



Code & Coffee

OpenTelemetry

A Unified Approach to Observability

Michal Lukac, 14.2.2025

OpenTelemetry

- open-source observability framework that helps developers and DevOps teams get full visibility into their applications
- <https://opentelemetry.io/>



OpenTelemetry

What is Observability & Why Does it Matter?

"In modern cloud-native applications, traditional monitoring is no longer enough. With microservices, Kubernetes, and hybrid-cloud infrastructures, troubleshooting system failures is getting more difficult."

"Observability helps us answer the key questions: Why is my system slow? What's causing failures? Where is the bottleneck?"

"Without proper observability tools, debugging is like flying a plane without an instrument panel—you have no visibility into what's going wrong."

The Challenges Without OpenTelemetry

- 📌 Vendor Lock-in – Hard to switch between monitoring tools
- 📌 Multiple Instrumentation Methods – Different SDKs for each framework (Jaeger, Zipkin, Prometheus)
- 📌 High Operational Overhead – Debugging required manual data correlation from scattered sources.
- 📌 Modern applications are highly distributed (microservices, cloud-native apps).
- 📌 Traditional monitoring falls short – logs, metrics, and traces are siloed.

What is OpenTelemetry?

- 📌 open-source observability framework for collecting, processing, and exporting telemetry data
 - Logs
 - Metrics
 - Traces
- 📌 CNCF Project: Vendor-neutral & widely adopted across cloud and APM tools.
- 📌 Observability provides insights into system behavior by correlating telemetry data.
- 📌 OpenTelemetry helps standardize and streamline observability adoption.

Why OpenTelemetry?

 Standardization

 Flexibility

 Extensibility

 Open Source & Community-Driven

Core Components of OpenTelemetry

- 1 API (Application Programming Interface)
- 2 SDK (Software Development Kit)
- 3 Collector (Optional but Powerful!)
- 4 Exporters

How OpenTelemetry Works

(The Data Flow)

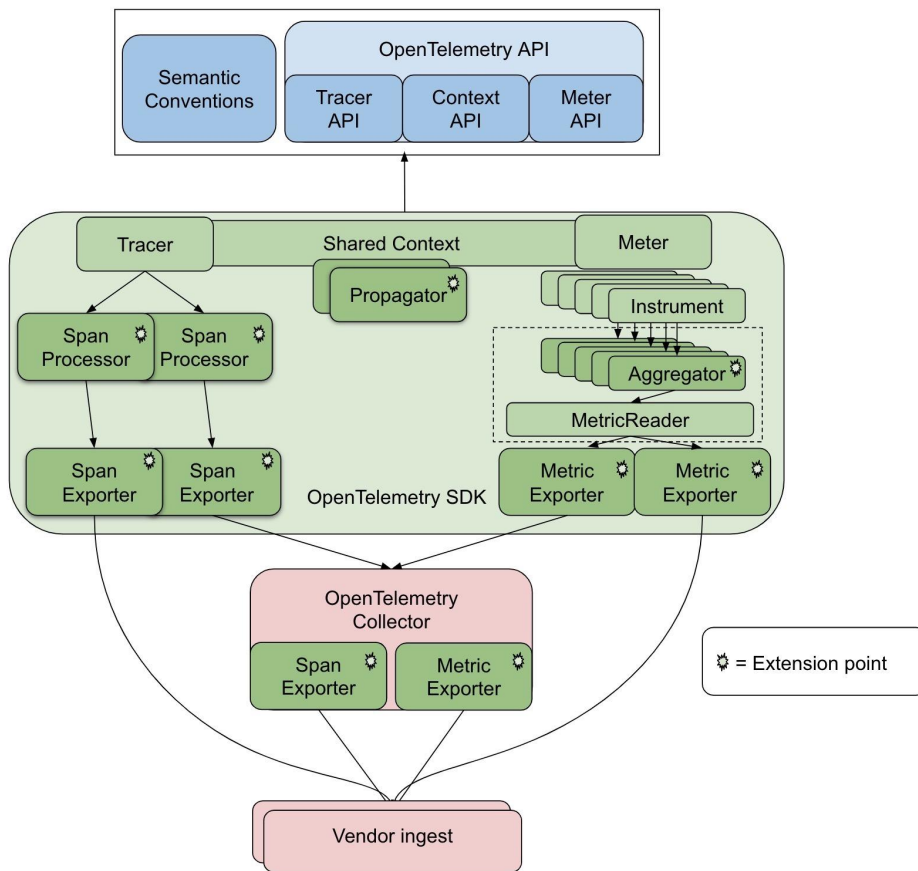
1 Instrumentation (Embedding OpenTelemetry in Your Application)

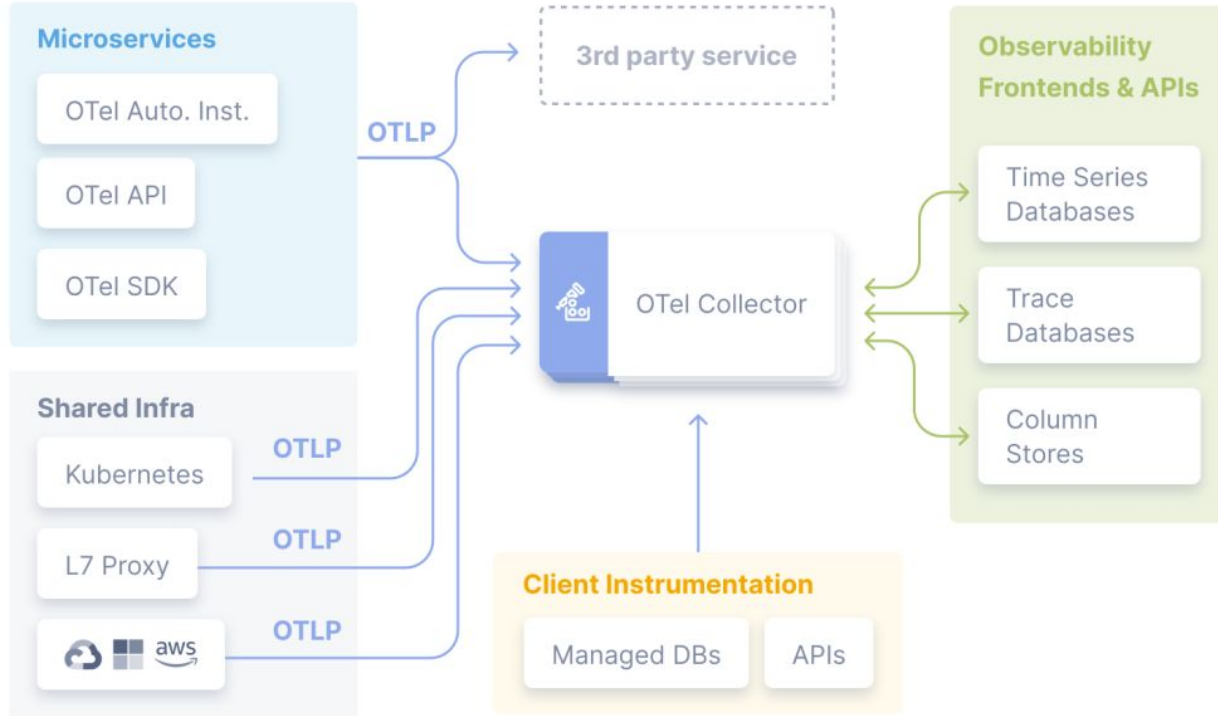
2 Exporting to the Collector

3 Processing & Aggregation

4 Sending Data to the Desired Observability Backend


🎯 Finally, exporters ship the telemetry data to backends like Datadog, Prometheus, or Jaeger, where it can be visualized and analyzed.





OpenTelemetry in Action

python

 Copy code

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import ConsoleSpanExporter, SimpleSpanProcessor

# Initialize Tracer
trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)

# Setup Exporter
span_processor = SimpleSpanProcessor(ConsoleSpanExporter())
trace.get_tracer_provider().add_span_processor(span_processor)

# Create a Span
with tracer.start_as_current_span("process_order"):
    print("Order is being processed...")
```

The Three Pillars - Traces, Metrics, Logs

- 1 Traces: Understanding Request Flow
- 2 Metrics: Measuring System Health
- 3 Logs: Capturing System Events

OpenTelemetry Request Flow

- 📌 Step 1: Instrumentation (Data Collection at Source)
- 📌 Step 2: Data Processing via OpenTelemetry Collector
- 📌 Step 3: Exporting Data to Backends
- 📌 Step 4: Observability Data is Stored & Analyzed

Why Use OpenTelemetry?

(Real-World Use Cases)

👁️ Common Use Cases:

- ✓ Distributed tracing for microservices visibility
- ✓ Performance optimization & latency troubleshooting
- ✓ Correlating logs, metrics, and traces for full-stack monitoring
- ✓ Monitoring serverless and cloud-native applications
- ✓ Enabling multi-cloud and hybrid cloud observability
- ✓ Cost optimization by reducing reliance on multiple vendor-specific tools

Use Case #1 – Distributed Tracing in Microservices

- User authentication (to verify customer identity)
- Payment processing
- Order dispatch to drivers
- Delivery tracking

Use Case #2 – Performance Optimization

How long API requests take to process

Which database queries are slow

Which services have the highest failure rates

- 1 Analyze tracing data to see where the slow request originates
- 2 Pinpoint the affected service (maybe a third-party payment API is slowing down)
- 3 Automatically trigger alerts when API latency exceeds a threshold"

Use Case #3 – Troubleshooting & Root Cause Analysis

- ✓ With OpenTelemetry, engineers can:
 - ✓ Instantly visualize the failing requests with tracing data
 - ✓ Analyze historical metrics to detect patterns in crashes
 - ✓ Use logs to pinpoint exact error messages & affected transactions
- ✓ Reduces Mean Time to Resolution (MTTR) by 70% or more by eliminating guessing and manual log correlation.

Use Case #4 – Observability in Serverless & Cloud-Native Apps

✓ OpenTelemetry Helps By:

- ✓ Tracing short-lived serverless functions, mapping dependent cloud resources
- ✓ Collecting custom serverless metrics (execution time, cold start delays, errors)
- ✓ Exporting observability data to AWS CloudWatch, Azure Monitor, or Google Operations Suite
- ✓ Enables deep visibility into ephemeral workloads without relying solely on cloud provider-specific tools.

Use Case #5 – Multi-Cloud & Hybrid Cloud Observability

✓ How OpenTelemetry Helps:

- ✓ Provides cross-platform observability without vendor lock-in
- ✓ Sends telemetry data from AWS, Azure, and GCP into one central monitoring system
- ✓ Helps correlate transactions spanning multiple cloud regions seamlessly

🎯 Key Benefit:

- ✓ Enables platform-agnostic observability, ensuring businesses have one unified visibility layer across multiple cloud providers.

So far ...

- ✓ OpenTelemetry solves key observability challenges in microservices, cloud-native, and hybrid cloud environments.
- ✓ Distributed tracing minimizes debugging time by showing how requests flow through services.
- ✓ Serverless workloads, multi-cloud apps, and external API monitoring are simplified with OpenTelemetry's trace & metrics collection.
- ✓ Businesses improve system performance, reduce downtime, and optimize costs using a vendor-neutral, open-source observability standard.

Distributed Tracing with OpenTelemetry + Jaeger / Zipkin / AWS X-Ray

- ✓ Jaeger – Open-source distributed tracing platform
- ✓ Zipkin – Popular tracing system originally developed at Twitter
- ✓ AWS X-Ray – Managed tracing service by AWS for serverless and cloud applications

Metrics Monitoring with OpenTelemetry + Prometheus + Grafana

- ✓ Prometheus – Collects & stores metrics like request latency, error rates, and system usage
- ✓ Grafana – Visualizes Prometheus data in dashboards

Request latency for checkout service

Database query response times

CPU & memory usage

OpenTelemetry for Logs – ELK Stack & Datadog

- ✓ ELK Stack (Elasticsearch, Logstash, Kibana) – The most widely used log aggregation platform
- ✓ Datadog Logs & Monitoring – Cloud-based full-stack monitoring tool
- ✓ Splunk, New Relic, Sumo Logic, etc.

Serverless & Cloud-Native Integrations (AWS, Azure, GCP)

- ✓ AWS CloudWatch + AWS X-Ray to monitor Lambda functions & ECS workloads
- ✓ GCP Operations Suite (formerly Stackdriver Monitoring) for tracing in Google Cloud
- ✓ Azure Monitor + Application Insights for full application observability

90% of requests are fast (~200ms), but 10% take 5+ seconds

Slow requests correlate with cold-start issues due to insufficient provisioned concurrency

Implementing OpenTelemetry – Challenges & Solutions

- ✓ 1. Complex Instrumentation & Learning Curve
- ✓ 2. Performance Overhead & Resource Usage
- ✓ 3. Managing & Storing High Volumes of Telemetry Data
- ✓ 4. Tooling & Integration Complexity
- ✓ 5. Lack of Expertise & Best Practices Compliance

Challenge #1 – Complex Instrumentation & Learning Curve

✓ Best Practices to Overcome This Challenge:

- ✓ Use Auto-Instrumentation: Many OpenTelemetry SDKs support auto-instrumentation for frameworks like FastAPI, Spring Boot, and Express.js—reducing the need for manual coding.
- ✓ Start Small, Scale Gradually: Instead of overhauling everything at once, begin instrumenting one critical service, learn from the process, and gradually expand.
- ✓ Establish Organization-wide Standards: Create guidelines & templates for developers to follow—standardizing how OpenTelemetry is configured across services.

Challenge #2 – Performance Overhead & Resource Usage

✓ Best Practices to Overcome This Challenge:

- ✓ Implement Sampling Strategies: Instead of collecting every trace, use techniques like probabilistic sampling to only capture representative requests (e.g., 10% of traffic).
- ✓ Use Batching & Compression: The OpenTelemetry Collector can batch and compress telemetry data before exporting, reducing network and storage overhead.
- ✓ Optimize Exporter Configurations: Configure exporters asynchronously to send data in background threads, ensuring minimal impact on request processing time.

Challenge #3 – Managing & Storing High Volumes of Telemetry Data

✓ Best Practices to Overcome This Challenge:

- ✓ Set Data Retention Policies: Define how long telemetry data should be stored based on its importance (e.g., only retain error traces for 30 days).
- ✓ Filter & Downsample Low-Priority Data: Only store high-value telemetry (e.g., transaction failures), while filtering out redundant data.
- ✓ Leverage Cost-Effective Storage: Consider archiving older telemetry data in cost-effective storage solutions like AWS S3 Glacier or log aggregation platforms with built-in retention controls.

Challenge #4 – Tooling & Integration Complexity

✓ Best Practices to Overcome This Challenge:

- ✓ Leverage OpenTelemetry Collector Pipelines: The Collector can translate telemetry data between different formats, ensuring smooth integration with existing tools.
- ✓ Adopt a Gradual Rollout Strategy: Start by mirroring telemetry data to both existing APM tools & OpenTelemetry backends, allowing teams to compare and validate results before fully switching.
- ✓ Use OpenTelemetry's Multi-Exporter Feature: OpenTelemetry allows sending telemetry to multiple destinations simultaneously, ensuring compatibility with old and new observability layers during transition.

Challenge #5 – Lack of Expertise & Best Practices

✓ Best Practices to Overcome This Challenge:

- ✓ Invest in Team Training: Provide engineers with hands-on OpenTelemetry workshops, tutorials, and documentation.
- ✓ Leverage Community & CNCF Resources: OpenTelemetry has an active CNCF community with Slack groups, GitHub discussions, and open-source contributors who can offer guidance.
- ✓ Use OpenTelemetry SDK Libraries & Pre-Built Instrumentation: Many frameworks already support built-in OpenTelemetry instrumentation, reducing the learning curve.

Summary

- ✓ Start small and scale gradually to avoid OpenTelemetry adoption pitfalls.
- ✓ Optimize data collection & storage to control performance impact and costs.
- ✓ Use OpenTelemetry Collector pipelines to integrate with existing observability stacks.
- ✓ Establish best practices & standardize instrumentation across teams.
- ✓ Invest in OpenTelemetry documentation and training to build internal expertise.



But how does it really work ?

Available Exporters in OpenTelemetry .NET

✓ Tracing Exporters:

- Jaeger (OpenTelemetry.Exporter.Jaeger)
- Zipkin (OpenTelemetry.Exporter.Zipkin)
- OTLP (OpenTelemetry Protocol) → Supported by Grafana Tempo, DataDog, AWS X-Ray, New Relic (OpenTelemetry.Exporter.OpenTelemetryProtocol)
- Console (for debugging)

✓ Metrics Exporters:

- Prometheus (OpenTelemetry.Exporter.Prometheus.AspNetCore)
- OTLP (for pushing metrics to third-party tools like DataDog/Tempo)

✓ Logs Exporters:

- OTLP (for structured logging)

C# OpenTelemetry Example (ASP.NET Core with Jaeger Exporter)

- ✓ Uses OpenTelemetry .NET SDK
- ✓ Configures Tracing & Metrics
- ✓ Sends collected telemetry to Jaeger

```
sh
dotnet add package OpenTelemetry.Api
dotnet add package OpenTelemetry.Extensions.Hosting
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.Http
dotnet add package OpenTelemetry.Instrumentation.SqlClient
dotnet add package OpenTelemetry.Exporter.Jaeger
dotnet add package OpenTelemetry.Exporter.Zipkin
dotnet add package OpenTelemetry.Exporter.Console
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore
```

Copy code

```
using OpenTelemetry.Trace;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Exporter;

var builder = WebApplication.CreateBuilder(args);

// Define OpenTelemetry resource attributes
var resourceBuilder = ResourceBuilder.CreateDefault()
    .AddService("MyMultiExporterApp");

// OpenTelemetry Tracing Setup
builder.Services.AddOpenTelemetry()
    .WithTracing(tracerProviderBuilder =>
        tracerProviderBuilder
            .SetResourceBuilder(resourceBuilder)
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddSqlClientInstrumentation()
            .AddJaegerExporter(options =>
            {
                options.AgentHost = "localhost"; // Send traces to local Jaeger instance
                options.AgentPort = 6831;
            })
            .AddZipkinExporter(options =>
            {
                options.Endpoint = new Uri("http://localhost:9411/api/v2/spans"); // Expose Zipkin endpoint
            })
            .AddOtlpExporter(options =>
            {
                options.Endpoint = new Uri("http://localhost:4317"); // Push traces to Otlp endpoint
            })
            .AddConsoleExporter()); // Debugging traces in console
```

```
// OpenTelemetry Metrics Setup (Prometheus)
builder.Services.AddOpenTelemetry()
    .WithMetrics(metricProviderBuilder =>
        metricProviderBuilder
            .SetResourceBuilder(resourceBuilder)
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddOtlpExporter() // Optional: Send metrics to external observability tool
            .AddPrometheusExporter()); // Expose Prometheus metrics endpoint

var app = builder.Build();

// Expose Prometheus Metrics for Scraping
app.UseOpenTelemetryPrometheusScrapingEndpoint();

// Map Simple API
app.MapGet("/", async () =>
{
    using var activitySource = new System.Diagnostics.ActivitySource("MyMultiExporterApp");
    using (var activity = activitySource.StartActivity("Processing Request"))
    {
        await Task.Delay(500); // Simulate processing delay
    }

    return Results.Ok("Hello from OpenTelemetry with Multiple Exporters!");
});

app.Run();
```

Instrumentation – How Does OpenTelemetry Collect Data?

OpenTelemetry collects data by injecting code hooks into your application's functions, libraries, or frameworks.

1. Auto-Instrumentation (Automatic)

- OpenTelemetry automatically instruments well-known frameworks like ASP.NET Core, HttpClient, SQLClient, and gRPC without modifying your code.
- It does this by attaching itself to method calls, much like a Profiler or AOP (Aspect-Oriented Programming) technique.

2. Manual Instrumentation (Explicit)

- You can directly tell OpenTelemetry when to create a "trace span" or collect a "metric" in your code.
- This is useful when tracking business logic inside custom application functions.

Example (C# Manual Tracing – Creating a Custom Trace Span):

📌 What Happens?

OpenTelemetry creates a timestamped trace event and records metadata (duration, tags, and logs)

If auto-instrumentation exists, it automatically creates a parent-child trace hierarchy.

The OpenTelemetry SDK processes and exports this event based on configuration.

csharp


Copy code

```
using System.Diagnostics;

var activitySource = new ActivitySource("MyApp");


// Manually create a trace span inside a request handler
using (var activity = activitySource.StartActivity("Processing Order"))
{
    // Simulated work
    await Task.Delay(500);
}
```

csharp

 Copy code

```
// Simplified internal code from ASP.NET Core's HostingApplication.cs
private void ProcessRequest(HttpContext httpContext)
{
    var diagnosticListener = new DiagnosticListener("Microsoft.AspNetCore");
    if (diagnosticListener.IsEnabled("Microsoft.AspNetCore.Hosting.HttpRequestIn"))
    {
        diagnosticListener.Write("Microsoft.AspNetCore.Hosting.HttpRequestIn", new
        {
            HttpContext = httpContext
        });
    }
}
```

csharp

 Copy code


```
using System.Diagnostics;

var listener = new DiagnosticListener("Microsoft.AspNetCore");
using IDisposable subscription = listener.Subscribe(new MyDiagnosticObserver());
Console.ReadLine();

// Define a custom observer to listen for events
public class MyDiagnosticObserver : IObservable<KeyValuePair<string, object>>
{
    public void OnNext(KeyValuePair<string, object> value)
    {
        Console.WriteLine($"Event: {value.Key}, Data: {value.Value}");
    }

    public void OnError(Exception error) { }
    public void OnCompleted() { }
}
```

csharp

 Copy code

```
using System.Diagnostics;

// Create a DiagnosticSource named "MyCustomApp"
var diagnosticSource = new DiagnosticListener("MyCustomApp");

Console.WriteLine("Press Enter to trigger an event...");
while (true)
{
    Console.ReadLine();
    if (diagnosticSource.IsEnabled("MyApp.Operation"))
    {
        diagnosticSource.Write("MyApp.Operation", new { Message = "Important operation" });
    }
}
```

csharp

 Copy code

```
DiagnosticListener.AllListeners.Subscribe(new AllListenersObserver());

class AllListenersObserver : IObservable<DiagnosticListener>
{
    public void OnNext(DiagnosticListener value)
    {
        if (value.Name == "Microsoft.AspNetCore") // Capturing ASP.NET core events
        {
            value.Subscribe(new EventObserver());
        }
    }

    public void OnCompleted() { }
    public void OnError(Exception error) { }
}

class EventObserver : IObservable<KeyValuePair<string, object>>
{
    public void OnNext(KeyValuePair<string, object> kvp)
    {
        Console.WriteLine($"Event {kvp.Key} captured: {kvp.Value}");
    }
}
```

🚀 Scenario	Should You Use `TraceIdRatioBasedSampler`?	Recommended Sampling Rate `p`
Development & Debugging	❌ No (Use `AlwaysOnSampler`)	$p = 1.0$
Microservices with High Volume	✅ Yes (Use `TraceIdRatioBasedSampler`)	$p = 0.1 - 0.2$
High-Traffic Production APIs	✅ Yes (Use `TraceIdRatioBasedSampler` with ParentBased)	$p = 0.05$
Critical Business Transactions	❌ No (Trace all critical requests)	$p = 1.0$

Component	DiagnosticSource Name	What It Emits
ASP.NET Core HTTP Requests	<code>Microsoft.AspNetCore.Hosting</code>	Incoming & outgoing web requests
HttpClient (Outgoing Requests)	<code>System.Net.Http</code>	All outbound API calls
SQL Queries (SQLClient)	<code>Microsoft.Data.SqlClient</code>	Executed SQL queries
Entity Framework Core (EF Core)	<code>Microsoft.EntityFrameworkCore</code>	Database query execution
gRPC Calls	<code>Grpc.Net.Client</code>	Traces gRPC client communication
File System I/O (System.IO.Stream)	<code>System.IO.Stream</code>	File read/write events
.NET Runtime Events	<code>System.Runtime</code>	GC collections, thread pool usage
DNS Queries	<code>System.Net.NameResolution</code>	Traces DNS lookups

Real-world Example (End-to-End Request)

- ◆ Step-by-Step: HTTP Request Lifecycle

- 1 User sends an HTTP request (GET /products)
 - 2 OpenTelemetry auto-detects incoming request and creates a trace span
 - 3 Trace is propagated to database, capturing SQL timing
 - 4 The response is returned – trace metadata is attached to the response headers
 - 5 OpenTelemetry exports the trace to Jaeger
 - 6 OpenTelemetry records logs in Graylog
 - 7 Prometheus scrapes HTTP request response timing and stores it as metrics
- 📌 The engineer can now view traces (Jaeger), metrics (Prometheus), and logs (Graylog) in real-time.

Question	Answer
How does OpenTelemetry collect data?	It injects hooks into frameworks via auto or manual instrumentation
How does it track requests across services?	It passes Trace IDs in HTTP headers (<code>`traceparent`</code>)
How does it send telemetry?	It buffers, processes, and exports data in batches
Where does the data go?	It exports to tools like Jaeger, Prometheus, Graylog via configured exporters
How does it avoid overhead?	Sampling, batch processing, and using the OpenTelemetry Collector

Under the Hood – How Does OpenTelemetry Hook Into Applications?

Mechanisms OpenTelemetry Uses to Hook Into Code

1. .NET Diagnostics & Event Listeners:

- OpenTelemetry listens to framework events using `System.Diagnostics.DiagnosticSource`.
- Example: When ASP.NET Core handles an HTTP request, OpenTelemetry auto-detects and logs it.

2. Method Interceptors & Middleware Wrapping:

- OpenTelemetry instruments built-in frameworks (e.g., HttpClient, Entity Framework) by wrapping method calls with telemetry hooks.

3. W3C Trace-Context Header Propagation:

- HTTP Requests include traceparent headers (traceparent: 00-a3f1...-1234-5678-01)
- Downstream microservices read these headers to maintain the same trace ID.

4. Dependency Injection Initialization:

Key Features in This Example

- ✓ Automatic request tracing (`AddAspNetCoreInstrumentation()`)
- ✓ Database queries tracing (`AddSqlClientInstrumentation()`)
- ✓ HTTP client requests tracing (`AddHttpClientInstrumentation()`)
- ✓ Jaeger exporter configuration (`AddJaegerExporter()`)
- ✓ Manual span creation (`ActivitySource` for custom tracing)

Recap – Why OpenTelemetry?

✓ Key Takeaways:

- ✓ OpenTelemetry unifies telemetry collection (Traces, Metrics, Logs) 🚀
- ✓ No vendor lock-in – Instrument once, send data anywhere ↺
- ✓ Distributed tracing simplifies debugging in microservices ☁️
- ✓ Optimizes system performance & reduces mean time to resolution (MTTR) 📊
- ✓ Works across cloud, hybrid, and on-prem systems 🏗️
- ✓ Integrates seamlessly with Jaeger, Prometheus, ELK, AWS X-Ray, Datadog, and more 💡
- ✓ Organizations that adopt OpenTelemetry save time, money, and effort on observability costs & maintenance. 🎯

How to Get Started with OpenTelemetry

- ◆ ❶ Pick a Language SDK: Start with OpenTelemetry for Python, Java, Go, .NET, or JavaScript.
- ◆ ❷ Use Auto-Instrumentation Where Possible: Many frameworks already support OpenTelemetry out-of-the-box to reduce manual effort.
- ◆ ❸ Deploy the OpenTelemetry Collector: Use it to process and export telemetry efficiently.
- ◆ ❹ Choose Exporters & Integrate With Your Existing Tools: Send data to Prometheus, Jaeger, Datadog, AWS X-Ray, or ELK.
- ◆ ❺ Optimize & Fine-Tune Observability Setup: Use sampling, data filtering, and batching to reduce resource overhead.

Resources & Learning More

- 📌 Official OpenTelemetry Docs: <https://opentelemetry.io/docs/>
- 📌 GitHub Repository: <https://github.com/open-telemetry>
- 📌 CNCF OpenTelemetry Community: <https://cloudnative.slack.com/>
- 📌 Interactive OpenTelemetry Sandbox: Explore real-world OpenTelemetry examples
- 📌 Tutorials & Blogs from Observability Experts

Thank You + Q&A