



Emit Telemetry Data to an OpenTelemetry-Compatible Monitoring Tool

Version 2025.1
2025-06-03

Emit Telemetry Data to an OpenTelemetry-Compatible Monitoring Tool

PDF generated on 2025-06-03

InterSystems IRIS® Version 2025.1

Copyright © 2025 InterSystems Corporation

All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble®, InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™, HealthShare® Health Connect Cloud™, InterSystems® Data Fabric Studio™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Congress Street, Boston, MA 02114, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: support@InterSystems.com

Table of Contents

Emit Telemetry Data to an OpenTelemetry-Compatible Monitoring Tool.....	1
1 Configure the Target Endpoint	1
2 Emit Metrics	1
3 Emit Logs	2
4 Emit Traces	3
4.1 Deactivate Tracing	7
5 Error Handling and Recovery	7

Emit Telemetry Data to an OpenTelemetry-Compatible Monitoring Tool

[OpenTelemetry](#) (OTel) is an open source framework and toolkit for generating, exporting, and collecting telemetry data.

On supported systems, this version of InterSystems IRIS leverages the OpenTelemetry SDK to provide support for exporting and emitting telemetry data as [OpenTelemetry Protocol](#) signals over HTTP (OTLP/HTTP) to the [OpenTelemetry Collector](#) or any other compatible monitoring tool.

Note: This feature is *not* available for macOS, Windows, and AIX systems in this version of InterSystems IRIS.

You can configure InterSystems IRIS to emit the following types of signals:

- [Metrics](#) — the measurements which you have configured the InterSystems IRIS [/api/monitor API](#) to collect.
- [Logs](#) — events which InterSystems IRIS records to either the [system messages log](#) or the [audit database](#).
- [Traces](#) — information about how a request moves through your application.

To learn more about how these different types of signals work within OTel, refer to the OTel documentation's pages for [metrics](#), [logs](#), and [traces](#).

InterSystems IRIS pushes these signals to the endpoint that you specify, as described in [Configure the Target Endpoint](#). InterSystems IRIS emits metrics and logs at a regular interval, based on a common configuration parameter; otherwise, you can enable and configure the emission of each type of signal independently, as described in the corresponding sections which follow.

1 Configure the Target Endpoint

To specify the endpoint to which an InterSystems IRIS instance sends OTLP/HTTP signals, set the environment variable `OTEL_EXPORTER_OTLP_ENDPOINT` on the instance's host system to the desired address. For instructions on setting environment variables, refer to your operating system's documentation.

If you enable OTLP/HTTP emission and an `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable is not set, the instance emits signals to the default endpoint for the [OpenTelemetry Collector](#)'s OTLP/HTTP receiver:

`http://localhost:4318`.

2 Emit Metrics

InterSystems IRIS can emit all of the metric events that the [/api/monitor API](#) collects (including your [custom application metrics](#)) to the OTLP/HTTP endpoint that you designate, at regular intervals.

To configure your instance to emit metric events:

1. Configure the instance to collect all of the metrics that you want to collect. If you want to collect interoperability production metrics, you must [manually enable them](#). You can also configure the instance to collect [custom application metrics](#).
2. Configure your instance's OpenTelemetry exporter to start emitting metrics when the instance starts. You can do this in any of the following ways:
 - From the [Management Portal](#): navigate to **System Administration > Configuration > Additional Settings > Monitor**, and select **Enable OTel Metrics**. Then, select **Save**.
 - Change the [OTELMetrics](#) parameter by modifying the Config.Monitor class (as described in the class reference) or by [editing the CPF file](#) directly.
3. If needed, modify the frequency at which the exporter will emit metrics. By default, the exporter emits signals every 10 seconds. You can change this interval in any of the following ways:
 - From the Management Portal: navigate to **System Administration > Configuration > Additional Settings > Monitor**. Update the **OTel Exporter Interval** field with the length of the desired interval, in seconds. Then, select **Save**.
 - Change the [OTELInterval](#) parameter by modifying the Config.Monitor class (as described in the class reference) or by [editing the CPF file](#) directly.
4. If you configured your instance in the preceding steps by modifying CPF parameters, restart the instance to allow your changes to take effect.

3 Emit Logs

InterSystems IRIS can emit OTLP/HTTP signals for the same categories of log events which would be part of a [structured log](#) file—namely, events which are recorded to the [system messages log](#) (messages.log) or to the [audit database](#). InterSystems IRIS emits structured log events to the OTLP/HTTP endpoint that you designate, at regular intervals.

To configure your instance to emit log events:

1. Configure your instance's OpenTelemetry exporter to start emitting log events when the instance starts. You can do this in any of the following ways:
 - From the [Management Portal](#): navigate to **System Administration > Configuration > Additional Settings > Monitor**, and select **Enable OTel Logs**. Then, select **Save**.
 - Change the [OTELLogs](#) parameter by modifying the Config.Monitor class (as described in the class reference) or by [editing the CPF file](#) directly.
2. As needed, configure the minimum severity level that a log event must meet or exceed in order to be emitted by the instance's OpenTelemetry exporter. Severity levels are the same as those [used in the structured log](#). The default severity level threshold is WARN. At this level, the exporter emits log events from the WARN, SEVERE, and FATAL levels; it does not emit log events from the DEBUG2, DEBUG, and INFO levels.

You can change the minimum severity level in any of the following ways:

- From the Management Portal: navigate to **System Administration > Configuration > Additional Settings > Monitor**. Select the desired threshold severity level from the **OTel Log Level** drop-down menu. Then, select **Save**.
- Change the [OTELLogLevel](#) parameter by modifying the Config.Monitor class (as described in the class reference) or by [editing the CPF file](#) directly.

3. If needed, modify the frequency at which the exporter will emit log events. By default, the exporter emits signals every 10 seconds.

You can change this interval in any of the following ways:

- From the Management Portal: navigate to **System Administration > Configuration > Additional Settings > Monitor**. Update the **OTel Exporter Interval** field with the length of the desired interval, in seconds. Then, select **Save**.
 - Change the [OTELInterval](#) parameter by modifying the Config.Monitor class (as described in the class reference) or by [editing the CPF file](#) directly.
4. If you configured your instance in the preceding steps by modifying CPF parameters, restart the instance to allow your changes to take effect.

4 Emit Traces

A trace records how a request moves through an application. It consists of one or more nestable spans, which represent the constituent units of work that the application performs as part of responding to the request. To record a trace, the application's code must include instruments which generate spans, populate them with information, and contextualize them as part of a continuous trace. For more information about the OpenTelemetry specification for traces, refer to the [OpenTelemetry documentation](#).

Within InterSystems IRIS, the %Trace package provides a straightforward API for instrumenting your application code to produce traces in a way that conforms to the OTel specification. Once instruments within your application are producing traces, InterSystems IRIS emits them to the OTLP/HTTP endpoint that you designate.

Edit your application code to produce traces using the %Trace API as follows:

1. Create an instance of the %Trace.TracerProvider class to serve as your application's [Tracer Provider](#).

The constructor for this class accepts an optional argument: an array which is used to set the TracerProvider object's ResourceAttributes property. If you wish to specify global attributes about the application, define an array containing the desired key-value pairs and then pass it to the constructor method by reference, as in the following example:

ObjectScript

```
set attributes("service.name") = "test_service"
set attributes("service.version") = "2.0"
set tracerProv = ##class(%Trace.TracerProvider)%.New(.attributes)
```

Python

```
attributes = {}
attributes["service.name"] = "test_service"
attributes["service.version"] = "2.0"
attrArray = iris.arrayref(attributes)
tracerProv = iris.cls('%Trace.TracerProvider')._New(.attrArray)
```

2. In most situations, it is preferable to instantiate a single TracerProvider object at startup, for shared use by instrumentation code across the namespace throughout the entire life cycle of the application. To do so, provide the newly created TracerProvider object to the **SetTracerProvider()** method of the %Trace.Provider class, as follows:

ObjectScript

```
do ##class(%Trace.Provider).SetTracerProvider(tracerProv)
```

Python

```
iris.cls('%Trace.Provider').SetTracerProvider(tracerProv)
```

When you need to access the `TracerProvider` object within your instrumentation code (as described in later steps), use the complementary **`GetTracerProvider()`** method to recall it:

ObjectScript

```
set tracerProv = ##class(%Trace.Provider).GetTracerProvider()
```

Python

```
tracerProv = iris.cls('%Trace.Provider').GetTracerProvider()
```

3. To instrument your application code, first use the `TracerProvider` object's `GetTracer()` method to instantiate a `Tracer`. (The `Tracer` object generates the actual spans of the trace.)

`GetTracer()` accepts two arguments, *Name* and *Version*. Use these arguments to uniquely identify the application or application component that you are tracing and specify its version number. For example:

ObjectScript

```
set tracer = tracerProv.GetTracer("service.orderprocessor", "2.0.2")
```

Python

```
tracer = tracerProv.GetTracer("service.orderprocessor", "2.0.2")
```

4. Start a root span using the `Tracer` object's `StartSpan()` method. In order, **`StartSpan()`** accepts the following arguments, which are used to define the properties of the span:
 - a. *Name* — A string, used to set the span's name field
 - b. *Parent* — (Optional.) A `%Trace.Context` object that identifies the span which you wish to specify as the parent span for the new span. If you do not provide a *Parent* and you have not previously specified an active span (as described in a later step), then the method initializes the span as a root span, with a unique Trace ID.
 - c. *Spankind* — (Optional.) A string, identifying the span as belonging to one of the OpenTelemetry specification's recognized [span kinds](#). If you do not provide a *Spankind*, the span is classified as `Internal` by default.
 - d. *Attributes* — (Optional.) An array of key-value pairs, passed by reference.
 - e. *StartTime* — (Optional.) A timestamp recording the span's start time, in **`$ZTIMESTAMP`** format. If not provided, *StartTime* is set to the current time.

For example, code which initializes a root span for processing a retail transaction may resemble the following:

ObjectScript

```
set rootAttr("customer.id") = customer.ID
set rootAttr("product.id") = product.ID
set rootSpan = tracer.StartSpan("order", , "Server", .rootAttr)
```

Python

```
rootspan = {}
rootAttr("customer.id") = customer.ID
rootAttr("product.id") = product.ID
rootAttrArray = iris.arrayref(rootAttr)
set rootSpan = tracer.StartSpan("order", , "Server", .rootAttrArray)
```


5. As needed, nest child spans hierarchically within this root span. For simple implementations, you can manually specify the parent span for a new span by creating a `%Trace.Context` object which identifies the desired parent as the `ActiveSpan`, and then providing that `Context` object as the *Parent* argument of **StartSpan()**.

However, attempting to manage context across lexical scopes using this manual approach would be impractical. For this reason, the `%Trace` API provides a dynamic scoping mechanism for managing context.

To manage the distributed context of your trace dynamically, perform the following steps:

- a. Designate the parent span as the "active" span using the `Tracer` object's `SetActiveSpan()` method, as follows:

ObjectScript

```
set rootScope = tracer.SetActiveSpan(rootSpan)
```

Python

```
rootScope = tracer.SetActiveSpan(rootSpan)
```

SetActiveSpan() returns an instance of the `%Trace.Scope` class. This `Scope` object acts as a record that the corresponding span is active.

- b. Start a new span by invoking **StartSpan()** *without* specifying a *Parent*, as follows:

ObjectScript

```
set childSpan1 = tracer.StartSpan("order_payproc")
```

Python

```
childSpan1 = tracer.StartSpan("order_payproc")
```

As long as the active span's `Scope` object remains in memory, **StartSpan()** initializes new spans as children of the active span by default when no other *Parent* is specified.

- c. To nest spans further, invoke **SetActiveSpan()** on a child span to designate it as the new active span and generate a new `Scope` object. The active span is identified by the newest `Scope` object which exists in memory at a given time. Therefore, once you have generated a `Scope` object for a new active span, **StartSpan()** will initialize new spans as its children by default.

Continuing the previous example, the following code starts a new span *childSpan2* as a child of *childSpan1* (which is itself a child of *rootSpan*):

ObjectScript

```
set child1Scope = tracer.SetActiveSpan(childSpan1)
set childSpan2 = tracer.StartSpan("order_payproc_addnewcard")
```

Python

```
child1Scope = tracer.SetActiveSpan(childSpan1)
childSpan2 = tracer.StartSpan("order_payproc_addnewcard")
```

- d. When you destroy the `Scope` object for the current active span, the span which was previously active becomes the default parent for **StartSpan()** once again (assuming you have not destroyed its `Scope` object as well). Continuing the previous examples, the following code starts a new span *childSpan3* as a child of *rootSpan* and a sibling of *childSpan1*:

ObjectScript

```
kill child1Scope
set childSpan3 = tracer.StartSpan("order_sendconfirm", , "Server")
```

Python

```
iris.execute('kill childScope')
childSpan3 = tracer.StartSpan("order_sendconfirm", , "Server")
```

6. As needed, define information about your spans. Available methods for this purpose include the following:

- Use the Span object's `AddEvent()` method to add a [span event](#).
- Use the Span object's `AddLink()` method to add a [span link](#).
- Modify the `TraceFlags` and `TraceState` properties of the Span object's `Context` property (an instance of `%Trace.SpanContext`) to modify a span's trace flags and trace state, respectively.

Note: The `TraceFlags` property provides a bit which specifies whether or not the span will be sampled for export. By defining the logic which sets the value of this bit conditionally, you can define a sampling algorithm for tracing within your application.

Continuing the previous examples, the following code enhances the "order_payproc" span (*childSpan1*) with a "paymentdeclined" event and a link to a hypothetical span named *paymentProcLivenessSpan*:

ObjectScript

```
set eventAttr("declined.reason")="Unknown error occurred."
do childSpan1.AddEvent("paymentdeclined", .eventAttr)
do childSpan1.AddLink(paymentProcLivenessSpan.Context)
```

Python

```
eventAttr = {}
eventAttr("declined.reason") = "Unknown error occurred."
eventAttrArray = iris.arrayref(eventAttr)
childSpan1.AddEvent("paymentdeclined", .eventAttrArray)
childSpan1.AddLink(paymentProcLivenessSpan.Context)
```

7. Before you end a span, update the span's status using the Span object's `SetStatus()` method, as in the following example:

ObjectScript

```
do childSpan1.SetStatus("Ok")
```

Python

```
childSpan1.SetStatus("Ok")
```

`SetStatus()` accepts one argument (a string) which can have three possible values corresponding to the three [span statuses](#) recognized by the OpenTelemetry specification.

8. End each span using the Span object's `End()` method. **End()** accepts an optional argument: a timestamp, in **\$TIMESTAMP** format. If a timestamp is provided, **End()** records that time as the end time for the span. Otherwise, **End()** sets the span's end time to the current time, as in the following example:

ObjectScript

```
do childSpan1.End()
```

Python

```
childSpan1.End()
```

InterSystems IRIS invokes the OpenTelemetry SDK to export the span when you end it, assuming that the 'sampled' bit in the span `Context` property's `TraceFlags` has been set appropriately.

- If you are using **SetActiveSpan()** to manage nested spans across lexical scopes (as suggested in a preceding step), destroy the Scope object for each active span after you end the span. Continuing the previous examples, the following code would conclude the trace encompassed by *rootSpan* and prepare the application to record a new trace:

ObjectScript

```
do rootSpan.SetStatus("Ok")
do rootSpan.End()
kill rootScope
```

Python

```
rootSpan.SetStatus("Ok")
rootSpan.End()
iris.execute('kill rootScope')
```

- As needed, import and recompile the code you have edited to enable tracing for your application.

4.1 Deactivate Tracing

To deactivate tracing for your application after you have instrumented it, perform the following steps:

- Edit your code so that it initializes any Tracer Provider that your application uses as an instance of `%Trace.NoopTracerProvider` (*instead of* `%Trace.TracerProvider`). Revisiting the example provided in the preceding [instrumentation instructions](#), deactivation would require you to change the line of code which sets the *tracerProv* object so that it reads as follows:

ObjectScript

```
set tracerProv = ##class(%Trace.NoopTracerProvider).%New(.attributes)
```

Python

```
tracerProv = iris.cls('%Trace.NoopTracerProvider')._New(.attrArray)
```

Assuming that *tracerProv* has been set as the Tracer Provider which serves the entire namespace, no further edits would be necessary.

- As needed, import and recompile the code you have edited to deactivate tracing for your application.

5 Error Handling and Recovery

If the OpenTelemetry-compatible tool which was receiving signals at the OTLP/HTTP endpoint becomes unavailable due to an unexpected system error, the InterSystems IRIS instance's OpenTelemetry exporter logs an error to the system messages log (messages.log). It then stops emitting signals from the instance.

The `SYS.Monitor.OTel` class provides methods to help you test whether communication at the OTLP/HTTP endpoint has been restored: `TestLogs()`, `TestMetrics()`, and `TestTraces()`.

Once you have resolved the cause of the error and restored the OTLP/HTTP connection, you can resume the emission of signals as follows:

- Open a Terminal session on the instance and navigate to the `%SYS` namespace.
- To resume the emission of metrics and logs, execute the following command:

Terminal

```
do ##class(SYS.Monitor.OTel).Start()
```

3. To resume the emission of traces, execute the following command:

```
do ##class(SYS.Monitor.OTel).EnableTraces()
```