Observability

Overview

Observability focuses on the development of an application or system, and the rich instrumentation you need, not simply to poll and monitor it for thresholds or defined health checks, but to support the ability to ask questions about how the software or system works.

A system is appropriately observable when instrumented sufficiently such that any question required to support it at the level of quality required of it, can be answered just from observing it from the outside. This includes not having to add new instrumentation to get your new question answered. Adding new instrumentation isn't wrong (quite the opposite! Instrument as needed in order to get the level of observability required to support the service), but your system isn't appropriately observable until you don't have to add more to ask anything you need to know about it. Observability isn't about mindlessly laboring over instrumenting every CPU instruction, every mouse move the user makes, or how long it takes to add two numbers together, it is about understanding what level of service the system needs to deliver, and instrumenting to meet those requirements.

All DFS systems and applications should follow the pillars of the MELT model, the utilization of Metrics, Events, Logs and Traces to support the observability objectives in a given domain. Each of these characteristics is detailed in the sections below.

Observability vs. Monitoring

**Monitoring**, what we usually talk about, is a verb; something we perform against our applications and systems to determine their state. From basic fitness tests and whether they’re up or down, to more proactive performance health checks. We monitor applications to detect problems and anomalies. As troubleshooters, we use it to find the root cause of problems and gain insights into capacity requirements and performance trends over time.[2]

**Observability**, on the other hand, is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. So, in contrast to monitoring - which is something we actually do - observability (as a noun), is more a property of a system. As such, we need to be thoughtful of what can be observed that then can be monitored. If IT systems and applications don’t adequately externalize their state, then even the best monitoring can fall short. [2]

The CL Playbook for Observability

* **Metrics**
  + Latency
  + Traffic
  + Error Rate
  + Saturation
* **Events**
  + State Change
  + Security Events
  + Administrative Events
* **Logs**
  + Running Configuration
  + Request/Response Detail
  + Request/Response Payload
  + Functional failures
* **Traces**
  + An operation name
  + A start timestamp and finish timestamp
  + A set of key:value span Tags
  + A set of key:value span Logs
  + A SpanContext

Metrics

Metrics are a numerical representation of data that can be used to determine a service or component’s overall behavior over time. Metrics comprise a set of attributes (e.g., name, value, label, and timestamp) that convey information about SLAs, SLOs, and SLIs.

Metrics are a measured value derived from system performance. Metrics can be readily correlated across infrastructure components to get a holistic view of system health and performance. Metrics are low overhead to collect, inexpensive to store, dimensional for quick analysis, and a great way to measure overall health.

Standard metrics emitted by DFS systems should, at a minimum, conform to the Google SRE Handbook's 'Four Golden Signals:' Latency, Traffic, Error Rate, and Saturation. Depending on the context of the system, there may be many metrics that comprise the complete picture of each of the golden signals. Rarely will a single metric provide sufficient coverage to complete the observability and monitoring requirements for a given signal. Typically, SREs and ops engineers use metrics to trigger alerts whenever a system value deviates from a specified threshold.

Limitations on Metrics

You get a lot of information from metrics in a really compact, cost-effective format. So, why wouldn’t we use metrics all the time? Well, simply put, metrics require careful decision-making. For example, if you knew ahead of time you wanted to know the 50th percentile (median) and the 95th percentile of the metric you’re capturing, you could instrument that, collect it on all of your aggregates, and then graph it. But let’s say you wanted to know the 95th percentile for just the data of a particular item in a component or system. You can’t calculate that after the fact; you would need all the raw sample events to do that. So, for metrics, you must decide ahead of time about how you want to analyze the data and set it up to support that analysis.

Minimum Metric Set for DFS Applications and Infrastructure

All DFS systems and applications should consider SRE's four 'golden signals' as the minimum metric set for observability. The four golden signals are latency, traffic, error rate and saturation. Each signal is detailed below.

**Latency**

The time it takes to service a request. It’s important to distinguish between the latency of successful requests and the latency of failed requests. For example, an HTTP 500 error triggered due to loss of connection to a database or other critical backend might be served very quickly; however, as an HTTP 500 error indicates a failed request, factoring 500s into your overall latency might result in misleading calculations. On the other hand, a slow error is even worse than a fast error! Therefore, it’s important to track error latency, as opposed to just filtering out errors.

**Traffic**

A measure of how much demand is being placed on your system, measured in a high-level system-specific metric. For a web service, this measurement is usually HTTP requests per second, perhaps broken out by the nature of the requests (e.g., static versus dynamic content). For an audio streaming system, this measurement might focus on network I/O rate or concurrent sessions. For a key-value storage system, this measurement might be transactions and retrievals per second.

**Error Rate**

The rate of requests that fail, either explicitly (e.g., HTTP 500s), implicitly (for example, an HTTP 200 success response, but coupled with the wrong content), or by policy (for example, "If you committed to one-second response times, any request over one second is an error"). Where protocol response codes are insufficient to express all failure conditions, secondary (internal) protocols may be necessary to track partial failure modes. Monitoring these cases can be drastically different: catching HTTP 500s at your load balancer can do a decent job of catching all completely failed requests, while only end-to-end system tests can detect that you’re serving the wrong content.

**Saturation**

How "full" a service is. A measure of system fraction, emphasizing the resources that are most constrained (e.g., in a memory-constrained system, show memory; in an I/O-constrained system, show I/O). Note that many systems degrade in performance before they achieve 100% saturation, so having a saturation target is essential.

In complex systems, saturation can be supplemented with higher-level load measurement: can your service properly handle double the traffic, handle only 10% more traffic, or handle even less traffic than it currently receives? For very simple services that have no parameters that alter the complexity of the request (e.g., "Give me a nonce" or "I need a globally unique monotonic integer") that rarely change configuration, a static value from a load test might be adequate. As discussed in the previous paragraph, however, most services need to use indirect signals like CPU utilization or network bandwidth that have a known upper bound. Latency increases are often a leading indicator of saturation. Measuring your 99th percentile response time over some small window (e.g., one minute) can give a very early signal of saturation.

Finally, saturation is also concerned with predictions of impending saturation, such as "It looks like your database will fill its hard drive in 4 hours."

Resources for Metrics

**Metrics On The Web**

[Google: Monitoring Distributed Systems (The 4 Golden Signals)](https://sre.google/sre-book/monitoring-distributed-systems/)

[The Golden Signals in SRE for Microservices](https://www.youtube.com/watch?v=uM9Sqeyp1uo)

[Simplify application monitoring with SRE Golden Signals](https://www.youtube.com/watch?v=rnnhtzIgjvQ)

[Applying Site Reliability Engineering 'Golden Signals' to your Kubernetes Cluster](https://www.youtube.com/watch?v=esyX35J2mq8)

[Monitoring the Golden Signals for Kubernetes](https://www.youtube.com/watch?v=db_YZdElHr0)

[Observability of Distributed Systems (class SRE implements DevOps)](https://www.youtube.com/watch?v=SoZZzB-yTOk)

**DFS Metrics Resources**

[AppDynamics Documentation](https://discoverfinancial.sharepoint.com/sites/DLife/bu/bt/Groups/appdynamics/Pages/default.aspx)

[AppDynamics Workshops](https://discoverfinancial.sharepoint.com/sites/DLife/bu/bt/Groups/appdynamics/Pages/AppDynamics-Events.aspx)

AppDyanmics University: Reach out on the [AppDynamics on Yammer](https://web.yammer.com/main/groups/eyJfdHlwZSI6Ikdyb3VwIiwiaWQiOiIxMTY3NDMwMiJ9/new) site and ask to be registered for the vendor courses.

Events

Conceptually, an event can be defined as a discrete action happening at a moment in time. You can choose which attributes are important to send when defining an event. There’s no hard-and-fast rule about what data an event can contain—you define an event as you see fit. Events are valuable, because you can use them to confirm that a particular action occurred at a particular time. Because events are basically a history of every individual thing that happened in your system, they can be aggregated to answer more advanced questions. Events become more powerful when you add more metadata to them. [7]

Limitations of Events

One may be thinking events sound awesome (“Let’s collect one of everything that happens all the time!”). However, event collection comes with a cost. Every event takes some amount of computational energy to collect and process. They also potentially take up lots of space. So for relatively infrequent things events are great, but one wouldn’t want to collect an event for everything a system or component does. [7]

Guidelines for Events

* Generate one event per service component or service interaction. For example, a single API request should generate a log line or event at the edge (ELB/ALB), the load balancer (Nginx), the API service, each microservice it gets passed off to, and for every interaction with each storage layer. There are other sources of information and events that may be relevant when debugging. For example, your database likely generates metrics detailing the length of the request queue, or other critical internal statistics. One event per 'hop' is the current best practice.
* Wrap any call out to any other service as a timing event. Finding where the system has become slow can involve either distributed tracing or comparing the view from multiple directions. For example, a DB may report that a query took 100ms, but the service may argue that it actually took 10 seconds. They can both be right--if the database doesn’t start counting time until it begins executing the query, and it has lengthy request queue.
* Collect sufficient context: Each event should be as wide as possible, with as many high-cardinality dimensions as possible, as this provides many ways to identify and group as many similar events as possible.

Minimum Event Set for DFS Applications and Infrastructure

The minimum event set for DFS Applications and Infrastructure falls into three categories: state change events, security events and administrative events

**State Change**

* Up / Down
* Degradation\*
* Critical Resource Threshold Breached

**Security Events**

* Access Violations
* Malformed requests
* Critical Security Configuration Changes
* Administrative Access Security Events

**Administrative Events**

* System Configuration Modifications
* System Composition Changes

Resources for Events

**Events On the Web**

[demystifying melt](https://itbrief.com.au/story/demystifying-m-e-l-t-the-key-data-for-business-observability)

[secure event monitoring with influxdb](https://www.influxdata.com/solutions/security-event-monitoring-with-influxdb/)

[The Google workbook on monitoring](https://sre.google/workbook/monitoring/)

**DFS Event Resources**

* [Influx Data](https://discover.service-now.com/runway?id=product&product=1bfe47331be524d05c7411b4bd4bcb20)
* [Argus](https://discover.service-now.com/runway?id=product&product=73c2d81cdbfee810322841a40596199a)
* [Chef Automate](https://chef-automate.discoverfinancial.com/)
* Simple Network Management Protocol (SNMP)
* [Netscout](https://discover.service-now.com/runway?id=product&product=be629f371be524d05c7411b4bd4bcb8a)
* [Solarwinds](https://solarwinds.discoverfinancial.com/)
* Telegraf

Logs

In their most fundamental form, logs are essentially just lines of text a system produces when certain code blocks get executed. Logs are incredibly valuable for troubleshooting applications, databases, caches, load balancers, or older proprietary systems that aren’t friendly to inprocess instrumentation, to name a few.[7] Similar to events, log data is discrete—it’s not aggregated—and can occur at irregular time intervals. Logs are also usually much more granular than events. In fact, one event can correlate to many log lines. [7] Log data is sometimes unstructured, and therefore hard to parse in a systematic way; however, these days you’re more likely to encounter “structured log data” that is formatted specifically to be parsed by a machine. Structured log data makes it easier and faster to search the data and derive events or metrics from the data.

Minimum Log Set for DFS Applications and Infrastructure

* Running Configuration: details about a system's current running configuration
* Request/Response Detail: who or what was accessed, who/what accessed it
* Request/Response Payload: what was submitted in the request, what was provided in the response. ***Note:*** non-public or sensitive request/response payloads **should not** be logged, or only logged with the infromation obfuscated.
* All events listed in the "Minimal Event Set for ABC Applications and Infrastructure"
* Functional failures (if possible)

Understanding Logs

Logs in general come in three forms but are fundamentally the same: a timestamp and a payload of some context. The three forms are:

* Plaintext
  + A log record might be free-form text. This is also the most common format of logs.
* Structured
  + Much evangelized and advocated for in recent days. Typically, these logs are emitted in the JSON format.
* Binary
  + Think logs in the Protobuf format, MySQL binlogs used for replication and point-in-time recovery, systemd journal logs, the pflog format used by the BSD firewall pf that often serves as a frontend to tcpdump.

Debugging rare or infrequent pathologies of systems often entails debugging at a very fine level of granularity. Event logs, in particular, shine when it comes to providing valuable insight along with ample context into the long tail that averages and percentiles don’t surface. As such, event logs are especially helpful for uncovering emergent and unpredictable behaviors exhibited by components of a distributed system.

Failures in complex distributed systems rarely arise because of one specific event happening in one specific component of the system. Often, various possible triggers across a highly interconnected graph of components are involved. By simply looking at discrete events that occurred in any given system at some point in time, it becomes impossible to determine all such triggers. To nail down the different triggers, one needs to be able to do the following:

1. Start with a symptom pinpointed by a high-level metric or a log event in a specific system
2. Infer the request lifecycle across different components of the distributed architecture
3. Iteratively ask questions about interactions among various parts of the system

In addition to inferring the fate of a request throughout its lifecycle (which is usually short lived), it also becomes necessary to be able to infer the fate of a system as a whole (measured over a duration that is orders of magnitudes longer than the lifecycle of a single request).

Traces and metrics are an abstraction built on top of logs that pre-process and encode information along two orthogonal axes, one being request-centric (trace), the other being system-centric (metric).

The Pros and Cons of Logs

Logs are, by far, the easiest to generate. The fact that a log is just a string or a blob of JSON or typed key-value pairs makes it easy to represent any data in the form of a log line. Most languages, application frameworks, and libraries come with support for logging. Logs are also easy to instrument, since adding a log line is as trivial as adding a print statement. Logs perform really well in terms of surfacing highly granular information pregnant with rich local context, so long as the search space is localized to events that occurred in a single service.

The utility of logs, unfortunately, ends right there. While log generation might be easy, the performance idiosyncrasies of various popular logging libraries leave a lot to be desired. Most performant logging libraries allocate very little, if any, and are extremely fast. However, the default logging libraries of many languages and frameworks are not the cream of the crop, which means the application as a whole becomes susceptible to suboptimal performance due to the overhead of logging. Additionally, log messages can also be lost unless one uses a protocol like RELP to guarantee reliable delivery of messages. This becomes especially important when log data is used for billing or payment purposes.

Evolving beyond traditional log monitoring

Modern software teams need log management solutions that deliver performance and analysis capabilities that create a foundation for observability—all within a solution designed to support full-stack observability across all telemetry types. What are the key elements to include when looking for a modern log management solution?

* A platform that supports the ever-expanding open a source market to ensure what you build today exists collaboratively within an ecosystem of technologies and will work with future technologies that may not even exist yet\_
* A single unified platform for all telemetry data to ensure log data can be enriched and correlated with other key data types to provide a comprehensive view of the entire ecosystem and health\_
* The ability to store key metrics about log details so that aggregate trends over time are available without needing to crunch massive data volumes from scratch again and again\_
* The choice of anomaly detection to use AI so teams can transition from reactive to proactive detection as they evolve on their journey\_
* The option to automatically combine log-level detail with curated content for metrics, events, and traces to help you understand the health and performance of your entire software stack\_
* High availability: In cloud environments, that might mean building log management that is deployed across multiple availability zones\_
* Scalability: Many traditional log solutions are bound by limited compute, storage, or a stitched together patch quilt of varied technologies on different platforms, all of which require meticulous behind-the-scenes management and additional overhead\_
* Parsing: the ability to turn log information into meaningful data\_
* Aggregation: the ability to record and transfer logs from several data sources regardless of where they originated\_
* Storage: the ability to store any type of data securely for extended periods, both for original log data and for high-level metrics when appropriate, enabling a broader set of use cases and trend recognition across the board\_
* Analysis: the ability to dissect data by querying and visualizing it\_
* Alerting: the ability to be notified in real time when an incident is taking place and the option of enabling more sophisticated alerting to reduce fatigue\_

With the abovementioned, teams can benefit from modern log management on their path to observability. Observability is about providing access to all of the data you need, making it easy to query, analyze, dig into details, and provide meaningful alerts that tell a story rather than generating message upon message without context. This ultimately leads to better application performance, minimized downtime, and improved user satisfaction.

Resources for Logs

**Logs On the Web**

[Logs and Metrics and Traces, Oh My!](https://www.youtube.com/watch?v=O0XNSU-I-sg)

[Real-Time Log Analytics using Splunk | Basic Searching Log File | Splunk Training | Intellipaat](https://www.youtube.com/watch?v=EPfz82XRQFk)

[Observability Explained with LogDNA](https://www.youtube.com/watch?v=bvVgP4tw_Hc)

**DFS Logs Resources**

**CLS**

[CLS HELP (Chat Teams Forum to assist any question you may have)](https://teams.microsoft.com/l/channel/19%3a686d3860882447f69fbad98979e8805a%40thread.skype/CLS%2520Help?groupId=bdfd4785-040d-402d-a90d-851c1d5949b0&tenantId=f3f068cf-080c-4824-a912-f8c4633bd454)

CLS Office Hours - Tuesday's 9-10am CT.

[For hands-on assistance with our on-call developer, click here](https://discover.webex.com/discover/j.php?MTID=m2b3ed0f0f05cc84607e1a960db94a22a)

**SPLUNK**

Today, Splunk Enterprise is used by our Cybersecurity department to collect, analyze and respond to potential security incidents.

The Splunk documentation can help provide a better understanding of the product and the logs that enable the ability to query the data to generate meaningful information:

[Overview of Splunk Enterprise](https://docs.splunk.com/Documentation/Splunk/8.2.1/Overview/AboutSplunkEnterprise)

[Splunk Search Tutorial](https://docs.splunk.com/Documentation/Splunk/8.2.1/SearchTutorial/WelcometotheSearchTutorial)

[Splunk Reference Guide for Searches](https://docs.splunk.com/Documentation/Splunk/8.2.1/SearchReference/ListOfSearchCommands)

**DFS COURSES**

[Azure LOG ANALYTICS Inside-Out](https://discover.udemy.com/course/azure-log-analytics/)

[Advanced SQL Server Transaction Log Analysis](https://discover.udemy.com/course/advanced-sql-server-transaction-log-analysis/)

**Logging Books**

[The 3 Pillars of Observability](https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html)

[Monitoring-Demystified-Guide:A guide for logging, tracing, metrics](https://techbeacon.com/enterprise-it/monitoring-demystified-guide-logging-tracing-metrics)

Traces

Traces that are stitched together form special events called “spans”; spans help you track a causal chain through a microservices ecosystem for a single transaction. To accomplish this, each service passes correlation identifiers, known as “trace context,” to each other; this trace context is used to add attributes on the spans. Generate unique request IDs at the edge of your infrastructure, and propagate them through the entire request lifecycle, including to your databases (in the comments field).

IT and DevOps teams can use distributed tracing to monitor applications. Distributed tracing is particularly well-suited to debugging and monitoring modern distributed software architectures, such as microservices. Developers can use distributed tracing to help debug and optimize their code.

As mentioned before, The “span” is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system. Each component of the distributed system contributes a span - a named, timed operation representing a piece of the workflow. Spans can contain “References” to other spans, which allows multiple Spans to be assembled into one complete Trace - a visualization of the life of a request as it moves through a distributed system. Each span encapsulates the following state according to the OpenTracing specification:

* An operation name
* A start timestamp and finish timestamp
* A set of key:value span Tags
* A set of key:value span Logs
* A SpanContext

Trace Tags

Tags are key:value pairs that enable user-defined annotation of spans in order to query, filter, and comprehend trace data. Span tags should apply to the whole span. Examples may include tag keys like db.instance to identify a database host, http.status\_code to represent the HTTP response code, or error which can be set to True if the operation represented by the Span fails.

Trace Logs

Logs are key:value pairs that are useful for capturing span-specific logging messages (see above Logs) and other debugging or informational output from the application itself. Logs may be useful for documenting a specific moment or event within the span (in contrast to tags which should apply to the span as a whole).

Trace SpanContext

The SpanContext carries data across process boundaries. Specifically, it has two major components:

* An implementation-dependent state to refer to the distinct span within a trace
  + i.e., the implementing Tracer’s definition of spanID and traceID
* Any Baggage Items
  + These are key:value pairs that cross process-boundaries.
  + These may be useful to have some data available for access throughout the trace

Resources for Traces

**On the Web**

[Open Telemetry](http://opentelemetry.io/)

[Open Tracing](http://opentracing.io/)

[Observability and Tracing - Tupperware](https://discoverfinancial.sharepoint.com/:p:/r/sites/TheTupperwareProject/_layouts/15/Doc.aspx?sourcedoc=%7BBA68D945-79A0-4702-A6CA-B54B788F71D9%7D&file=Course%2010%20-%20-%20Observability%20%E2%80%93%20Tupperware%20Observability%20Tools%20and%20Configurations.pptx&action=edit&mobileredirect=true)

Relationships within the MELT Framework

Metrics work well for large bodies of data or data collected at regular intervals when you know what you want to ask ahead of time, but they are less granular than event data. Events are useful when the data is relatively small or sporadic in nature, or when you don’t know the specific aggregates you want to see ahead of time. And each individual event is stored until it’s deleted

Additional Observability Resources and Supporting References

Observability On the Web

[1] [Instana: Observability vs Monitoring: What's the difference?](https://www.instana.com/blog/observability-vs-monitoring/)

[2] [thenewstack: Monitoring and Observability: What's the difference and why does it matter](https://thenewstack.io/monitoring-and-observability-whats-the-difference-and-why-does-it-matter/)

[3] [Scalyr: Observability in your systems, the why and how](https://www.scalyr.com/blog/observability-production-systems-why-how/)

[4] [New Relic: What is Observability?](https://newrelic.com/resources/ebooks/what-is-observability)

[5] [thenewstack: Logging and Monitoring, and why you need both](https://thenewstack.io/logging-and-monitoring-why-you-need-both/)

[6] [Google: Monitoring Distributed Systems (The 4 Golden Signals)](https://sre.google/sre-book/monitoring-distributed-systems/)

[7] [New Relic, MELT 101: An introduction to the four essential telemetry data types](https://newrelic.com/content/dam/new-relic/resources/white-papers/melt-101-four-essential-telemetry-data-types.pdf)

Observability Books

[1] [O'reilly: Distributed Systems Observability by Cindy Sridharan. ISBN: 9781492033424](https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/)

Conferences

Webinars [1] [BrightTALK: Blueprint Series: The Complete Guide to Observability](https://www.brighttalk.com/webcast/18160/414555/blueprint-series-the-complete-guide-to-observability)

[2] [BrightTALK: Full Observabiity: From Push to Production](https://www.brighttalk.com/webcast/17527/425244?utm_source=brighttalk-portal&utm_medium=web&utm_campaign=topic&utm_content=on-demand)