Observability for developers

What is observability?

IBM

Contents

01 →
Introduction

02 →
What is observability?

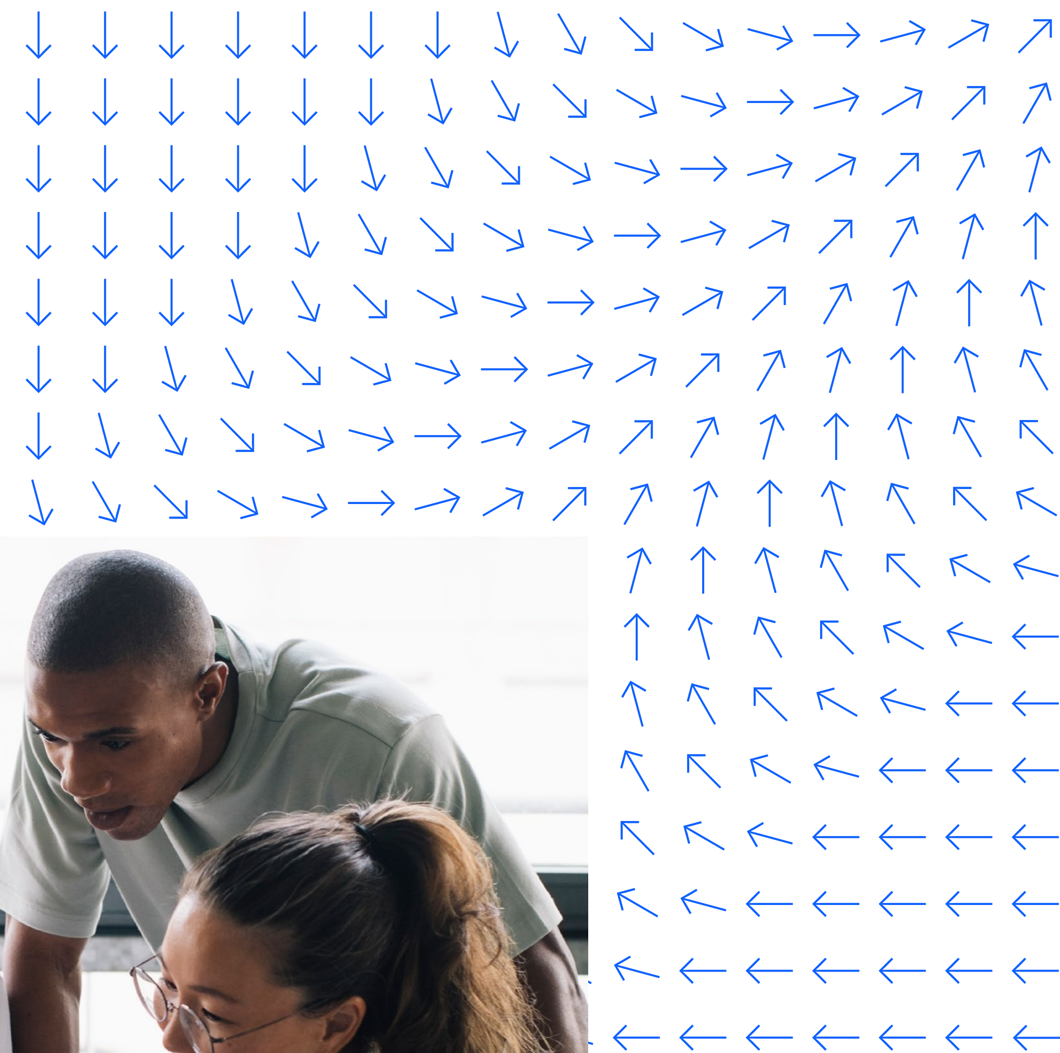
03 →
How to achieve full end-to-end observability

04 →
Observability standards and open source

05 →
SLO methodology

06 →
Beyond open source

07 →
Is IBM Instana right for you?



01

Introduction

Developers are faced with a growing challenge: How do we troubleshoot software that may be comprised of many disparate services running in a variety of languages and platforms? How can we notice critical changes, see into our black box services, and discern the true causes of errors?

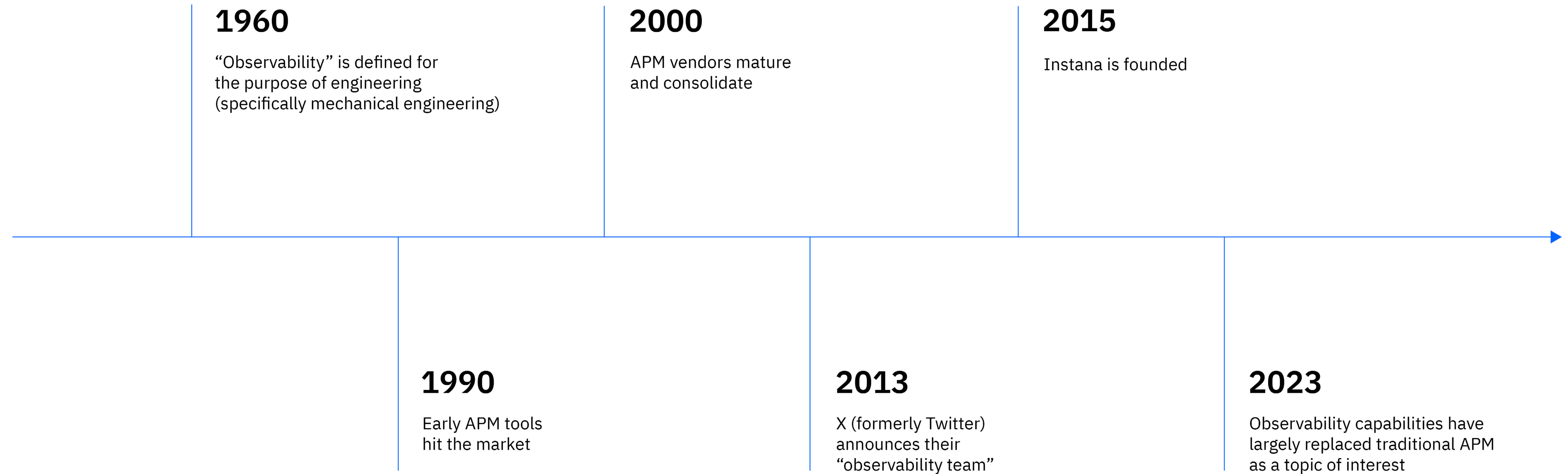
Not too long ago, debugging a program usually meant one thing: browsing error logs. This approach was fine for small teams running simple, local programs in a small number of instances.

But things in IT are always changing. Software architecture paradigms have evolved from monoliths to microservices. Developer responsibilities have changed with the emergence of DevOps, which represents a shift in the way developers take responsibility for programs after delivery.

The origins of observability
The term observability was first defined for engineering purposes in R.E. Kalman's 1960 paper titled "On the general theory of control systems."[1] The term observability as it pertained to mechanical engineering was defined as the ability to understand the inner state of a system by measuring its outputs.

Fifty years later, developers and software professionals were still monitoring their systems using tedious instrumentation tools—if they were monitoring at all. As far back as 2013, the shift toward distributed systems was already underway. That's when X (formerly known as Twitter) announced they were creating a new "observability team" to centralize and standardize the collection of telemetry data across Twitter's hundreds of services.[2] The term observability enters the mainstream.

Observability today
Over a decade later, IT teams continue to face services that are even more granular and job responsibilities that are more cross functional. Microservices are giving way to serverless, with DevOps leading to site reliability engineering (SRE). Observability is as important as ever and the scope of the challenge is growing exponentially.



What is observability?

Scientific definitions aside, observability in software development is looking at what's happening inside an application or system—and identifying what we're seeing. In short, observability is visibility plus understanding.

Modern distributed software systems absolutely require observability tools. There's simply no other way for a developer to constantly monitor the complexity that comes when you break applications into many tiny pieces. Luckily, this complicated architecture is also the very thing that allows observability tools to exist.

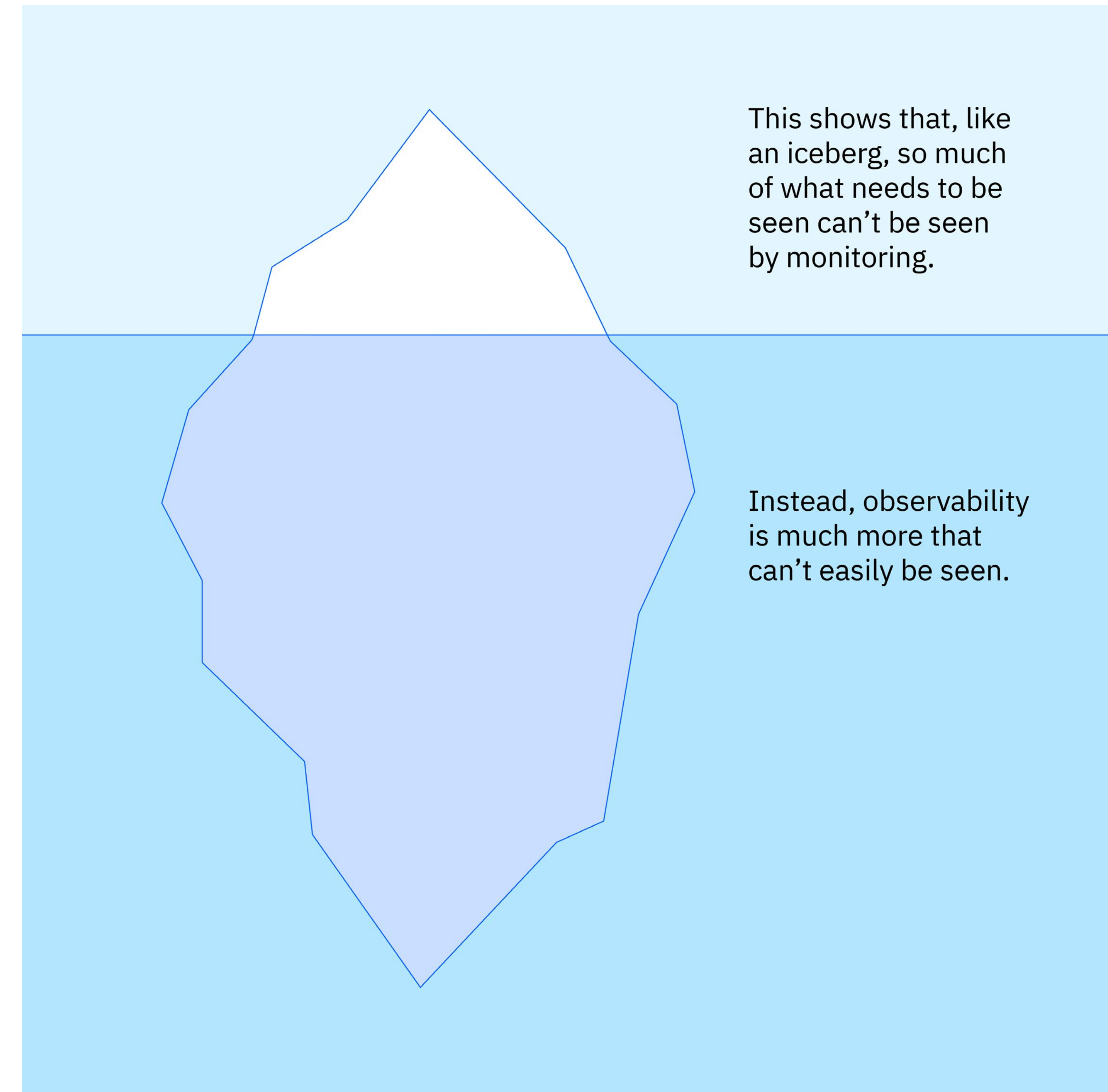
Software systems enable unified observability because services typically have a universal control plane and rely on common languages such as HTTP, RPC or something else to communicate with each other. In addition, it would be difficult to imagine relying on outdated logs to keep IT operations running smoothly.

How is observability different from monitoring?

You can't get a true picture of where you're going if you don't know where you've been. Application performance monitoring (APM) is an important set of capabilities that coexist with modern observability.

Traditional monitoring focuses on measuring predefined aspects of known components. That was a big idea in the 1980s, but the distributed architectures today include application components that are always changing, sometimes every second. Traditional APM tools simply can't keep up with this fast-moving, dynamic environment.

Obviously, the worst time to discover you're missing some crucial metric is when you're in the middle of triaging a production outage. This is where observability becomes critical. Instead of predetermining what to measure, observability tools see everything that's happening between, and even inside, services. This functionality lets you answer questions you couldn't have even anticipated—such as when you confront unknown unknowns.



Monitoring can be useful. Sometimes. Monitoring tools and metrics-based alerts are great for answering basic questions such as:

- Is my service online and available to customers?
- What % of requests have errors?
- What is the average latency of a request?
- How much memory is Redis using?
- Do I need to make a linear scaling action?

However, in the course of maintaining healthy applications, we often need to be able to answer deeper questions like these:

- Are users experiencing any errors along business-critical paths?
- Why is the error rate dramatically higher for this specific subset of customers?
- Where is the bottleneck in the system causing latency for a specific endpoint?
- What else changed within my system that could be causing this error?
- What service do I need to update to fix this error?

How to achieve full end-to-end observability

Answering those deeper questions requires building a system that monitors the complete lifecycle of a user request—from the client to the persistence layer. For application developers, this means instrumenting clients and backends to emit useful telemetry data.

Once we have this telemetry, we need to collect it, process it, store it and analyze it in order to turn the data into useful, actionable insights. Typically, an agent or collector gathers the telemetry signals and forwards to a database for storage. Time-series and columnar databases play a very important role in the efficient storage and querying of metrics data.

Finally, for analysis and insights, a UI is required in order to query the database. This will allow the data to be displayed as graphs and dashboards. Alerts also need to be configured to quickly notify devs if there's a problem including automated root cause analysis and smart alerts.

The telemetry signals
These are the data formats used to collect information from application services.

- **Traces:** Rich, request scoped timelines
- **Events:** Structured logs tracking external changes like deployments
- **Metrics:** Aggregable numbers about events or systems
- **Profiles:** Run-time level metrics
- **Logs:** Timestamped records of events
- **Exceptions:** Structured logs for tracking errors

It's important to keep in mind no one signal or set of signals constitutes observability. However, distributed tracing is arguably the most important signal in observability.

What are distributed traces?
Most every developer is familiar with stack traces—they pop up every day in error logs. But distributed tracing is what you get when the function calls of a single service are replaced with the network calls of a distributed application.

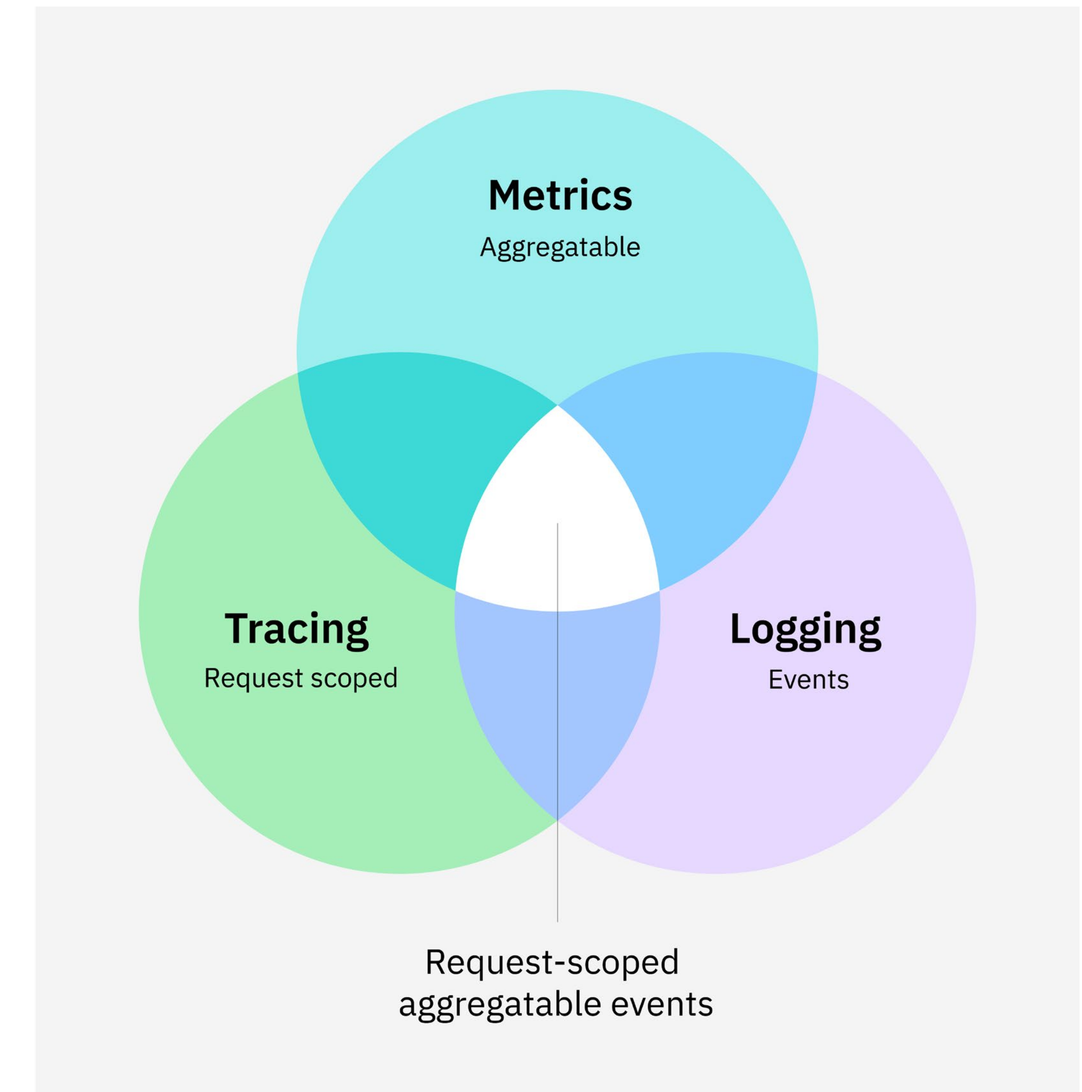
Distributed traces are made up of “spans”—a derivative of timespans. Each span includes a start time, an end time, any number of custom attributes and a reference to its parent span. Whenever a service processes a request, it creates a span that references the span from the calling service. This is received through the request headers following the W3C Trace Context specification.

Custom spans can also be created to add more granularity to the trace within a specific service. For example, if there’s a function that does a lot of processing after receiving results from a database, it might be helpful to enclose that function in a custom span.

But traces are more than just timelines. They’re also about much more than just latency. By tracing the path of individual requests, you can contextualize logs, metrics and other signals in a way that helps answer questions from a user-centric (request-scoped) point of view. Traces include a data structure that helps you:

- Understand request flows through the entire system.
- Instantly visualize your system topology.
- Derive metrics from the richness of trace metadata.
- Enrich logs with context by attaching them to a specific span.

Putting it all together
While these signals are sometimes useful in isolation, you can get the most valuable answers when all of this data can be correlated and searched in a meaningful way. Peter Bourgon, the well-known software engineer, elegantly laid out the crucial sweet spot for information where metrics, traces and logs overlap.[3] The results are request-scoped, aggregable events.



Observability standards and open source

In 2019, the OpenTracing and OpenCensus projects merged and became OpenTelemetry. This Open Source project under the Cloud Native Computing Foundation (CNCF) is quickly establishing open standards for telemetry that are widely adopted by both open-source tools and vendor observability platforms that include IBM® Instana®.

The OpenTelemetry project works in three areas to simplify the collection of telemetry data: specifications and standards, instrumentation tools, and pipeline tools.

The specifications include:

- OTLP—a binary format for efficiently representing metrics traces and logs.
- The W3C Trace Context specification for propagating a parent traceId to downstream services.
- Language APIs that you can call from your application or library code.

The instrumentation tools include:

- Language SDKs that connect the Language APIs to specific implementations. These can be either so-called standard ones or plugins provided by a community or vendor and used for processing and exporting telemetry.
- Auto-Instrumentation libraries that automatically call the Language APIs when relevant events occur within your framework or library.

And finally, for processing and transmission, there is the OpenTelemetry Collector. This expansive tool can act as an agent on your node to collect and forward all telemetry data. There is also a large ecosystem of receivers, processors and exporters available as plugins.

Open-source observability backends
The OpenTelemetry project does not provide a backend for storing and analyzing your monitoring data but a number of open-source tools are compatible with the OTLP signals.

The OpenTelemetry Demo project provides an example microservices application that can be run in docker or Kubernetes. It includes three two open-source telemetry databases: Prometheus for metrics and logs, and Jaeger for traces. It also includes Grafana for visualizing some of the metrics from Prometheus.

05

SLO methodology

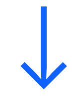
By now, you've probably been hearing a lot about service level objectives (SLOs) and service level indicators (SLIs).

SLO methodology represents a new way to think about software performance and health that goes hand in hand with observability. Observability as a concept comes from control theory, which focuses on the optimal—not perfect—operation of machines. SLO methodology teaches that IT teams who strive for perfection often achieve worse outcomes than when setting realistic targets.

It all starts with service level indicator. An SLI is any metric or statistic that can be converted into a percentage. In the context of running software, SLIs are things like the percentage of requests that were served successfully or the percentage of requests that had acceptable latency.

An SLO is a target bound for an SLI. SLOs are often expressed as a certain "number of nines." For example, 4 nines would indicate that an SLI meets the target 99.99% of the time. Very importantly, SLOs should never be "100%." This is unrealistic in almost every situation. It's best to leave yourself an error budget—some leeway for planned and unplanned outages.

In fact, one team at Google found that they could increase overall system reliability by artificially causing downtime for their service in order to prevent other teams from expecting it to be 100% reliable.[3] It is not recommended to replicate this in your organization.



Service level agreements or SLAs may appear to be similar to SLOs, but they are used for very different purposes. An SLA is part of a business contract and it specifies what happens if the target is violated—usually a financial penalty of some kind.

SLAs are not interesting to developers until they are violated. SLOs are extremely useful to developers because they help you understand the overall reliability of applications and to determine if it's safe to invest in new features.

06

Beyond open source

With the open-source tools from the previous chapter, you can gather a lot of telemetry data from services and begin to put it together in a useful way. However, it's difficult to say that you have truly achieved observability.

Observability demands that you can answer questions about unknown unknowns. Your telemetry should be able to adapt and change as your services do. So far, no open-source tool can provide that level of automation. But a solution such as IBM Instana can.

Observability solutions are adaptable and dynamic
Your applications are likely comprised of dozens, hundreds or even thousands of services using different languages and technologies. Maintaining consistent manual instrumentation across the entire application's surface area would be virtually impossible.

By relying on automations such as dynamic service discovery and automatic instrumentation, you can achieve a much more complete understanding of your systems before incidents occur. That's because while you're in the middle of triaging a production outage is not the time to discover that you're missing a key piece of the puzzle.

The automatic correlation benefit
A major benefit of an enterprise observability solution is the ability to correlate machine, infrastructure, and application or services metrics and traces. Distributed traces deliver an understanding of the request's flow, while metrics provide the necessary performance points. Correlating these manually, however, is quite cumbersome. The main reason people hate having multiple dashboards for different services is that it can be almost impossible to match any data and gather the overall context of the issue.

The biggest benefit of automatic correlation is the immediate insight. When looking at an issue or incident, the solution does all the detective work. This provides important pieces of information as contextual evidence and leads you right to the area of interest.

Is IBM Instana right for you?

IBM Instana is an enterprise observability platform that includes automated application performance monitoring capabilities. It’s designed for businesses operating complex, modern, cloud-native applications no matter where they reside—on premises, in public and private clouds, on mobile devices or in an IBM Z® environment.

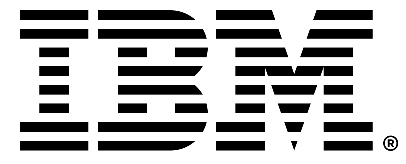
IBM Instana helps you control modern hybrid applications with AI-powered discovery of deep contextual dependencies inside hybrid applications. IBM Instana also provides visibility into development pipelines to help enable closed-loop DevOps automation.

These capabilities provide actionable feedback needed for clients as they optimize application performance, enable innovation and mitigate risk. These features help DevOps increase efficiency and add value to software delivery pipelines so they can meet their service and business-level objectives.

See the power of IBM Instana for yourself. Sign up today for a free 14-day trial of the full version of the product. No credit card required.

IBM Instana free trial →

Explore IBM Instana →



1. R. E. Kalman, “On the general theory of control systems,” August 1960.

2. Twitter blog, “Observability at Twitter,” 9 September 2013, Accessed July 2023.

3. Google online SRE book, “Service Level Objectives,” Accessed July 2023.

© Copyright IBM Corporation 2023

IBM Corporation
New Orchard Road
Armonk, NY 10504

Produced in the United States of America
August 2023

IBM, the IBM logo, IBM Z, and Instana are trademarks or registered trademarks of International Business Machines Corporation, in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on ibm.com/trademark.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.

It is the user’s responsibility to evaluate and verify the operation of any other products or programs with IBM products and programs. THE INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING WITHOUT ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY OR CONDITION OF NON-INFRINGEMENT. IBM products are warranted according to the terms and conditions of the agreements under which they are provided.