KubeCon CloudNativeCon Japan 2025

Debugging OpenTelemetry:

Ensuring Your Observability Signals Are Spot On

Kasper Borg Nissen, Developer Advocate at dash0

@phennex kaspernissen.xyz

dash0

Who?

Developer Advocate at Dash0

KubeCon+CloudNativeCon EU/NA 24/25 Co-Chair

CNCF Ambassador

Golden Kubestronaut

CNCG Aarhus, KCD Denmark Organizer

Co-founder & Community Lead Cloud Native Nordics

dash0

tl;dr

Part 1: What is OpenTelemetry?	Part 2: Common Pitfalls	Part 3: Language Specific Challenges	Part 4: Best Practices & Tools
OpenTelemetry			
A standardized way to collect telemetry - vendor-neutral and open.	Most issues stem from misconfigurations that fail silently	Each SDK has quirks - setup and defaults vary more than you'd expect.	Route everything through the Collector and test locally with console exporters.

Part 1:

What is OpenTelemetry ?

OpenTelemetry in a nutshell

What it is

2nd largest CNCF project by contributor count

A set of various things focused on letting you collect telemetry about systems:

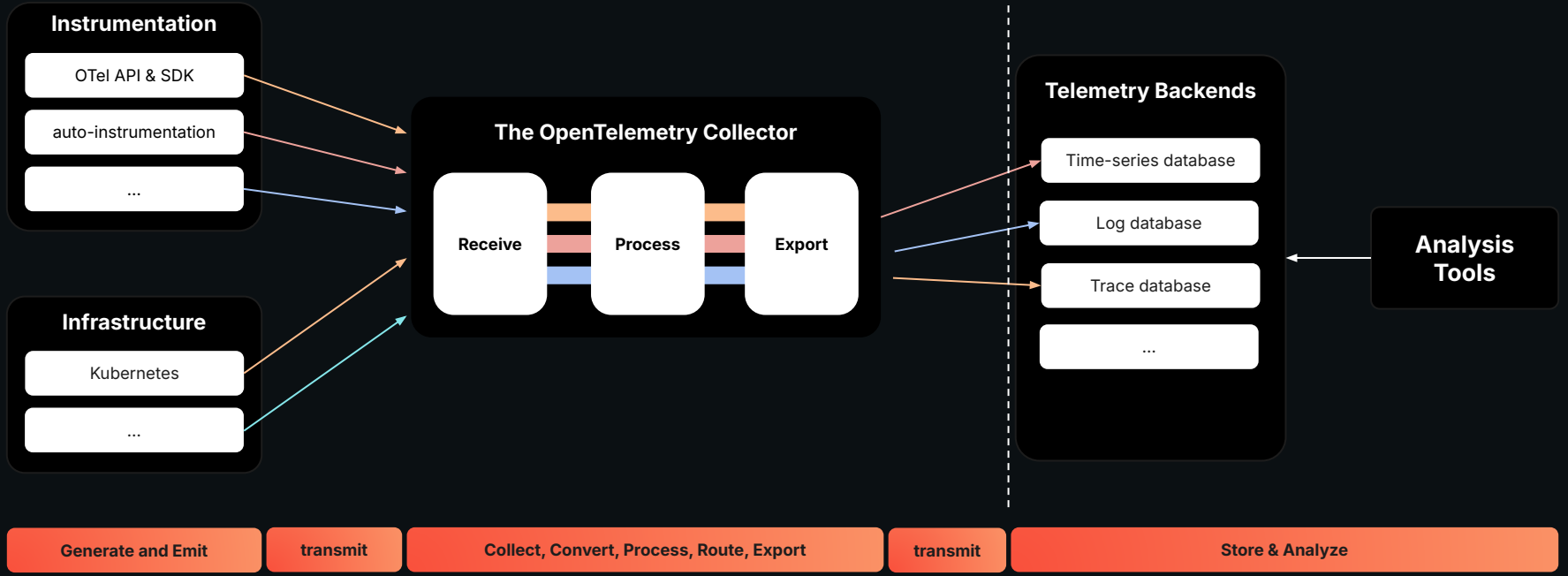
- Data models
- API specifications
- Semantic conventions
- Library implementations in many languages
- Utilities
- and much more

OpenTelemetry in a nutshell

What it is NOT

- Proprietary
- An all-in-one observability tool
- A data storage or dashboarding solution
- A query language
- A Performance Optimizer
- Feature complete

OpenTelemetry: A 1000 miles view



OpenTelemetry: A 1000 miles view

Collection of Telemetry is standardized

OpenTelemetry

"The last observability agent you will ever install"

Vendor space

dash0

... and many more.

Generate and Emit	transmit	Collect, Convert, Process, Route, Export	transmit	Store & Analyze

Why OpenTelemetry?

Instrument once, use everywhere

Separate telemetry generation from analysis

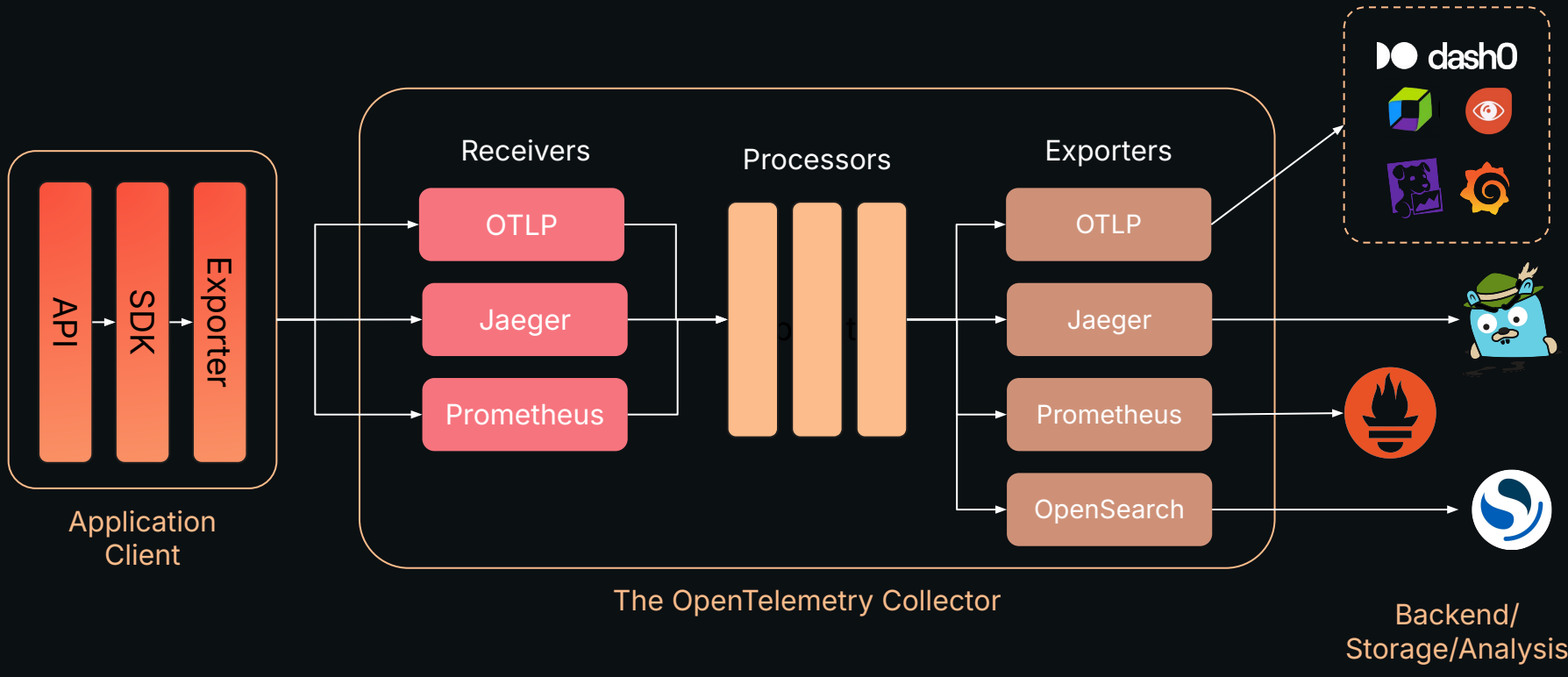
Make software observable by default

Improve how we use telemetry

Why Debugging OpenTelemetry Matters

OpenTelemetry is powerful	→	… but complex
Easy to misconfigure	→	… hard to diagnose
Silent failures	→	… equals missing signals
Debugging skills	→	… equals reliable observability

An OpenTelemetry Pipeline



An OpenTelemetry Pipeline

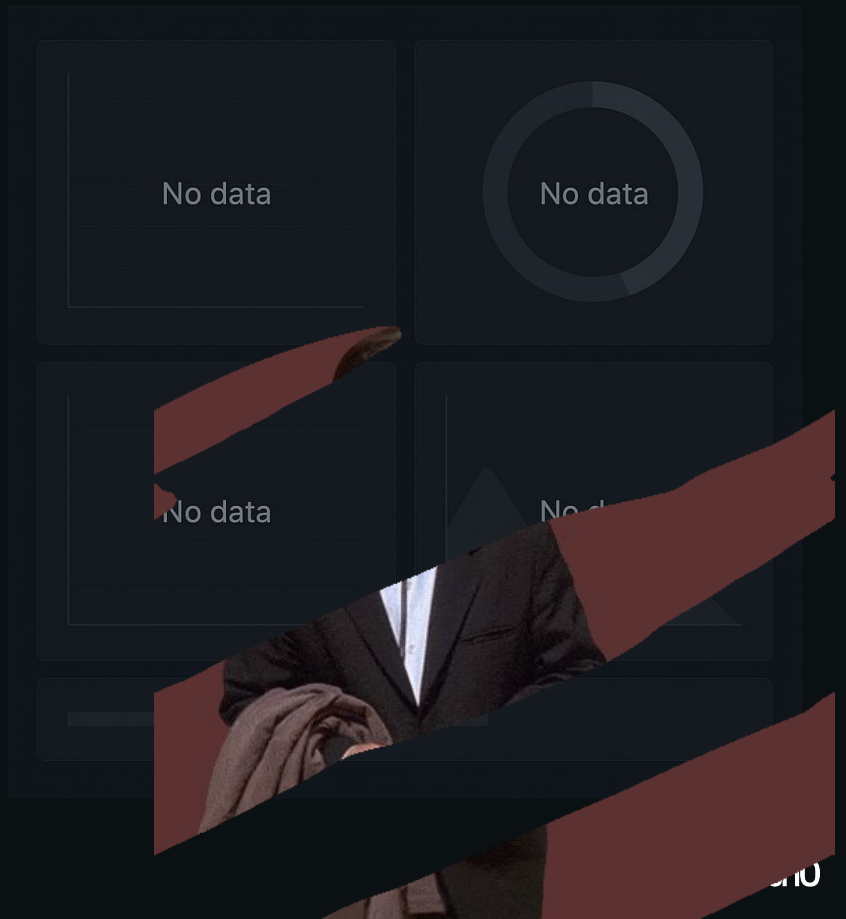
```
service:
   pipelines:
     logs:
       receivers: [otlp,filelog]
       processors: [k8sattributes,resourcedetection,batch,resource]
       exporters: [otlp/dash0]
     metrics:
       receivers: [otlp,kubeletstats]
       processors: [k8sattributes,resourcedetection,batch,resource]
       exporters: [prometheus,otlp/dash0]
     traces:
       receivers: [otlp]
       processors: [k8sattributes,resourcedetection,batch,resource]
       exporters: [jaeger,otlp/dash0]
```

Part 2:

Common Pitfalls

Most common mistakes?

- Wrong protocol or port
- Missing service name
- Span context not propagated
- Spans started but never ended
- Semantic conventions mismatched
- SDK initialized too late
- Exporter not flushed on shutdown



Pitfall 1: Incorrect Export Configuration

- Protocol mismatch (gRPC vs HTTP)
- Wrong port (4317 vs 4318)
- Missing OTEL_EXPORTER_OTLP_PROTOCOL

Pitfall 2: Missing or Incorrect Service Name

- Required for trace correlation
- Shows as “`unknown_service`” in backends
- Must be set via env or resource attribute

Pitfall 3: Context Propagation Issues

- New span started outside parent context
- Go: forgot to pass context
- Leads to orphan spans or broken traces

Span Context Object

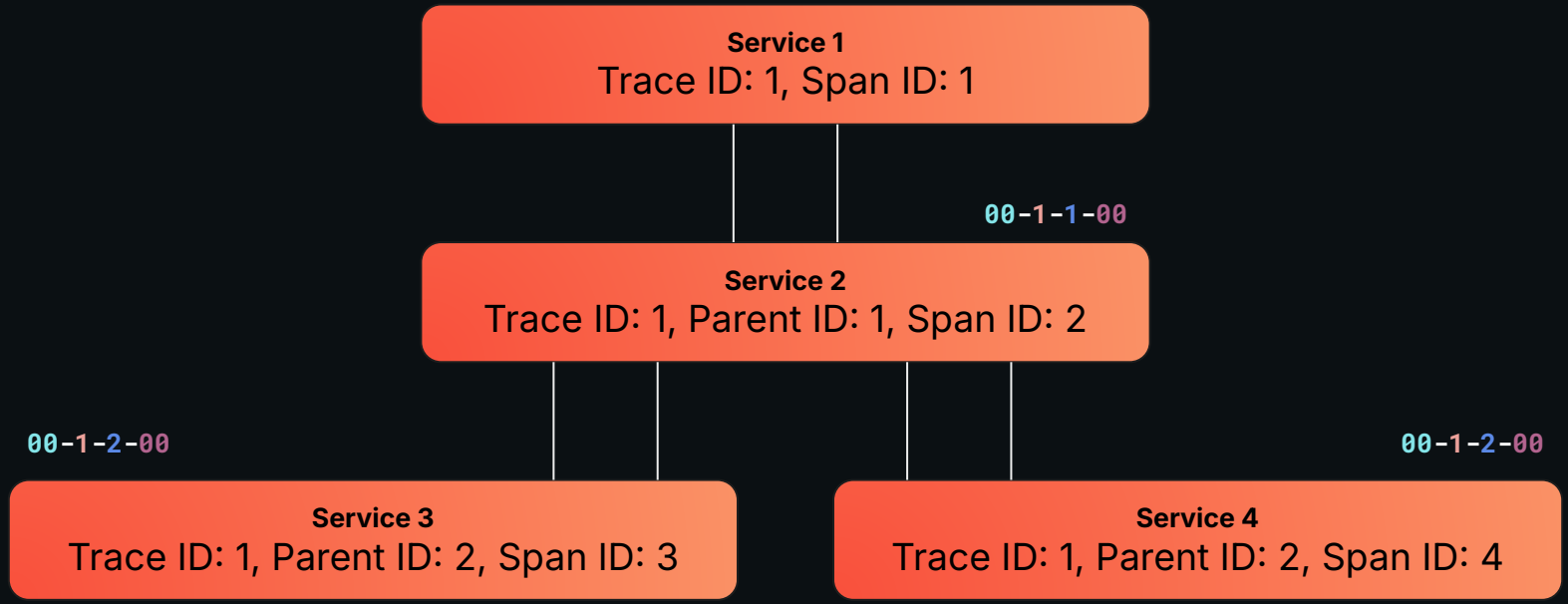
```
Trace ID      :  aeba5efdddf0c01648dad40186c7fbf8
Span ID       :  6b45h68de9ab26a2
Parent ID     :  0a99c68359ade48a
Name          :  POST /purchase/{order-number}
Kind          :  Server
Start time    :  2025-08-15 06:25:28.1624527 +0000 UTC
End time      :  2025-08-15 06:25:28.7430470 +0000 UTC
Status code   :  Unset
```

Kinds:
- Server
- Client
- Consumer
- Producer
- Internal

Only assign `Error` status code otherwise leave `Unset`

`Unset = OK`

Trace & Span Relationship



W3 Trace Context: <version>-<trace-id>-<span-id>-<trace-flags>

Pitfall 4: Initialization & Shutdown Issues

- Tracer initialized after app starts
- Exporter flushed too early or never
- Java: Agent not attached
- Go: No tracer provider set = no spans

Pitfall 5: Semantic Convention Mismatches

- Wrong attribute names = no enrichment
- Example: `http.method` vs `http.request.method`
- Standardized conventions matter

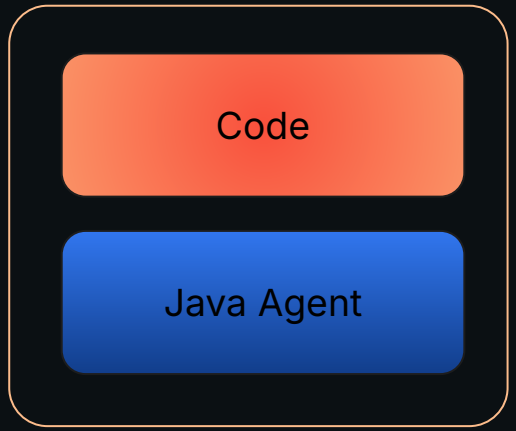
Part 3:

Language Specific Challenges

Java

Auto-instrumentation with the Java Agent.

1. Java agent setup via JVM flag: `-javaagent`
2. Bytecode manipulation via `java.lang.instrument.Instrumentation API` to modify the bytecode of classes at load time
3. Auto-detect and replies relevant instrumentation modules from frameworks/libraries
4. Context Propagation via OpenTelemetry's Context Propagation APIs (injects/extracts context from e.g. HTTP headers (`traceparent`)
5. Instrumentation behavior is configured using environment variables or a otel.properties file
6. Exporting data via the configured exporters (defaults to `OTLP http/protobuf` to `localhost:4318`)



Demo

Tea Ceremony Tracer ☕

A small Spring Boot web application that serves a `/tea` endpoint returning a random Japanese tea type and temperature.

@phennex kaspernissen.xyz

dash0

Node.js

Auto-instrumentation in node.js

1. Uses `--require` to load instrumentation before app start
2. Patches core and third-party modules
   a. Scans `node_modules`
   b. Detects installed libraries (e.g. `express`, `http`, `mysql`, etc)
   c. Dynamically loads instrumentation plugins from `@opentelemetry/instrumentation-*`
   d. Monkey-patches libraries
3. Creates and manage spans automatically (Incoming/outgoing HTTP requests)
4. Spans are batched and exported via SDK

```
node --require @opentelemetry/auto-instrumentations-node/register app.js
```

or

```
export NODE_OPTIONS="--require @opentelemetry/auto-instrumentations-node/register"
```

Demo

Ramen Ratings API 

A lightweight Express application that serves a `/ramen` endpoint returning a random ramen type and a rating between 0–5.

@phennex kaspernissen.xyz

dash0

Go

Manual Instrumentation

1. Import `otel`, `sdk/trace`, `otlptracegrpc`, etc
2. Create and set a `TraceProvider`
3. Use `tracer.Start(ctx, "name")` + `defer span.End()`
4. Setup and flush your exporter (`otlp`, `stdout`,etc.)

Demo

Sakura Stats Service 🌸

A minimal HTTP service in Go that exposes a `/sakura` endpoint returning fictional cherry blossom bloom data.

@phennex kaspernissen.xyz

dash0

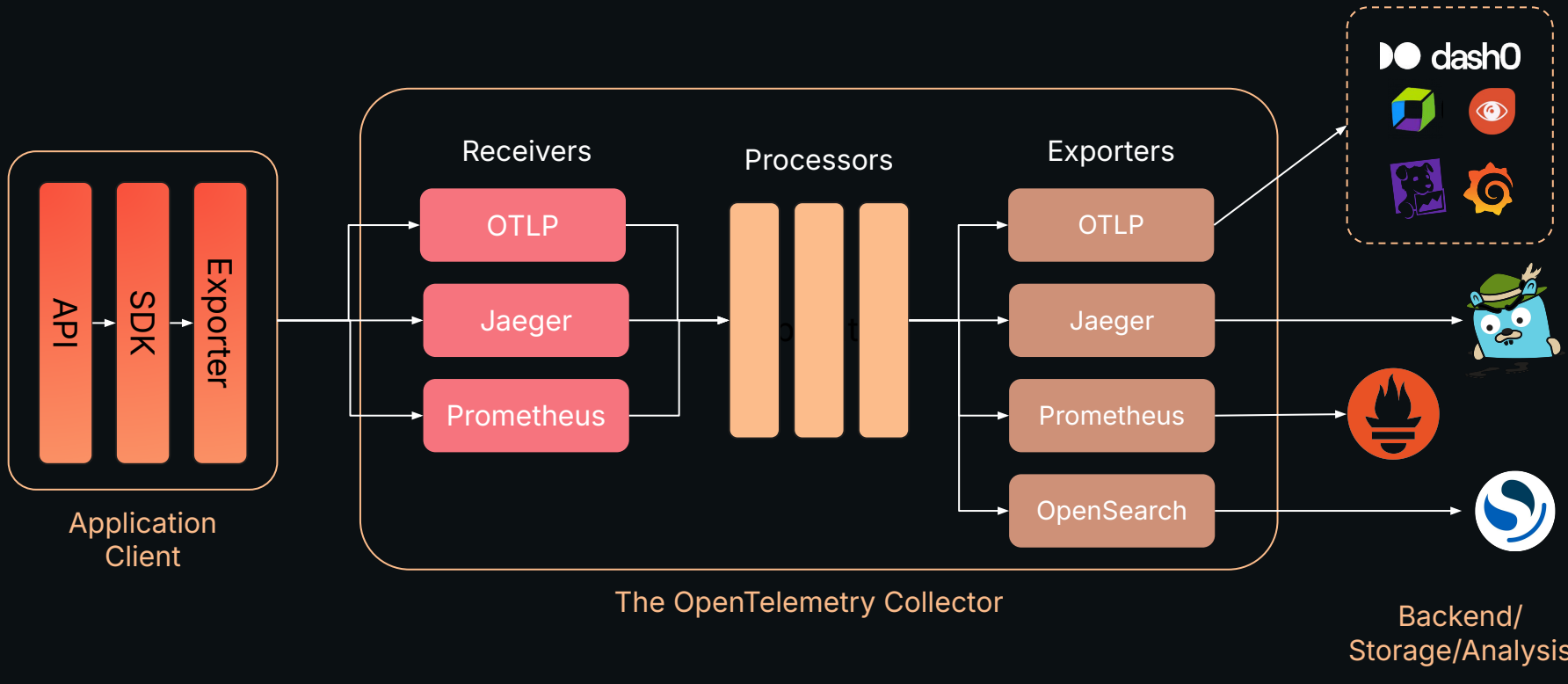
Comparison

Feature/Behaviour	Java	Node.js	Go
Instrumentation	Auto via `-javaagent`	Auto via `--require`	Manual
Default OTLP protocol	`http/protobuf`	`http/protobuf`	`grpc`
Default OTLP port	4318	4318	4317
Local Exporter `OTEL_TRACES_EXPORTER`	`console`	`console`	`stdout`
Debug logger	`OTEL_LOG_LEVEL=debug` and `OTEL_JAVAAGENT_DEBUG=true`	`OTEL_LOG_LEVEL=debug`	`OTEL_LOG_LEVEL=debug`
Context propagation	Handled by agent	Handled by SDK	Explicit context propagation

Part 4:

Best Practices & Tools

The Collector as a Central Routing Layer



Checklist

- ✅ Match protocol + port
- ✅ Set service name
- ✅ Initialize SDK early
- ✅ End all spans
- ✅ Use debug + console exporters
- ✅ Watch semantic conventions
- ✅ Always test with a local collector

Tools & Tips

```
OTEL_SERVICE_NAME=”your service”
OTEL_RESOURCE_ATTRIBUTES="service.namespace=dash0,service.version=1.0,service.instance.id=$(uuidgen)"

OTEL_LOG_LEVEL=debug

OTEL_TRACES_EXPORTER=stdout / console
OTEL_METRICS_EXPORTER=stdout / console
OTEL_LOGS_EXPORTER=stdout / console

OTEL_JAVAAGENT_DEBUG=true
```

```
docker run -p 4317:4317 -p 4318:4318 --rm otel/opentelemetry-collector --config=/etc|/otelcol/config.yaml --config="yaml:exporters::debug::verbosity: detailed"
```

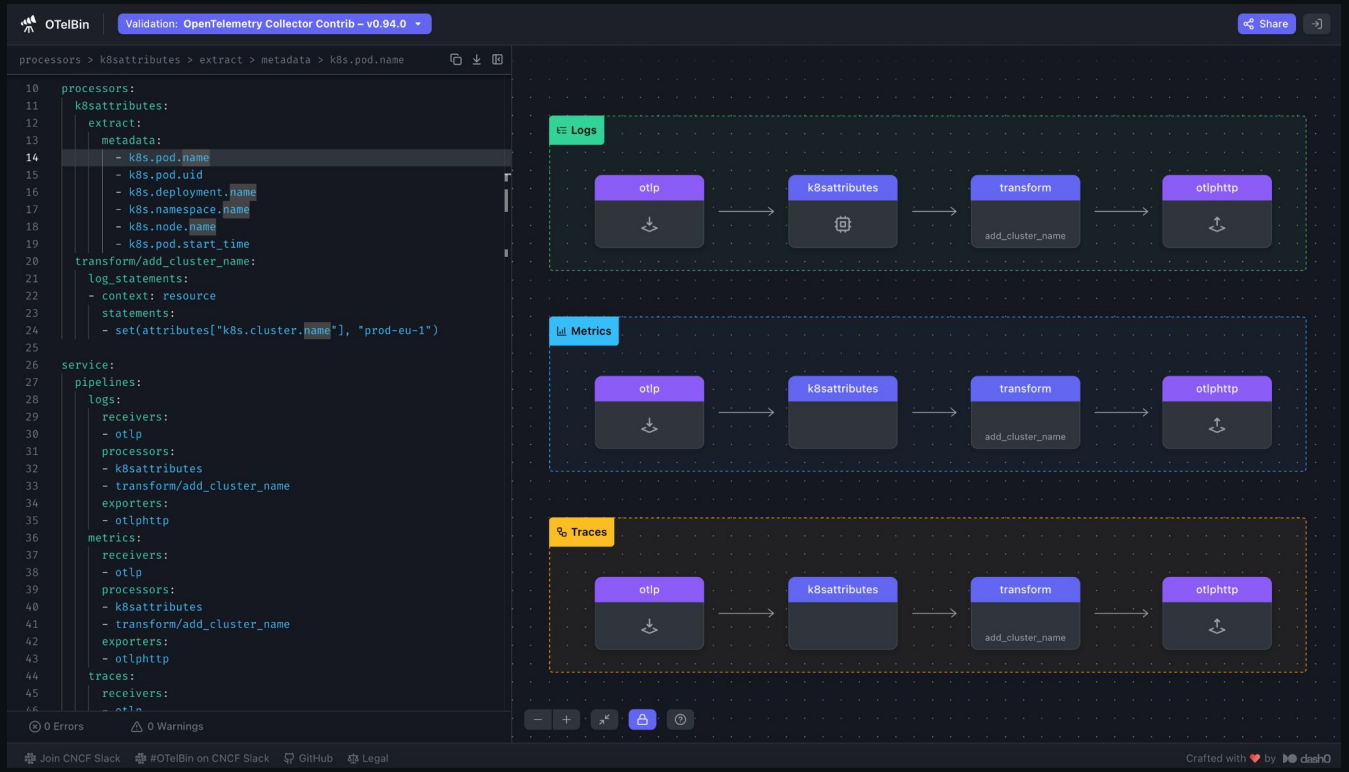
Tools - Otelbin.io

Forever free, OSS

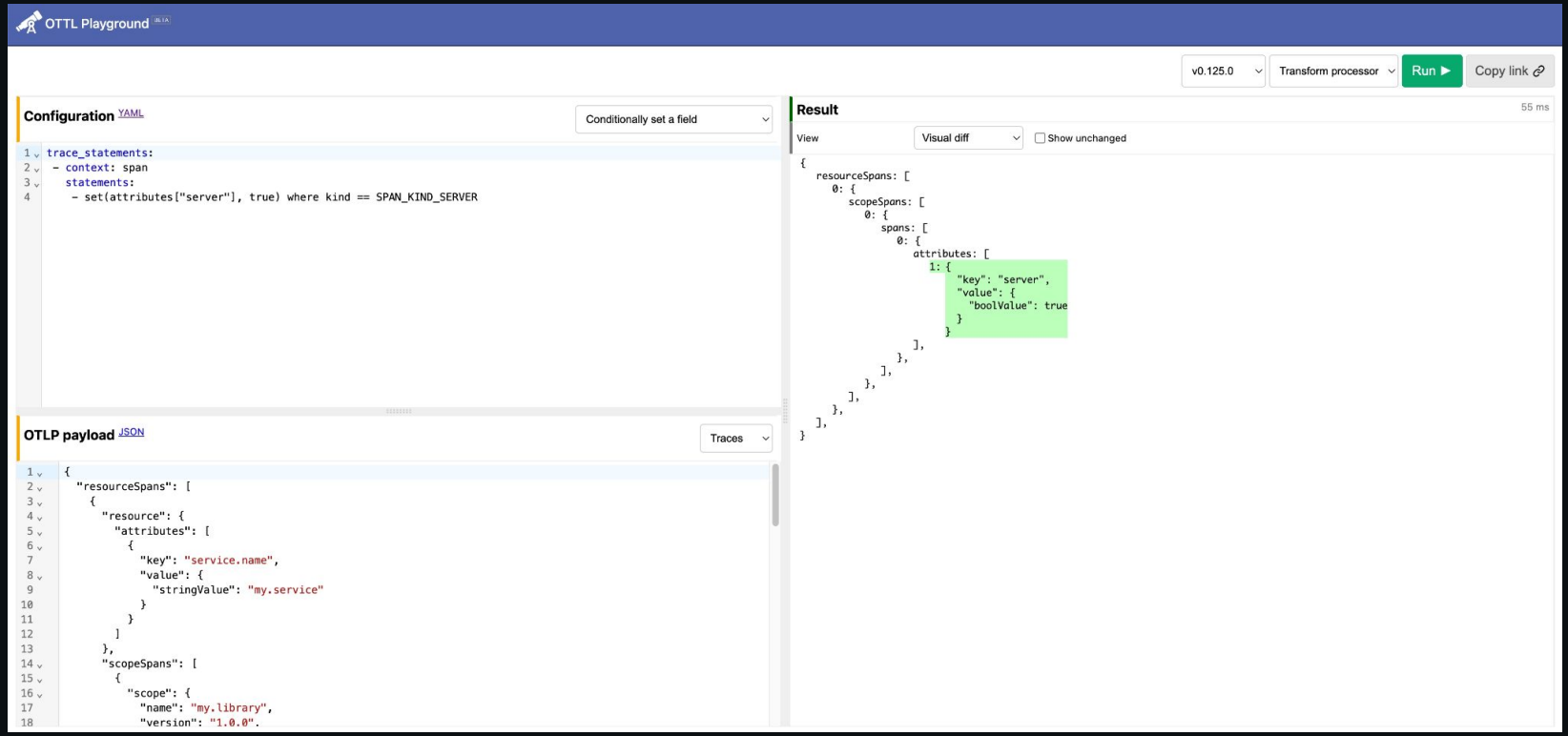
Editing, visualization and validation of OpenTelemetry Collector configurations

With ❤ by Dash0!

https://www.otelbin.io/



Tools - ottl.run



But, it's not just about getting the pipeline working...
...it's about emitting good telemetry

Good and bad telemetry

✅ Good Telemetry Is:

- **Structured**: Uses consistent, semantic field names
- **Contextual**: Includes service, region, version, deployment ID
- **Correlated**: Connects across traces, metrics, and logs
- **Trustworthy**: Accurate timestamps, proper status codes
- **Useful**: Helps you answer real questions about the system

⛔ Bad Telemetry Is:

- Noisy, redundant, or inconsistent
- Missing key context (e.g. no trace ID in logs)
- Misleading (e.g. incorrect span names or tags)
- Unusable in queries or alerts

“Telemetry without context is just data”

Michelle Mancioppi, Head of Product, Dash0

@phennex kaspernissen.xyz

dash0

Key Takeaways

- Always verify your telemetry
- Remember the usual suspects
- Context: know your language
- Embrace the Collector
- Use the available tools and don’t shy away from logs

KubeCon CloudNativeCon Japan 2025

Thank you!

Get in touch!
Stop by our booth and chat!

Kasper Borg Nissen, Developer Advocate at dash0

@phennex kaspernissen.xyz

dash0

Abstract

OpenTelemetry has become the go-to framework for unifying observability signals across metrics, logs, and traces. However, implementing OpenTelemetry often comes with its own set of challenges: broken instrumentation, missing signals, and misaligned semantic conventions that undermine its effectiveness. Debugging these issues can be daunting, leaving teams stuck with incomplete or unreliable observability data.

In this session, Kasper will demystify the debugging process for OpenTelemetry. Attendees will learn how to identify and troubleshoot common issues, ensure signals are transferred correctly, and align instrumentation with semantic conventions for consistent insights. Through live demos, Kasper will showcase techniques for validating resource configurations, debugging signal pipelines, and building confidence in your observability setup. This session is designed for anyone looking to unlock the full potential of OpenTelemetry and create robust observability practices.