

The Phantom Pathways of a Distributed Trace

Dave McAllister

NGINX



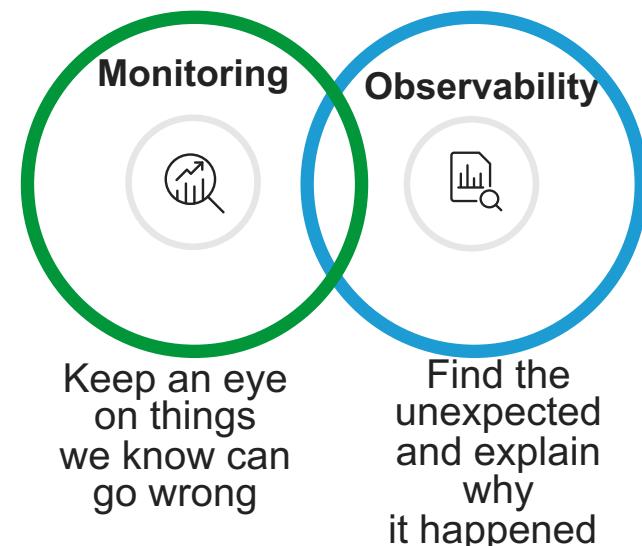
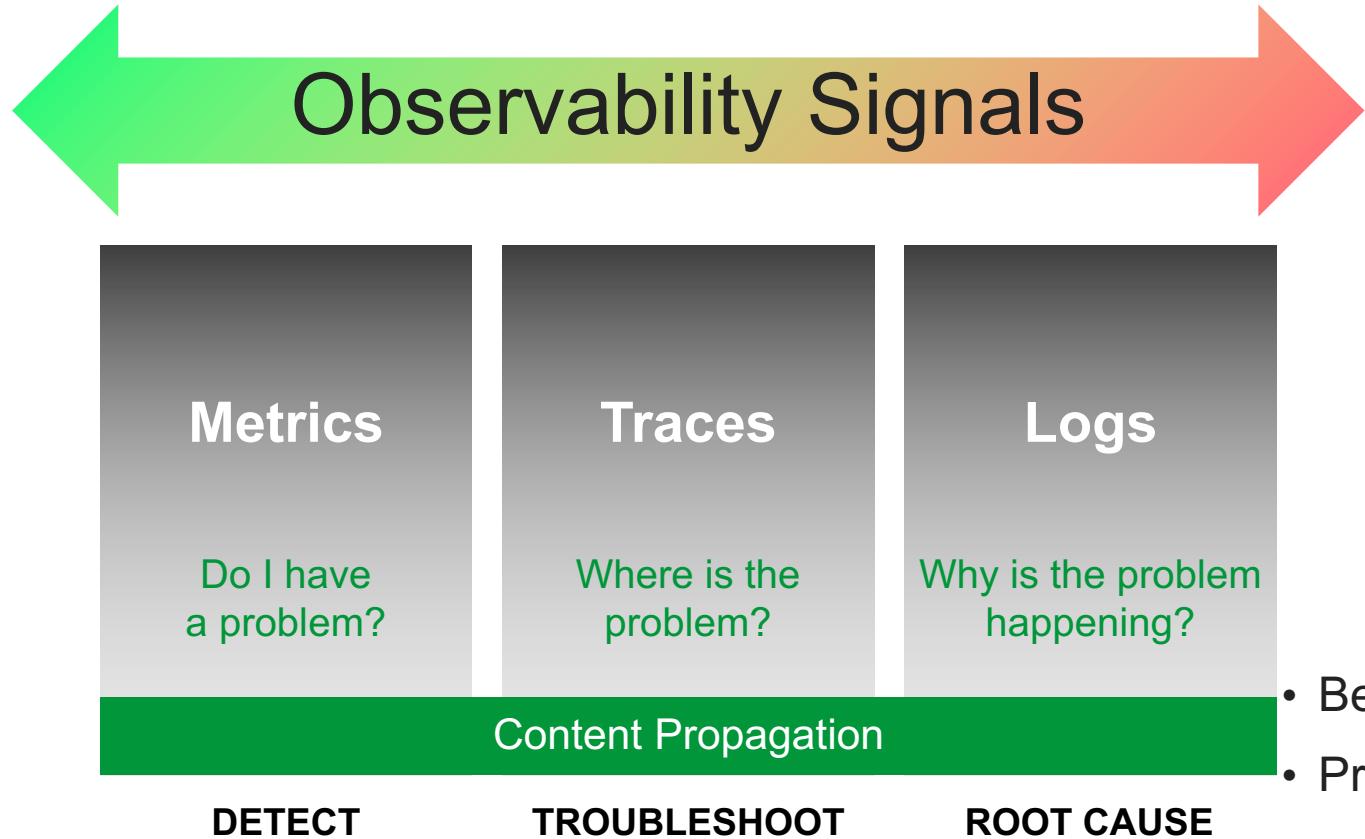
Observability is a scary data problem



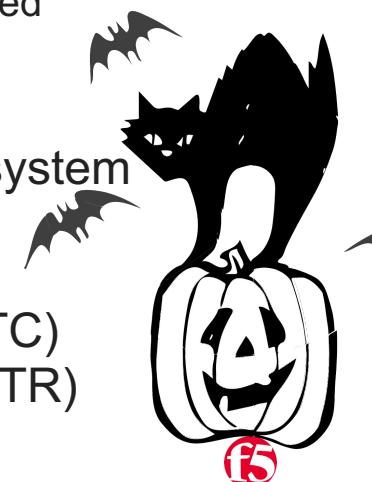
The more observable a system, the quicker we can understand why it's acting up and fix it

Observability

Observability helps detect, investigate and resolve the *unknown unknowns* – FAST



- Better visibility to the state of the system
- Precise and predictive alerting
- Reduces Mean Time to Clue (MTTC) and Mean Time to Resolution (MTTR)

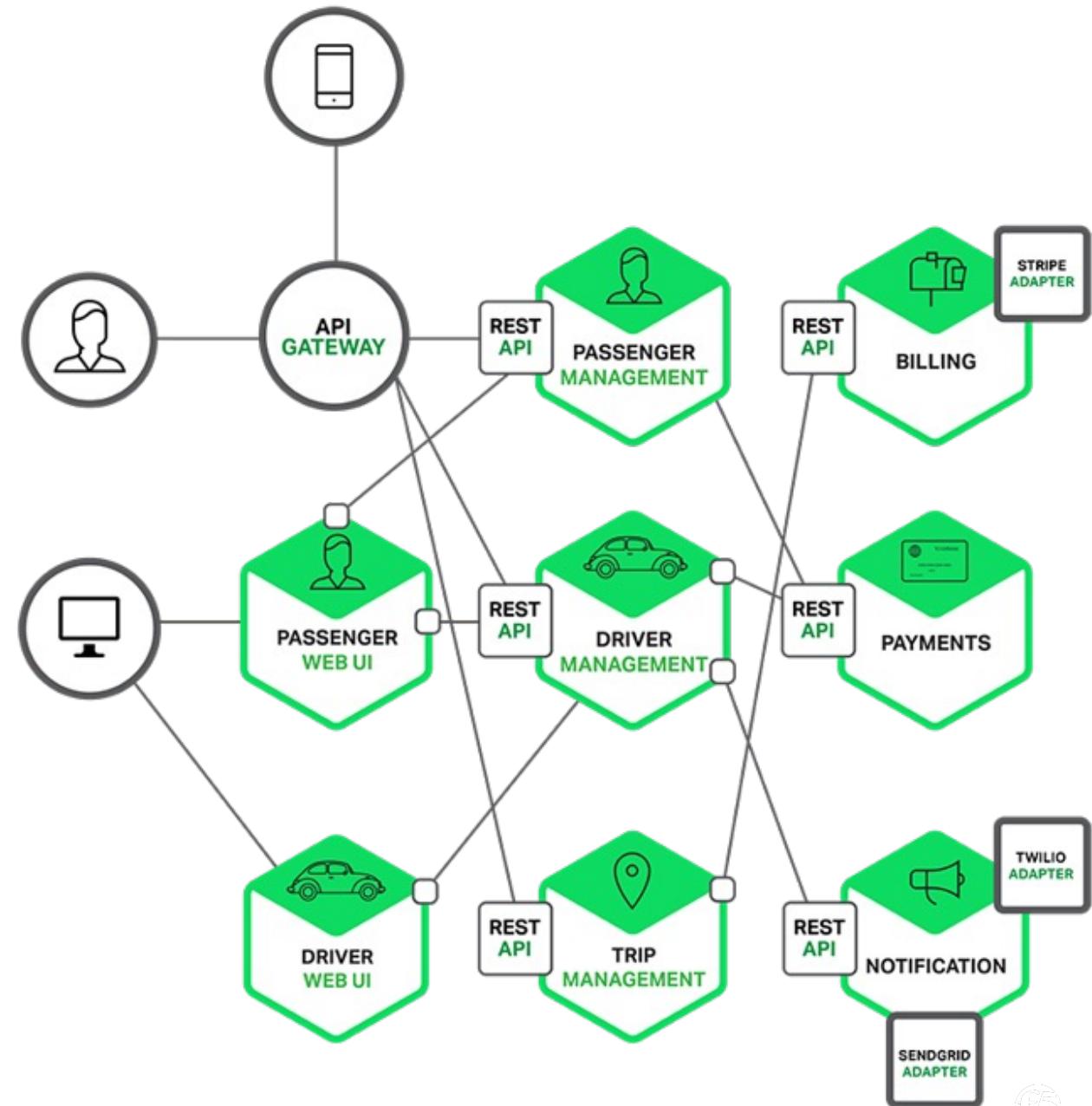


So why observability?

Microservices!

Single application composed of many loosely coupled and independently deployable smaller services

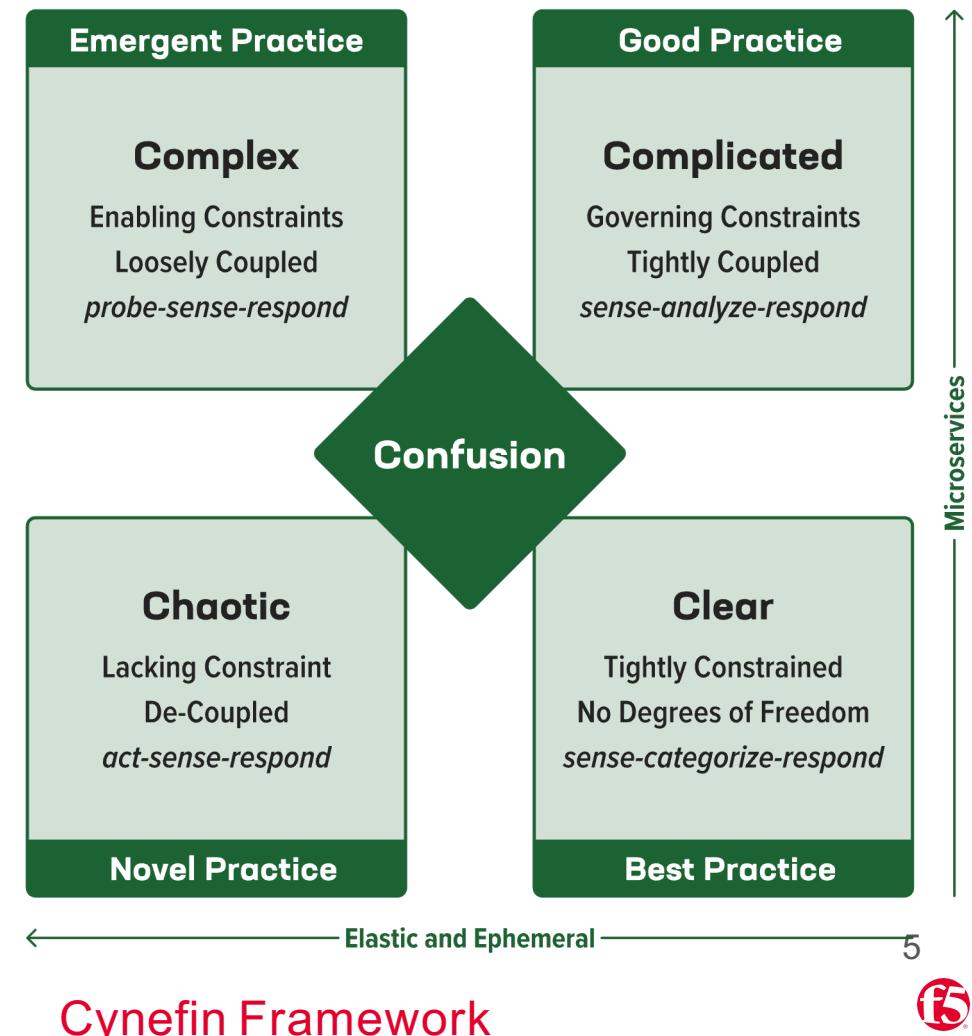
- Often polyglot in nature
- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Often in Cloud environments
- Organized around business capabilities
- Each potentially owned by a small team



μServices add challenges

Especially when we consider this in a cloud

- Microservices create complex interactions.
- Failures don't exactly repeat.
- Debugging multitenancy is painful.
- So much data!

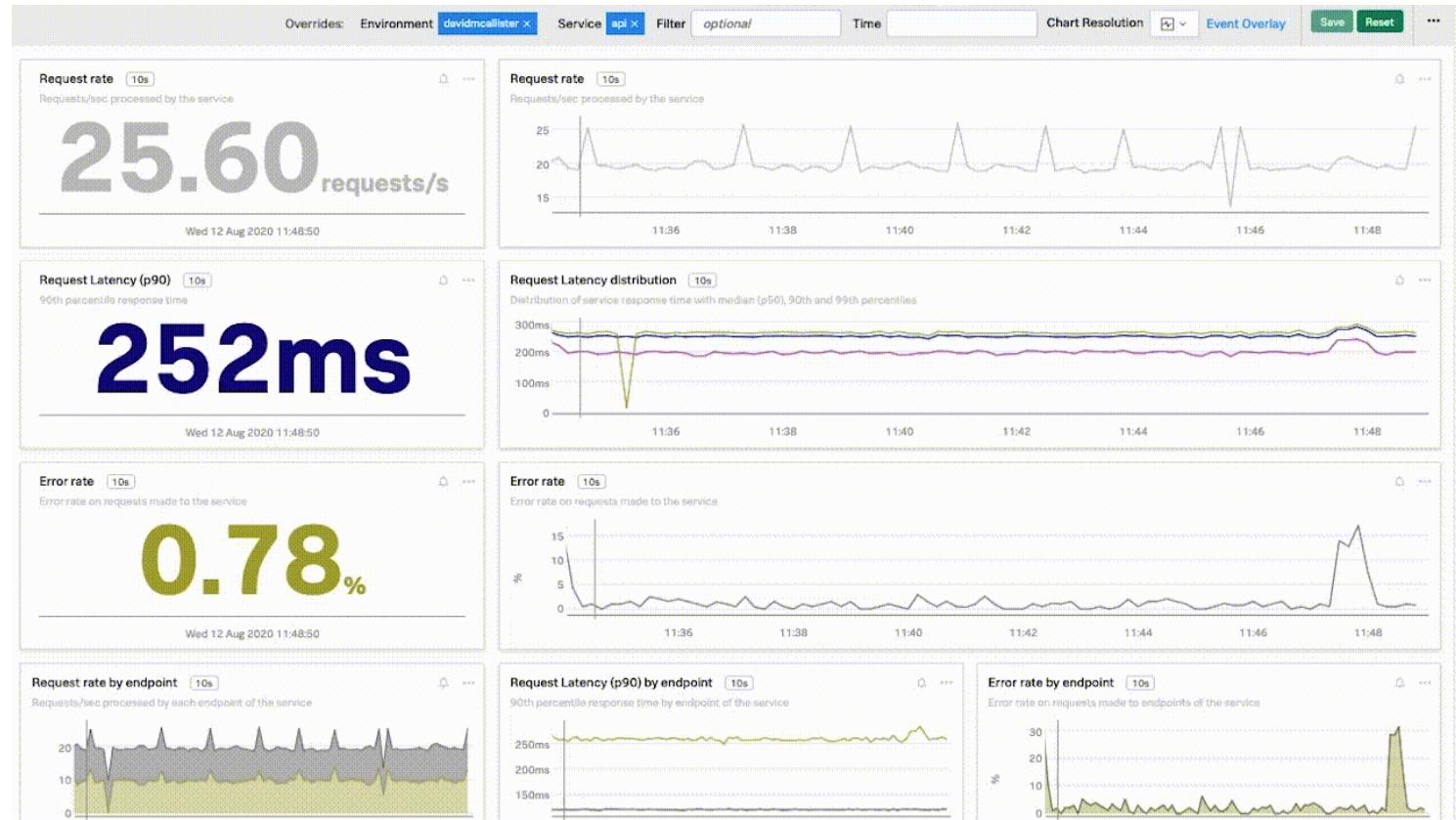


Tracing is a data problem



So what's distributed tracing good for?

- Tracks requests
- Provides actionable insights into app/user experiences
- Defines additional metrics for alerting, debugging
- Rapid MTTC, MTTR



Different models driven by observability signals

RUM, Synthetics, NPM, APM, Infrastructure

Get the most out of APM

See application dependencies, isolate issues and optimize performance with automatically generated service maps, multi-dimensional tag analysis, and full-fidelity trace search.

[Documentation](#) [Developer Docs](#)

The dashboard includes sections for Troubleshooting with APM, Auto Instrument your Java App, and Analyze service performance. It features charts for Top Services by Error Rate, Top Business Workflows by Error Rate, and Top Business Workflows by Duration (P90). The interface also includes tabs for Services and Business Workflows, and a sidebar with links to Explore, Tag Spotlight, Traces, Database Query Performance, and AlwaysOn Profiling.

-8d Environment: jshaw

APM Overview

Top Services by Error Rate

Service	Error Rate
checkout	1.01%
authorization	1%
api	1%
lap	1%

4:49 PM 02/08/2023 6:57 PM TODAY

Top Business Workflows by Error Rate

Workflow	Error Rate
api:/checkout	2.03%

4:49 PM 02/08/2023 6:53 PM TODAY

Top Services by Latency (P90)

Service	Latency (ms)
api	257ms
checkout	208ms
mySql	97ms
catalog	70ms
payment	60ms

4:49 PM 02/08/2023 6:57 PM TODAY

Top Business Workflows by Duration (P90)

Workflow	Duration (ms)
api:/checkout	264ms
api:/catalog	124ms

4:49 PM 02/08/2023 6:53 PM TODAY

Services **Business Workflows** Sort Active alerts, then name Search

NAME **LATENCY** **REQUESTS & ERRORS**

Service	Latency (ms)	Requests (1/s)	Errors (1/s)
api:/catalog	124ms	10/s	0/s
api:/checkout	270ms	2.88/s	0.06/s

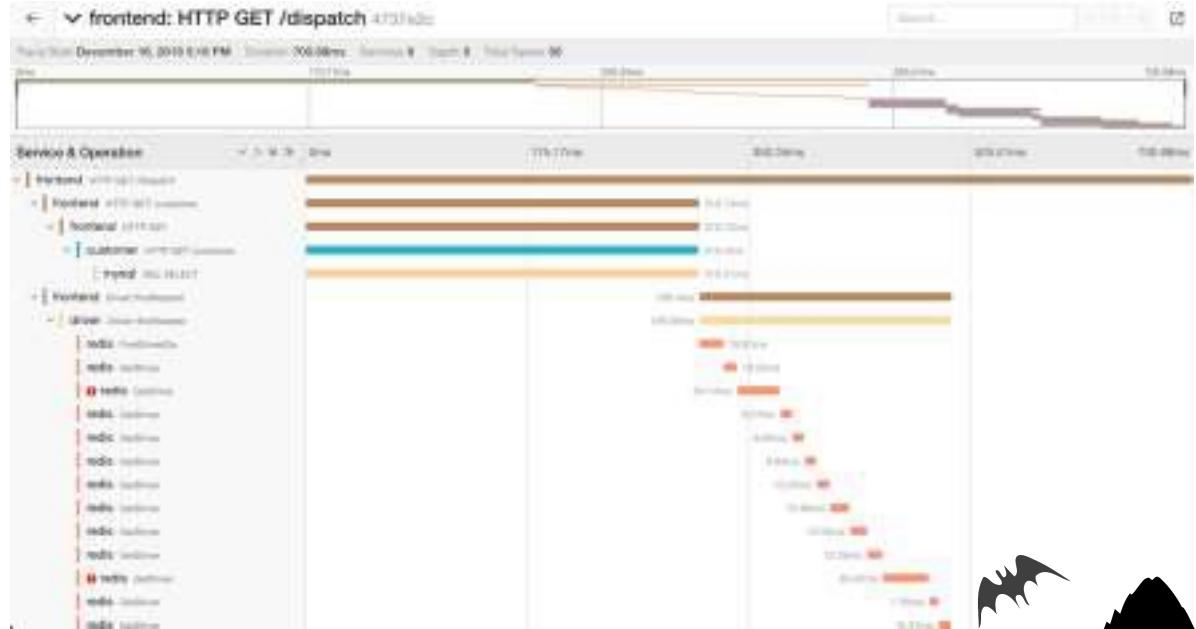
4:49 PM 02/08/2023 6:53 PM TODAY

Explore **Tag Spotlight** **Traces** **Database Query Performance** **AlwaysOn Profiling**

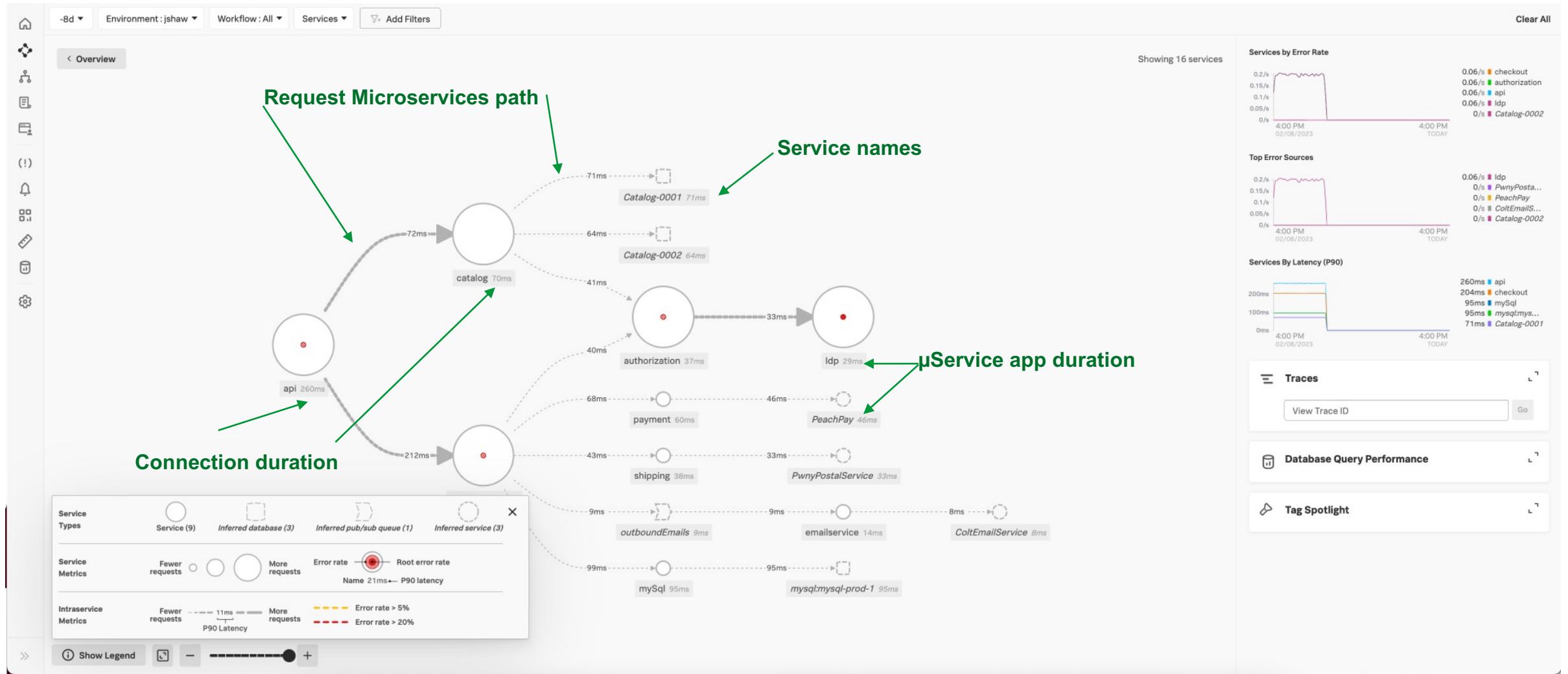
©2022 F5

Tracing concepts

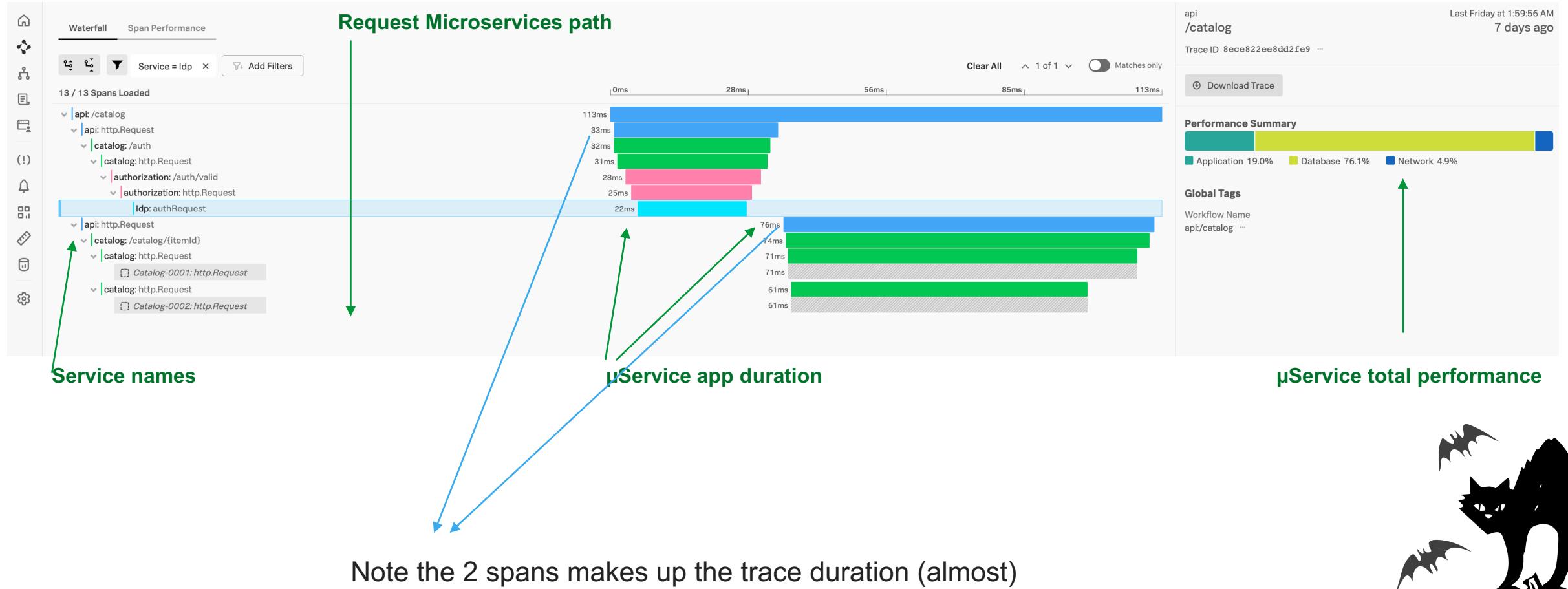
- Span
 - Represents a **single unit of work** in a system
- Trace
 - Defined implicitly by its spans
- Distributed Context
 - Tracing identifiers
 - Tags
 - Options that are propagated from parent to child spans



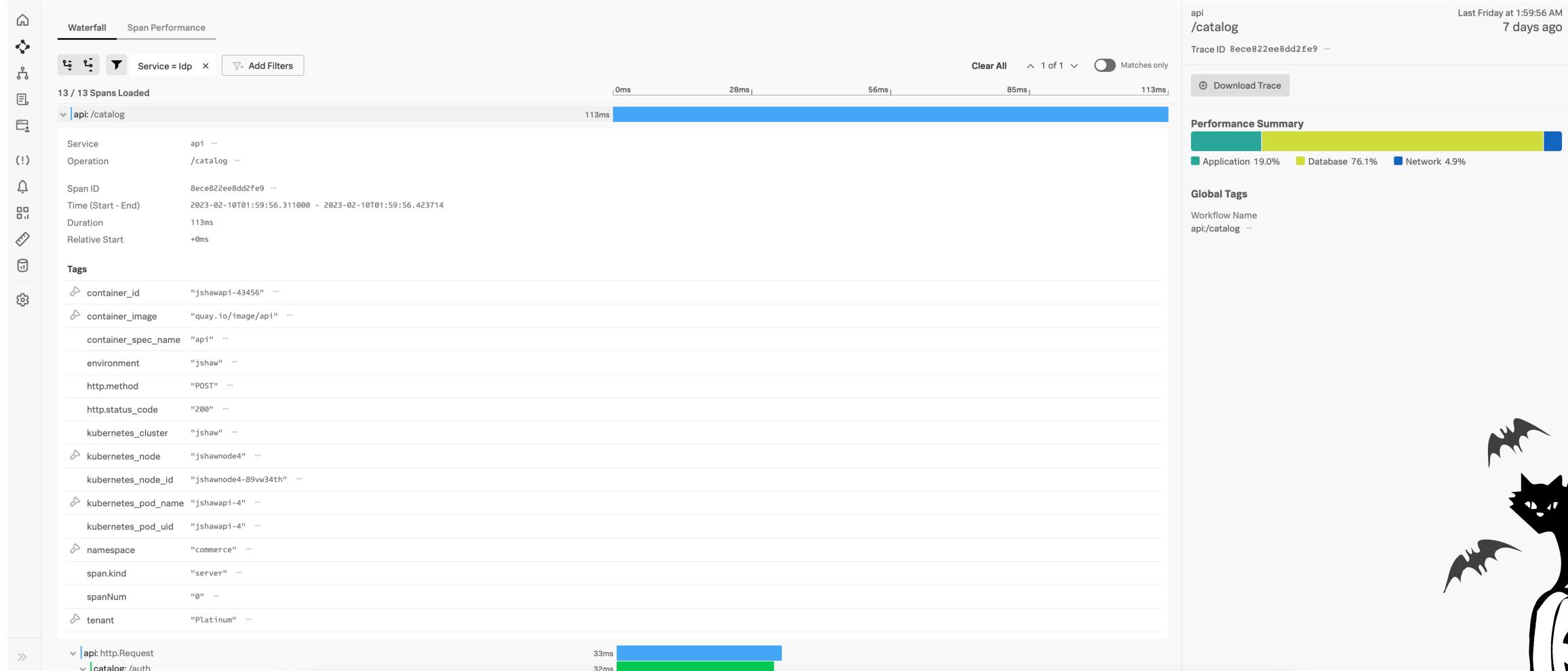
Let's look at a trace



A different way of looking at a trace



Observability includes baggage



What do we need for tracing?

- Generated unique IDs
- Auto propagation
- Telemetry consolidation
- Distributed environment capable
- Standards-based agents, cloud-integration
- Automated code instrumentation
- Support for developer frameworks
- Any code, any time



Tracing API concepts - OpenTelemetry

- **TracerProvider** is the entry point of the API. It provides access to Tracers.
 - Stateful object holding configuration with a global provider and possibly additional ones.
- **Tracer** is the class responsible for creating Spans.
 - Named and optionally versioned with each instrumentation library using values guaranteed to be globally-unique.
 - Delegates getting active Span and marking a given Span as active to the **Context**.
- **Span** is the API to trace an operation.
 - Immutable **SpanContext** represents the serialized and propagated portion of a Span.

Enabling distributed tracing

- Two basic options
 - Traffic Inspection (e.g., service mesh with context propagation)
 - Code Instrumentation with context propagation
- Focusing on Code
 - Add a client library dependency
 - Focus on instrumenting all service-to-service communication
 - Enhance spans (key value pairs, logs)
 - Add additional instrumentation (integrations, function-level, async calls)

What this basically means

Traces

1. Instantiate a tracer
2. Create spans
3. Enhance spans
4. Configure SDK

- Automatic
 - Just add the appropriate files to the app.
- This is language dependent

- Manual
 - Import the OTel API and SDK
 - Configure the API
 - Configure the SDK
 - Create your traces
 - Create your metrics
 - Export your data



What problems are you trying to solve?

A tiny subset of answers*

- Performance issues
- Mean time to resolution and detection is too high
- Metrics and logs are missing valuable context
- More data types to provide better answers



A span for everything or bare minimum

It depends

Remember why you want to trace

Start with service boundaries and 3rd party calls

Iterative process

Beware of information overload

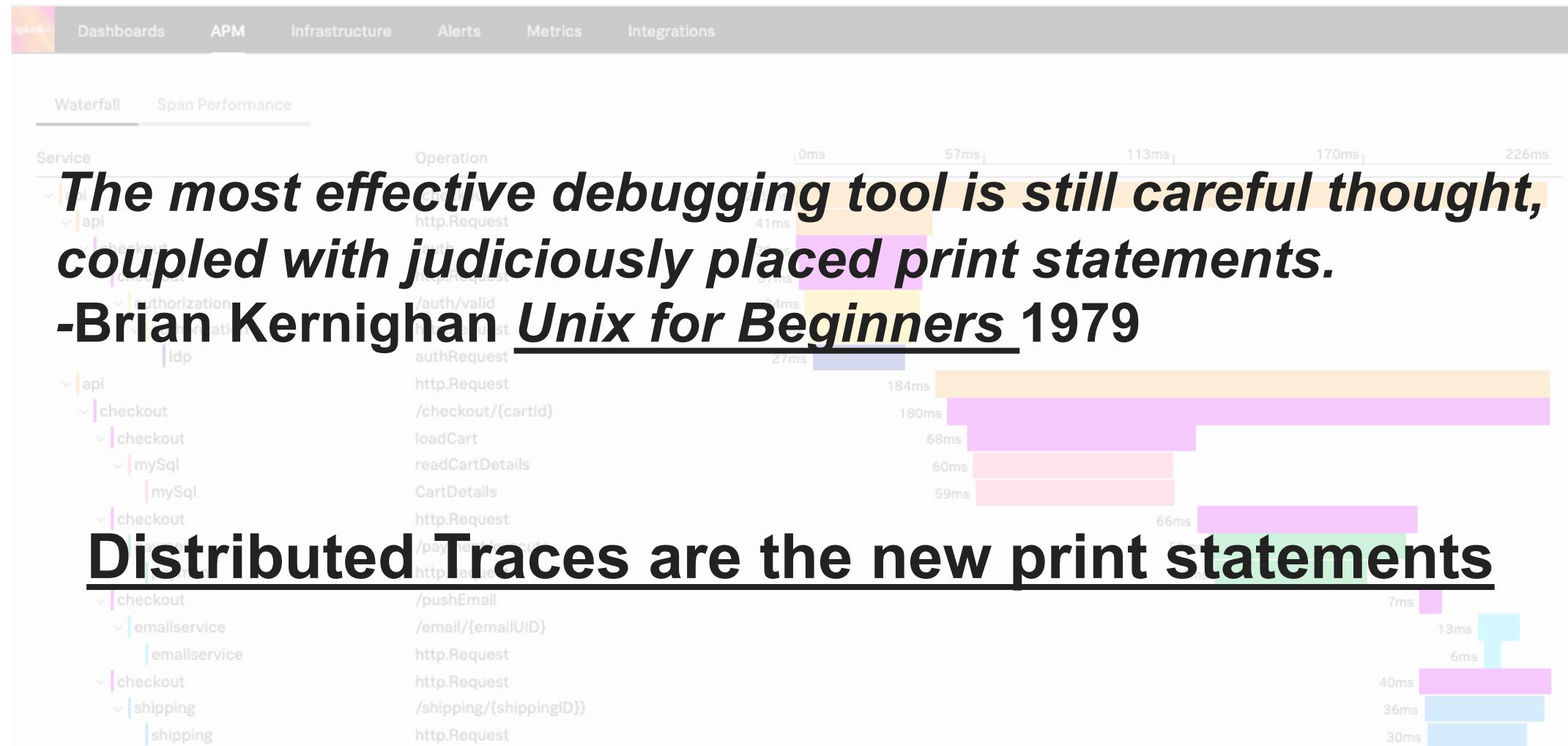


Distributed Tracing Summary

- Gives insights into the app and its infrastructure
- Requires effort to return value
 - The “win” is when the project is already Otel instrumented
- Is a good “User Happiness” proxy
- Does not magically solve your issues



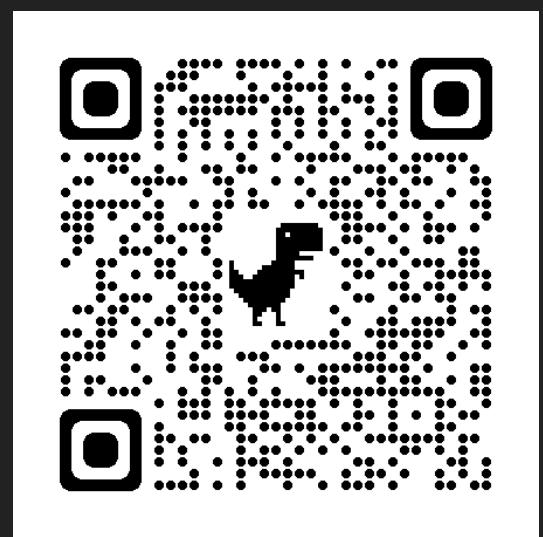
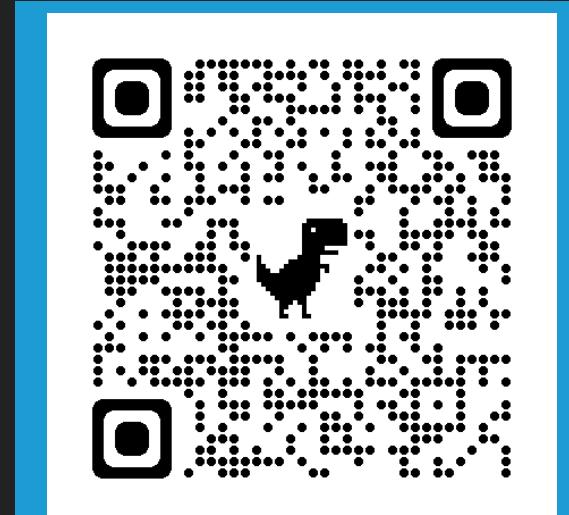
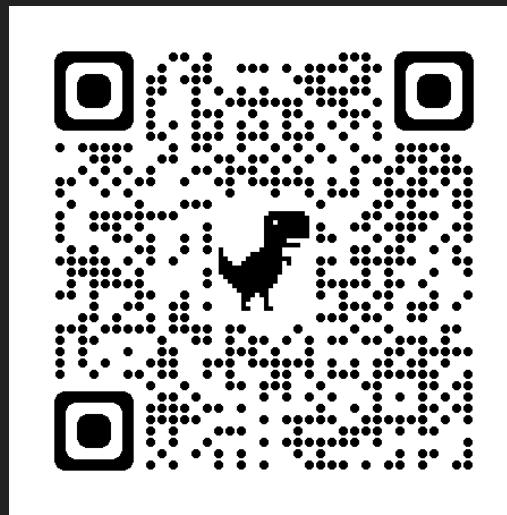
Closing Thought



Thanks!



[Slides on GitHub](#)



[LinkedIn: in/davemc](#)

[NGINX Community Slack](#)

THANKS!