

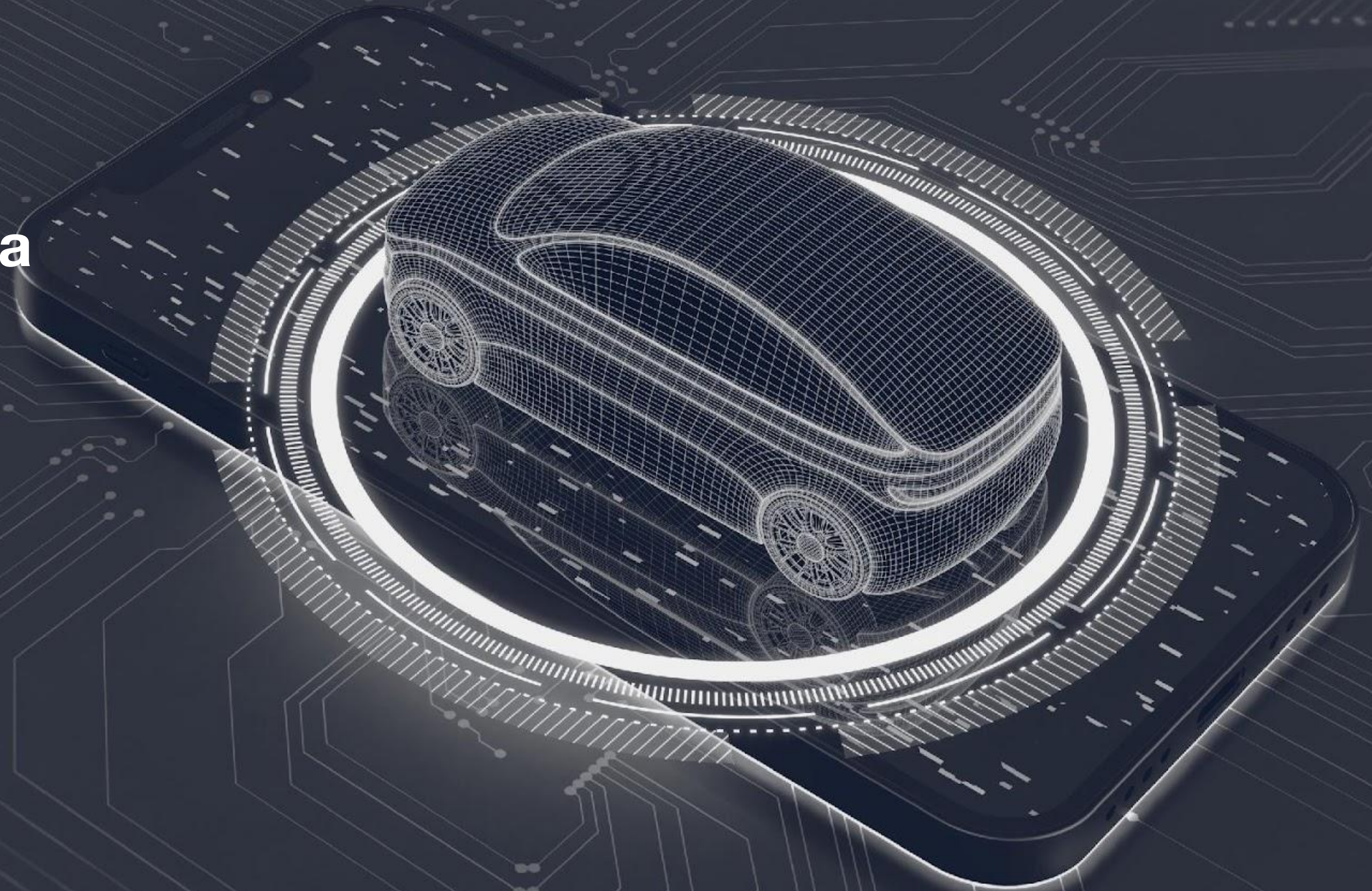
# ROSCon 2023

---

Apex.AI

Improving your application's  
algorithms and optimizing  
performance using trace data

Christophe Bédard  
October 20, 2023





# Agenda

---

We'll learn how to use trace data to improve performance

- 1. Introduction**
- 2. Tracing**
- 3. Instrumentation**
- 4. Trace data analysis**
- 5. Examples and use-cases**
- 6. Conclusion**

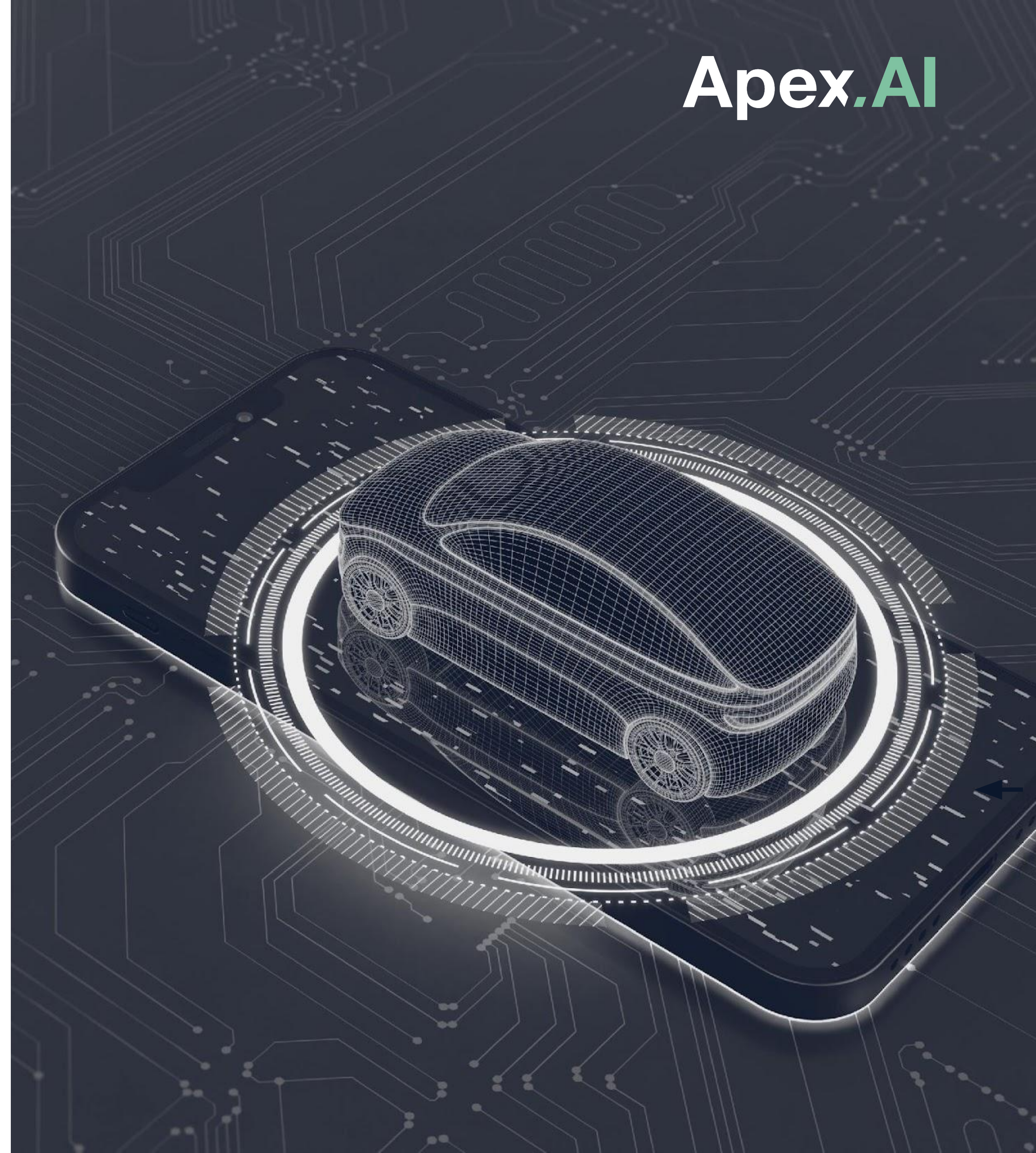


# Introduction

---

- Performance analysis
- Understand what happened during execution
- Extract high-level information
- Find cause of bugs
- Identify misconfigurations
- Optimize performance
- Extract various performance metrics

Apex.AI





# Tracing

- Low-level software tracing
  - Fast low-level recording at runtime
  - Low performance impact
  - Payload is usually raw binary data
  - Needs to be processed for it to be useful
- Need to instrument source code in order to collect data
  - Applications
  - Linux kernel (built-in)
  - Drivers
  - Etc.
- `ros2_tracing`: tracing instrumentation and tools for ROS 2
- This presentation is about trace data processing, not `ros2_tracing` itself
  - See ROS World 2021 presentation: *Tracing ROS 2 with ros2\_tracing* ([vimeo.com/652633418](https://vimeo.com/652633418))
  - Or the paper:  
*ros2\_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2* ([doi.org/10.1109/LRA.2022.3174346](https://doi.org/10.1109/LRA.2022.3174346))

- Record low-level execution information
- Trace data needs to be processed

# Why trace? When to trace?

- Tracing is one of many tools in our huge toolbox
- Not always the right tool
  - Optimize a specific function or lines of code → profiler
  - Debug a specific function or lines of code → debugger
  - What happened at 9:27 pm? → logs
- Tracing examples
  - One instance is taking longer than usual: I/O, kernel scheduling, etc.
  - Anomalies or unexpected behaviour with messages or system: logic bug, executor misconfiguration, etc.
- Especially useful to understand complex systems
  - Like large and/or distributed systems
  - Since complex systems can make debuggers less effective
- Also useful in general to visualize a system
  - Might give hints to optimize your system
  - Even if you're not necessarily looking for performance issues

Tracing is useful to understand the execution of complex systems.

# Instrumentation — `ros2_tracing`

- Instrumentation is built into ROS 2 by default on Linux starting from ROS 2 Iron Irwini
- Information about the main elements of ROS 2
- Objects
  - Node, publisher, subscription, timer
- Events
  - Callback execution (subscription, timer)
  - Message publication, message taking
  - Internal executor phases, etc.
- Uses the LTTng tracer for instrumentation and recording

- Information about callbacks and messages
- Now included by default on Linux as of ROS 2 Iron Irwini



# Instrumentation — Custom

1. Define your tracepoints in a tracepoint provider (shared library)
  - a. Tracepoint names
  - b. Arguments
2. Add tracepoints to your code
3. Run your application and collect data

```
$ ros2 trace --ust 'ros2:*' 'my_app:*
```

- This is a simplified version
- For more information, see:  
[lttng.org/docs/v2.13/#doc-instrumenting](https://lttng.org/docs/v2.13/#doc-instrumenting)

```
// tp.h: tracepoint definition in tp provider header
#include <lttng/tracepoint.h>
LTTNG_UST_TRACEPOINT_EVENT(
    my_app, my_tracepoint,
    LTTNG_UST_TP_ARGS(int, count_arg),
    LTTNG_UST_TP_FIELDS(
        lttng_ust_field_integer(int, count, count_arg))
)

// Insert tracepoint somewhere in your code
#include "tp.h"
lttng_ust_tracepoint(my_app, my_tracepoint, 42);
```

# Trace data analysis — How to use trace data?

- Combining trace data from multiple sources helps understand the execution
  - Multiple applications, including ROS 2 + additional application-specific info
  - Linux kernel
  - Distributed systems (with time synchronization)
- Examples
  - ROS 2: callback executions, message publications
  - Linux kernel: scheduling, I/O, system calls, etc.
- Add information specific to your own nodes
  - Instrument and trace your nodes
  - Can provide information about the processing performed by your nodes
- Trace processing tools
  - `tracetools_analysis`: very basic Python API
  - Eclipse Trace Compass: powerful trace viewer and analysis framework
- Trace Compass can display Linux kernel and ROS 2 trace data
  - And more!

Combine and visualize trace data from multiple sources or layers.



# Example

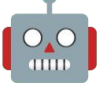




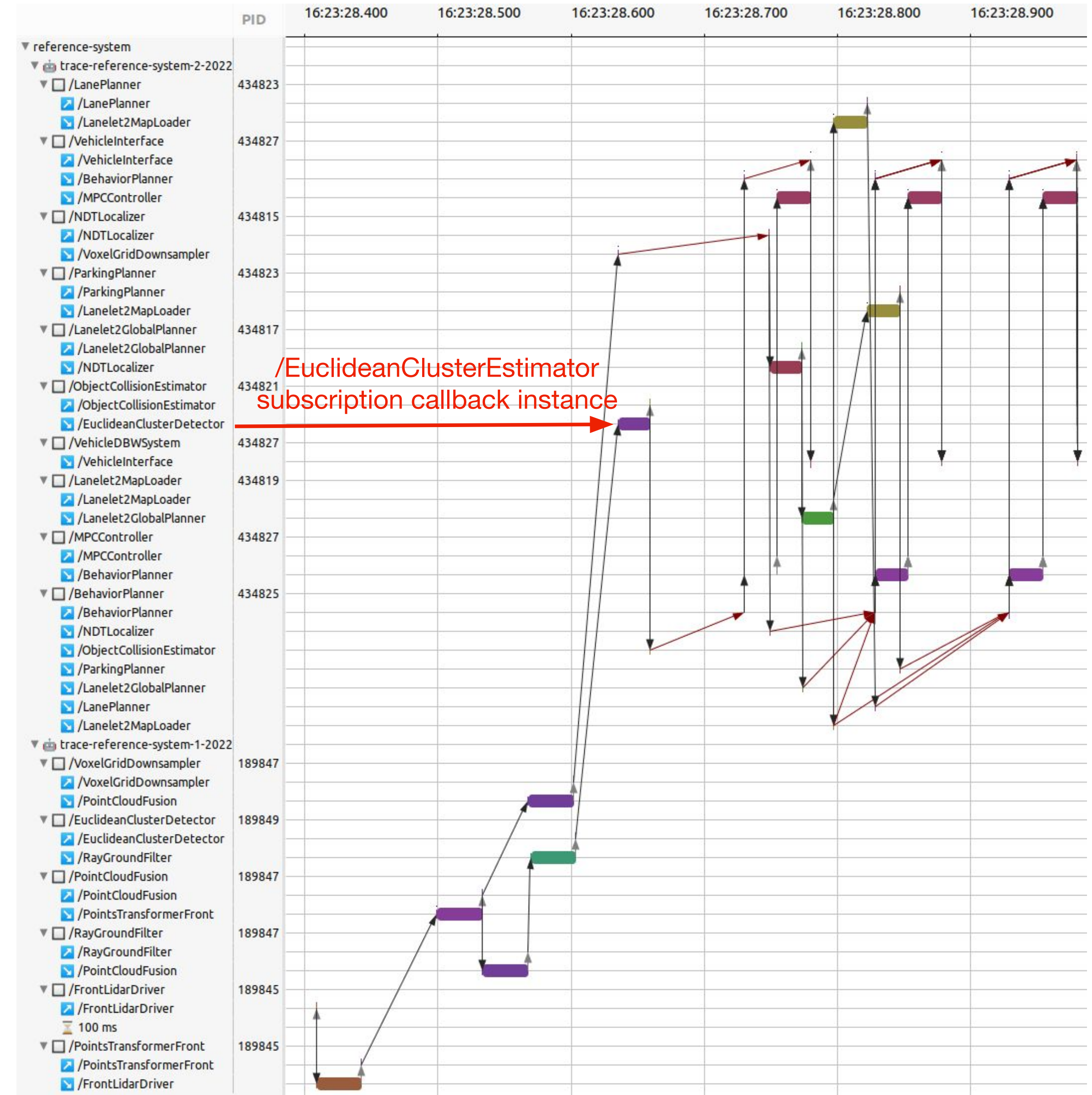
- Viewing ROS 2 trace data with Trace Compass
- Horizontal axis is time
- Rows
  -  system (1 trace/system)
    -  node
      -  timer
      -  subscription
      -  publisher
- Shows
  - Rectangles: callback executions
  - Arrows: message publications
- Autoware reference\_system
- For a single end-to-end process pipeline instance starting from one LiDAR message

Figure 1. Callbacks and messages over time in Trace Compass.





# Example (2)

- State of executors over time
  - Green: executing (callback)
  - Orange: waiting for work
- 1 executor/process
- Some executors are busier than others
- Could explain message processing delays
  - Message reception → callback



Figure 2. State of single-threaded executors over time.  
Figure 3. State of multi-threaded and single-threaded executors over time.



# Example (3)

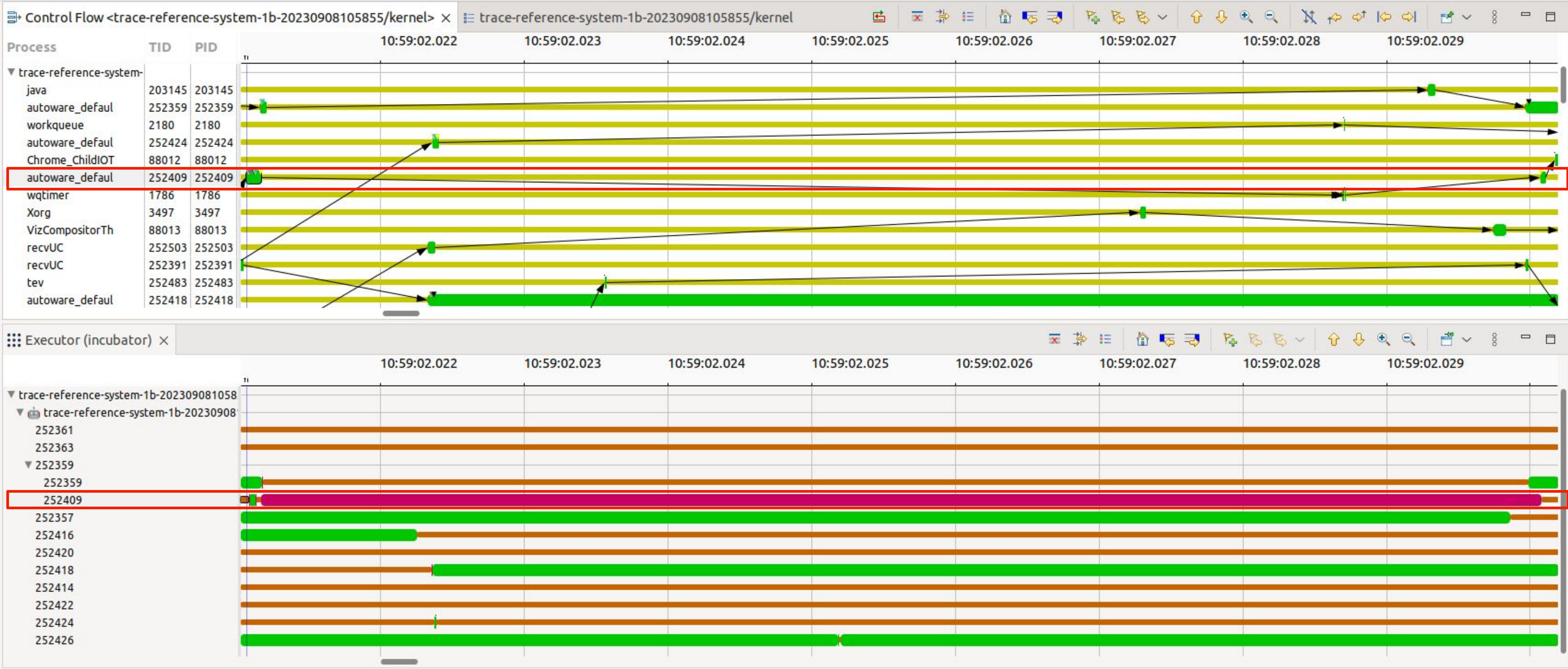


Figure 4. Kernel scheduling view (control flow) vs. executor view: thread scheduling explains some executor delays.



# Example – Taking it to the next level

- We've implemented a Trace Compass plugin for Apex.Grace, our fork of ROS 2
- This is somewhat specific to our custom executor
  - Executor-centric view (below) vs. node-centric (previously)
  - See presentation from ROS World 2021 executor workshop: *Executor with wait-set and polling subscription*

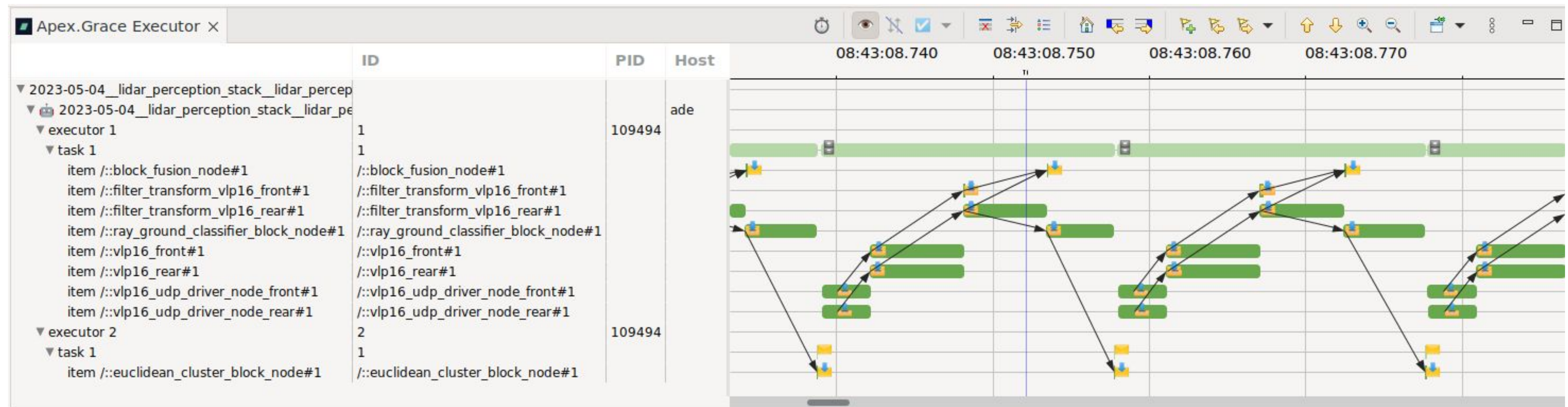


Figure 5. State of Apex.Grace executor and message publications over time.



# Example – Taking it to the next level (2)

- We can see the callback executions over time
  - What if we want to know more about what happens in our callback?
- Custom tracepoints to provide information about processing done inside a function or callback
- Display durations over time

```
lttng_ust_tracepoint(interval_start, "camera msg", (uint64_t) id);
// Do some processing
// ...
lttng_ust_tracepoint(interval_end, "camera msg", (uint64_t) id);
```

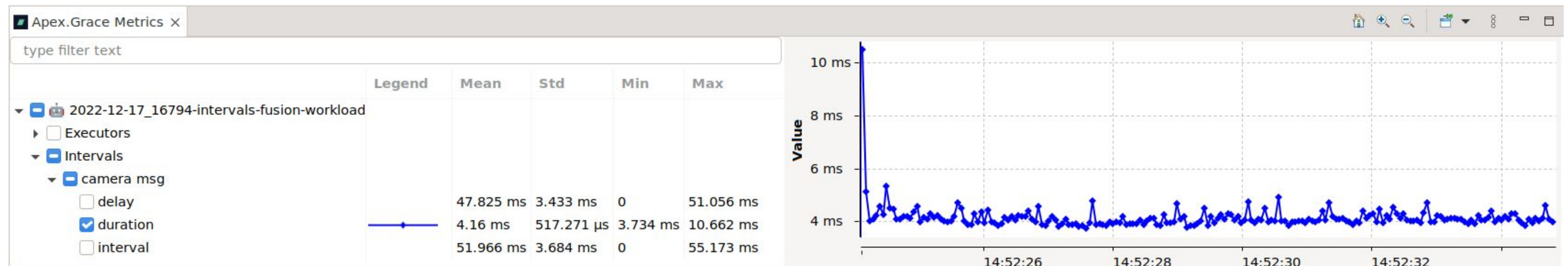


Figure 6. Duration (interval\_end - interval\_start) values over time.

# Use-cases — Our own experience

- Using tracing and our Trace Compass-based tool, we've identified and fixed multiple issues
- Executor misconfiguration
  - Very visually obvious, but would've never guessed otherwise
  - I wasn't even looking for an issue
- Performance issue
  - Could clearly see that a node couldn't process messages fast enough: bottleneck
- Performance instability due to bad execution logic
  - Would've been hard to find otherwise
- Execution strategy optimization for our LiDAR stack
  - Compare different execution strategies both visually and using KPIs
  - Make changes and optimize
  - Useful for system integrators, not only for core application developers

Being able to visualize the execution of your system is very powerful, even if you're not really looking for issues.



# Conclusion

- Tracing can help understand the execution of an application
  - Even if you're not looking for performance issues!
- Collect trace data from multiple sources and analyze the combined data
- ROS 2 has built-in tracing instrumentation
- Eclipse Trace Compass can display ROS 2 trace data
  - Could use a bit more love, though!
- Add custom application-specific instrumentation
- Implement your own Trace Compass plugin



# Thank you!

Apex.AI

[christophe.bedard@apex.ai](mailto:christophe.bedard@apex.ai)

## Links

- [github.com/ros2/ros2\\_tracing](https://github.com/ros2/ros2_tracing)
- [tracecompass.org](https://tracecompass.org)
- [github.com/christophebedard/ros2-message-flow-analysis](https://github.com/christophebedard/ros2-message-flow-analysis)
- [github.com/christophebedard](https://github.com/christophebedard)

## Relevant papers

- C. Bédard, I. Lütkebohle, and M. Dagenais, “**ros2\_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2,**” *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.
- C. Bédard, P.-Y. Lajoie, G. Beltrame, and M. Dagenais, “**Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems,**” *Robotics and Autonomous Systems*, vol. 161, p. 104361, 2023.