



# What Java Devs Need To Know About Observability

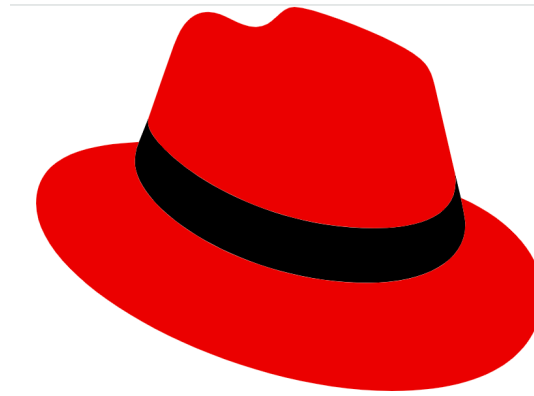
July 20 2022

---

Ben Evans (He / Him)  
Senior Principal Software Engineer  
[beevans@redhat.com](mailto:beevans@redhat.com)

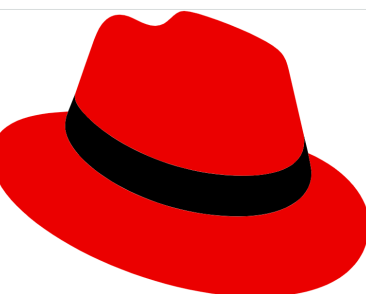
# About Me - Career

- Red Hat, SPSE
- New Relic, Lead Architect
- jClarity, Co-founder (acq MSFT)
- Deutsche Bank
  - Chief Architect (Listed Derivatives)
- Morgan Stanley
  - Google IPO
- Sporting Bet, Chief Architect



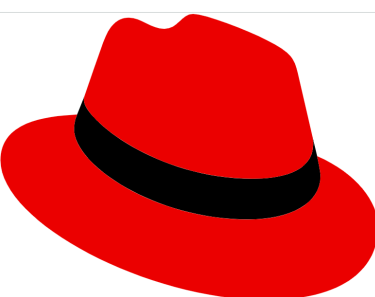
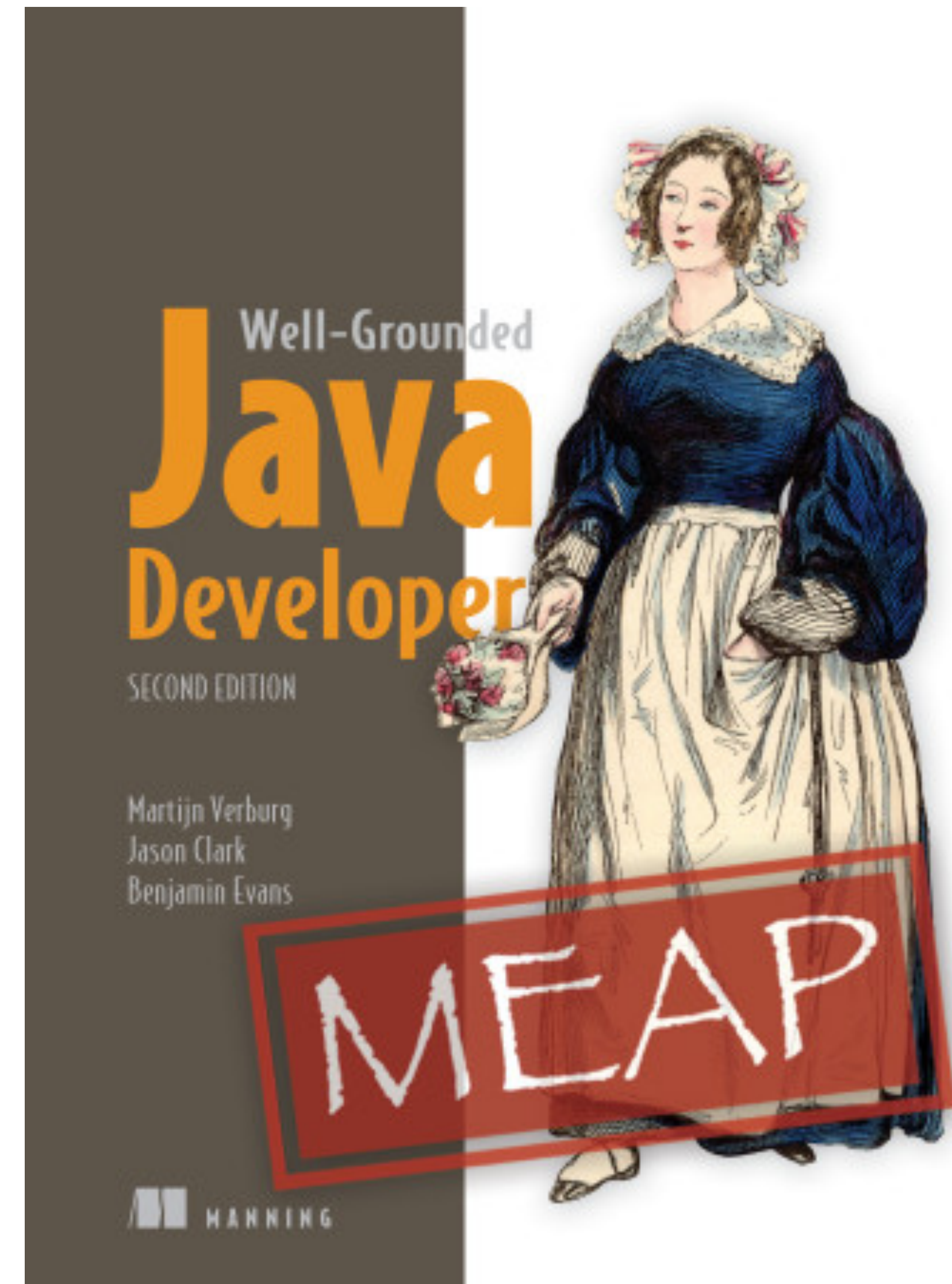
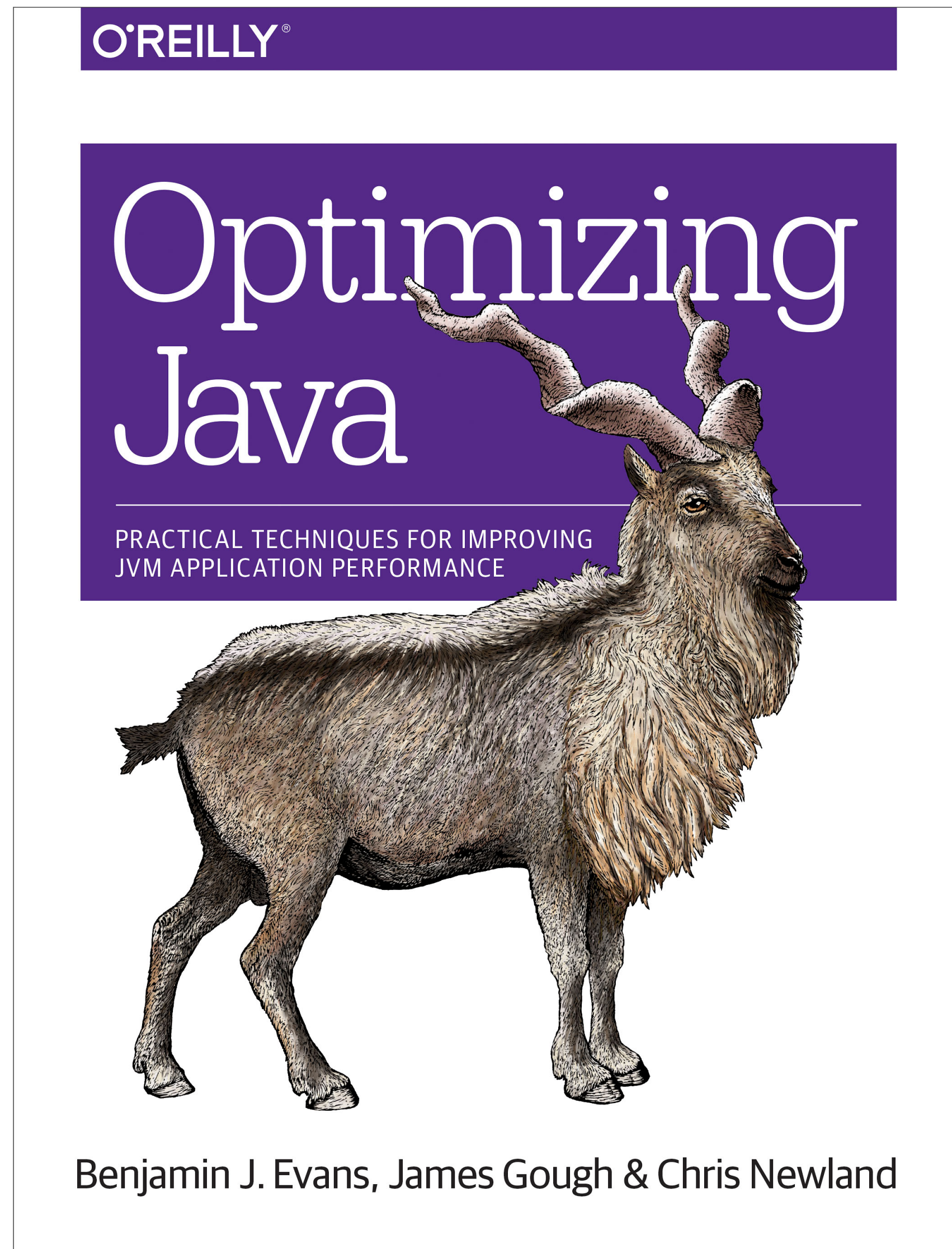
# About Me - Community

- Java Champion
- JavaOne Rock Star Speaker
- Java Community Process Executive Committee
- London Java Community
  - Organising Team
  - Co-founder, AdoptOpenJDK





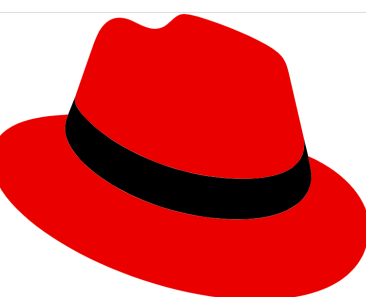
# Recent Books





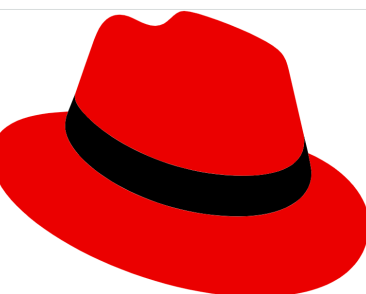
# Installing Java & Demos

- We will use Adoptium JDK - <https://adoptium.net/>
- OpenJDK / Temurin 17 (LTS) with HotSpot JVM
- Repos for Demos
  - <https://github.com/kittylyst/OTel>
  - <https://github.com/kittylyst/observability-java-devs>
  - <https://github.com/newrelic/newrelic-opentelemetry-examples>
  - <https://quarkus.io/guides/opentelemetry>
- My website: <https://kittylyst.com>



# Agenda

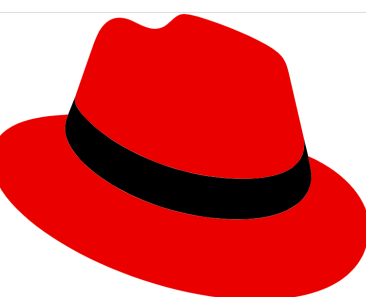
- Observability Concepts
- Introducing OpenTelemetry
- Java & OpenTelemetry
- Conclusions & The Future





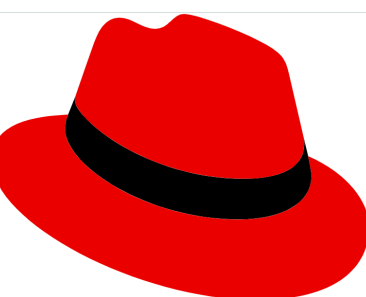
# Observability Concepts

- What is Observability?
- How Do We Understand Cloud Native Apps?
- What Data Do We Need To Collect?
- What Architectural Implications Are There?



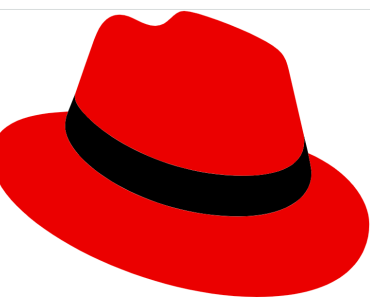
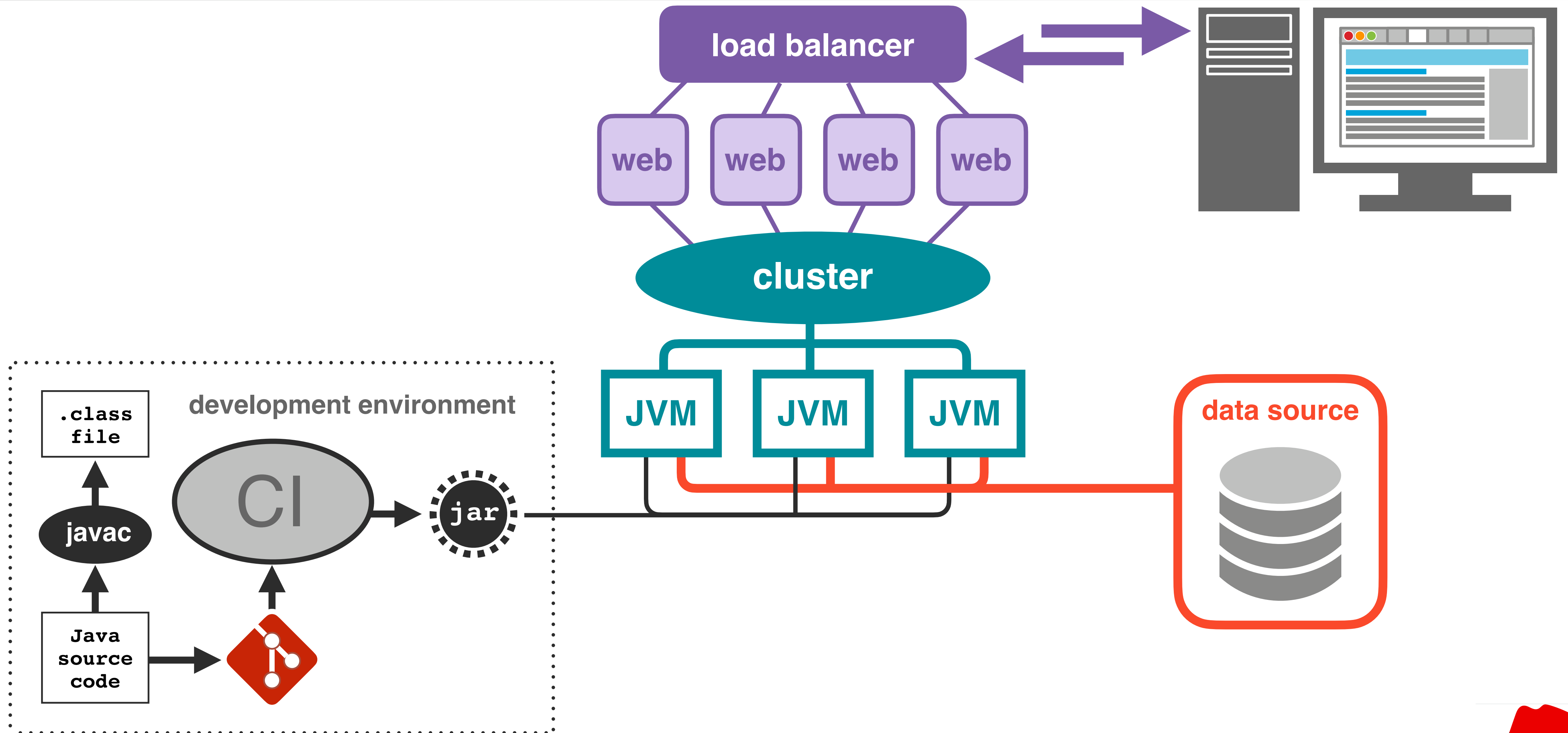
# History of APM

- Application Performance Monitoring (APM) has ~15 year history
- Different world
  - Release cycles measured in months, not days
  - Monoliths, not microservices
  - En premises, not cloud-native
- Manual & semi-manual instrumentation
- Relatively simple architectures
  - Allowed ops teams to develop intuition for failure modes



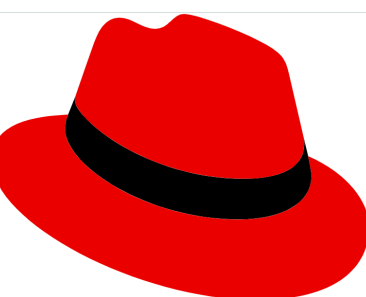


# A Typical Architecture



# What is Observability?

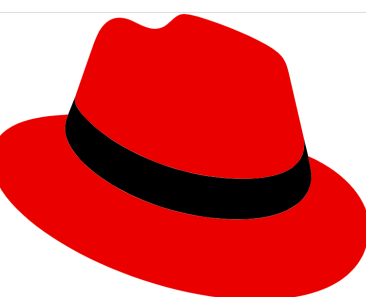
- Conceptually simple
  - Instrument systems & applications to collect observability data
  - Send this data to a system that can store & analyze it
  - Provide insights (for devops, management, SREs) into systems
  - Get answers to questions that you didn't know you'd have to ask
- System control theory
  - How well can internal state of a system be inferred from outside?
- Actionable insights from the entire system
  - Not just one piece
    - Tell you the overall health





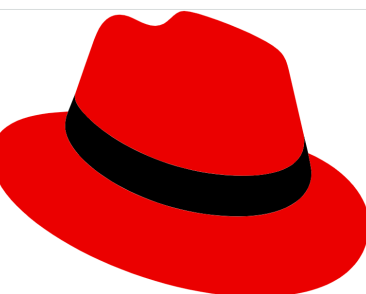
# How Do We Understand Cloud-Native Apps?

- Modern apps are much more complex
  - More services & components
  - More complex topology
  - More sources of change & more rapid change
- New technologies with new behaviours
  - Dynamically scaling services
  - Container environments
  - Kubernetes
  - Function-as-a-Service
  - Kafka



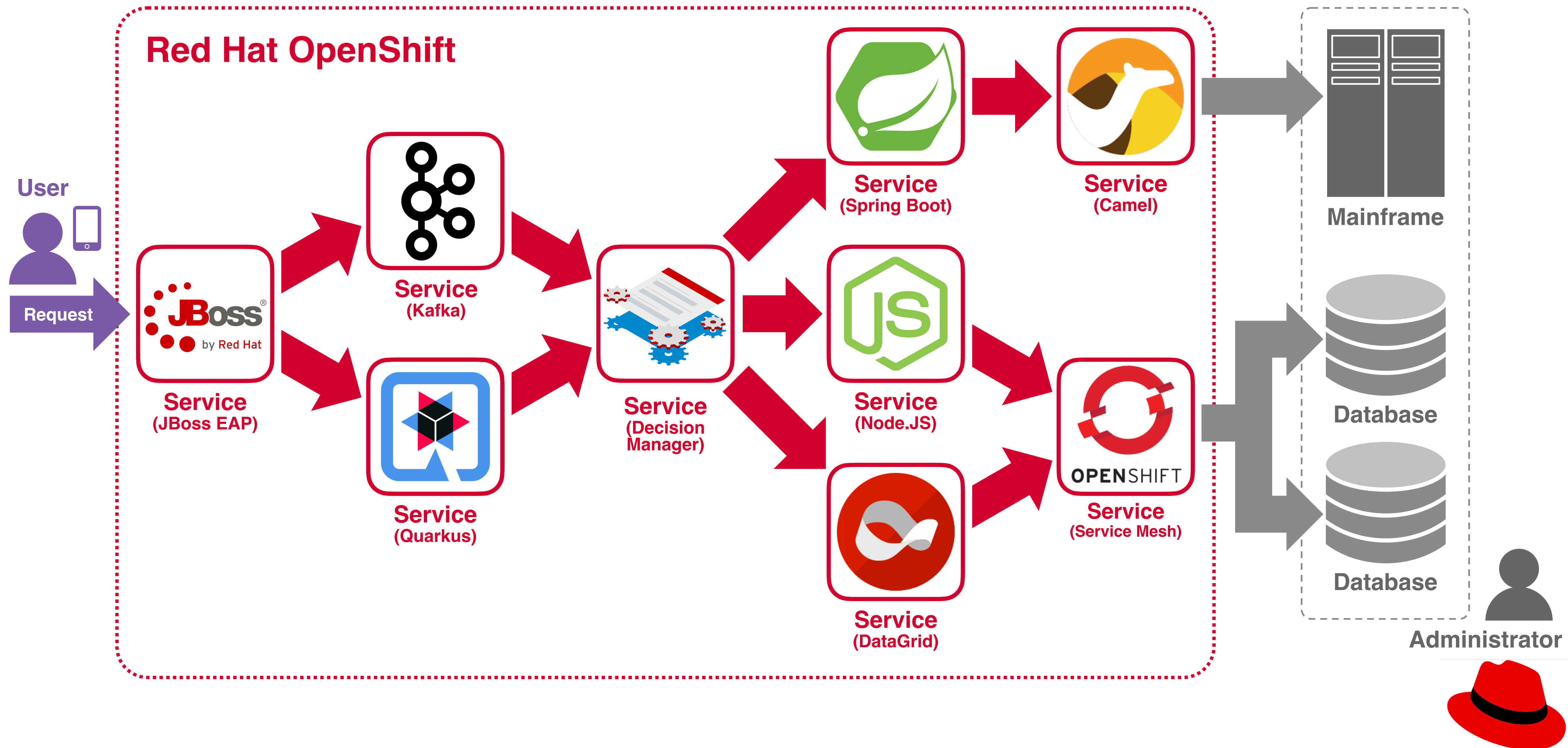
# Java In Containers

- Java designed for VM-on-bare-metal world
- Containers are here
- What do Java developers have to do to adapt?
- How does container Java measure up to other tech stacks?
  - Footprint
  - Density
  - Startup time

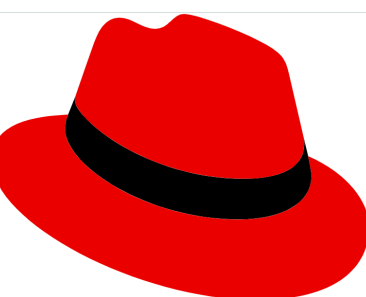
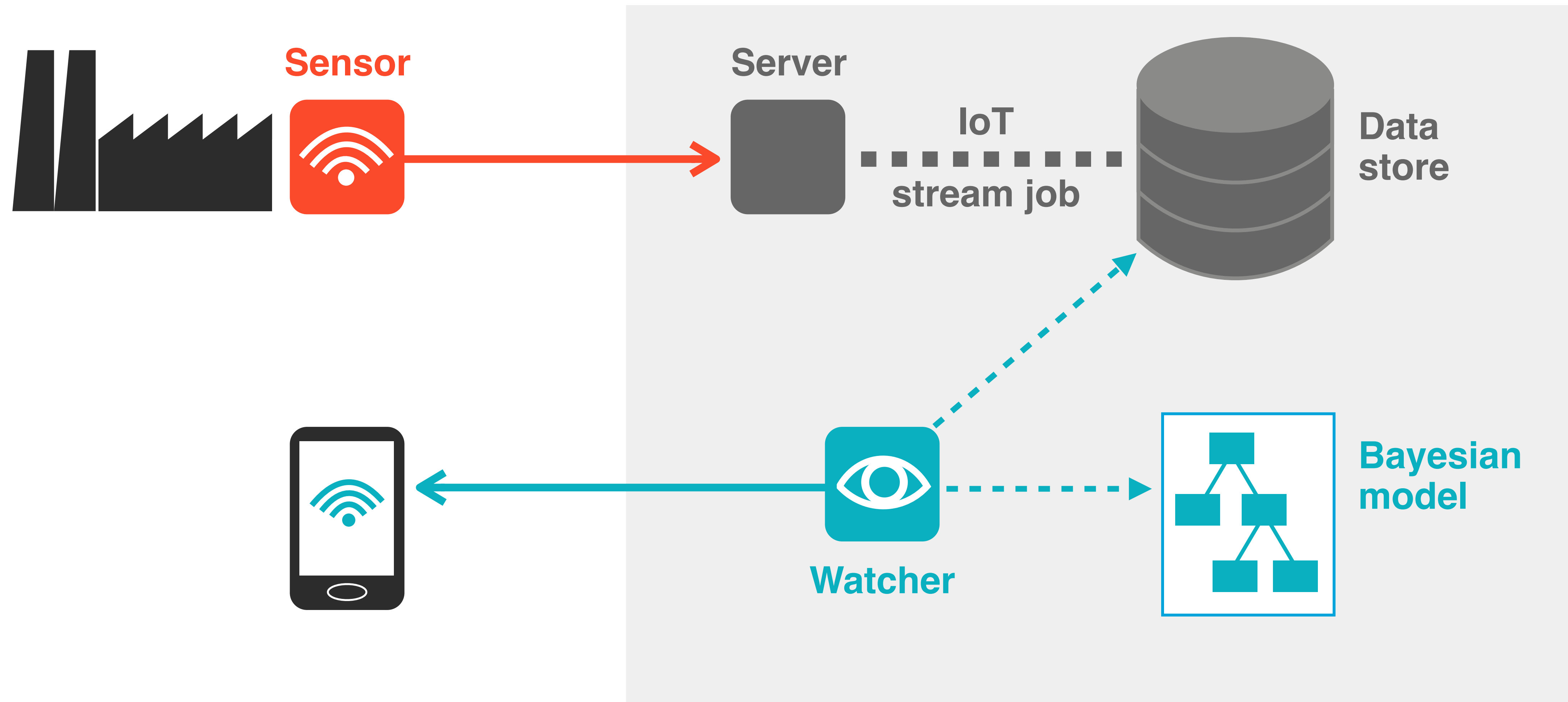




# Distributed System running on OpenShift

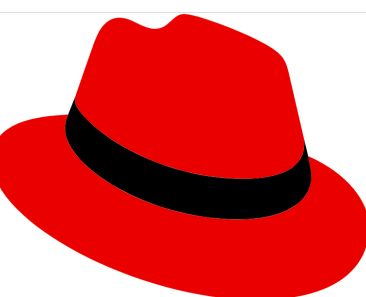


# IoT / Cloud Example



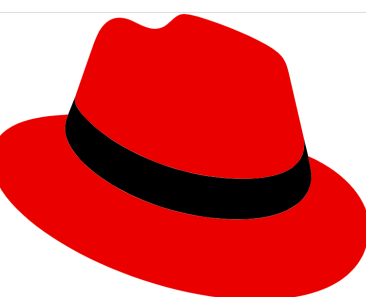
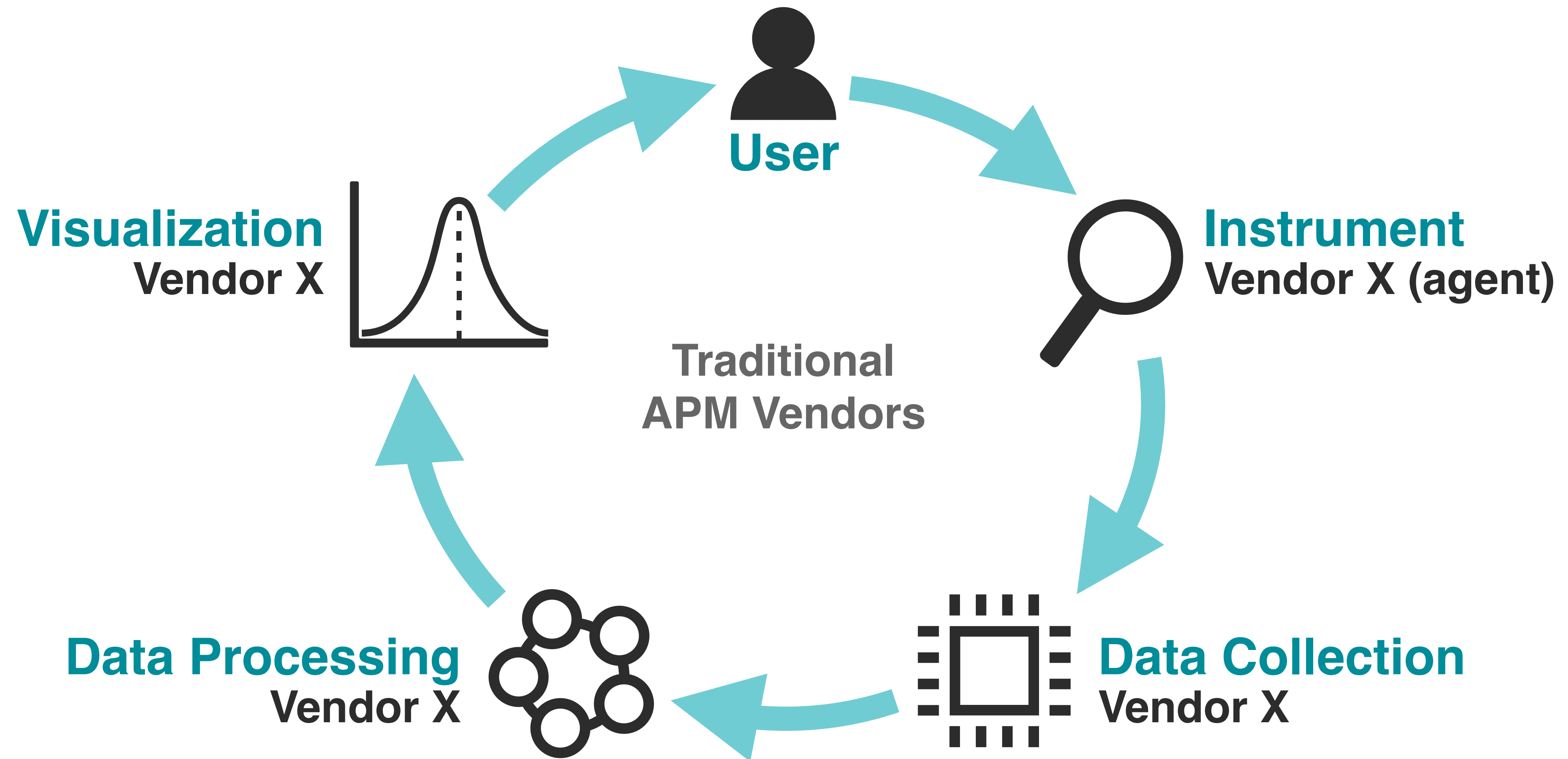
# User Perspective

- What is the overall health of my solution?
- What is the root cause of errors and defects ?
- What are the performance bottlenecks?
- Which of them could impact customer experience?

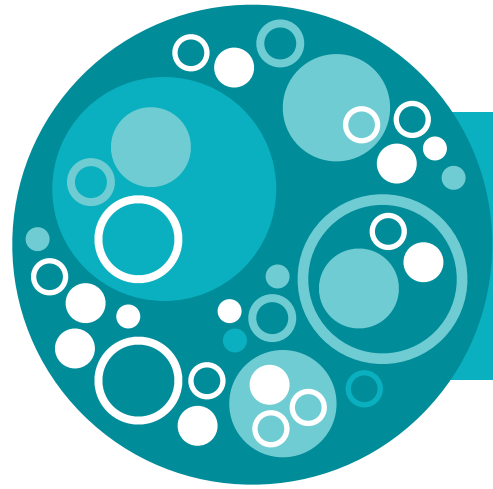




# Lock In Approach



# Complexity of Microservice Architectures



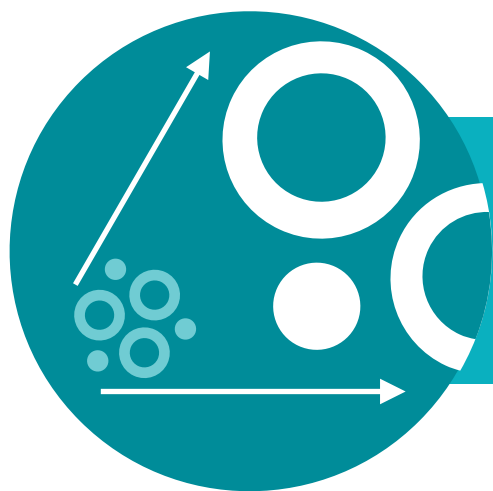
**Large numbers of smaller services**



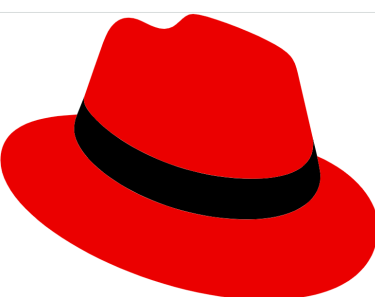
**Multiple Relevant Personas (Dev, DevOps, etc)**



**Heterogenous Tech Stacks**



**Scale out of service clusters**



# Portfolio Components

## Log Management

Log vendors collect and analyze system logs, often both for IT operations troubleshooting use cases and for security teams.

## Infrastructure Monitoring

Infrastructure monitoring tools collect data about the performance of infrastructure, including servers and storage devices, alerting users when performance problems occur.

## Alerting

Alerting vendors manage oncall personnel schedules, collect alerts from the range of monitoring systems in use and issue alerts to the correct responders when incidents occur.

## Performance Testing

Performance, or load, testing simulates user traffic at volume to test performance from the browser or app perspective.

## Application Performance Monitoring

APM products surface insight about app performance, including load, response time and errors. APM products include those that identify the line of code associated with a problem, as well as those that use distributed tracing technologies to deliver insight into performance.

## Serverless Monitoring

Serverless vendors that have developed mechanisms for collecting more performance data than is made available by the functions as a service providers.

## Real User Monitoring

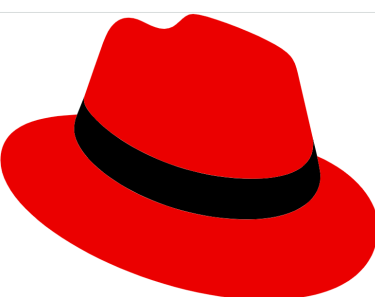
RUM tracks real users in order to isolate whether a performance problem is related to end user geography, browser type, operating system, device type or other common characteristics.

## Synthetic Monitoring

These tools generate synthetic traffic that's designed to mimic real user traffic in order to identify problems before they affect real users.

## Event Correlation

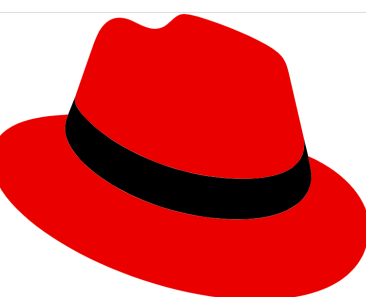
Event analytics tools analyze events issued across systems, correlating those that appear related into a single incident that responders can dig into to assess the root cause.





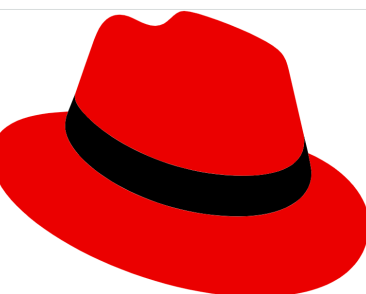
# Tools & Aspects of Observability

Tool	Usage
Infrastructure monitoring	Health and performance of the containers and environment
Application performance monitoring (APM)	Investigate application behaviour at the service level. Determine where calls are going and how they perform.
Real user monitoring	Understand the experience of real users by using data from browsers about how your site performs - and isolate issues to the frontend or backend.
Synthetic monitoring	Measure the impact that third-party APIs and network issues have on the performance and reliability of your app.
Log analysis	Dig deeper into the context of why issues are occurring



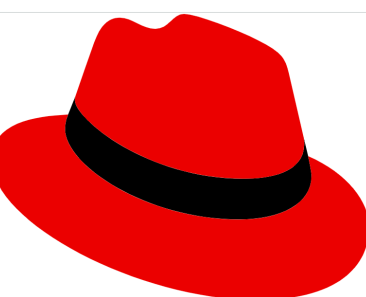
# What Data Do We Need To Collect?

- Traces
- Metrics
- Logs
- What Can We Do With The Data?

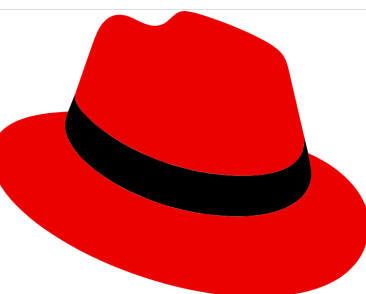
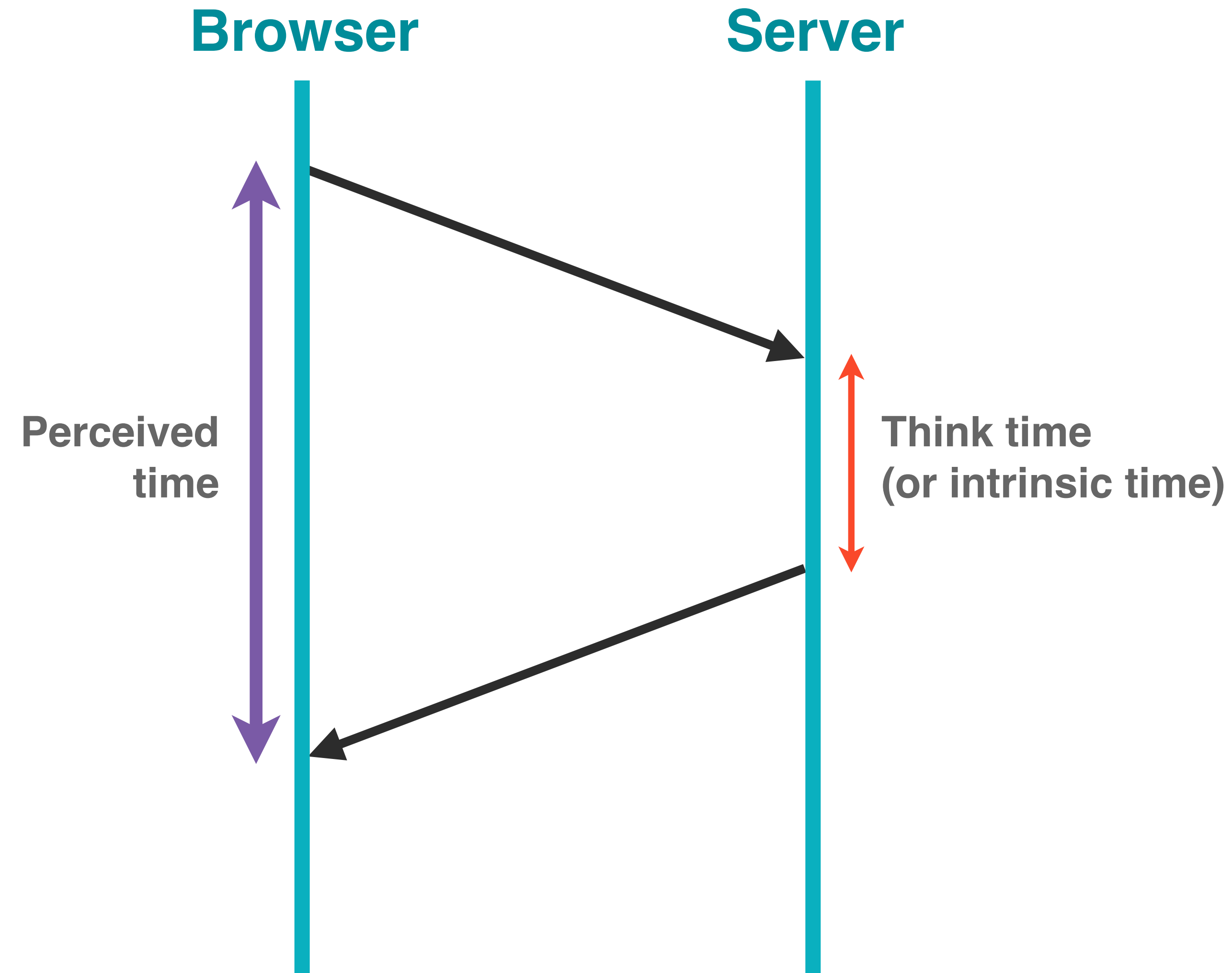


# Distributed Traces

- Shows for each service invocation
  - Which instance was called
  - Which method was invoked
  - How the request performed
  - What the results were
- Parts of a single user call (spans) collected into a trace
  - Which services were invoked
  - Which containers / hosts they were running on
  - What the results of each call were
- Traces form a tree structure of spans

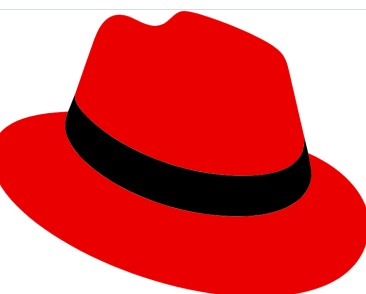
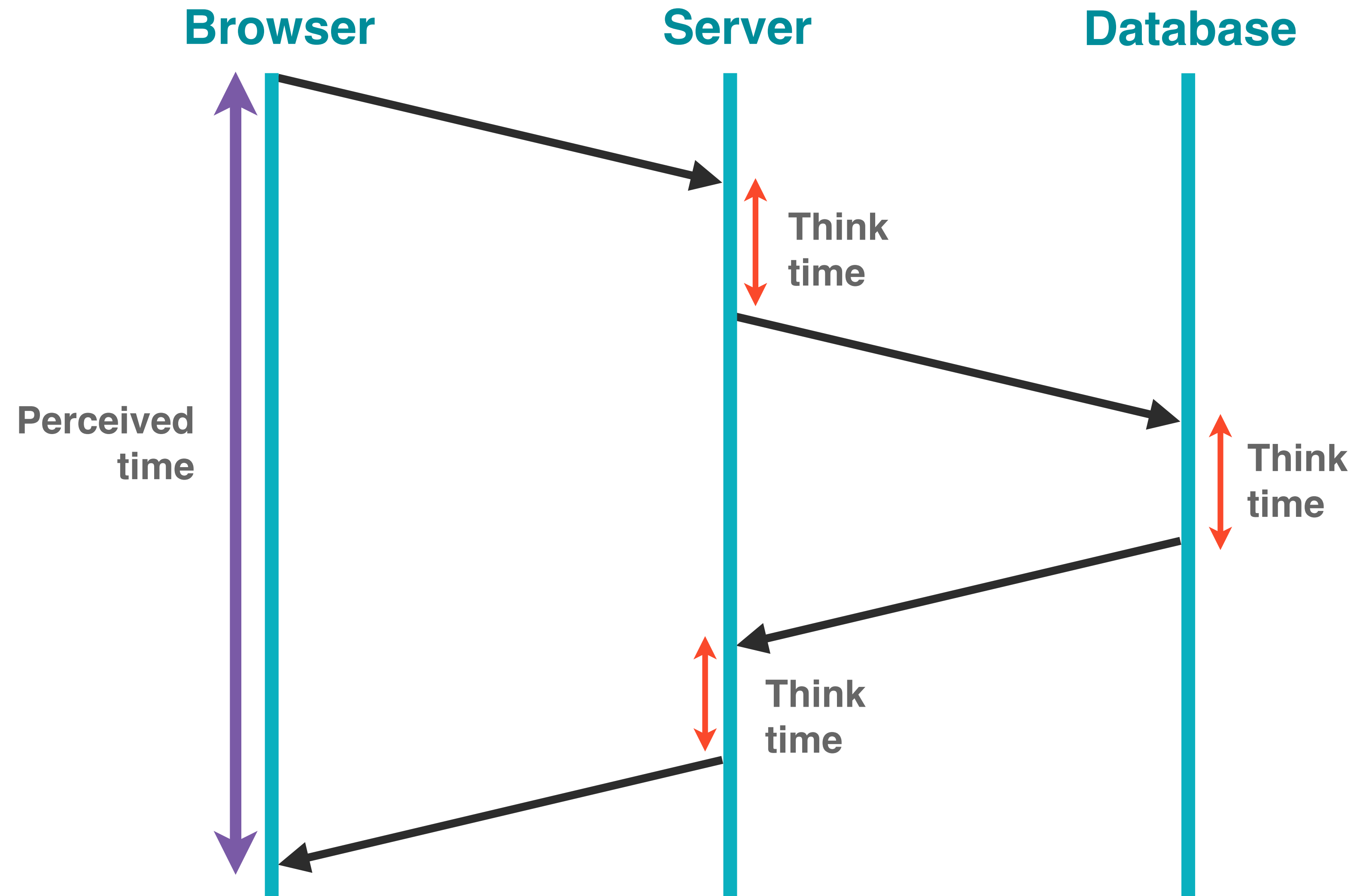


# Understanding Tracing

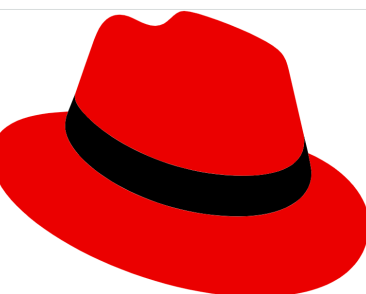
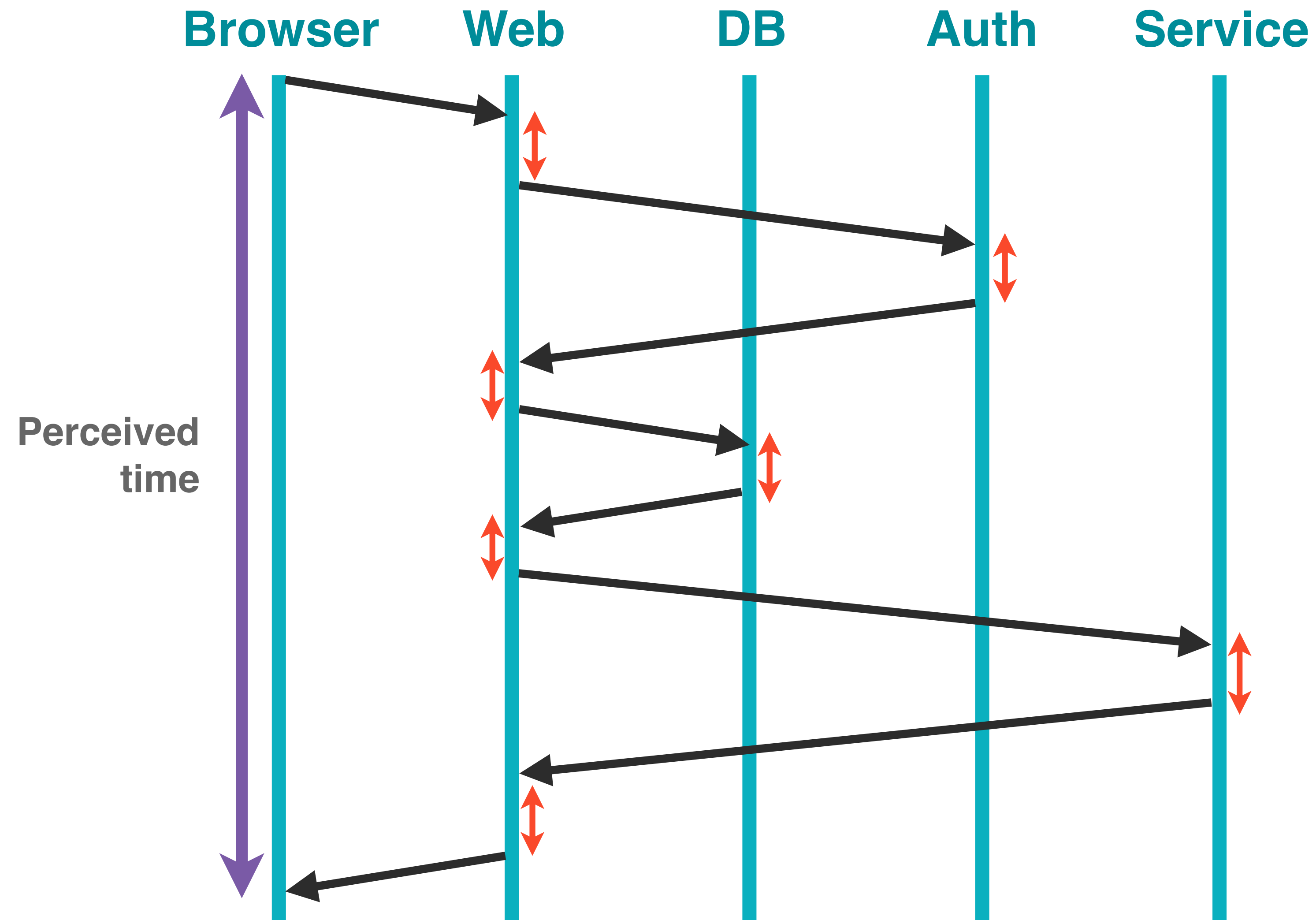




# Multi-Tier Apps

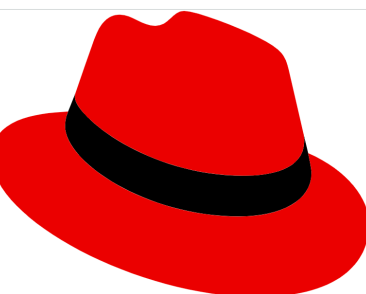


# Distributed Tracing



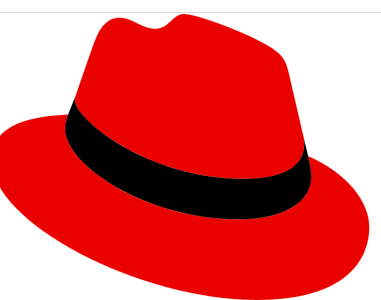
# Sample Span

```
{
  "trace_id": "kDMI7LTxLxTj220awNARJw==",
  "span_id": "9ir6veJ4Hdw=",
  "parent_span_id": "bgnsqqPvjYQ=",
  "name": "Sample-8",
  "kind": 1,
  "start_time_unix_nano": 1588334156464409000,
  "end_time_unix_nano": 1588334156470454639,
  "attributes": [{
    "key": "attr",
    "string_value": "value3"
  }],
  "status": {
    "message": "ok"
  }
}
```



# Distributed Tracing Components

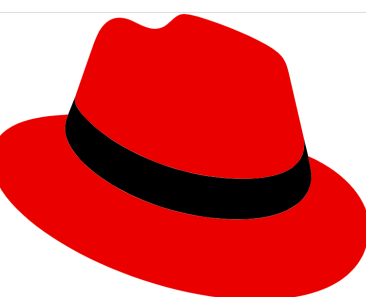
- Instrumentation of application & middleware
- Distributed context propagation
- Ingest & Storage
- Search & retrieval
- Visualization





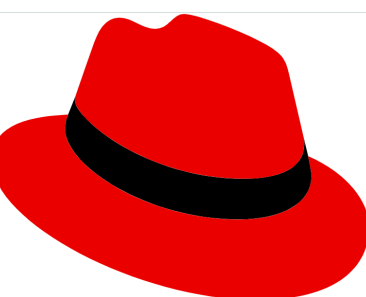
# Distributed Tracing Components - Tools

- Instrumentation of application & middleware
  - OpenTelemetry
- Distributed context propagation
  - OpenTelemetry
- Ingest & Storage
  - OpenTelemetry Collector, Jaeger
- Search & retrieval
  - Jaeger
- Visualization
  - Jaeger



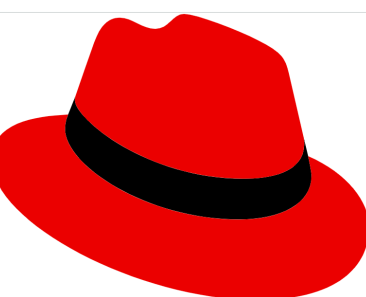
# Distributed Traces

- Visualization is the hard part
  - Traceview is usual form
  - Representation of how the tree of spans was generated
  - Too low-level
- New visualizations needed
  - Especially as systems become more complex
- What about data sampling?



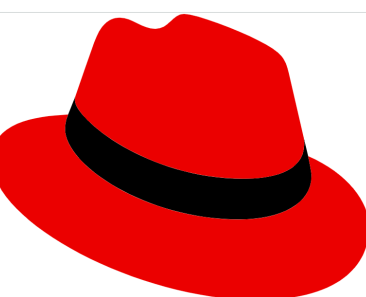
# Metrics

- Numbers measuring specific activity over a time interval
- Typically 4 parts: Timestamp, Name, Value, Dimensions
- Dimensions represent different values of some tag
  - Present as key / value
  - Values must allow for aggregation
  - Not all metrics systems support dimensions
- Volume doesn't scale with request traffic (unlike logs or traces)



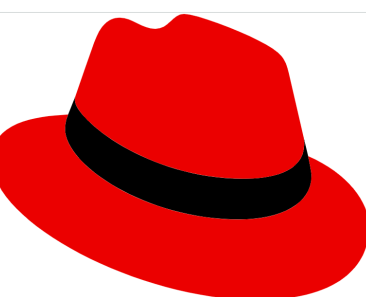
# Metrics - Examples

- System metrics (CPU, memory, disk)
- Infrastructure metrics (e.g. AWS CloudWatch)
- Web tracking scripts (e.g. Google Analytics)
- Application metrics (APM, error tracking)
- Business metrics (e.g. customer sign-ups)



# Architectural Aspects of Metrics

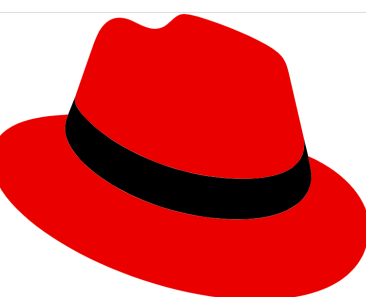
- Dimensionality
  - Does the consumer support key / value annotation of the measurement?
  - Other alternative is hierarchical
- Aggregation Discipline
  - Client-side - discrete samples converted to a rate before pub
  - Server-side - aggregation occurs at server
- Publishing
  - Client Push
  - Server Poll
- NB: Micrometer handles Key Differences Between Systems





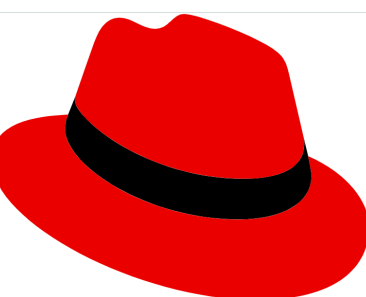
# Logs

- Immutable records of discrete events that happen over time
- 3 possible types - plain text, structured and binary
  - Not all products support all 3 types
- Examples
  - System & server logs (syslog)
  - Firewall & network system logs
  - Application (log4j)
  - Platform & server logs (Apache, nginx, databases)



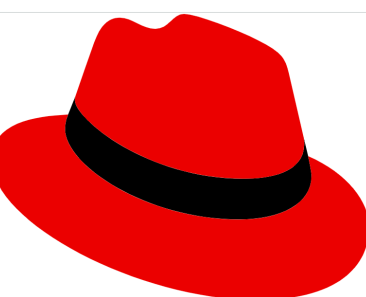
# Logs

- 2 types - Centralized & Distributed
- Centralized
  - Typical Observability pattern
  - Aggregate logs from individual microservices to single location
  - Single searchable, filterable and groupable dataset
- Distributed
  - Less common
  - Seen when some system constraint prevents centralized logging
    - E.g. bandwidth, contention with application processes
  - Some logging systems work better when deployed on app hosts



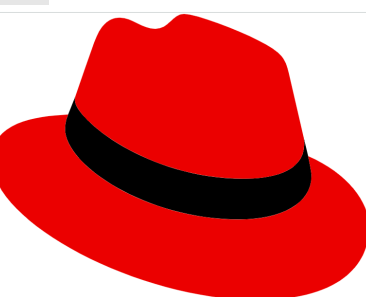
# What Can We Do With The Data?

- Some observables can be sourced in multiple ways
- E.g. overall response time
  - Log file
  - Metric
  - Infer from trace
- Decide which route is the golden source
  - May be differences between same signal sourced in different ways



# Aspects of the data

Pillar	Usage	Characteristic
Metric	Trend and graph	Aggregateable
Logs	Grep for results	Discrete Events
Traces	Identify slow subcomponents / services	Request Scoped



# Actionable Insights

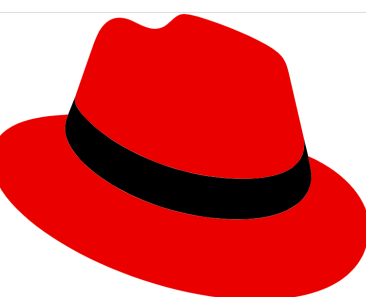
Error Type	Signal	Stakeholders
Service Alert	Metric	SRE, DevOps
Service Timeout	Tracing	DevOps, SDE
Infrastructure Problem	Metric	SRE, DevOps
Configuration Error	Metric	DevOps
Memory Leak	Metric, Log	SDE, DevOps



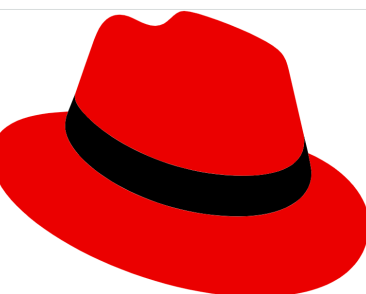
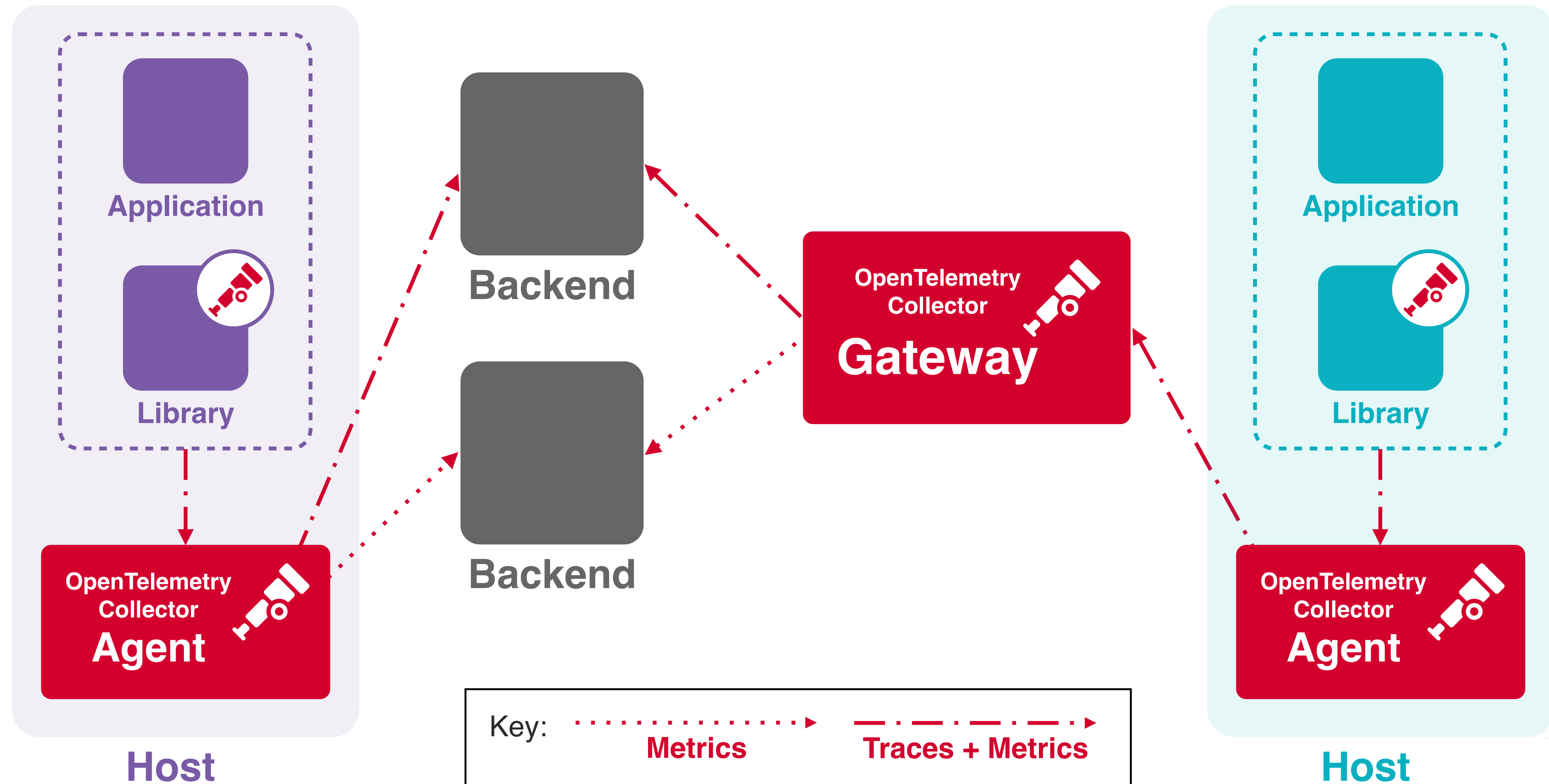


# What Architectural Implications Are There?

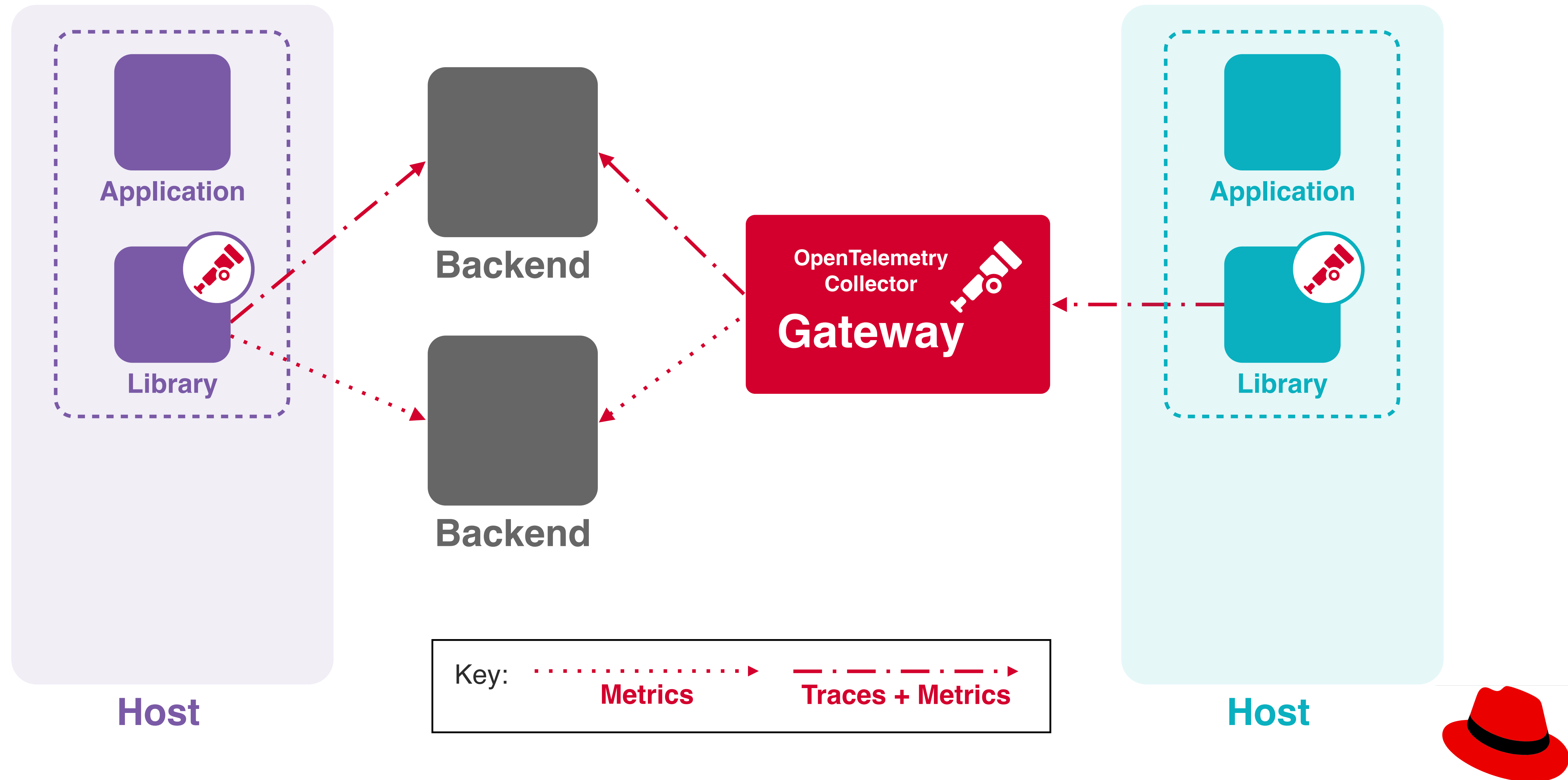
- Multiple Different Architectural Approaches
  - Implications for both apps & Observability components
- Most Common Choices (Observability)
  - Aggregate on host, send aggregates to SaaS
  - Send everything to SaaS (within Cloud region)
  - Keep everything on cluster (e.g. eBPF)
  - Roll Your Own



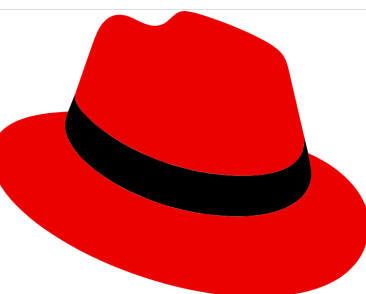
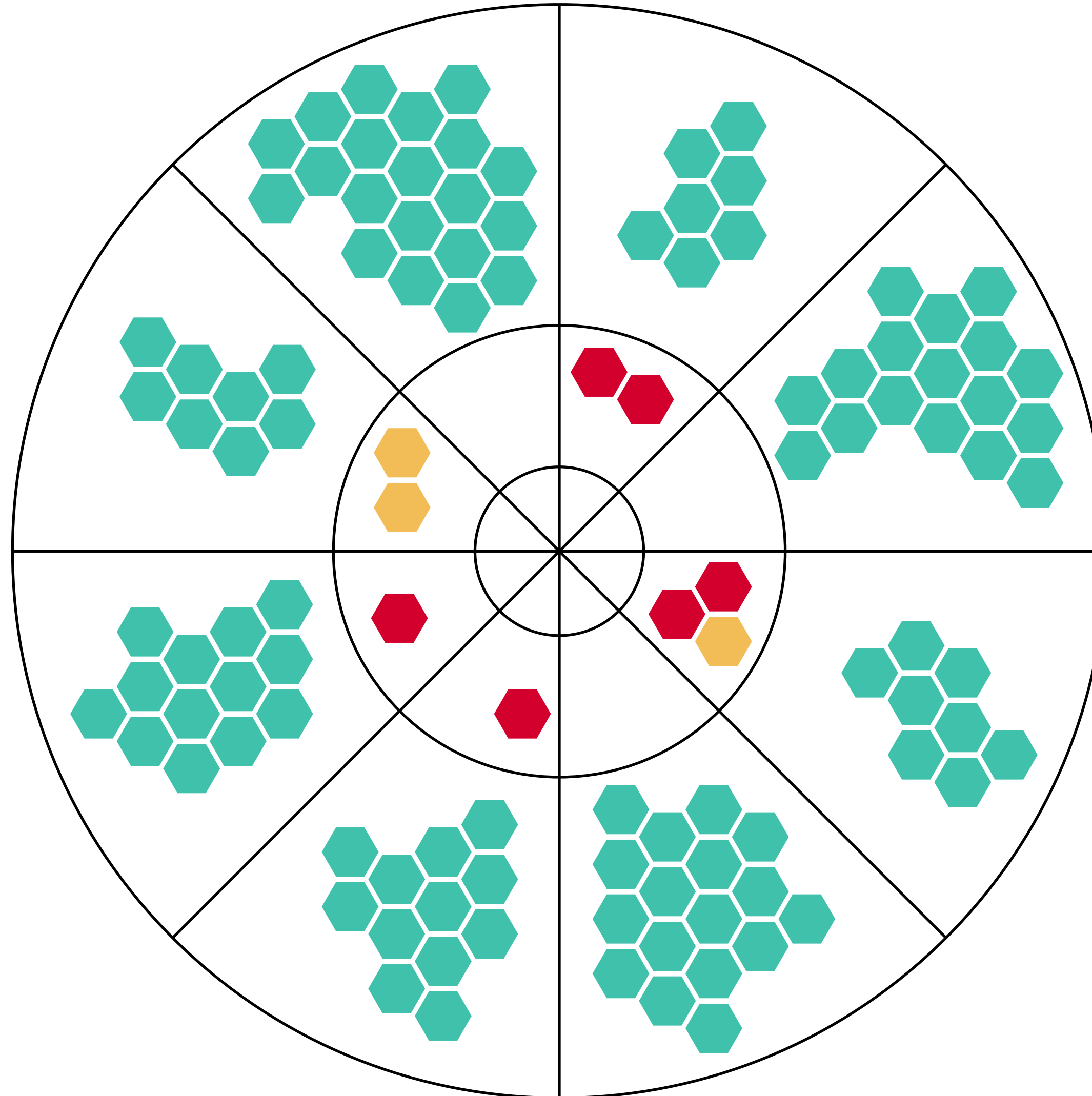
# Reference Architecture: Aggregate on Host



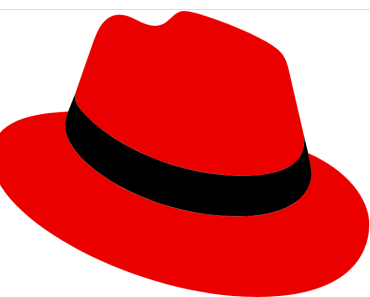
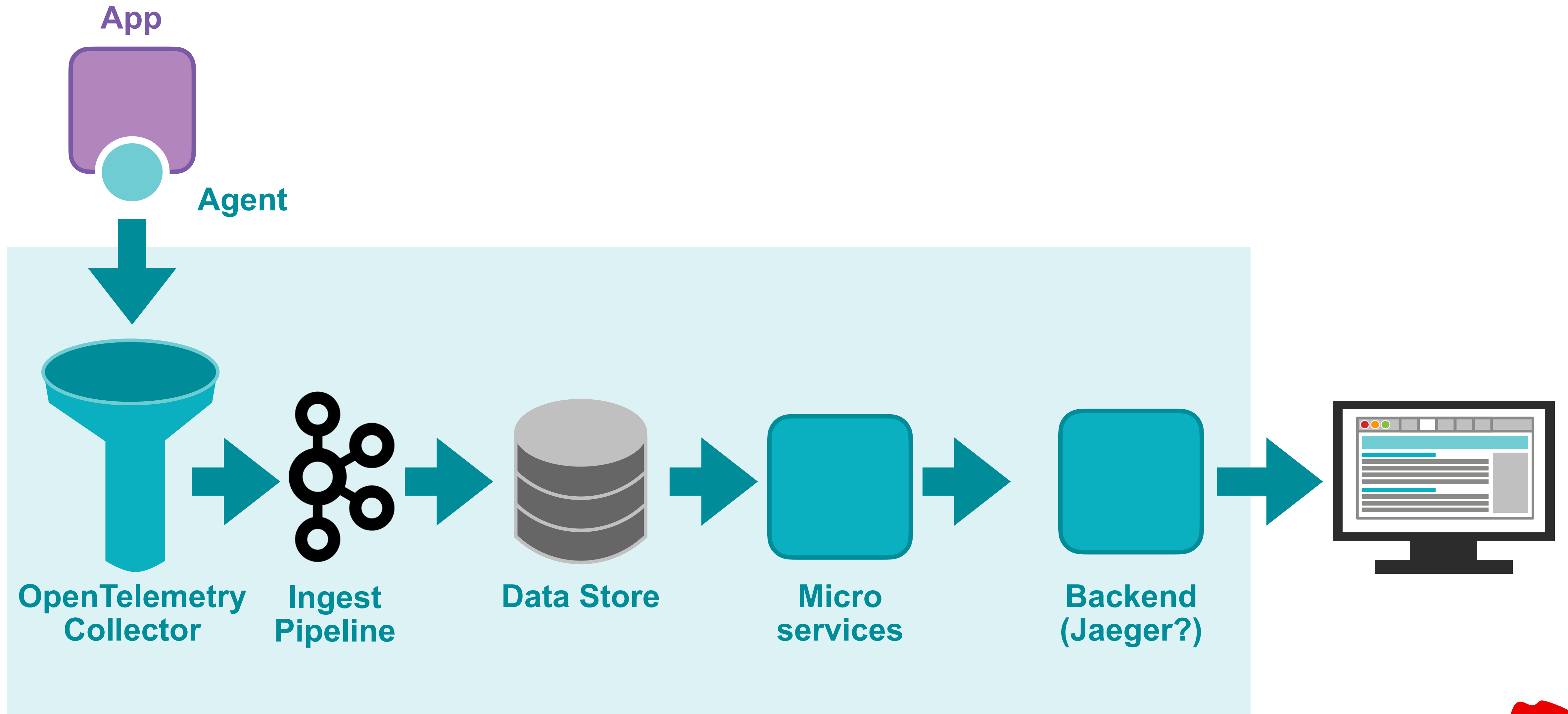
# Reference Architecture: Direct Send



# Architecture - Keep On Cluster



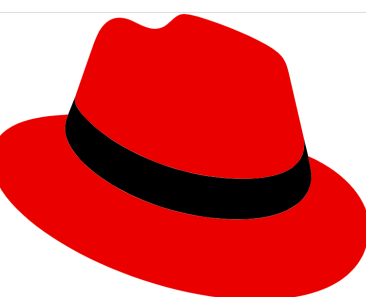
# Architecture - Roll Your Own





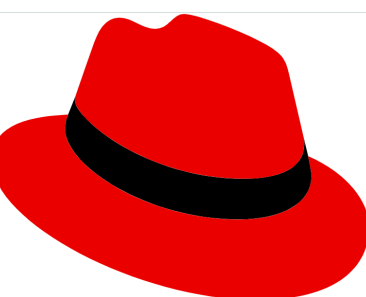
# Understanding Streaming Architectures

- Key ideas
  - Batch vs streaming
  - Event time vs processing time
  - Data windowing
  - Watermarks
  - Filtering & Joining (& related FP-like concepts)
- Useful blog posts (Tyler Akidau)
  - <https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/>
  - <https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/>



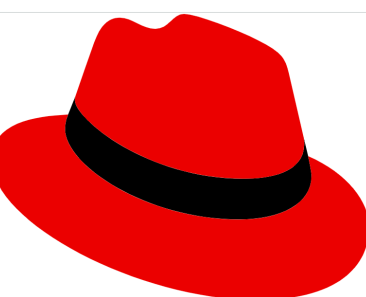
# Observability

- Helps handle the complexity of modern architecture apps
- Allows you to answer questions about your application & business
- Is a mindset, not a practice or product
- Absorbs & extends classic monitoring systems
- Helps identify the root cause of issues
- Predicts what could go wrong



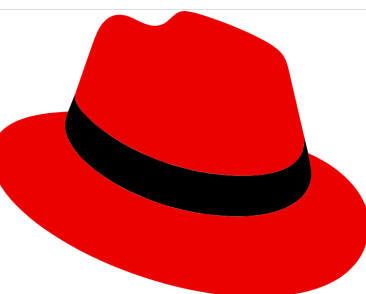
# Introducing OpenTelemetry

- Origins
- What is OpenTelemetry?
- OpenTelemetry Components
- Open Source Aspects
- Current Status
- Running OpenTelemetry



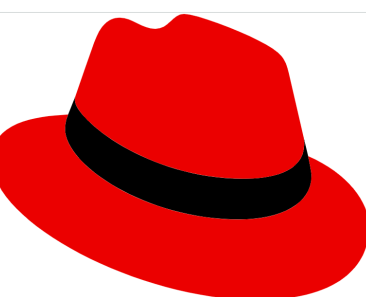
# Origin of OpenTelemetry

- APM / Monitoring dominated by proprietary vendors
- Various OSS projects started in response
  - Prometheus / OpenCensus
  - OpenTracing / Zipkin / Jaeger
  - Elasticsearch / Logstash / Kibana (ELK stack)
- Inflection point in the market
  - Switch from proprietary to OSS led
  - Move from APM -> Observability
  - Many vendors switching to OSS
  - Lots of Observability startups that have always been (partially) OSS



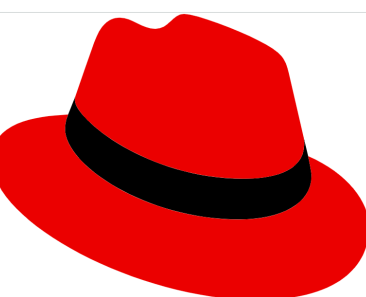
# What Is OpenTelemetry?

- OpenTelemetry is a set of standards, formats & libraries
  - NOT a data ingest or Observability backend
  - Aggregation possible at different levels & stages
  - Supports Observability data architecture options
- Explicitly cross-platform: NOT Java-specific
  - Java is mature implementation
  - Go, .NET, Node also all fairly mature
- Works with bare metal & VMs
  - But explicitly part of the CNCF - definitely cloud-first



# Comparing OTel & other OSS tools

- OpenTelemetry
  - Framework that integrates with OSS & commercial products
  - Collect telemetry from apps written in many languages
  - OTel Tracing based on merger of OpenTracing / OpenCensus
- Other early adopter cloud tools
  - collectd: a daemon that collects metrics
  - statsd: a daemon that listens for statistics
  - fluentd: a daemon that unifies log collection
  - Zipkin, Jaeger: OSS distributed tracing back-ends
    - Jaeger has deprecated own client libs in favour of OpenTelemetry





# What Are Components of OpenTelemetry?

## OpenTelemetry Tracing Components

The **API** contains the interfaces used by developers when instrumenting their applications and libraries.

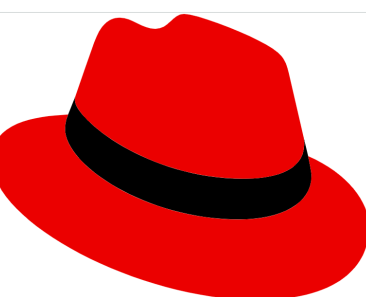
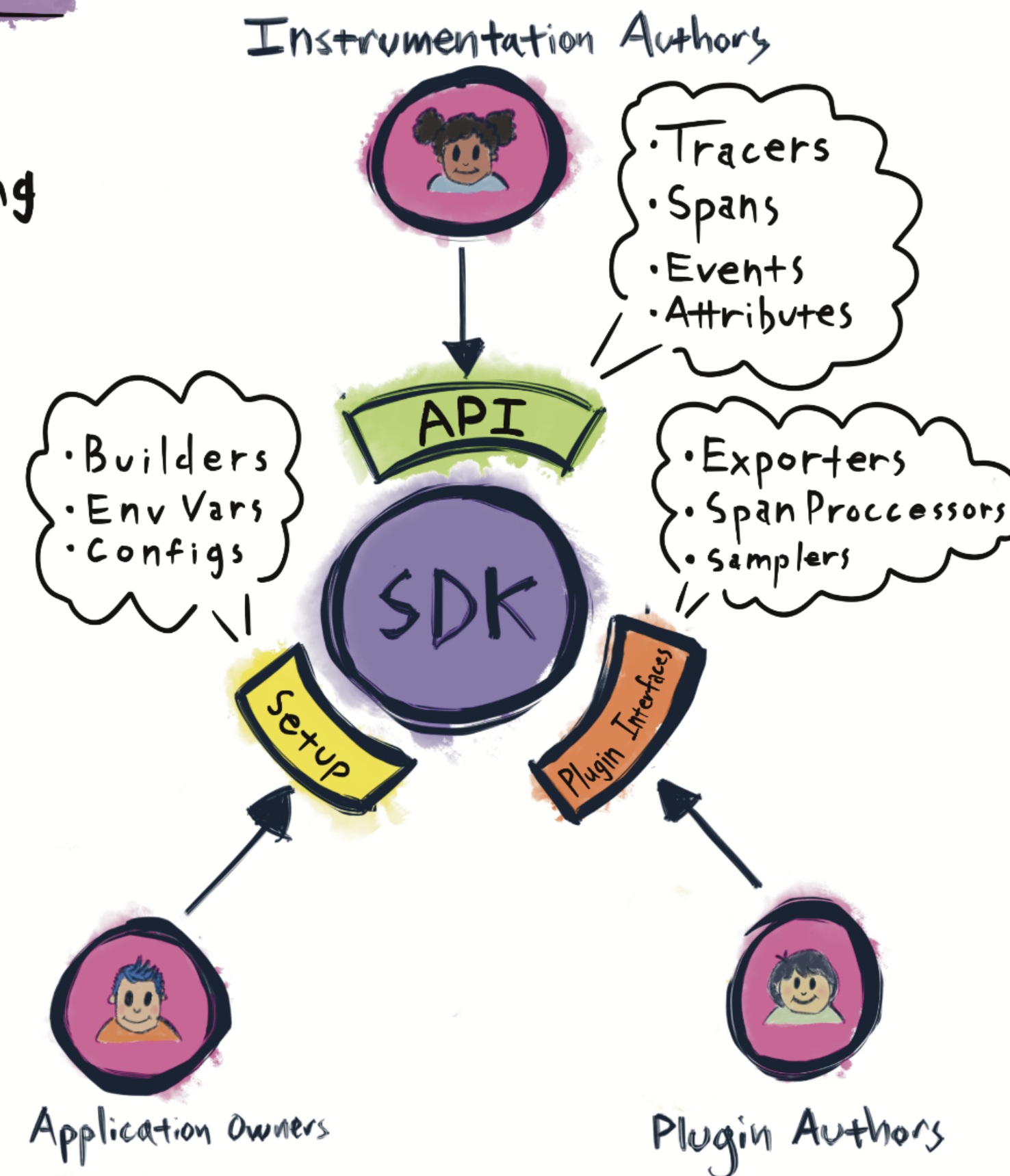
The **SDK** contains the constructors used by application owners to configure their setup, and the interfaces used by plugin authors to write integrations.

## Long Term Support

**API:** 3 year support guarantee

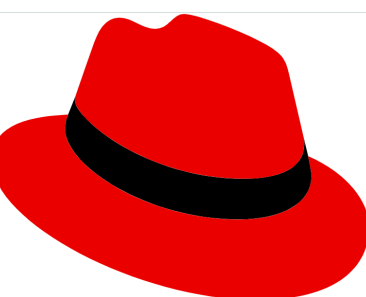
**Plugin Interfaces:** 1 year support guarantee

**Constructors:** 1 year support guarantee



# OpenTelemetry Components

- The API contains:
  - Interfaces used by developers to instrument their apps & libraries
- The SDK contains:
  - Constructors used by application owners to configure their set-up
  - Interfaces used by plug-in authors to write integrations
- Long term support
  - API: 3 years support guarantee
  - Plug-in Interfaces: 1 year support guarantee
  - Constructors: 1 year support guarantee

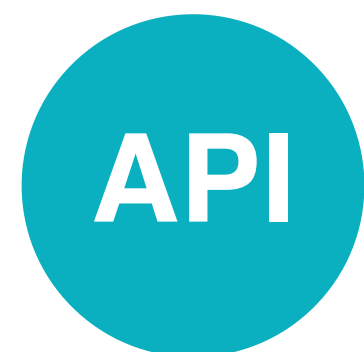


# OpenTelemetry Components



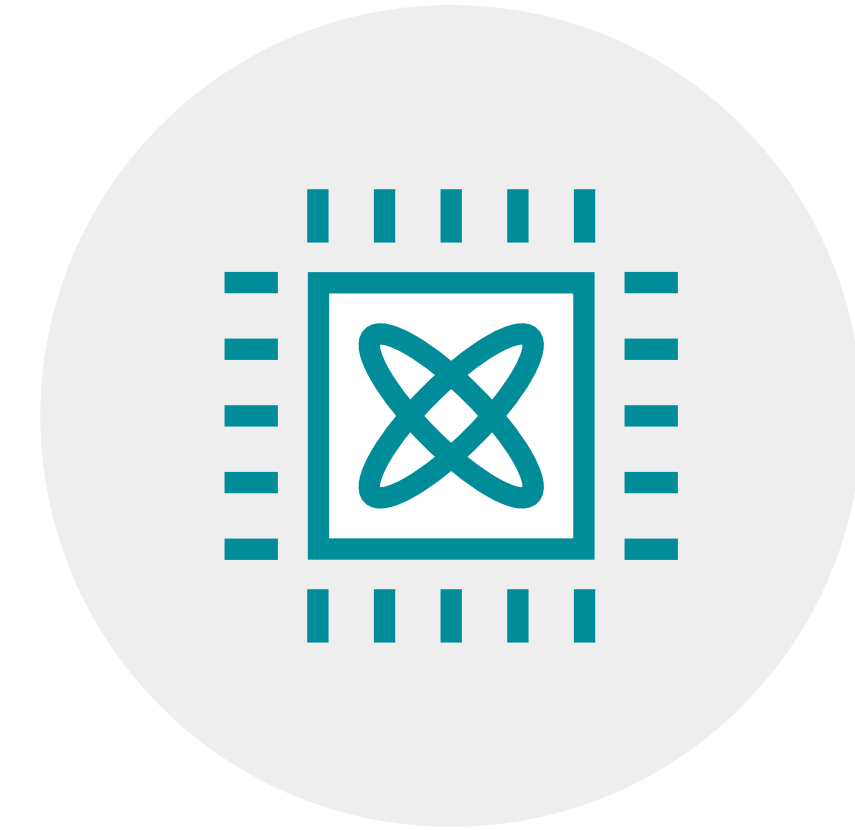
## Specification

Describes the cross-language requirements and expectations for all implementations



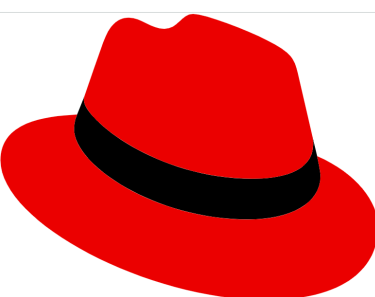
## Instrumentation

Make every library and application observable out-of-the-box



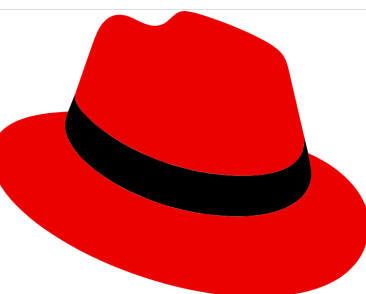
## Collector

Vendor-agnostic implementation on how to receive, process and export telemetry data

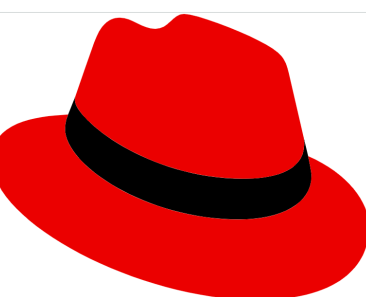
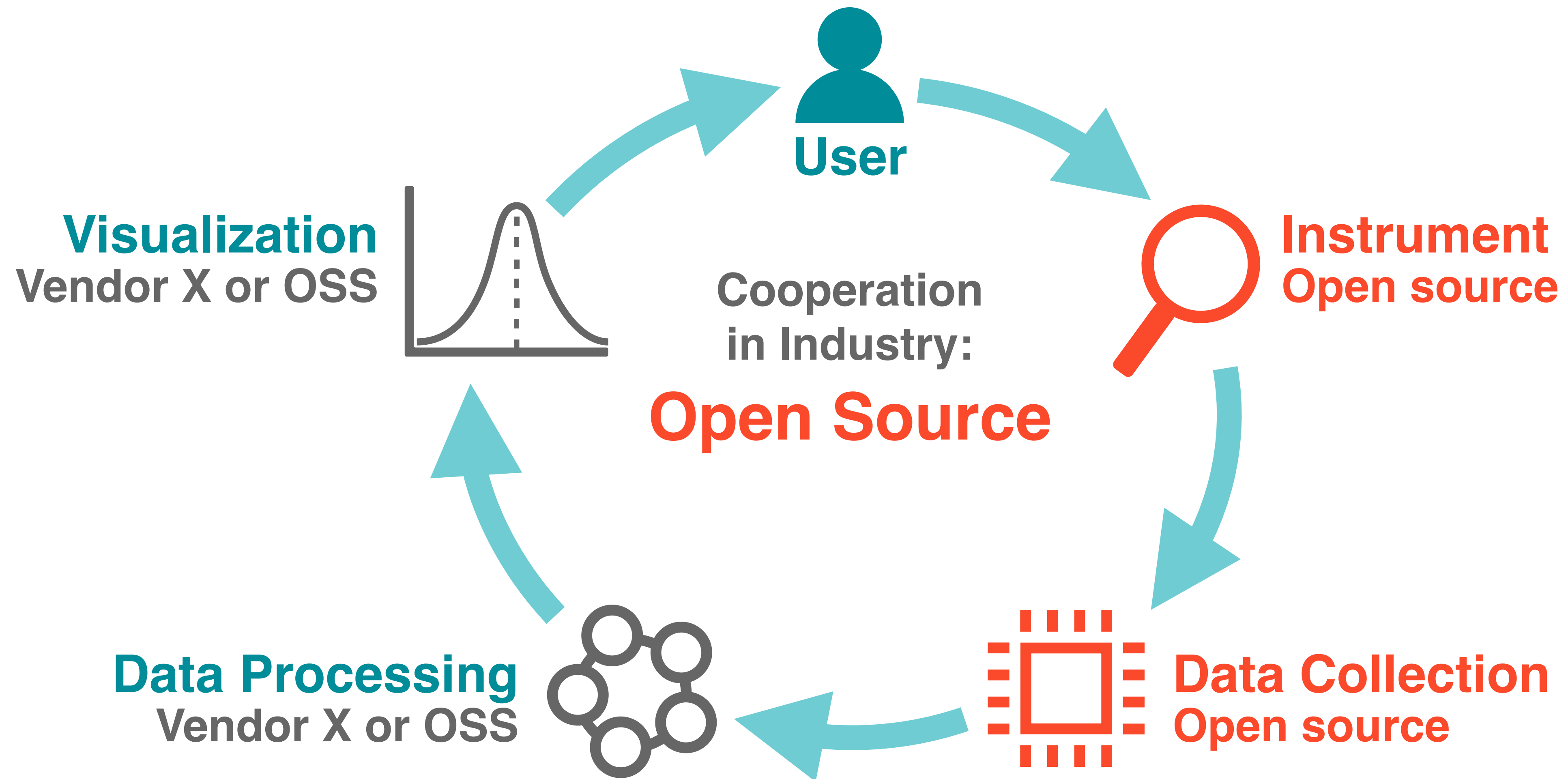


# Isn't This Just APM with New Marketing Terms?

- Vastly reduced vendor lock-in
- Open specification wire protocols
- Open-source client components
- Standardised architecture patterns
- Increasing quantity of open-source backend components

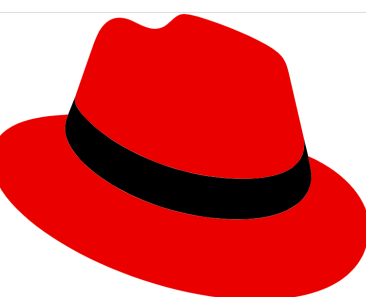
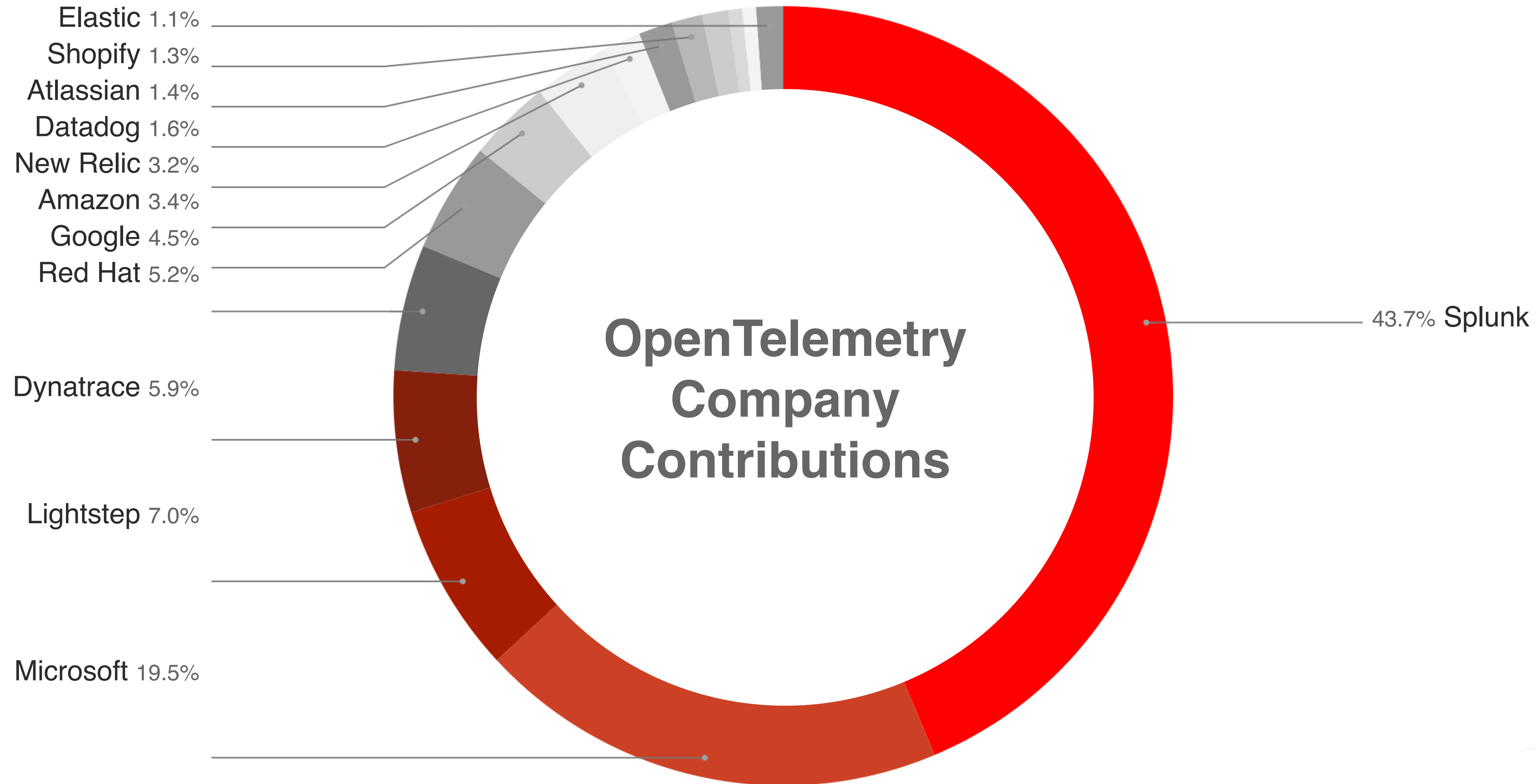


# Open Source Approach





# Who's Contributing?



# The Three Pillars



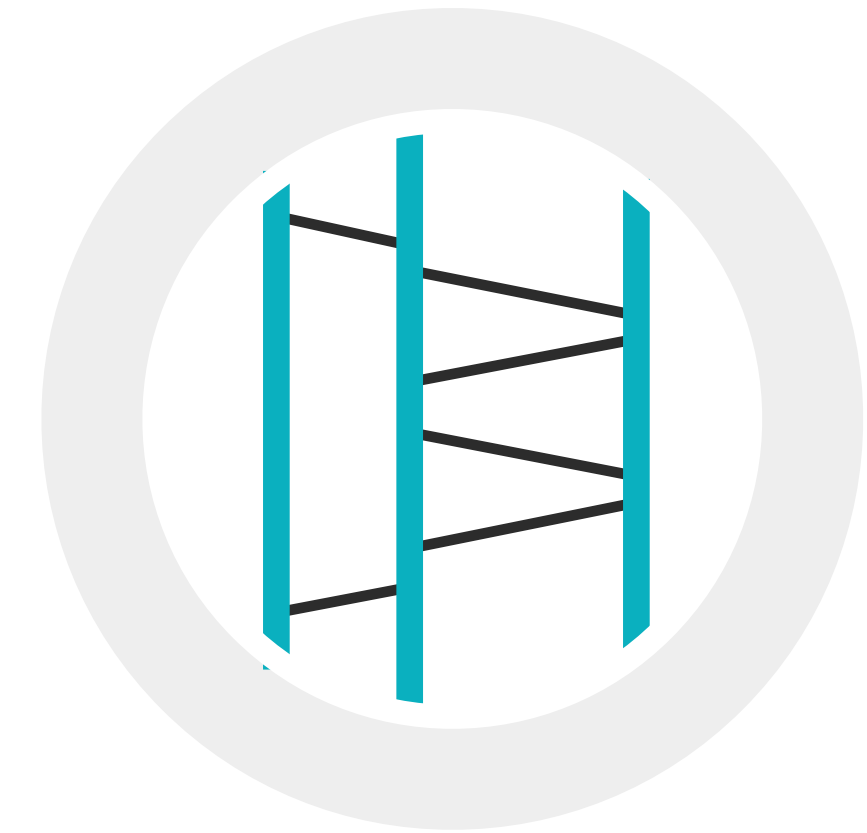
## Metrics

Numbers describing a particular process or activity measured over intervals of time



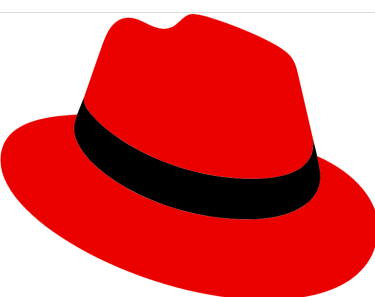
## Logs

Immutable record of discrete events that happen over time



## Traces

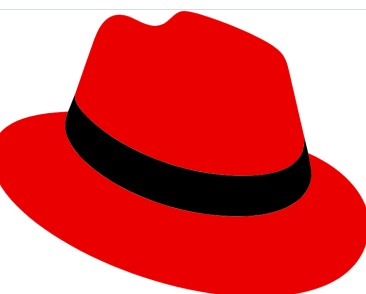
Data that shows which line of code is falling to gain better visibility at the individual user level for events that have occurred





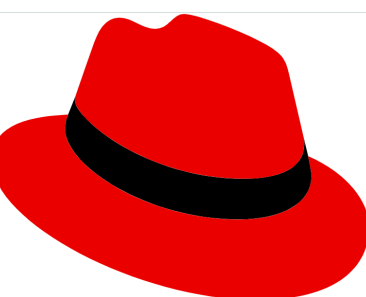
# What Is Current Status Of OpenTelemetry?

- Distributed Tracing is at v1.0
  - Track the progress of a single request
  - Replaces OpenTracing completely
- Metrics is at 1.0
  - Both application & runtime metrics
  - Still some ongoing work
  - Will Metrics replace Prometheus over time?



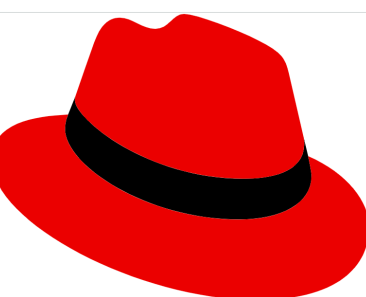
# What Is Current Status Of OpenTelemetry?

- Logging is still in DRAFT
  - Not expected to reach v1.0 until late 2022
  - Anything that is not a trace or metric is a log
  - Do Logs really split into subtypes: Logs & Events
- Different language implementations have different maturities
  - Java / JVM is a mature implementation
  - Other langs, e.g. Python, may have more issues



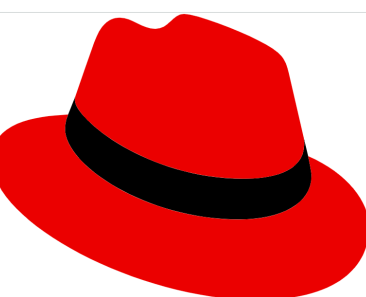
# Different Areas Have Different Competitors

- Traces
  - No serious OSS competitors to OTel
- Metrics
  - Prometheus already well established for K8S
    - Less well-established elsewhere
    - Prometheus has somewhat stagnated of late
- Logs
  - Log landscape is rich with incumbent solutions
    - Both OSS and vendor
- Many companies want a unified approach

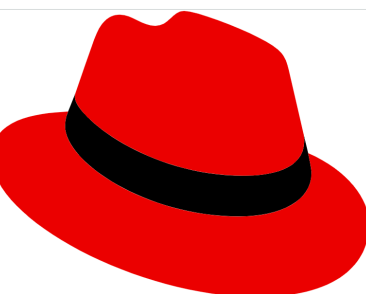
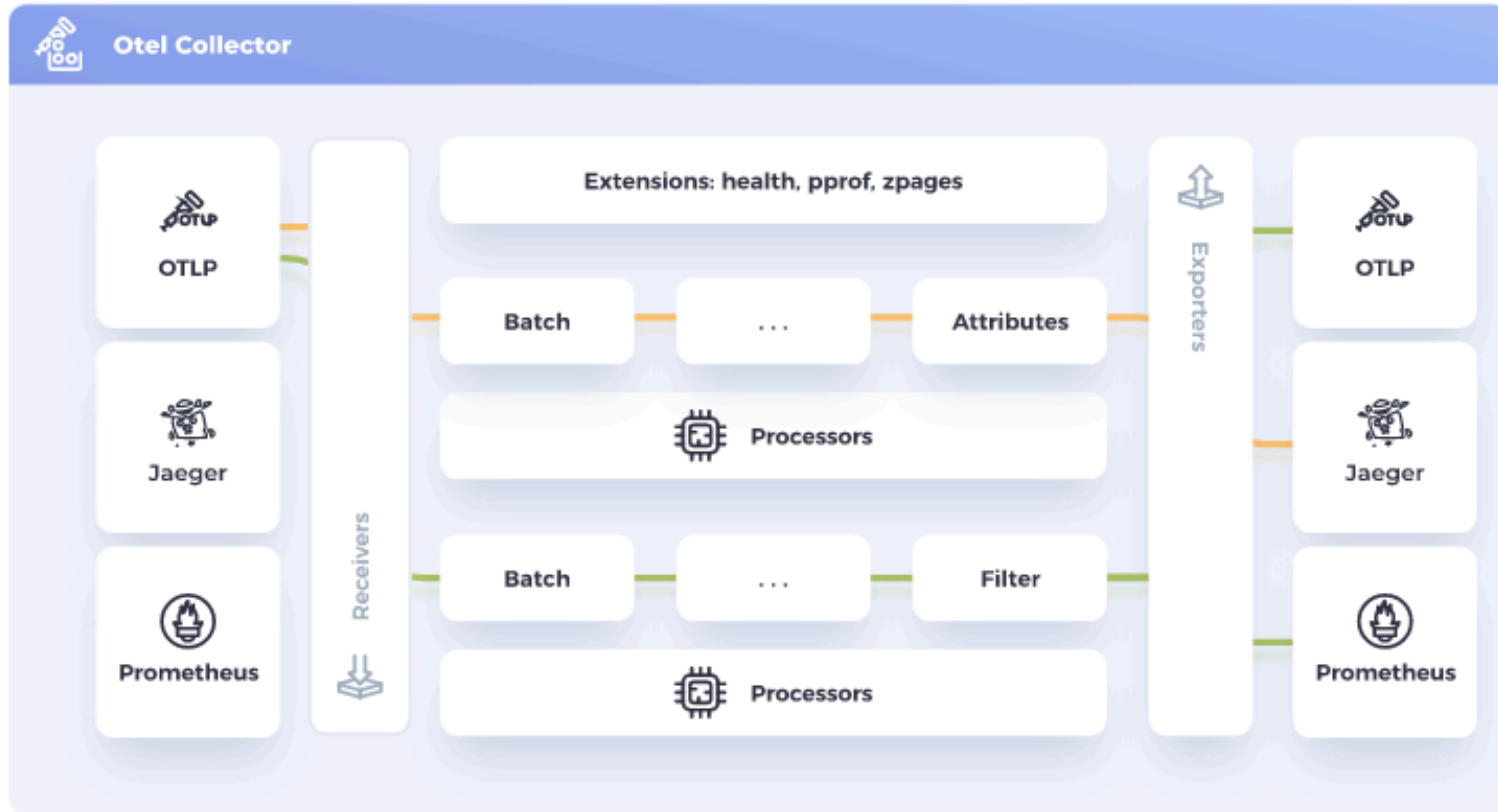


# Running OpenTelemetry

- The Collector
  - Network service that can
    - Receive
    - Process
    - Export telemetry data
- Written in Go
- Vendor neutral
- Works with a wide variety of data formats
  - Routes to a variety of OSS or vendor backends



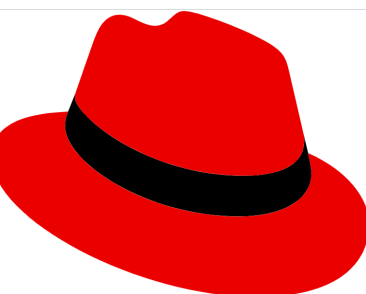
# Collector Architecture



# Running A Collector

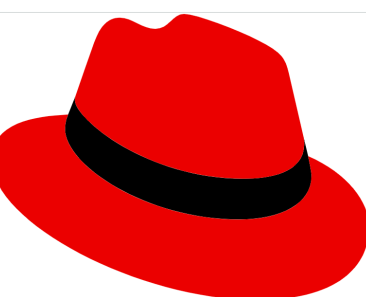
# DEMO

<https://github.com/kittylust/OTel>



# Collector Objectives

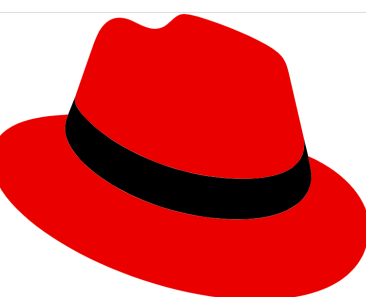
- Usability: Reasonable default config & works out of the box
- Performance: Performant under varying loads and configurations
- Observability: Good example of an observable service
- Extensibility: Customizable without touching the core code
- Unification: Single codebase supports traces, metrics & log





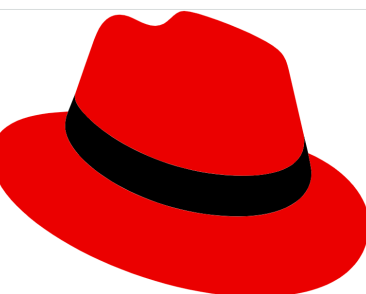
# OTLP

- OpenTelemetry Protocol (OTLP) defines
  - Encoding
  - Transport
  - Delivery
- Performance is a key concern
  - Typically encoded as Protobuf over HTTP/2
  - Java implementation uses GRPC libraries for high-performance
  - Other encodings possible, e.g. HTTP / JSON



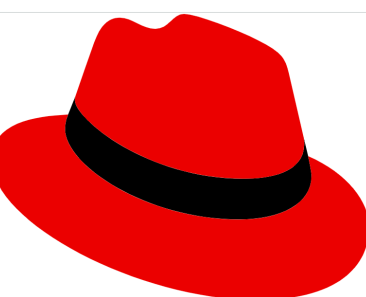
# Java & OpenTelemetry

- Basic Concepts
- Traces in OpenTelemetry
- Metrics in OpenTelemetry
- Auto-Instrumentation
- JFR & OpenTelemetry



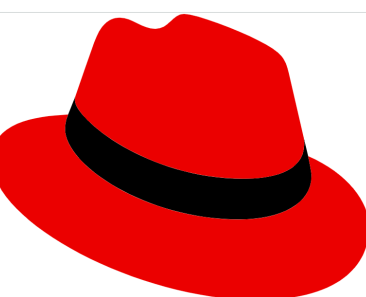
# Manual & Automatic Instrumentation

- Manual Instrumentation requires calls to telemetry library
  - Record traces or metrics
  - Spans created & finalised
    - Headers used to propagate IDs between services
  - Similar approach to logging (e.g. log4j)
- Automatic Instrumentation
  - Depends upon bytecode weaving
    - Java Agent - `java.lang.instrument`
    - Framework support (special classloader)
  - Halfway house
    - Using special annotations

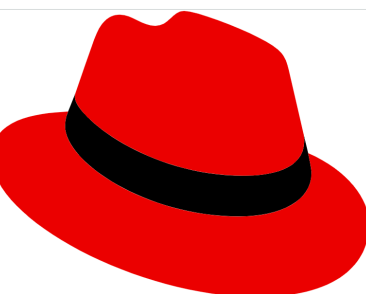
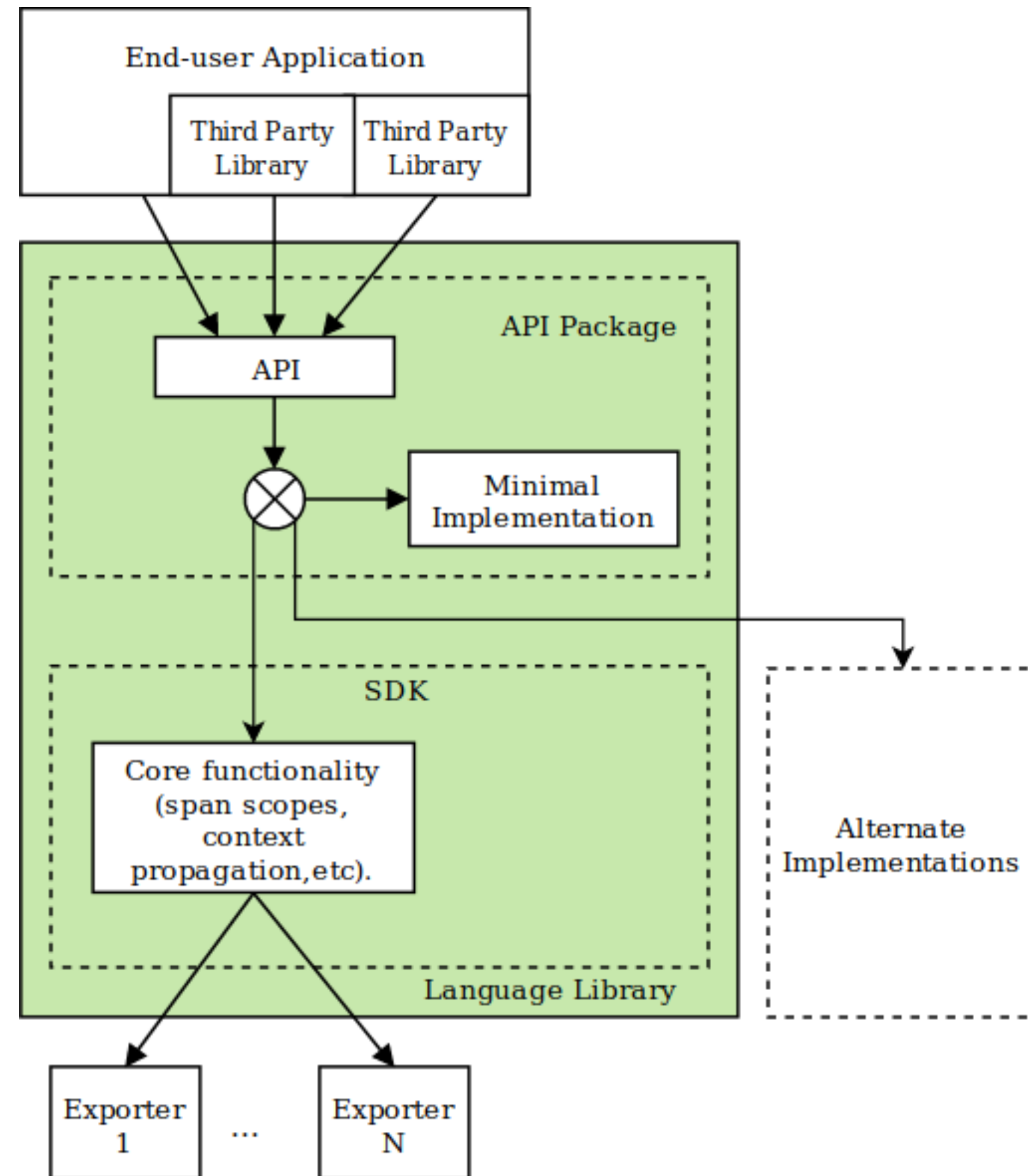


# Java Otel Project Structure

- Three main projects under the open-telemetry org
- opentelemetry-java: Manual instrumentation including API & SDK
- opentelemetry-java-instrumentation: Auto instrumentation agent
- opentelemetry-java-contrib: Helpful & standalone libraries
  - JMX metric gathering
  - JFR support (Beta)
- Focused on Traces & Metrics for now



# Application Architecture



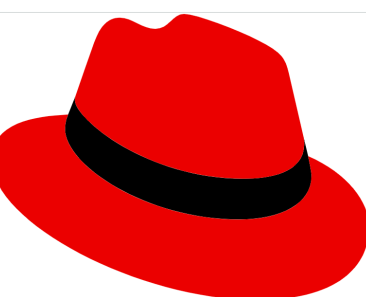
# Tracing Architecture

- Key components
  - API
  - SDK
  - Exporter (OTLP)
- Some configuration is always required



# Tracing in Java OpenTelemetry

- Traces are sampled
  - Trade-off due to load
- Open Questions
  - Exactly how to conduct the sampling?
  - Should everything be sampled at the same rate?
  - Sample errors more frequently?
  - Long-tail sampling?
- Only client libs provided by OTel Java projects
  - Other components are also needed

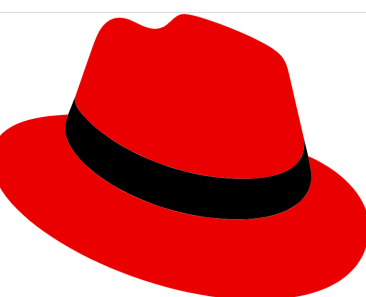




# Manual Instrumentation (Traces)

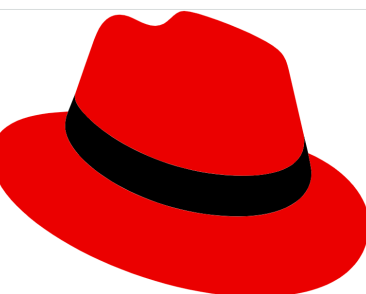
# DEMO

<https://github.com/newrelic/newrelic-opentelemetry-examples>  
otel-nr-dt



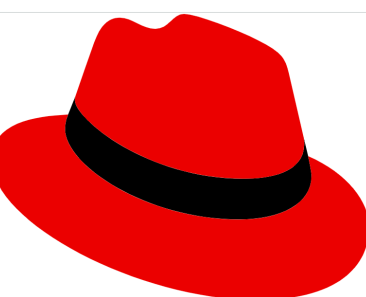
# Metrics in Java OpenTelemetry

- Metric collection is controlled by the OpenTelemetry SDK
- Meter providers (usually global) are used as an entry point
  - Meter objects are usually cached
- Metric event is captured along with timestamp & any metadata
- 3 primary instruments
  - Counter (only increases)
  - Measure (a value aggregated over time)
  - Observer (current set of values at a point in time)



# Aggregation in OpenTelemetry

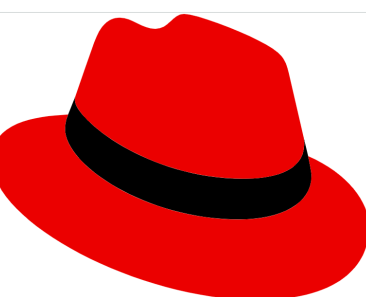
- Aggregation is absolutely key
  - Performed by the SDK, not app code
  - Developer has limited control over aggregation - by design
  - Some architectures aggregate at multiple scales
- Common aggregations
  - Sum, count, last value, histogram
  - Percentiles are more complex (can't aggregate easily)



# Manual Instrumentation (Metrics)

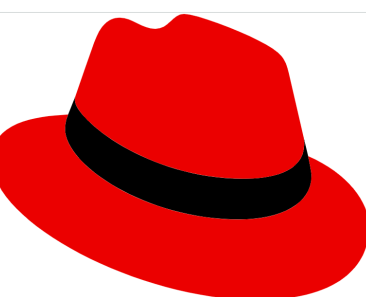
# DEMO

<https://github.com/newrelic/newrelic-opentelemetry-examples>  
sdk-nr-config



# Status of Otel Metrics

- Manually instrumented code
  - API is guaranteed to be stable
- Semantic conventions still not 100% stabilized
  - Meaning of some data might change in the future
- Code is 100% production ready
  - Precise nature of emitted data might change (slightly)



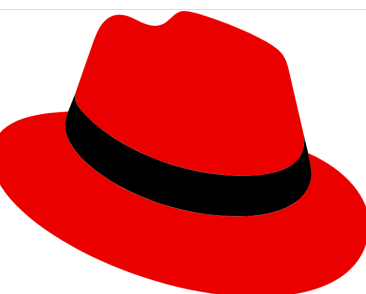
# Java Auto-Instrumentation

- Problems with Manual Instrumentation
- Java Agents
- Demo: Framework Support (Quarkus)
- OTel Java Agent



# Problems With Manual Instrumentation

- A lot of manual work to instrument code
  - And to keep it up to date
- Confirmation Bias
  - How can you be sure what's important to instrument?
  - What happens as the application changes over time?
- Often only find out what's important in an outage

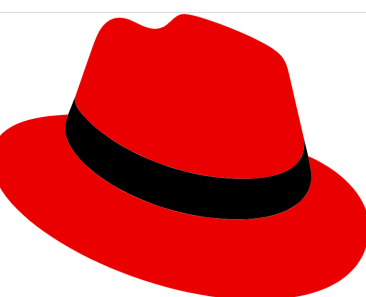




# Java Agents

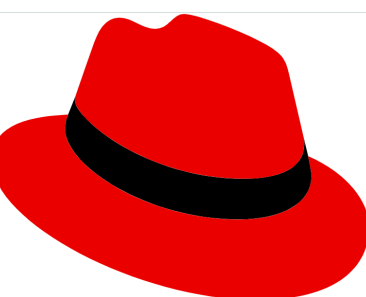
- Tooling component
  - Written in Java
  - Uses `java.lang.instrument`
  - Modifies the bytecode of methods at load time
- To install, use a startup flag:

`-javaagent:<path-to-agent-jar>=<options>`



# Java Agents

- The agent jar must
  - Contain a manifest (`Manifest.MF`)
  - Manifest must contain the attribute `Premain-Class`
  - The Premain class must contain a `premain( )` method
    - Pre-registration hook for the agent
  - Specialized signature:  
`public static void premain(String args, Instrumentation instrumentation);`
  - From `java.lang.instrumentation`

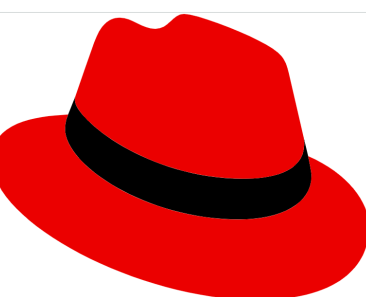


# Example Java Agent

- API provides a simple hook for agents
  - An entry point

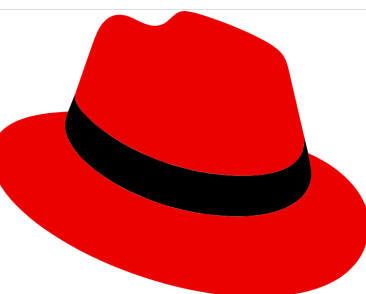
```
public class AllocAgent {  
    public static void premain(String args, Instrumentation instrumentation) {  
        AllocRewriter transformer = new AllocRewriter();  
        instrumentation.addTransformer(transformer);  
    }  
}
```

- Real work is done in the transformer classes



# Bytecode Weaving

- Java bytecode can be rewritten (or “weaved”)
- Specialized libraries used for this
  - CGlib
  - ASM
  - ByteBuddy
- Modified version of ASM is used internally by the JDK
  - Lambda Expressions



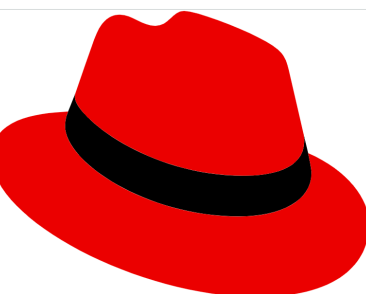
# Example Class Transformer

```
public class AllocRewriter implements ClassFileTransformer {

    @Override
    public byte[] transform(ClassLoader loader, String className, Class<?> redef,
        ProtectionDomain pd, byte[] bytes throws IllegalClassFormatException {

        ClassReader reader = new ClassReader(bytes);
        ClassWriter writer = new ClassWriter(reader, COMPUTE_FRAMES | COMPUTE_MAXS);

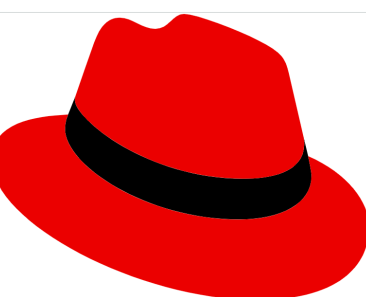
        ClassVisitor coster = new ClassVisitor(Opcodes.ASM5, writer) {
            @Override
            public MethodVisitor visitMethod(int ax, String name, String desc,
                String sig, String[] ex) {
                MethodVisitor bmv = super.visitMethod(ax, name, desc, sig, ex);
                return new AllocationRecordingMethodVisitor(bmv, ax, name, desc);
            }
        };
        reader.accept(coster, ClassReader.EXPAND_FRAMES);
        return writer.toByteArray();
    }
}
```



# Framework Instrumentation (Traces)

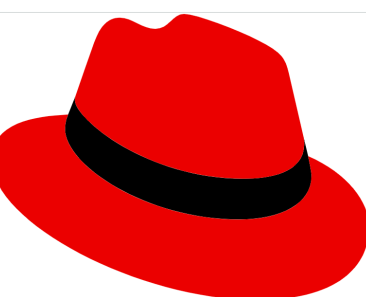
DEMO

<https://quarkus.io/guides/opentelemetry>



# Otel Java Auto Instrumentation

- Provides Java agent that attaches to any Java 8+ application
- Dynamically injects bytecode to capture telemetry traces
- Supports many popular libraries and frameworks OOTB
- By default, uses OTLP exporter & local collector

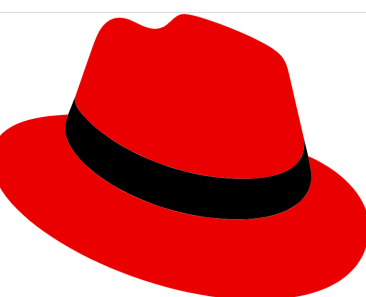




# Auto Instrumentation (Traces)

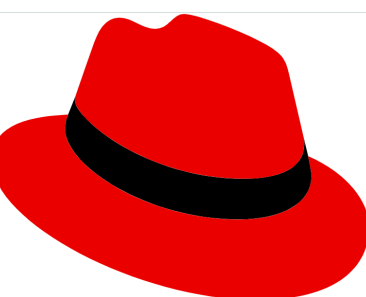
# DEMO

<https://github.com/newrelic/newrelic-opentelemetry-examples>  
agent-nr-config



# Conclusions & Future Roadmap

- Observability is reaching more & more developers
- OpenTelemetry is gathering steam
- Expect Metrics to reach 1.0 Spring 2022
  - Agreeing JVM metrics piece is a key deliverable
- Logs will take longer
- Analysts anticipate OTel will become majority format in 2023



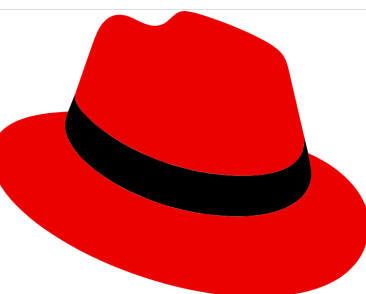
# What Are The Challenges?

- Diversity of tools & technologies in use
- Large matrix of possible deployments
- Need to understand which Observability architecture to use
- OpenTelemetry is still maturing
  - Long-term support for standards that aren't finished yet?
  - OSS groups and standards need more participation
  - Long tail for older tech already deployed
  - Logs are an interesting case



# A Short Reading List

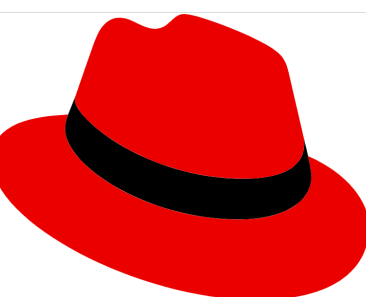
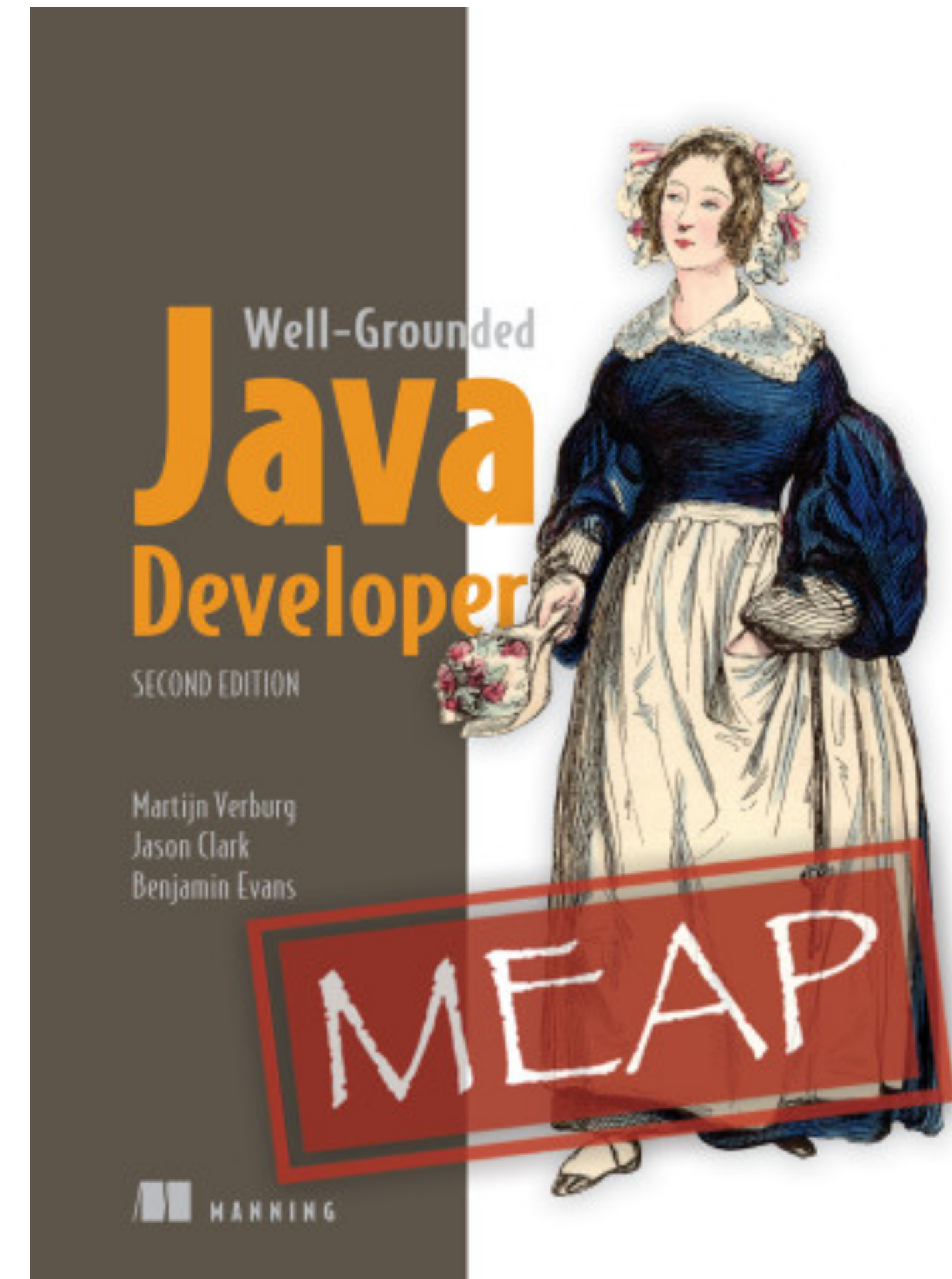
- Streaming Systems - T. Akidau, S. Chernyak & R. Lax
- Optimizing Java - B. Evans, J. Gough & C. Newland
- Reactive Systems in Java - C. Escoffier & K. Finnigan
- Oracle Java Magazine (free subscription)



# Thank You - New Book (& Discount Code)

<https://www.manning.com/books/the-well-grounded-java-developer-second-edition>

- The Well-Grounded Java Developer (2nd Edition)
- Discount code: ctwjbcn22



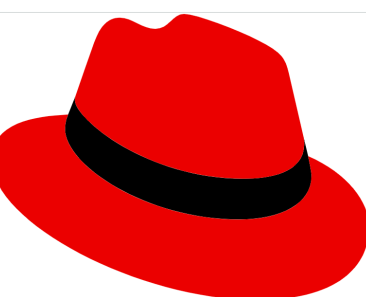
# Case Study: Spring Micrometer

- What is Spring Micrometer?
- Basic Concepts
- Metric Filters
- Demo - Monster Combat



# What Is Spring Micrometer

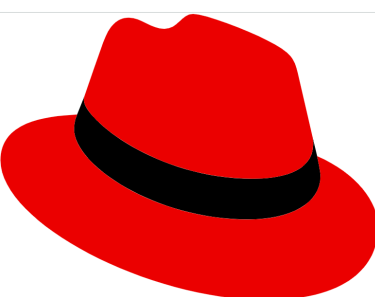
- Vendor-neutral application metrics facade
- Interfaces for instruments with dimensional data model
  - Timers (regular and long)
  - Gauges
  - Counters
  - Distribution summaries (histograms)
- Export to vendor & OSS solutions





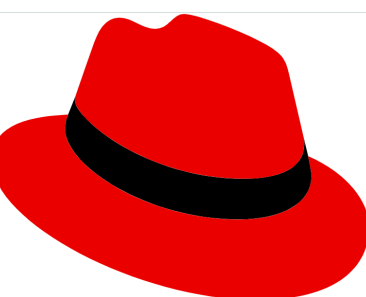
# Facading Over Metric Consumer Systems

- Micrometer provides a core (as an SPI) & module per consumer
- Handles Key Differences Between Systems
- Dimensionality
  - Does the consumer support key / value annotation of the measurement?
  - Other alternative is hierarchical
- Aggregation Discipline
  - Client-side - discrete samples converted to a rate before pub
  - Server-side - aggregation occurs at server
- Publishing



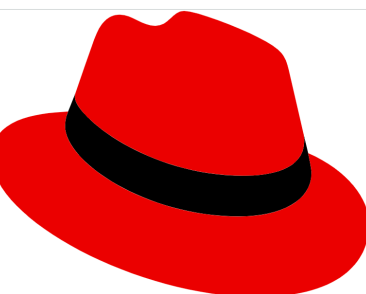
# Basic Concepts - Meters & Registries

- Meter - interface for collecting metrics
- MeterRegistry - creates & holds Meter objects
  - Each consumer has a specific registry
  - SimpleMeterRegistry - in-memory only for experimenting / dev
  - CompositeMeterRegistry - holds multiple registries (multi-pub)
  - Metrics.globalRegistry - static global registry
- Meters named all-lowercase with dot separators
  - Automatically mapped to conventions of consumer



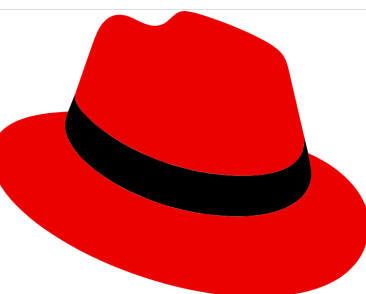
# Basic Concepts - Meters & Instruments

- Common Instrument types
  - Counter - count of all events
  - Gauge - single metric value
  - Timer - count & total time of all timed events
  - LongTaskTimer - long running events that need updating data
  - DistributionSummary - tracks the distribution of non-timed events
- Less Common
  - TimeGauge, FunctionCounter & FunctionTimer
- Dimensions represented as Tag objects
  - Tags also named as dotted lower-case (with non-null values)



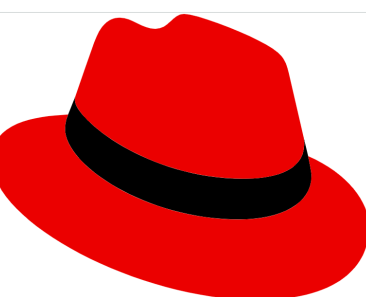
# Metric Filters

- Provide greater control over
  - How & when meters are registered
  - What kinds of statistics they emit
- Three basic functions
  - Deny / Accept meters being registered
  - Transform meters (change name, tags, units)
  - Configure distribution statistics (appropriate meter types)



# Docker & Podman

- Docker is several things
  - Most popular container format
  - Daemon to build & run containers
  - Desktop software to develop with
  - The company that builds the above
- Not all of the software is OSS
- Docker Inc. have decided to start charging license fees
- Podman is a fully OSS replacement - free forever
  - Developed by Red Hat
  - Available for Linux, Mac & Windows



# Basic Podman (Docker) commands

- `build` - make a container from a Dockerfile
- `run` - run the container's main command
- `exec` - run another command on a container
- `ps` - list containers (-a for all)
- `commit` - duplicate a container to another ID
- `cp` - copy files to or from a container
  - Will discuss ports etc later

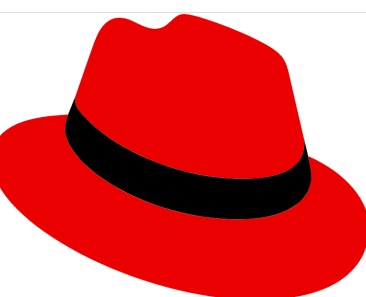


# Micrometer Instrumentation

# DEMO

`https://github.com/ebullient/monster-combat`

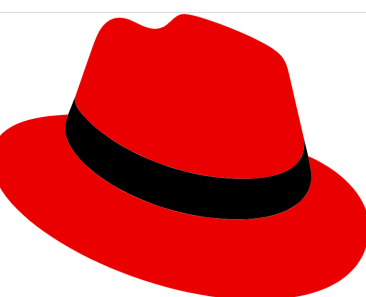
Thanks: Erin Schnabel





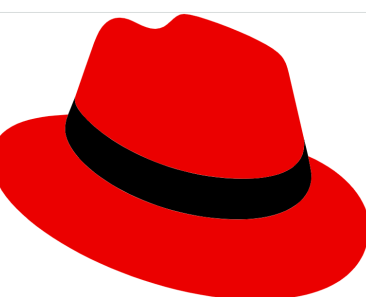
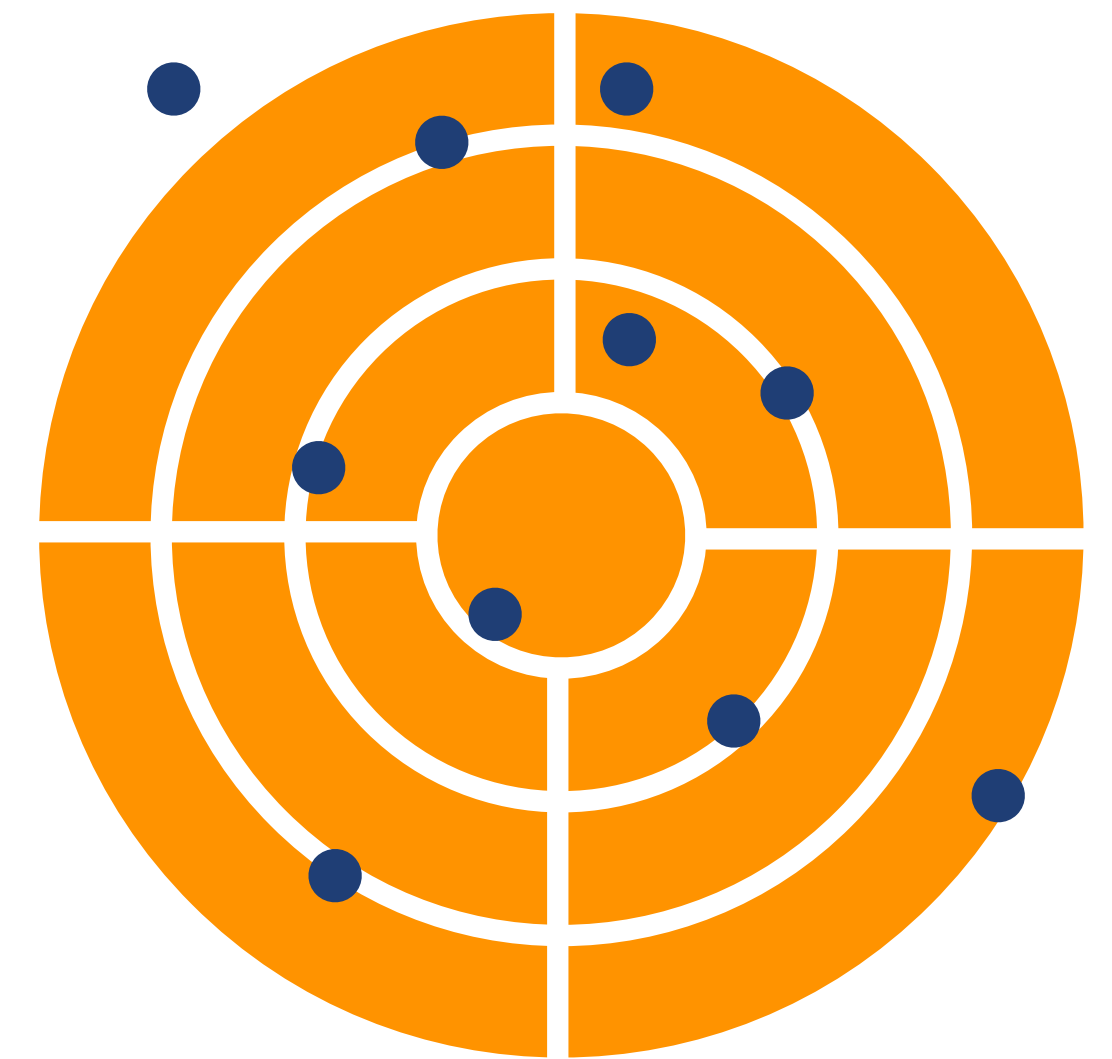
# Statistics & Data Handling

- Humans are poor at guessing
  - Measurements can be subjective
  - Especially Time measurements
- We all have cognitive biases
  - Especially Confirmation Bias
- Best tool against cognitive biases is data
  - Data can overwhelm
  - Patterns aren't always easy to spot by eye



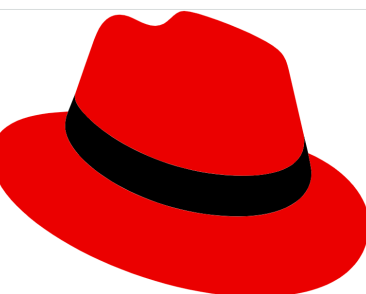


# Systematic and Random Error



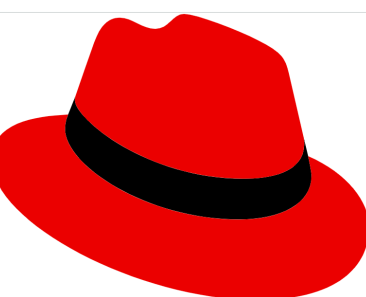
# Know Basic Statistics

- Everyone should know:
  - Mean
  - Mode
  - Percentiles
  - Probability distributions
- Sometimes useful
  - Standard Deviation (be careful)
  - Significance Levels
  - Central Limit Theorem
  - p-values



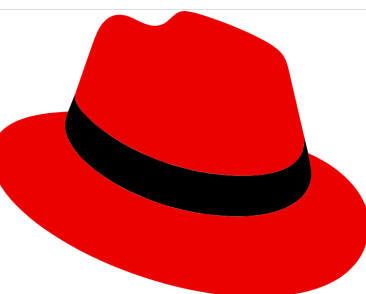
# Statistics Summarize Data Distributions

- Real datasets are approximated by theoretical distributions
- Statistical measures (average, variance, etc) summarize data
- Information is always lost in this process
- Usefulness of various different stats depends on the distribution
- Normally-distributed statistics
  - Easy and familiar to many
  - BUT aren't a very good model for software performance
    - Especially standard deviation



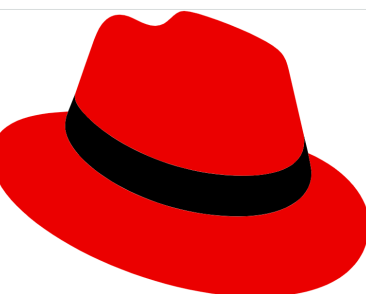
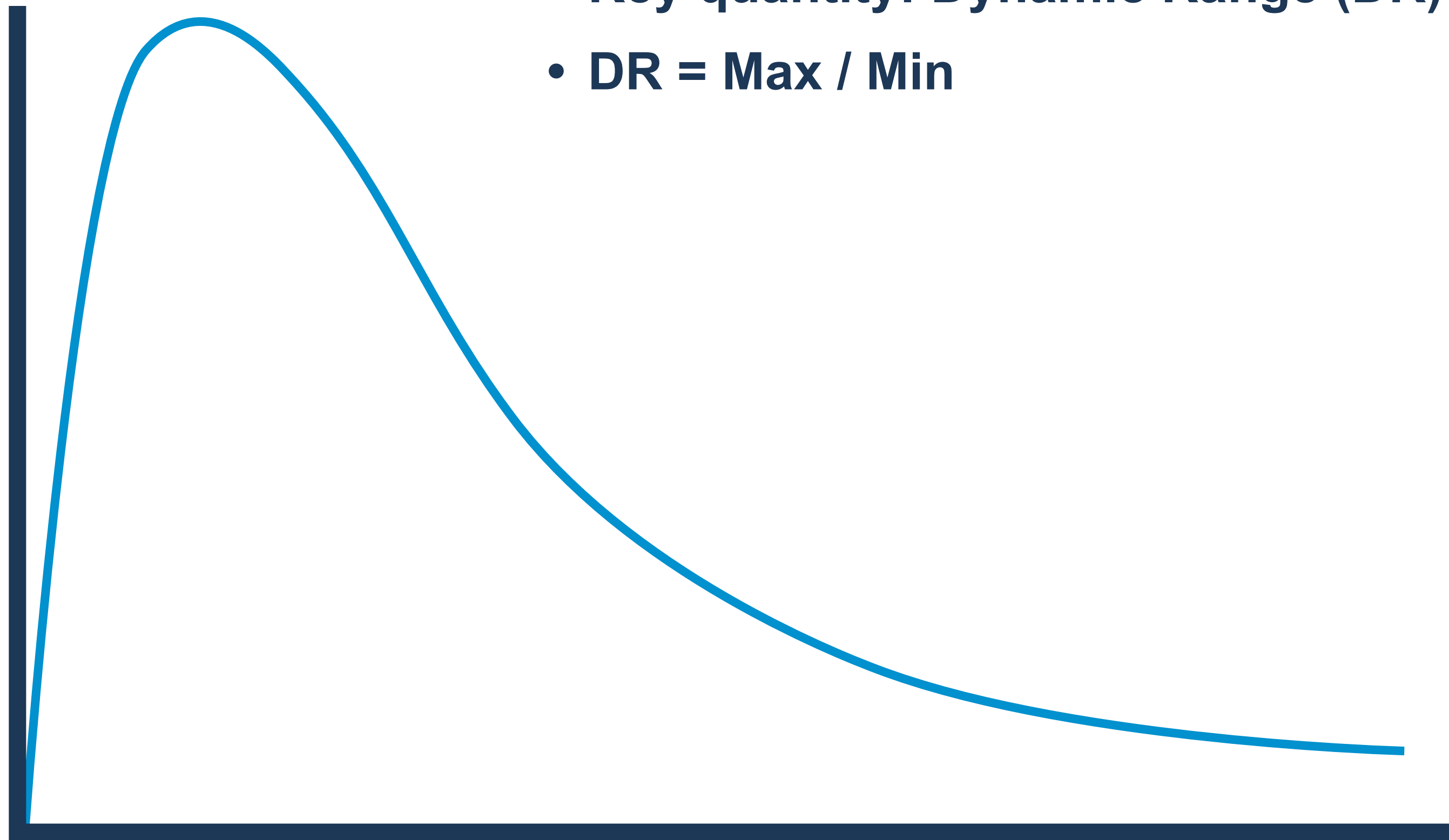
# Non-Normal Statistics

- Real data is often not normally distributed
- JVM applications have a “hot path” where everything works
  - Deviations from the path add latency
  - Latency  $\gg$  random error (& latency is never negative)
- Gives rise to a “long tail” distribution
  - Technically, a specific kind of Gamma distribution
  - Important information is contained in the tail



# Gamma distribution

- Key quantity: Dynamic Range (DR)
- $DR = \text{Max} / \text{Min}$



# Long-tail Percentiles

- One useful technique is “long-tail percentiles”

- Compensates for the high dynamic range

- Example

- Getter method timing

50.0% level was 23 ns

90.0% level was 30 ns

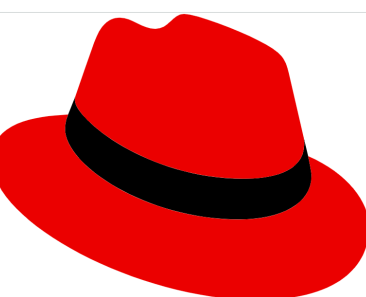
99.0% level was 43 ns

99.9% level was 164 ns

99.99% level was 248 ns

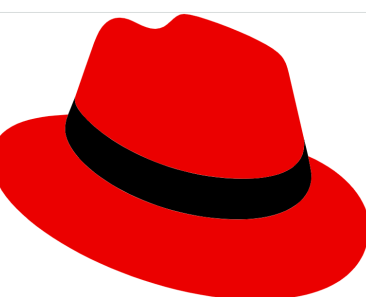
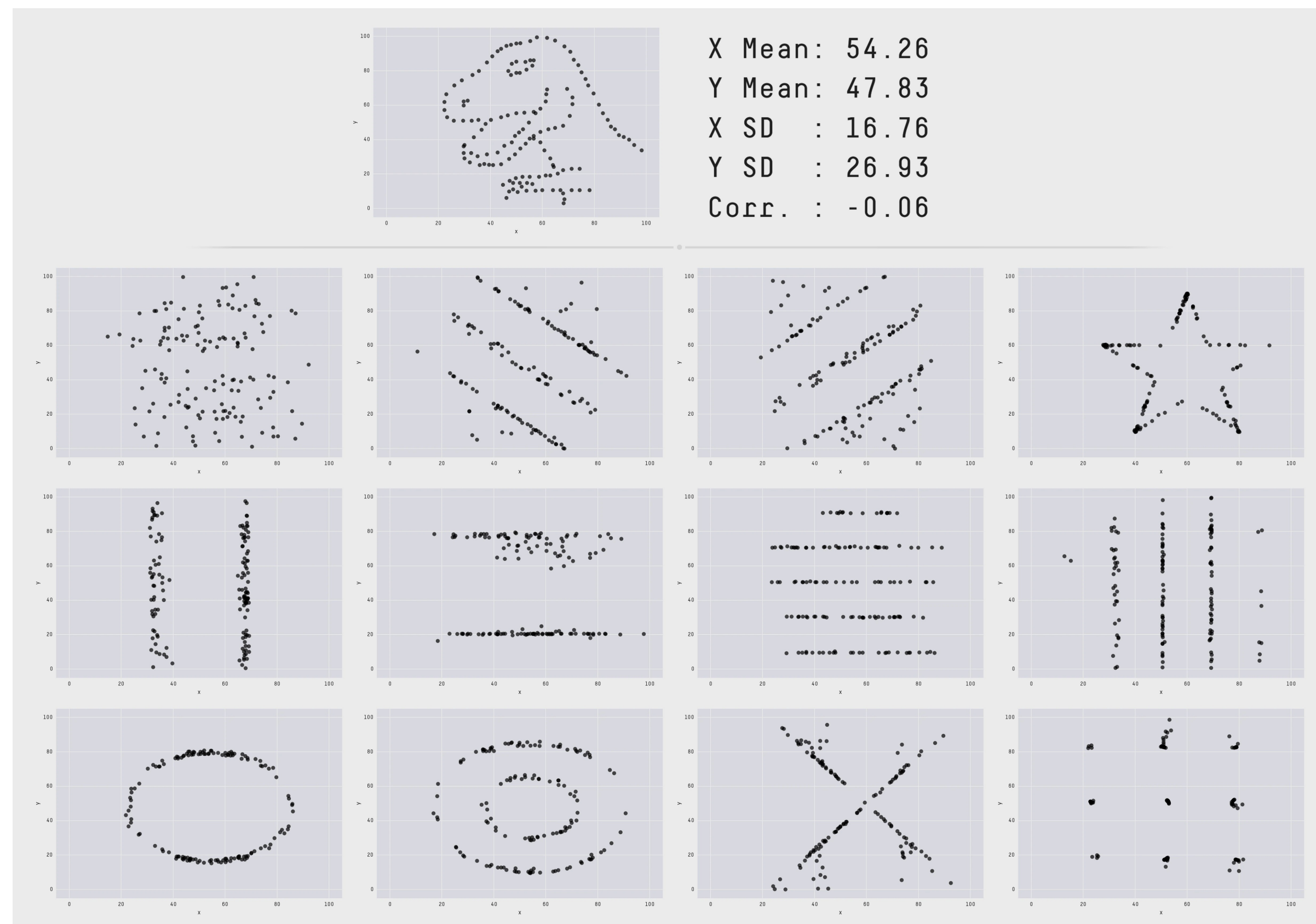
99.999% level was 3,458 ns

99.9999% level was 17,463 ns



# Useful Blog Posts

- “Statistics for Software” - M. Hashemi (Paypal)
- “Same Stats, Different Graphs” - J. Matejka & G. Fitzmaurice (Autodesk)



# What is JFR?

- A profiling tool to gather diagnostics & profiling data
  - From an in-flight Java application running in Hotspot
- Proprietary tool in old Java 8, OSS in Java 11 & 8u262
- Low overhead
  - Oracle claim ~1% impact to steady state performance
  - Observed impact ~3% for a reasonable profile
  - Custom profiles can be created
- GUI console available - Mission Control (JMC)



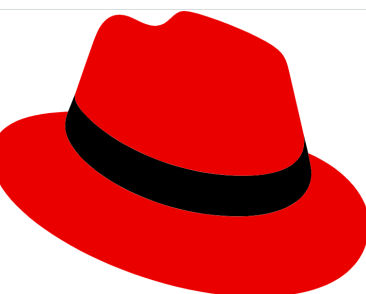


# Using Flight Recorder

- JFR is started with a command line flag
- Generates an output file

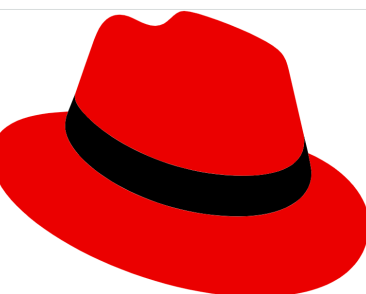
```
java -XX:StartFlightRecording=duration=200s,filename=flight.jfr Klass
```

- Can be challenging to work with in containers
- A streaming solution exists (Java 14+)



# JFR Event Streaming

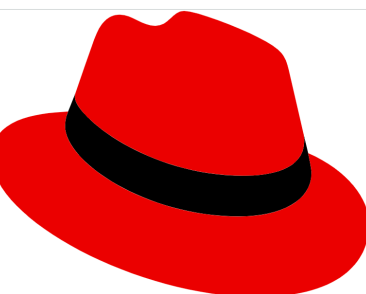
- Java 14 brought JFR Event Streaming
  - A new usage mode for JFR
  - Java 17 is only LTS with Streaming
- API lets programs receive callbacks for JFR events
  - Can respond to them immediately
  - Needed for OTel (due to time window restrictions)
- One obvious way to use this is as a Java Agent



# Example JFR Java Agent

```
public class AgentMain implements Runnable {
    public static void premain(String agentArgs, Instrumentation inst) {
        try {
            Logger.getLogger("AgentMain").log(Level.INFO, "Attaching JFR Monitor");
            new Thread(new AgentMain()).start();
        } catch (Throwable t) {
            Logger.getLogger("AgentMain").log(Level.SEVERE, "Unable to attach JFR Monitor", t);
        }
    }

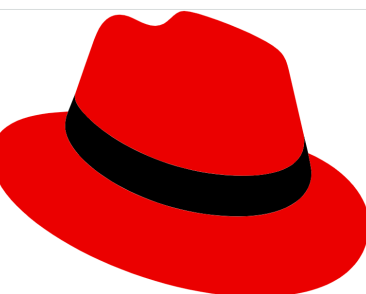
    public void run() {
        var sender = new JfrStreamEventSender();
        try (var rs = new RecordingStream()) {
            rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
            rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
            rs.onEvent("jdk.CPULoad", sender);
            rs.onEvent("jdk.JavaMonitorEnter", sender);
            rs.start();
        }
    }
}
```



# Auto Instrumentation (Metrics)

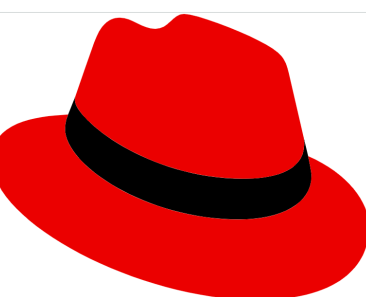
# DEMO

<https://github.com/open-telemetry/opentelemetry-java-contrib/>



# JFR & Open Instrumentation

- JFR is key piece of the ecosystem - not all of it
  - Part of the pivot towards Open Instrumentation
  - JFR can be bridged to OpenTracing and other OSS tools
- JFR-based solution for OpenTelemetry metrics
- Hoping to define a standard set of JVM metrics
  - Useable by any Java implementation of OTel



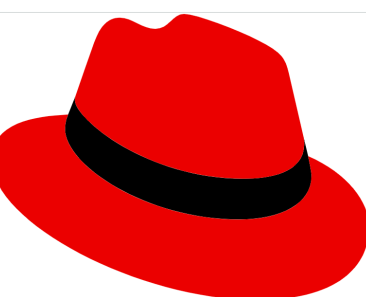
# Agenda

- What's JFR & Why Do We Need It?
- Command Line
- JMC
- Programmatic
- Automated
- The Future

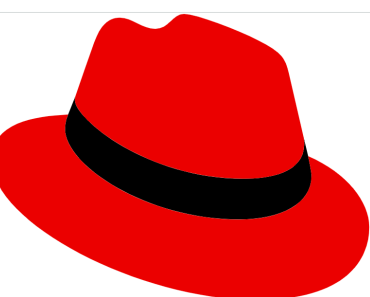
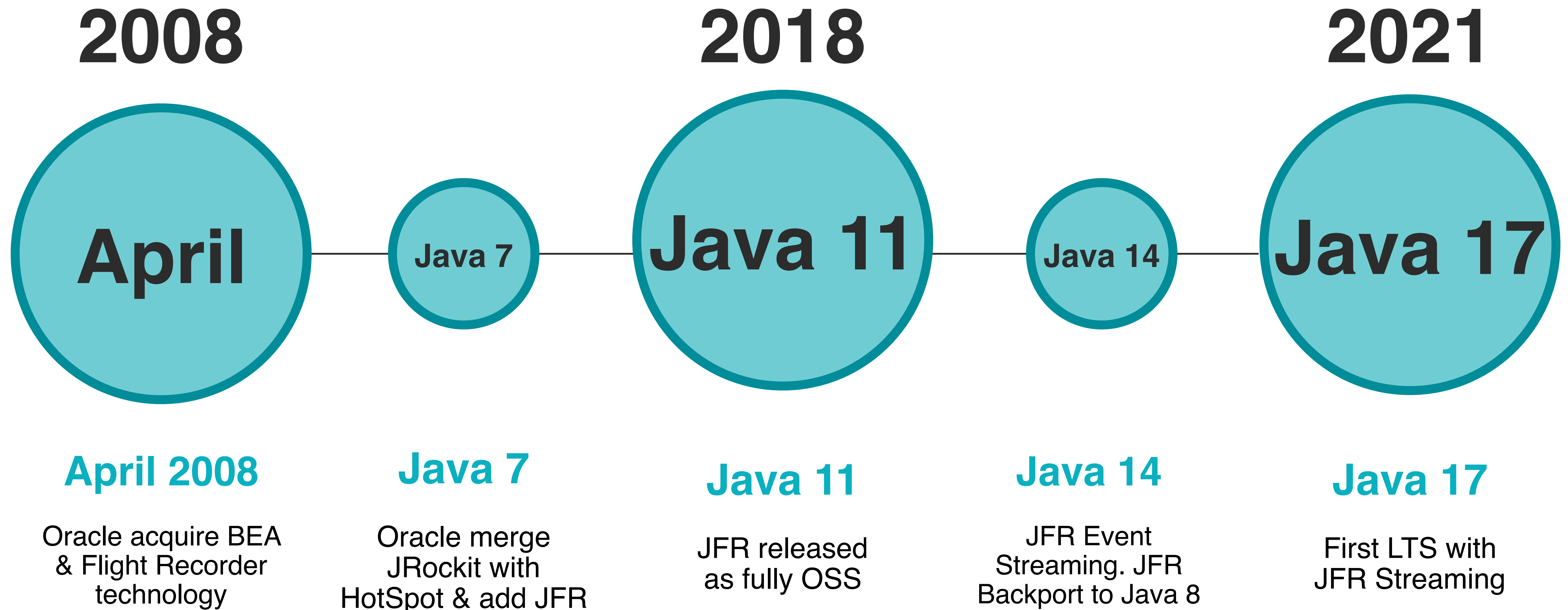


# What's JDK Flight Recorder (JFR)?

- A profiling tool to gather diagnostics & profiling data
  - From an in-flight Java application
- Event-based, many different types of event
- Low overhead
  - Oracle claim ~1% impact to steady state performance
  - Observed: ~3-5% (depending on sensitivity)
- Hotspot-specific technology
  - Directly integrated into the core of the VM
  - Some work being done to make it work with GraalVM



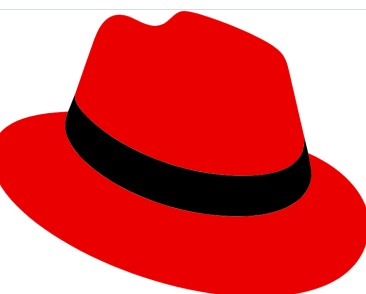
# History & How We Got Here





# Introducing JDK Flight Recorder

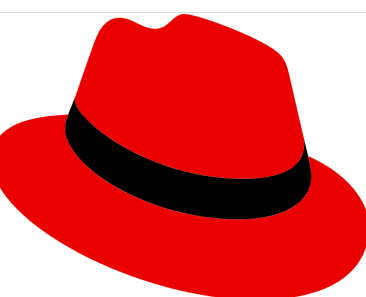
- JFR is started in various ways
  - With a command line flag
  - Dynamically at runtime
  - Via JMX
- Generates an output file
  - High-performance binary file
- Ships with 2 pre-configured profiles
  - Called default and profile (XML configs)
  - Can also create a completely custom profile



# Why?

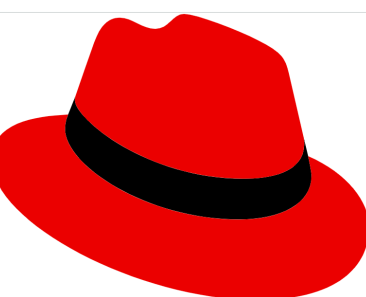
- Cloud-Native is increasingly our reality
- 70% of apps are containerized (& rising fast)
  - Caveat - depends on market segment & maturity
- Increasing Complexity of Microservice Architectures
- Observability has grown out of APM & Monitoring segments
  - Old approaches are not suitable for Cloud Native

<https://newrelic.com/resources/report/2022-state-of-java-ecosystem>

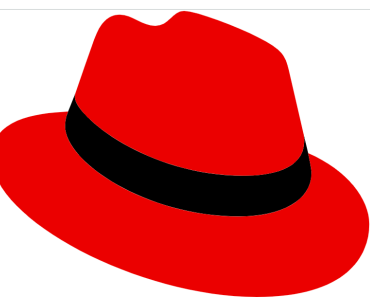
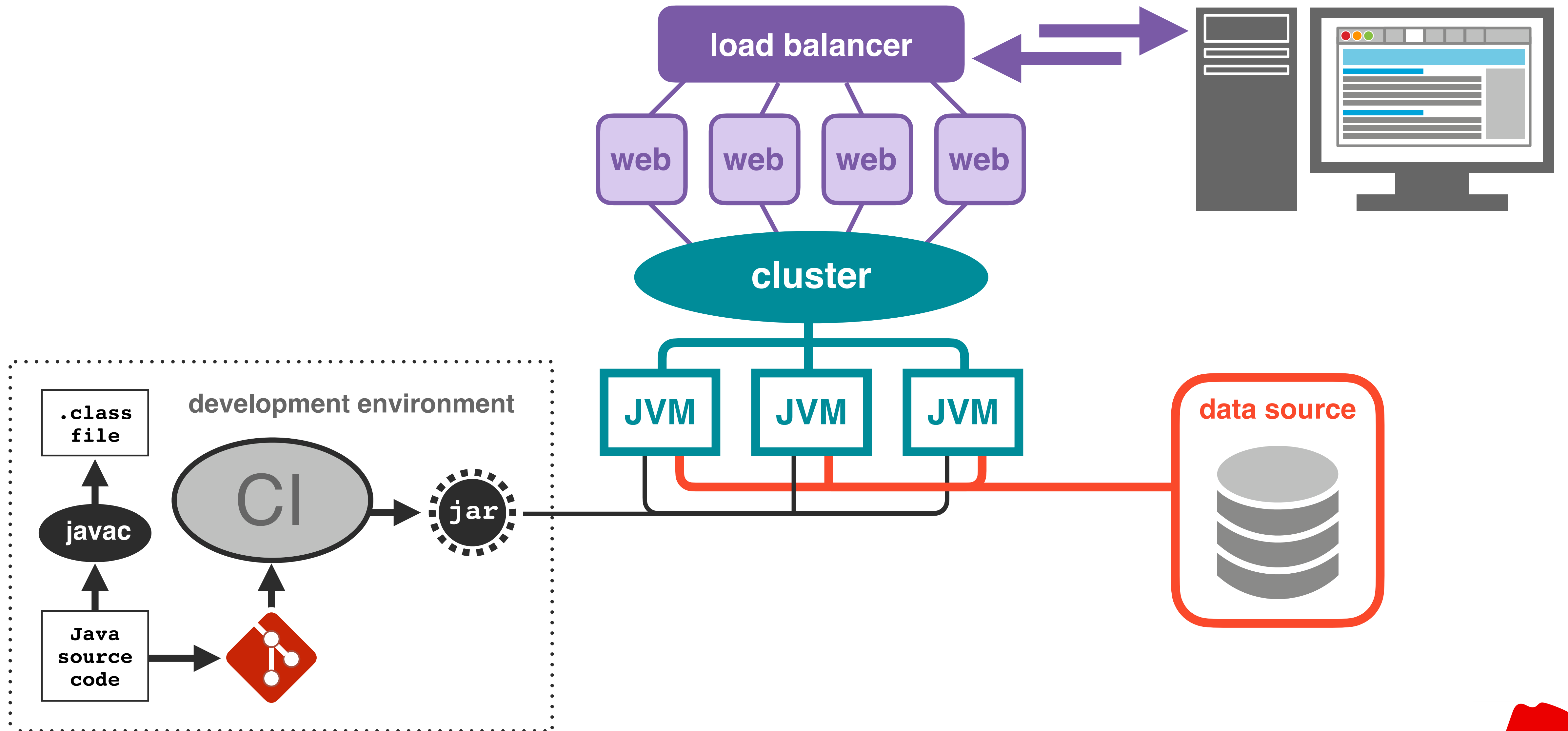


# History of APM

- Application Performance Monitoring (APM) has ~15 year history
- Different world
  - Release cycles measured in months, not days
  - Monoliths, not microservices
  - En premises, not cloud-native
- Manual & semi-manual instrumentation
- Relatively simple architectures
  - Allowed ops teams to develop intuition for failure modes

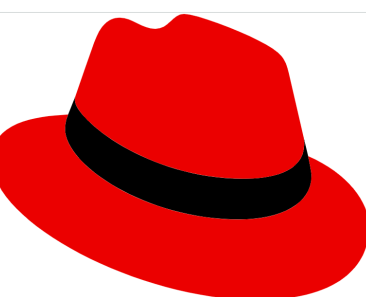


# Old School Architecture



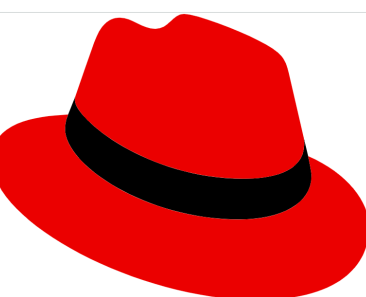
# How Do We Understand Cloud-Native Apps?

- Modern apps are much more complex
  - More services & components
  - More complex topology
  - More sources of change & more rapid change
- New technologies with new behaviours
  - Dynamically scaling services
  - Container environments
  - Kubernetes
  - Function-as-a-Service
  - Kafka



# What is Observability?

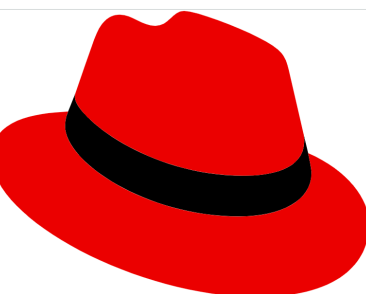
- Conceptually simple
  - Instrument systems & applications to collect observability data
  - Send this data to external system that can store & analyze it
  - Provide insights (for devops, management, SREs) into systems
  - Get answers to questions that you didn't know you'd have to ask
- System control theory
  - How well can internal state of a system be inferred from outside?
- Actionable insights from the entire system
  - Not just one piece
    - Tell you the overall health



# JFR Flags & Command Line Tools

```
-XX:StartFlightRecording=\  
    disk=true,\  
    dumponexit=true,\  
    filename=recording.jfr,\  
    maxsize=1024m,\  
    maxage=1d,\  
    settings=profile,\  
    path-to-gc-roots=true
```

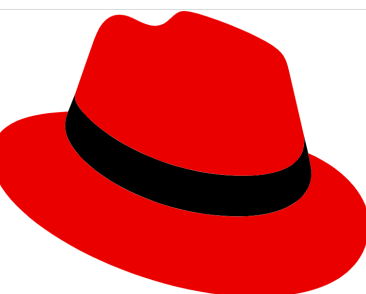
```
java -XX:+FlightRecorder  
    -XX:StartFlightRecording=duration=200s,filename=flight.jfr Klass
```



# Using jcmd

- The Java command - jcmd can be used to control JFR
- Can start and stop
- Dump a current snapshot

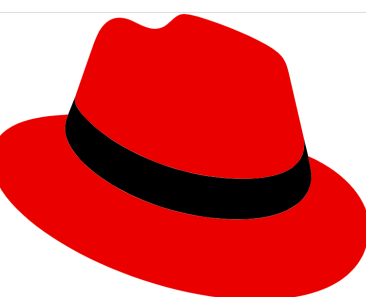
```
$ jcmd <pid> JFR.start name=Recording1 settings=default  
$ jcmd <pid> JFR.dump filename=recording.jfr  
$ jcmd <pid> JFR.stop
```





# Best Practices

- Use JFR as a “ring buffer”
- Use jcmd to dump the file as required
- Ssh in & dump the buffer
  - Allows you to “go back in time”
- JFR command-line tooling
  - Jfr command



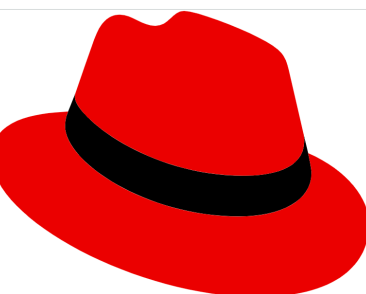
# JFR Command Line

# DEMO



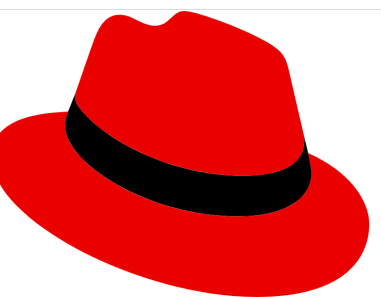
# JDK Mission Control

- A graphical tool used to display JFR output data
  - JFR files are dense, binary files
  - No official specification of the file format
  - JMC has its parser (uses OSGi)
  - Other parsers exist
- JMC originally bundled with Oracle JDK download
- Now a separate project: <https://jdk.java.net/jmc/>



# JDK Mission Control

# DEMO




# Starting Flight Recorder From JMC

Start Flight Recording

Start Flight Recording

Edit recording settings and then click Finish to start the flight recording.



Filename:

flight\_recording\_180144greymalkinMain47117.jfr

Browse...

Name:

My Recording

☐ Time fixed recording

Recording time: 1 min

☒ Continuous recording

Maximum size:

Maximum age: 1h

Event settings:

Profiling - on server

Template Manager

Description:

Low overhead configuration for profiling, typically around 2 % overhead.

Note:

Continuous recordings will need to be dumped to access the data. Right-click on the

Tip:

See the [Recording Wizard Help](#) for more information.

< Back

Next >

Cancel

Finish




# JMC Profiling Options

Start Flight Recording

Event Options for Profiling

Change the event options for the flight recording.



Oracle JDK

Garbage Collector:

Normal

Compiler:

Detailed

Method Sampling:

Maximum

Thread Dump:

Every 60 s

Exceptions:

Errors Only

Synchronization Threshold:

10 ms

File I/O Threshold:

10 ms

Socket I/O Threshold:

10 ms

☐ Heap Statistics

☐ Class Loading

☒ Allocation Profiling

< Back

Next >

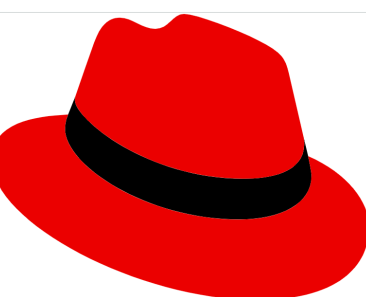
Cancel

Finish



# Programmatic JFR

- JFR ships a programmatic API
  - For working with JFR files
- In jdk.jfr module
  - NOT a java.\* module
  - Implementation-dependent (i.e. Hotspot only)
- Parse files
  - Read individual events
  - Handle common JFR data types



# JFR File Parsing

```
var recordingFile = new RecordingFile(Path.of(fileName));
while (recordingFile.hasMoreEvents()) {
    var event = recordingFile.readEvent();
    if (event != null) {
        var details = decodeEvent(event);
        if (details == null) {
            // Log a failure to recognise details
        } else {
            // Process details
            System.out.println(details);
        }
    }
}
```

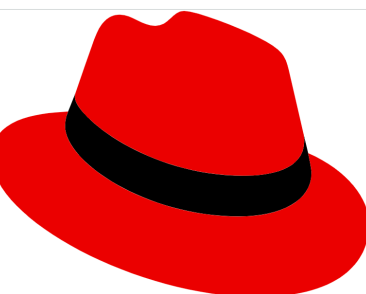




# Programmatic JFR

# DEMO

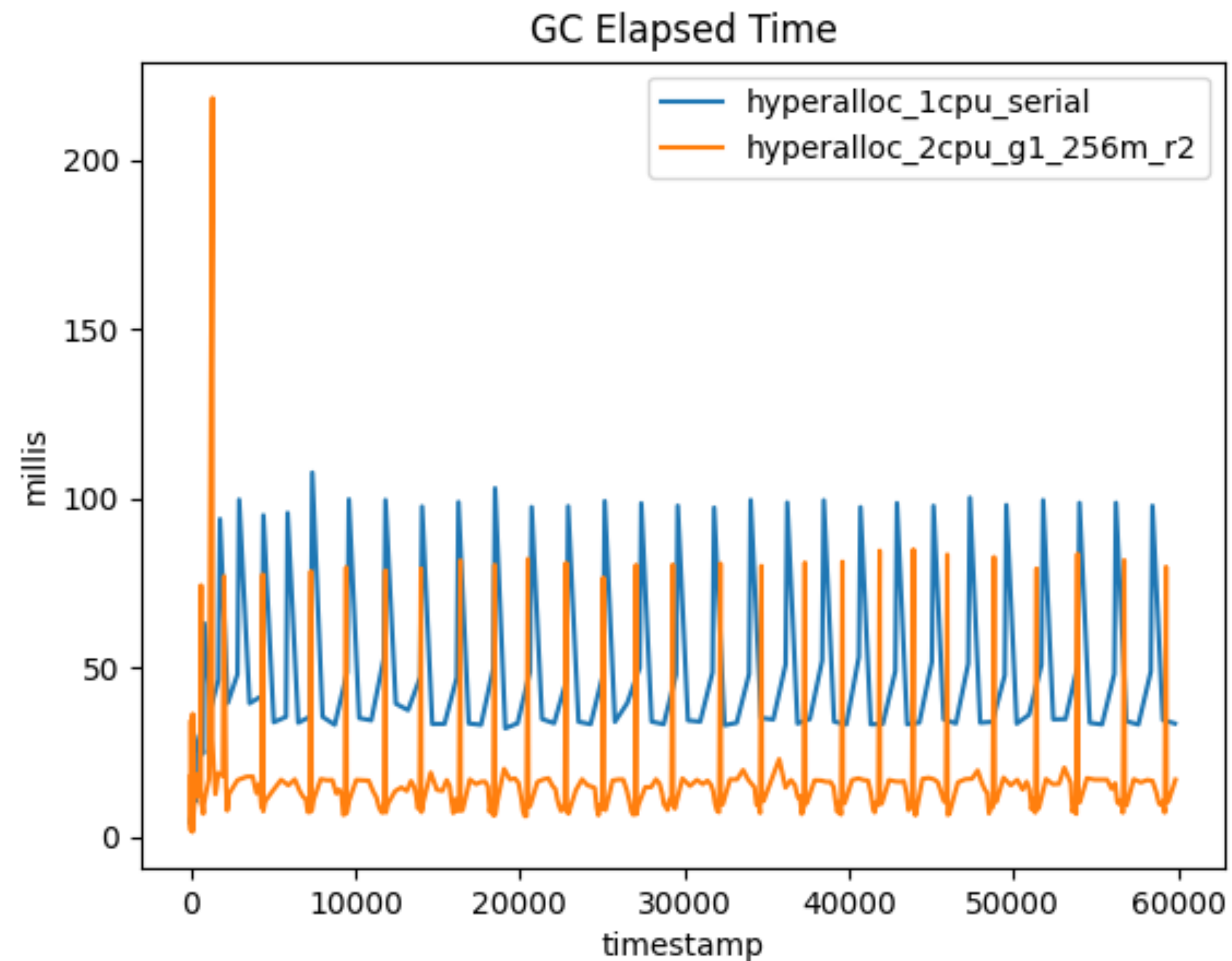
<https://github.com/kittylust/jfr-hacks>



# JFR Analytics

- OSS Project
  - Provides query interface to JFR data sources
  - Write in an SQL dialect to extract events
  - Gunnar Morning (Red Hat)
- Relies on Apache Calcite
- Source code: <https://github.com/moditect/jfr-analytics>

# JFR Analysis - Example

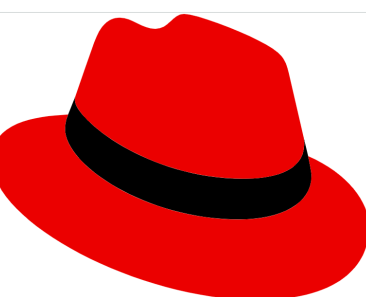


<https://developers.redhat.com/articles/2022/04/19/best-practices-java-single-core-containers>



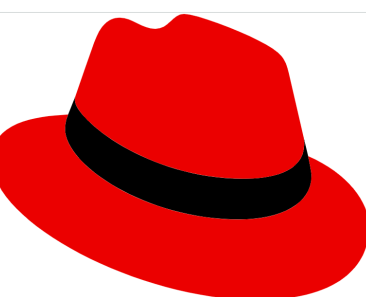
# DEMO

`https://github.com/kittylust/jfr-hacks`

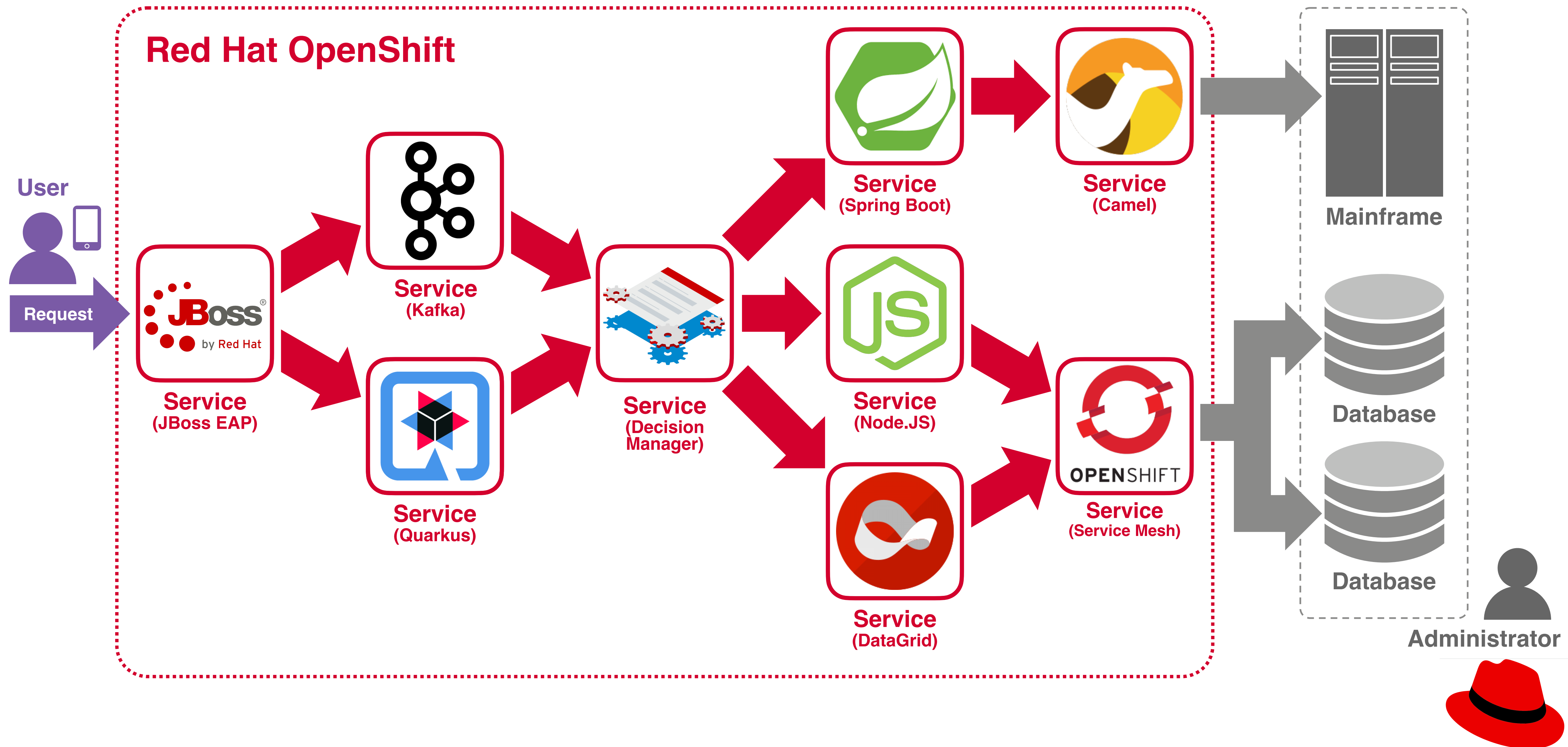


# Automated

- File handling is geared towards debugging a single VM
- Awkward for monitoring / Observability
  - Ideally want a stream of telemetry data
- Solutions
  - On-demand recordings (Cryostat - Red Hat)
  - Create a pseudo stream (New Relic)
  - Send Sequence of chunks (Datadog)
  - JFR Streaming (Java 17 only :( )



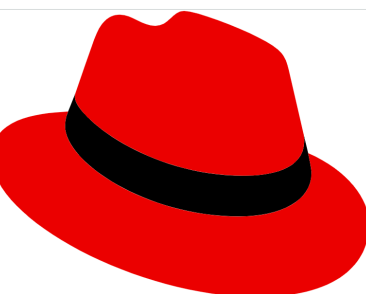
# Distributed System running on OpenShift



# Automated JFR - Pseudo-Stream

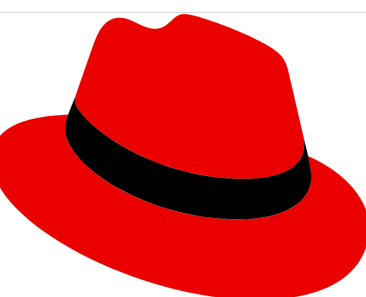
# DEMO

<https://github.com/newrelic/newrelic-jfr-core/>



# The Future

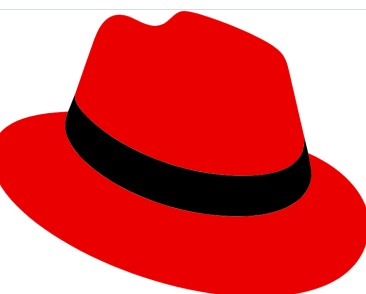
- JFR continues to evolve
- New event types
  - E.g. Detect value-based classes
- JFR as an Observability data source
  - OpenTelemetry
- Profiling is a major new area of interest
  - Datadog profiler
- Look at Java 17 adoption rates
  - JFR Streaming





# JFR Event Streaming

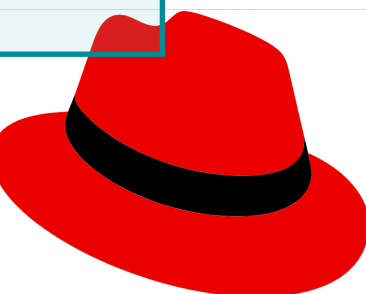
- Java 14 introduced JFR Event Streaming
  - A new usage mode for JFR
  - Java 17 is the first LTS with streaming capability
- API lets programs receive callbacks for JFR events
  - Can respond to them immediately.
- One obvious way to use this is as a Java Agent
  - Start a background thread to receive events



# Example JFR Java Agent

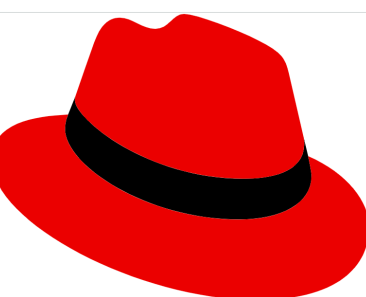
```
public class AgentMain implements Runnable {
    public static void premain(String agentArgs, Instrumentation inst) {
        try {
            Logger.getLogger("AgentMain").log(Level.INFO, "Attaching JFR Monitor");
            new Thread(new AgentMain()).start();
        } catch (Throwable t) {
            Logger.getLogger("AgentMain").log(Level.SEVERE, "Unable to attach JFR Monitor", t);
        }
    }

    public void run() {
        var sender = new JfrStreamEventSender();
        try (var rs = new RecordingStream()) {
            rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
            rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
            rs.onEvent("jdk.CPULoad", sender);
            rs.onEvent("jdk.JavaMonitorEnter", sender);
            rs.start();
        }
    }
}
```



# Event Filtering

- JFR API provides basic filtering
  - reduce the number of events that callbacks process
- Filter Types
  - Enabled - should the event be recorded at all
  - Threshold - duration below which an event is not recorded
  - Stack trace - if stack trace from `Event.commit()` should be recorded
  - Period - interval at which the event is emitted, if periodic



# The Three Pillars of Observability



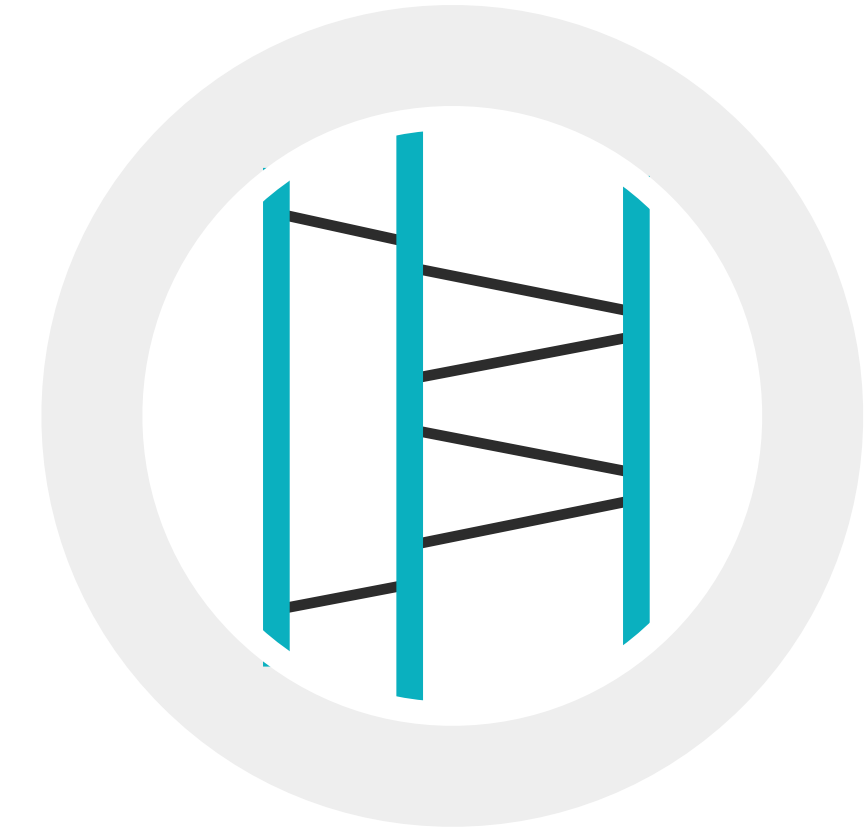
## Metrics

Numbers describing a particular process or activity measured over intervals of time



## Logs

Immutable record of discrete events that happen over time



## Traces

Data that shows which line of code is falling to gain better visibility at the individual user level for events that have occurred

