ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG006

BETTER SOFTWARE OBSERVABILITY USING TANGO CONTROLS WITH OPENTELEMETRY - EXPERIENCE AT MAX IV

A. F. Joubert*, L. Zhu, B. Bertrand, MAX IV, Lund, Sweden

Abstract

Distributed software systems are complex and the interactions across multiple machines can be difficult to debug and monitor. Log messages are not enough for observability. We need more information about the communication between applications, how each one is executing, and its internal state. In practice, applications can be made more observable using software frameworks such as OpenTelemetry. The Tango Controls framework has built-in support for OpenTelemetry in C++ and Python since version 10.0.0. We are using it operationally at the MAX IV synchrotron. We provide examples of the traces, trends, and other data available when running at scale on a beamline with hundreds of devices. We report on the compute and performance impact for client and server software applications, as well as practical issues. For the backend servers that ingest and query the telemetry data (running Grafana Tempo for traces and Grafana Loki for logs) we report on the compute resources required..

INTRODUCTION

The distributed control systems used at large experimental physics facilities are complex with a large number of applications interacting between many different machines. The MAX IV synchrotron in Lund, Sweden, is no exception in this regard. Our Tango Controls [1] system includes nearly 400 machines hosting 26k Tango devices. This can be challenging to debug using just log messages.

We need to understand the communication between applications, how each one is executing, its internal state, and details about any errors, in order to get a picture of how the complete system is operating. The software term for this is *observability* [2]. The three types of data, or signals, that applications can emit to achieve this are *traces*, *metrics* and *logs* [3]. Recently, a fourth signal, *profiling*, has started to become available [4].

We will focus on traces. These allow us to see the causal relationship between remote calls across various applications in the control system, in other words, distributed traces. Traces show what is happening within each application as it processes the request, and how different applications interact.

Figure 1 shows an example of a distributed trace from a Tango Control system. Each coloured rectangle is a *span*, and these show which method in the application was executing and how long it took. Different colours indicate different actors in the call. The client application, the PyTango library [5], the `Leader` device and the `Follower` device. Each span includes many details such as filepath and line number, host name, thread ID, version information, etc.

* anton.joubert@maxiv.lu.se

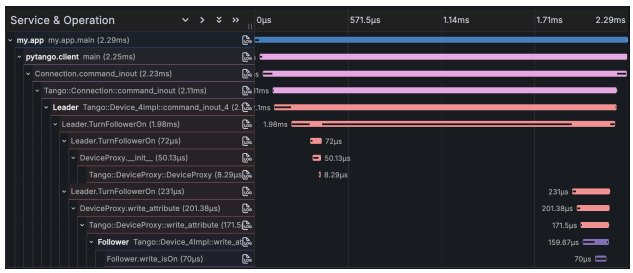

Figure 1: Example of a distributed trace across three applications (client script and two servers). The client script commands the `Leader` server to turn on a `Follower` server. The vertical axis shows a simplified call stack, and the horizontal axis shows the elapsed time since the first call.

OpenTelemetry

OpenTelemetry [6] is an open-source and vendor-agnostic software framework that provides a practical way to realise observability. The project maintains specifications and conventions, and then implements this in many different programming languages. Their software libraries allow applications and 3rd-party libraries to be instrumented.

In order to instrument a software library, one has to add OpenTelemetry function calls at key places of interest. For example, starting a new span, with the necessary attributes, when an external call is received. Once a library is instrumented, applications that use the library can typically do so without any changes to their own code. When the application runs, the telemetry signals are automatically emitted.

Telemetry Backends

Given an instrumented application, the telemetry signals have to be sent somewhere. The immediate destination is a *collector* service, which then passes it on to a telemetry backend. The backend typically maintains a database for all the signals (traces, metrics, etc.), and provides a web interface to view the information. Some backends include their own collectors. The basic architecture is shown in Fig. 2.

There are many companies providing such backends [7–11]. We use Grafana [12], specifically their *Tempo* product for traces and metrics, and *Loki* for logs).

IMPLEMENTATION IN TANGO CONTROLS

Tango Controls added support for OpenTelemetry starting in version 10.0.0. Traces and logs can be sent to a telemetry backend. Metrics and profiling have not been added yet. A high-level specification of features in Tango Controls is in the Request For Comment 19 document [13].

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG006

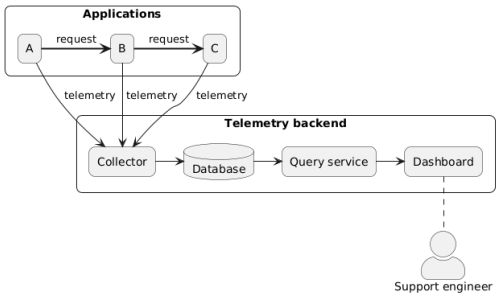

Figure 2: Typical architecture of a telemetry backend receiving signals from instrumented applications.

The most fundamental change required to implement the distributed tracing was a mechanism to pass the *context* of the traces across process boundaries. This is known as *context propagation* [14]. The context is a small amount of data that allows correlation of requests across the system. Typically, two unique identifiers (IDs) and some flags. Previously, users had to work around the missing context by passing their own IDs along with command data [15] — this approach had severe limitations and a non-standard implementation.

Tango follows the W3C Trace Context [16] specification recommended by OpenTelemetry. An example string is: `00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01`. This results in at least 55 bytes of network traffic overhead with each remote call. The trace context was added into Tango's CORBA [17] interface description language specification from version 6 (supported from Tango 10.0.0).

Significant changes were required in Tango's core C++, Python, and Java libraries. The OpenTelemetry libraries were used, and the trace context and spans handled at 100+ points within these libraries. Just the initial C++ work took around 6 months, with 4.2k new lines of code [18]. Adding support to emit logs to OpenTelemetry collectors was trivial, in comparison to the work to support traces.

The end result of all this work is that OpenTelemetry support can now be enabled trivially for existing Tango devices and clients with zero code changes. Simply, by using the latest version of Tango.

If the control system has a mixture of applications running old and new Tango software versions, communication is backwards compatible, as always. The trace context only propagates between client and server if both are running the new software. This means traces will be disjoint.

Tango adopted an opt-in approach to this major feature — first at compile-time, and then via environment variables. Users also have to deploy their own OpenTelemetry backend, and provide the network addresses of the collectors (also via environment variables). There is support for sending signals over HTTP, HTTPS and gRPC [19] protocols.

An unexpected time sink was the effort related to packaging. Adding OpenTelemetry added many new software library dependencies. Updating all the continuous integration and packaging tooling around the Tango core projects was time consuming. Keeping these packages updated is an additional maintenance burden.

An unexpected bonus for the core Tango developers was the new insight that the OpenTelemetry traces provide about the complex interaction within the code. It has helped give developers a better understanding of what is really happening "behind the scenes". This new tool will help in discovering and improving performance issues.

EXAMPLES

In this section we review a small selection of examples taken from beamlines at MAX IV. The screenshots show data from the Grafana Tempo and Grafana Loki web user interfaces.

Camera

We have a desktop graphical user interface (GUI) application connected to a Tango device for a Basler [20] camera, shown in Fig. 3. The GUI periodically requests readings of the attributes it needs to display. A distributed trace related to such a request is shown in Fig. 4. We see the client-side method calls in purple, followed by server-side method calls in blue. Two of the server-side spans have a noticeably longer duration — these require calls to the camera hardware, while the rest are just accessing values in memory. In this example, the client is using an asynchronous read request, so once it has sent the request, it is no longer waiting (the purple traces end early). The detail of a single span is shown in Fig. 5.

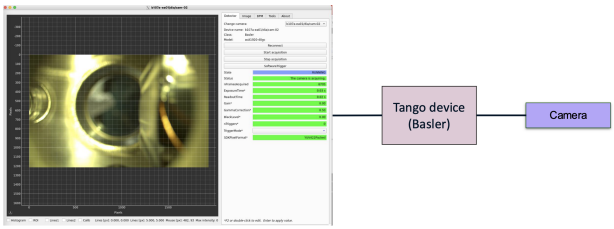

Figure 3: Block diagram showing logical network connections between desktop GUI, Tango device and hardware.

Filtering the traces for display is critical as there is typically a huge volume available (3k spans per second, in this beamline). Grafana Tempo allows filtering via simple graphical widgets, e. g., service name, status, duration, tags, etc. For more powerful searches, it has a custom query language [21]. We can filter for slower-than-normal traces with something like `{resource.service.name="Basler" && duration>10ms}`. Figure 6 shows an example of one of those traces. Here we see an initial delay of almost 8 ms before the attributes start being read. This indicates that the Tango device was busy servicing another request, since it is configured to limit access to one client at a time.

Another benefit of the traces is that they capture errors. For example, if a Python method throws an exception, the full traceback is available in the span, and the overall trace is

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG006

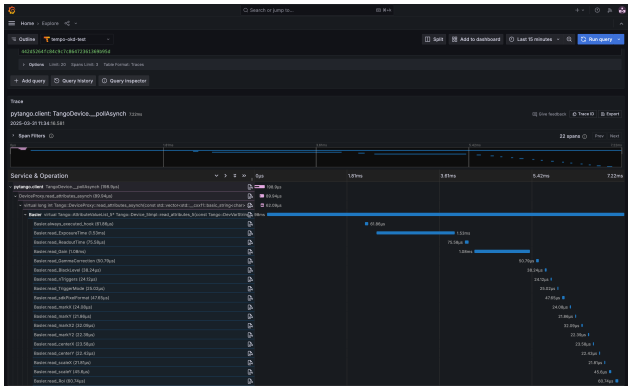

Figure 4: Example trace showing the calls when the GUI client triggers a periodic read of many attributes from the camera's `Balser` Tango device. The long calls for exposure time and gain require communication with the camera hardware.

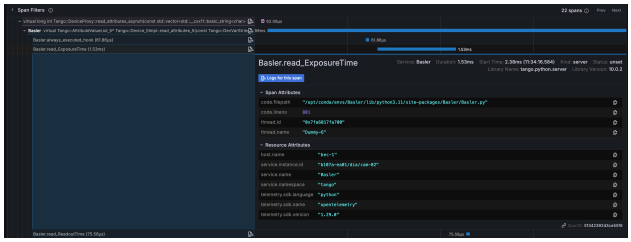

Figure 5: Example of the details available within one span, from the `Balser` Tango device in Fig. 4.

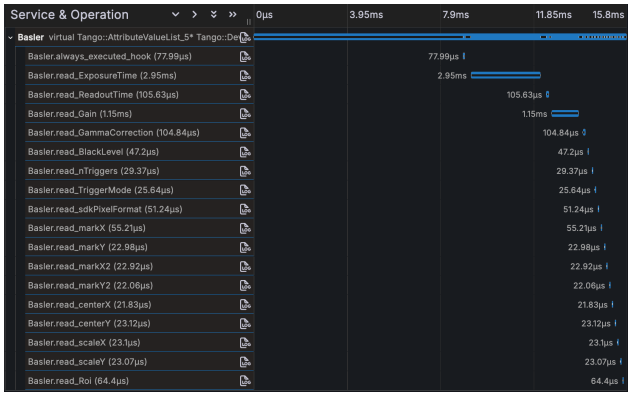

Figure 6: Example of a slower than normal trace for the `Balser` Tango device when GUI reads attributes. There is an initial delay before the attributes are read.

marked with an error status. This allows filtering on errors, and quick diagnosis.

Logs

Another powerful feature of OpenTelemetry is the ability to link log messages with specific spans, and *vice versa*. In Grafana Tempo, we simply select the log icon next to a specific span and it opens a view of the relevant logs in Grafana Loki. This is illustrated in Fig. 7.

Metrics From Traces

Grafana Tempo can create metrics from traces internally, allowing time series plots of almost any of the attributes recorded by the spans. A simple example of the rate of spans, grouped by Tango device class (OpenTelemetry service name) is shown in Fig. 8.

Diagnosing a Real Issue

A different beamline reported that when multiple GUIs were reading attributes from a specific Tango device (`BrooksMFC` class), it took a considerable time to respond. We had a suspicion that the cause was a lack a caching in the device, so that each read resulted in access to hardware. However, we used the telemetry feature to confirm this diagnosis.

This required a simple update of the PyTango library in the environments running the Tango device server and client application, and then restarting the applications with telemetry enabled.

We could immediately see that each GUI was reading 12 attributes periodically, and the device's internal polling was reading another 8 attributes periodically. The traces also showed that the access to hardware for reading each attribute was generally around 50 ms, but randomly spiked to around 400 ms — an example is shown in Fig. 9. We could see that when different client requests coincided, the later caller was blocked until the first client had been served (see Fig. 10).

A time-series plot (Fig. 11) shows the statistics of the duration for the reads. We could see 90% of the reads were under 0.5 s, but the slowest 10% took up to 2 seconds. This confirmed the anecdotes of "slow" user experience with data.

The telemetry information indicated we could fix the problem by including the GUI attributes in the device's polling configuration. The GUI would then read cached values with a fast response (but increased latency).

Service Graph

The distributed trace data can be used to infer a graph of the connected services in the monitored system. In a pure Tango Control system, the services are the classes of Tango devices and user applications.

Grafana Tempo provides such a feature. It renders a graph of the services, showing the connections between them, the rate of requests, and fraction of errors. This is useful for high-level monitoring. Clicking on a service node allows quick access to metric timelines. Figure 12 illustrates some of this.

PERFORMANCE IMPACT

Performance is important for the core Tango Controls libraries, so the impact of the telemetry feature was investigated. A simple benchmark was run using the following scenario:

- Measure the time for a client to read a single double precision floating point value from a server.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG006

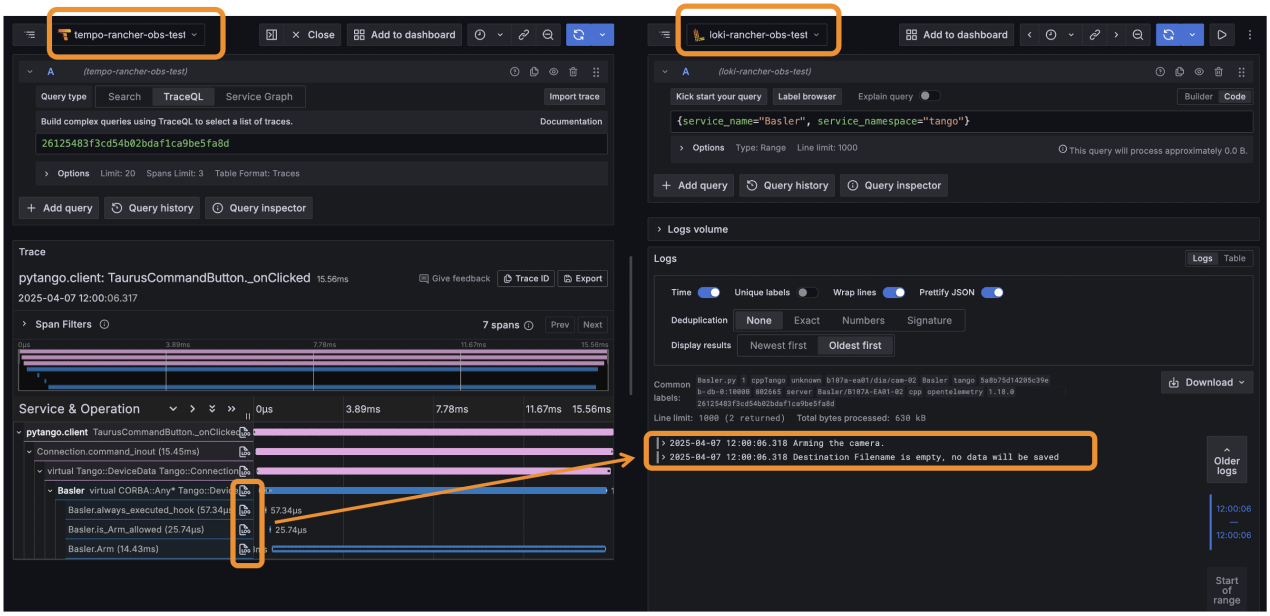

Figure 7: Example of accessing the log message associated with a single span. Clicking a log icon in the left pane (Tempo), opens the right pane (Loki) filtering for only the logs of interest (bottom right).

Figure 8: Example of the rate of spans per second for various classes of Tango device.

- Single host running server, client, Tango database, and telemetry backend (i. e., using loopback network).
- Times averaged over 30k reads.
- Traces and logs sent over gRPC.
- Traces dropped if in-process buffer full

The results of the benchmark are shown in Table 1. Comparing column A and B, we see that having the feature compiled in, but disabled has an insignificant effect. Comparing column A and C, we see significant overhead when telemetry is enabled. However, differences on the order of tens of microseconds will not be noticeable in practice, for all but the most extreme cases. The Python implementation is noticeably slower, which is to be expected as it wraps the C++ implementation, and adds Python-level OpenTelemetry calls as well. Still, a few hundred microseconds more per read is generally not that noticeable, and on the same order as the network ping time between the different machines.

Another way to view the impact is to analyse the CPU usage on the computer hosting the Tango device servers. Figure 13 shows the CPU usage trends with telemetry disabled

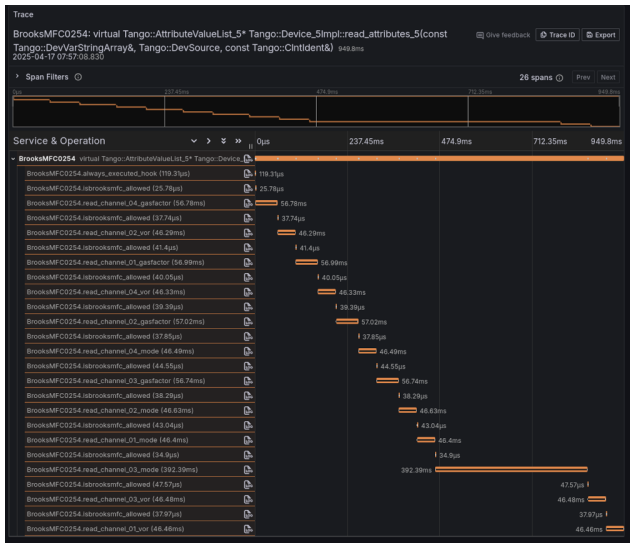

Figure 9: Example of the traces from a GUI reading 12 attributes from the `BrooksMFC` Tango device. One of the attributes takes almost 400 ms to read.

and enabled. There is no significant change when telemetry enabled.

The final way we considered the impact was qualitatively. We had telemetry enabled on all Tango device servers for five months at a beamline at MAX IV. The beamline staff did not report any unusual slow down or issues running their scientific experiments. Their only comment was some new error log messages that were seen from time to time when traces could not be sent to the collector (this has since been addressed). In other words, enabling telemetry was effectively transparent for them.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG006

Table 1: Performance Comparison Under Different Telemetry and Environment Configurations. The Reference Is *(A) Compilation Off* so It Has Zero Overhead. The Values After the ± Indicate the Standard Deviation

Server	Client	Telemetry OFF (A) Compilation off Time [µs]	Telemetry OFF (A) Compilation off Overhead [µs]	Telemetry OFF (B) Environment var off Time [µs]	Telemetry OFF (B) Environment var off Overhead [µs]	Telemetry ON (C) Environment var on Time [µs]	Telemetry ON (C) Environment var on Overhead [µs]
C++	**C++**	10 ± 3	0	11 ± 4	1	26 ± 11	16
C++	**Python**	20 ± 4	0	21 ± 2	1	115 ± 61	95
Python	**C++**	43 ± 13	0	42 ± 11	-1	181 ± 85	139
Python	**Python**	61 ± 31	0	59 ± 19	-2	313 ± 105	252

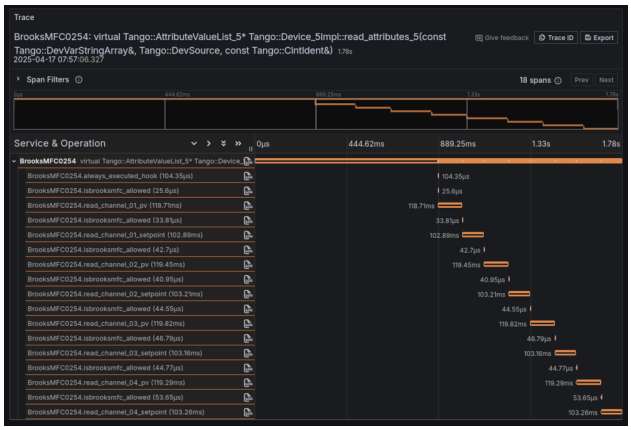

Figure 10: Example of traces from the `BrooksMFC` Tango device's internal polling reading 8 attributes. There is a significant delay at the beginning, while another client is serviced.

Figure 11: Example of the time duration quantiles for spans accessing the `BrooksMFC` Tango device. The fastest 50% of calls are under 1 ms, the fastest 90% of calls are under 0.5 s, and the 99% quantile shows the slowest calls take between 1 s and 2 s.

COMPUTE RESOURCES

Our Grafana Tempo and Loki backends were deployed in our production Kubernetes [22] cluster. For data reported in this paper, we used the following for our test environment, and measured the rates below:

- Resources (2 replicas)
  - 8 cores @ 2 GHz each
  - 16 GB RAM (250 MB per component: distributor, ingestor, compactor, querier, etc.)
- Storage
  - 50 GB generated per day (only kept 150 GB)
  - 250 read/writes per second
  - S3 storage
  - Ceph [23] using 4 CPU cores, 20 GB RAM
  - 10 krpm SAS [24] drives
- Network
  - 10 Gbps Ethernet
  - 3k spans/second (from 230 HTTP requests/sec)

We observed that the above resources were insufficient when querying. Queries slowed down from a couple of seconds to tens of seconds when the system was heavily loaded. Ingesting of traces was not a problem.

Telemetry was enabled for only 1 out of 16 beamlines at MAX IV. That beamline had 900 Tango devices and 1

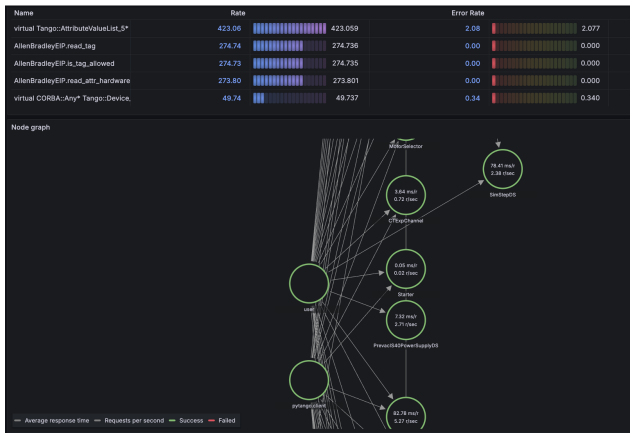

Figure 12: A portion of the service graph showing services, connections, and request rates.

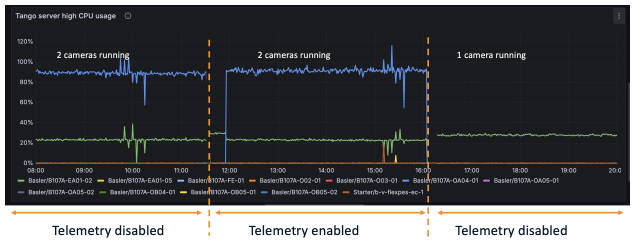

Figure 13: CPU usage trend for a single host running multiple Tango camera device servers, with up to two capturing images. Telemetry was initially disabled, then enabled for four hours, and then disabled again.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG006

control room workstation. Logging was only enabled on 1 device, so the Loki backend had very little load.

FUTURE WORK

A number of improvements have become obvious for Tango Controls:

- Allow telemetry to be enabled and disabled at runtime, rather than just at startup.
- Add more details to spans such as the command input arguments, new property values, and calling client's information.
- Instrument more of the core functions and ensure trace context is maintained when work is passed to a background thread.
- Instrument more of the applications in the Tango Controls ecosystem. For example, include details of requests from web-based user interfaces.
- Add support for metrics and profiling signals.

For MAX IV's infrastructure, we envisage the following:

- Finalise a new monitoring cluster, which will handle the telemetry traffic from all beamlines and many other monitoring functions.
- Implement tail sampling [25], using Grafana Alloy [26]. This can greatly reduce the volume of traces we need to process. E. g., keep trace slower than 5 ms, keep all error traces, keep 5% of other traces. That would exclude around 75% of traces. It would also be useful to allow custom filtering based on Tango device class, so full detail could be obtained, when necessary.
- Create custom dashboards to simplify monitoring of common issues.

The new monitoring cluster will have 5 machines. Each with dual CPUs, 512 GB RAM, 13 TB effective storage (on 5 SAS solid state drives), and 4x10 Gbps Ethernet ports. Grafana Tempo and Loki will run on 3 of these machines, sharing the resource with other monitoring tools. Initially, there would be 1.5 TB of Ceph storage allocated. The resource allocation will be adjusted as we learn more.

CONCLUSION

The concept of observability was discussed and its importance for engineers maintaining large distributed control systems. We provided a brief overview of the OpenTelemetry framework, how it was added to the core Tango libraries, and the effort involved.

We provided a few examples of the power of this feature using the Tango control system at one of MAX IV's beamlines. The performance impact on the control system is minimal, but the compute resources required for the backend can be significant.

The groundwork has been done, but there are still many improvements to be made. As a community we have to learn how to best use this new tool to allow us and diagnose and fix problems more quickly.

REFERENCES

[1] Tango Controls, https://www.tango-controls.org

[2] H. Joslyn, *Cloud Native Observability for DevOps Teams*. The New Stack, 2021.

[3] C. Sridharan, *Distributed Systems Observability: A Guide to Building Robust Systems*. Santa Rosa, CA, USA: O'Reilly Media, Inc., 2018.

[4] OpenTelemetry announces support for profiling, https://opentelemetry.io/blog/2024/profiling

[5] PyTango documentation, https://tango-controls.readthedocs.io/projects/pytango

[6] OpenTelemetry, https://opentelemetry.io

[7] Jaeger, https://www.jaegertracing.io

[8] Elastic, https://www.elastic.co

[9] Dash0, https://www.dash0.com

[10] SigNoz, https://signoz.io

[11] Zipkin, https://zipkin.io

[12] Grafana, https://grafana.com

[13] Tango Controls RFC-19, https://gitlab.com/tango-controls/rfc/-/blob/main/19/Telemetry.md

[14] Context propagation, https://opentelemetry.io/docs/concepts/context-propagation/.

[15] S. N. Twum *et al.*, "Implementing an Event Tracing Solution with Consistently Formatted Logs for the SKA Telescope Control System", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 311–316. doi:10.18429/JACoW-ICALEPCS2021-TUBL02

[16] W3C Trace Context recommendation, https://www.w3.org/TR/trace-context/.

[17] Common Object Request Broker Architecture specification, https://www.omg.org/spec/CORBA/.

[18] cppTango OpenTelemetry merge request, https://gitlab.com/tango-controls/cppTango/-/merge_requests/1197

[19] gRPC, https://grpc.io

[20] Basler AG, https://www.baslerweb.com

[21] Grafana TraceQL Documentation, https://grafana.com/docs/tempo/latest/traceql/.

[22] Kubernetes, https://kubernetes.io

[23] Ceph, https://ceph.io

[24] SAS, https://en.wikipedia.org/wiki/Serial_Attached_SCSI

[25] OpenTelemetry Sampling, https://opentelemetry.io/docs/concepts/sampling/.

[26] Grafana Alloy, https://grafana.com/docs/alloy/latest/.