Full-Stack Observability Essentials

Using OpenTelemetry for Flexibility

JOANA CARVALHO
PERFORMANCE ENGINEER, POSTMAN

CONTENTS

As applications become more and more complex, there is an increased need to change how organizations monitor the state of their entire system. Increased migration to the cloud brings different, more elaborate architecture paradigms — microservices and all components they enable. Although microservices enable faster, more aggressive scaling, flexibility, and cost control, it becomes harder for a single person to assess the system's overall status.

Observability (affectionately abbreviated as "o11y") brings clarity; it empowers teams to ask questions about their system and business and get clear answers driven by the signals collected. The challenge is getting started:

How to instrument? Auto-instrumentation? Manual? Both?
At what point in the product lifecycle to start?
What framework or vendor to select?

These questions can quickly become overwhelming. OpenTelemetry provides the freedom to instrument a system once and maintains the flexibility of choosing any back-end observability or analytics solution without getting locked into a single vendor.

WHAT IS OBSERVABILITY?

Observability was introduced in control theory by the engineer [Rudolf Kámál](). He defined it as "the ability to understand the current state of a system using only its external outputs." When a system is a black box, we cannot understand it; therefore, it becomes harder to maintain and predict. That's where high-quality telemetry and observability enter.

Observability and telemetry signals — logs, metrics, traces, events, and metadata — work together to correlate individual systems' health with the business' overall health. It highlights what's going on within the systems, processes, and microservices of an entire tech stack purely from the existing data streams collected. Ultimately, observability gives developers and operations teams a greater understanding of their systems. To understand and manage services, we need to measure, observe, and quantify their behaviors.

We define KPIs that make sense for our domain to determine what's expected of our service. Defining these KPIs is paramount to understanding what actions need to be taken if the appropriate level of service is not delivered — ideally before the service consumer notices degradation.

The four concepts described in Figure 1 on page three help build a relationship of trust that can be maintained and quantified.

chronosphere

The shift to cloud native demands a new approach to observability

How quickly can you find the root cause of a problem? An explosion of data can slow you down and drive up costs. Learn how to know, triage, and understand root cause to achieve faster remediation.

READ THE EBOOK

chronosphere

The shift to cloud native demands a new approach to observability

Great observability leads to competitive advantage, world-class customer experiences, and happier developers. However, an explosion of data in a cloud native world can slow you down and drive up costs. Understanding the three phases of observability means being able to know, triage, and understand root causes to achieve faster remediation.

READ THE EBOOK

Figure 1

SLA	Text agreement between a provider and a consumer to set expectations for the deliverable (e.g., responsiveness, performance), also defining the consequences of failing an SLO.
SLO	Value or range of values that serves as a benchmark for SLIs. These should be meaningful, acceptable, measurable, and achievable (e.g., uptime).
SLIs	Measurements tied to the consumer expectations of the service. They provide the definition of good for the consumer and the actions on failure (e.g., latency, response time, availability).
Error Rate	The maximum time a system can be unavailable to the end user (e.g., availability SLO of 99.99% [4 nines] equals a total downtime of 52.56 minutes in a year).

THE DIFFERENCE BETWEEN OBSERVABILITY AND MONITORING

Let's start by saying that observability doesn't replace monitoring; quite the contrary — observability is meant to amplify its potential.

- **Monitoring** is an action; a human or an automated process can do it if they know what signals to look for. It can generate alerts, provide insights, suggest actions, measure traffic or real-user activity, and warn when issues occur.
- **Observability**, on the other hand, lets you understand why the problem occurred. It is an approach that enables teams to ask questions about the holistic state of a system.

You can monitor a system, but if the telemetry is ineffective, insufficient, or inaccurate, there is no way to infer its status, and then observability cannot be achieved.

KEY NOTIONS AND CONCEPTS OF FULL-STACK OBSERVABILITY

Full-stack observability is the ability to understand what is happening in a system at any time. Collecting, correlating, and aggregating telemetry from the system's components provides insight into its behavior, performance, and health. Through full-stack observability, teams can fully understand the dependencies across domains and system topology.

Contrary to traditional monitoring systems, full-stack observability enables IT teams to proactively react, predict, and prevent problems using artificial intelligence and machine learning, which, considering the amount of data collected, would be nearly impossible otherwise. Presenting the data in unified analytics and intuitive dashboards can give the observer a complete picture of the system's health.

SYSTEM STATE

By instrumenting every line of code, endpoint, hardware, or software piece, we are generating outputs and signals that document its activity. When these signals are analyzed together, they can provide insight into the relationships and dependencies of the system.

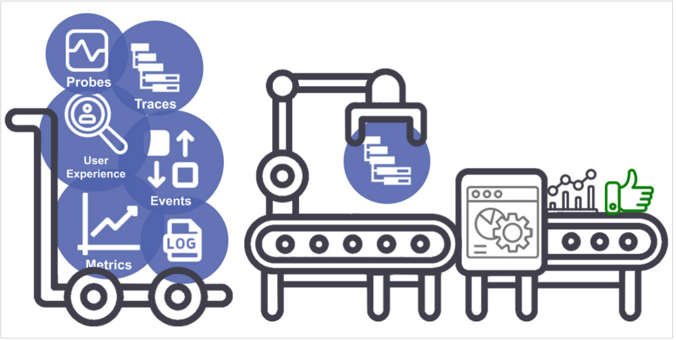
The table below notes some of the more common signals:

Table 1

SIGNAL	DESCRIPTION
Metrics	A numerical representation of a value calculated or aggregated over a period (e.g., percent of memory used, queue size of a message broker, number of errors per second)
Events	• Immutable time-stamped records of events over time • Usually emitted by the service due to actions in the code
Logs	• Lines of text describing an event that occurred, usually resulting from a block of code execution • Can be output in plain text, structured text (like JSON), or binary code • Handy for troubleshooting systems less prone to instrumentation (e.g., databases, load balancers) • Tend to be more granular than events
Probes	• The most basic implementation of observability • Usually achieved by exposed endpoints used by external components (e.g., load balancers, app servers) to make decisions based on the returned state
Traces	• Represent the flow of a single transaction or request as it goes through a system • Show the path followed, each component's latency added, and other relevant information that might point to bottlenecks or issues • Should be able to answer questions quickly and clearly about system availability and performance
User experience	The end user's perspective when interacting with the software (e.g., Application Performance Index [Apdex])

If you ever Google-searched "observability," you might have come across the Three Pillars of Observability: logs, traces, and metrics. Often, this automatically creates a visualization in our minds, similar to the orbs illustrated below:

Figure 2: Common telemetry signals collected and processed



With that mental image, it is easy to jump to conclusions, such as assuming that a metric line starting to increase means something is wrong, which is only one part of the picture in many cases. With telemetry, it's not just about quantity or spending all storage quota on it but about quality and relevance — the questions it answers.

MICROSERVICES

The single responsibility principle was first coined by Robert C. Martin and became the base of the microservices philosophy:

"Gather together the things that change for the same reasons. Separate those things that change for different reasons."

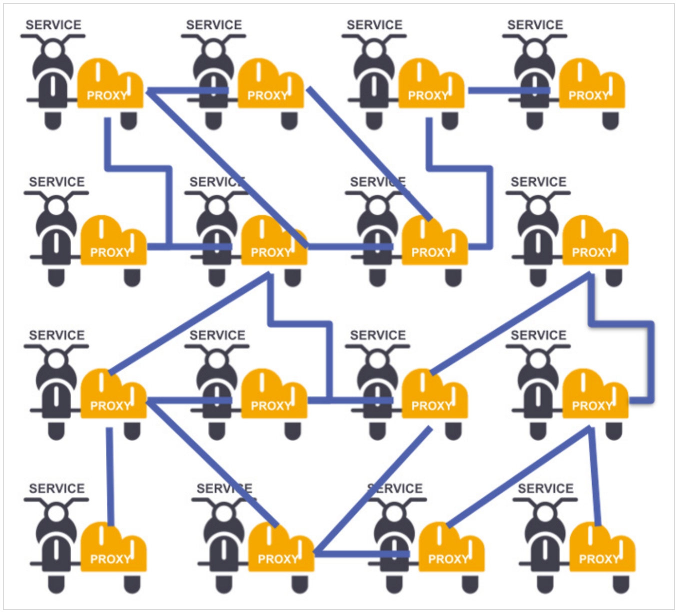
A microservice architecture follows this approach by arranging an application into smaller, loosely coupled services that can be developed, deployed, and maintained independently. Communication happens through application programming interfaces (APIs) that serve as building blocks of the overall system while isolating other components from issues that can negatively impact the system. Development teams can select the best stack for the problem at hand — being able to alter, enhance, and scale without crossing other services' borders.

This architecture also brings challenges as companies can easily reach hundreds or thousands of microservices, making security, observability, and network policies more complex. As the number of deployed services increases, handling communication, observability, and errors becomes a monstrous task.

One of the patterns used to deal with these challenges is service meshes, which are useful for companies with services starting in the hundreds that need to manage, observe, and secure communication between them by moving the logic from the services into an infrastructure layer.

Although not all service mesh implementations are what's shown in Figure 3, most requests are routed between microservices through proxies that live in their infrastructure, decoupling business logic from network function. The individual proxies sit beside the service — sometimes called a "sidecar" for that reason. All these decoupled proxies form a service mesh network.

Figure 3: Service mesh example



The connectivity between services made by their individual proxies

Service meshes can help bring visibility into the application layer without code changes, making it easier to configure metrics collection. All traffic goes through the mesh proxies, enabling greater visibility into all service interactions. Each proxy reports its portion of the interaction, providing information on inbound and outbound traffic and service-level metrics. Access logs with service calls and distributed traces, generated by the mesh on every service within it, make it easier to follow requests across multiple services and proxies.

TELEMETRY QUERYING

Storing all telemetry from a system in one place is compelling; however, IT teams must be able to interrogate the data to answer meaningful questions. They can rely on query languages for analysis, which can then be used to create visualizations, troubleshoot, perform business analysis, etc. Let's see some querying languages in action using OpenTelemetry DevStats to check how many companies have contributed over the last 10 years:

Table 2

LANGUAGE	QUERY	RESULT
PostgreSQL	`select count(name) from shcom where series = 'hcomcontributions' and period = 'y10'`	> 597
PromQL	`sum_over_time(shcom{series="hcomcontributions"}[10y]) by(name)`	
Atlas	`series,hcomcontributions,:eq, now-10y,now,:time-span, :count`	
Flux	`from(db: "shcom") |> range(start: -10y) |> filter(fn: (r) => r.series == "hcomcontributions") |> count()`	

Combining OpenTelemetry with the flexibility of a powerful query language allows users to gain more value from the telemetry stored. Using the query languages mentioned here, we can jump into the next section to see how we can present the results in simple ways.

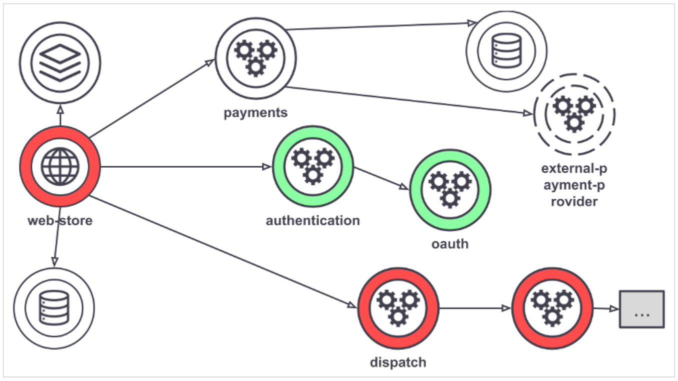
VISUALIZATION AND REPORTING

Gathering and correlating all system data is challenging, especially without the tools or techniques to visualize and process the telemetry in a meaningful manner, which is vital. Understanding dependencies throughout the stack and across services, processes, and hosts give observers and SRE teams the ability to predict and understand the topology of their service.

It's essential to understand the intended user when setting up dashboards, reports, or single visualizations. Give each different persona the level of detail and context they need, reducing the effort of reading and interpreting the information provided. Visualizations are a great way to deliver context and information — following are several visualization types that each provide value in unique ways.

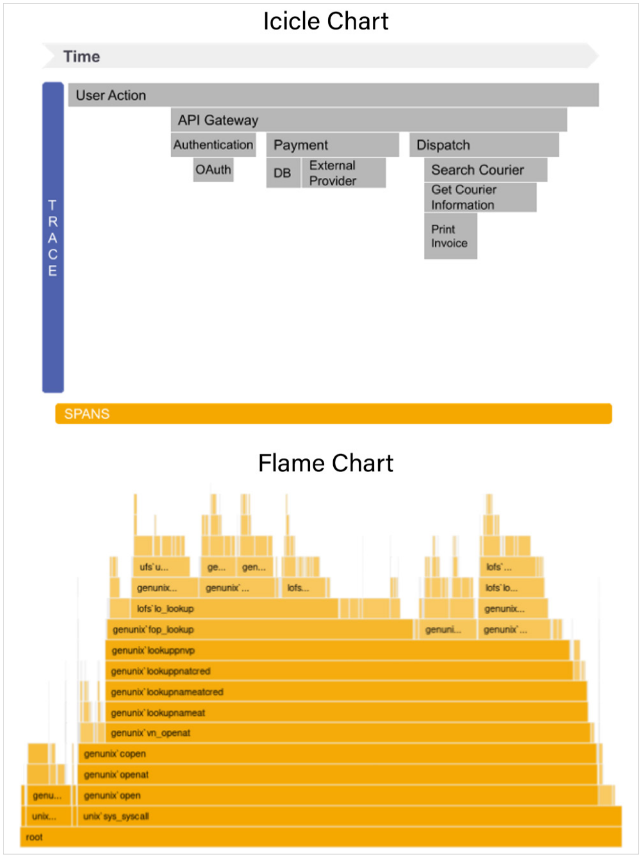
In multi-cloud environments, applications can reach millions of components, making it incredibly challenging to understand the context between them. One way to visualize their relationships is through **topology maps** (Figure 4). Analyzing distributed traces is often one of the more effective ways to perform a root cause analysis, which becomes easier to profile with the help of visual aids.

Figure 4



Flame charts and **icicle charts** (Figure 5) help developers see the relationship between service calls and impact on a single trace, such as unusually high latency or errors, and how they impact other calls and services. They represent each call on a distributed trace as a time-stamped horizontal or vertical line that contains details for each span.

Figure 5



A **trace map** (Figure 6) shows the connection between spans and lets one quickly understand the relationship between services. Like trace maps, **trace trees** (Figure 7) are represented vertically with the parent span as the root node.

Figure 6

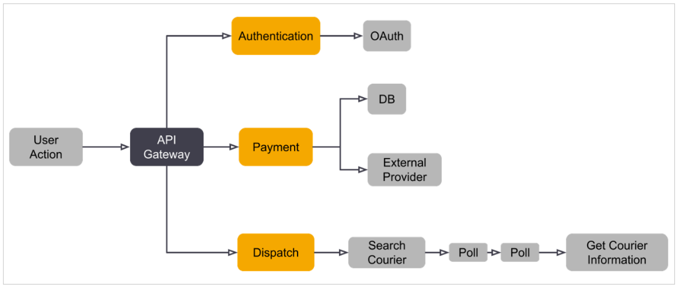
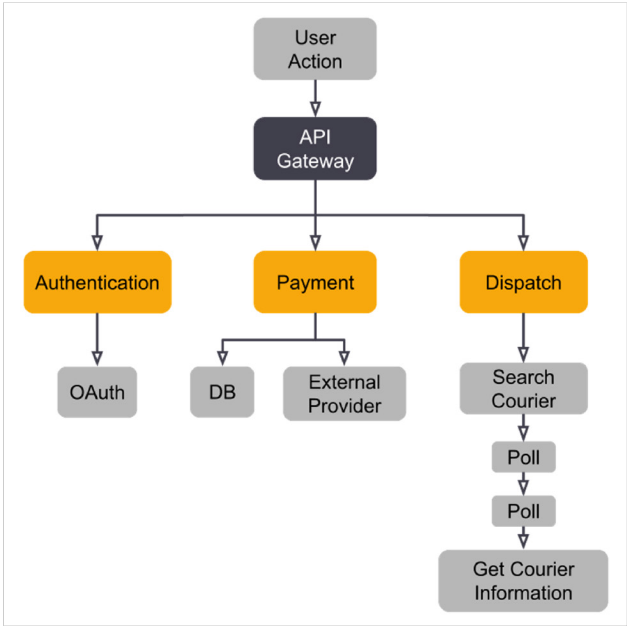
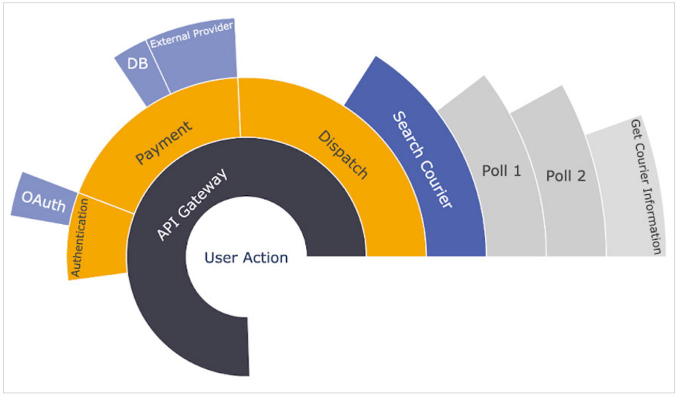


Figure 7



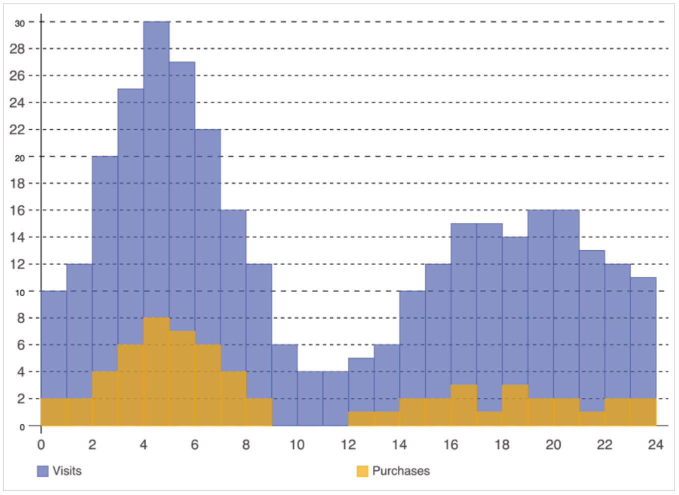
A **sunburst diagram** (Figure 8) illustrates a trace using a set of rings and semirings. The circle is the parent span, and the semirings fan out to describe the parent-child relationship.

Figure 8



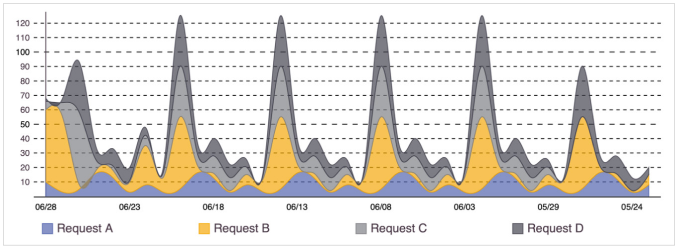
Histograms illustrate data distribution over a continuous interval or period and demonstrate the frequency of occurrences to quickly identify gaps or atypical values. In Figure 9, we can easily identify website traffic in relation to purchases.

Figure 9



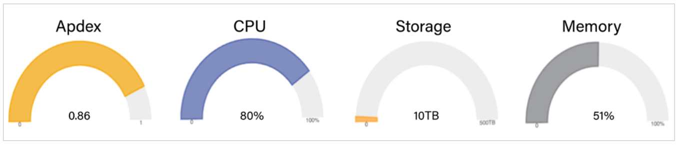
Similar to a line chart, an **area chart** has a color filling underneath that represents the total value at each point. When using more than one series for comparison, they are stacked to show how individual values affect the whole. Figure 10 shows changes over time and how each request impacts the total volume:

Figure 10



A **gauge chart** is one of the most effective visualizations for making quick decisions, as it shows one metric, its scale, and where a value is along that scale (e.g., a car speedometer). In Figure 11 are examples where this visualization can be used:

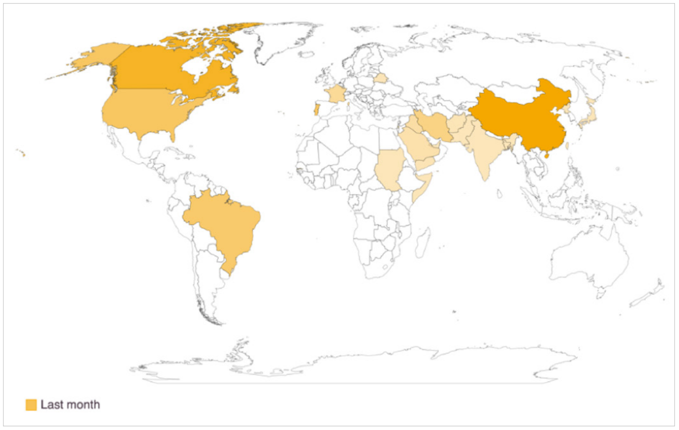
Figure 11



Unlike other charts, **geographical charts** are most valuable when they have added interactivity to see the exact values. Their use of color gradients to represent numerical variations across geographical regions helps to quickly identify data such as that shown in Figure 12, conversion rates by country.

SEE FIGURE 12 IN NEXT COLUMN

Figure 12



SHIFT-LEFT OBSERVABILITY

Today, organizations think about security, performance, and quality earlier in a product's lifecycle. There is a strong argument in favor of the shift-left principle happening across the industry — and observability practices are no exception. As we've extensively covered here, observability is meant to answer questions, one of the most important when developing a new change being, "Should this go to production?"

The **shift-left** principle involves moving testing, assessment, quality, and performance to the earlier stages of the DevOps lifecycle. This way, teams can have feedback sooner, and the cost and complexity of solving issues decreases while the frequency and speed of delivery increases. **Shift right**, while similar, is done in production environments. It's a practice gaining traction as demand for more performant, resilient, reliable software increases. Using real-world environments for testing under the same conditions that users experience enables the detection of bugs that would be harder to identify in lower environments. It also helps ensure that added noise doesn't affect the experience.

Integrating observability principles into the entire SDLC helps apply and maintain the same standards and SLOs used in production. Using the telemetry to create pipeline gates to detect anomalies and validations for acceptance tests prevents issues from bubbling up. If caught early, problems are easier to handle, cost less to fix, and require less context switching from teams, as the concepts are still present.

Another (significant) advantage of shifting observability left is increasing stakeholders' visibility of the complexities and challenges that teams face. Seeing the status of applications from left to right can also reduce the finger-pointing culture and help developers feel more connected to production environments.

Make the right telemetry and proper visualizations available to meet people's needs. Irrelevant or too much information is the first factor to contribute to ignoring dashboards. Remember that observability is as relevant for engineers as it is for businesses. Organizations can achieve a blameless culture when there is an understanding of why an incident occurred, how to handle it, and how we can prevent it in the future.

OPENTELEMETRY

Inspired by OpenTracing and OpenCensus, OpenTelemetry has one goal: to give developers a vendor-agnostic specification for telemetry, thus enabling teams to trace a request from start to finish by instrumenting each transaction. Since introducing these tools, the industry has recognized the need for collaboration to lower the shared cost of the software instrumentation required to gain this visibility. OpenTracing and OpenCensus have led the way in that effort.

OpenTracing became a Cloud Native Computing Foundation (CNCF) project in 2016, and Google made the OpenCensus project open source in 2018. The two competing tracing frameworks shared the same goal but weren't mutually compatible. Although competition usually means better software, this is not necessarily true in the world of open source. It can often lead to poor adoption, contribution, and support. To stop "the Tracing Wars," it was announced at KubeCon 2019 in Barcelona that the projects would converge into OpenTelemetry and join the CNCF. Hence, OpenTelemetry, or OTel for short, was born.

GOAL AND AUDIENCE

With the ultimate goal of providing a unified set of vendor-neutral standards, libraries, integrations, APIs, and software development kits (SDKs), OpenTelemetry has become the de facto standard for adding flexible full-stack observability to cloud-native applications. This open standard enables companies with any technology stack to gather observability data from all their systems.

As OpenTelemetry is open source, the maturity level of each component will depend on the language and the interest taken by that particular community. The following table shows the current maturity status of its elements for some of the languages it supports:

Table 3: OpenTelemetry code instrumentation state

LANGUAGE	TRACING	METRICS	LOGGING
Java	Stable	Stable	Experimental
.NET	Stable	Stable	ILogger: Stable OTLP log exporter: Experimental
Go	Stable	Experimental	Not yet implemented
JS	Stable	Development	Roadmap
Python	Stable	Experimental	Experimental

INSTRUMENTATION

There are two main ways to instrument apps using OpenTelemetry: manual and auto-instrumentation. If you had to instrument every microservice manually, it would likely mean spending almost as much time building and maintaining telemetry as building and maintaining the software itself. That's where auto-instrumentation steps in to make it possible to collect application-level telemetry without manual changes to the code — it allows tracing a transaction's path as it navigates different components, including application frameworks, communication protocols, and data stores.

Four instrumentation libraries are available:

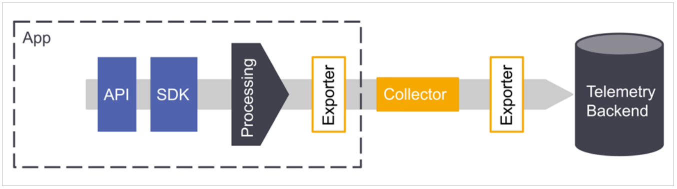
- **Core** contains all language instrumentation libraries available.
- **Instrumentation** adds to the Core library by adding extra language-specific capabilities.
- **Contrib** includes extra helpful libraries and standalone utilities that don't fit the scope of the previous two.
- **Distribution** adds vendor-specific customization.

Tracing all operations involved in a transaction provides an end-to-end view of how the service functions. We can then visualize, aggregate, and inspect the data to understand the experience of individual users, identify bottlenecks, and map out dependencies between services.

KEY COMPONENTS

When integrating OpenTelemetry into your application stack, the telemetry delivery pipeline will look like Figure 13. The topology will depend on how your application architecture is structured and how deeply you want to instrument.

Figure 13: OpenTelemetry pipeline



Source: Schema based on "OpenTelemetry: beyond getting started"

When opting to use OpenTelemetry, these are the pipeline components to consider:

- **APIs** – Define how applications speak to one another and are used to instrument an application or service. They are generally available for developers to use across popular programming languages (e.g., Ruby, Java, Python). Because they are part of the OpenTelemetry standard, they will work with any OpenTelemetry-compatible back-end system moving forward — eliminating the need to re-instrument in the future.
- **SDK** – Is also language-specific, providing the bridge between APIs and the exporter. It can sample traces and aggregate metrics.
- **Collector** – Processes, filters, aggregates, and batches telemetry. It will enable greater flexibility for receiving and sending data to multiple back ends. It has two primary deployment models:
  - As an Agent that lives within the application or the same host as the application, acting as a source of data for the host (by default, OpenTelemetry assumes a locally running collector is available).
  - As a Gateway working as a data pipeline that receives, exports, and processes telemetry.

The collector consists of three components:

Table 4

RECEIVER	PROCESSOR	EXPORTER
Can push or pull data that will get into the collector (e.g., Jaeger, Prometheus).	Sit between receivers and exporters and run on the data; filter, format, and enrich data before it's sent through the exporter to a back end. Although not required, some might be recommended based on the data source.	Can push or pull data into one or multiple configured back ends or destinations (e.g., OTLP, Kafka). It separates instrumentation from the back-end configuration, so users can switch back ends without re-instrumenting the code.

OpenTelemetry is a library framework for receiving, processing, and exporting telemetry, which requires a back end to receive and store the data — the purpose of the collector.

WHY USE A STANDARD SPECIFICATION?

Correlating and analyzing data can be cumbersome for organizations trying to obtain insight into their applications, especially nowadays with highly ephemeral infrastructure and countless services to manage. OpenTelemetry aims to simplify data collection to focus on data analysis and processing while creating a standard that eliminates the previous proprietary and in-house solutions.

With 93 pull requests per week in all repositories and more than 450 companies backing up and maintaining the project in the last year alone, OpenTelemetry provides access to an extensive set of telemetry collection frameworks. The community's involvement also means it will respond and offer support with new technologies, without users having to wait for a vendor's support.

CONCLUSION

To reach full-stack observability of modern distributed application stacks, data must be collected, processed, and correlated visually from the entire system. Achieving this can be challenging for development teams when they have to support all stacks, frameworks, and providers within their organization. By standardizing how frameworks and applications collect and send observability data, OpenTelemetry aims to solve some of the challenges created by the heterogeneity of stacks, giving teams a vendor-neutral, portable, and pluggable solution that's easily configured with open-source and commercial solutions alike.

ADDITIONAL RESOURCES:

- OpenTelemetry Documentation – https://opentelemetry.io/docs
- OpenTelemetry DevStats Dashboard – https://opentelemetry.devstats.cncf.io/d/8/dashboards
- "Getting Started With OpenTelemetry" Refcard – https://dzone.com/refcardz/getting-started-with-opentelemetry
- "Getting Started With Log Management" Refcard – https://dzone.com/refcardz/log-management
- OpenTelemetry implementation status per language – https://github.com/open-telemetry/opentelemetry-specification/blob/main/spec-compliance-matrix.md
- OpenTelemetry Twitter – https://twitter.com/opentelemetry

WRITTEN BY JOANA CARVALHO,
PERFORMANCE ENGINEER, POSTMAN

Joana has been a performance engineer for the last 10 years. She analyzed root causes from user interaction to bare metal, performance tuning, and new technology evaluation. Her goal is to create solutions to empower the development teams to own performance investigation, visualization, and reporting so that they can, in a self-sufficient manner, own the quality of their services. Currently working at Postman, she mainly implements performance profiling, evaluation, analysis, and tuning.

600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300