OPENSOURCE.COM

The Open Source Guide to DevOps Monitoring Tools

ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: https://opensource.com/story

Email us: open@opensource.com

Chat with us in Freenode IRC: #opensource.com

SUPPORTED BY RED HAT

DAN BARKER

DAN SPENT 12 YEARS in the military as a fighter jet mechanic before transitioning to a career in technology as a Software Engineer. He's now the Chief Architect at the National Association of Insurance Commissioners (NAIC). He's leading technical and cultural transformations for the NAIC, a nonprofit organization focused on consumer protection in the insurance industry. He's an active participant in the CNCFs Serverless Working Group and CloudEvents project. Dan is also an organizer of the DevOps KC Meetup and the DevOpsDays KC conference.

CONTACT DAN

Website: http://danbarker.codes

Email: dan@danbarker.codes

Twitter: https://twitter.com/@barkerd427

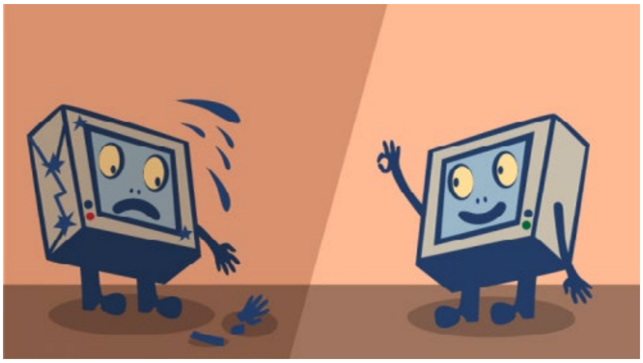
INTRODUCTION

CHAPTERS

GET INVOLVED | ADDITIONAL RESOURCES

A tale of two views

ONCE UPON A TIME, I was troubleshooting some vexing problems in an application that needed to be scaled several orders of magnitude with only a couple of weeks to re-architect it. We had no log aggregation, no metrics aggregation, no distributed tracing, and no visualization. Most of our work had to be done on the actual production nodes using tools like **strace** and grepping through logs. These are great tools, but they don't make it easy to analyze a distributed system across dozens of hosts. We got the job done, but it was painful and involved a lot more guessing and risk than I'd prefer.



At a different job, I was helping to troubleshoot an app in production that was suffering from an out of memory (OOM) issue. The problem was inconsistent, as it didn't seem to correlate with running time, load, time of day, or any other aspect that would provide some predictability. This was obviously going to be a difficult problem to diagnose on a system that spanned hundreds of hosts with many applications calling it. Luckily, we had log aggregation, distributed tracing, metrics aggregation, and a plethora of visualizations. We looked at our memory graph and saw a distinct spike in memory usage, so we used that spike to alert us so we could diagnose the issue in real time when it occurred.

When we received an alert, we went to our log aggregation system to correlate the logs to the memory spike. We found the OOM error and the related calls around it. We now understood what application was calling the service that resulted in the spike and used that information to find the exact transaction that caused the issue. We determined that someone had stored a huge file in a database that our service was now trying to load, but the service was running out of memory before it could fully load and process the record. We should have been defending against this in the first place, but we were happy to find it so quickly and fix it with very little effort. Once we understood the error, we discovered a lot of records had large files like this, and we didn't need that part of the record to function properly.

You might think the second situation happened a long time after the first and we had improved over time. Or maybe you suspect that when I changed jobs, my new company had better tooling. In reality, the second situation happened before the first. I moved from a company with fairly advanced observability tools to one with no observability tools. It was strikingly disturbing as the developer to have an application in production and know nothing about it. I learned a lot about the importance of system observability and the related tools as I began rebuilding that infrastructure. Also, Mike Julian's *Practical Monitoring* [1] is a must-read for those who want to know more about their systems.

Observability principles

So, what are observability tools? Actually, what is observability?

Observability isn't just a marketing term; it's a component of control theory [2]. If you want to get a quick primer, this video [3] might be helpful. Basically, observability means that you can estimate a particular state of a system based on an output. More generally, a system's state should be deterministic from its outputs. Controllability, the mathematical dual of observability [4], of a system requires that a system state be determined by the inputs to the system.

This is a fairly simple concept, but it's very challenging to put into practice. In a sufficiently complex system, it may be nearly impossible to implement full observability. However, you should strive to get the right outputs that allow you to determine the system's state, especially when you encounter a failure mode.

Observability tool types

Over the next few chapters we'll dig into different types of observability tools. For each type, we'll cover what they're used for, what specific tools are available, some use cases, and any unique characteristics that may come up during your search for a new tool. These are presented in the order you should implement them. Metrics aggregation is first, as it's often easy to instrument an application built with any modern language. Second is logging because it will require more application modifications but provides tremendous value. Third is alerting and visualizations, which require the first two types for full functionality. And last is distributed tracing, as it may not be necessary in a simple monolith and is much harder to implement fully.

Metrics aggregation

This type of tool generally consists of time-series data. Time-series data is time-ordered data, and it is normally collected with an internally consistent interval. This consistency allows for some advanced calculations to be applied to the series and provides for predictive analytics using simple regressions or more advanced algorithms.

Log aggregation

These tools deal with data types that are related more to events than to a series of consistent data points. This output is often emitted as a system enters some undesired state. Some systems output a lot of logs that don't fit this condition. We'll cover more of the do's and don'ts of logging in a future chapter.

Alerting/visualizations

This may not appear to fit with the other types listed, as it's really subsequent to the others, but it provides a consumable output for the other types and can produce its own outputs. These types of tools generally make the system more understandable to humans. They also help create a more interactive system through both proactive and reactive notifications about negative system states.

Distributed tracing

Much like tracing within a single application, distributed tracing allows you to follow a single transaction through an entire system. This allows you to home in on specific transactions that might be experiencing problems. Due to performance concerns, a sampling algorithm is often applied.

Common DevOps features

There are several aspects you should look for in any type of observability tool. We'll cover these generally now and will bring them back up in later chapters.

OpenAPI

This specification was previously called Swagger but renamed when it was adopted by the OpenAPI Initiative within the Linux Foundation. The OpenAPI Specification is a language-agnostic tool that can automatically generate documentation of methods, parameters, and models. This is commonly used to generate RESTful interfaces in HTTP, but it is also protocol-agnostic. A user can create a client in almost any language if one doesn't already exist. Every tool should have this type of API (or should be getting it soon). If your tool doesn't have it yet, you may want to look elsewhere. Tools that haven't implemented this specification or don't have it on their roadmap likely have other deficiencies in adopting open, modern standards and code.

Open source

There are a lot of good tools in this space that aren't open source but may be the right fit for your company. If you pick one of those tools, make sure its documentation and accessory tooling are open source. Open source observability tools can provide valuable insights into how your other observability tools are functioning (or maybe not functioning). They also offer all the other benefits of any open source project which you can read more about on opensource.com [5].

Open standards

Regardless of whether or not a tool is open source, it should always use open standards when possible. We've already discussed one of these, OpenAPI, but there are many more. We'll discuss these standards in the appropriate sections to ensure you know they exist and where they're used.

Wide dissemination

Part of observability and openness is allowing everyone to view data. The tools you pick should be open by default. You may want to restrict some areas, but you'll want to default to open and limit access only if it's absolutely required. You never know who in your company might want to solve your problem or who you'll need to bring in to help solve a problem. The last thing you'll want are access barriers when troubleshooting your income source.

Federated model (preferred)

This is similar to defaulting to open, but it allows everyone to provide input and control their own areas more locally. Many legacy systems are architected in a way that requires all data to flow through a central system regardless of need. This also centralizes control around that data. A federated system allows for local aggregation, processing, and control while allowing a central organization to collect the same data or summarized data. The central system likely only wants a subset of the data stored at the local level. This model increases agility, flexibility, and usability.

In the following chapters, we'll be exploring each of the observability tool types in more detail. We'll also help you choose the right tool for your use case.

Links

[1] https://www.practicalmonitoring.com/

[2] https://en.wikipedia.org/wiki/Control_theory

[3] https://www.youtube.com/watch?v=iRZmJBcg1ZA

[4] https://en.m.wikipedia.org/wiki/Duality_(mathematics)

[5] https://opensource.com/

4 open source monitoring tools

ISN'T MONITORING JUST MONITORING? Doesn't it include logging, visualization, and time-series data? The terminology around monitoring has caused a lot of confusion over the years and has led to some poor tools that tout the ability to do everything in one format. Observability proponents recognize there are many levels for observing a system. Metrics aggregation is primarily time-series data, and that's what we'll discuss in this chapter.

Features of time-series data

Counters

A counter is a metric that represents a numeric value that will only increase. (In other words, a counter should never decrease.) Counters accumulate values and present the current total when requested. These are commonly used for things like the total number of web requests, number of errors, number of visitors, etc. This is analogous to the person with a counter device standing at the entrance to an event counting all the people entering. There is generally no option to decrement the counter without resetting it.

Gauges

A gauge is similar to a counter in that it represents a single numeric value, but it can also decrease. It is essentially a representation of some value at a point in time. A thermometer is a good example of a gauge. It moves up and down with the temperature and offers a point-in-time reading. Other uses include CPU usage, memory usage, network usage, and number of threads.

Quantiles

Quantiles aren't a type of metric, but they're germane to the next two sections, histograms and summaries. Let's clarify our understanding of quantiles with an example. A percentile is a type of quantile. Percentiles are something we see regularly, and they should help us understand the general concept more easily. A percentile has 100 "buckets" of values. We often see them related to testing or performance and generally stated as someone scoring within the 85th percentile or some other value. This means the person scoring within that percentile had a real value that fell within the bucket between the 85th and 86th percentile. This person also scored in the top 15% of all students. We don't know the scores in the bucket based off this metric, but that can be derived based on the sum of all scores in the bucket divided by the count of those scores. Quantiles allow us to understand our data better than using a mean or some other statistical function that doesn't take into account outliers and uneven distributions.

Histograms

A histogram is a little more complicated than a counter or a gauge. It is a sample of observations. It consists of a counter, which counts all the observations, and what is essentially a gauge that sums the values of the observations. It uses "buckets" or groupings to segment the values in order to bound the datasets in a productive way. This is commonly seen with quantiles related to request service-level agreements (SLAs). Let's say we want to ensure 95% of our requests are below 500ms. We could use a bucket with an upper bound of 0.5s to collect all values that fall under 500ms. We would then be able to determine how many of the total requests have fallen into that bucket. We can also determine how far we are from our SLA, but this can be difficult to do (as is explained more in the Prometheus documentation [1]).

Histograms are aggregate metrics that are accumulated from multiple instances into a central server. This provides an opportunity to understand the system as a whole rather than on a node by node basis.

Summaries

Summaries are similar to histograms in that they are a sample of observations, but the aggregation occurs on the server side. Also, the estimate of the quantile is more accurate than in a histogram. A summary also uses a sliding time window, so it serves a slightly different case than a histogram but is generally used for the same types of metrics. I normally use a histogram unless I need a very accurate measure of the quantile.

Push/pull

No chapter can be written about metrics aggregation tools without addressing the push vs. pull debate. What is it? The debate centers around whether it is better for your metrics aggregation system to have data pushed to it or to have your metrics aggregation system reach out and gather the data by scraping an endpoint. Multiple articles discuss this (like this one [2] and this one [3]). My perspective is that it mostly doesn't matter. Additional research is left to the reader's discretion.

Tool options

There are many tools available, both open source and commercial. We will focus on open source tools, but some of these have an open core model with a paid component.

Some of these tools feature additional components of observability—principally alerting and visualizations. These will be covered in this section as additional features and won't be covered in subsequent chapters.

Prometheus

This is the most well-recognized time-series monitoring solution for cloud-native applications. It is hosted within the Cloud Native Computing Foundation (CNCF), but it was created by Matt Proud and Julius Volz and sponsored by SoundCloud, with external contributors coming in early to help develop it. Brian Brazil of Robust Perception [4] has built a business of helping companies adopt Prometheus. He also has an excellent blog [5] on his website. The Prometheus documentation [6] is extensive and provides a lot of detail for understanding and using the tool.

Prometheus [7] is a pull-based system that uses local configuration to describe the endpoints to collect from and the interval desired for collection. Each endpoint has a client collecting the data and updating that representation upon each request (or however the client is configured). This data is collected and saved in a highly efficient storage engine on local disk. The storage system uses an append-only file per metric. This storage isn't lossy, which means the fidelity of data from a year ago is as high as the data you are collecting today. However, you may not want to keep that much data locally. Fortunately, there is an option for remote storage for long-term retention and analysis.

Prometheus includes an advanced expression language for selecting and presenting data called PromQL. This data can be displayed graphically, tabularly, or used by external systems through a REST API. The expression language allows a user to create regressions, analyze real-time data, or trend historical data. Labels are also a great tool for filtering and querying data. Labels can be associated with each metric name.

Prometheus also offers a federation model, which encourages more localized control by allowing teams to have their own Prometheis while central teams [8] can also have their own. The central systems could scrape the same endpoints as the local Prometheis, but they can also scrape the local Proemetheis to get the aggregated data that the local instances are collecting. This reduces overhead on the endpoints. This federation model also allows local instances to collect data from each other.

Prometheus comes with AlertManager to handle alerts. This system allows for aggregation of alerts as well as more complex flows to limit when an alert is sent. Let's say 10 nodes suddenly go down at the same time a switch goes down. You probably don't need to send an alert about the 10 nodes, as everyone who receives them will likely be unable to do anything until the switch is fixed. With the AlertManager, it's possible to send an alert only to the networking team for the switch and include additional information about other systems that might be affected. It's also possible to send an email (rather than a page) to the systems team so they know those nodes are down and they don't need to respond unless the systems don't come up after the switch is repaired. If that occurs, then AlertManager will reactivate those alerts that were suppressed by the switch alert.

Graphite

Graphite [9] has been around for a long time, and the recent book *The Art of Monitoring* [10] covers Graphite in detail. Graphite has become ubiquitous in the industry, with many large companies using it at scale.

Graphite is a push-based system that receives data from applications by having the application push the data into Graphite's Carbon component. Carbon stores this data in the Whisper database, and that database and Carbon are read by the Graphite web component that allows a user to graph their data in a browser or pull it through an API. A really cool feature is the ability to export these graphs as images or data files to easily embed them in other applications.

Whisper is a fixed-size database that provides fast, reliable storage of numeric data over time. It is a lossy database, which means the resolution of your metrics will degrade over time. It will provide high-fidelity metrics for the most recent collections and gradually reduce that fidelity over time.

Graphite also uses dot-separated naming, which implies dimensionality. This dimensionality allows for some creative aggregation of metrics and relationships between metrics. This enables aggregation of services across different versions or data centers and (getting more specific) a single version running in one data center in a specific Kubernetes cluster. Granular-level comparisons can also be made to determine if a particular cluster is underperforming.

Another interesting feature of Graphite is the ability to store arbitrary events that should be related to time-series metrics. In particular, application or infrastructure deployments can be added and tracked within Graphite. This allows the operator or developer troubleshooting an issue to have more context about what has happened in the environment related to the anomalous behavior being investigated.

Graphite also has a substantial list of functions [11] that can be applied to metrics series. However, it lacks a powerful query language, which some other tools include. It also lacks any alerting functionality or built-in alerting system.

InfluxDB

InfluxDB [12] is a relatively new entrant, newer than Prometheus. It uses an open core model, which means scaling and clustering cost extra. InfluxDB is part of the larger TICK stack [13] (of Telegraf, InfluxDB, Chronograf, and Kapacitor), so we will include all those components' features in this analysis.

InfluxDB uses a key-value pair system called tags to add dimensionality to metrics, similar to Prometheus and Graphite. The results are similar to what we discussed previously for the other systems. The metric data can be of type **float64**, **int64**, **bool**, and **string** with nanosecond resolution. This is a broader range than most other tools in this space. In fact, the TICK stack is more of an event-aggregation platform than a native time-series metrics-aggregation system.

InfluxDB uses a system similar to a log-structured merge tree for storage. It is called a time-structured merge tree in this context. It uses a write-ahead log and a collection of read-only data files, which are similar to Sorted Strings Tables but have series data rather than pure log data. These files are sharded per block of time. To learn more, check out this great resource on the InfluxData website [14].

The architecture of the TICK stack is different depending on if it's the open source or commercial version. The open source InfluxDB system is self-contained within a single host, while the commercial version is inherently distributed. This is true of the other central components as well. In the open source version, everything runs on a single host. No data or configuration is stored on external systems, so it is fairly easy to manage, but it isn't as robust as the commercial version.

InfluxDB includes a SQL-like language called InfluxQL for querying data from the databases. The primary means for querying data is the HTTP API. The query language doesn't have as many built-in helper functions as Prometheus, but those familiar with SQL will likely feel more comfortable with the language.

The TICK stack also includes an alerting system. This system can do some mild aggregation but doesn't have the full capabilities of Prometheus' AlertManager. It does offer many integrations, though. Also, to reduce load on InfluxDB, continuous queries can be scheduled to store results of queries that Kapacitor will pick up for alerting.

OpenTSDB

OpenTSDB [15] is an open source time-series database, as its name implies. It's unique in this collection of tools in that it stores its metrics in Hadoop. This means it is inherently scalable. If you already have a Hadoop cluster, this might be a good option for metrics you want to store over the long term. If you don't have a Hadoop cluster, the operational overhead might be too large of a burden for you to bear. However, OpenTSDB now supports Google's Bigtable as a backend, which is a cloud service you don't have to operate.

OpenTSDB shares a lot of features with the other systems. It uses a key-value pairing system it calls tags for identifying metrics and adding dimensionality. It has a query language, but it is more limited than Prometheus' PromQL. It does, however, have several built-in functions that help with learning and usage. The API is the main entry point for querying, similar to InfluxDB. This system also stores all data forever, unless there's a time-to-live set in HBase, so you don't have to worry about fidelity degradation.

OpenTSDB doesn't offer an alerting capability, which will make it harder to integrate with your incident response process. This type of system might be great for long-term Prometheus data storage and for performing more historical analytics to reveal systemic issues, rather than as a tool to quickly identify and respond to acute concerns.

OpenMetrics standard

OpenMetrics [16] is a working group seeking to establish a standard exposition format for metrics data. It is influenced by Prometheus. If this initiative is successful, we'll have an industry-wide abstraction that would allow us to switch between tools and providers with ease. Leading companies like Datadog [17] have already started offering tools that can consume the Prometheus exposition format, which will be easy to convert to the OpenMetrics standard once it's released.

It's also important to note that the contributors to this project include Google and InfluxData (among others). This likely means InfluxDB will eventually adopt the OpenMetrics standard. This may also mean that one of the three largest cloud providers will adopt it, if Google's involvement is an indicator. Of course, the exposition format is already being used in the Google-created Kubernetes project [18]. SolarWinds, Robust Perceptions, and SpaceNet are also involved.

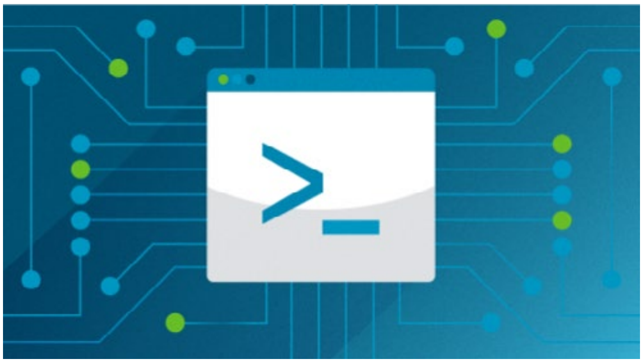
Links

[1] https://prometheus.io/docs/practices/histograms/
[2] https://thenewstack.io/exploring-prometheus-use-cases-brian-brazil/
[3] https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/
[4] https://www.robustperception.io/
[5] https://www.robustperception.io/blog
[6] https://prometheus.io/docs/
[7] https://prometheus.io/
[8] https://prometheus.io/docs/introduction/faq/#what-is-the-plural-of-prometheus
[9] https://graphiteapp.org/
[10] https://artofmonitoring.com/
[11] http://graphite.readthedocs.io/en/latest/functions.html
[12] https://www.influxdata.com/
[13] https://www.thoughtworks.com/radar/platforms/tick-stack
[14] https://docs.influxdata.com/influxdb/v1.5/concepts/storage_engine/
[15] http://opentsdb.net/
[16] https://github.com/RichiH/OpenMetrics
[17] https://www.datadoghq.com/blog/monitor-prometheus-metrics/
[18] https://opensource.com/resources/what-is-kubernetes

3 open source log aggregation tools

HOW IS METRICS AGGREGATION different from log aggregation? Can't logs include metrics? Can't log aggregation systems do the same things as metrics aggregation systems? These are questions I see a lot. I've also seen vendors pitching their log aggregation system as the solution to all observability problems. Log aggregation is a valuable tool, but it isn't normally a good tool for time-series data.

A couple of valuable features in a time-series metrics aggregation system are the regular interval and the storage system customized specifically for time-series data. The regular interval allows a user to derive real mathematical results consistently. If a log aggregation system is collecting metrics in a regular interval, it can potentially work the same way. However, the storage system isn't optimized for the types of queries that are typical in a metrics aggregation system. These queries will take more resources and time to process using storage systems found in log aggregation tools.



So, we know a log aggregation system is likely not suitable for time-series data, but what is it good for? A log aggregation system is a great place for collecting event data. These are irregular activities that are significant. An example might be access logs for a web service. These are significant because we want to know what is accessing our systems and when. Another example would be an application error condition—because it is not a normal operating condition, it might be valuable during troubleshooting.

A handful of rules for logging:

- DO include a timestamp
- DO format in JSON
- DON'T log insignificant events
- DO log all application errors
- MAYBE log warnings
- DO turn on logging
- DO write messages in a human-readable form
- DON'T log informational data in production
- DON'T log anything a human can't read or react to

Cloud costs

When investigating log aggregation tools, the cloud might seem like an attractive option. However, it can come with significant costs. Logs represent a lot of data when aggregated across hundreds or thousands of hosts and applications. The ingestion, storage, and retrieval of that data are expensive in cloud-based systems.

As a point of reference from a real system, a collection of around 500 nodes with a few hundred apps results in 200GB of log data per day. There's probably room for improvement in that system, but even reducing it by half will cost nearly $10,000 per month in many SaaS offerings. This often includes retention of only 30 days, which isn't very long if you want to look at trending data year-over-year.

This isn't to discourage the use of these systems, as they can be very valuable—especially for smaller organizations. The purpose is to point out that there could be significant costs, and it can be discouraging when they are realized. The rest of this chapter will focus on open source and commercial solutions that are self-hosted.

Tool options

ELK

ELK [1], short for Elasticsearch, Logstash, and Kibana, is the most popular open source log aggregation tool on the market. It's used by Netflix, Facebook, Microsoft, LinkedIn, and Cisco. The three components are all developed and maintained by Elastic [2]. Elasticsearch [3] is essentially a NoSQL, Lucene search engine implementation. Logstash [4] is a log pipeline system that can ingest data, transform it, and load it into a store like Elasticsearch. Kibana [5] is a visualization layer on top of Elasticsearch.

A few years ago, Beats were introduced. Beats are data collectors. They simplify the process of shipping data to Logstash. Instead of needing to understand the proper syntax of each type of log, a user can install a Beat that will export NGINX logs or Envoy proxy logs properly so they can be used effectively within Elasticsearch.

When installing a production-level ELK stack, a few other pieces might be included, like Kafka [6], Redis [7], and NGINX [8]. Also, it is common to replace Logstash with Fluentd, which we'll discuss later. This system can be complex to operate, which in its early days led to a lot of problems and complaints. These have largely been fixed, but it's still a complex system, so you might not want to try it if you're a smaller operation.

That said, there are services available so you don't have to worry about that. Logz.io [9] will run it for you, but its list pricing is a little steep if you have a lot of data. Of course, you're probably smaller and may not have a lot of data. If you can't afford Logz.io, you could look at something like AWS Elasticsearch Service (ES) [10]. ES is a service Amazon Web Services (AWS) offers that makes it very easy to get Elasticsearch working quickly. It also has tooling to get all AWS logs into ES using Lambda and S3. This is a much cheaper option, but there is some management required and there are a few limitations.

Elastic, the parent company of the stack, offers [11] a more robust product that uses the open core model, which provides additional options around analytics tools, security tools, and reporting. It can also be hosted on Google Cloud Platform or AWS. This might be the best option, as this combination of tools and hosting platforms offers a cheaper solution than most SaaS options and still provides a lot of value. This system could effectively replace or give you the capability of a security information and event management (SIEM) system [12].

The ELK stack also offers great visualization tools through Kibana, but it lacks an alerting function. Elastic provides alerting functionality within the paid X-Pack add-on, but there is nothing built in for the open source system. Yelp has created a solution to this problem, called ElastAlert [13], and there are probably others. This additional piece of software is fairly robust, but it increases the complexity of an already complex system.

Graylog

Graylog [14] has recently risen in popularity, but it got its start when Lennart Koopmann created it back in 2010. A company was born with the same name two years later. Despite its increasing use, it still lags far behind the ELK stack. This also means it has fewer community-developed features, but it can use the same Beats that the ELK stack uses. Graylog has gained praise in the Go community with the introduction of the Graylog Collector Sidecar written in Go [15].

Graylog uses Elasticsearch, MongoDB [16], and the Graylog Server under the hood. This makes it as complex to run as the ELK stack and maybe a little more. However, Graylog comes with alerting built into the open source version, as well as several other notable features like streaming, message rewriting, and geolocation.

The streaming feature allows for data to be routed to specific Streams in real time while they are being processed. With this feature, a user can see all database errors in a single Stream and web server errors in a different Stream. Alerts can even be based on these Streams as new items are added or when a threshold is exceeded. Latency is probably one of the biggest issues with log aggregation systems, and Streams eliminate that issue in Graylog. As soon as the log comes in, it can be routed to other systems through a Stream without being processed fully.

The message rewriting feature uses the open source rules engine Drools [17]. This allows all incoming messages to be evaluated against a user-defined rules file enabling a message to be dropped (called Blacklisting); a field to be added or removed; or the message to be modified.

The coolest feature might be Graylog's geolocation capability, which supports plotting IP addresses on a map. This is a fairly common feature and is available in Kibana as well, but it adds a lot of value—especially if you want to use this as your SIEM system. The geolocation functionality is provided in the open source version of the system.

Graylog, the company, charges for support on the open source version if you want it. It also offers an open core model for its Enterprise version that offers archiving, audit logging, and additional support. There aren't many other options for support or hosting, so you'll likely be on your own if you don't use Graylog (the company).

Fluentd

Fluentd [18] was developed at Treasure Data [19], and the CNCF [20] has adopted it as an Incubating project. It was written in C and Ruby and is recommended by AWS [21] and Google Cloud [22]. Fluentd has become a common replacement for Logstash in many installations. It acts as a local aggregator to collect all node logs and send them off to central storage systems. It is not a log aggregation system.

It uses a robust plugin system to provide quick and easy integrations with different data sources and data outputs. Since there are over 500 plugins available, most of your use cases should be covered. If they aren't, this sounds like an opportunity to contribute back to the open source community.

Fluentd is a common choice in Kubernetes environments due to its low memory requirements (just tens of megabytes) and its high throughput. In an environment like Kubernetes [23], where each pod has a Fluentd sidecar, memory consumption will increase linearly with each new pod created. Using Fluentd will drastically reduce your system utilization. This is becoming a common problem with tools developed in Java that are intended to run one per node where the memory overhead hasn't been a major issue.

Links

[1] https://www.elastic.co/webinars/introduction-elk-stack
[2] https://www.elastic.co/
[3] https://www.elastic.co/products/elasticsearch
[4] https://www.elastic.co/products/logstash
[5] https://www.elastic.co/products/kibana
[6] http://kafka.apache.org/
[7] https://redis.io/
[8] https://www.nginx.com/
[9] https://logz.io/
[10] https://aws.amazon.com/elasticsearch-service/
[11] https://www.elastic.co/cloud
[12] https://en.wikipedia.org/wiki/Security_information_and_event_management
[13] https://github.com/Yelp/elastalert
[14] https://www.graylog.org/
[15] https://opensource.com/tags/go
[16] https://www.mongodb.com/
[17] https://www.drools.org/
[18] https://www.fluentd.org/
[19] https://www.treasuredata.com/
[20] https://www.cncf.io/
[21] https://aws.amazon.com/blogs/aws/all-your-data-fluentd/
[22] https://cloud.google.com/logging/docs/agent/
[23] https://opensource.com/resources/what-is-kubernetes

5 alerting and visualization tools

PERHAPS IT'S CLEAR BY THE NAME what alerting and visualization tools are used for, but it might not be clear why they are observability tools or why they're separated here. Some systems include the visualization component in their main product, so why separate it here? Observability comes from control theory and describes our ability to understand a system based on its inputs and outputs. This chapter focuses on the output component of observability.

Alerting and visualization systems are focused on understanding the outputs of other systems. This is why they're grouped together. Visualization and alerting tools could be described as tools that provide structured representations of system outputs. Alerts are basically a synthesized understanding of negative system outputs, and visualizations are disambiguated structured representations focused on facilitating user comprehension.

As already mentioned, some systems come with these tools built in, and those have been covered in other sections with those tools.

Common types of alerts and visualizations

Alerts

Let's first cover what alerts are not. Alerts should not be sent if the human responder can't do anything about the problem. This includes alerts that go to multiple individuals with only a few who can respond or situations where every anomaly in the system triggers an alert. This leads to alert fatigue and receivers ignoring all alerts within a specific medium until the system escalates to a medium that isn't already saturated.

For example, if an operator is getting hundreds of emails a day from the alerting system, that operator is going to ignore all emails from the alerting system. The operator will only respond to a real incident when he or she is experiencing the problem, emailed by a customer, or called by the boss. In this case, alerts have lost their meaning and usefulness.

Alerts are not a constant stream of information or a status update. They are meant to convey a problem from which the system can't automatically recover, and they are sent only to the individual most likely to be able to recover the system. Everything that falls outside this definition isn't an alert and is only hurting your employees and company culture.

Everyone has a different set of alert types, so I'll not cover things like priority levels (P1-P5) or models that use words like Informational, Warning, and Critical. Instead, I'll describe the generic categories emergent in complex systems' incident response.

You might have noticed I mentioned an "Informational" alert type right after I wrote that alerts shouldn't be informational. Well, not everyone agrees, but also I don't consider something an alert if it isn't sent to anyone. It is a data point that many systems refer to as an alert. It represents some event that should be known but not responded to. It is generally part of the visualization system of the alerting tool and not an event that triggers actual notifications. Mike Julian covers this and other aspects of alerting in his book *Practical Monitoring* [1]. It's a must read for work in this area.

Non-informational alerts consist of types that can be responded to or require action. I group these into two categories: internal outage and external outage. (Most companies have more levels than this for prioritizing their response efforts.) Degraded system performance is considered an outage in this model, as it's usually unknown how bad the impact is to each user.

Internal outages are lower priority than external outages, but they still need to be responded to quickly. They often include internal systems that company employees use or components of applications that are visible only to company employees.

External outages consist of any system outage that would immediately impact a customer. These don't include a system outage that prevents releasing updates to the system. They do include customer-facing application failures, database outages, and networking partitions that hurt availability or consistency if either can impact a user. They also include outages of tools that may not have direct impact on users, as the application continues to run, but this transparent dependency impacts performance. This is common when the system uses some external service or data source that isn't necessary for full functionality but may cause delays as the application performs retries or handles errors from this external dependency.

Visualizations

There are a lot of visualization types, and I won't cover them all here. It's a fascinating area of research. On the data analytics side of my career, this is a constant struggle of learning and applying that knowledge. We need to provide simple representations of complex system outputs for the widest dissemination of information. Google Charts [2] and Tableau [3] have a wide selection of visualization types. We'll cover the most common visualizations and some innovative solutions for quickly understanding systems.

Line chart

The line chart is probably the most common and ubiquitous visualization available. It also does a pretty good job of producing an understanding of a system over time. A line chart in a metrics system would have a line for each unique metric or some aggregation of metrics. This can get confusing when there are a lot of metrics in the same dashboard (as evidenced below), but most systems can select specific metrics to view rather than having all of them visible. Also, anomalous behavior is easy to spot if it's significant enough to escape the noise of normal operations. Below we can see purple, yellow, and light blue lines that might indicate anomalous behavior.

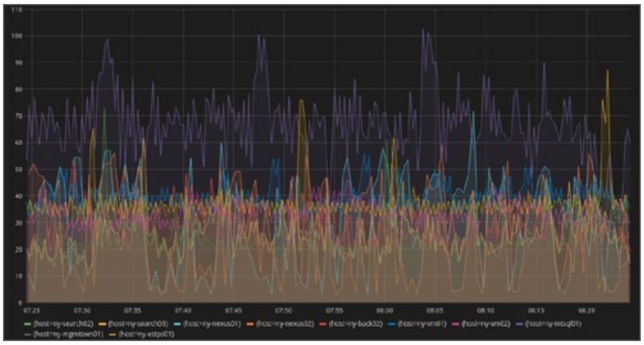


Image source: Stackoverflow.com (Creative Commons BY SA 3.0)

Another feature of a line chart is that you can often stack them to show relationships. For example, you might want to look at requests on each server individually, but also in aggregate. This allows you to understand both the overall system as well as each instance in the same graph.

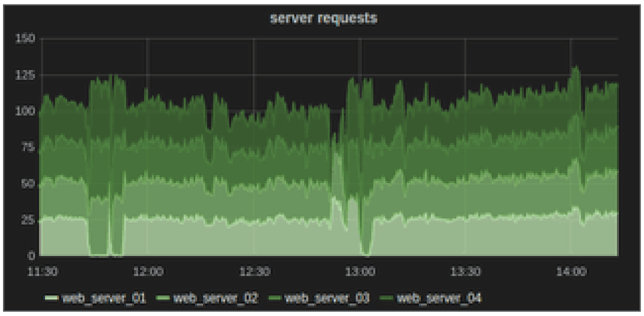


Image source: Grafana (© Grafana Labs)

Heatmaps

Another common visualization is the heatmap. It is useful when looking at histograms. This type of visualization is similar to a bar chart but can show gradients within the bars representing the different percentiles of the overall metric. For example, maybe you're looking at request latencies, and you want to quickly understand the overall trend as well as the distribution of all requests. A heatmap is great for this, and it can use color to disambiguate the quantity of each section with a quick glance. The heatmap below shows the higher concentration around the centerline of the graph with an easy-to-understand visualization of the distribution vertically for each time bucket. We might want to review a couple of points in time where the distribution gets wide while the others are fairly tight like at 14:00. This distribution might be a negative performance indicator.

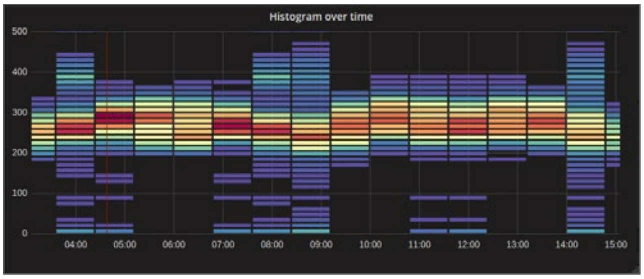


Image source: Grafana.org (© Grafana Labs)

Gauges

The last common visualization I'll cover is used to understand a single metric quickly. Gauges can be used to represent a single metric, like your speedometer represents your speed or your gas gauge represents the amount of gas in your car. Similar to the gas gauge, most monitoring gauges clearly indicate what is good and what isn't. Often (as is shown below), good is represented by green, getting worse by orange, and "everything is breaking" by red. The middle row below shows traditional gauges.

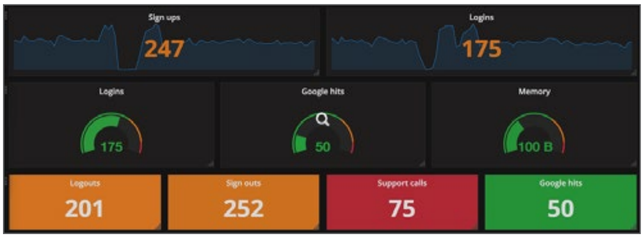


Image source: Grafana.org (© Grafana Labs)

This image shows more than just traditional gauges, though. The other gauges are single stat representations that are similar to the function of the classic gauge. They all use the same color scheme for quickly indicating system health with just a glance. Arguably, the bottom row is probably the best example of a gauge that allows you to glance at a dashboard and know that everything is healthy (or not). This type of visualization is usually what I put on a top-level dashboard. It offers a full, high-level understanding of system health in seconds.

Flame graphs

A less common visualization is the flame graph. It's not ideal for dashboarding or quickly observing high-level system concerns; it's normally seen when trying to understand a specific application problem. Netflix's Brendan Gregg introduced them in 2011 [4]. This visualization focuses on CPU and memory and the associated frames. The X-axis lists the frames alphabetically, and the Y-axis shows stack depth. Each rectangle is a stack frame and includes the function being called. The wider the rectangle, the more it appears in the stack. This method is invaluable when trying to diagnose system performance at the application level and I urge everyone to give them a try.

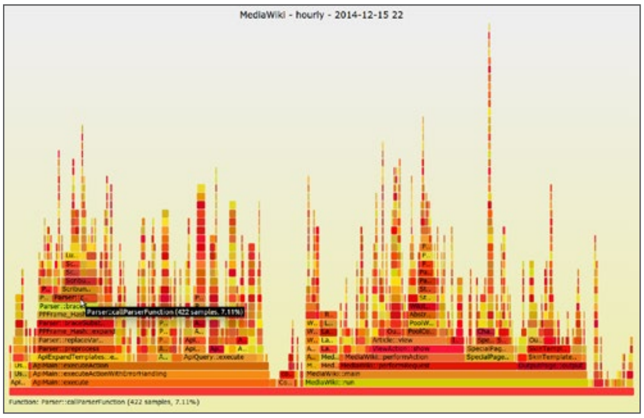


Image source: Wikimedia.org (Creative Commons BY SA 3.0)

Tool options

There are several commercial options for alerting, but this is Opensource.com, so we're not even gonna mention them! We'll cover systems that are being used at scale by real companies that you can use at no cost. Hopefully, you'll be able to contribute new and innovative features to make these systems even better.

Alerting tools

Bosun

If you've ever done anything with computers and gotten stuck, the help you received was probably thanks to a Stack Exchange system. Stack Exchange runs many different websites around a crowdsourced question-and-answer model. Stack Overflow [5] is very popular with developers, and Super User [6] is popular with operations. However, there are now hundreds of sites ranging from parenting to sci-fi and philosophy to bicycles.

Stack Exchange open sourced its alert management system, Bosun [7], around the same time Prometheus and its AlertManager [8] system were released. There were a lot of similarities in the two systems, and that's a really good thing. Like Prometheus, Bosun is written in Golang. Bosun's scope is more extensive than Prometheus' as it can interact with systems beyond metrics aggregation. It can also ingest data from log and event aggregation systems. It supports Graphite, InfluxDB, OpenTSDB, and Elasticsearch.

Bosun's architecture consists of a single server binary, a backend like OpenTSDB, Redis, and scollector agents. The scollector agents [9] automatically detect services on a host and report metrics for those processes and other system resources. This data is sent to a metrics backend. The Bosun server binary then queries the backends to determine if any alerts need to be fired. Bosun can also be used by tools like Grafana [10] to query the underlying backends through one common interface. Redis is used to store state and metadata for Bosun.

A really neat feature of Bosun is that it lets you test your alerts against historical data. This was something I missed in Prometheus several years ago when I had data for an issue I wanted alerts on, but no easy way to test my new alert to make sure it would work. I had to create and insert dummy data to test the alert. That was a very time-consuming process, and this system alleviates that.

Bosun also has the usual features like showing simple graphs and creating alerts. It has a powerful expression language for writing alerting rules. However, it only has email and HTTP notification configurations, which means connecting to Slack and other tools requires a bit more customization (which its documentation covers [11]). Similar to Prometheus, Bosun can use templates for these notifications, which means they can look as awesome as you want them to. You can use all your HTML and CSS skills to create the baddest email alert anyone has ever seen.

Cabot

Cabot [12] was created by a company called Arachnys [13]. Many may not know who that is or what it does, but you have probably felt its impact without knowing it. It has built the leading cloud-based solution for fighting financial crimes. That sounds pretty cool, right? At a previous company, I was involved in similar functions around “know your customer" laws [14]. Many companies would see it as very bad press to be linked to a terrorist group funneling money through their systems. These solutions also help defend against less atrocious offenders like fraudsters who pose a risk to the institution, even if less so.

So why did Arachnys create Cabot? Well, it is kind of a Christmas present to everyone, as it was a Christmas project it built because its developers couldn’t wrap their heads around Nagios [15]. And really, who can blame them? Cabot was written with Django and Bootstrap, so it should be easy for most to contribute to the project. Another interesting factoid is that the name comes from the creator’s dog.

The Cabot architecture is similar to Bosun in that it doesn’t collect any data. Instead, it accesses data through the APIs of the tools it is alerting for. Therefore, Cabot uses a pull (rather than a push) model for alerting. It reaches out into each system’s API and retrieves the information it needs to make a decision based on a specific check. Cabot stores the alerting data in a Postgres database and also has a cache using Redis.

Cabot natively supports Graphite [16], but it also supports Jenkins [17], which is rare in this area. Arachnys [18] uses Jenkins like a centralized cron, but I like this idea of treating build failures like outages. Obviously, a build failure isn’t as critical as a production outage, but it could still alert the team and escalate if the failure isn’t resolved. Who actually checks Jenkins every time an email comes in about a build failure? Yeah, me too!

Another interesting feature is that Cabot can integrate with Google Calendar for on-call rotations. Cabot calls this feature Rota, which is a British term for a roster or rotation. This makes a lot of sense, and I wish other systems would take this idea further. Cabot doesn’t support anything more complex than primary and backup personnel, but there is certainly room for additional features. The docs say if you want something more advanced, you should look at a commercial option.

StatsAgg

StatsAgg [19]? How did that make the list? Well, it’s not every day you come across a publishing company that has created an alerting platform. I think that deserves recognition. Pearson [20] isn’t just a publishing company anymore, though. It has several web presences and a joint venture with O’Reilly Media. However, I still think of the company as the people who published my school books and tests.

StatsAgg isn’t just an alerting platform; it’s also a metrics aggregation platform. And it’s kind of like a proxy for other systems. It supports Graphite, StatsD, InfluxDB, and OpenTSDB as inputs, but it can also forward those metrics to their respective platforms. This is an interesting concept, but potentially risky as loads increase on a central service. However, if the StatsAgg infrastructure is robust enough, it can still produce alerts even when a backend storage platform has an outage.

StatsAgg is written in Java and only consists of the main server and UI, which keeps complexity to a minimum. It can send alerts based on regular expression matching and is focused on alerting by service rather than host or instance. Its goal was to fill a void in the open source observability stack, and I think it does that quite well.

Visualization tools

Grafana

Almost everyone knows about Grafana [21] and many have used it. I have been using it for years whenever I need a simple dashboard. The tool I used before was deprecated, and Grafana made that okay when at first I was fairly distraught when I saw the deprecation notice. Grafana was gifted to us by Torkel Ödegaard. Oddly, Grafana is another project that was created around Christmas time and released in January 2014. It has come a long way in only a few years. It started life as a Kibana dashboarding system, which Torkel forked into what became Grafana.

Grafana’s sole focus is presenting monitoring data in a more usable and pleasing way. It can natively gather data from Graphite, Elasticsearch, OpenTSDB, Prometheus, and InfluxDB. There’s an Enterprise version that uses plugins for more data sources, but there’s no reason those other data source plugins couldn’t be created as open source, as the Grafana plugin ecosystem already offers many other data sources.

What does Grafana do for me? It provides a central location for understanding my system. It is web-based, so anyone can access the information, although it can be restricted using different authentication methods. Grafana can provide knowledge at a glance using many different types of visualizations. However, it has started integrating alerting and other features that aren’t traditionally combined with visualizations.

Now you can set alerts visually. That means you can look at a graph, maybe even one showing where an alert should have triggered due to some degradation of the system, click on the graph where you want the alert to trigger, and then tell Grafana where to send the alert. That’s a pretty powerful addition that won’t necessarily replace an alerting platform, but it can certainly help augment it by providing a different perspective on alerting criteria.

Grafana has also introduced more collaboration features. Users have been able to share dashboards for a long time, meaning you don’t have to create your own dashboard for your Kubernetes [22] cluster because there are several already available—with some maintained by Kubernetes developers and others by Grafana developers.

The most significant addition around collaboration is annotations. Annotations allow a user to add context to part of a graph. Then other users can use this context to understand the system better. This is an invaluable tool when a team is in the middle of an incident and communication and common understanding are critical. Having all the information right where you're already looking makes it much more likely that knowledge will be shared across the team quickly. It's also a nice feature to use during blameless postmortems when the team is trying to understand how the failure occurred and learn more about their system.

Vizceral

Netflix created Vizceral [23] to understand its traffic patterns better when performing a traffic failover. Unlike Grafana, which is a more general tool, Vizceral serves a very specific use-case. Netflix no longer uses this tool internally and says it is no longer actively maintained, but it still updates the tool periodically. I highlight it here primarily to point out an interesting visualization mechanism and how it can help solve a problem. It's worth running it in a demo environment just to better grasp the concepts and witness what's possible with these systems.

Links

[1] https://www.practicalmonitoring.com/
[2] https://developers.google.com/chart/interactive/docs/gallery
[3] https://libguides.libraries.claremont.edu/c.php?g=474417&p=3286401
[4] http://www.brendangregg.com/flamegraphs.html
[5] https://stackoverflow.com/
[6] https://superuser.com/
[7] http://bosun.org/
[8] https://prometheus.io/docs/alerting/alertmanager/
[9] https://bosun.org/scollector/
[10] https://grafana.com/
[11] https://bosun.org/notifications
[12] https://cabotapp.com/
[13] https://www.arachnys.com/
[14] https://en.wikipedia.org/wiki/Know_your_customer
[15] https://www.nagios.org/
[16] https://graphiteapp.org/
[17] https://jenkins.io/
[18] https://www.arachnys.com/
[19] https://github.com/PearsonEducation/StatsAgg
[20] https://www.pearson.com/us/
[21] https://grafana.com/
[22] https://opensource.com/resources/what-is-kubernetes
[23] https://github.com/Netflix/vizceral

3 open source distributed tracing tools

DISTRIBUTED TRACING SYSTEMS enable tracking a request through a software system that is distributed across multiple applications, services, and databases as well as intermediaries like proxies. This allows for a deeper understanding of what is happening within a software system. These systems produce graphical representations that show how much time the request took on each step and lists each known step.

A user reviewing this content can determine where the system is experiencing latencies or blockages. Instead of testing the system like a binary search tree when requests start failing, operators and developers can see exactly where the issues begin. This can also reveal where performance changes might be occurring from deployment to deployment. It's always better to catch regressions automatically by alerting to the anomalous behavior rather than having your customers tell you.

How does this tracing thing work? Well, each request gets a special ID that's usually injected into the headers. This ID uniquely identifies that transaction. This transaction is normally called a trace. The trace is the overall abstract idea of the entire transaction. Each trace is made up of spans. These spans are the actual work being performed, like a service call or a database request. Each span also has a unique ID. Spans can create subsequent spans called child spans, and child spans can have multiple parents.

Once a transaction (or trace) has run its course, it can be searched in a presentation layer. There are several tools in this space that we'll discuss later, but the picture below is of Jaeger [1] from my Istio walkthrough [2]. It shows multiple spans of a single trace. The power of this is immediately clear as you can better understand the transaction's story at a glance.

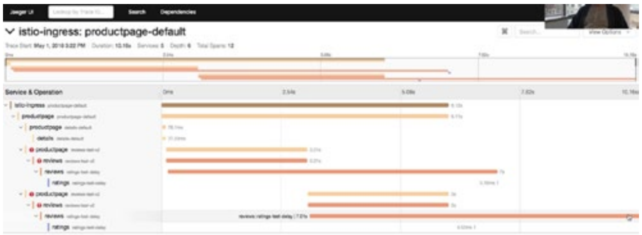


Image by Dan Barker (Creative Commns BY SA 4.0)

This demo is using Istio's built-in OpenTracing implementation, so I can get tracing without even modifying my application. It also uses Jaeger, which is OpenTracing-compatible. So what is OpenTracing? Let's find out.

OpenTracing API

OpenTracing [3] is a spec that grew out of Zipkin [4] to provide cross-platform compatibility. It offers a vendor-neutral API for adding tracing to applications and delivering that data into distributed tracing systems. A library written for the OpenTracing spec can be used with any system that is OpenTracing compliant. Zipkin, Jaeger, and AppDash are examples of open source tools that have adopted the open standard, but even proprietary tools like Datadog and Instana are adopting it. This is expected to continue as OpenTracing reaches ubiquitous status.

OpenCensus

Okay, we have OpenTracing, but what is this OpenCensus [5] thing that keeps popping up in my searches? Is it a competing standard, something completely different, or something complementary? That answer depends on who you ask. I will do my best to explain the difference (as I understand it).

OpenCensus is a more holistic or all-inclusive approach. OpenTracing is focused on establishing an open API and spec and not on open implementations for each language and tracing system. OpenCensus provides not only the specification but also the language implementations and wire protocol. It also goes beyond tracing by including additional metrics that are normally outside the scope of distributed tracing systems.

OpenCensus allows viewing data on the host where the application is running, but it also has a pluggable exporter system for exporting data to central aggregators. The current exporters produced by the OpenCensus team are Zipkin, Prometheus, Jaeger, Stackdriver, Datadog, and SignalFx, but anyone can create an exporter.

From my perspective, there's a lot of overlap. One isn't necessarily better than the other, but it's important to know what each does and doesn't do. OpenTracing is primarily a spec with others doing the implementation and opinionation. OpenCensus provides a holistic approach for the local component with more opinionation but still requires other systems for remote aggregation.

Tool options

Zipkin

Zipkin was one of the first systems of this kind. It was developed by Twitter based on the Google Dapper paper [6] about the internal system Google uses. Zipkin was written using Java, and it can use Cassandra or ElasticSearch as scalable backends. Most companies should be satisfied with one of those options. The lowest supported Java version is Java 6. It also uses the Thrift [7] binary communication protocol, which is popular in the Twitter stack and is hosted as an Apache project.

The system consists of reporters (clients), collectors, a query service, and a web UI. Zipkin is meant to be safe in production by transmitting only a trace ID within the context of a transaction to inform receivers that a trace is in process. The data collected in each reporter is then transmitted asynchronously to the collectors. The collectors store these spans in the database, and the web UI presents this data to the end user in a consumable format. The delivery of data to the collectors can occur in three different methods: HTTP, Kafka, and Scribe.

The Zipkin community [8] has also created Brave [9], a Java client implementation compatible with Zipkin. It has no dependencies, so it won't drag your projects down or clutter them with libraries that are incompatible with your corporate standards. There are many other implementations, and Zipkin is compatible with the OpenTracing standard, so these implementations should also work with other distributed tracing systems. The popular Spring framework has a component called Spring Cloud Sleuth [10] that is compatible with Zipkin.

Jaeger

Jaeger [11] is a newer project from Uber Technologies that the CNCF [12] has since adopted as an Incubating project. It is written in Golang, so you don't have to worry about having dependencies installed on the host or any overhead of interpreters or language virtual machines. Similar to Zipkin, Jaeger also supports Cassandra and ElasticSearch as scalable storage backends. Jaeger is also fully compatible with the OpenTracing standard.

Jaeger's architecture is similar to Zipkin, with clients (reporters), collectors, a query service, and a web UI, but it also has an agent on each host that locally aggregates the data. The agent receives data over a UDP connection, which it batches and sends to a collector. The collector receives that data in the form of the Thrift [13] protocol and stores that data in Cassandra or ElasticSearch. The query service can access the data store directly and provide that information to the web UI.

By default, a user won't get all the traces from the Jaeger clients. The system samples 0.1% (1 in 1,000) of traces that pass through each client. Keeping and transmitting all traces would be a bit overwhelming to most systems. However, this can be increased or decreased by configuring the agents, which the client consults with for its configuration. This sampling isn't completely random, though, and it's getting better. Jaeger uses probabilistic sampling, which tries to make an educated guess at whether a new trace should be sampled or not. Adaptive sampling is on its roadmap [14], which will improve the sampling algorithm by adding additional context for making decisions.

AppDash

AppDash [15] is a distributed tracing system written in Golang, like Jaeger. It was created by Sourcegraph [16] based on Google's Dapper and Twitter's Zipkin. Similar to Jaeger and Zipkin, AppDash supports the OpenTracing standard; this was a later addition and requires a component that is different from the default component. This adds risk and complexity.

At a high level, AppDash's architecture consists mostly of three components: a client, a local collector, and a remote collector. There's not a lot of documentation, so this description comes from testing the system and reviewing the code. The client in AppDash gets added to your code. AppDash provides Python, Golang, and Ruby implementations, but OpenTracing libraries can be used with AppDash's OpenTracing implementation. The client collects the spans and sends them to the local collector. The local collector then sends the data to a centralized AppDash server running its own local collector, which is the remote collector for all other nodes in the system.

Links

[1] https://www.jaegertracing.io/
[2] https://www.youtube.com/watch?v=T8BbeqZ0Rls
[3] http://opentracing.io/
[4] https://zipkin.io/
[5] https://opencensus.io/
[6] https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf
[7] https://thrift.apache.org/
[8] https://zipkin.io/pages/community.html
[9] https://github.com/openzipkin/brave
[10] https://cloud.spring.io/spring-cloud-sleuth/
[11] https://www.jaegertracing.io/
[12] https://www.cncf.io/
[13] https://en.wikipedia.org/wiki/Apache_Thrift
[14] https://www.jaegertracing.io/docs/roadmap/#adaptive-sampling
[15] https://github.com/sourcegraph/appdash
[16] https://about.sourcegraph.com/

GET INVOLVED

If you find these articles useful, get involved! Your feedback helps improve the status quo for all things DevOps.

Contribute to the Opensource.com DevOps resource collection, and join the team of DevOps practitioners and enthusiasts who want to share the open source stories happening in the world of IT.

The Open Source DevOps team is looking for writers, curators, and others who can help us explore the intersection of open source and DevOps. We're especially interested in stories on the following topics:

- DevOps practical how to's
- DevOps and open source
- DevOps and talent
- DevOps and culture
- DevSecOps/rugged software

Learn more about the Opensource.com DevOps team: https://opensource.com/devops-team

ADDITIONAL RESOURCES

The ultimate DevOps hiring guide

This free download provides advice, tactics, and information about the state of DevOps hiring for both job seekers and hiring managers.

Download it now: The ultimate DevOps hiring guide

The Open Organization Guide to IT Culture Change

In The Open Organization Guide to IT Culture Change, more than 25 contributors from open communities, companies, and projects offer hard-won lessons and practical advice on how to create an open IT department that can deliver better, faster results and unparalleled business value.

Download it now: The Open Organization Guide to IT Culture Change

WRITE FOR US

Would you like to write for Opensource.com? Our editorial calendar includes upcoming themes, community columns, and topic suggestions: https://opensource.com/calendar

Learn more about writing for Opensource.com at: https://opensource.com/writers

We're always looking for open source-related articles on the following topics:

Big data: Open source big data tools, stories, communities, and news.

Command-line tips: Tricks and tips for the Linux command-line.

Containers and Kubernetes: Getting started with containers, best practices, security, news, projects, and case studies.

Education: Open source projects, tools, solutions, and resources for educators, students, and the classroom.

Geek culture: Open source-related geek culture stories.

Hardware: Open source hardware projects, maker culture, new products, howtos, and tutorials.

Machine learning and AI: Open source tools, programs, projects and howtos for machine learning and artificial intelligence.

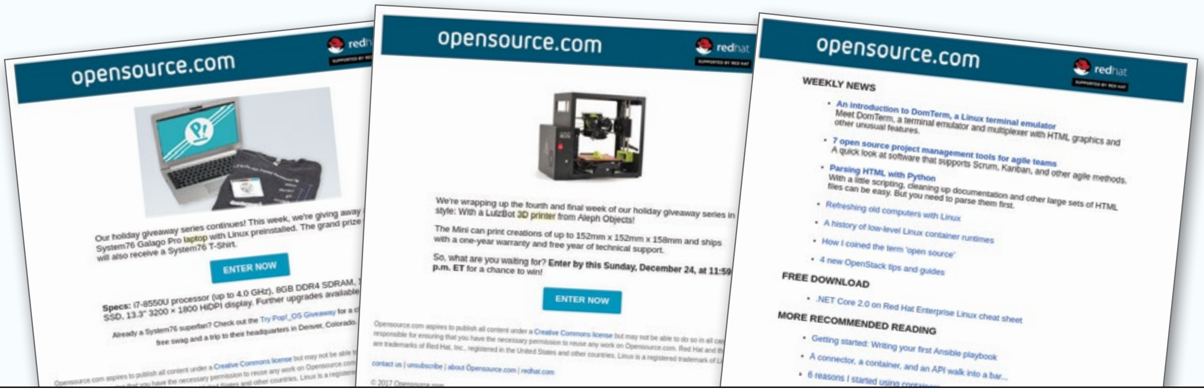
Programming: Share your favorite scripts, tips for getting started, tricks for developers, tutorials, and tell us about your favorite programming languages and communities.

Security: Tips and tricks for securing your systems, best practices, checklists, tutorials and tools, case studies, and security-related project updates.

Keep in touch!

Sign up to receive roundups of our best articles, giveaway alerts, and community announcements.

Visit opensource.com/email-newsletter to subscribe.



Our holiday giveaway series continues! This week, we're giving away
System76 Galago Pro laptop with Linux preinstalled. The grand prize
will also receive a System76 T-Shirt.

Specs: i7-8550U processor (up to 4.0 GHz), 8GB DDR4 SDRAM,
SSD, 13.3" 3200 × 1800 HiDPI display. Further upgrades available

Already a System76 superfan? Check out the Try Pop!_OS Giveaway for a c
free swag and a trip to their headquarters in Denver, Colorado.

We're wrapping up the fourth and final week of our holiday giveaway series in style: With a LulzBot 3D printer from Aleph Objects!

The Mini can print creations of up to 152mm x 152mm x 158mm and ships with a one-year warranty and free year of technical support.

So, what are you waiting for? **Enter by this Sunday, December 24, at 11:59 p.m. ET** for a chance to win!

Opensource.com aspires to publish all content under a Creative Commons license but may not be able to do so in all ca
responsible for ensuring that you have the necessary permission to reuse any work on Opensource.com. Red Hat and th
are trademarks of Red Hat, Inc., registered in the United States and other countries. Linux is a registered trademark of Li

contact us | unsubscribe | about Opensource.com | redhat.com

© 2017 Opensource.com

WEEKLY NEWS

- An introduction to DomTerm, a Linux terminal emulator
  Meet DomTerm, a terminal emulator and multiplexer with HTML graphics and other unusual features.
- 7 open source project management tools for agile teams
  A quick look at software that supports Scrum, Kanban, and other agile methods.
- Parsing HTML with Python
  With a little scripting, cleaning up documentation and other large sets of HTML files can be easy. But you need to parse them first.
- Refreshing old computers with Linux
- A history of low-level Linux container runtimes
- How I coined the term 'open source'
- 4 new OpenStack tips and guides

FREE DOWNLOAD

- .NET Core 2.0 on Red Hat Enterprise Linux cheat sheet

MORE RECOMMENDED READING

- Getting started: Writing your first Ansible playbook
- A connector, a container, and an API walk into a bar...
- 6 reasons I started using containers