BASIC OPENTELEMETRY

We expect our websites, apps and online services to load almost instantaneously

Think of the frustration some of us feel when websites take more than two seconds to load.

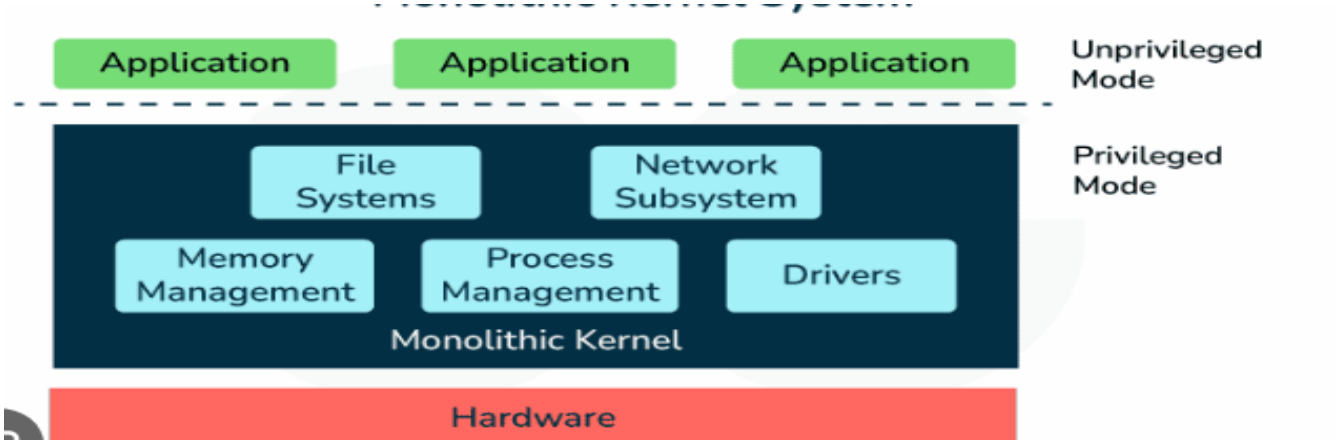
On the backend a feat of engineering is needed to keep a global system running to make sure that we have access to Netflix or Instagram only a tap away

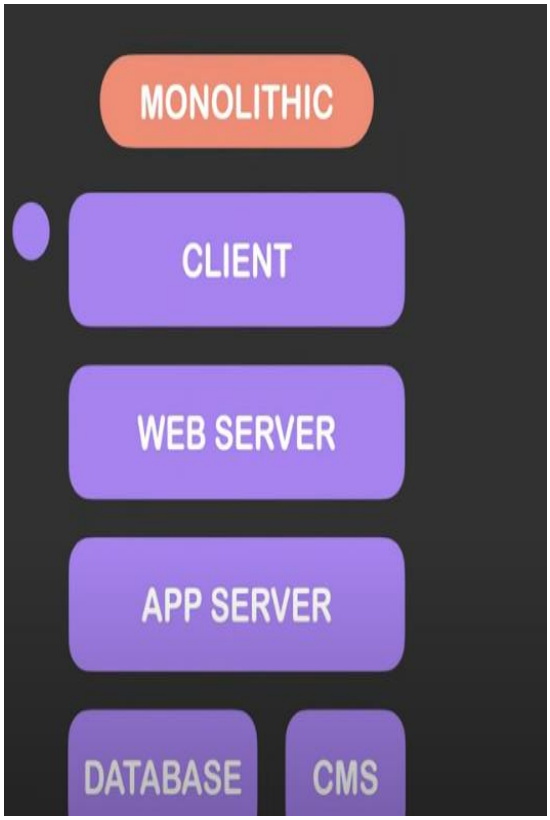
Let's have a look at Netflix for a minute. Netflix at its peak consumes 37% of internet bandwidth in the US, there are 1000s of people clicking play at the same time, with activity peaking in the evening. Though, as a global platform, it's a constant pick. The challenge is how to run a service with zero loss while processing over 400 billion events daily, and 17 gigabyte per second during peak.

WHAT ARE MICROSERVICES?

Historically, developers used monoliths with large complex code bases.

A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications. The word “monolith” is often attributed to something large and glacial, which isn't far from the truth of a monolith architecture for software design.





A single monolith will contain all the code for all the business activities and application performed.

This is all fine and dandy if we have a small app.

But what happens if your application turns out to be successful, users will like it and begin to depend on it, traffic increases dramatically, and always inevitably use requests improvements and additional features.

So, more developers are roped into work on the growing application. Before too long, our application becomes a big ball of mud, a situation where no single developer understands the entirety of the application. Our once simple application has now become larger and complex. Multiple independent development teams are simultaneously.

Working on the same codebase and simultaneously changing the same sections of code,

then it becomes virtually impossible to know who is working on what and then

code quality suffers, it becomes harder and harder for individual development teams to make changes

without having to calculate what the impact will be to other teams and teams can lose sight of how that code might be incompatible with others, among other issues.

This generally results in slower, less reliable applications, and not to mention longer and longer development schedules.

HERE COMES MICROSERVICES.

The main principle behind a microservice architecture is that the applications are simpler to build and maintain when broken down into smaller pieces. When using microservices, we isolate software functionality into multiple independent modules that are individually responsible for performing precisely defined standalone tasks.

Microservice architectures let us split applications into distinct independent services, each managed by different teams.

It leads to naturally delegating the responsibilities for building highly scaled applications, allowing work to be done independently on individual services without impacting the work of other developers in other groups working on the same overall application.

Observability means how well we can understand what is going on internally in a system based on its outputs.

Especially as systems become more distributed and complex. It's hard to see what's going on inside your application and why things may be going wrong.

M.E.L.T

When talking about observability, we need to define the data types necessary to understand the performance and health of our application.

M: Metrics

Metrics are measurements collected at regular intervals and must have a timestamp and name, one or more numeric values and a count of how many events are represented. These include error rate, response time or output.

E: Events

An event is a discrete action happening at any moment in time. Take a vending machine, for instance, an event could be the moment when a user makes a purchase from the machine.

Adding metadata to events makes them much more powerful. With the vending machine example, we could add additional attributes such as item category, and payment type. This

allows questions to be asked such as how much money was made from each item category, or what is the most common payment type use.

L: Logs

Logs come directly from our app, exporting detailed data and detailed context around an event. So, engineers can recreate what happened millisecond by millisecond.

We all have probably logged something when you use things like system out print, or console log.

T: Traces

Traces follow a request from the initial request to the returned output.

It requires the casual chain of events to determine relationships between different entities. Traces are very valuable for highlighting inefficiencies, bottlenecks and roadblocks in the user experience as they can be used to show the end-to-end latency of individual cores and a distributed architecture.

However, getting that data is very difficult. We would have to manually instrument every single service one by one layer by layer. This will take as much time as writing the code itself, which is annoying. Luckily there are some awesome open-source projects as well as companies that make this a lot easier.

HISTORY

In 2016, open tracing was released as a CNCF project focused only on distributed tracing.

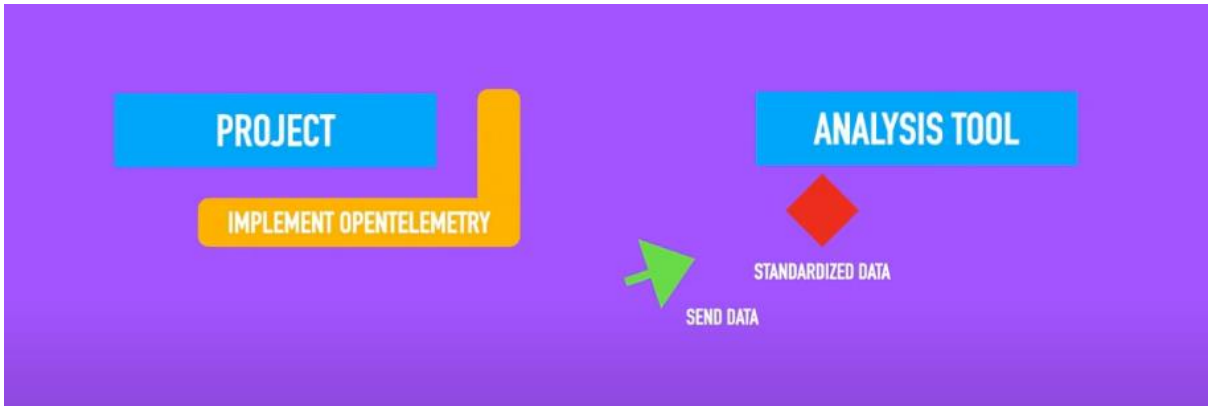
Because the libraries were lightweight and simple, it could be used to fit any use case. While it made it easy to instrument data, it made it hard to instrument software that was shipped as binaries without a lot of manual engineering work. In 2018, a similar project called Open census was open source out of Google, which supported both the capturing retracing permission and metrics.

While it made it easier to get telemetry data from software that was shipped as binaries like Kubernetes, and databases, it made it hard to use the API to instrument custom implementations, not part of the default use case. Both projects were able to make observability easy for modern applications and expedite wide adoption of distributed tracing by the software industry.

However, developers had to choose between two options with pros and cons.

It turns out that the approaches of the two projects were complimentary rather than contradictory. There was a no reason why we couldn't have both the abstract vendor neutral API and a well-supported default implementation.

In late 2019, the two projects merged to form open telemetry. This brought forward the idea of having a single standard for observability instead of two competing standards.



Let's say a project is built to listen out for requests. So, like listening out for a get request, for example, we have decided that we want to measure our app's performance based on the requests that it makes. To do this, the first step as we mentioned at the start, would be to implement open telemetry into the project.

We are doing this to help us standardize the data. Once we have implemented open telemetry and standardize the data, we need to think about what we are going to do with the data, how we're going to view it, and so on.

For this, we can use an analysis tool.

By analysis tool, I mean, any type of tool that gives us observability, we have a few of these tools, one that focuses specifically on tracing, one that focuses specifically on metrics, and one that looks at everything in one platform.

We are then going to send our data to our analysis tool of choice.

TRACING

Open telemetry allows us to essentially standardize our data.

The next part is viewing the data in a way that we can analyse what is happening behind the scenes, we will do this with a tracing system. In software engineering tracing involves a specialized use of logging to record information about a program's execution.

This information is typically used by programmers to debug by using the information contained in a trace log to diagnose any problems that might arise with a particular software or app.

A distributed request tracing is a method used to debug and monitor applications built using a micro service architecture.

Distributed tracing helps pinpoint where failures occur, and what causes poor performance.

Tracing data in Open Telemetry is a very important part.

SPANS

A span represents a unit of work or operation. Spans are the building blocks of Traces.

CONTEXT & PROPAGATION

These two concepts will allow us to understand the topic of tracing a lot better.

We know distributed tracing allows us to correlate events across service boundaries. But how do we find these correlations? For this, components in our distributed system need to be able to collect, store and transfer metadata. We refer to this metadata as context.

Context is divided into two types, span context and correlation context.

Span context represents the data required for moving trace information across boundaries.

It contains the following metadata. We have a trace ID, a span ID, the trace flags and trace state.

A correlation contacts carries user defined properties. This is usually things such as a customer ID, providers, host name, data region and other telemetry that gives us application specific performance insights.

Correlation context is not required, and components may choose to not carry or store this information.

A context will usually have information so we can identify the current span and trace .

Propagation is the mechanism we use to bundle up our context and transfer across services.

Setting up our Tracing.

```
const { LogLevel } = require("@opentelemetry/core")
const { ZipkinExporter } = require("@opentelemetry/exporter-zipkin")
const { NodeTracerProvider } = require("@opentelemetry/node")
const { SimpleSpanProcessor } = require("@opentelemetry/tracing")

const provider = new NodeTracerProvider({
  logLevel: LogLevel.ERROR
})

provider.register()

provider.addSpanProcessor(
  new SimpleSpanProcessor(
    new ZipkinExporter({
      serviceName: 'getting-started'
    })
  )
)
```

Provider is our tracer.

Provider.register means that we are registering our tracer.

And provider.addSpanProcessor is our Tracer Exporter.

METRICS

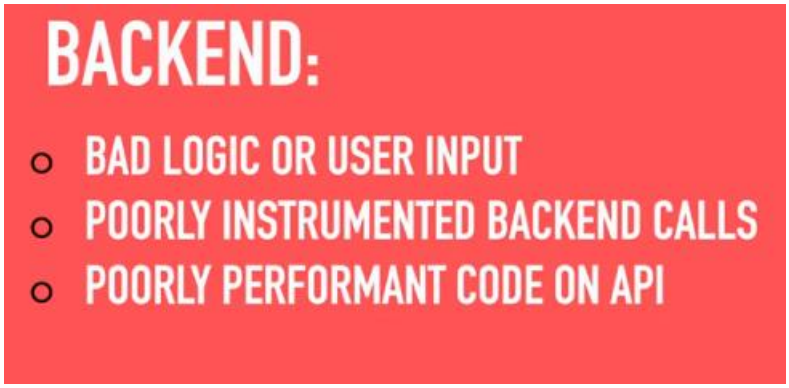
Unlike tracing which works in spans metrics are a numeric representation of data measured over intervals of time. Metrics can harness the power of mathematical modeling and prediction to derive knowledge of the behavior of a system over intervals of time in the present and future.

Since numbers are optimized for storage, processing, compression and retrieval, metrics enable longer retention of data as well as easier querying. This makes metrics perfectly suited

to building dashboards that reflect historical trends. metrics also allow for gradual reduction of data resolution. After a certain period, data can be aggregated into daily or weekly frequency.

USE CASES OF OPEN TELEMETRY

For BACKEND:



In the backend with open telemetry, we can pick up bad logic or user input leading to exceptions being thrown, poorly implemented downstream calls.

So, for example to infrastructures like databases or downstream API's, leading to exceptionally long response times.

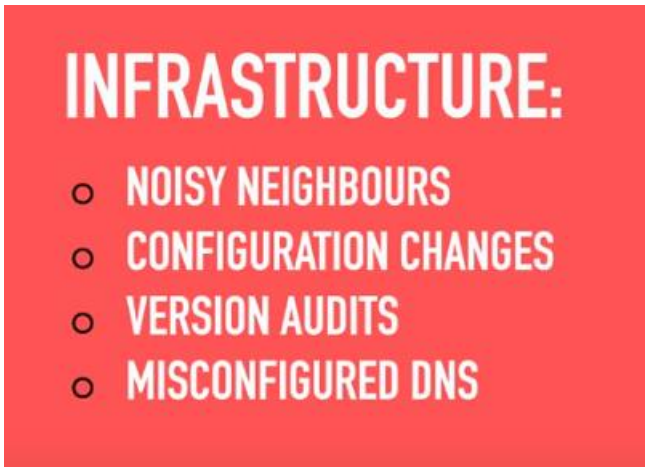
Or we can pick up poorly performing code on a single API, leading to exceptional response times.

For FRONTEND:

On the front end, with open telemetry, we can detect bad logic or user input leading to JavaScript errors. We can also use it to find poorly implemented JavaScript, making our UI prohibitively slow, despite performant API's.

And we can even use it to locate geo specific slowness requiring geo distribution.

For INFRASTRUCTURE:



For infrastructure, we can use it to identify noisy neighbours running on a host sapping resource from other apps, configuration changes, leading to performance degeneration version audit.

So, zero-day vulnerability checks, ensuring convict changes went through, or just miss figuration with your DNS making your apps inaccessible.