elastic

# Building a data foundation for modern observability

# Table of contents

# The case for telemetry data to manage your cloud applications

Modern application architectures are distributed and run hundreds of different services in a hybrid environment. They're running in multiple locations, on different cloud providers, and even on different continents. As software systems get more complex, it's imperative to collect and observe telemetry data, enabling you to understand the components in the system and the interactions between them, and to triage and act upon any problems that may arise.

Telemetry in software systems generally refers to the collection of data relevant to the performance of applications, services, and the infrastructure that they run on. Effectively collecting and storing telemetry data helps you make the most of your observability solution. Missing or inconsistent data makes it harder to troubleshoot future application issues.

This ebook will take you through the basics of telemetry and observability data, then show you how emerging projects like OpenTelemetry are shaping industry standards to transform modern observability platforms.

# Types of telemetry and monitoring data

When monitoring applications and infrastructure, telemetry data starts with four main types: logs, metrics, profiling, and traces. All four signals provide valuable insights for developers, architects, DevOps, and site reliability engineers (SREs). Combined with profiling and newer observability data types, these four golden signals provide a holistic view of your deployments and help you to identify and resolve issues. We'll walk through each of these data types and break them down a bit more.

## Logs: a primary signal for observability

Log messages, or logs, can come from several layers in your infrastructure or application stack. They can come from your infrastructure (hosts, servers, routers, or switches); from services like databases, message stores, or orchestration platforms; and, of course, from any applications that you write.  Log entries are created when something eventful happens in a piece of code. Or, more specifically, when a certain point in the code has been reached — a web page has been hit, an order has been placed, or a query took too long. The way to create a log message (let's just call these *logs* now) varies based on the programming language being used. But they tend to be calls that look something like `printf()` or `System.out.println()` . If you've ever tried to learn any programming language, you've probably written a log or seen something like this:

```
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello, World!");
    }
}
```

This code snippet would print out the ubiquitous software phrase "Hello, World!"

Actual logs will hopefully contain a bit more information. Logs can be structured, unstructured, or somewhere in between. They will often have a severity level associated with them, which gives you a bit of control over how "chatty" they are.

Logs, by their nature, are a "point-in-time" resource. They don't often carry the context of what happened earlier in a transaction or even all of the data associated with the request. Logs can also be pretty chatty, especially if you have the verbosity level turned up. Let's break down logs a bit further and then talk about other types of telemetry and how they can help you see the bigger picture.

## Unstructured and semi-structured logs

Unstructured, basic, or plain-text logs are basically free-form sets of characters gathered together. They'll often be human-readable and might even read like sentences. They can be single-line or multi-line in the same file, which can make parsing tricky. You might even find a mix of semi-structured and plain text in the same log file. Plain-text logs don't have any predefined structure and are just free-form with whatever the developer thought was important at the time. They might only consist of a payload and, if you're lucky, a timestamp.

```
2021-04-14T14:05:58.019Z Entered <processCard>

2021-04-14T14:05:59.123Z Calling <cardValidation> with [13]
digit card number

2021-04-14T14:05:59.723Z back from luhn algorithm, passed

2021-04-14T14:06:00.123Z card starts with [37] so it's an
american express
```

In the example above, the logs are pretty easy to read, and while it looks like the developer was trying to delimit fields, they are all free-form.

Semi-structured logs, on the other hand, have a somewhat predefined format. For example, a log from an Apache web server might look like this:

```
::1 - - [26/Dec/2020:16:16:29 +0200]
"GET /favicon.ico HTTP/1.1" 404 209

192.168.33.1 - - [26/Dec/2016:16:22:13 +0000]
"GET /hello HTTP/1.1" 404 499 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12;
rv:50.0) Gecko/20100101 Firefox/50.0"
```

At a glance, it's not obvious what's what. The log looks like it contains a date and a timezone offset, along with a few other fields. The second line has more data than the first, and now it looks like the first entry is an IP address. In reality, each line conveys the requesting IP address, a couple of fields for identity (which in this case were blank, hence the `-` ), the timestamp (in the `[ ]` ), the method and page being requested, along with the version (in quotes), the response code, and the size of the response in bytes. The second entry has a couple of additional fields: the referrer (again, blank) and the user agent, which describes the browser that the client used.

A more complex example comes from an entry in the MySQL slow log:

```
::1 - - [26/Dec/2020:16:16:29 +0200]
"GET /favicon.ico HTTP/1.1" 404 209

# Time: 2021-08-09T14:01:47.811234Z

# User@Host: root[root] @ localhost [] Id:    14

# Query_time: 2.475469  Lock_time: 0.000287
Rows_sent: 10  Rows_examined: 3145718

use employees;

SET timestamp=1628540304;

SELECT last_name, MAX(salary) AS salary FROM employees
INNER JOIN salaries ON employees.emp_no = salaries.emp_no
GROUP BY last_name ORDER BY salary DESC LIMIT 10;
```

Rather than the approach taken by the Apache logs, which rely on punctuation and whitespace to delimit fields, this log has more of a two-dimensional approach. The lines that start with the `#` provide some context. The first `#` is the timestamp, the second `#` is identification, and the third `#` has some statistics. Note that each of those lines can also be further broken down. Finally, the remainder of the entry shows the time and the query that was slow.

Log aggregation tools will often parse plain-text and semi-structured logs into individual log entries. Semi-structured logs make it a bit easier to break down log entries into discrete fields, but there's still some amount of parsing that needs to be done to accomplish this. For common formats such as the Apache and MySQL slow logs, log aggregation tools will parse out the individual fields such as `query_time` and `user` information for the slow log. Or the `response_code` and `response_size_in_bytes` from the Apache httpd log. In the case of free-form or plain-text logs, you'll likely just get a timestamp (which, if not provided in the log message, will be added at the time of import) and the payload or text.

Another example of semi-structured logs is system logs ( `/var/log/syslog` on Ubuntu, or `/var/log/messages` and `/var/log/secure` on CentOS), which may have any number of formats within them.

```
22b056fea322c83c7266352fa0950011037bc1f17a9ec89f432 error:
context deadline exceeded"

Aug 29 04:48:27 build-host-97 auditd[405]: Audit daemon
rotating log files

Aug 29 04:48:31 build-host-97 containerd: time="2021-08-
29T04:48:31.279212236Z" level=info msg="shim disconnected"
id=f1bc5d1169ad440784

9fa0deee83cb4764ff22274867f402e2122c8d1e46210b

Aug 29 04:48:31 build-host-97 containerd: time="2021-08-
29T04:48:31.279329273Z" level=error msg="copy shim log"
error="read /proc/self/fd/100: file already closed"

Aug 29 04:48:31 build-host-97 dockerd: time="2021-08-
29T04:48:31.279261706Z" level=info msg="ignoring event"
container=f1bc5d1169ad44078

49fa0deee83cb4764ff22274867f402e2122c8d1e46210b
module=libcontainerd namespace=moby topic=/tasks/delete
type="*events.TaskDelete"
```

```
Aug 29 04:48:31 build-host-97 kernel: br-bc0db7c1b74e:
port 22(veth4f2bb06) entered disabled state

Aug 29 04:48:31 build-host-97 NetworkManager[523]: <info>
[1630212511.3521] manager: (vethd1243bb): new Veth device
(/org/freedesktop/NetworkManager/Devices/38192)

Aug 29 04:48:31 build-host-97 avahi-daemon[480]:
Withdrawing address record for fe80::d430:2dff:fe92:1516
on veth4f2bb06.

Aug 29 04:48:31 build-host-97 kernel: br-bc0db7c1b74e:
port 22(veth4f2bb06) entered disabled state

Aug 29 04:48:31 build-host-97 avahi-daemon[480]:
Withdrawing workstation service for vethd1243bb.

Aug 29 04:48:31 build-host-97 kernel: device veth4f2bb06
left promiscuous mode

Aug 29 04:48:31 build-host-97 kernel: br-bc0db7c1b74e:
port 22(veth4f2bb06) entered disabled state
```

As you can see, logs can be formatted to be easy to read, but when each application and service has a different format, it is harder to aggregate the logs from your entire application stack.

## Structured logs

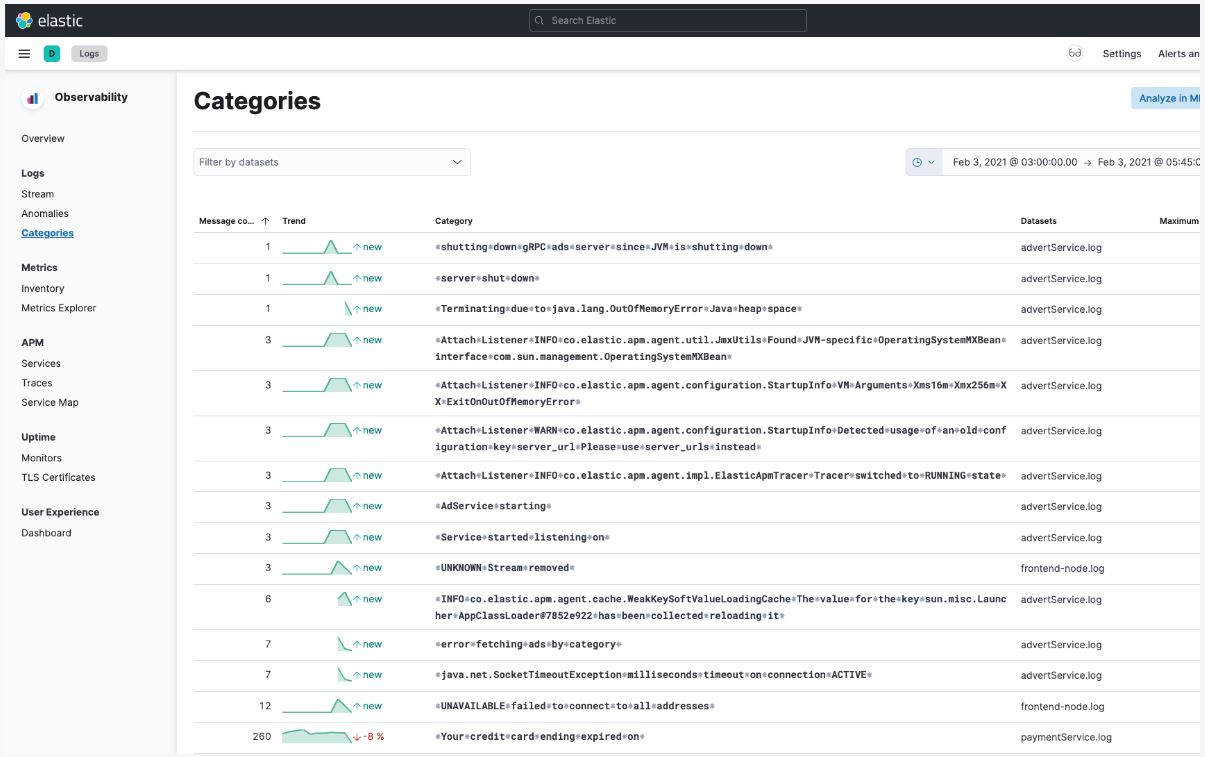
While semi-structured logs make it easier for machines to import or ingest logs, structured logs take the guesswork out of parsing. How? They start out parsed. By leveraging JSON formatting as a logging format, the fields and their values can be made explicit.

```
[
  {
    "current_user": "root",

    "lock_time.sec": 0.000287,

    "query": "SELECT last_name, MAX(salary)
AS salary FROM employees INNER JOIN salaries
ON employees.emp_no = salaries.emp_no GROUP
BY last_name ORDER BY salary DESC LIMIT 10;",

    "query_time": 2475469000,

    "rows_examined": 3145718,

    "rows_sent": 10,

    "schema": "employees",

    "thread_id": 14,

    "timestamp": "2021-08-09T14:01:45.000Z",

    "user_domain": "localhost",

    "user_name": "root"
  }
]
```

For example, if the above slow log entry had been structured at the beginning, it would look like this.

This newly formatted record has the same information as the multi-line, plain-text entry, but the individual fields have been identified and can be centrally aggregated, searched, filtered, and examined. There's also a benefit when exploring via the command line as well. When this information was in an unstructured, multi-line format, it would have required some pretty complex command-line skills to find all queries against the `employees` table, which took more than 1.5 seconds. Now that it has discrete fields, you can leverage command-line tools that understand JSON, such as jq, to query and filter.

## Log volumes and "signal to noise"



# Categories

| Message co... ↑ | Trend | Category | Datasets | Maximum |
|---|---|---|---|---|
| 1 | ↑ new | `*shutting*down*gRPC*ads*server*since*JVM*is*shutting*down*` | advertService.log | |
| 1 | ↑ new | `*server*shut*down*` | advertService.log | |
| 1 | ↑ new | `*Terminating*due*to*java.lang.OutOfMemoryError*Java*heap*space*` | advertService.log | |
| 3 | ↑ new | `*Attach*Listener*INFO*co.elastic.apm.agent.util.JmxUtils*Found*JVM-specific*OperatingSystemMXBean*interface*com.sun.management.OperatingSystemMXBean*` | advertService.log | |
| 3 | ↑ new | `*Attach*Listener*INFO*co.elastic.apm.agent.configuration.StartupInfo*VM*Arguments*Xms16m*Xmx256m*XX*ExitOnOutOfMemoryError*` | advertService.log | |
| 3 | ↑ new | `*Attach*Listener*WARN*co.elastic.apm.agent.configuration.StartupInfo*Detected*usage*of*an*old*configuration*key*server_url*Please*use*server_urls*instead*` | advertService.log | |
| 3 | ↑ new | `*Attach*Listener*INFO*co.elastic.apm.agent.impl.ElasticApmTracer*Tracer*switched*to*RUNNING*state*` | advertService.log | |
| 3 | ↑ new | `*AdService*starting*` | advertService.log | |
| 3 | ↑ new | `*Service*started*listening*on*` | advertService.log | |
| 3 | ↑ new | `*UNKNOWN*Stream*removed*` | frontend-node.log | |
| 6 | ↑ new | `*INFO*co.elastic.apm.agent.cache.WeakKeySoftValueLoadingCache*The*value*for*the*key*sun.misc.Launcher*AppClassLoader@7852e922*has*been*collected*reloading*it*` | advertService.log | |
| 7 | ↑ new | `*error*fetching*ads*by*category*` | advertService.log | |
| 7 | ↑ new | `*java.net.SocketTimeoutException*milliseconds*timeout*on*connection*ACTIVE*` | advertService.log | |
| 12 | ↑ new | `*UNAVAILABLE*failed*to*connect*to*all*addresses*` | frontend-node.log | |
| 260 | ↓ -8 % | `*Your*credit*card*ending*expired*on*` | paymentService.log | |

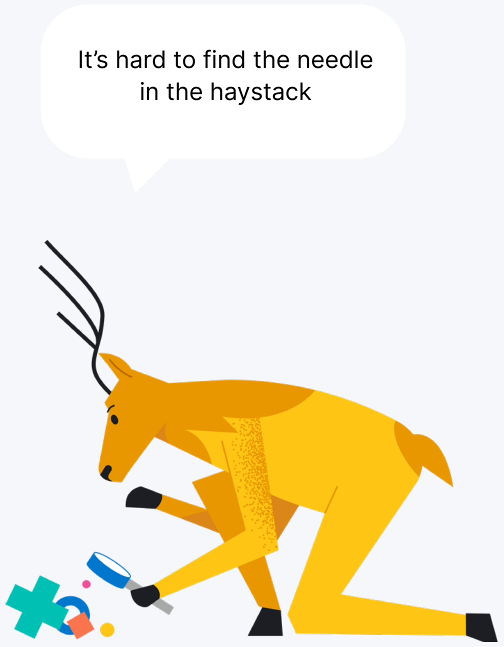


**Image: log categorization in Elastic Observability**

The software and infrastructure for a large application stack can generate a large volume of data: potentially multiple terabytes per day (or much more if you turned on that DEBUG level). This stream of telemetry data adds up quickly and can tax observability systems that don't scale well.

Logs are often a great place to look when you know that something has gone wrong and have some idea where. Unfortunately, issues and errors often get drowned out by the sheer volume of data.

For observability solutions, powerful search and filtering capabilities are crucial when sifting through large volumes of log data. When you're running multiple hosts, containers, and applications, centralizing telemetry data drastically reduces triaging time and minimizes performance issues.

## Tech Tip

### Schema-on-write versus schema-on-read

Whether your logs are structured or unstructured, their insights will be derived from your analytics and queries used against this rich data. While schema-on-write (indexing data during ingestion) is common and resource efficient, having tools that support schema-on-read (indexing raw data, applying runtime fields at query) provides more flexibility to operations teams when analyzing log data. Schema-on-read allows you to dynamically extract or query new fields after the data has been indexed and stored.

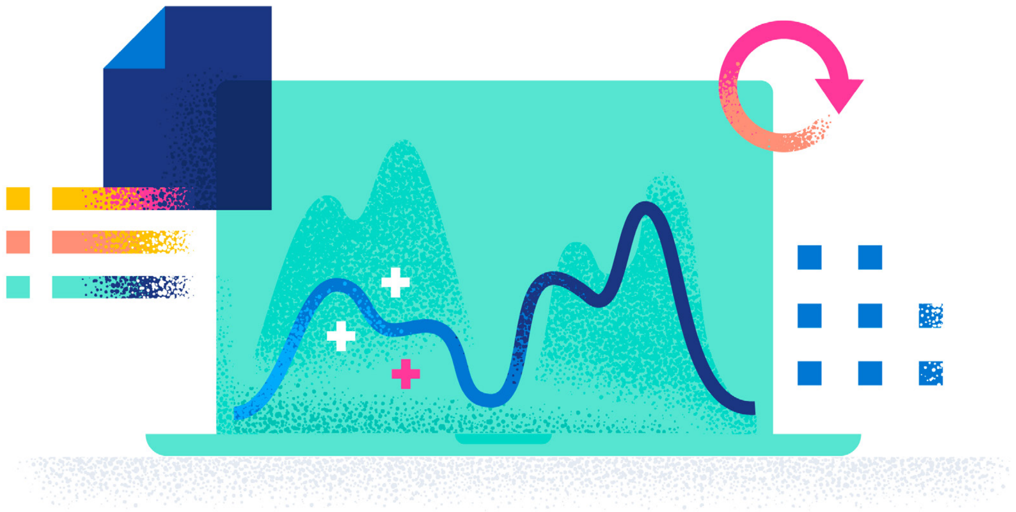
You don't always know exactly what is in the data you collect. Having both schema-on-write and schema-on-read lets you explore it further and apply the lessons learned.

## Metrics: time-series data for system behavior

While logs are generated when something happens, metrics tend to be continually updated and provide a summary of system behavior, often over a specific time period. As we've seen, logs can have numeric values embedded in them, like the `rows_examined` from the MySQL slow log example. Metrics are time series data and represent resource usage or events. These metrics could come from the operating system (CPU usage, free memory), or they might come from applications or services (failed requests, response time). Metrics are always numeric and can be whole numbers or decimal. Like logs, metrics only exist if someone had the foresight to provision for them. Metrics need to be explicitly gathered, calculated, and made available. And they can only provide the details that they are configured to deliver.

Metrics don't just need to be programmed — an understanding of what they signify is important, too, because they are usually a "point-in-time" view of the data. For example, if we track memory usage every minute, everything could look fine, but under the covers and in between those per-minute samplings, applications may be trying to chew up more RAM and experiencing memory allocation failures. Metrics may inadvertently miss cyclical fluctuations, unfortunately.

Metrics also tend to lose value as they age — it might be important to know down-to-the-minute resource consumption for the last few days, but that level of granularity is probably not needed for events from six months ago.



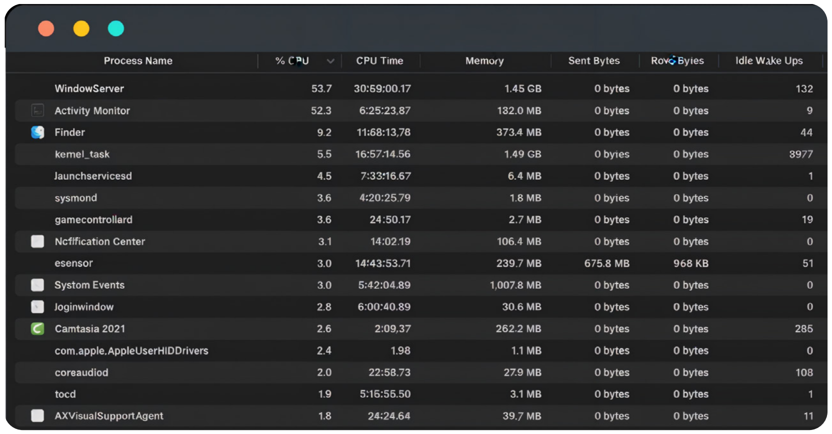
There are different families of metrics related to software and infrastructure monitoring, and you'll likely encounter the following types of metrics:

- **Counters**
- **Accumulators**
- **Utilization or ratios**
- **Aggregation metrics**

This list is not exhaustive but should serve as a good starting point. Let's take a closer look at the different types of metrics and the general use case for each, along with some examples.

There are different ways metrics get "published" — perhaps via an API or even via standard interfaces and protocols (Micrometer, Telegraf, and Prometheus are commonly used metric delivery mechanisms). How metrics are delivered doesn't impact what they mean. However they are published, it's important to note that there is definitely some "fuzziness" around the different metric types. Data may be stored in one manner, but accessed in another.

It's probably useful to use a few concrete examples of commonly used metrics when troubleshooting performance issues on a personal computer. Operating systems include tools to get a high-level overview of system performance: Activity Monitor on macOS, Task Manager on Windows-based operating systems, or simply `top` on *nix-flavored systems or macOS.



| Process Name | % CPU | CPU Time | Memory | Sent Bytes | Rovd Byies | Idle Wake Ups |
|---|---|---|---|---|---|---|
| WindowServer | 53.7 | 30:59:00.17 | 1.45 GB | 0 bytes | 0 bytes | 132 |
| Activity Monitor | 52.3 | 6:25:23.87 | 132.0 MB | 0 bytes | 0 bytes | 9 |
| Finder | 9.2 | 11:68:13,78 | 373.4 MB | 0 bytes | 0 bytes | 44 |
| kemel_task | 5.5 | 16:57:14.56 | 1.49 GB | 0 byies | 0 bytes | 8977 |
| Jaunchservicesd | 4.5 | 7:33:16.67 | 6.4 MB | 0 bytes | 0 bytes | 1 |
| sysmond | 3.6 | 4:20:25.79 | 1.8 MB | 0 byies | 0 bytes | 0 |
| gamecontrollard | 3.6 | 24:50.17 | 2.7 MB | 0 bytes | 0 bytes | 19 |
| Ncfification Center | 3.1 | 14:02.19 | 106.4 MB | 0 bytes | 0 bytes | 0 |
| esensor | 3.0 | 14:43:53.71 | 239.7 MB | 675.8 MB | 968 KB | 51 |
| Systom Events | 3.0 | 5:42:04.89 | 1,007.8 MB | 0 bytes | 0 bytes | 0 |
| Joginwindow | 2.8 | 6:00:40.89 | 30.6 MB | 0 bytes | 0 bytes | 0 |
| Camtasia 2021 | 2.6 | 2:09,37 | 262.2 MB | 0 bytes | 0 bytes | 285 |
| com.apple.AppleUserHIDDrivers | 2.4 | 1.98 | 1.1 MB | 0 bytes | 0 bytes | 0 |
| coreaudiod | 2.0 | 22:58.73 | 27.9 MB | 0 bytes | 0 bytes | 108 |
| tocd | 1.9 | 5:16:55.50 | 3.1 MB | 0 bytes | 0 bytes | 1 |
| AXVisualSupportAgent | 1.8 | 24:24.64 | 39.7 MB | 0 bytes | 0 bytes | 11 |

**Image: screenshot of Activity Monitor on macOS**

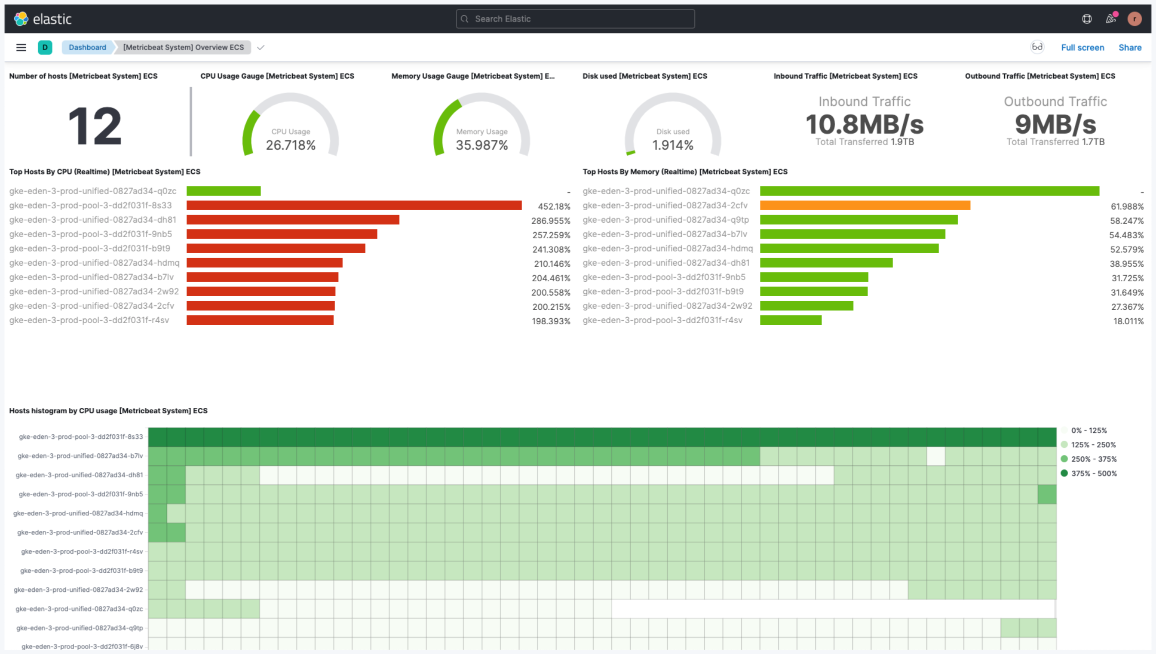

elastic
Search Elastic
Dashboard
[Metricbeat System] Overview ECS
Full screen
Share
Number of hosts [Metricbeat System] ECS
12
CPU Usage Gauge [Metricbeat System] ECS
CPU Usage 26.718%
Memory Usage Gauge [Metricbeat System] E...
Memory Usage 35.987%
Disk used [Metricbeat System] ECS
Disk used 1.914%
Inbound Traffic [Metricbeat System] ECS
Inbound Traffic 10.8MB/s Total Transferred 1.9TB
Outbound Traffic [Metricbeat System] ECS
Outbound Traffic 9MB/s Total Transferred 1.7TB
Top Hosts By CPU (Realtime) [Metricbeat System] ECS
gke-eden-3-prod-unified-0827ad34-q0zc -
gke-eden-3-prod-pool-3-dd2f031f-8s33 452.18%
gke-eden-3-prod-unified-0827ad34-dh81 286.955%
gke-eden-3-prod-pool-3-dd2f031f-9nb5 257.259%
gke-eden-3-prod-pool-3-dd2f031f-b9t9 241.308%
gke-eden-3-prod-unified-0827ad34-hdmq 210.146%
gke-eden-3-prod-unified-0827ad34-b7lv 204.461%
gke-eden-3-prod-unified-0827ad34-2w92 200.558%
gke-eden-3-prod-unified-0827ad34-2cfv 200.215%
gke-eden-3-prod-pool-3-dd2f031f-r4sv 198.393%
Top Hosts By Memory (Realtime) [Metricbeat System] ECS
gke-eden-3-prod-unified-0827ad34-q0zc -
gke-eden-3-prod-unified-0827ad34-2cfv 61.988%
gke-eden-3-prod-unified-0827ad34-q9tp 58.247%
gke-eden-3-prod-unified-0827ad34-b7lv 54.483%
gke-eden-3-prod-unified-0827ad34-hdmq 52.579%
gke-eden-3-prod-unified-0827ad34-dh81 38.955%
gke-eden-3-prod-pool-3-dd2f031f-9nb5 31.725%
gke-eden-3-prod-pool-3-dd2f031f-b9t9 31.649%
gke-eden-3-prod-unified-0827ad34-2w92 27.367%
gke-eden-3-prod-pool-3-dd2f031f-r4sv 18.011%
Hosts histogram by CPU usage [Metricbeat System] ECS
0% - 125%
125% - 250%
250% - 375%
375% - 500%

Image: screenshot of Metricbeat activity monitor

The first few columns of the screenshot above show the following commonly used performance metrics for any machine:

- Percentage of CPU (`% CPU`)
- CPU Time
- Memory
- Sent Bytes
- Received Bytes (`Rcvd Bytes`)

## Counters

In general, counter metrics count things and are incremented by one each time something happens. Some common examples are things like `page_faults` , which essentially track how many times an application tries to access virtual memory that isn't loaded in physical memory. Another good example if you're familiar with web services is `page_views` — the number of times a webpage has been accessed.

While counters usually increase over time, counters can also be used to keep track of counts. In this case, they might indicate things like the number of open files, outstanding requests, or people in line.

Counters can be cumulative in that they keep track of the value since the beginning of a process — like when a program starts or when the host machine was last rebooted — but they can also be based on a set time period.

## Accumulators

Accumulators are similar to counters, but rather than incrementing by one, they get incremented (or decremented) by a value when something happens. They can also be measured by period, since startup, or since forever. A few accumulators from the list are `sent_bytes` , `received_bytes` , and `cpu_time` . If you're plotting out accumulators over time, make sure that you're plotting the deltas, and not the cumulative sum (unless that's what you want).

## Usage metrics

Usage metrics are generally "point-in-time" metrics that get checked periodically. Examples include CPU or memory usage, which are often shown as a gauge. Whereas accumulators are keeping track, usage metrics are simply the state of something at a given point in time. It's important to note that how metrics are calculated is different from how metrics are accessed. The `memory` metric might be implemented by the system keeping track of the actual memory allocations and releases, in which case it's an accumulator as opposed to an overall lookup.

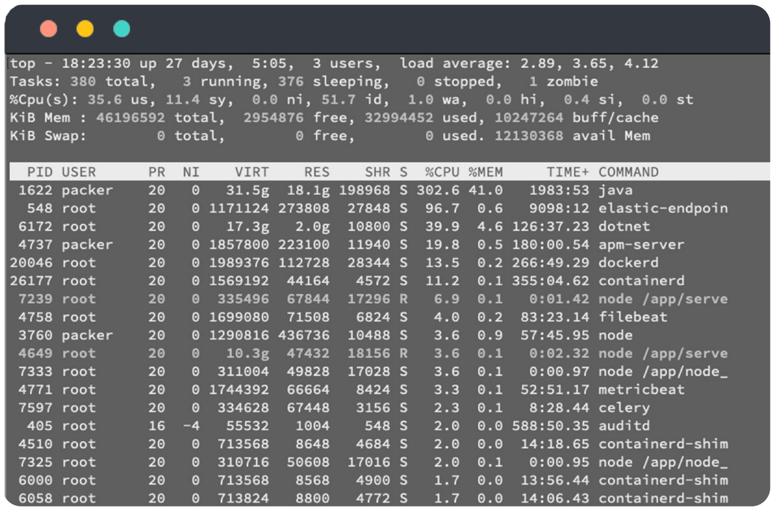
## Utilization or ratio metrics

Utilization metrics, such as the `percentage_of_cpu` , are a comparison between the amount of a resource that is used versus what is available. In the example on the previous page, `WindowServer` was using 53.7% of the available CPU.

A quick note, though — metrics can be misleading, and it's important to know where some of them are coming from. With CPU utilization, you might think that it's 53.7% out of 100%, but that's not the case; if you add up the percentages on that CPU percentage column, it's already over 160%. In fact, the way it's calculated, this laptop actually tops out at 1,600% CPU.

## Aggregation metrics

We've covered some of the fundamentals around metrics: different classifications, different ways to use them, and a couple of caveats. What about different ways to *gather* metrics? Of course, you can open up `top` and see a snapshot of your system, but then you'd be back in the same situation of structured versus unstructured logs, and have to parse things.



```
top - 18:23:30 up 27 days,  5:05,  3 users,  load average: 2.89, 3.65, 4.12
Tasks: 380 total,   3 running, 376 sleeping,   0 stopped,   1 zombie
%Cpu(s): 35.6 us, 11.4 sy,  0.0 ni, 51.7 id,  1.0 wa,  0.0 hi,  0.4 si,  0.0 st
KiB Mem : 46196592 total,  2954876 free, 32994452 used, 10247264 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 12130368 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1622 packer    20   0   31.5g  18.1g 198968 S 302.6 41.0   1983:53 java
  548 root      20   0 1171124 273808  27848 S  96.7  0.6   9098:12 elastic-endpoin
 6172 root      20   0   17.3g   2.0g  10800 S  39.9  4.6 126:37.23 dotnet
 4737 packer    20   0 1857800 223100  11940 S  19.8  0.5 180:00.54 apm-server
20046 root      20   0 1989376 112728  28344 S  13.5  0.2 266:49.29 dockerd
26177 root      20   0 1569192  44164   4572 S  11.2  0.1 355:04.62 containerd
 7239 root      20   0  335496  67844  17296 R   6.9  0.1   0:01.42 node /app/serve
 4758 root      20   0 1699080  71508   6824 S   4.0  0.2  83:23.14 filebeat
 3760 packer    20   0 1290816 436736  10488 S   3.6  0.9  57:45.95 node
 4649 root      20   0   10.3g  47432  18156 R   3.6  0.1   0:02.32 node /app/serve
 7333 root      20   0  311004  49828  17028 S   3.6  0.1   0:00.97 node /app/node_
 4771 root      20   0 1744392  66664   8424 S   3.3  0.1  52:51.17 metricbeat
 7597 root      20   0  334628  67448   3156 S   2.3  0.1   8:28.44 celery
  405 root      16  -4   55532   1004    548 S   2.0  0.0 588:50.35 auditd
 4510 root      20   0  713568   8648   4684 S   2.0  0.0  14:18.65 containerd-shim
 7325 root      20   0  310716  50608  17016 S   2.0  0.1   0:00.95 node /app/node_
 6000 root      20   0  713568   8568   4900 S   1.7  0.0  13:56.44 containerd-shim
 6058 root      20   0  713824   8800   4772 S   1.7  0.0  14:06.43 containerd-shim
```

Image: top running in a terminal

Luckily, many common services provide APIs or other interfaces that allow you to poll to retrieve metrics; simply check the documentation for the service in question. In addition, there's also a hybrid type of metrics: aggregation metrics.

Aggregation metrics are good when you don't want to know the exact metric value at a single point in time (for example, CPU utilization), which might require you to grab the value(s) more often than you'd like. Instead, services provide aggregated metrics. You get the average CPU usage for the past 10 minutes versus the CPU usage every 10 seconds. Aggregated metrics have the benefit of taking up less storage space and overhead than discrete methods. In this case, rather than one metric every 10 seconds, we have one every 600 seconds. The drawback: the longer the time between measurements, the more likely you'll miss a significant event. A short CPU spike in a 10-minute window might not impact the average a lot. When leveraging aggregated metrics, it's common to also include `min()` and `max()` aggregations for the metric, in addition to the `avg()` .

Like logs, metrics only tell part of the story. They are simply data points without context. When you start looking at logs and traces together, you can start to see a combined view. And now you're able to see the bigger picture (like what actually happened when the CPU spiked).

## Traces: end-to-end visibility into your application

Logs and metrics show interesting events that have happened in your systems and key performance indicators of resource utilization. However, they don’t show where your applications are spending their time. This is where application performance monitoring (APM) comes in. APM traces show what your applications and services are doing.

Generally, traces are depicted in what’s called a waterfall view as a distributed trace. A distributed trace shows the path transactions (or requests) take through your system. These transactions include calls to microservices and other applications, as well as requests to data stores and other external services. The waterfall shows nested calls, broken down into spans for each service. It may also include additional function calls within a service, as shown below.

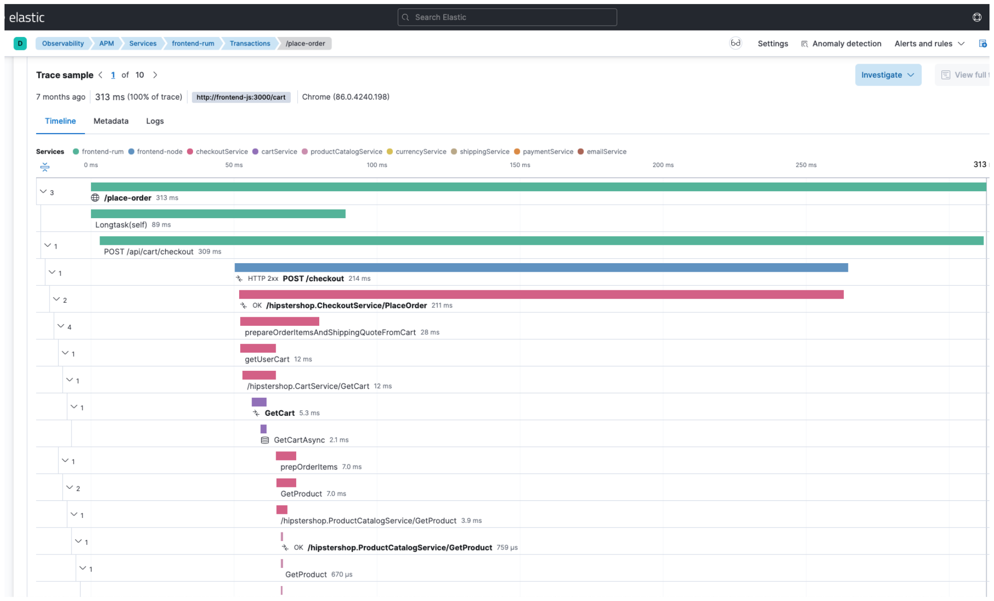

elastic
Search Elastic
Observability
APM
Services
frontend-rum
Transactions
/place-order
Settings
Anomaly detection
Alerts and rules
Trace sample 1 of 10
Investigate
7 months ago
313 ms (100% of trace)
http://frontend-js:3000/cart
Chrome (86.0.4240.198)
Timeline
Metadata
Logs
Services ● frontend-rum ● frontend-node ● checkoutService ● cartService ● productCatalogService ● currencyService ● shippingService ● paymentService ● emailService
0 ms
50 ms
100 ms
150 ms
200 ms
250 ms
313
/place-order 313 ms
Longtask(self) 89 ms
POST /api/cart/checkout 309 ms
HTTP 2xx POST /checkout 214 ms
OK /hipstershop.CheckoutService/PlaceOrder 211 ms
prepareOrderItemsAndShippingQuoteFromCart 28 ms
getUserCart 12 ms
/hipstershop.CartService/GetCart 12 ms
GetCart 5.3 ms
GetCartAsync 2.1 ms
prepOrderItems 7.0 ms
GetProduct 7.0 ms
/hipstershop.ProductCatalogService/GetProduct 3.9 ms
OK /hipstershop.ProductCatalogService/GetProduct 759 μs
GetProduct 670 μs

**Image: distributed trace showing service call interactions**

While logs and metrics need to be explicitly implemented, traces usually require you to add configuration code to your applications and services to enable them. This enablement process is called *instrumentation*. Once your services are instrumented, you can see where your applications are spending their time. The more of your applications and services that you instrument, the more you can leverage distributed tracing and follow transactions as they propagate across your application stack.

Application traces can also detect and visualize the dependencies between applications and internal or external services and can be used to gauge the overall health of your applications.

## Tech Tip

**APM needs to be cost-effective**

Application performance monitoring (APM) and the distributed traces it generates are among the most important signals in modern observability. From the moment a user starts interacting with the application until they have achieved their desired results, APM tracks their experience through its distributed traces.

However, the cost (time and complexity to instrument applications) often limits where and how often APM is deployed. The advent of auto-instrumentation from CNCF projects like OpenTelemetry makes instrumenting your applications easier and lowers the overhead. Finding a more cost-effective APM tool may allow your teams to monitor and manage a broader set of your applications, including dev environments (which are sometimes ignored due to cost).

## Transactions

Transactions are events that correspond to a logical unit of work. They are often associated with an incoming request or similar task for a monitored service. Transactions can include multiple spans as well as additional attributes, like data about the environment in which the event is recorded as shown below.

In the context of APM, transactions usually refer to web transactions and are inclusive of all activity from the time a request is submitted to when a response is received.

Transactions have additional attributes associated with them, like data about the environment in which the event is recorded:

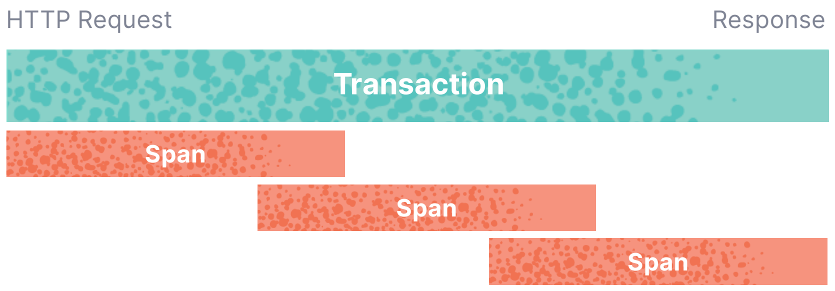
**Service:** environment, framework, language, etc.

**Host:** architecture, hostname, IP, etc.

**Process:** args, PID, PPID, etc.

**URL:** full, domain, port, query, etc.

**User:** (if supplied) email, ID, username, etc.



A few examples of transactions include:

- **A request to your server**
- **A batch job**
- **A background job**

## Spans

Usually depicted in horizontal bars in the waterfall view of an APM analysis tool, spans are the individual units or pieces of the workflow. These segments are the core of distributed tracing. Spans measure from the start to the end of an activity and contain information about the execution of a specific code path.

Common attributes of a span include:

- **Start time**
- **Finish time**
- **A name**
- **A type**

Application trace data also serves as a mechanism to centralize errors and exceptions, which lowers mean time to detection (MTTD). It also allows you to quickly triage and resolve issues, lowering mean time to resolution (MTTR).

When instrumenting your code for APM, you'll usually have the option to enrich the instrumentation as little or as much as you'd like. For example, you can add custom transaction definitions or enrich your traces with custom metadata.

Tracing can also help you gauge the end-user experience holistically and within your application environment. The request isn't done just because a service sent a response; it still has to be rendered in the browser. Where traces may fall short is around visibility to third-party services and the user's interactions on the internet, including their browser. If these additional steps are done inefficiently, you may end up with unhappy users (or worse, *former* users).

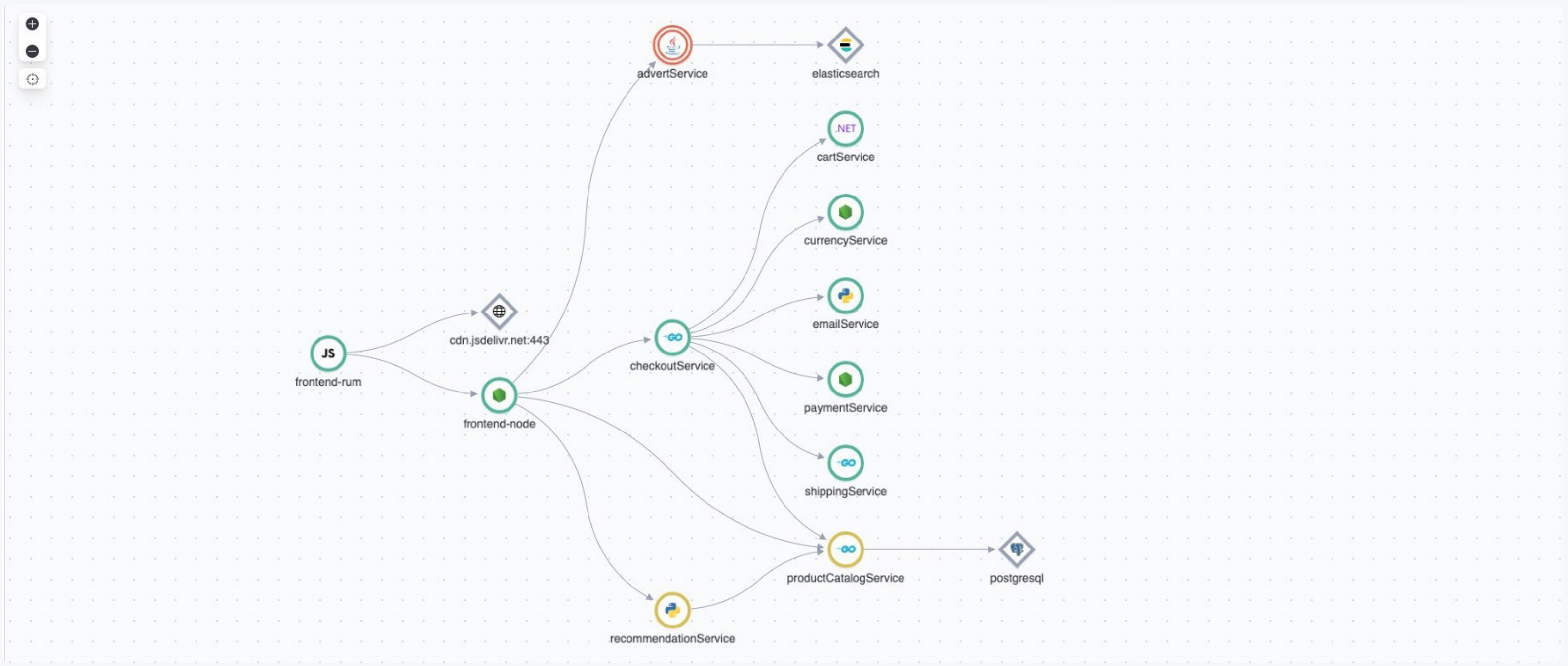


**Image: connections and health indicators**

## Continuous profiling: the fourth pillar of observability

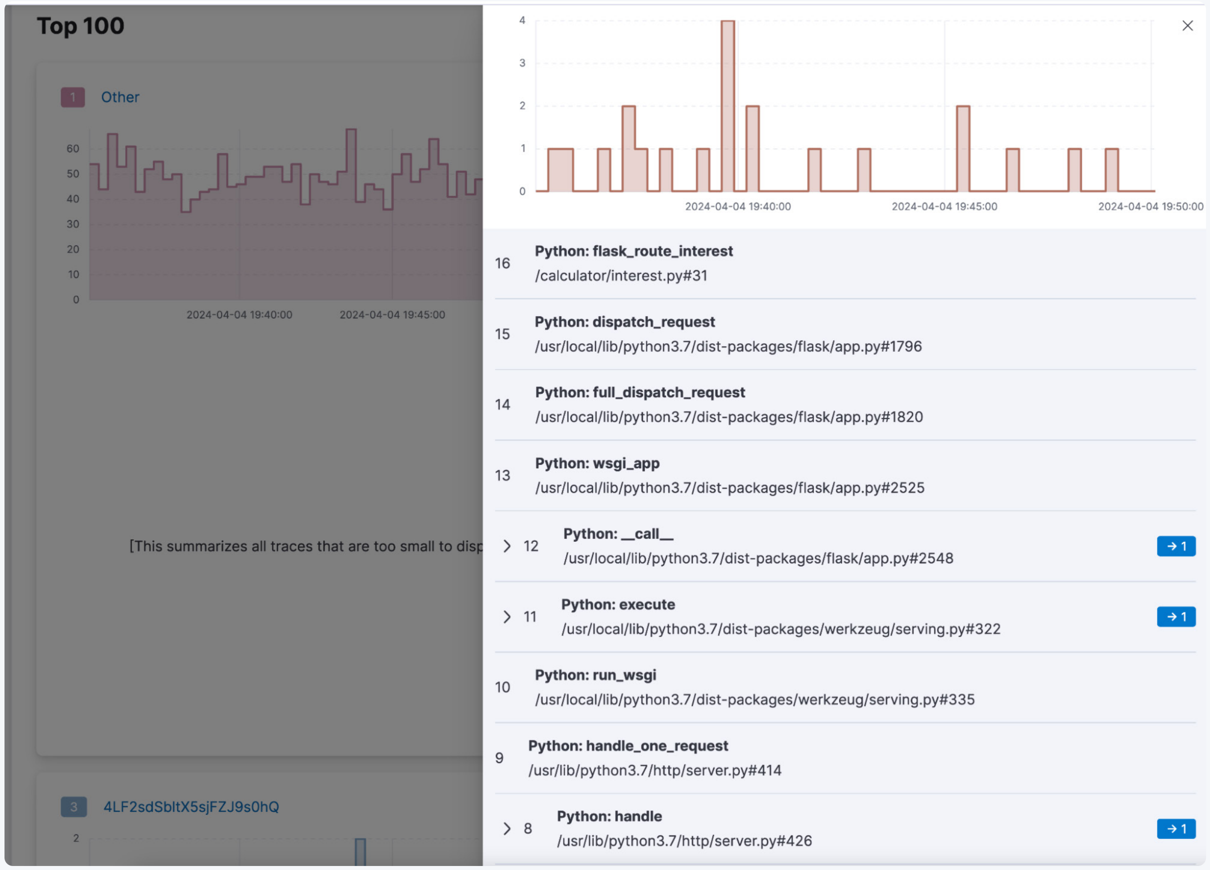
Alongside logs, metrics, and traces, profiling adds a fourth and newer signal to observability. Profiling offers a method of analyzing a program or code to identify which parts are using the most resources — down to a single line of code.

Continuous profiling is a type of round-the-clock profiling. Through continuous profiling, you can profile every line of code in your machine and in different languages. And if the profiling tool can work with minimal resource overhead, it can be used both in development and in production. Continuous profiling provides visibility into the bigger picture of your code's resources over time and helps eliminate guesswork when debugging and fixing issues.

Profiling tools typically provide developers with a few key summaries, low-level visualizations, and data types as seen next.

## Stacktraces

Continuous profiling depends on stacktraces, which are a snapshot of the call stack of an application at a specific moment in time. A stacktrace serves as a historical record of a call stack and allows you to trace the sequence of function calls that the program has made up to that point. They're a simple roadmap that explains what happened on the road from point A to point B.

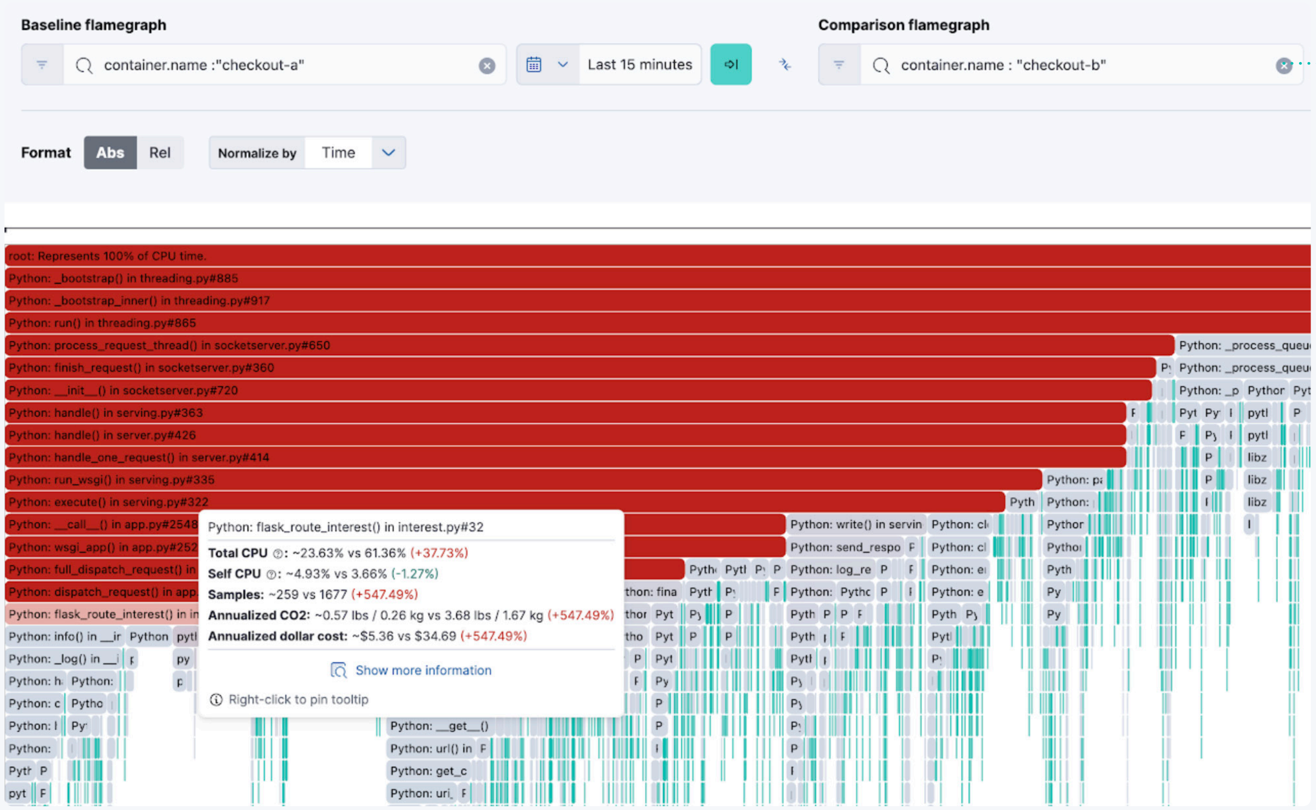


```
16  Python: flask_route_interest
    /calculator/interest.py#31
15  Python: dispatch_request
    /usr/local/lib/python3.7/dist-packages/flask/app.py#1796
14  Python: full_dispatch_request
    /usr/local/lib/python3.7/dist-packages/flask/app.py#1820
13  Python: wsgi_app
    /usr/local/lib/python3.7/dist-packages/flask/app.py#2525
> 12  Python: __call__                                               → 1
      /usr/local/lib/python3.7/dist-packages/flask/app.py#2548
> 11  Python: execute                                                → 1
      /usr/local/lib/python3.7/dist-packages/werkzeug/serving.py#322
10  Python: run_wsgi
    /usr/local/lib/python3.7/dist-packages/werkzeug/serving.py#335
9   Python: handle_one_request
    /usr/lib/python3.7/http/server.py#414
> 8   Python: handle                                                 → 1
      /usr/lib/python3.7/http/server.py#426
```

Image: an example of a stacktrace

## Flamegraphs

For a visual representation of a stacktrace, developers turn to flamegraphs. In a flamegraph, each function of a stacktrace is represented with a rectangle, and each rectangle's width represents time. The more time spent on a function, the wider the rectangle. In a flamegraph, rectangles are stacked vertically; the number of stacked rectangles represents each stack's depth (or the number of functions called to reach its current function).



Baseline flamegraph: container.name :"checkout-a"

Last 15 minutes

Comparison flamegraph: container.name : "checkout-b"

Format: Abs | Rel

Normalize by: Time

Python: flask_route_interest() in interest.py#32

**Total CPU**: ~23.63% vs 61.36% (+37.73%)

**Self CPU**: ~4.93% vs 3.66% (-1.27%)

**Samples:** ~259 vs 1677 (+547.49%)

**Annualized CO2:** ~0.57 lbs / 0.26 kg vs 3.68 lbs / 1.67 kg (+547.49%)

**Annualized dollar cost:** ~$5.36 vs $34.69 (+547.49%)

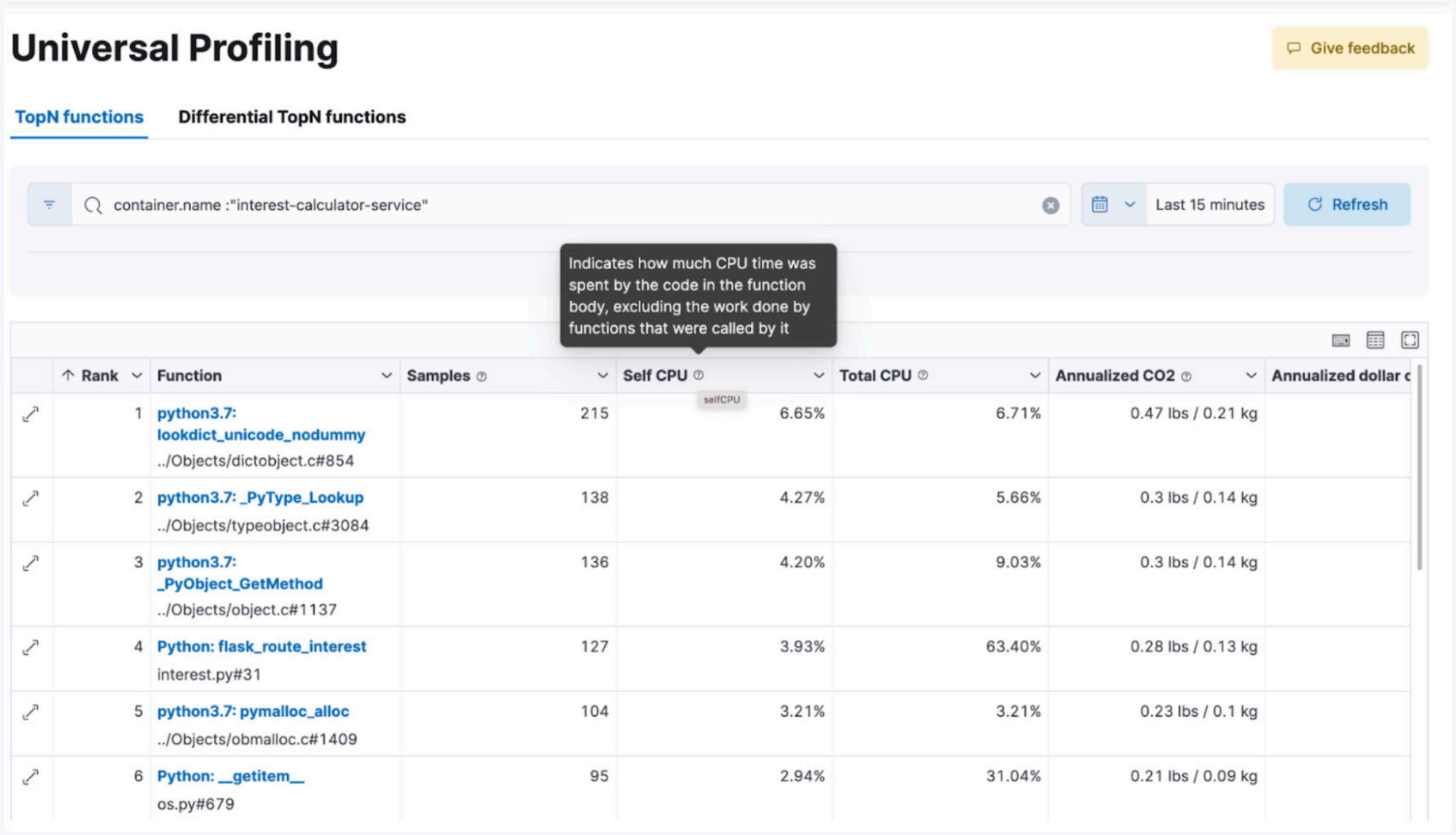
Show more information

Right-click to pin tooltip

root: Represents 100% of CPU time.
Python: _bootstrap() in threading.py#885
Python: _bootstrap_inner() in threading.py#917
Python: run() in threading.py#865
Python: process_request_thread() in socketserver.py#650
Python: finish_request() in socketserver.py#360
Python: __init__() in socketserver.py#720
Python: handle() in serving.py#363
Python: handle() in server.py#426
Python: handle_one_request() in server.py#414
Python: run_wsgi() in serving.py#335
Python: execute() in serving.py#322
Python: __call__() in app.py#2548
Python: wsgi_app() in app.py#252
Python: full_dispatch_request() in
Python: dispatch_request() in app
Python: flask_route_interest() in in
Python: _process_queu
Python: write() in servin
Python: send_respo
Python: log_re
Python: __get__()
Python: url() in
Python: get_c

Flamegraphs allow developers to quickly identify the functions that are consuming the most resources through an easy-to-understand graph.

Given that functions are the building blocks for an application or service, this visibility into its resource usage can be crucial for performance and optimizing applications. More sophisticated profiling tools may present information such as the most frequently sampled functions, broken down by CPUs, annualized cost estimates, and annualized CO2 to understand energy efficiency.



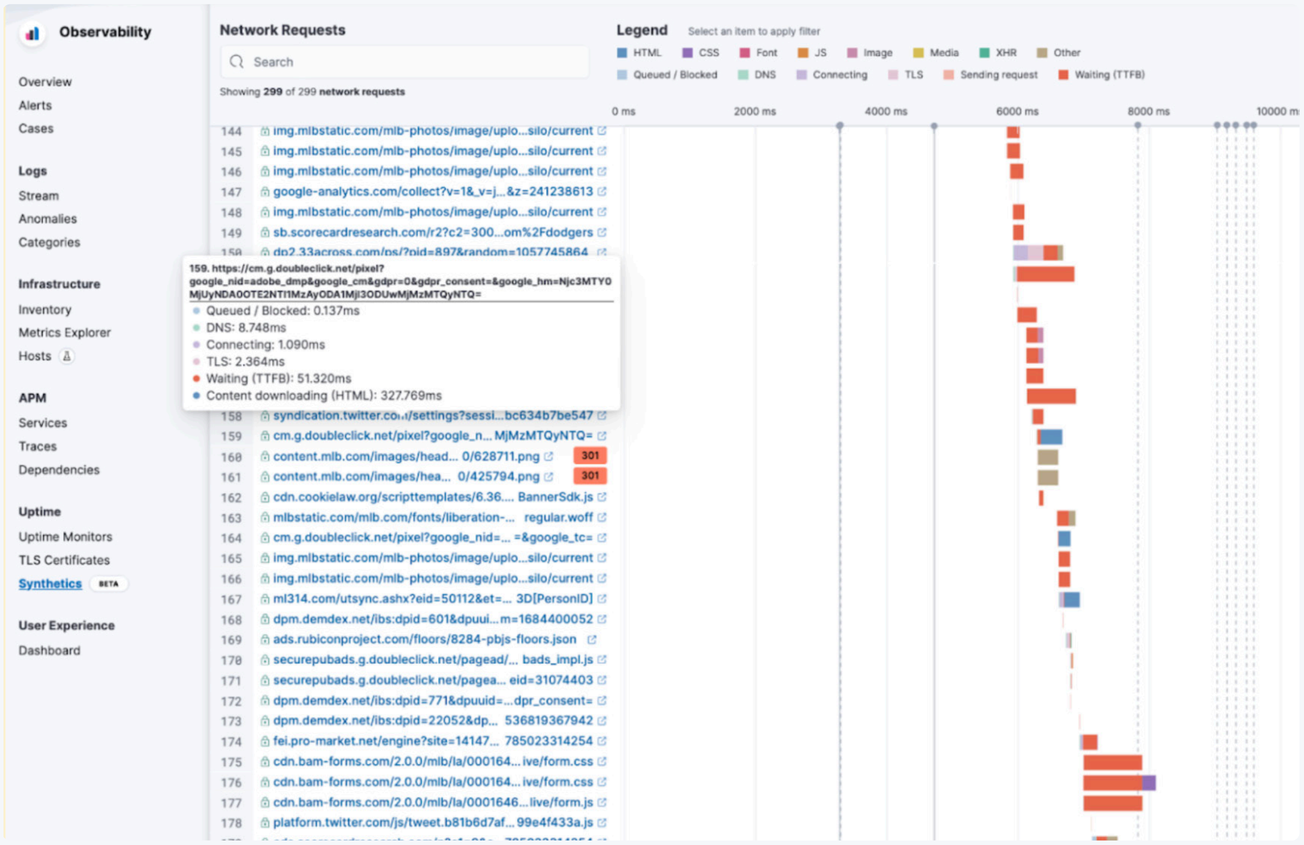
Indicates how much CPU time was spent by the code in the function body, excluding the work done by functions that were called by it

| Rank | Function | Samples | Self CPU | Total CPU | Annualized CO2 | Annualized dollar c |
|---|---|---|---|---|---|---|
| 1 | python3.7: lookdict_unicode_nodummy ../Objects/dictobject.c#854 | 215 | 6.65% | 6.71% | 0.47 lbs / 0.21 kg | |
| 2 | python3.7: _PyType_Lookup ../Objects/typeobject.c#3084 | 138 | 4.27% | 5.66% | 0.3 lbs / 0.14 kg | |
| 3 | python3.7: _PyObject_GetMethod ../Objects/object.c#1137 | 136 | 4.20% | 9.03% | 0.3 lbs / 0.14 kg | |
| 4 | Python: flask_route_interest interest.py#31 | 127 | 3.93% | 63.40% | 0.28 lbs / 0.13 kg | |
| 5 | python3.7: pymalloc_alloc ../Objects/obmalloc.c#1409 | 104 | 3.21% | 3.21% | 0.23 lbs / 0.1 kg | |
| 6 | Python: __getitem__ os.py#679 | 95 | 2.94% | 31.04% | 0.21 lbs / 0.09 kg | |

Image: this topN functions view shows the most frequently sampled functions, broken down by CPU time, annualized CO2, and annualized cost estimates

## Synthetic monitoring: understanding customer journeys

Logs, metrics, traces, and profiling are generally “after-the-fact” tools, which are helpful when trying to resolve issues, solve problems, or identify areas that can be improved in an application ecosystem. The downside is that when you’re using them to investigate something, it usually means that a problem has impacted your customers.



Showing **299** of 299 **network requests**

```
144  img.mlbstatic.com/mlb-photos/image/uplo...silo/current
145  img.mlbstatic.com/mlb-photos/image/uplo...silo/current
146  img.mlbstatic.com/mlb-photos/image/uplo...silo/current
147  google-analytics.com/collect?v=1&_v=j...&z=241238613
148  img.mlbstatic.com/mlb-photos/image/uplo...silo/current
149  sb.scorecardresearch.com/r2?c2=300...om%2Fdodgers
150  dp2.33across.com/ps/?pid=897&random=1057745864
158  syndication.twitter.co.../settings?sessi...bc634b7be547
159  cm.g.doubleclick.net/pixel?google_n... MjMzMTQyNTQ=
160  content.mlb.com/images/head... 0/628711.png          301
161  content.mlb.com/images/hea...  0/425794.png          301
162  cdn.cookielaw.org/scripttemplates/6.36.... BannerSdk.js
163  mlbstatic.com/mlb.com/fonts/liberation-...  regular.woff
164  cm.g.doubleclick.net/pixel?google_nid=... =&google_tc=
165  img.mlbstatic.com/mlb-photos/image/uplo...silo/current
166  img.mlbstatic.com/mlb-photos/image/uplo...silo/current
167  ml314.com/utsync.ashx?eid=50112&et=... 3D[PersonID]
168  dpm.demdex.net/ibs:dpid=601&dpuui...m=1684400052
169  ads.rubiconproject.com/floors/8284-pbjs-floors.json
170  securepubads.g.doubleclick.net/pagead/... bads_impl.js
171  securepubads.g.doubleclick.net/pagea... eid=31074403
172  dpm.demdex.net/ibs:dpid=771&dpuuid=...dpr_consent=
173  dpm.demdex.net/ibs:dpid=22052&dp...  536819367942
174  fei.pro-market.net/engine?site=14147...  785023314254
175  cdn.bam-forms.com/2.0.0/mlb/la/000164... ive/form.css
176  cdn.bam-forms.com/2.0.0/mlb/la/000164... ive/form.css
177  cdn.bam-forms.com/2.0.0/mlb/la/0001646...live/form.js
178  platform.twitter.com/js/tweet.b81b6d7af...99e4f433a.js
```

```
159. https://cm.g.doubleclick.net/pixel?
google_nid=adobe_dmp&google_cm&gdpr=0&gdpr_consent=&google_hm=Njc3MTY0
MjUyNDA0OTE2NTI1MzAyODA1MjI3ODUwMjMzMTQyNTQ=
● Queued / Blocked: 0.137ms
● DNS: 8.748ms
● Connecting: 1.090ms
● TLS: 2.364ms
● Waiting (TTFB): 51.320ms
● Content downloading (HTML): 327.769ms
```

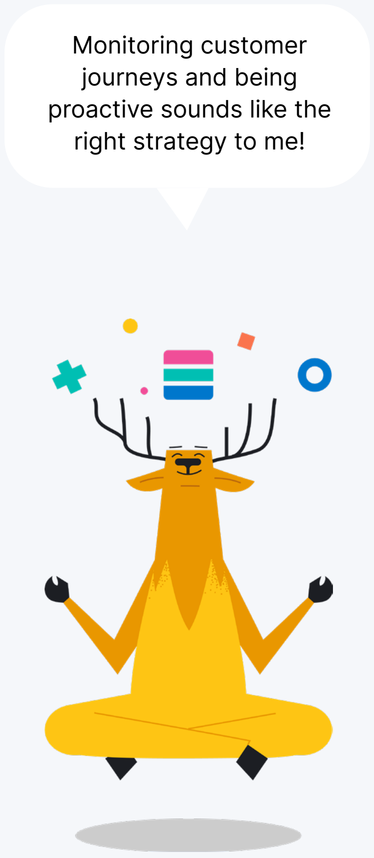

Monitoring customer journeys and being proactive sounds like the right strategy to me!

Image: network requests chart for synthetic monitoring

Additional synthetic monitoring metrics and data vary by vendor but could include:

- **First contentful paint (FCP), which measures initial rendering and page loading**
- **Largest contentful paint (LCP), which measures loading performance**
- **Cumulative layout shift (CLS), which measures visual stability, transfer size (size of fetched resource), object weight, and network requests**

The great thing about synthetic monitoring is that it can be completely automated and run on your schedule: you don't need human users to interact with your applications. It's a proactive monitoring system that allows you to test applications, detect performance issues, and address application weaknesses before a product launches. Plus, synthetic monitoring can help you execute tests on demand or at regular intervals, providing a continuous monitoring solution that can help you ensure your systems are delivering great customer experiences, all day, every day.

Ultimately, infrastructure monitoring and application observability alone are not enough to reliably predict end-user experience: these insights are only available by simulating the actual user experience, which is what synthetic monitoring provides.

# OpenTelemetry: the future of observability

## OpenTelemetry and its impact on observability data

One of the main challenges with observability data and telemetry is its diverse nature. Every application, environment, and organization is pretty unique in terms of the technologies they use, manage, and interact with. How do you bring together all this operational data from a variety of sources so you can easily troubleshoot and analyze the performance of your applications?

For many organizations, the answer lies in OpenTelemetry, an open source project and collection of APIs, SDKs, and tools. OpenTelemetry helps you instrument, generate, and collect telemetry data (metrics, logs, and traces), allowing your team to process and securely transmit all your telemetry data in a consistent format.

And since its inception in 2019, OpenTelemetry has become the industry standard in cloud-native infrastructures to collect and transfer high volumes of observability and monitoring data. It can be used with a variety of vendors and tools, including open source tools and commercial offerings.

## Open standards with Semantic Conventions (SemConv)

What's most important from a **data** standpoint is that OpenTelemetry defines Semantic Conventions: a common naming scheme for observability signals that makes it easier to consume and correlate all your data.

Just as OpenTelemetry provides a standard way to instrument applications, SemConv provides a standardized naming convention (schema) for your codebase, libraries, and platforms to help generate a consistent telemetry format. OpenTelemetry's SemConv currently cover: events data, logs data, metrics, traces, and resources.

## Tech Tip

### Origins of OpenTelemetry

A vendor-neutral, open source framework for collecting and exporting telemetry data, OpenTelemetry was formed by merging two other open standards, OpenTracing and OpenCensus. OpenTelemetry provides libraries and APIs for instrumenting code and collecting data, as well as tools and integrations for analyzing, visualizing, and storing the data.

SemConv help organizations solve a common problem: too often, teams spend unnecessary time transferring siloed data or transforming data that's structured in different schemas. Valuable time that could be spent identifying problems and finding solutions.

Another big benefit of SemConv is the decoupling of vendor-specific semantics. With OpenTelemetry's SemConv, data users can avoid vendor lock-in of their data. They can easily move between observability solutions without the need to adapt their data collection as long as the solutions are OpenTelemetry compliant. SemConv's common schema for telemetry data aims to help make widespread OTel adoption frictionless across vendors.

## 2024 Observability Landscape

**OpenTelemetry expected to be the future of observability data**

75%

**of organizations** are evaluating OTel, experimenting with it, or have it in production

87%

**of respondents** expect OTel to be the standard for observability data in three to five years

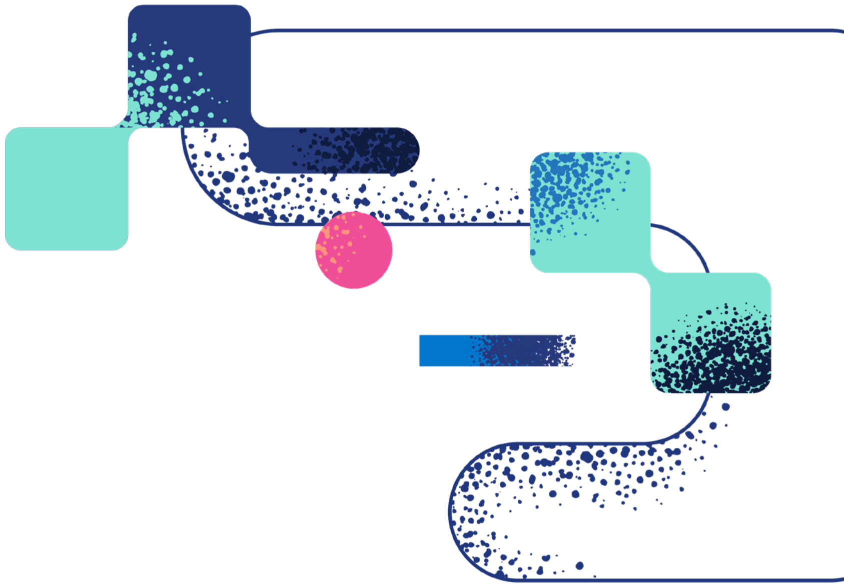
A recent survey conducted by Dimensional Research across 500 observability decision makers showed that 87% of respondents expect OpenTelemetry to be the standard for observability data.

**Source: "2024 Observability Landscape: A survey of Observability decision makers," survey conducted by Dimensional Research and sponsored by Elastic**

## Future of OpenTelemetry and SemConv with Elastic Common Schema (ECS)

Having a set of agreed-upon vocabulary is key to the future of modern observability. Most observability practitioners and vendors are subscribing to the notion that OpenTelemetry will be the de facto standard for telemetry data in the near future and thus SemConv in OpenTelemetry needs to be this shared vocabulary.

The Elastic Common Schema (ECS) has been a largely dominant standard for the logs and security domains. That is why Elastic contributing ECS to the OpenTelemetry project has been pivotal to converge toward a common schema for observability and security data handled by OpenTelemetry and to accelerate the project's adoption within the industry.



## Building a unified observability platform with your data

As you can tell from our review of modern observability and telemetry data, there are many different types, each offering a piece of information and visibility into your environment and your application's performance.

But to get true value and insights from your telemetry data, you will need a unified observability platform that brings it all together, enabling you to do more effective root cause analysis to resolve real-time issues or to archive your data for future analysis, exploration, and optimization.

By bringing all of your telemetry and observability data together on a unified platform, you will be setting the foundation to truly understand application performance and pave the way for more sophisticated capabilities like APM, AIOps, generative AI, and business observability. It's the first step to modern observability, which accelerates problem resolution and delivers unified context and correlation across all your data, at scale. Want to learn more about APM and distributed traces? Read our new ebook: An introduction to APM: the what, why and how.

# Elastic Observability: an open and extensible observability solution built on Search AI

Start here

If you're considering a modern and unified observability platform, Elastic Observability is the ideal solution to get started collecting and exploring all your telemetry data. It's a platform made to work with data built from open standards and captures all your logs, metrics, and data at scale. Elastic Observability offers traces, continuous profiling, synthetic monitoring, and APM under one unified solution.

Start a free trial of **Elastic Observability** to gather and visualize your telemetry data and improve your users' experience. Visit elastic.co/observability to learn more.