eBook

# Service Mesh for Developers, Part 1: Exploring the Power of Observability and OpenTelemetry

solo.io

## Table of Contents

solo.io

## Introduction

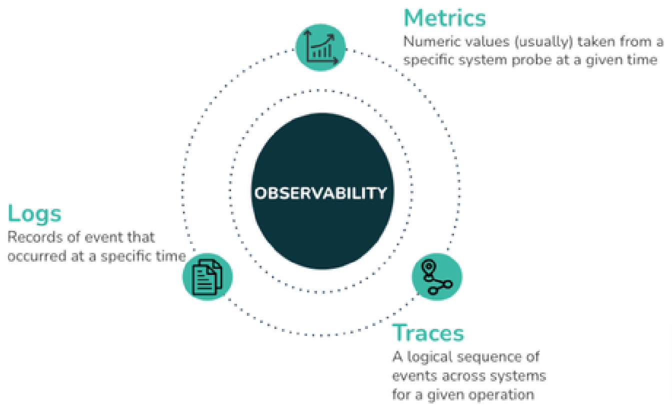
In today's complex application landscapes, observability is crucial for debugging intricate systems. With service mesh architectures, developers have powerful tools to enhance observability and streamline debugging. In this eBook, we embark on a journey to explore how observability within a service mesh improves application debugging.

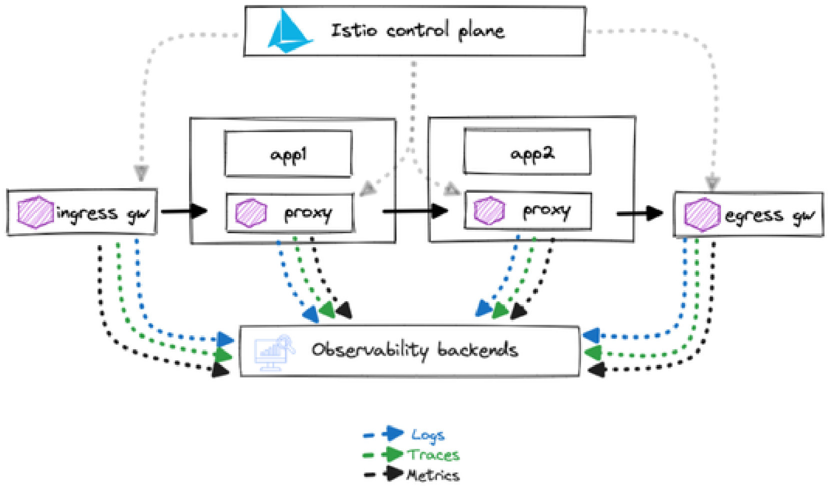
In the first eBook of the series, we explore the benefits and techniques of using observability within a service mesh for effective debugging. We explain how observability helps to understand complex systems and how service mesh provides features like distributed tracing, metrics, and logging for valuable insights into application behavior.

Throughout the series, we cover various aspects of observability, including testing in production and live debugging. We'll leverage observability tools, such as OpenTelemetry, within a service mesh to uncover real-time insights, validate application behavior, and promptly resolve issues, ensuring reliable and high-performing applications.

## Understanding Observability and Service Mesh

Observability helps us understand and debug complex systems by providing insights into application behavior. It allows us to identify and resolve issues effectively, ensuring system reliability and performance. With observability, we gain visibility into request flows, errors, and system performance that can enable us to improve our applications.


Metrics
Numeric values (usually) taken from a specific system probe at a given time
OBSERVABILITY
Logs
Records of event that occurred at a specific time
Traces
A logical sequence of events across systems for a given operation



Service mesh architectures, such as Istio, enhance observability by offering features that include:

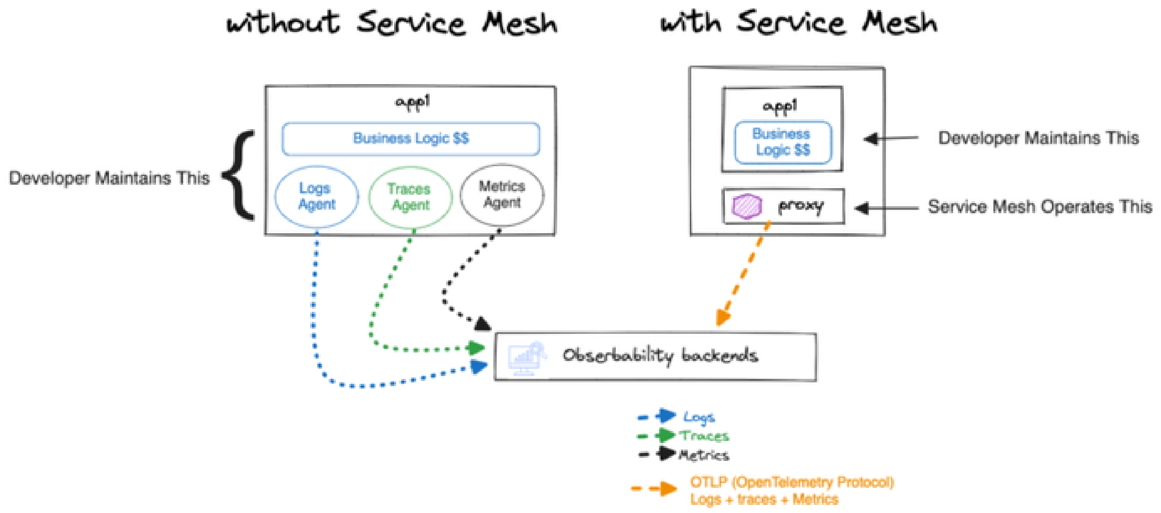
- **Distributed tracing** tracks request journeys, identifying bottlenecks and performance issues.
- **Metrics** provide quantitative data on response times, errors, and resource utilization.
- **Logging** captures events and messages, aiding issue tracking and troubleshooting.

In essence, observability helps us understand and debug complex systems, while service mesh architectures provide integration of different observability tools. By leveraging distributed tracing, metrics, and logging, developers gain valuable insights to troubleshoot effectively and ensure optimal system performance.

# What is OpenTelemetry

OpenTelemetry (OTel) helps developers achieve observability in their applications by collecting crucial data for understanding and debugging. Instrumenting applications with OpenTelemetry captures telemetry data on requests, errors, and performance, facilitating efficient problem-solving.

The value of OpenTelemetry is enhanced by its compatibility with different service mesh implementations. The service mesh manages communication between application components, and OpenTelemetry seamlessly integrates with them. This flexibility allows developers to choose monitoring tools without impact on the application.

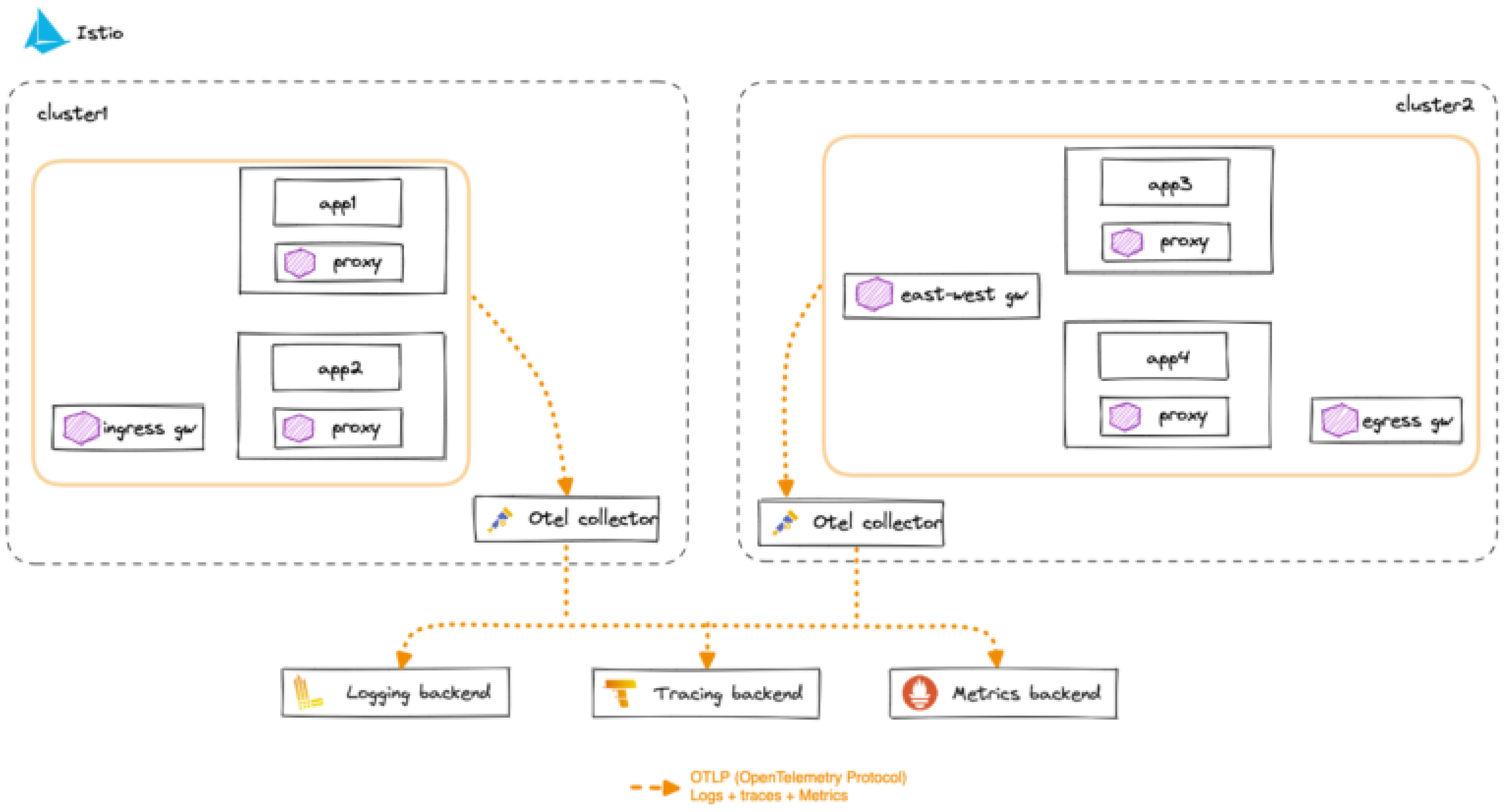


OTel is based on the three pillars of Observability:

- **Logs** capture textual records of events
- **Traces** provide a distributed view of request flows
- **Metrics** quantify performance and behavior

## OpenTelemetry Collector

OpenTelemetry is an open-source observability framework that provides APIs and SDKs for instrumenting applications. The OpenTelemetry Collector is a separate component that collects, processes, and exports telemetry data from various sources, acting as a flexible intermediary with customization capabilities.



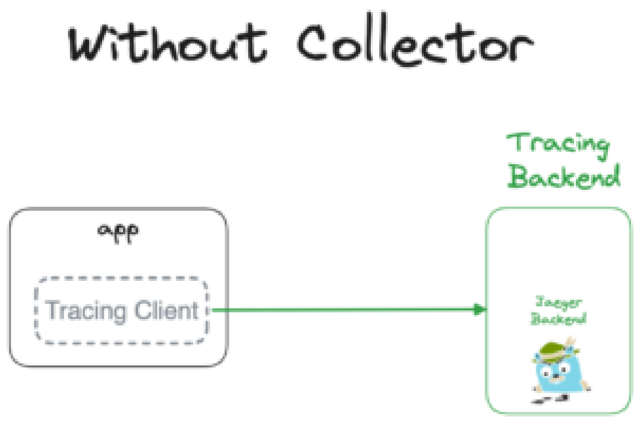
## Distributed Tracing with OpenTelemetry Collector

OpenTelemetry facilitates distributed tracing in the service mesh. It helps track requests as they move through different parts of our system and applications, providing insights into their flow and behavior.

To enable trace stitching, tracing tools require the passage of information through headers. As a result, developers must design their applications to appropriately propagate the relevant headers.

In OTel, these are:

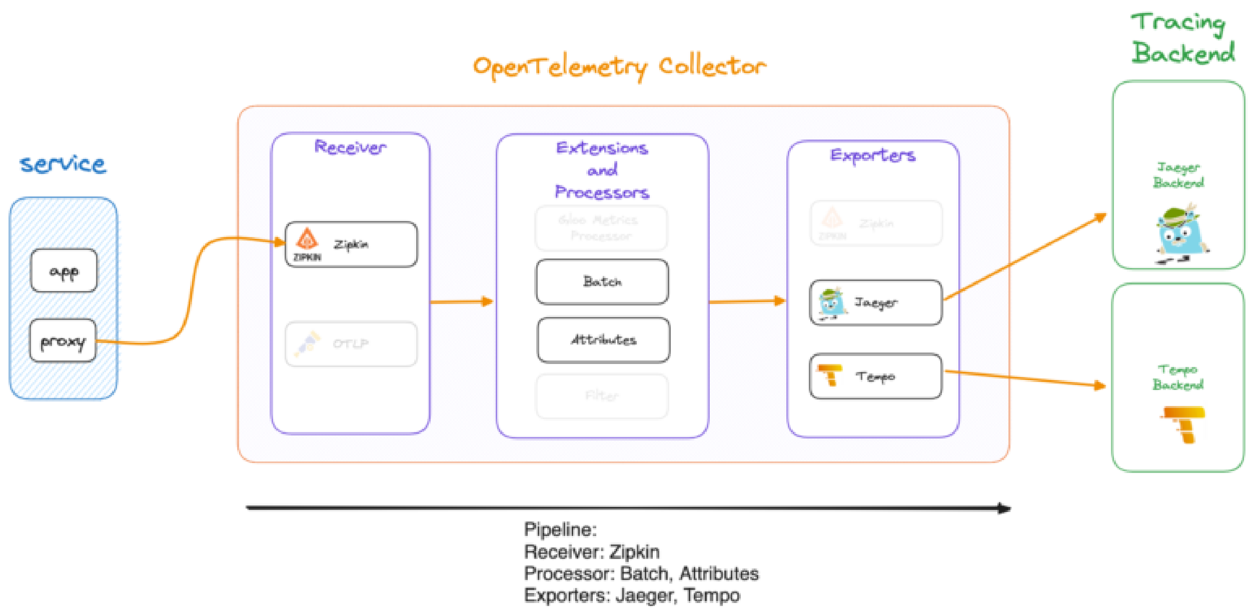
- **Trace Context Headers**: The most essential headers for trace propagation are *traceparent* and *tracestate*. *traceparent* contains the trace ID, span ID, and trace flags, while *tracestate* includes additional contextual information.
- **Correlation Headers**: Headers like correlation-id or x-correlation-id help correlate related requests across different services or components.
- **Baggage Headers**: Baggage headers like baggage-{key} are used to propagate custom contextual information throughout the trace.



In the depicted image, the Tracing Client establishes a direct connection with the Tracing Backend. However, in complex topologies, such direct connections may not be feasible, especially in distributed systems.

## OpenTelemetry Collector Components

To enable distributed tracing, OpenTelemetry utilizes the OpenTelemetry Collector, which has components like receivers, processors, and exporters. Receivers collect tracing data from various sources, processors enhance and manipulate the data, and exporters send it to external systems or visualization tools.



For example, in Istio, data emitted from the sidecar proxies is captured by the OpenTelemetry Collector, allowing us to visualize the request flow. By integrating with tools like Jaeger or Zipkin, we can gain insights into request journeys, identify performance issues, and resolve errors.

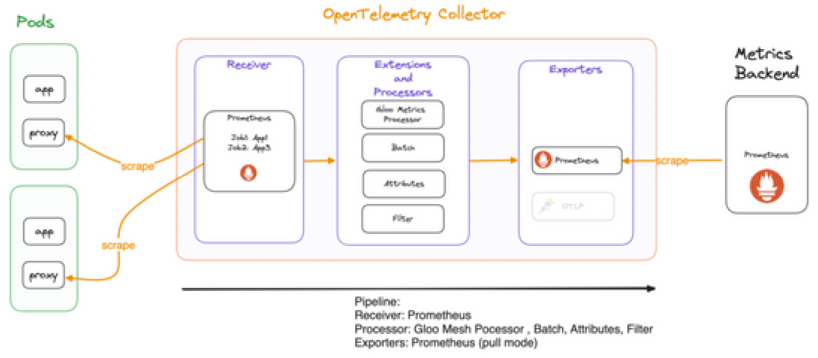
## Metrics Collection with OpenTelemetry

OpenTelemetry Collector collects and exports metrics from applications within a service mesh, providing valuable insights into application performance.

Metrics present a different complexity compared to tracing. Unlike tracing, where information is pushed, the model for metrics involves pulling data. Consequently, a client like Prometheus is required alongside the application. To prevent public exposure of metric endpoints, the client is deployed next to the applications and directly targets the container.

The OpenTelemetry Collector separates the application's deployment location from Prometheus by employing a client that carries out similar tasks to Prometheus.

Metrics exporters in OpenTelemetry Collector send collected metrics to monitoring systems like Prometheus. These systems visualize and analyze the metrics, offering information on application health and performance.



For example, when instrumenting service mesh with OpenTelemetry Collector, we collect metrics for analysis. OpenTelemetry Collector integrates with a service mesh, scraping metrics that are made available from the sidecar proxies next to your services. We export these metrics to Prometheus or Grafana, visualizing response times, error rates, and resource utilization. This helps identify bottlenecks and troubleshoot issues.

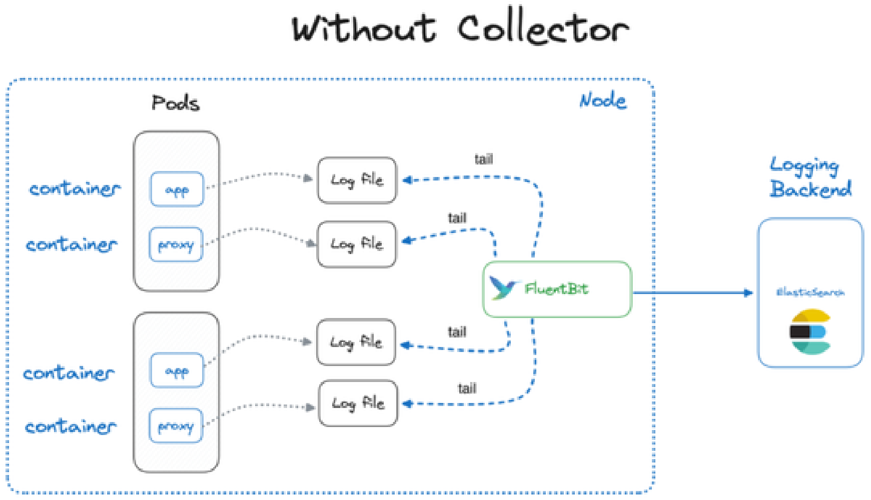
Using Istio as the service mesh empowers operators to enhance the monitoring experience by providing the ability to manipulate the metrics exposed by Envoy, enabling greater control over the data received by backends.

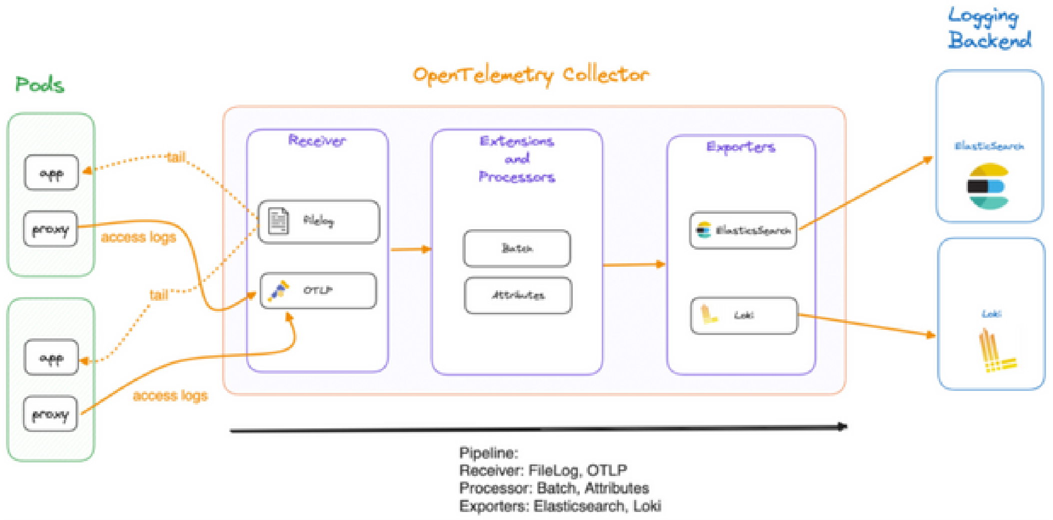
Later in the eBook, we will explore how Grafana can assist with visualization.

# Logging and OpenTelemetry

The OpenTelemetry Collector aids in debugging by integrating with logging frameworks to capture detailed application logs.

As you can see in the picture, without an OTel Collector, you need an agent (i.e. fluentbit) to collect the application logs. That means adding another tool to the stack which then, needs to be maintained.



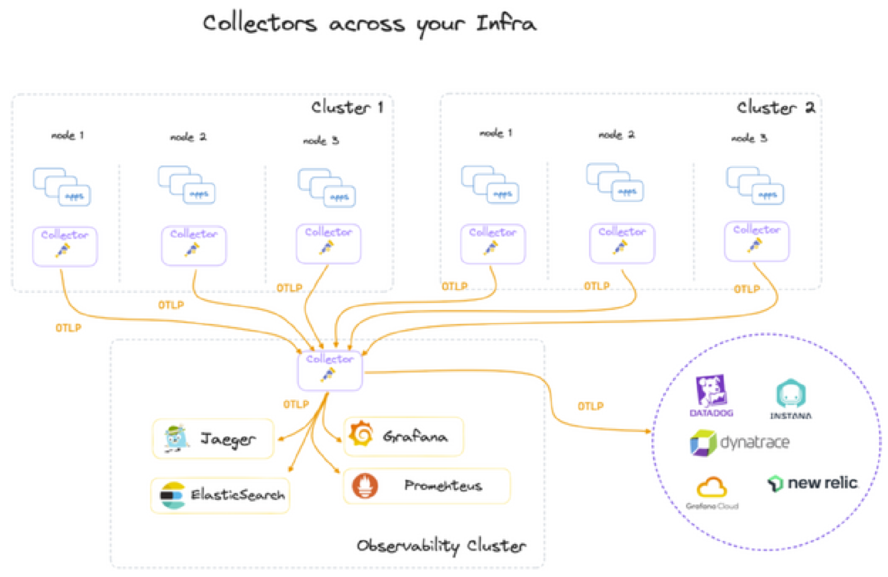


To implement logging with OpenTelemetry Collector, configure it to collect and forward logs to log analysis tools like ELK Stack, Splunk or Loki.

## Transport Telemetry Across Your Complex Topology

Transporting telemetry across a complex topology becomes seamless with the OpenTelemetry Collector. By leveraging this powerful tool, developers can effortlessly combine multiple collectors to gather and transmit telemetry data from various sources. With the OpenTelemetry Collector, the complexities of transporting telemetry across diverse components are efficiently managed, enabling comprehensive observability and streamlined monitoring in even the most intricate environments.

As shown in the picture below, you have the flexibility to configure multiple collectors that transmit Telemetry signals to other collectors across different clusters using the standard OTel protocol, OTLP. This capability centralizes the operational control of communication within a single component, the collector, instead of relying on individual proxies when using a service mesh.



Many popular providers such as Grafana Cloud, Dynatrace, Datadog, New Relic, Instana, and others actively support and facilitate the integration of OpenTelemetry (OTel) by embracing the OpenTelemetry Protocol (OTLP). This allows for easier and more seamless integration of OTel with these providers' services and tools.

## 8 Best Practices for Effective Debugging with OpenTelemetry

1. **Comprehensive instrumentation**: Ensure thorough instrumentation of your applications using OpenTelemetry to capture relevant telemetry data. In the case of using a service mesh (like Gloo Mesh), the platform does this for you.

2. **Trace context propagation**: Utilize OpenTelemetry's context propagation mechanism to maintain trace context across distributed components of your application. This allows you to follow the path of a request through different services and identify potential bottlenecks or issues. In the case of service mesh such as Istio and platforms built on top of them (like Gloo Mesh), this task is reduced to just propagating headers.

3. **Granular trace sampling**: Configure trace sampling rates appropriately to balance the volume of collected traces with the overhead of capturing and processing them. Adjust the sampling rate based on the importance and performance impact of specific operations or services.

4. **Log correlation**: Correlate logs with traces by including trace identifiers (such as trace IDs and span IDs) in log entries. This correlation helps in linking log events with specific trace spans, enabling easier troubleshooting and understanding of the request flow.

## 8 Best Practices for Effective Debugging with OpenTelemetry (cont.)

5. **Error handling and logging**: Implement robust error handling mechanisms and log errors and exceptions with relevant context. Use structured logging formats and include essential details such as error codes, timestamps, and relevant request information. This helps in pinpointing the source of errors during debugging.

6. **Custom attributes and metadata**: Leverage OpenTelemetry's ability to add custom attributes and metadata to captured telemetry data. Include additional contextual information, such as user IDs, session IDs, or specific request parameters, to enhance the visibility and understanding of application behaviour during debugging. As well as removing any sensitive data like passwords or keys that should not be exposed.

7. **Visualization and analysis tools**: Utilize visualization and analysis tools compatible with OpenTelemetry, such as observability platforms or logging solutions like ELK Stack or Grafana. These tools provide a rich set of features for visualizing and analyzing telemetry data, making it easier to spot anomalies, detect patterns, and identify performance issues.

8. **Collaboration and knowledge sharing**: Foster collaboration among developers and teams by sharing telemetry data and insights captured by OpenTelemetry. Collaborative debugging sessions, code reviews, and post-mortem analyses can help in identifying and resolving complex issues more efficiently.

## Conclusion

OpenTelemetry offers significant benefits for observability and debugging within a service mesh. It provides a standardized approach to capture metrics, traces, and logs, enabling developers to gain deep insights into their applications' behavior. By incorporating OpenTelemetry into the development process, developers can proactively monitor and debug their services, ensuring optimal performance.

OpenTelemetry provides a holistic view of the system's behavior and it gives us visibility into complex architecture. This allows developers to pinpoint performance bottlenecks, and optimize their applications.

In the next eBook of the series, we'll shift our focus to an intriguing aspect of application development: Testing in Production. In Debugging: Mastering Testing in Production within a service mesh, we will explore how observability, within the service mesh architecture, enables us to conduct thorough testing in live production environments. We will uncover the benefits of testing in production, discuss different strategies and techniques, and showcase practical examples of how observability empowers developers to validate application behavior and ensure reliability in real-world scenarios. Join us!