scenario, you'll create a clock synchronization issue in the first instance of Temperature_publisher. The clock will move forward in the future by a few seconds, causing the DataReader to drop some sensor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

To simulate this scenario, enter `clock_forward 0` in the first Temperature_publisher instance.

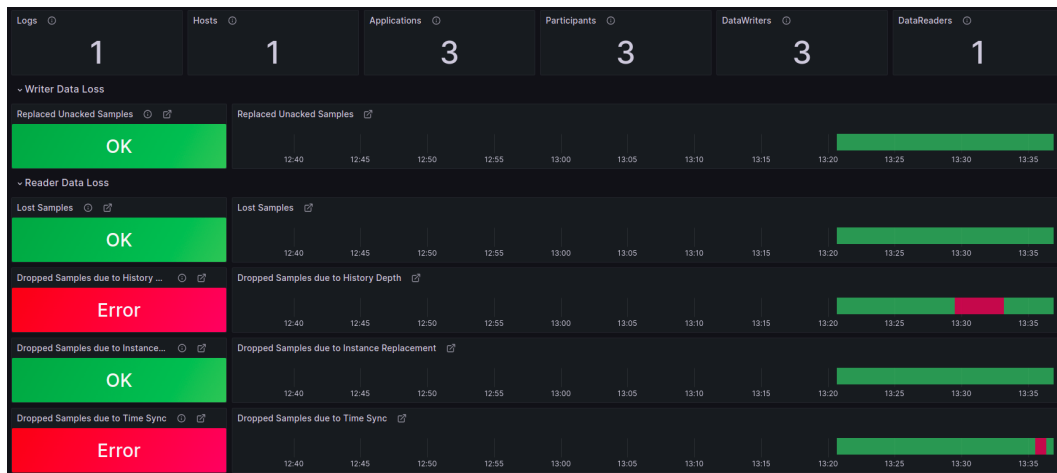
```
Command> clock_forward 0
```

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.

Dropped Samples due to Time Sync						
DataReader Name	Topic Name	Registered Type Name	Host Name	Domain Id	DDS GUID	Status
SensorSubscriber/Temperature...	Temperature	Temperature	sugarloaf	57	01013131.4856CAC6.BD3F33F...	ERROR

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?

Links

/SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader

Participant Name	Host Name	Process Id	Domain Id	Platform	Product Version
SensorSubscriber/Temperature DomainParticipant	sugarloaf	887615	57	x64Linux4gcc7.3.0	7.2.0.0
DataReader Name	Topic Name	Registered Type Name	DDS GUID		
SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader	Temperature	Temperature	01013131.4856CAC6.BD3F33F5.80000007		
Log Errors	Log Warnings				
0	1				

Participant Name SensorSubscriber/Temperature DomainParticipant		Host Name sugarloaf	Process Id 887615	Domain Id 57	Platform x64Linux4gcc7.3.0	Product Version 7.2.0.0
DataReader Name SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader			Topic Name Temperature	Registered Type Name Temperature		DDS GUID 01013131.4856CAC6.BD3F33F5.80000007
Applications 3	Participants 3		DataWriters 3		DataReaders 1	
Connex Logs						
Time	Facility	Log Level	Message			
2023-07-21 17:35:11.471460	MIDDLEWARE	WARNING	[01013131.4856CAC6.BD3F33F5.80000007][Entry=DR,MessageKind=DATA]RECEIVE FROM 0101BD17.3B55DC3B.7AE004C2.80000002] PRESCstReaderCollator_isNewer...			

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).

Participant Name SensorSubscriber/Temperature DomainParticipant		Host Name sugarloaf	Process Id 887615	Domain Id 57	Platform x64Linux4gcc7.3.0	Product Version 7.2.0.0
DataReader Name SensorSubscriber/Temperature DomainParticipant/TemperatureDataReader			Topic Name Temperature		Registered Type Name Temperature	DDS GUID 01013131.4856CAC6.BD3F33F5.80000007
Applications 3	Participants 3		DataWriters 3		DataReaders 1	
Connext Logs						
Time	Facility	Log Level	Message			
2023-07-21 17:35:11.471460	MIDDLEWARE	WARNING	[01013131.4856CAC6.BD3F33F5.80000007][Entity=DR,MessageKind=DATA][RECEIVE FROM 0101BD173B55DC3B7AE004C2.80000002] PRESCstReaderCollator_isNewer...			

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

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.

6.3.6 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*.

DataWriter Name ▾	Topic Name ▾	Registered Type Name ▾	Host Name ▾	Domain Id ▾	DDS GUID ▾
SensorPublisher_2/Temperature Do...	Temperature	Temperature	sugarloaf	57	0101FF9B.23468324.ED76EEBA.800...
SensorPublisher_1/Temperature Do...	Temperature	Temperature	sugarloaf	57	0101BD17.3B55DC3B.7AE004C2.800...
SensorPublisher_1/Temperature Do...	Temperature	Temperature	sugarloaf	57	0101BD17.3B55DC3B.7AE004C2.800...

DataWriter Name ▾	Topic Name ▾	Registered Type Name ▾	Host Name ▾	Domain Id ▾	DDS GUID ▾
SensorPublisher_2/Temperature Do...	Temperature	Temperature	sugarloaf	57	0101FF9B.23468324.ED76EEBA.800...
SensorPublisher_1/Temperature Do...	Temperature	Temperature	sugarloaf	57	0101BD17.3B55DC3B.7AE004C2.800...
SensorPublisher_1/Temperature Do...	Temperature	Temperature	sugarloaf	57	0101BD17.3B55DC3B.7AE004C2.800...

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** indicator 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 Middleware Collection log verbosity to open the Log Control dashboard.
4. From the Log Control dashboard, select **DEBUG** from the **Log Collection Verbosity** drop down menu.

Note that the verbosity setting color changes to yellow, indicating a change has been made. Also, the Set Collection Verbosity button is blue and enabled.

Links

/SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0

Participant Name	Host Name	Process Id	Domain Id	Platform	Product Version
SensorPublisher_1/Temperature DomainParticipant	sugarloaf	887546	57	x64Linux4gcc7.3.0	7.2.0.0
DataWriter Name	Topic Name	Registered Type Name	DDS GUID		
SensorPublisher_1/Temperature DomainParticipant/Sensor with ID=0	Temperature	Temperature	0101BD17.3B55DC3B.7AE004C2.80000002		
Log Errors	Log Warnings				
0	1				

Logs 0	Hosts 0	Applications 3	Participants 3	DataWriters 3	DataReaders 1
--------	---------	----------------	----------------	---------------	---------------

Application Name	Host Name	Process Id	GUID
SensorPublisher_2	sugarloaf	887582	4AD5AFCE.FF6259D6.C8DB1684.D354BDD6
SensorSubscriber	sugarloaf	887615	58CB1035.7FBE4FE9.5A48D9F9.77012341
SensorPublisher_1	sugarloaf	887546	EE0C9862.014FC9FA.F76FF228.B1D196B9

Application Name	Host Name	Process Id	GUID
/applications/SensorPublisher_1	sugarloaf	887546	EE0C9862.014FC9FA.F76FF228.B1D196B9
Log Errors	Log Warnings		
0	0		
Log Verbosity Configuration (click on panel to configure log verbosity)			
Middleware Collection	Middleware Forwarding		
WARNING	WARNING		

Application Name	Host Name	Process Id	GUID
/applications/SensorPublisher_1	sugarloaf	887546	EE0C9862.014FC9FA.F76FF228.B1D196B9
Log Collection Verbosity	Log Forwarding Verbosity		
MIDDLEWARE	MIDDLEWARE		
WARNING	WARNING		
SILENT			
ERROR			
WARNING			
INFORMATIONAL			
DEBUG			

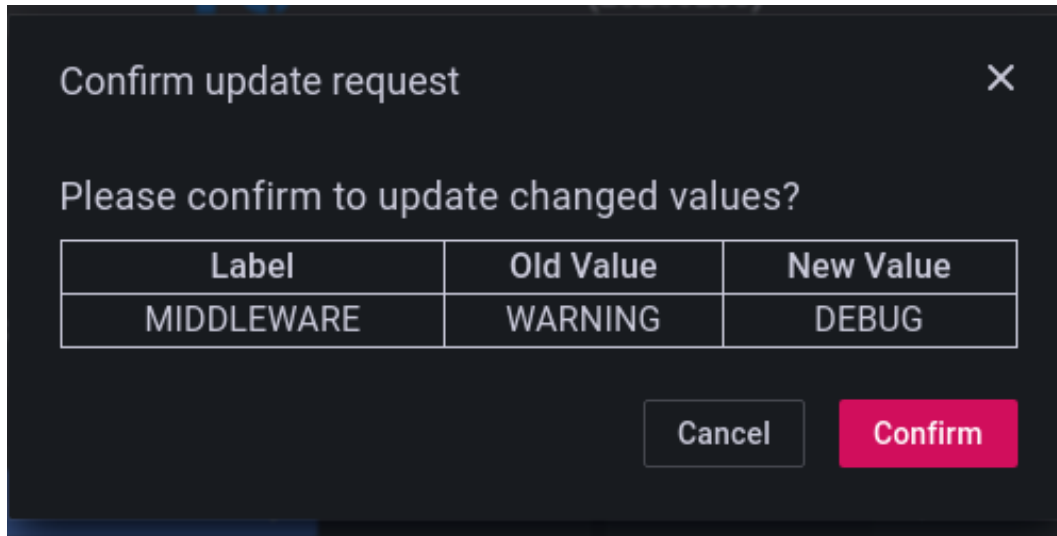
Log Collection Verbosity

MIDDLEWARE

DEBUG

Set Collection Verbosity

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



The application Collection verbosity is now DEBUG. If you examine the terminal window for SensorPublisher_1, you will see something similar to the following.

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

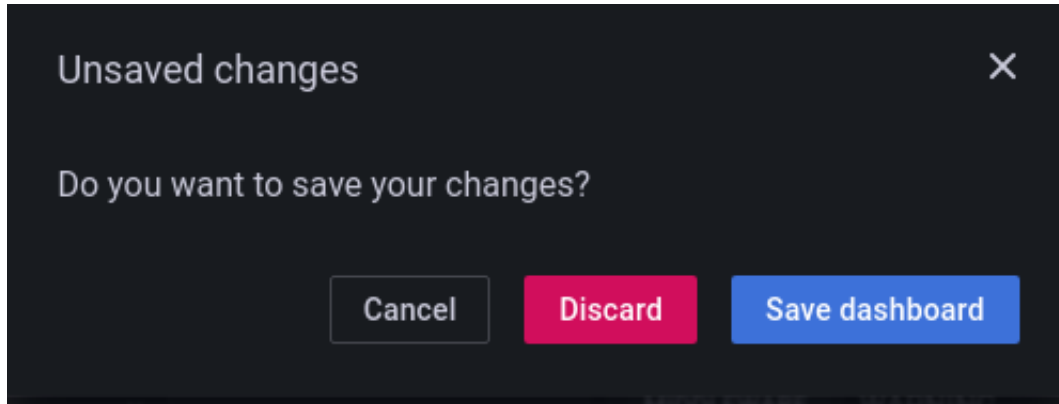
At this point, the `SensorPublisher_1` application is generating log messages at the `DEBUG` level as shown in the terminal window, but the log messages are not being forwarded to *Observability Collector Service* because the Forwarding Verbosity is still at `WARNING`.

To set the logging Forwarding verbosity to `DEBUG`, repeat steps 4 - 5 above using the Log Forwarding Verbosity pull down.

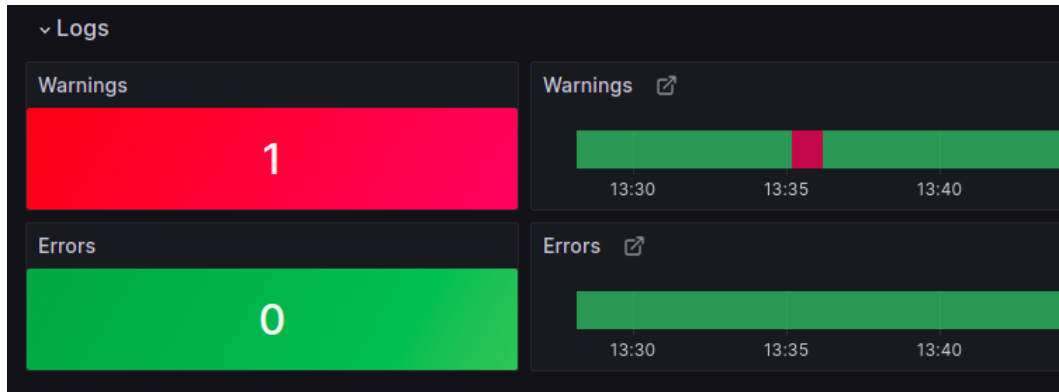
After setting both the Collection and Forwarding verbosity to `DEBUG`, you should see `DEBUG` messages for the `SensorPublisher_1` application in the Log Dashboard. To view the Log Dashboard, click the Home icon to go back to the Alert Home dashboard.



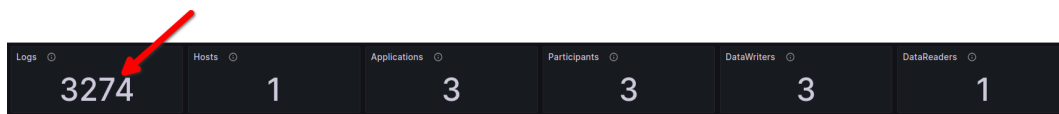
Note: If you are using the dashboards as an admin user, you will be prompted to save your changes. Select the **Discard** button; the changes to the dashboard do not need to be saved, since they are set in the application. This prompt does not appear when running with lesser permissions.



Now that both Collection and Forwarding verbosity are set to DEBUG, you can examine the DEBUG log messages. Note that there are no additional logs indicated in the Warnings and Errors Logs indicators.



From the Alert Home dashboard, select the **Logs** indicator to open the Log Dashboard.



Verify that DEBUG messages are shown in the Log Dashboard.

You can manipulate the Log Control settings to verify application and dashboard behavior as shown in Table 6.6.

Table 6.6: Collection and Forwarding Log Verbosity DEBUG Behavior

Collection Verbosity	Forwarding Verbosity	Application Log Output	DEBUG	Grafana Connex DE- BUG Logs
WARNING	WARNING	NO		NO
DEBUG	WARNING	YES		NO
WARNING	DEBUG	NO		NO
DEBUG	DEBUG	YES		YES

System Logs (contains all logs in the system)			
Applications	Participants	DataWriters	DataReaders
3	3	3	1
Connex Logs			
Time	Facility	Log Level	Message
2023-07-21 18:35:40.066198	MIDDLEWARE	DEBUG	MIGInterprete_parse:SECURE RTPS_PREFIX from 0XDFCD91E1,0X68682354
2023-07-21 18:35:40.066187	MIDDLEWARE	DEBUG	COMMENDActiveFacadeReceiver_loop:CoObsnt##00Rcv parsing message
2023-07-21 18:35:40.066178	MIDDLEWARE	DEBUG	RTINetioReceiver_receiveFastr:CoObsnt##00Rcv received 292 bytes
2023-07-21 18:35:40.066167	MIDDLEWARE	DEBUG	NDDS_Transport_UDP_receive_iEA:CoObsnt##00Rcv received 292 bytes from 0X100007F[54967
2023-07-21 18:35:40.066148	MIDDLEWARE	DEBUG	NDDS_Transport_UDP_receive_iEA:CoObsnt##00Rcv blocking on 0X1EE8
2023-07-21 18:35:40.066116	MIDDLEWARE	DEBUG	RTIEventActiveGeneratorThread_loop:Mo##Command firing events
2023-07-21 18:35:40.066085	MIDDLEWARE	DEBUG	COMMENDActiveFacadeReceiver_loop:CoObsnt##00Rcv returning message loan
2023-07-21 18:35:40.066054	MIDDLEWARE	DEBUG	accepted sn(000000000,00000011), dataRcvd.lead(0,11), nextRelSn(0,12), reservedCount(1)
2023-07-21 18:35:40.066046	MIDDLEWARE	DEBUG	RTIEventActiveGeneratorThread_loop:Mo##Command gathering events
2023-07-21 18:35:40.065694	MIDDLEWARE	DEBUG	[DFCD91E1.68682677.45DBCCD8.00020087(Entity=DR,MessageKind=DATA)]RECEIVE FROM DFCD91E1.68682354.A07665E8.00020082] COMMENDSReaderService_on...
2023-07-21 18:35:40.065656	MIDDLEWARE	DEBUG	MIGInterprete_parse:DATA from 0XDFCD91E1,0X68682354
2023-07-21 18:35:40.065644	MIDDLEWARE	DEBUG	MIGInterprete_parse:INFO_TS from 0XDFCD91E1,0X68682354
2023-07-21 18:35:40.065517	MIDDLEWARE	DEBUG	MIGInterprete_parse:SECURE RTPS_PREFIX from 0XDFCD91E1,0X68682354
2023-07-21 18:35:40.065501	MIDDLEWARE	DEBUG	COMMENDActiveFacadeReceiver_loop:CoObsnt##00Rcv parsing message
2023-07-21 18:35:40.065483	MIDDLEWARE	DEBUG	RTINetioReceiver_receiveFastr:CoObsnt##00Rcv received 292 bytes
2023-07-21 18:35:40.065404	MIDDLEWARE	DEBUG	NDDS_Transport_UDP_receive_iEA:CoObsnt##00Rcv received 292 bytes from 0X100007F[54967
2023-07-21 18:35:39.980350	MIDDLEWARE	DEBUG	RTIEventActiveGeneratorThread_loop:CoObsnt##00Rcv sleeping (00000000,27AE2542)
2023-07-21 18:35:39.980328	MIDDLEWARE	DEBUG	RTIEventActiveGeneratorThread_loop:CoObsnt##00Rcv rescheduling events
2023-07-21 18:35:39.980281	MIDDLEWARE	DEBUG	RTIEventActiveGeneratorThread_loop:CoObsnt##00Rcv firing events
2023-07-21 18:35:39.980208	MIDDLEWARE	DEBUG	RTIEventActiveGeneratorThread_loop:CoObsnt##00Rcv gathering events

6.3.7 Close the Applications

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

Chapter 7

Telemetry Data

7.1 Introduction

Connex Observability Framework enables you to instrument your *Connex* 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.

7.2 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)
- Participant
- 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. See *Metrics* for detailed information about the metrics available in this release.

Table 7.1 lists details of each available resource. The resource names follow REST best practices for naming.

Table 7.1: Observable Resource Names

Resource	Resource Name	Dashboard Resource Name	How to Configure
Application	/applications/ <AppName>	<AppName>	To set <AppName>, configure the participant_factory_qos.application_name QoS policy field for an application. For more information, see MONITORING QoSPolicy (DDS Extension)
Participant	/applications/ <AppName>/ domain_participants/ <ParticipantName>	<AppName>/ <ParticipantName>	To set <ParticipantName>, configure the participant_qos.participant_name QoS policy field for a Participant. 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 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 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 QoS policy field for a DataWriter. For more information, see ENTITY NAME QoSPolicy (DDS Extension)
DataReader	/applications/ <AppName>/ domain_participants/ <ParticipantName>/ data_readers/ <DataReaderName>	<AppName>/ <ParticipantName>/ <DataReaderName>	To set <DataReaderName>, configure the reader_qos.

7.2. Resources

The **Dashboard Resource Name** column describes how resource names appear in *RTI Connex 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.

7.3 Metrics

This section details the metrics you can collect from *Connex* entities. Each metric has a unique name and specifies a general feature of a measurable *Connex* resource. For example, a Datawriter is a measurable resource; the metric `dds_datawriter_protocol_sent_heartbeats_total` specifies the total number of heartbeats sent by a DataWriter.

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_datawriter_protocol_sent_heartbeats_total` corresponding to a DataWriter DW1 identified by a resource GUID GUID1.

Labels (or attributes) 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_datawriter_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 Connex 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 *Connex* system. For example, the `dds_participant_presence` indicates the presence of a Participant in a *Connex* system. There will be a time series for each Participant 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 Participant 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.

7.3.1 Application Metrics

The following tables describe the metrics and labels generated for *Connex* 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 7.2: Application Labels

Prometheus Label Name	Description
<code>guid</code>	Application resource GUID
<code>hostname</code>	Name of the host computer for the application
<code>process_id</code>	Process ID for the application
<code>name</code>	Fully qualified resource name (/applications/<AppName>)

Table 7.3: Application Metrics

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

7.3.2 Participant Metrics

The following tables describe the metrics and labels generated for *Connex* participants. Only the `dds_participant_presence` metric has all of the participant labels listed in the table below. All other participant metrics have the `guid` label only.

The Participant resource contains statistic variable metrics such as `dds_participant_udp4_usage_in_net_pkts_count`, `dds_participant_udp4_usage_in_net_pkts_mean`, `dds_participant_udp4_usage_in_net_pkts_min`, and `dds_participant_udp4_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_participant_udp4_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 *RTI 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_participant_udp4_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 *RTI 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_participant_udp4_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_participant_udp4_usage_in_net_pkts_mean` is calculated by averaging all individual mean metrics sent by *RTI Monitoring Library 2.0* to *Collector Service* over the last scraping period.

Table 7.4: Participant Labels

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

Table 7.5: Participant Metrics

Metric Name	Description	Type
<code>dds_participant_presence</code>	Indicates the presence of the participant and provides all label values for a participant instance	Gauge
<code>dds_participant_udp4_usage_in_net_pkts_count</code>	The UDPv4 transport in packets count over the last scraping period	Gauge
<code>dds_participant_udp4_usage_in_net_pkts_mean</code>	The UDPv4 transport in packets mean (packets/sec) over the last scraping period	Gauge
<code>dds_participant_udp4_usage_in_net_pkts_min</code>	The UDPv4 transport in packets min mean (packets/sec) over the last scraping period	Gauge
<code>dds_participant_udp4_usage_in_net_pkts_max</code>	The UDPv4 transport in packets max mean (packets/sec) over the last scraping period	Gauge

continues on next page

Table 7.5 – continued from previous page

Metric Name	Description	Type
dds_participant_udp4_usage_in_net_bytes_count	The UDPv4 transport in bytes count over the last scraping period	Gauge
dds_participant_udp4_usage_in_net_bytes_mean	The UDPv4 transport in bytes mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_in_net_bytes_min	The UDPv4 transport in bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_in_net_bytes_max	The UDPv4 transport in bytes max mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_pkts_count	The UDPv4 transport out packets count over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_pkts_mean	The UDPv4 transport out packets mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_pkts_min	The UDPv4 transport out packets min mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_pkts_max	The UDPv4 transport out packets max mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_bytes_count	The UDPv4 transport out bytes count over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_bytes_mean	The UDPv4 transport out bytes mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_bytes_min	The UDPv4 transport out bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp4_usage_out_net_bytes_max	The UDPv4 transport out bytes max mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_pkts_count	The UDPv6 transport in packets count over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_pkts_mean	The UDPv6 transport in packets mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_pkts_min	The UDPv6 transport in packets min mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_pkts_max	The UDPv6 transport in packets max mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_bytes_count	The UDPv6 transport in bytes count over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_bytes_mean	The UDPv6 transport in bytes mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_bytes_min	The UDPv6 transport in bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_in_net_bytes_max	The UDPv6 transport in bytes max mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_pkts_count	The UDPv6 transport out packets count over the last scraping period	Gauge

continues on next page

Table 7.5 – continued from previous page

Metric Name	Description	Type
dds_participant_udp6_usage_out_net_pkts_mean	The UDPv6 transport out packets mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_pkts_min	The UDPv6 transport out packets min mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_pkts_max	The UDPv6 transport out packets max mean (packets/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_bytes_count	The UDPv6 transport out bytes count over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_bytes_mean	The UDPv6 transport out bytes mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_bytes_min	The UDPv6 transport out bytes min mean (bytes/sec) over the last scraping period	Gauge
dds_participant_udp6_usage_out_net_bytes_max	The UDPv6 transport out bytes max mean (bytes/sec) over the last scraping period	Gauge

7.3.3 Topic Metrics

The following tables describe the metrics and labels generated for *Connex* 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 7.6: Topic Labels

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

Table 7.7: 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

7.3.4 DataWriter Metrics

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

Table 7.8: DataWriter Labels

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

Table 7.9: DataWriter Metrics

Metric Name	Description	Type
dds_datawriter_presence	Indicates the presence of the DataWriter and provides all label values for a DataWriter instance	Gauge
dds_datawriter_liveliness_lost_total	See total_count field in the LIVELINESS_LOST Status	Counter
dds_datawriter_deadline_missed_total	See total_count field in the OFFERED_DEADLINE_MISSED Status	Counter
dds_datawriter_incompatible_qos_total	See total_count field in the OFFERED_INCOMPATIBLE_QOS Status	Counter
dds_datawriter_reliable_cache_full_total	See full_reliable_writer_cache field in the RELIABLE_WRITER_CACHE_CHANGED Status	Counter
dds_datawriter_reliable_cache_high_watermark_total	See high_watermark_reliable_writer_cache field in the RELIABLE_WRITER_CACHE_CHANGED Status	Counter
dds_datawriter_reliable_cache_unack_samples	See unacknowledged_sample_count field in the RELIABLE_WRITER_CACHE_CHANGED Status	Gauge
dds_datawriter_reliable_cache_unack_samples_peak	See unacknowledged_sample_count_peak field in the RELIABLE_WRITER_CACHE_CHANGED Status	Gauge
dds_datawriter_reliable_cache_replaced_unack_samples_total	See replaced_unacknowledged_sample_count field in the RELIABLE_WRITER_CACHE_CHANGED Status	Counter
dds_datawriter_reliable_reader_activity_inactive_count	See inactive_count field in the RELIABLE_READER_ACTIVITY_CHANGED Status	Gauge
dds_datawriter_cache_samples_peak	See sample_count_peak field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_datawriter_cache_samples	See sample_count field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_datawriter_cache_alive_instances	See alive_instance_count field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_datawriter_cache_alive_instances_peak	See alive_instance_count_peak field in the DATA_WRITER_CACHE_STATUS	Gauge
dds_datawriter_protocol_pushed_samples_total	See pushed_sample_count field in the DATA_WRITER_PROTOCOL_STATUS	Counter
dds_datawriter_protocol_pushed_sample_bytes_total	See pushed_sample_bytes field in the DATA_WRITER_PROTOCOL_STATUS	Counter
dds_datawriter_protocol_sent_heartbeats_total	See sent_heartbeat_count field in the DATA_WRITER_PROTOCOL_STATUS	Counter
dds_datawriter_protocol_pulled_sample_count	See pulled_sample_count field in the DATA_WRITER_PROTOCOL_STATUS	Counter

7.3.5 DataReader Metrics

The following tables describe the metrics and labels generated for *Connex* 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 7.10: DataReader Labels

Prometheus Label Name	Description
<code>guid</code>	DataReader resource GUID
<code>owner_guid</code>	Resource GUID of the owner entity (subscriber)
<code>dds_guid</code>	DataReader DDS GUID
<code>hostname</code>	Name of the host computer for the participant this DataReader is registered with
<code>domain_id</code>	DDS domain ID for the participant this DataReader is registered with
<code>topic_name</code>	The topic name for this DataReader
<code>registered_type_name</code>	The registered type name for this DataReader
<code>name</code>	Fully qualified resource name (/applications/<AppName>/domain_participants/<ParticipantName> /publishers/<PublisherName>/data_readers/<DataReaderName>)
<code>participant_guid</code>	Resource GUID of the participant this DataReader is registered with

Table 7.11: DataReader Metrics

Metric Name	Description	Type
<code>ddsd_datareader_presence</code>	Indicates the presence of the DataReader and provides all label values for a DataReader instance	Gauge
<code>ddsd_datareader_sample_rejected_total</code>	See total_count field in the SAMPLE_REJECTED Status	Counter
<code>ddsd_datareader_liveliness_not_alive_count</code>	See not_alive_count field in the LIVELINESS_CHANGED Status	Gauge
<code>ddsd_datareader_deadline_missed_total</code>	See total_count field in the REQUESTED_DEADLINE_MISSED Status	Counter
<code>ddsd_datareader_incompatible_qos_total</code>	See total_count field in the REQUESTED_INCOMPATIBLE_QOS Status	Counter
<code>ddsd_datareader_sample_lost_total</code>	See total_count field in the SAMPLE_LOST Status	Counter
<code>ddsd_datareader_cache_samples_peak</code>	See sample_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
<code>ddsd_datareader_cache_samples</code>	See sample_count field in the DATA_READER_CACHE_STATUS	Gauge

continues on next page

Table 7.11 – continued from previous page

Metric Name	Description	Type
dds_datareader_cache_old_source_samples_total	See sets_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_datareader_cache_tolerance_source_ts_dropped_samples_total	See tolerance_source_timesamp_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_datareader_cache_content_filter_dropped_samples_total	See content_filter_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_datareader_cache_replaced_dropped_samples_total	See replaced_dropped_sample_count field in the DATA_READER_CACHE_STATUS	Counter
dds_datareader_cache_samples_dropped_by_instance_replaced_total	See total_samples_dropped_by_instance_replacement field in the DATA_READER_CACHE_STATUS	Counter
dds_datareader_cache_alive_instances	See alive_instance_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_datareader_cache_alive_instances_peak	See alive_instance_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
dds_datareader_cache_no_writers_instances	See no_writers_instance_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_datareader_cache_no_writers_instances_peak	See no_writers_instance_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
dds_datareader_cache_disposed_instances	See disposed_instance_count field in the DATA_READER_CACHE_STATUS	Gauge
dds_datareader_cache_disposed_instances_peak	See disposed_instance_count_peak field in the DATA_READER_CACHE_STATUS	Gauge
dds_datareader_cache_compressed_samples_total	See compressed_samples field in the DATA_READER_CACHE_STATUS	Counter
dds_datareader_protocol_received_samples_total	See received_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_received_sample_bytes_total	See received_sample_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_duplicate_samples_total	See duplicate_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_duplicate_sample_bytes_total	See duplicate_sample_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter

continues on next page

Table 7.11 – continued from previous page

Metric Name	Description	Type
dds_datareader_protocol_received_heartbeats_total	See received_heartbeat_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_sent_nacks_total	See sent_nack_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_sent_nack_bytes_total	See sent_nack_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_rejected_samples_total	See rejected_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_out_of_range_rejected_samples_total	See out_of_range_rejected_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_received_fragments_total	See received_fragment_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_dropped_fragments_total	See dropped_fragment_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_reassembled_samples_total	See reassembled_sample_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_sent_nack_fragments_total	See sent_nack_fragment_count field in the DATA_READER_PROTOCOL_STATUS	Counter
dds_datareader_protocol_sent_nack_fragment_bytes_total	See sent_nack_fragment_bytes field in the DATA_READER_PROTOCOL_STATUS	Counter

7.3.6 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 7.12: DDS Entity Proxy Metrics

Metric Name	Description
<code>dds_application_empty_metric</code>	A proxy for applications metrics that always provides a value of zero.
<code>dds_participant_empty_metric</code>	A proxy for applications metrics that always provides a value of zero.
<code>dds_topic_empty_metric</code>	A proxy for applications metrics that always provides a value of zero.
<code>dds_datawriter_empty_metric</code>	A proxy for applications metrics that always provides a value of zero.
<code>dds_datareader_empty_metric</code>	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 7.13 and Table 7.14.

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 7.13 lists derived Raw `error` metrics that are “enabled”.

Table 7.13: Raw Error Metrics (enabled)

Metric Name	Recording Rule
dds_datareader_cache_content_filter_dropped_samples_errors	rate(dds_datareader_cache_content_filter_dropped_samples_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datareader_cache_replaced_dropped_samples_errors	rate(dds_datareader_cache_replaced_dropped_samples_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datareader_cache_samples_dropped_by_instance_replaced_errors	rate(dds_datareader_cache_samples_dropped_by_instance_replaced_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_rejected_samples_errors	rate(dds_datareader_protocol_rejected_samples_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_out_of_range_rejected_samples_errors	rate(dds_datareader_protocol_out_of_range_rejected_samples_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_dropped_fragments_errors	rate(dds_datareader_protocol_dropped_fragments_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_topic_inconsistent_errors	rate(dds_topic_inconsistent_total[1m]) >bool 0 or dds_topic_empty_metric
dds_datawriter_incompatible_qos_errors	rate(dds_datawriter_incompatible_qos_total[1m]) >bool 0 or dds_datawriter_empty_metric
dds_datareader_incompatible_qos_errors	rate(dds_datareader_incompatible_qos_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datawriter_liveliness_lost_errors	rate(dds_datawriter_liveliness_lost_total[1m]) >bool 0 or dds_datawriter_empty_metric
dds_datawriter_reliable_reader_activity_inactive_count_errors	rate(dds_datawriter_reliable_reader_activity_inactive_count[1m]) >bool 0 or dds_datawriter_empty_metric
dds_datareader_liveliness_not_alive_count_errors	rate(dds_datareader_liveliness_not_alive_count[1m]) >bool 0 or dds_datareader_empty_metric
dds_datareader_cache_tolerance_source_ts_dropped_samples_errors	rate(dds_datareader_cache_tolerance_source_ts_dropped_samples_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datawriter_deadline_missed_errors	rate(dds_datawriter_deadline_missed_total[1m]) >bool 0 or dds_datawriter_empty_metric
dds_datareader_deadline_missed_errors	rate(dds_datareader_deadline_missed_total[1m]) >bool 0 or dds_datareader_empty_metric
dds_datawriter_reliable_cache_replaced_unack_samples_errors	rate(dds_datawriter_reliable_cache_replaced_unack_samples_total[1m]) >bool 0 or dds_datawriter_empty_metric
dds_datareader_sample_lost_errors	rate(dds_datareader_sample_lost_total[1m]) >bool 0 or dds_datareader_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_datawriter_protocol_pulled_samples_total[1m]) <bool 0 or dds_
↳datawriter_empty_metric
```

To

```
rate(dds_datawriter_protocol_pulled_samples_total[1m]) >bool 10 or dds_
↳datawriter_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 7.14 lists derived raw `error` metrics that are “disabled”.

Table 7.14: Raw Error Metrics (disabled)

Metric Name	Recording Rule
dds_datawriter_protocol_sent_heartbeats_errors	rate(dds_datawriter_protocol_sent_heartbeats_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_received_nacks_errors	rate(dds_datawriter_protocol_received_nacks_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_received_nack_bytes_errors	rate(dds_datawriter_protocol_received_nack_bytes_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_received_nack_fragments_errors	rate(dds_datawriter_protocol_received_nack_fragments_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_received_nack_fragment_bytes_errors	rate(dds_datawriter_protocol_received_nack_fragment_bytes_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datareader_protocol_received_heartbeats_errors	rate(dds_datareader_protocol_received_heartbeats_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_sent_nacks_errors	rate(dds_datareader_protocol_sent_nacks_total[1m]) <bool 0 or dds_datareader_empty_metric

continues on next page

Table 7.14 – continued from previous page

Metric Name	Recording Rule
dds_datareader_protocol_sent_nack_bytes_errors	rate(dds_datareader_protocol_sent_nack_bytes_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_sent_nack_fragments_errors	rate(dds_datareader_protocol_sent_nack_fragments_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_sent_nack_fragment_bytes_errors	rate(dds_datareader_protocol_sent_nack_fragment_bytes_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datawriter_protocol_pulled_samples_errors	rate(dds_datawriter_protocol_pulled_samples_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pulled_sample_bytes_errors	rate(dds_datawriter_protocol_pulled_sample_bytes_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pulled_fragments_errors	rate(dds_datawriter_protocol_pulled_fragments_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pulled_fragment_bytes_errors	rate(dds_datawriter_protocol_pulled_fragment_bytes_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pushed_samples_errors	rate(dds_datawriter_protocol_pushed_samples_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pushed_sample_bytes_errors	rate(dds_datawriter_protocol_pushed_sample_bytes_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pushed_fragments_errors	rate(dds_datawriter_protocol_pushed_fragments_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_pushed_fragment_bytes_errors	rate(dds_datawriter_protocol_pushed_fragment_bytes_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datareader_cache_compressed_samples_errors	rate(dds_datareader_cache_compressed_samples_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_duplicate_samples_errors	rate(dds_datareader_protocol_duplicate_samples_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_duplicate_sample_bytes_errors	rate(dds_datareader_protocol_duplicate_sample_bytes_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_received_samples_errors	rate(dds_datareader_protocol_received_samples_total[1m]) <bool 0 or dds_datareader_empty_metric

continues on next page

Table 7.14 – continued from previous page

Metric Name	Recording Rule
dds_datareader_protocol_received_sample_bytes_errors	rate(dds_datareader_protocol_received_sample_bytes_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_received_fragments_errors	rate(dds_datareader_protocol_received_fragments_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_protocol_reassembled_samples_errors	rate(dds_datareader_protocol_reassembled_samples_total[1m]) <bool 0 or dds_datareader_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_participant_udp4_usage_in_net_pkts_errors	rate(dds_participant_udp4_usage_in_net_pkts_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp4_usage_in_net_bytes_errors	rate(dds_participant_udp4_usage_in_net_bytes_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp4_usage_out_net_pkts_errors	rate(dds_participant_udp4_usage_out_net_pkts_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp4_usage_out_net_bytes_errors	rate(dds_participant_udp4_usage_out_net_bytes_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp6_usage_in_net_pkts_errors	rate(dds_participant_udp6_usage_in_net_pkts_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp6_usage_in_net_bytes_errors	rate(dds_participant_udp6_usage_in_net_bytes_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp6_usage_out_net_pkts_errors	rate(dds_participant_udp6_usage_out_net_pkts_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_participant_udp6_usage_out_net_bytes_errors	rate(dds_participant_udp6_usage_out_net_bytes_mean[1m]) <bool 0 or dds_participant_empty_metric
dds_datawriter_reliable_cache_full_errors	rate(dds_datawriter_reliable_cache_full_total[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_reliable_cache_high_watermark_errors	rate(dds_datawriter_reliable_cache_high_watermark_total[1m]) <bool 0 or dds_datawriter_empty_metric

continues on next page

Table 7.14 – continued from previous page

Metric Name	Recording Rule
dds_datawriter_reliable_cache_unack_samples_errors	rate(dds_datawriter_reliable_cache_unack_samples[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_reliable_cache_unack_samples_peak_errors	rate(dds_datawriter_reliable_cache_unack_samples_peak[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_protocol_send_window_size_errors	rate(dds_datawriter_protocol_send_window_size[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_cache_samples_errors	rate(dds_datawriter_cache_samples[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_cache_samples_peak_errors	rate(dds_datawriter_cache_samples_peak[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_cache_alive_instances_errors	rate(dds_datawriter_cache_alive_instances[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datawriter_cache_alive_instances_peak_errors	rate(dds_datawriter_cache_alive_instances_peak[1m]) <bool 0 or dds_datawriter_empty_metric
dds_datareader_sample_rejected_errors	rate(dds_datareader_sample_rejected_total[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_samples_errors	rate(dds_datareader_cache_samples[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_samples_peak_errors	rate(dds_datareader_cache_samples_peak[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_alive_instances_errors	rate(dds_datareader_cache_alive_instances[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_alive_instances_peak_errors	rate(dds_datareader_cache_alive_instances_peak[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_no_writers_instances_errors	rate(dds_datareader_cache_no_writers_instances[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_no_writers_instances_peak_errors	rate(dds_datareader_cache_no_writers_instances_peak[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_disposed_instances_errors	rate(dds_datareader_cache_disposed_instances[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_disposed_instances_peak_errors	rate(dds_datareader_cache_disposed_instances_peak[1m]) <bool 0 or dds_datareader_empty_metric
dds_datareader_cache_old_source_ts_samples_errors	rate(dds_datareader_cache_old_source_ts_dropped_samples_total[1m]) <bool 0 or dds_datareader_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 7.15 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 7.15: Aggregate Error Metrics

Metric Name	Recording Rule
dds_excessive_bandwidth_errors	$\text{clamp_max} \left(\left(\text{sum} \left(\text{dds_custom_excessive_bandwidth_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_sent_heartbeats_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_received_nacks_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_received_nack_bytes_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_received_nack_fragments_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_received_nack_fragment_bytes_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_received_heartbeats_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_sent_nacks_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_sent_nack_bytes_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_sent_nack_fragments_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_sent_nack_fragment_bytes_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pulled_samples_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pulled_sample_bytes_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pulled_fragments_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pulled_fragment_bytes_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pushed_samples_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pushed_sample_bytes_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pushed_fragments_errors} \right) + \text{sum} \left(\text{dds_datawriter_protocol_pushed_fragment_bytes_errors} \right) + \text{sum} \left(\text{dds_datareader_cache_content_filter_dropped_samples_errors} \right) + \text{sum} \left(\text{dds_datareader_cache_compressed_samples_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_duplicate_samples_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_duplicate_sample_bytes_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_received_samples_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_received_sample_bytes_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_received_fragments_errors} \right) + \text{sum} \left(\text{dds_datareader_protocol_reassembled_samples_errors} \right) \right), 1 \right)$
dds_saturation_errors	$\text{clamp_max} \left(\left(\text{sum} \left(\text{dds_custom_saturation_errors} \right) + \text{sum} \left(\text{dds_application_process_memory_usage_resident_memory_bytes_errors} \right) + \text{sum} \left(\text{dds_application_process_memory_usage_virtual_memory_bytes_errors} \right) + \text{sum} \left(\text{dds_participant_udp4_usage_in_net_pkts_errors} \right) + \text{sum} \left(\text{dds_participant_udp4_usage_in_net_bytes_errors} \right) + \text{sum} \left(\text{dds_participant_udp4_usage_out_net_pkts_errors} \right) + \text{sum} \left(\text{dds_participant_udp4_usage_out_net_bytes_errors} \right) \right), 100 \right)$
7.3. Metrics	

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_datareader_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_datareader_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_datareader_cache_alive_instances_errors
  expr: >
    rate(dds_datareader_cache_alive_instances[1m]) <bool 0 or dds_
↳datareader_empty_metric
```

The `dds_datareader_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_datareader_cache_alive_instances_errors
  expr: >
    dds_datareader_cache_alive_instances >bool 3 or dds_datareader_empty_
↳metric
```

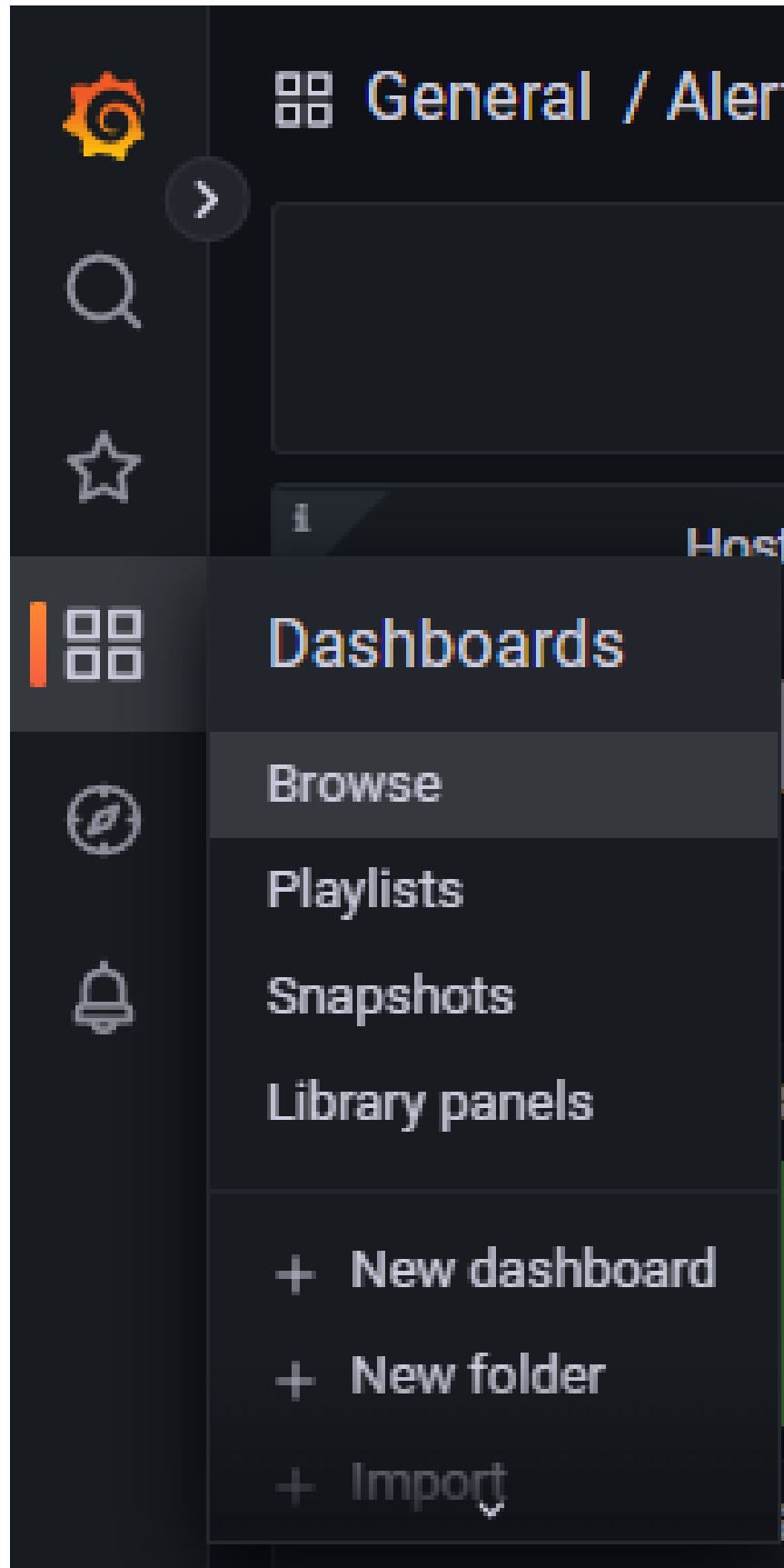
Important: After updating the `monitoring_recording_rules.yml` file, you must restart all Docker containers for *Observability Framework* by running `rtiobservability -t` followed by `rtiobservability -s`. The Prometheus server will read the updated file after restarting the containers.

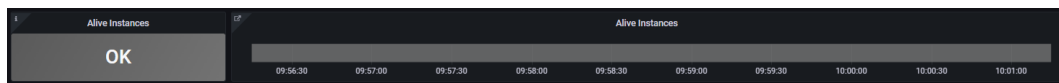
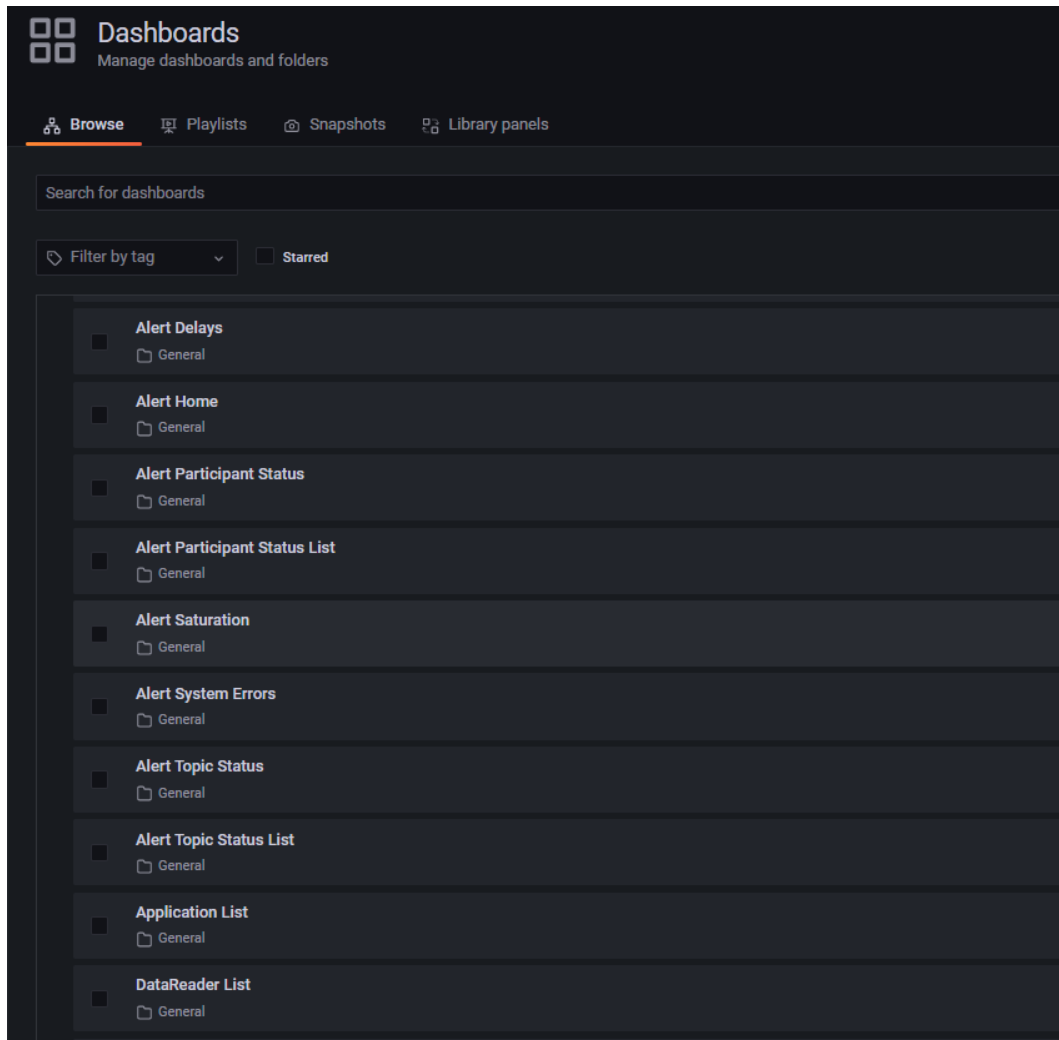
Update the Alert “Category” Dashboard

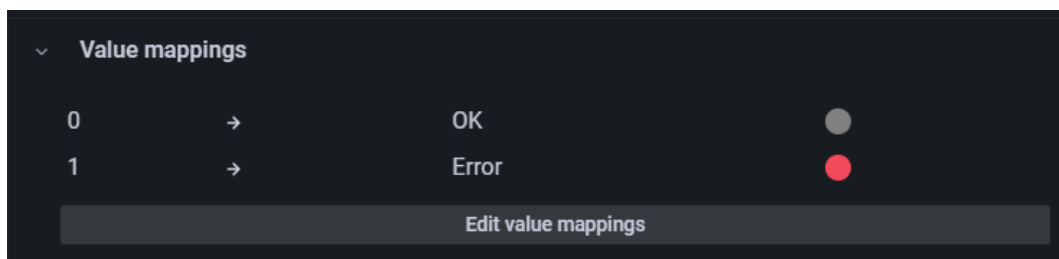
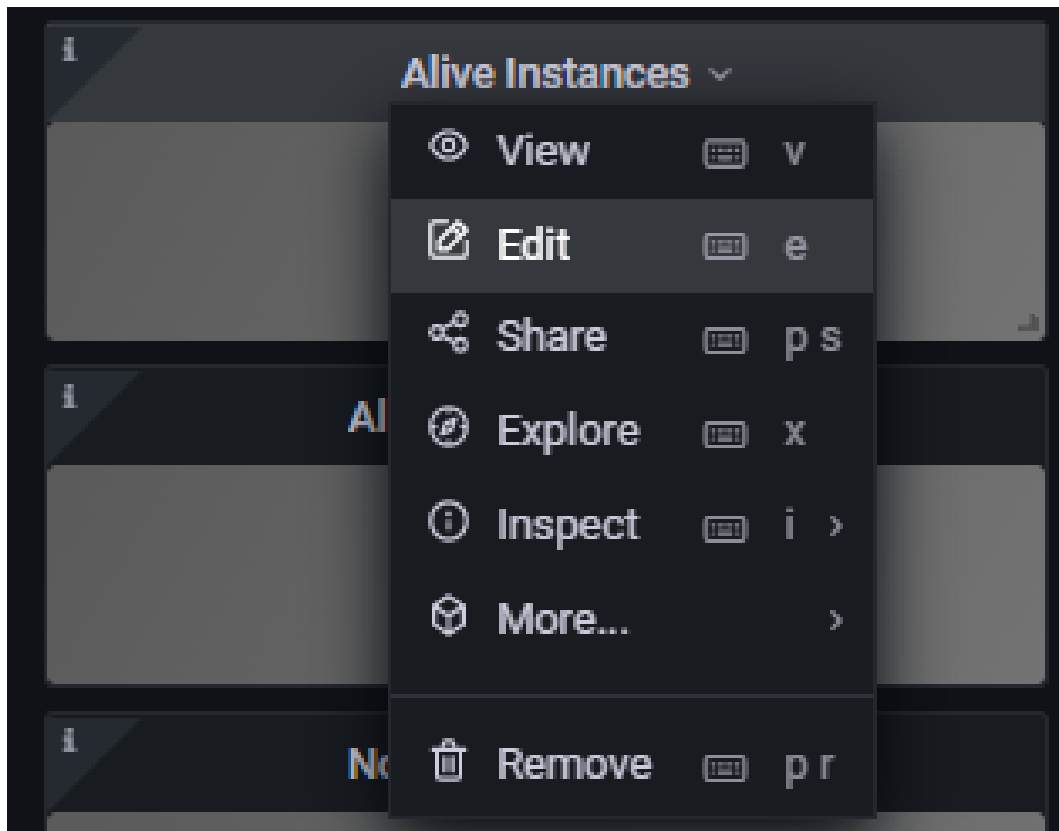
Note: The Grafana images in this section were generated with Grafana version 9.2.1. If you are using a different version of Grafana, the interface may be slightly different.

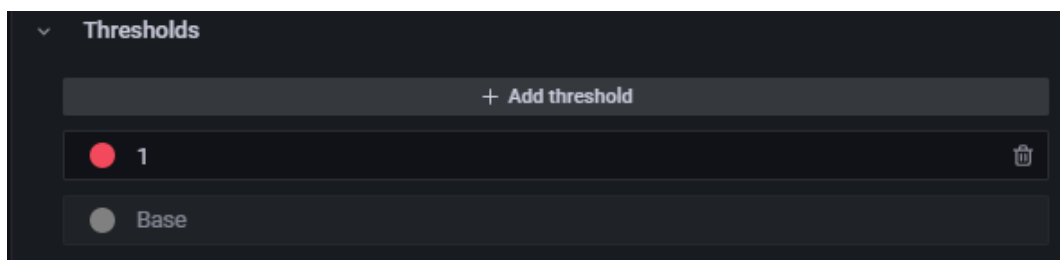
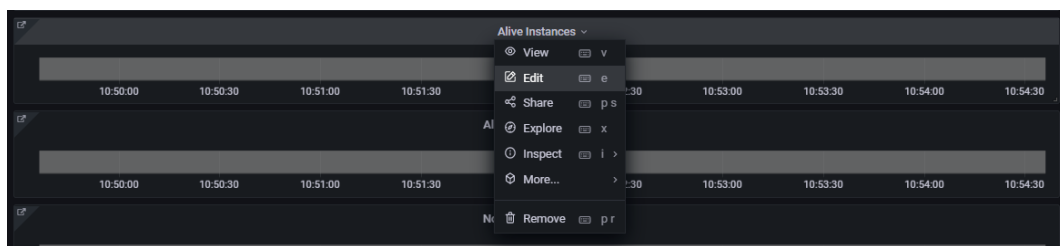
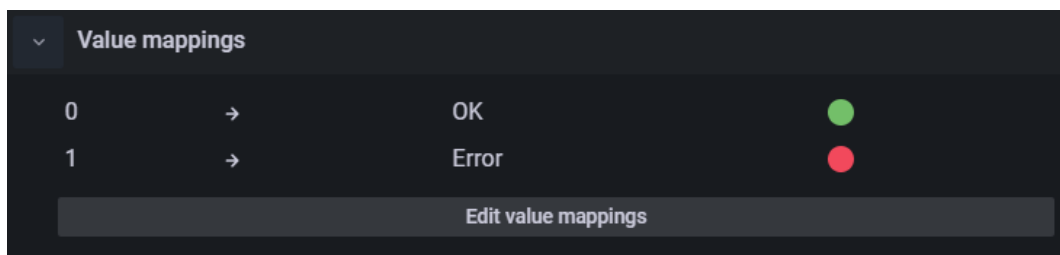
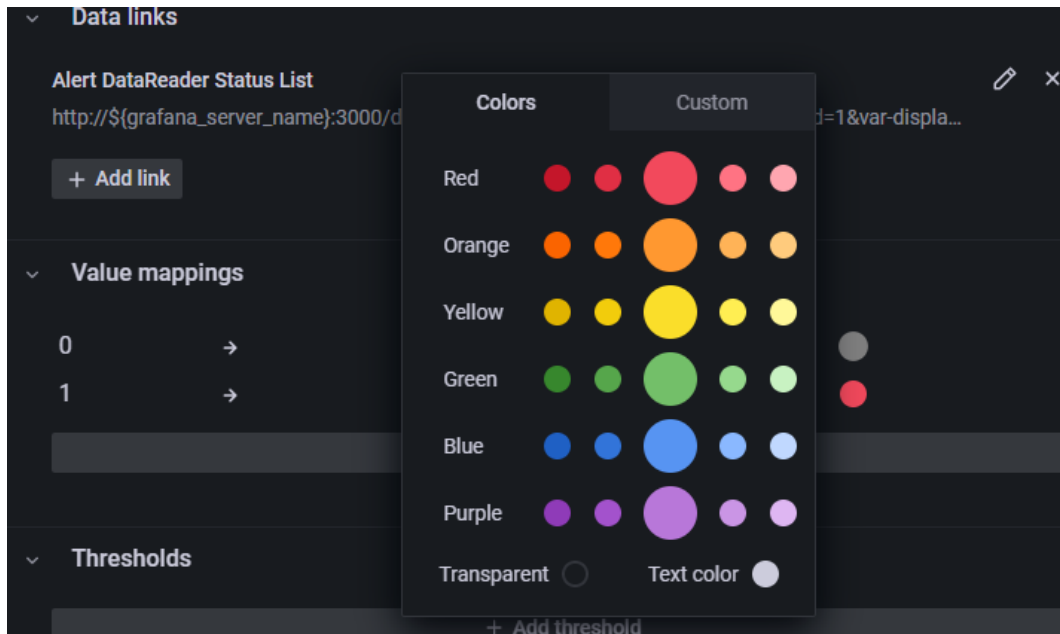
Locate the Alert “Category” dashboard for the metric rule you are enabling. The metric in our example, `dds_datareader_cache_alive_instances_errors`, is in the **Saturation** group (see Table 7.15), so the **Alert Saturation** dashboard is used in the following steps.

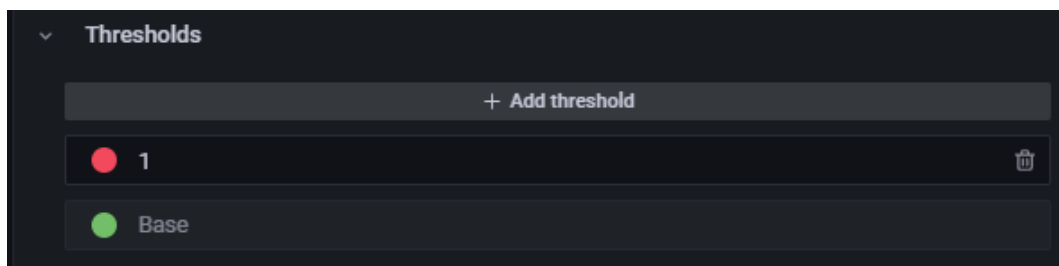
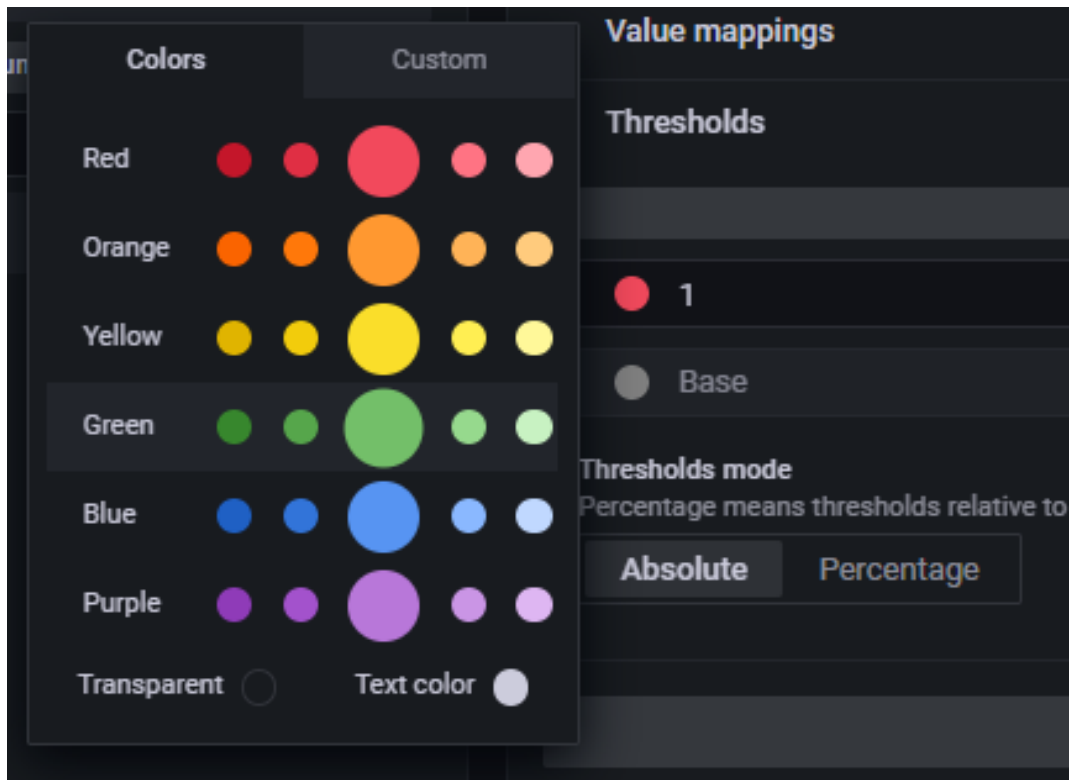
1. Go to **Dashboards > Browse** to open the list of dashboards.
2. Select the **Alert Saturation** dashboard from the list.
3. Once on the **Alert Saturation** dashboard, scroll down to the **Alive Instances** row under the **Reader Cache** section.
4. Select **Alive Instances > Edit** from the status 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.



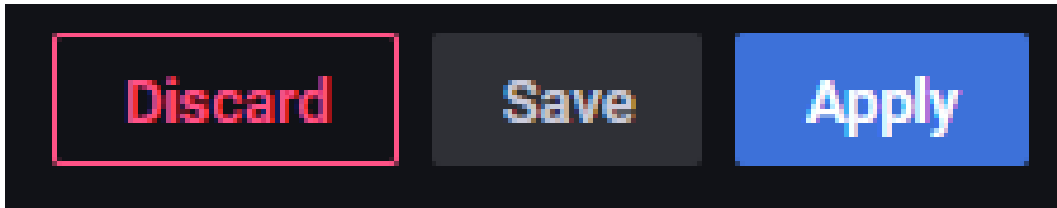




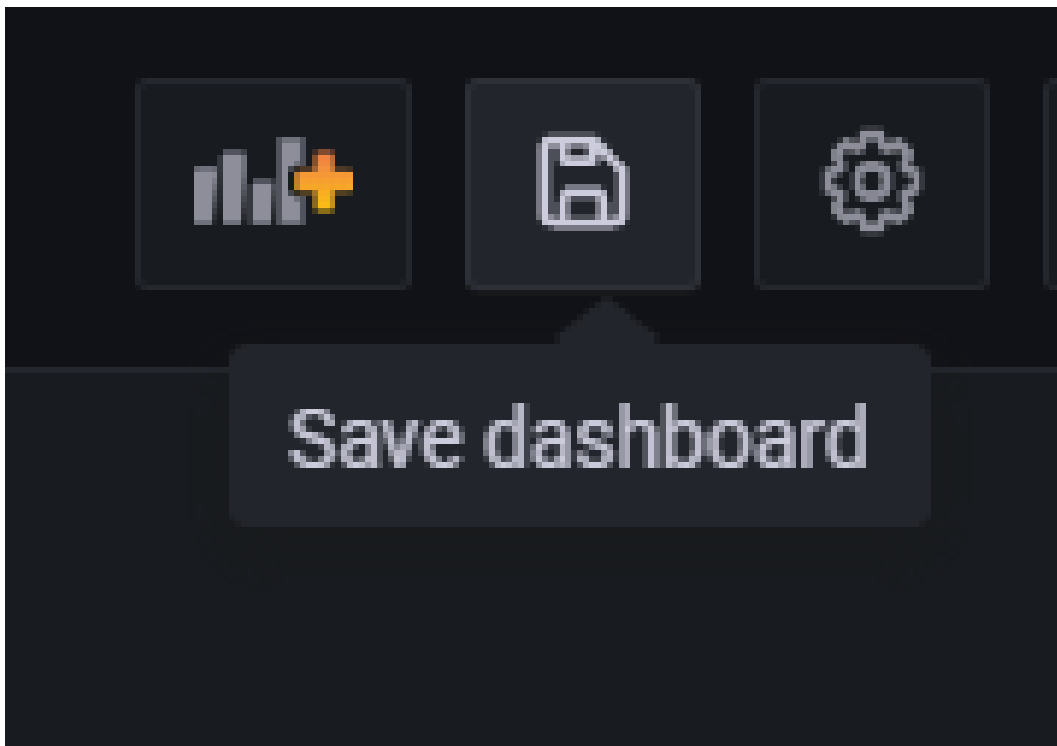




13. Select **Apply** at the top right to apply the changes and return to the **Alert Saturation** dashboard.

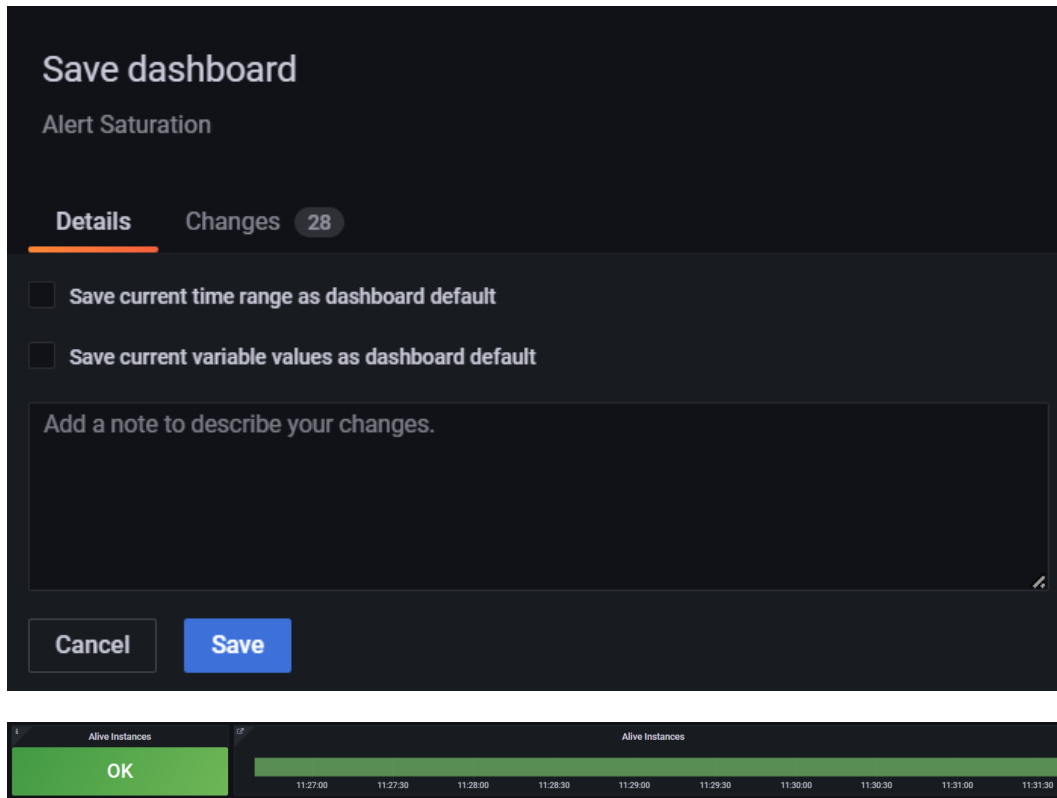


14. Select the **Save Dashboard** icon at the top right.



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

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



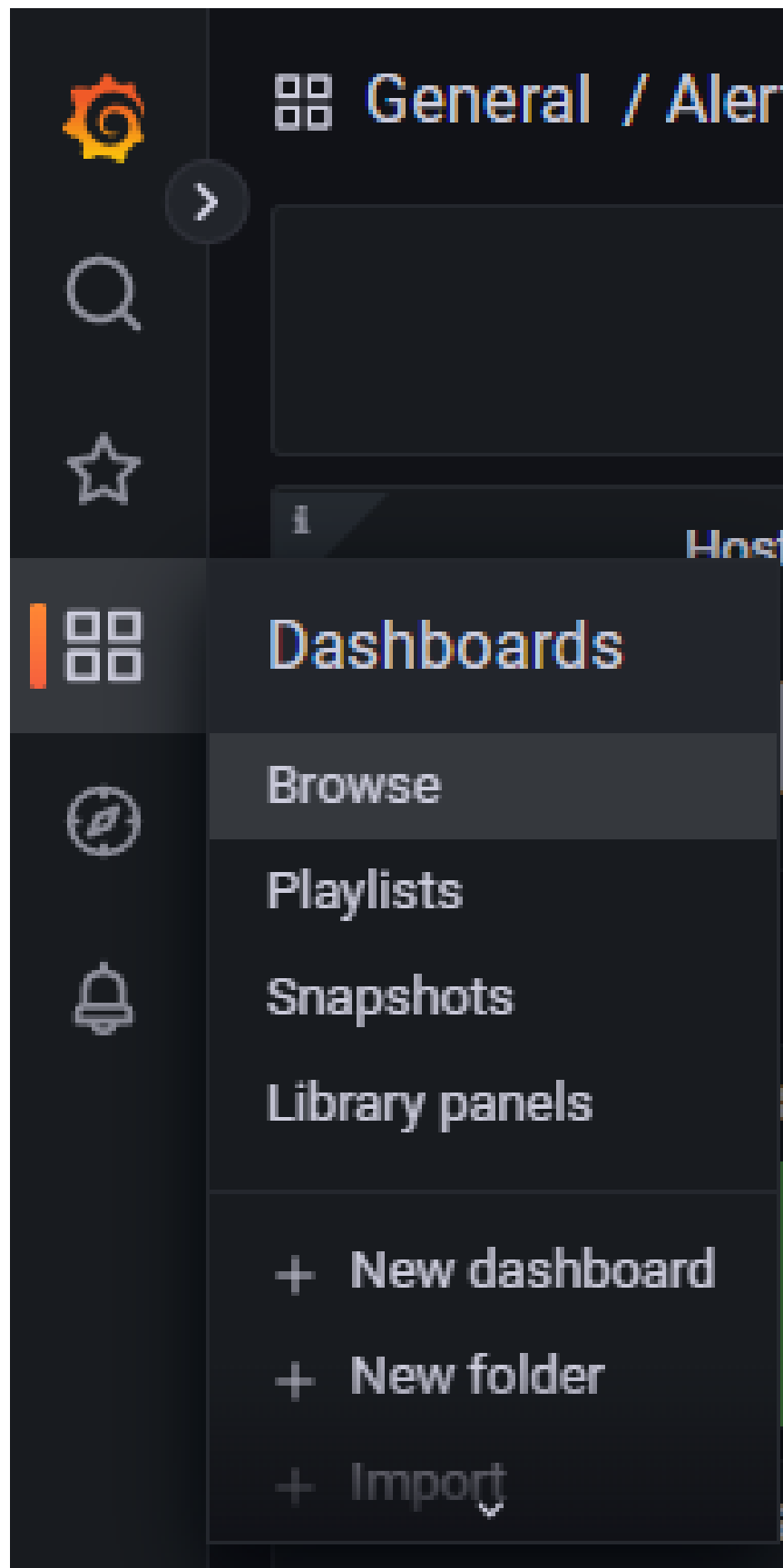
Update the “Entity” Status Dashboard

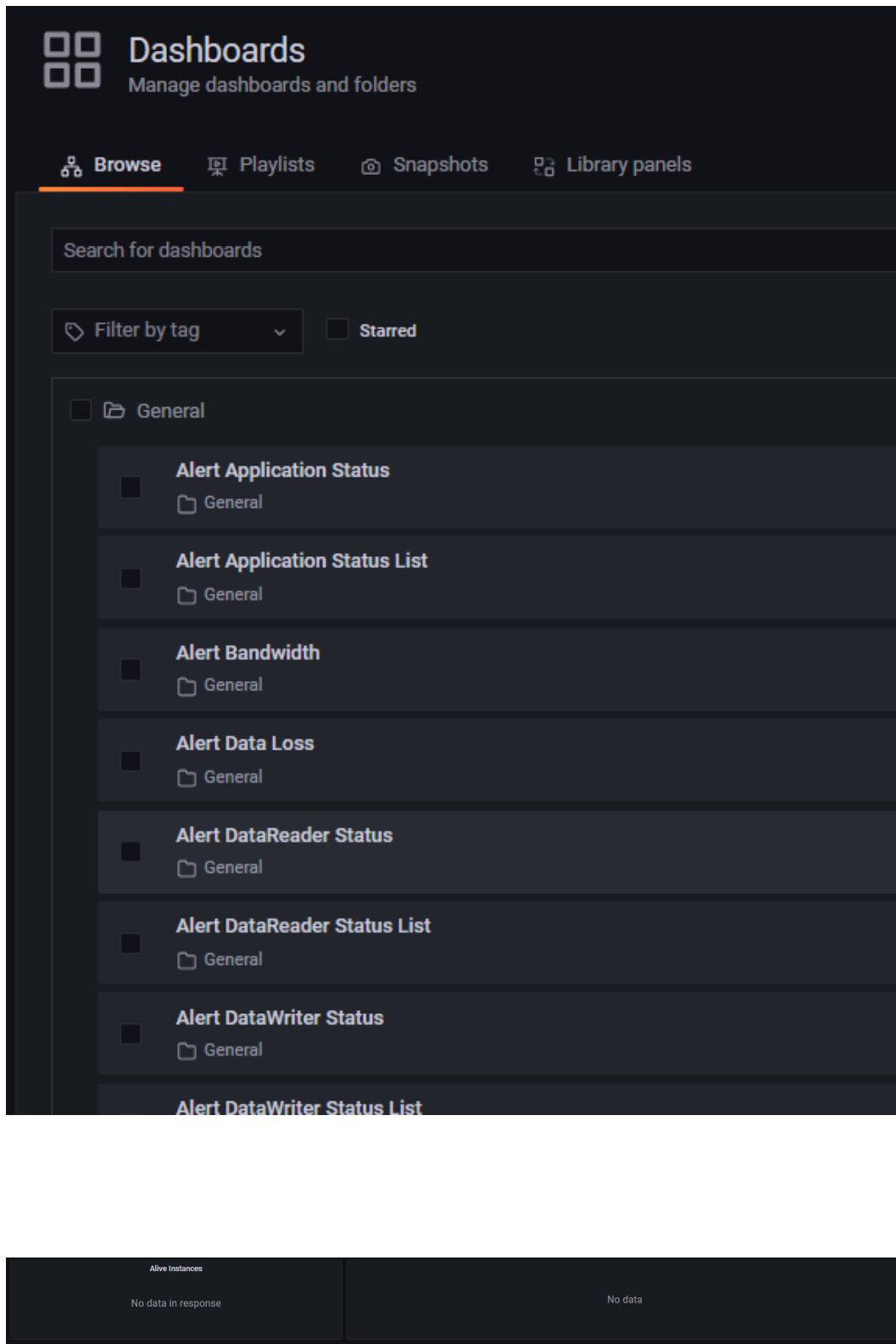
Locate the “Entity” status dashboard for the metric rule you are enabling. For the metric in our example, `dds_datareader_cache_alive_instances_errors`, we need to update the **Alert DataReader Status** dashboard.

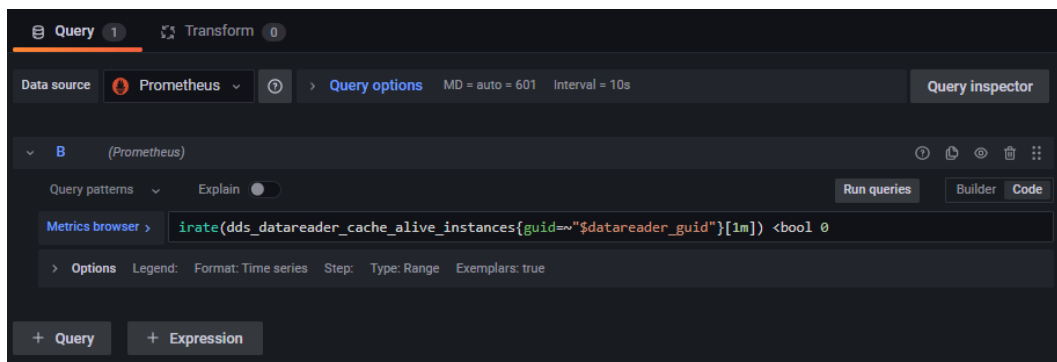
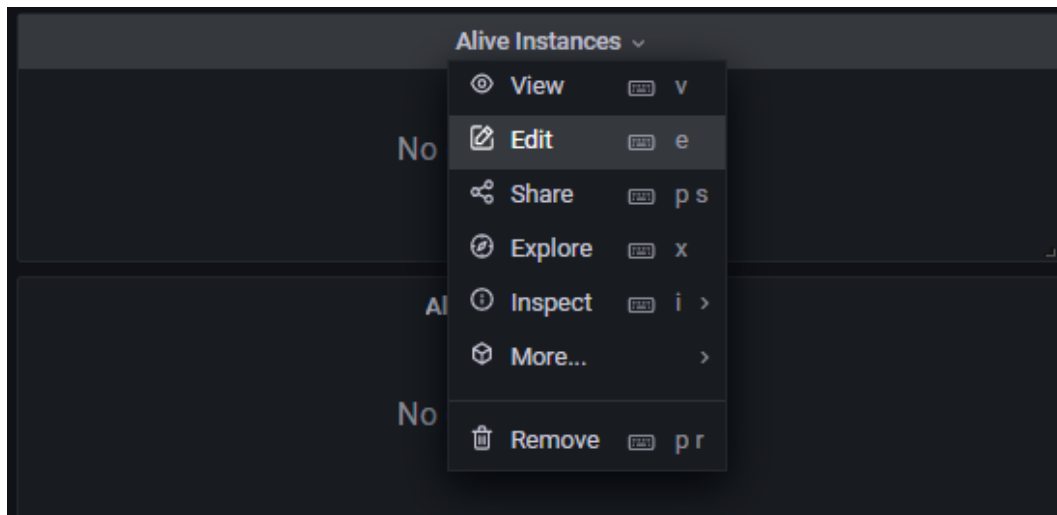
1. Go to **Dashboards > Browse** to open the list of dashboards.
2. Select the **Alert DataReader Status** dashboard from the list.
3. Once on the **Alert DataReader Status** dashboard, scroll down to the **Alive Instances** row under the **Saturation/Reader Cache** section.
4. Select **Alive Instances > Edit** from the status indicator panel menu.

The query for the panel is shown below.

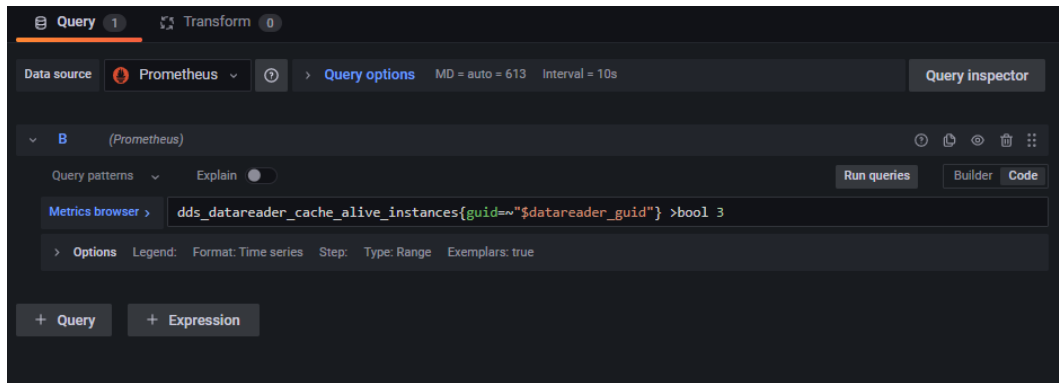
5. Edit the query to match the rule that was created for the `dds_datareader_cache_alive_instances_errors` metric. In the **Metrics browser** field, remove the `irate` calculation and set the limit



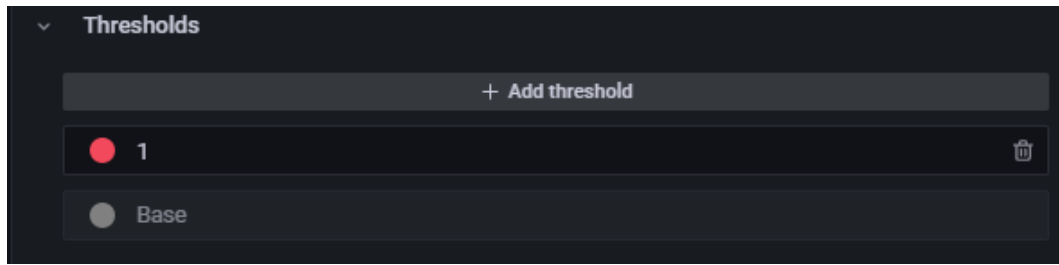




check to `>bool 3`, as shown below.



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.

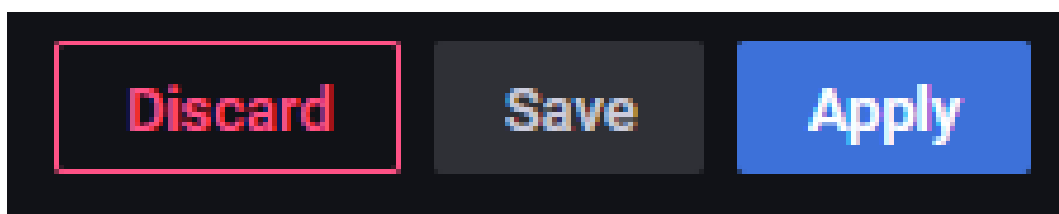
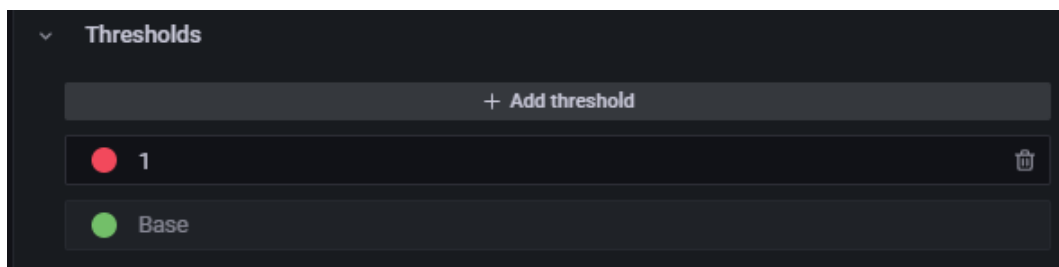
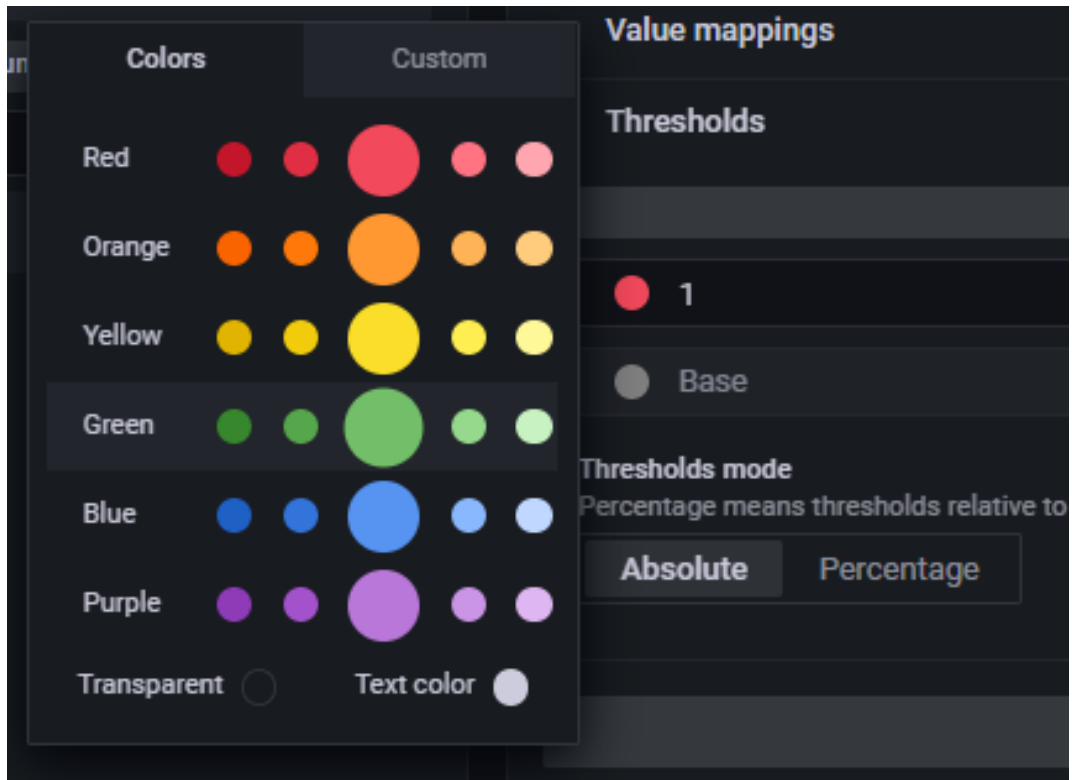
9. Select **Apply** at the top right to apply the change and return to the **Alert DataReader Status** dashboard.

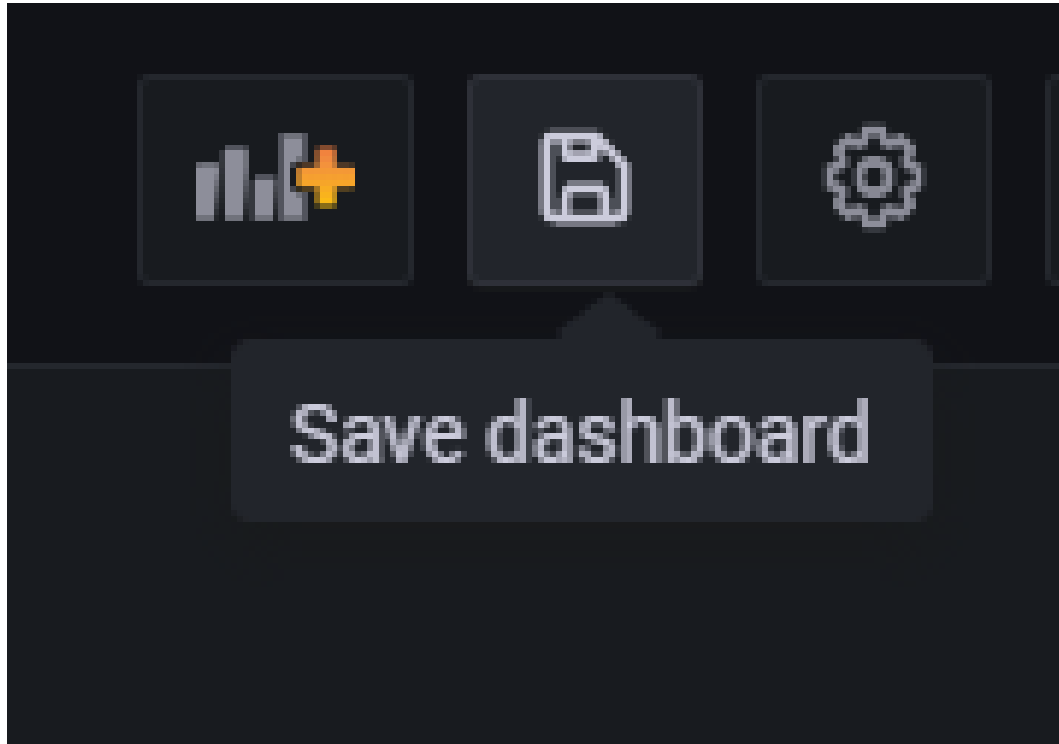
10. Select the **Save Dashboard** icon at the top right.

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

You have now enabled a rule for `dds_datareader_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





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.

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

Custom Error Metrics

Table 7.16 shows metrics that are not fully implemented.

Table 7.16: Custom Error Metrics

Metric Name	Description
dds_custom_excessive_bandwidth_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.

7.4 Logs

Observability Framework stores the log messages generated by *Connex* applications and stores them in a third-party backend for analysis.

Log messages generated before *Monitoring Library 2.0* is enabled are not stored.

Each log message is divided into three parts to facilitate analysis, as illustrated in Figure 7.1.

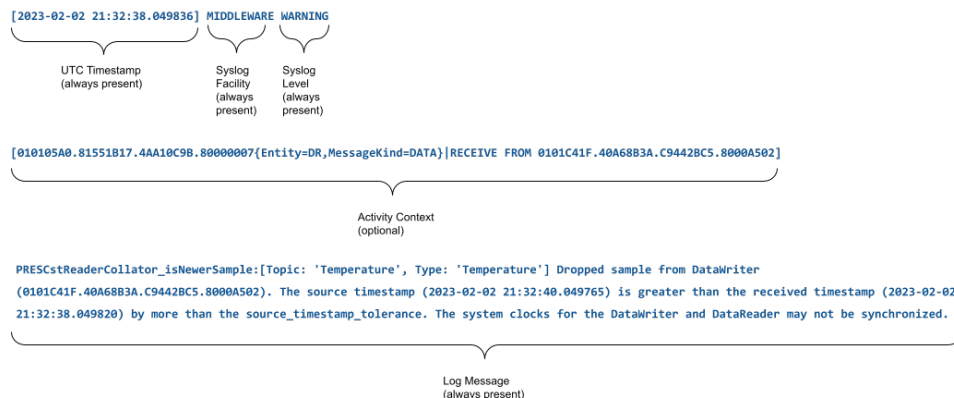


Figure 7.1: Log Message Format

This release only supports MIDDLEWARE as a Syslog Facility. Future releases will also provide support for SECURITY, SERVICE, and USER.

Syslog Level can have the following values: EMERGENCY, ERROR, WARNING, INFORMATIONAL, or DEBUG.

The mapping between *Connex* Log Levels and Syslog Levels is as follows:

Table 7.17: 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

Note: In the current release, it is not possible to filter only emergency (NDDS_CONFIG_LOG_LEVEL_FATAL_ERROR) log messages from *Observability Dashboards*.

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 Connex Core Libraries User's Manual*.

The Activity Context is available by default in all log messages generated by a *Connex* 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.

7.4.1 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 *Connex* application or remotely by sending a command.

In this release, the only way to send commands to change the logging verbosity is using *Observability Dashboards*. See *Change the Application Logging Verbosity* for further information.

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 QoS Policy \(DDS Extension\)](#). This QoS policy can be configured programmatically or via XML. The forwarding level can be changed only before the *Monitoring Library 2.0* is enabled.

Chapter 8

Monitoring Library 2.0

RTI Monitoring Library 2.0 is one component of the *Connex 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 (via dashboards) to change the set of generated and forwarded telemetry data at runtime.

The data forwarded by *Monitoring Library 2.0* is sent to a *Collector Service* instance. *Collector Service* forwards the data to other *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 (*rtmonitoring2*) and applications can use it in three different modes:

- **Dynamically loaded:** This is the default mode, which requires that the *rtmonitoring2* shared library is in the library search path.
- **Dynamic Linking:** The application is linked with the *rtmonitoring2* shared library.
- **Static Linking:** The application is linked with the *rtmonitoring2* static library.

The last two modes require calling the API `RTI_Monitoring_initialize` in your application before any other *Connex* APIs. This API is defined in the header file `ndds/monitoring/monitoring_monitoringClass.h`.

Dynamic and static linking are only supported in C and C++ applications.

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_datawriter_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_datawriter_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*.

8.1 Enabling Monitoring Library 2.0

To enable usage of *Monitoring Library 2.0* and to configure its behavior, you have to use the [MONITORING QoS Policy \(DDS Extension\)](#) 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">
    <participant_factory_qos>
      <monitoring>
        <enable>true</enable>
      </monitoring>
    </participant_factory_qos>
  </qos_profile>
</qos_library>
```

Alternatively, you can also use the snippet `BuiltinQosSnippetLib::Feature.Monitoring2.Enable` to enable the *Monitoring Library 2.0* in your XML configuration file:

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

In a typical application in addition to enabling the *Monitoring Library 2.0*, you will also want to configure the application name, the initial peer of the *Collector Service* to which the telemetry data will be forwarded, and the Observability Domain ID. The next XML configuration shows how to configure these parameters:

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

(continues on next page)

(continued from previous page)

```

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

```

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 \(DDS Extension\)](#).

8.2 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">
    <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 help identifying the resource that represents your *Connex* 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 RTI Observability Dashboards.

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

When `application_name` is not set, the RTI Observability Library will automatically assign a resource identifier with this format:

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

8.3 Changing the Default Observability Domain ID

To modify the domain for the *Monitoring Library 2.0 DomainParticipant* data, 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">
    <participant_factory_qos>

```

(continues on next page)

(continued from previous page)

```

    <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>

```

8.4 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` (located under `<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>
    <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
        -->
        <name>Monitoring Event DataWriter</name>
      </publication_name>
    </datawriter_qos>
  </qos_profile>
</qos_library>

```

(continues on next page)

(continued from previous page)

```

</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">
  <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>
          BuiltinQosLib:::Generic.Monitoring2
        </publisher_qos_profile_name>
        <event_settings>
          <!-- Change the configuration of the
               Observability Event DataWriter -->
          <datawriter_qos_profile_name>
            BuiltinQosLib:::Generic.Monitoring2
          </datawriter_qos_profile_name>
        </event_settings>
        <periodic_settings>
          <!-- Change the configuration of the
               Observability Periodic DataWriter -->
          <datawriter_qos_profile_name>
            BuiltinQosLib:::Generic.Monitoring2
          </datawriter_qos_profile_name>
        </periodic_settings>

```

(continues on next page)

(continued from previous page)

```

    <logging_settings>
      <!-- Change the configuration of the
           Observability Logging DataWriter -->
      <datawriter_qos_profile_name>
        BuiltinQosLib::Generic.Monitoring2
      </datawriter_qos_profile_name>
    </logging_settings>
  </distribution_settings>
</monitoring>
</participant_factory_qos>
</qos_profile>
</qos_library>

```

The `BuiltinQosLib::Generic.Monitoring2` profile disables the use of multicast discovery by setting the `<multicast_receive_addresses/>` element for the *Monitoring Library 2.0 DomainParticipant*. This setting prevents issues with multicast discovery in networks where multicast is not available. In addition, a single application should connect only to one *Collector Service* instance. Using multicast may lead to multiple *Collector Service* instances receiving the same data.

To enable multicast discovery, create a profile that overwrites the `BuiltinQosLib::Generic.Monitoring2` profile and set the `<multicast_receive_addresses>` element to the multicast address.

```

<qos_library name="MyQosLibrary">
  <qos_profile name="MyObservabilityProfile" base_name=
↪ "BuiltinQosLib::Generic.Monitoring2">
    <domain_participant_qos>
      <discovery>
        <multicast_receive_addresses>239.255.0.1</multicast_receive_
↪ addresses>
      </discovery>
    </domain_participant_qos>
  </qos_profile>
</qos_library>

```

8.5 Setting Collector Service Initial Peers

In most cases, the initial peer used to connect to *Collector Service* will be different from the initial peers used by the application. To change the initial peers for the *Monitoring Library 2.0 DomainParticipant*, use the `participant_factory_qos.monitoring.distribution_settings.dedicated_participant.collector_initial_peers` field:

```

<qos_library name="MyQosLibrary">
  <qos_profile name="MyApplicationProfile">
    <participant_factory_qos>
      <monitoring>
        <enable>true</enable>
        <distribution_settings>

```

(continues on next page)

(continued from previous page)

```
<dedicated_participant>
  <collector_initial_peers>
    <element>192.168.1.2</element>
  </collector_initial_peers>
</dedicated_participant>
</distribution_settings>
</monitoring>
</participant_factory_qos>
</qos_profile>
</qos_library>
```

If no `collector_initial_peers` are specified, or if it is explicitly set to an empty list, Monitoring Library 2.0 will use the `initial_peers` list of the QoS profile specified by `participant_factory_qos.monitoring.distribution_settings.dedicated_participant.participant_qos_profile_name` as the initial peers of the *Monitoring Library 2.0 DomainParticipant*.

If both are present, the value in `collector_initial_peers` takes precedence over `initial_peers`.

Chapter 9

Troubleshooting Observability Framework

This section provides solutions for issues you may run into while evaluating *Observability Framework*.

9.1 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_connexdds-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	
08f75e0fad2	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_connexdds-7..."	→
→ 5 minutes ago	Up 5 minutes	collector_service_observability	

(continues on next page)

(continued from previous page)

```
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.

9.1.1 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 Observability Workspace with the new port configuration.
4. Run `<installdir>/bin/rtiobservability -i` to create and run the Docker containers with the new port configuration.

9.1.2 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

9.2 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*).

9.2.1 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_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_participant_presence gauge
dds_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_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_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_participant_presence gauge
# TYPE dds_participant_udpv4_usage_in_net_pkts_period_ms gauge
# TYPE dds_participant_udpv4_usage_in_net_pkts_count gauge
# TYPE dds_participant_udpv4_usage_in_net_pkts_mean gauge
# TYPE dds_participant_udpv4_usage_in_net_pkts_min gauge
# TYPE dds_participant_udpv4_usage_in_net_pkts_max gauge
# TYPE dds_participant_udpv4_usage_in_net_bytes_period_ms gauge
# TYPE dds_participant_udpv4_usage_in_net_bytes_count gauge
# TYPE dds_participant_udpv4_usage_in_net_bytes_mean gauge
# TYPE dds_participant_udpv4_usage_in_net_bytes_min gauge
# TYPE dds_participant_udpv4_usage_in_net_bytes_max gauge
# TYPE dds_participant_udpv4_usage_out_net_pkts_period_ms gauge
# TYPE dds_participant_udpv4_usage_out_net_pkts_count gauge
# TYPE dds_participant_udpv4_usage_out_net_pkts_mean gauge
# TYPE dds_participant_udpv4_usage_out_net_pkts_min gauge
# TYPE dds_participant_udpv4_usage_out_net_pkts_max gauge
# TYPE dds_participant_udpv4_usage_out_net_bytes_period_ms gauge
# TYPE dds_participant_udpv4_usage_out_net_bytes_count gauge
# TYPE dds_participant_udpv4_usage_out_net_bytes_mean gauge
# TYPE dds_participant_udpv4_usage_out_net_bytes_min gauge
# TYPE dds_participant_udpv4_usage_out_net_bytes_max gauge
# TYPE dds_participant_udpv6_usage_in_net_pkts_period_ms gauge
# TYPE dds_participant_udpv6_usage_in_net_pkts_count gauge
# TYPE dds_participant_udpv6_usage_in_net_pkts_mean gauge
# TYPE dds_participant_udpv6_usage_in_net_pkts_min gauge
# TYPE dds_participant_udpv6_usage_in_net_pkts_max gauge
# TYPE dds_participant_udpv6_usage_in_net_bytes_period_ms gauge
# TYPE dds_participant_udpv6_usage_in_net_bytes_count gauge
# TYPE dds_participant_udpv6_usage_in_net_bytes_mean gauge
# TYPE dds_participant_udpv6_usage_in_net_bytes_min gauge
# TYPE dds_participant_udpv6_usage_in_net_bytes_max gauge
# TYPE dds_participant_udpv6_usage_out_net_pkts_period_ms gauge
# TYPE dds_participant_udpv6_usage_out_net_pkts_count gauge
# TYPE dds_participant_udpv6_usage_out_net_pkts_mean gauge
# TYPE dds_participant_udpv6_usage_out_net_pkts_min gauge
# TYPE dds_participant_udpv6_usage_out_net_pkts_max gauge
```

(continues on next page)

(continued from previous page)

```
# TYPE dds_participant_udp6_usage_out_net_bytes_period_ms gauge
# TYPE dds_participant_udp6_usage_out_net_bytes_count gauge
# TYPE dds_participant_udp6_usage_out_net_bytes_mean gauge
# TYPE dds_participant_udp6_usage_out_net_bytes_min gauge
# TYPE dds_participant_udp6_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).

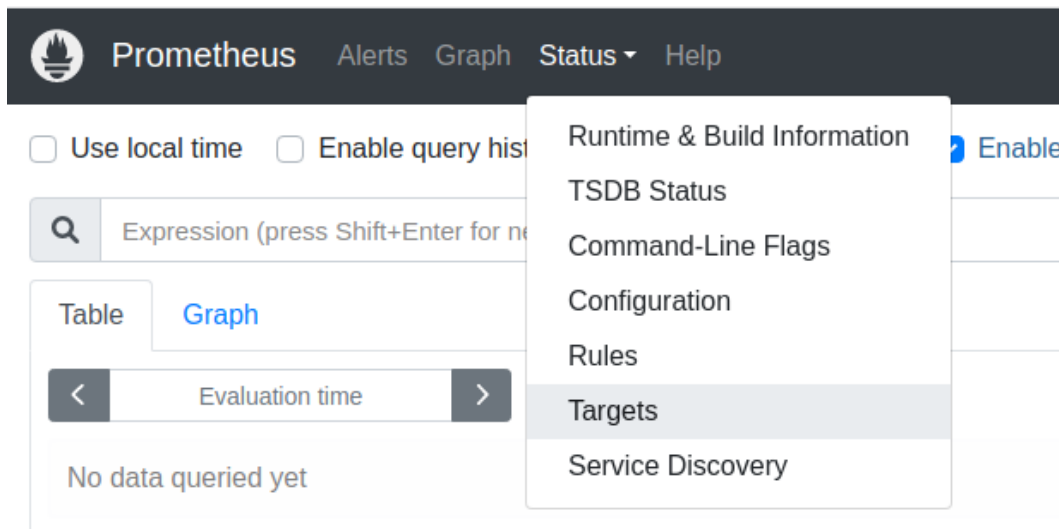
If your applications are run on a machine other than the one hosting *Observability Collector Service*, ensure that the initial peers list is configured with the IP address where *Observability Collector Service* is running.

For more information on configuring *Monitoring Library 2.0* for your application, see *Monitoring Library 2.0*.

9.2.2 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).
- Examine the Error to see if there is another cause being reported.

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	

9.2.3 Check that Grafana can Access Prometheus

Note: These steps can only be performed as a Grafana Admin user.

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 details might be slightly different.

Open the Grafana Data Sources Configuration page.

Select the “Prometheus” data source.

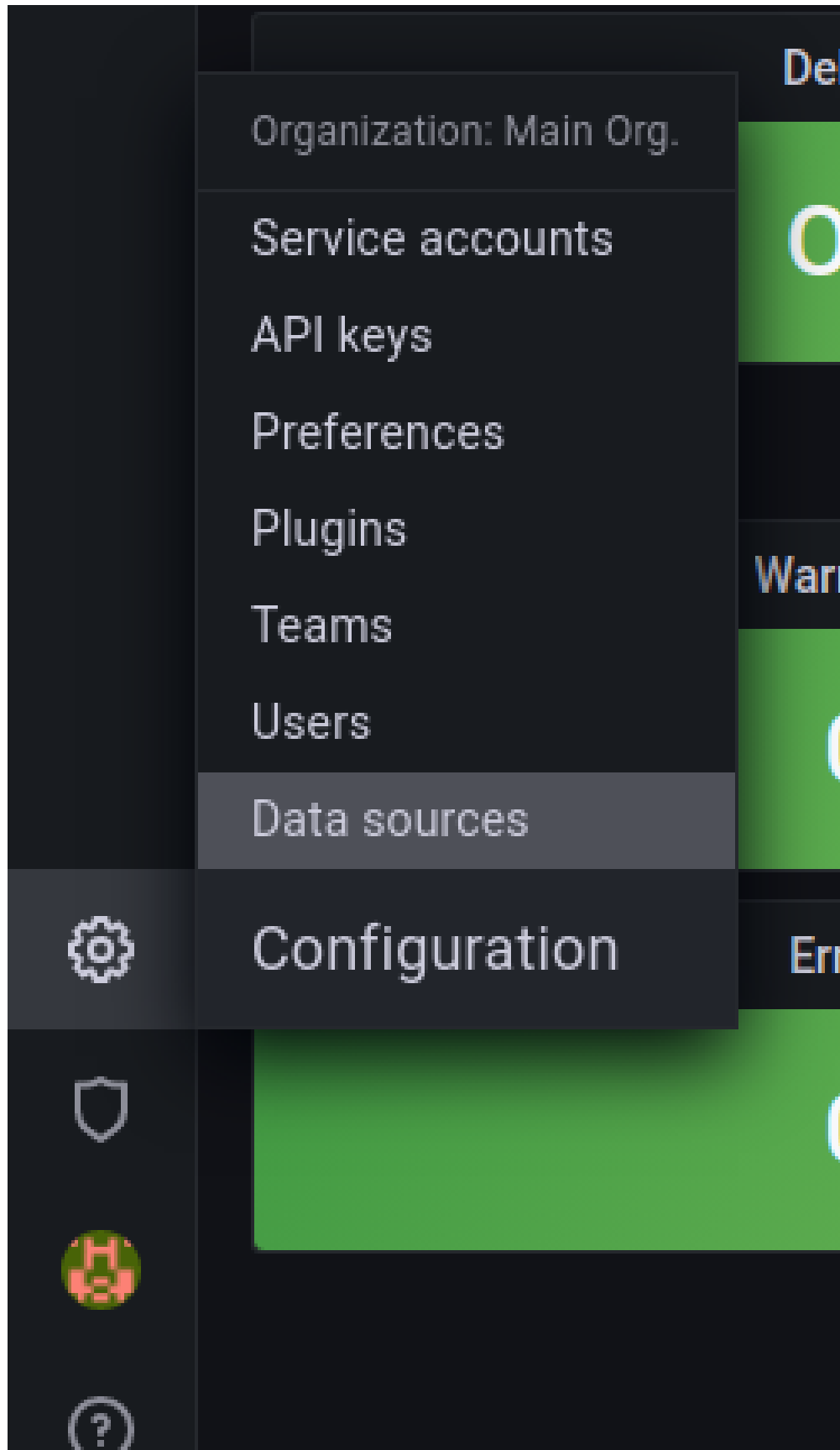
Scroll down and click **Test** to ensure that Grafana has connectivity with the Prometheus server.

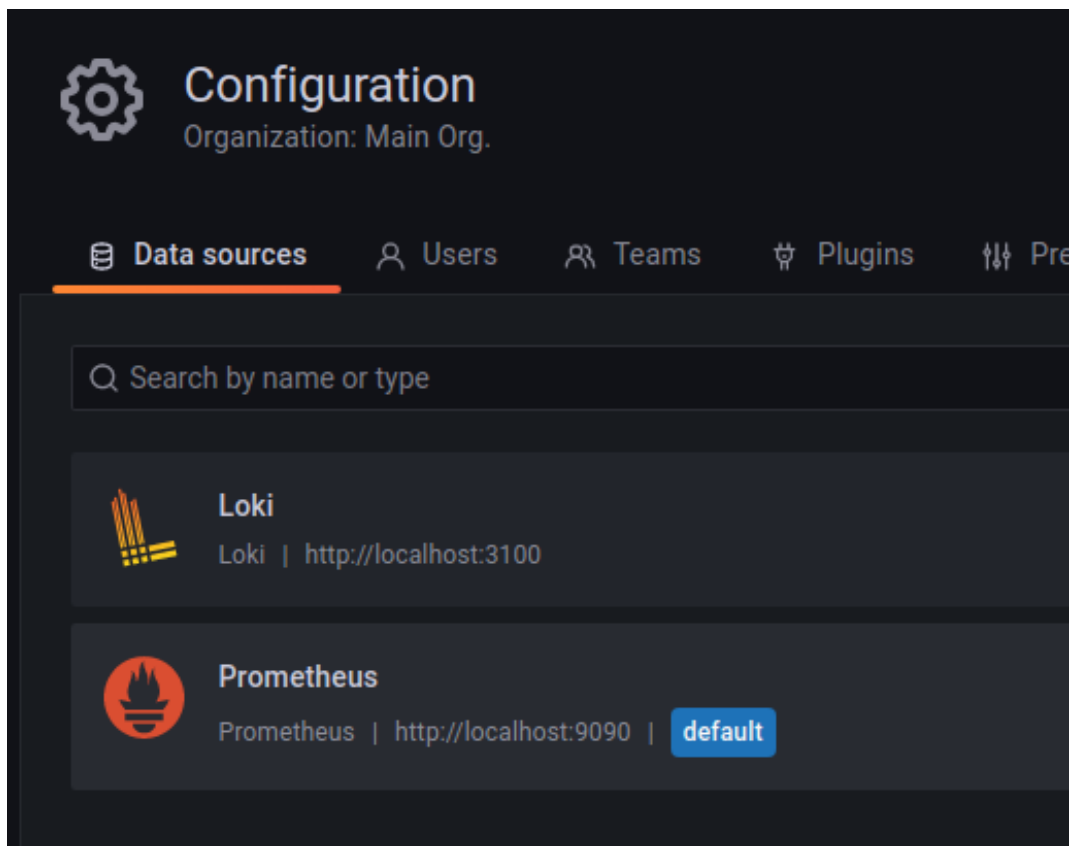
A Prometheus Data Source connectivity test that passes is shown below.


A Prometheus Data Source connectivity test that fails is shown below.



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.







 **Data Sources / Prometheus**
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



URL ⓘ http://localhost:9090


Allowed cookies ⓘ New tag (enter key to 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

+ Add header


Alerting

Manage alerts via Alerting UI 


Alertmanager data source ⓘ Choose 

Scrape Interval ⓘ 10s

Query timeout ⓘ 60s

HTTP Method ⓘ POST 

Misc

Disable metrics lookup ⓘ 

Custom query parameters ⓘ Example: max_source_resolution=5m&timeout=10

Exemplars

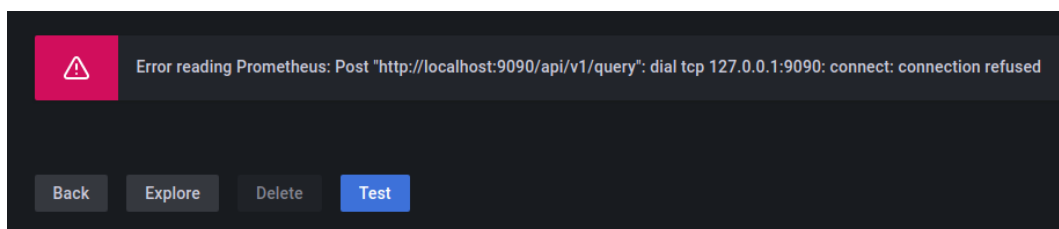
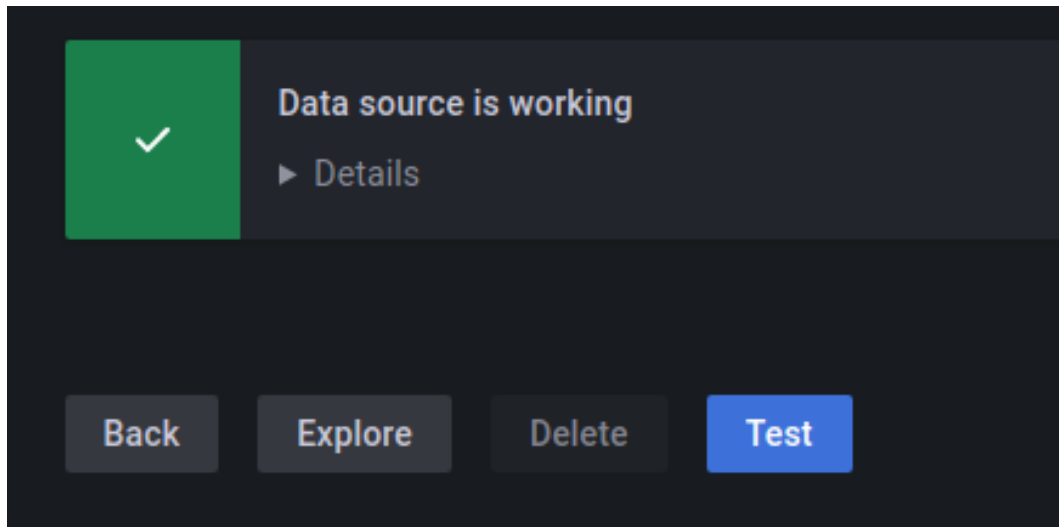
+ Add

Back

Explore

Delete

Test



Chapter 10

Glossary

Table 10.1: 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 11

Release Notes

Connex Observability Framework uses telemetry data to help identify and resolve potential issues in *Connex* applications. This product is not installed as part of a *Connex* 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 *Connex* applications.

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

11.1 Supported Platforms

The components of *RTI Connex Observability Framework* are supported on the following platforms:

- *RTI Observability Collector Service* is supported as a Docker image on linux/amd64 (x64Linux).
- *RTI Monitoring Library 2.0* is supported on all supported platforms in this release. See [Supported Platforms](#) in the *RTI Connex Core Libraries Release Notes*.

11.2 Compatibility

Connex Observability Framework is an optional product released with *RTI Connex 7.2.0*.

The current *Observability Framework* release is not compatible with *Connex 7.1.0*. This release works only with *Connex 7.2.0*.

11.3 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 11.1: 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.

11.4 What's New in 7.2.0

11.4.1 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>
  </qos_library>
</dds>
```

For more information about using XML QoS profiles, see [How to Load XML-Specified QoS Settings, in the RTI Connex DDS Core Libraries User's Manual](#)

11.4.2 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.

11.4.3 Support for RTI Observability Collector Service security

Starting with *RTI Connex 7.2.0*, *Collector Service* can be secured using the *RTI Security Plugins* to communicate with *Monitoring Library 2.0*. (see *Secured communications between RTI Monitoring Library 2.0 and RTI 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 *Security Plugins User's Manual*.

11.4.4 Name change from “RTI Observability Library” to “RTI Monitoring Library 2.0”

For details, see [RTI Connex Core Libraries What's New in 7.2.0](#).

11.4.5 Name change for some Observability metrics

This release changes the name of some metrics associated with *Connex* entities.

This change applies to the following eight metrics. See the *Metrics* section of this user manual for details about all available metrics.

Table 11.2: 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_times_tamp_dropped_samples_total	dds_datareader_cache_old_source_ts_dropped_samples_total
dds_datareader_cache_tolerance_source_times_tamp_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

11.4.6 Secured communications between RTI Monitoring Library 2.0 and RTI Observability Collector Service

For details, see [RTI Security Plugins What's New in 7.2.0](#).

11.4.7 Ability to set initial forwarding verbosity in MONITORING QoS policy

For details, see [RTI Connex Core Libraries What's New in 7.2.0](#).

11.4.8 Ability to set collector initial peers in MONITORING QoS policy

For details, see [RTI Connex Core Libraries What's New in 7.2.0](#).

11.4.9 Third-Party software changes

Observability Framework

The following third-party software is now used by the scripts that configure *Observability Framework*:

Table 11.3: 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 is now used by *Observability Collector Service*:

Table 11.4: Observability Collector Service Third-Party Software Changes

Third-Party Software	Version
Curl	8.1.2
Protobuf	21.12

The following third-party software used by *Observability Collector Service* has been upgraded:

Table 11.5: Collector Service Third-Party Software Upgrades

Third-Party Software	Previous Version	Current Version
CivetWeb	1.15	1.16
Grpc	1.38	1.52.1
OpenTelemetry C++	1.4.1	1.9.1

Docker containers for Observability Framework

The following third-party software is now used by the Docker containers created by *Observability Framework*:

Table 11.6: 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 11.7: 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 *Connex* products, see the “3rdPartySoftware” documents in your installation: <NDDSHOME>/doc/manuals/connex_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.

11.5 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 Connex Core Libraries Release Notes](#).

11.5.1 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]

11.5.2 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]

11.5.3 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]

11.5.4 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 *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]

11.5.5 Observability Collector Service could discard samples when monitoring large DDS applications

In the previous release, *RTI 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]

11.6 Previous Release

11.6.1 What's New in 7.1.0

Connex Observability Framework uses telemetry data to help identify and resolve potential issues in *Connex* applications. This product is not installed as part of a *Connex* 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 *Connex* 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 11.8: 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 11.9: 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.

Chapter 12

Copyrights and Notices

© 2023 Real-Time Innovations, Inc. All rights reserved. Oct 25, 2023

Trademarks

RTI, Real-Time Innovations, Connex, Connex Drive, NDDS, the RTI logo, 1RTI and the phrase, “Your Systems. Working as one.” are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished solely under and subject to RTI’s standard terms and conditions available at <https://www.rti.com/terms> and in accordance with your License Acknowledgement Certificate (LAC) and Maintenance and Support Certificate (MSC), except to the extent otherwise accepted in writing by a corporate officer of RTI.

Third-Party Software

RTI software may contain independent, third-party software or code that are subject to third-party license terms and conditions, including open source license terms and conditions. Copies of applicable third-party licenses and notices are located at community.rti.com/documentation. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

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/>