The Open Source Observability Landscape

Nathan LeClaire @dotpem

honeycomb.io

The Open Source Observability Landscape

Storing and Accessing Telemetry

Are you feeling overwhelmed by the huge explosion of choices for observability and monitoring in open source? How you are expected to develop and deploy applications is changing rapidly — from monoliths to microservices and more, complexity keeps growing. Luckily, however, there's been a renaissance in open source innovations as projects including ELK, Prometheus, and Jaeger grow in popularity and usage. You might have even heard of service meshes like Istio that can generate telemetry automatically.

How do you make sense of all these tools and how do they stack up when it comes to solving problems quickly? This guide is written to help you make sense of the most popular open source methods of storing and accessing telemetry, and how they relate to Honeycomb. In this paper, we'll cover the storage systems for persisting and accessing observability data — there are also open source projects such as OpenTelemetry that define a format for exporting data to these storage systems, but we'll save discussing those for another day.

Intro

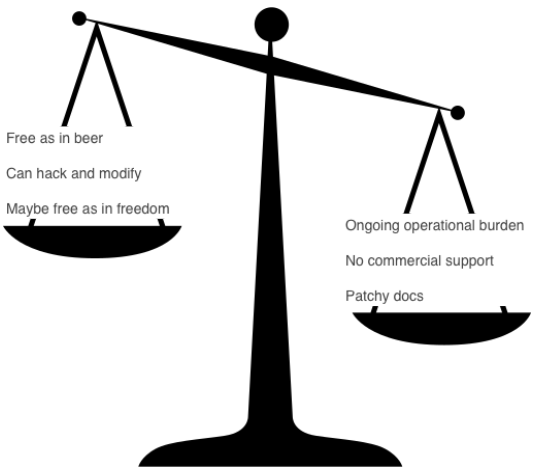
It used to be a simpler time — you had fewer services, often deployed to only one powerful server, that you were responsible for. Most of the things that could go wrong with these monoliths were well understood. You logged slow database queries as they happened and added MySQL indexes. You spotted network issues easily with high level metrics. You monitored disk space and memory usage remedied shortages when needed. When something went wrong, the path to finding out what happened was straightforward — likely you would access a shell on a server, read the logs, deduce what was happening and act accordingly to fix it.

Eventually, however, the Internet became a victim of its own success — traffic had to be served to more people, teams had to scale out their development, and

novel challenges began to brew. Given Linux's critical role in enabling large scale web applications in the first place, it's no surprise that the open source heritage of the infrastructure carried forward to new troubleshooting tools. Programmers solving their own problems began to build out the next generation of tooling to solve these complicated new issues and you stand to benefit by leveraging their work and experience.

General Costs and Benefits of OSS

While we will get into more specific details about projects further along in this guide, it's worth reflecting on the general cost/benefit analysis for using open source tooling. One big advantage of open source tools is obvious — the sticker price is $0, and anyone can freely download and start tinkering with the software for their own needs immediately. There's also a lot of appeal to the open source aspect in terms of addressing your own needs. If something goes wrong when running the software, you can go dive into the code and debug it. Likewise, if you have a particular need for a feature, no one is stopping you from developing it and working to get it contributed upstream. Open source politics aside, this is frequently a powerful model, and especially for those for whom ideological purity is important, the use of software where you have full unfettered access to the source code is attractive.



As is the case with most things in life, there's a counterpoint to every advantage listed above. While it's zero cost to start using the software, resources need to be invested in to maintaining it, and especially for observability-scale infrastructure, some of the underlying data stores can get pretty operationally hairy. That uses up valuable person-cycles that should be available for developing new features and keeping the apps themselves running smoothly. Documentation and support on such projects is frequently patchy and on a best-effort basis -

overworked and frustrated open source maintainers won't take your requests as seriously as a business whose well-being depends on it.

We at Honeycomb are a vendor, and we encourage you to pay for software when it suits your needs. A vendor can often be your best friend by freeing you up to shave fewer yaks and do more of what you really want to do - deliver more software. That said, we think open source and vendors coexist quite well - for instance Prometheus, with its relative ease of operation and solid handling of metrics data, makes a natural complement to buyers and users of Honeycomb.

The Tools

We'll go through a few examples of open source tools and describe what they are, how data is ingested into them, and a "Honeycomb Hot Take" on how their strengths and weaknesses compare to Honeycomb. In general, these tools fall into one of three buckets - logs, metrics, and tracing tools. Some folks have called these the three pillars of observability, but we don't particularly agree with that characterization — instead, we think there's a lot more potential in the observability movement than what these tools alone have to offer. Some of the outlined issues with the tools we'll discuss might help illuminate why.

The tools we will go over are:

1. *Prometheus* – A time series database for metrics
2. *Elasticsearch/Logstash/Kibana* – commonly called "ELK" for short - A log storage, processing, and querying stack
3. *Jaeger* – A system for distributed tracing

There are other, similar tools, to some of these, which we will note in section - but these hit the primary points of what's available in the "open source market" today.

Prometheus

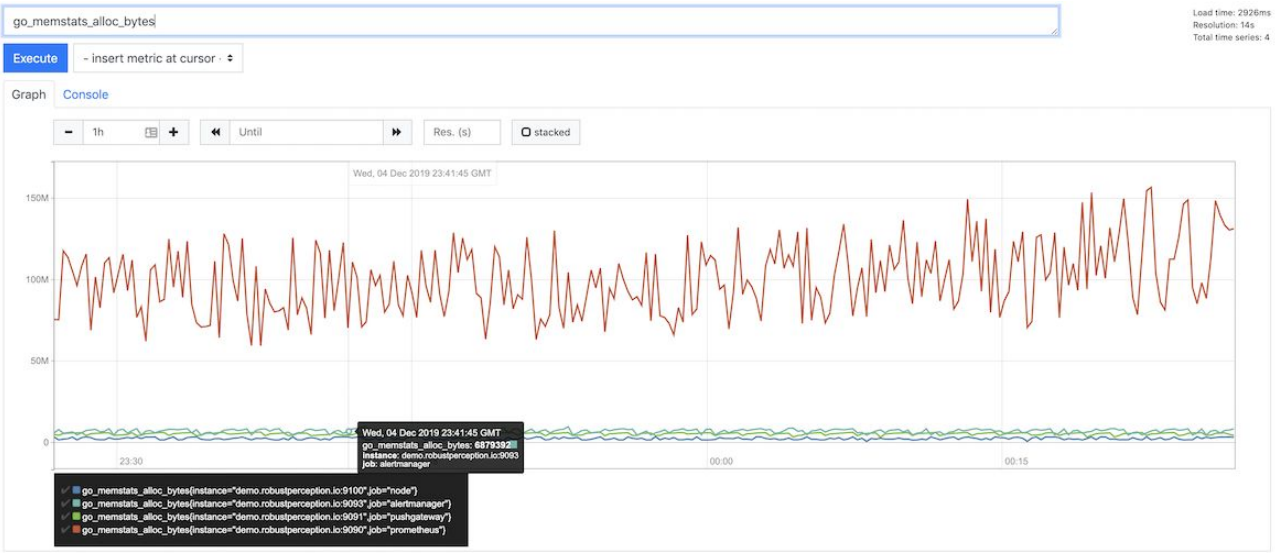
What is it?

Prometheus is a time series database for metrics monitoring. Its popularity has grown alongside the groundswell of popularity around open source container deployment methods such as Docker and Kubernetes. In metrics, you store a numeric time series representing what is happening in your system. These time series can be counters, which have a value that resets at each time bucket, or gauges, which are a cumulative number that goes up or down over time. An example of a counter would be the number of HTTP requests seen — every time a new request comes in, you increment the counter. A gauge, on the other hand, would be something like the number of running Kubernetes pods - that will go up and down over time, and it represents a cumulative number. Prometheus also has support for two more sophisticated types, histograms and summaries, which allow you to analyze more sophisticated distributions of data.

What would attract you to Prometheus? Its vibrant open source community, wide variety of integrations to get data in, and a more mature data model than traditional metrics systems like Graphite are all big selling points. In Prometheus, you can give each metric labels, which group metrics into segments of interest. For instance, a metric tracking HTTP request counts can be split according to status code, allowing us to know the number of HTTP 5XX errors. This is a big leap forward from Graphite, where metrics didn't have any notion of such metadata, and segmenting had to be done using kludgey solutions such as giving each metric a unique name (`app.http.status.500` , for instance).

Prometheus also supports a dynamic query language, PromQL, which enables you to explore the underlying data in a rich and flexible manner. Prometheus is

often paired with a dashboarding tool such as Grafana, which provides you an attractive and usable frontend for the underlying time series data. Prometheus also has a default UI for quick querying with PromQL directly.



Getting Data In

Prometheus is primarily based on a polling/scraping system, which will issue simple HTTP requests to targets who expose data in a plain text format. The format looks like this.

```
# HELP http_requests_total The total number of HTTP requests.
# TYPE http_requests_total counter
http_requests_total{method="post",code="200"} 1027 1395066363000
http_requests_total{method="post",code="400"}    3 1395066363000
```

Apps that publish Prometheus metrics therefore serve this format on an endpoint such as `/app/metrics`. Using Push Gateway, processes that are not long-lived (such as background jobs) can also push metrics to Prometheus. Getting data in to Prometheus, then, entails that you export this text format somehow. In the case of host level metrics, you can install the Node Exporter on every host to get some solid out-of-the-box scrape pages for things like CPU, memory, and disk usage.

The most valuable metrics such as error rates are exported from apps themselves in code that relies on Prometheus client bindings. For instance, to use a counter, you would update your code to have Prometheus-specific code like the following example.

```
class YourClass {
    static final Counter requests = Counter.build()
     .name("http_requests_total")
     .help("Total requests.")
     .labelNames(
         Arrays.asList(
         "http_status",
         "http_method"
     ))
     .register();

    void processGetRequest() {
        requests.labels("200", "GET").inc();
    }
}
```

Honeycomb Hot Take

We think having some basic metrics is essential, but our years of experience solving problems in production quickly led us to develop Honeycomb due to the fact that metrics do not support rich, high resolution data. Prometheus discourages you from using data that has too many dimensions, which you will often find is the best type of data for solving problems quickly. The following caveat is from the Prometheus documentation.

> CAUTION: Remember that every unique combination of key-value label pairs represents a new time series, which can dramatically increase the amount of data stored. Do not use labels to store dimensions with high cardinality (many different label values), such as user IDs, email addresses, or other unbounded sets of values.

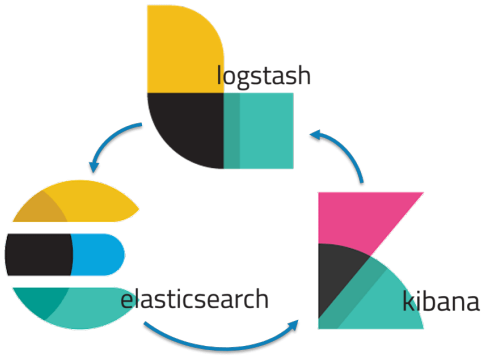
It's easy to send system data such as CPU, memory, and disk throughput to Prometheus, but these metrics are usually symptoms of problems, not the problems themselves. You are forced to go digging in other tools to divine the actual problem.

As one Prometheus user writes at https://www.robustperception.io/cardinality-is-key:

> *A Prometheus 2.x can handle somewhere north of ten millions series over a time window, which is rather generous, but unwise label choices can eat that surprisingly quickly... Some particular things to watch out for are breaking out metrics with labels per customer. This usually works okay when you've tens of customers, but when you get into the hundreds and later thousands this tends not to end well. Increasing the number of buckets in your histograms also tends to go sour, as histograms are often also broken down by other labels so the growth of both combines. It's not at all unusual that over half the resource usage of a Prometheus is due to less than ten metrics, and moving the label values into the metric name doesn't make a difference - it just makes your life harder!*

So where do metrics fit? Metrics are fantastic for you to track high level details such as disk usage, vanilla HTTP request counts, queue lengths, and so on. Those top level details can help you know when to dive in to a tool that handles the cardinality without a sweat such as Honeycomb. For instance, an alert generated by Prometheus could include a Templatized Query Link that allows you to jump right to a relevant Honeycomb query.

ELK



What is it?

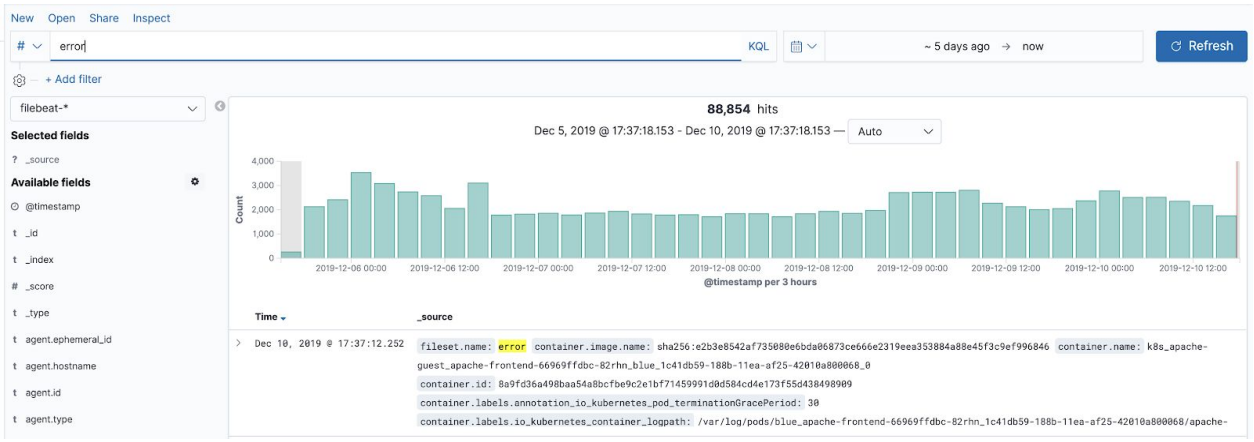
The "*ELK stack*", as you might hear it called, is composed of three components.

1. *Elasticsearch* – A text search engine that stores and makes available string data for searching.
2. *Logstash* – A program for parsing structure from logs and "tee"-ing them to various outputs such as Elasticsearch.
3. *Kibana* – A front end for querying data stored in Elasticsearch.

Most teams have a high volume of logs. For one thing, just about every process under the sun emits them. Linux core services alone spit out a ton of log data. Likewise, if you are relying on open source tools such as Docker and Kubernetes for your core infrastructure, you are at the mercy of whatever those systems export to understand them, and the main thing they export are plaintext logs. Most engineers learn how to print text to the console ("Hello World!") as their first program, and that natural reflex of printing output to understand the internal workings of a program doesn't ever leave. Logs work beautifully for understanding what's happening with your system on a single machine (e.g., when developing). On a distributed cluster, things get more complicated, and tools such as the ELK stack are needed for it to be feasible.

Elasticsearch grew up as an extension of the Apache Lucene search engine built in Java. As Elasticsearch grew a larger open source community, users began to

look for ways to get data into it as well as to visualize and query the stored information. Logstash, a way to parse and "stash" logs in whatever storage system was appropriate, evolved as an open source project to fit the former piece. Using Logstash, specific properties could be yanked from an otherwise unstructured line for easier troubleshooting, and logs could be sent to a variety of backends including Elasticsearch. Kibana evolved as the final piece of the trilogy to offer a UI for more easily querying and visualizing the relevant information.



Getting Data In

Since most programs expect to write their logging output to STDOUT or STDERR, the question of how to get that data in to downstream systems like Elasticsearch is answered by figuring out what glue layers you need to access those streams. In traditional deployments, you might have processes running on a server which write their logs to a file that you could then "tail" using Logstash's file input plugin. You could also write these logs to a common log sink like syslog or journald which logstash can also consume from. In the container era, systems such as Docker and Kubernetes often support exporting these logs to the proper destination natively. With a few configuration file settings, the container systems can take care of exporting logs in the proper input expected by Logstash for you.

If you're really good, you will be doing structured logging. Structured logging allows you to output the metadata of a log line in a computer readable format such as JSON, and can help to alleviate the weight of a middle system like

Logstash trying to parse relevant pieces out of otherwise messy and flattened log messages.

Honeycomb Hot Take

Independent of the stack used to produce the end result of log storage and querying, we find that logs have fundamental issues which blunts their utility for observability. You will likely find that centralized logging, while popular, tends to become too voluminous to be practical for fast production troubleshooting.

Log storage backends are optimized for plaintext search at the cost of other types of searching and aggregating, and answering your high level questions such as “Who is seeing this specific error?” and “When are they seeing this error?” becomes impractical. The signal to noise ratio is too low to yield consistent results. Additionally, futuristic features such as tracing and BubbleUp are table stakes for solving problems quickly in the modern world, but you will find that many centralized logging systems don’t support them.

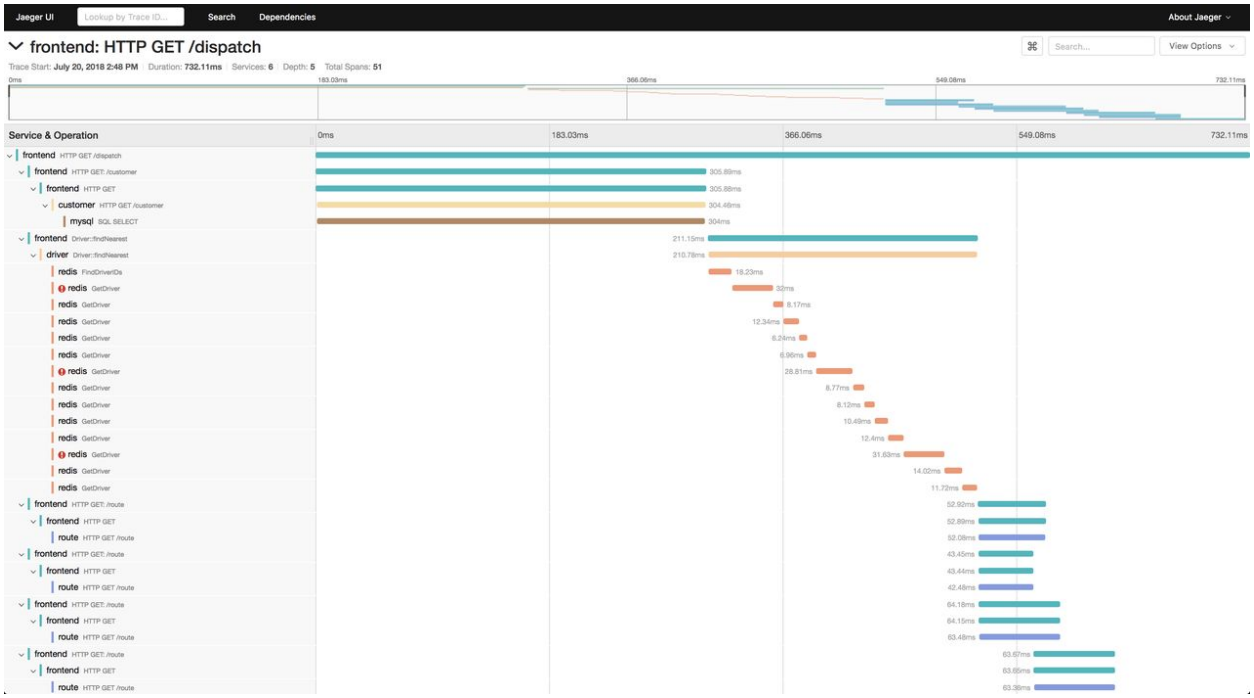
Logstash is written in Ruby and you must tune its performance in terms of memory and CPU usage carefully. You also have to monitor Logstash itself to avoid ingest lag or loss of logging data. You definitely do not want the system meant to help understand and eliminate problems to cause its own incidents if it gets backed up, and with the high volume most logging systems deal with, this is a very real possibility. Elasticsearch in particular is notoriously difficult to operate and scale out. Operational considerations should weigh heavily in your decision to move forward with a particular monitoring or observability tool, and we think you’ll find that with logging systems operations has a high cost.

Jaeger

What is it?

Jaeger is an open source solution for distributed tracing originally developed at Uber. Distributed tracing allows you to see the “flow" of one request as it moves through various services in the system. Once upon a time, web apps were frequently served from only one monolithic instance of a service. Over time, the complexity of the web applications that we deploy grew and understanding which component of a system was having issues became more difficult. For instance, imagine we have three services (A, B, and C) deployed and the end user communicates only with Service A. Service A calls out to Service B which in turn calls Service C to yield the end result of a given request. Understanding what’s happening in the system can become quite complicated because any slow down or error in Service C will affect Service B and Service A in turn.

Distributed tracing helps you to visualize these relationships more effectively using a “waterfall” style visualization that shows you which steps happened in sequence and which steps are taking the most time. Calls to upstream services are nested as spans within other spans such as the root span that initiated the distributed call graph. Everything in a given trace is tied together using a unique trace ID that is propagated across the network alongside service calls. Each span contains a set of associated metadata that describes what happened — hence, in addition to being able to visualize the latency of each step, you can also examine if errors were returned, which users were associated with the call, and in some cases, the IDs of other traces associated with the one under examination.



Getting Data In

In order to get data in to Jaeger, you must instrument your code. In addition to being the underlying storage system for accessing trace data, Jaeger also defines a protocol for sending tracing data that is compatible with the OpenTracing standard. You use libraries provided by the Jaeger format to send telemetry to Jaeger describing what is happening in your production apps. Let's take a look at what that entails using an example from the Jaeger Go Client's source code.

First we must initialize the Jaeger client, which we configure with various settings including how we should determine what data to sample, where Jaeger itself should log information about what is happening, and where we should send metrics about what the tracer code itself is doing.

```
cfg := jaegercfg.Configuration{}
jLogger := jaegerlog.StdLogger{}
jMetricsFactory := metrics.NullFactory{}
tracer, err := cfg.InitGlobalTracer(
    "serviceName",
    jaegercfg.Logger(jLogger),
    jaegercfg.Metrics(jMetricsFactory),
)
if err != nil {
    log.Printf(
       "Could not initialize jaeger tracer: %s",
       err.Error(),
    )
    return
}
opentracing.SetGlobalTracer(tracer)
```

In this example, we sample every span created (`jaeger.SamplerTypeConst`), we don't send the metrics anywhere (`metrics.NullFactory`), we log to `STDOUT` (`jaegerlog.StdLogger`), and we have indicated that we are tracing a service named `serviceName`.

This is all boilerplate setup and to actually emit data about what's happening in our app once this configuration is created, we must create spans. A common place to start is to create a span for each HTTP or RPC call.

In our application's HTTP handler, then, we would have some code like the following. We create a `Span` at the beginning of the call and add some associated metadata as tags using the `Span`'s `SetTag` method.

```
root := tracer.StartSpan("root")
root.SetTag("path", "/foo/bar")
root.SetTag("http_method", "POST")
// ... code for the actual work of the HTTP request ...
root.SetTag("http_status_code", statusCode)
root.Finish()
```

At any point, we could nest new spans (so called child spans) within the original span by calling the tracer again and passing in the "context" indicating where to spawn a new span from. This will "chunk out" various operations into their own distinct parts of the waterfall tree.

```
dbSpan := tracer.StartSpan(
    "db_call",
     opentracing.ChildOf(root.Context()),
)
dbSpan.SetTag("query", query)
execQuery(query)
dbSpan.Finish()
```

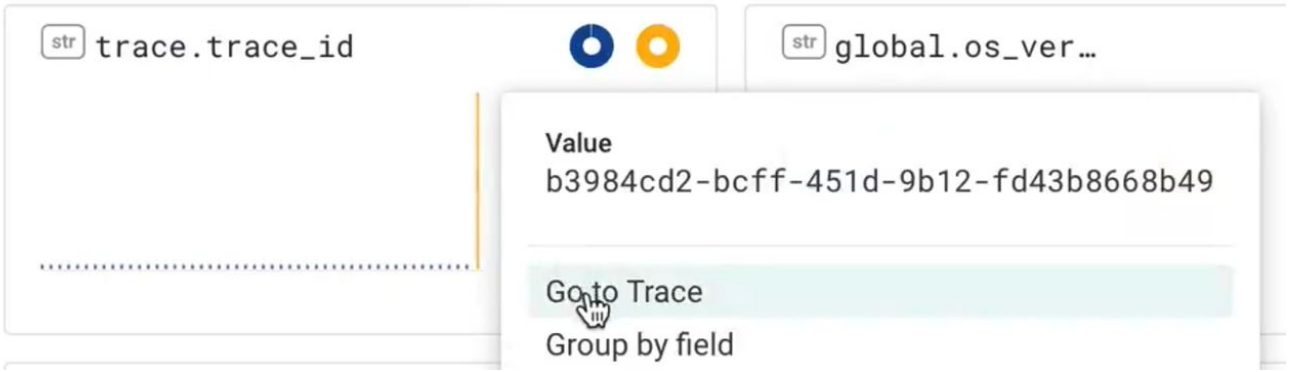
You can abstract much of this functionality into shared libraries that perform this tracing boilerplate automatically. This is what projects such as the Honeycomb Beelines do. Some projects such as Envoy allow you to generate span data automatically for every request that goes through a proxy instance, but code modification is still required to ensure that headers indicating which trace a given request is part of are passed along.

Honeycomb Hot Take

It's great that systems such as Jaeger exist to allow you to experiment with distributed tracing, but we think you will find they have limitations when it comes to the goal of solving problems quickly. For one thing, operational sophistication is still required to run your own tracing system at home. Jaeger has a nifty in-memory storage backend for traces that's useful for experimentation, but not feasible for production. Instead, if you use Jaeger in production you must choose between Apache Cassandra or Elasticsearch for

persisting traces. As we discussed in the ELK section, running yet another complicated data store with multiple failure modes is not a good tradeoff for most organizations, and it requires expertise that is difficult to access.

Jaeger has some limited support for filtering down to certain types of data but does not have options for sophisticated analysis and segmenting data. You will likely find Jaeger good for accessing a trace when you know you're interested in it, but difficult to use to find which trace you are interested in. Contrast that to Honeycomb, which supports very fast aggregate queries and allows you to BubbleUp to find the traces or trends of interest. You can even jump right to a trace associated with a highlighted problem.



The OpenCensus Collector can be leaned on to "tee" traces to Honeycomb as well as to other backends such as Jaeger. So if you are already sending to a tracing system such as Zipkin or Jaeger, you don't need to turn it off just to try Honeycomb. Traces can be sent to multiple backends at the same time.

Coexist for the Best Observability

Ultimately, every team will have different needs, and you will want to leverage different combinations of tools. One of the things we're most excited about at Honeycomb is folks using a project like Prometheus, with its low cardinality but excellence at metrics, alongside a system like Honeycomb which fits many of the complementary high cardinality use cases. Likewise, if you dump plain text logs into Honeycomb, you'll be disappointed, but teeing raw logs off to S3 using Logstash while sending high resolution detail to Honeycomb might allow you to hit a sweet spot of mostly rich data available with logs as a fallback.

As we briefly mentioned at the start of the paper, here we're covered only the storage systems for accessing generated data here, but innovation is happening in the open source standards for emitting the data as well. A new era of quality telemetry data is being ushered in as work on the OpenTelemetry project matures and more teams begin using it. OpenTelemetry promises to unify various systems into a single coherent model of emitting application telemetry, and is likely to reduce the overhead of switching between various storage systems as well. In the future, look for OpenTelemetry to gain popularity as it makes getting data into various storage systems much more seamless. Until then, coexisting for the best observability will help most teams to gain the maximum benefit from what's available in the world of open source.

Enjoyed this whitepaper? We'd love to set up a demo with you and learn more about your use case.

About the author

Nathan LeClaire (@dotpem) is a Go programmer and author who enjoys open source and DevOps. He lives in San Francisco, CA and currently works as a sales engineer for Honeycomb.

honeycomb.io

About Honeycomb

Honeycomb provides observability for modern dev teams to better understand and debug production systems. With Honeycomb teams achieve system observability and find unknown problems in a fraction of the time it takes other approaches and tools. More time is spent innovating and life on-call doesn't suck. Developers love it, operators rely on it and the business can’t live without it.

Follow Honeycomb on Twitter LinkedIn
Visit us at Honeycomb.io