New Relic

A Quick Introduction to Distributed Tracing

Gain Visibility and Reduce MTTR in Complex Application Environments

Table of Contents

Introduction: Cutting Through the Complexity	03
What Is Distributed Tracing?	05
Why Does Your Business Need Distributed Tracing?	08
How Does Distributed Tracing Work?	10
When Do You Use Distributed Tracing?	12
Why Is Sampling Important to Understand?	13
A Mini-Glossary of Distributed Tracing Terms	14
What's Next?	15

Introduction: Cutting Through the Complexity

Modern software development drives innovation for companies of all sizes and shapes, enabling them to deliver differentiated customer experiences, accelerate time to market, and gain the agility needed to meet their target outcomes. However, the downside of modern environments and architectures is complexity, making it more difficult to quickly diagnose and resolve performance issues and errors that impact customer experience.

The answer is observability, which cuts through software complexity with end-to-end visibility that enables teams to solve problems faster, work smarter, and create better digital experiences for their customers. Observability creates context and actionable insight by, among other things, combining four essential types of observability data: metrics, events, logs, and traces.

Traces—more precisely, distributed traces—are essential for software teams considering a move to (or already transitioning to) the cloud and adopting microservices. That's because distributed tracing is the best way to quickly understand what happens to requests as they transit through the microservices that make up your distributed applications.

Whether you're a business leader, DevOps engineer, product owner, site reliability engineer, software team leader, or other stakeholder, you can use this ebook to get a quick introduction into what distributed tracing is all about, how it works, and when your teams should be using it.

The shift to modern software development

Waterfall development with infrequent releases	Agile development with continuous deployment
Siloed teams	Cross-functional teams and DevOps
On-premises	Cloud computing
Monolithic applications	Distributed, microservices architectures
Monitoring application performance	Observing end-to-end software and systems performance
Reacting to performance issues	Working to prevent performance issues

What Is Distributed Tracing?

Distributed tracing is now table stakes for operating and monitoring modern application environments. But what is it exactly?

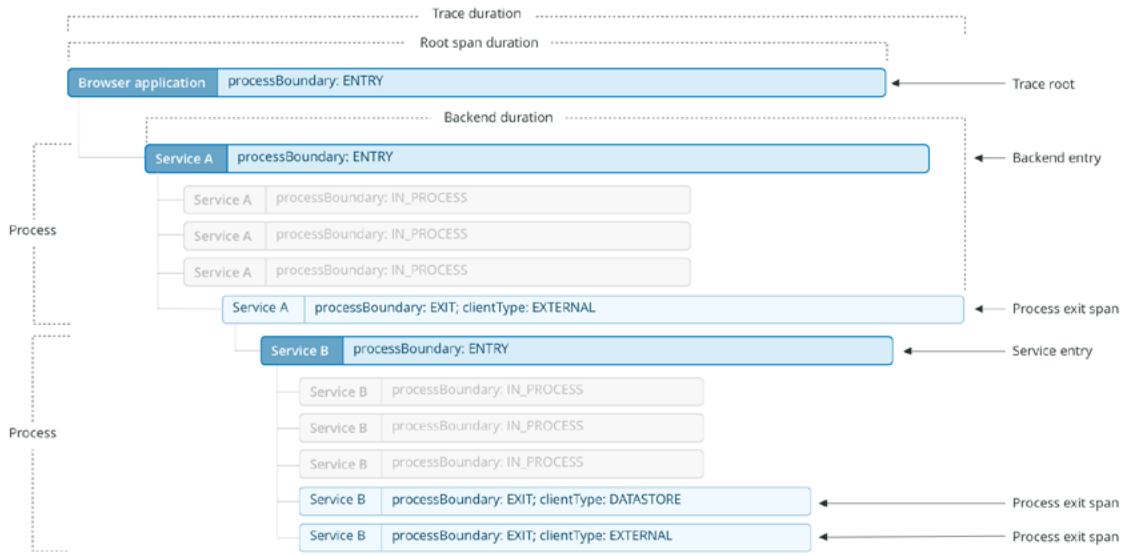
Distributed tracing is the capability for a tracing solution to track and observe service requests as they flow through distributed systems by collecting data as the requests go from one service to another. The trace data helps you understand the flow of requests through your microservices environment and pinpoint where failures or performance issues are occurring in the system—and why.

For instance, a request might pass through multiple services and traverse back and forth through various microservices to reach completion. The microservices or functions could be located in multiple containers, serverless environments, virtual machines, different cloud providers, on-premises, or any combination of these.

Connecting the dots

Combining traces with the other three essential types of telemetry data—metrics, events, and logs (which together with traces create the acronym MELT)—gives you a complete picture of your software environment and performance for end-to-end observability. You can learn more about the different types of telemetry data in "**MELT 101: An introduction to the four essential telemetry data types**."

A distributed trace has a tree-like structure, with "child" spans that refer to one "parent" span. This diagram shows some important span relationships in a trace.



This diagram shows how spans in a different trace relate to one another.

A brief history of distributed tracing

As companies began moving to distributed applications, they quickly realized they needed a way to have not only visibility into individual microservices in isolation but also the entire request flow.

Hence, distributed tracing became a best practice for gaining needed visibility into what was happening. However, software teams discovered that instrumenting systems for tracing then collecting and visualizing the data was labor-intensive and complex to implement. The time and resources spent building code to make distributed tracing work was taking time away from the development of new features.

Then two things happened: First, solutions such as New Relic began offering capabilities that enable companies to quickly and easily instrument applications for tracing, collect tracing data, and analyze and visualize the data with minimal effort. Second, open standards for instrumenting applications and sharing data began to be established, enabling interoperability among different instrumentation and observability tools.

A quick guide to distributed tracing terminology

- A **request** is how applications, microservices, and functions talk to one another.
- A **trace** is performance data about requests as they flow through microservices.
- A **span** represents operations or segments that are part of a trace.
- A **root span** is the first span in a trace.
- A **child span** is a subsequent span, which can be nested.

Why Does Your Business Need Distributed Tracing?

As new technologies and practices—cloud, microservices, containers, serverless functions, DevOps, site reliability engineering (SRE), and more—increase velocity and reduce the friction of getting software from code to production, they also introduce new challenges:

- More points of failure within the application stack
- Increased mean time to resolution (MTTR) due to the complexity of the application environment
- Less time to innovate because more time is needed to diagnose problems

For example, a slow-running request might be impacting the experience of a set of customers. That request is distributed across multiple microservices and serverless functions. Several different teams own and monitor the various services that are involved in the request, and none have reported any performance issues with their microservices. Without a way to view the performance of the entire request across the different services, it's nearly impossible to pinpoint where and why the high latency is occurring and which team should address the issue.

As part of an end-to-end observability strategy, distributed tracing addresses the challenges of modern application environments. By deeply understanding the performance of every service—both upstream and downstream—your software teams can more effectively and quickly:

- Identify and resolve issues to minimize the impact on the customer experience and business outcomes
- Measure overall system health and understand the effect of changes on the customer experience
- Prioritize high-value areas for improvement to optimize digital customer experiences
- Innovate continuously with confidence to outperform the competition

Gaining visibility into a massive data pipeline

Fleet Complete is the fastest-growing telematics provider in the world, serving more than 500,000 subscribers and 35,000 businesses in 17 countries, while experiencing tenfold growth in the past several years. It uses distributed tracing and other telemetry data to gain full visibility into its data-ingestion pipeline, which collects 1 billion data points every day.

> New Relic gave us all the insights we needed—both globally and into the different pieces of our distributed application. [As] we move data across our distributed system, New Relic enables us to see where bottlenecks are occurring as we call from service to service.
>
> **— Muhamad Samji**
> Architect, Fleet Complete

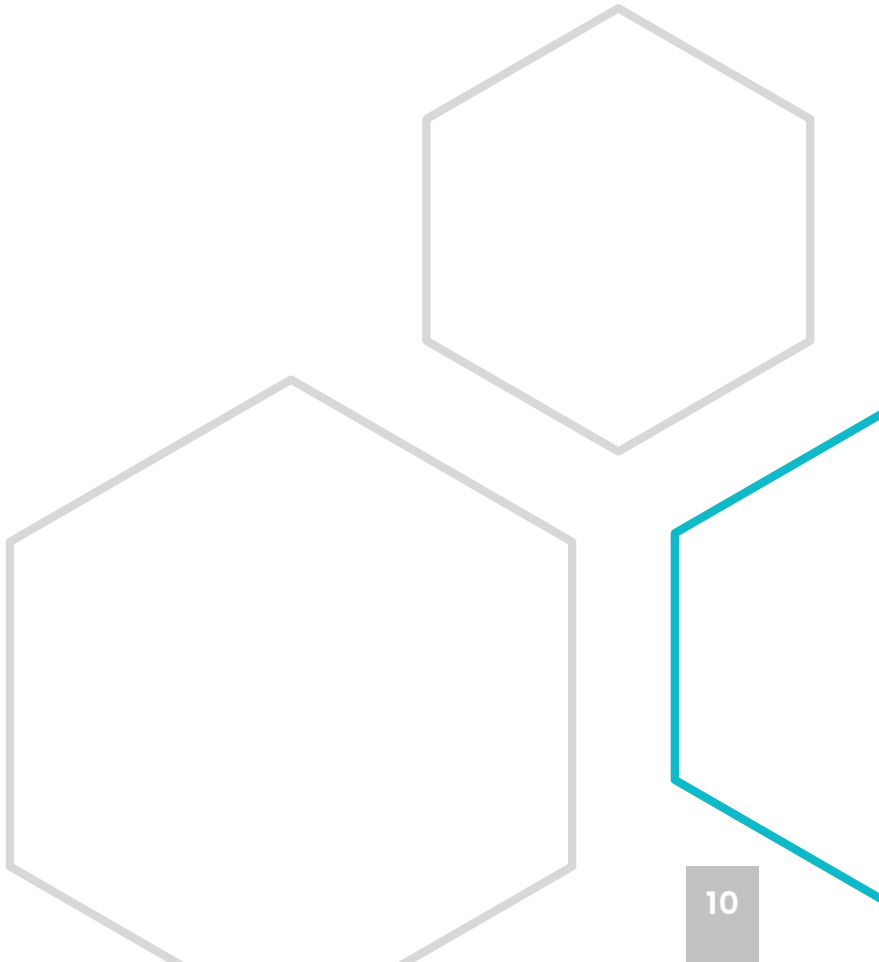
How Does Distributed Tracing Work?

Distributed tracing starts with instrumenting your environment to enable data collection and correlation across the entire distributed system. After the data is collected, correlated, and analyzed, you can visualize it to see service dependencies, performance, and any anomalous events such as errors or unusual latency.

Instrumentation

Instrumenting your microservices environment means adding code to services to monitor and track trace data. Solutions such as New Relic make it easy to instrument your applications for almost any programming language and framework. You can also use open source tools and open instrumentation standards to instrument your environment. **OpenTelemetry**, part of the **Cloud Native Computing Foundation (CNCF)**, is becoming the one standard for open source instrumentation and telemetry collection. Projects such as **OpenCensus** and **Zipkin** are also well established in the open source community. Some service meshes, such as **Istio**, also emit trace telemetry data.

New Relic is fully committed to supporting open standards for distributed tracing, so that your organization can ingest trace data from any source, whether that's open instrumentation or proprietary agents. Learn more about New Relic's support for **OpenTelemetry**, **OpenCensus**, and **Istio**.



Trace context

To make the trace identifiable across all the different components in your applications and systems, distributed tracing requires trace context. This means assigning a unique ID to each request, assigning a unique ID to each step in a trace, encoding this contextual information, and passing (or propagating) the encoded context from one service to the next as the request makes its way through an application environment. This lets your distributed tracing tool correlate each step of a trace, in the correct order, along with other necessary information to monitor and track performance.

W3C Trace Context is becoming the standard for propagating trace context across process boundaries. It lets all tracers and agents that conform to the standard participate in a trace, with trace data propagated from the root service all the way to the terminal service. New Relic supports the W3C Trace Context standard for distributed tracing.

Metrics and metadata

A single trace typically captures data about:

- Spans (service name, operation name, duration, and other metadata)
- Errors
- Duration of important operations within each service (such as internal method calls and functions)
- Custom attributes

Analysis and visualization

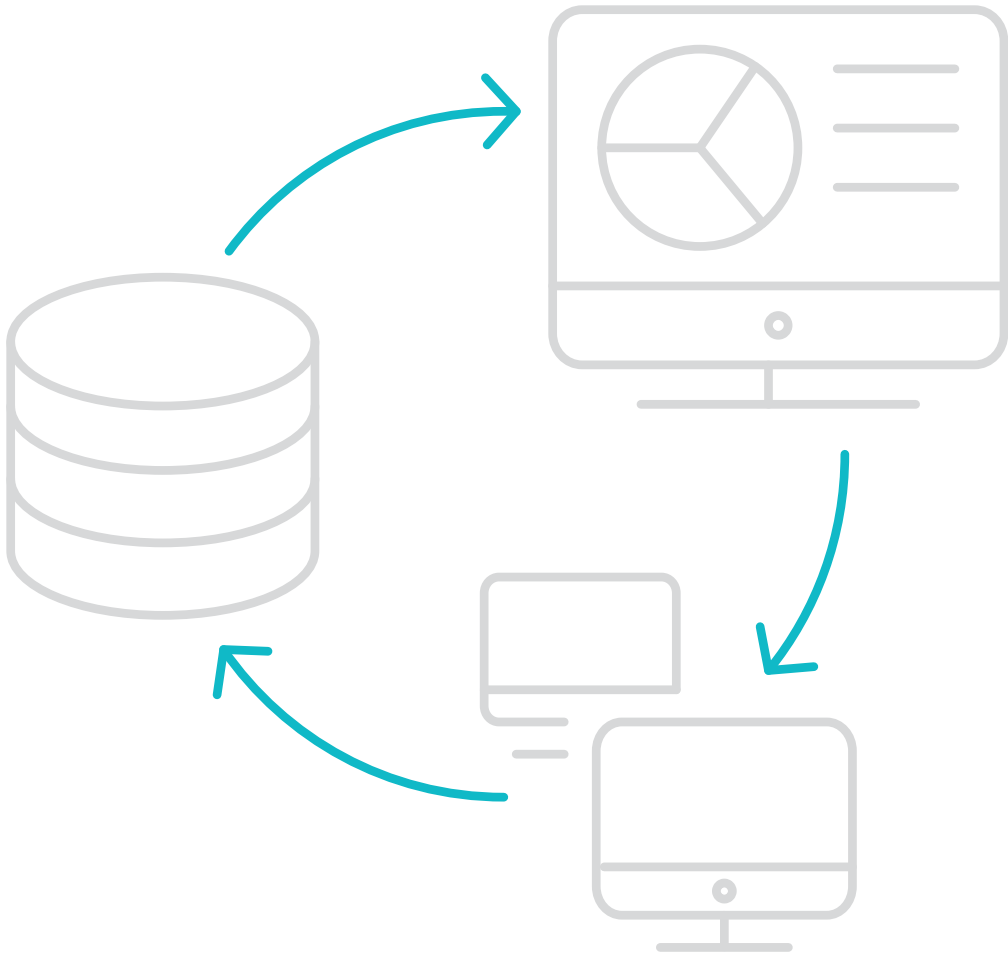
Collecting trace data would be wasted if software teams didn't have an easy way to analyze and visualize the data across complex architectures. A comprehensive observability platform allows your teams to see all of their telemetry and business data in one place. It also provides the context they need to quickly derive meaning and take the right action, and work with the data in ways that are meaningful to you and your business.

When Do You Use Distributed Tracing?

In general, distributed tracing is the best way for DevOps, operations, software, and site reliability engineers to get answers to specific questions quickly in environments where the software is distributed—primarily, microservices and/or serverless architectures. As soon as a handful of microservices are involved in a request, it becomes essential to have a way to see how all the different services are working together.

This means that you should use distributed tracing when you want to get answers to questions such as:

- What is the health of the services that make up a distributed system?
- What is the root cause of errors and defects within a distributed system?
- Where are performance bottlenecks that could impact the customer experience?
- Which services have problematic or inefficient code that should be prioritized for optimization?



Why Is Sampling Important to Understand?

As you can imagine, the volume of trace data can grow exponentially over time as the volume of requests increases and as more microservices are deployed within the environment. To manage the complexity and cost associated with transmitting and storing vast amounts of trace data, organizations can store representative samples of the data for analysis instead of saving all the data.

There are two approaches to sampling distributed traces:

Sampling Type	**Head-based sampling** makes the decision to collect and store trace data randomly while the root (first) span is being processed.	**Tail-based sampling** makes the decision to sample the request when it has completed and all information about that trace has been collected.
Advantages & Use Cases	• Works well for applications with lower throughput • Fast and simple to get up and running • Works well with a blend of monoliths and microservices • Little-to-no impact on application performance • Low-cost solution for sending trace data to third-party vendors • Random sampling can give sufficient visibility for some systems	• Works well for highly distributed, high-volume app environments • Captures and analyzes 100% of traces across a distributed system • Observes every span within a request and then decides which traces are most useful to save • Visualizes the most actionable data with errors, unusual latency, and anomalies • Lets you ask deeper system questions
Considerations	• Traces are sampled randomly • The sampling decision is made before traces have fully completed • Traces with errors or unusual latency might be sampled out and missed	• Usually offered in on-premises distributed tracing solutions, which burdens you with deploying, managing, and scaling complex software • Requires operational effort of planning for usage spikes, resiliency, cost, and scale for on-premises solutions • Results in additional costs for transmitting and storing vast amounts of data for on-premises solutions

A Mini-Glossary of Distributed Tracing Terms

Child span: Subsequent spans after the root span. Child spans can be nested.

Head-based sampling: Where the decision to collect and store trace data is made randomly while the root (first) span is being processed.

Observability: In control theory, observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. Observability involves gathering, visualizing, and analyzing metrics, events, logs, and traces (MELT) to gain a holistic understanding of a system's operation. Observability lets you understand *why* something is wrong, compared with monitoring, which simply tells you *when* something is wrong.

Request: How applications, microservices, and functions talk to one another.

Root span: The first span in a trace.

Sampling: Storing representative samples of tracing data for analysis instead of saving all the data.

Span: The primary building block of a distributed trace, a span represents a call within a request, either to a separate microservice or function. It's a named, timed operation representing a piece of the workflow.

Tail-based sampling: Where the decision to sample is made after the full trace information has been collected.

Trace: The tracking and collecting of data about requests as they flow through microservices as part of an end-to-end distributed system. A trace is made up of one or more spans.

What's Next?

Now that you understand how valuable distributed tracing can be in helping you find issues in complex systems, you might be wondering how you can learn more about getting started. Read the white paper, "**Gain an Edge with Distributed Tracing**."