elastic

Search. Observe. Protect.

Data building blocks for an observability solution

What is telemetry?

elastic.co →

Table of contents

Introduction

According to the Cambridge dictionary, telemetry is "the science or process of collecting information about objects that are far away and sending the information somewhere electronically." But the usage of the term 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.

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, even on different continents. In this ebook we'll go over the basics of telemetry, so you can better understand and identify the different types of telemetry data and how it can be leveraged for observability initiatives.

As software systems get more complex, it's imperative to collect and observe telemetry data, enabling us to understand the components in the system, the interactions between them, and to triage and act upon any problems that may arise. Collecting and storing as much telemetry data as possible is needed for an observability solution. And oftentimes, missing data will make it harder to troubleshoot future application issues.

Types of telemetry data

When monitoring applications and infrastructure, the types of telemetry data can be broken down into three main types: logs, metrics, and traces. All three signals provide valuable insights for developers, architects, DevOps, and site reliability engineers (SREs). Combined, they provide a holistic view of your deployments and help you to identify and resolve issues. We'll walk through each of these telemetry data types and break them down a bit more.

Logs

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, that 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.

> **TIP**
>
> Don't turn on `DEBUG` level logging unless you really need to!

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:

```
::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. There's what looks like a date and maybe 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 is conveying the requesting IP address, a couple 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, where it relies 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.

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

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 above, 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 itself, will be added at the time of import) and the payload or text itself.

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
```

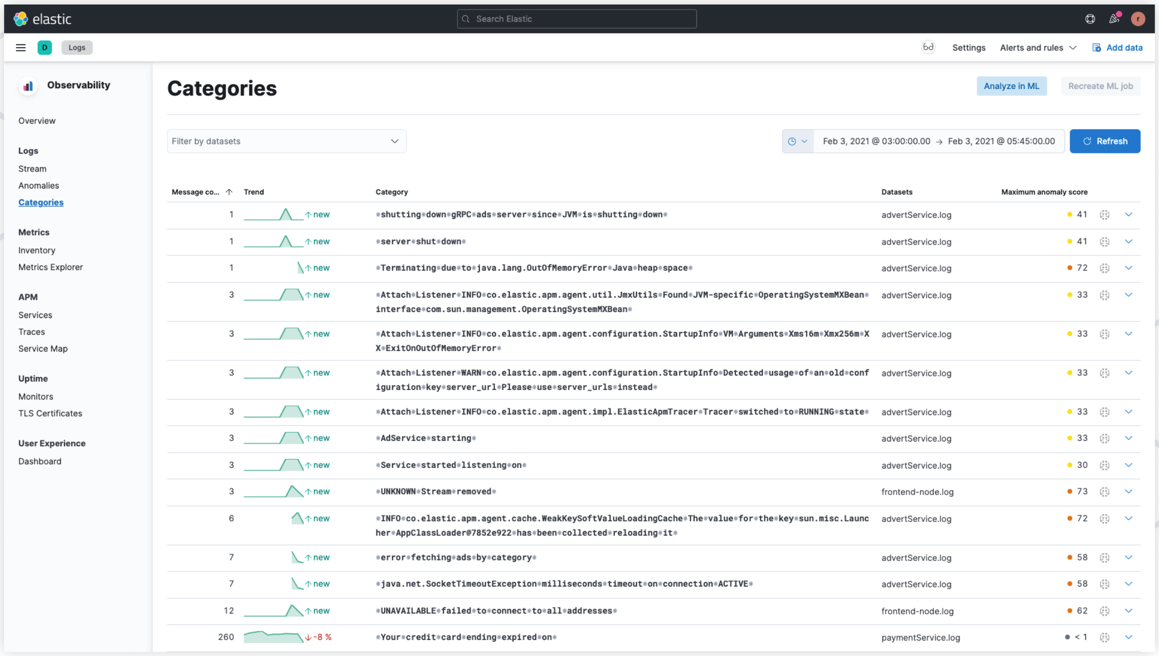
Structured logs

While semi-structured logs make it easier for machines to import or ingest logs, structured logs take the guesswork out of parsing: they start out parsed. By leveraging JSON formatting as a logging format, the fields and their values can be made explicit. For example, if the above slow log entry had been structured at the beginning, it could look like this:

```
[
  {
    "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"
  }
]
```

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 that 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"



Message co... ↑	Trend	Category	Datasets	Maximum anomaly score
1	↑ new	*shutting*down*gRPC*ads*server*since*JVM*is*shutting*down*	advertService.log	● 41
1	↑ new	*server*shut*down*	advertService.log	● 41
1	↑ new	*Terminating*due*to*java.lang.OutOfMemoryError*Java*heap*space*	advertService.log	● 72
3	↑ new	*Attach*Listener*INFO*co.elastic.apm.agent.util.JmxUtils*Found*JVM-specific*OperatingSystemMXBean*interface*com.sun.management.OperatingSystemMXBean*	advertService.log	● 33
3	↑ new	*Attach*Listener*INFO*co.elastic.apm.agent.configuration.StartupInfo*VM*Arguments*Xms16m*Xmx256m*XX*ExitOnOutOfMemoryError*	advertService.log	● 33
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	● 33
3	↑ new	*Attach*Listener*INFO*co.elastic.apm.agent.impl.ElasticApmTracer*Tracer*switched*to*RUNNING*state*	advertService.log	● 33
3	↑ new	*AdService*starting*	advertService.log	● 33
3	↑ new	*Service*started*listening*on*	advertService.log	● 30
3	↑ new	*UNKNOWN*Stream*removed*	frontend-node.log	● 73
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	● 72
7	↑ new	*error*fetching*ads*by*category*	advertService.log	● 58
7	↑ new	*java.net.SocketTimeoutException*milliseconds*timeout*on*connection*ACTIVE*	advertService.log	● 58
12	↑ new	*UNAVAILABLE*failed*to*connect*to*all*addresses*	frontend-node.log	● 62
260	↓ -8 %	*Your*credit*card*ending*expired*on*	paymentService.log	● < 1

It's hard to find the needle in the haystack

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, but quite often issues and errors 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.

Metrics

We've learned that logs are generated when something happens. Metrics, on the other hand, 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 above. 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, it's possible that applications are trying to chew up more RAM and experiencing memory allocation failures. Metrics may inadvertently miss cyclical fluctuations, unfortunately.

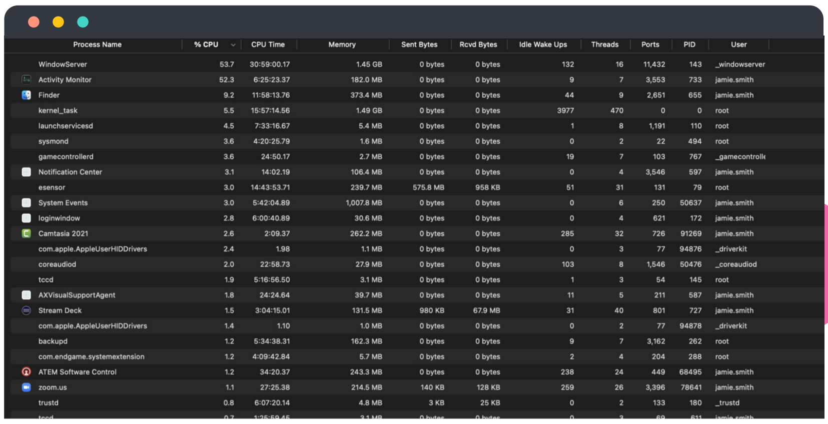
Metrics also tend to lose value as they age — it might be important to know down-to-the-minute resource consumption for the last few days, but that level of granularity is probably not needed for events from six months ago.

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

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

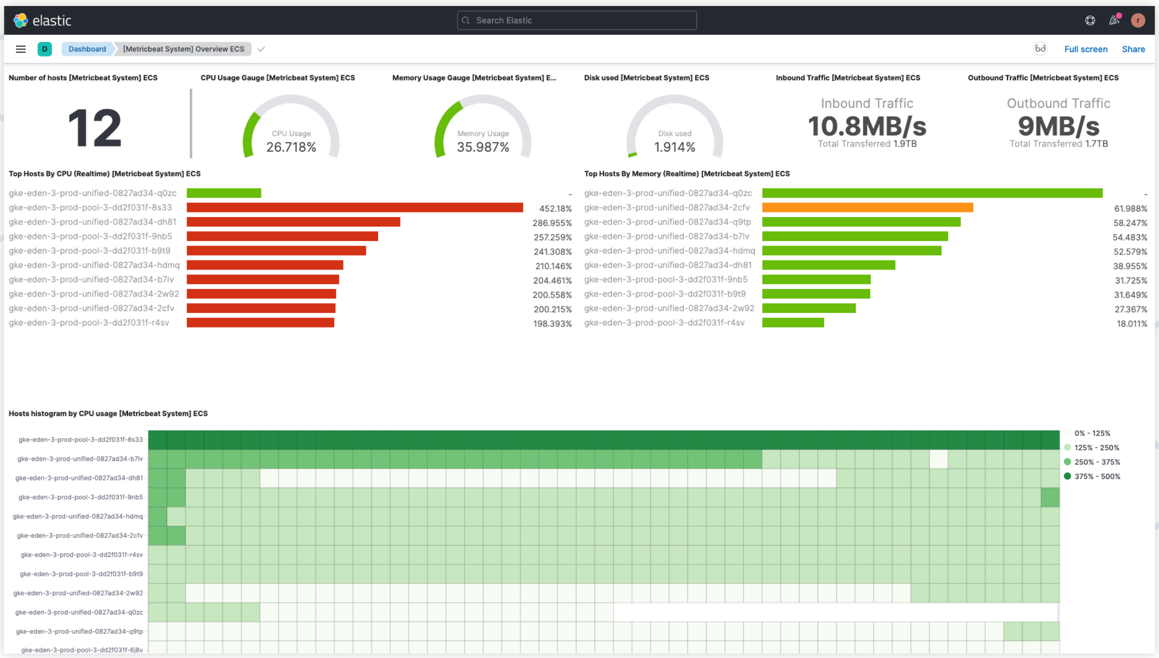
This list is not exhaustive but should serve as a good starting point. Let's take a closer look at the different types of metrics and the general use case for each, along with some examples. There are different ways metrics get "published" — perhaps via an API or even via standard interfaces and protocols (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	Rcvd Bytes	Idle Wake Ups	Threads	Ports	PID	User
WindowServer	53.7	30:59:00.17	1.45 GB	0 bytes	0 bytes	132	16	11,432	143	_windowserver
Activity Monitor	52.3	6:25:23.37	182.0 MB	0 bytes	0 bytes	9	7	3,553	733	jamie.smith
Finder	9.2	11:58:13.76	373.4 MB	0 bytes	0 bytes	44	9	2,651	655	jamie.smith
kernel_task	5.5	15:57:14.56	1.49 GB	0 bytes	0 bytes	3977	470	0	0	root
launchservicesd	4.5	7:33:16.67	5.4 MB	0 bytes	0 bytes	1	8	1,191	110	root
sysmond	3.6	4:20:25.79	1.6 MB	0 bytes	0 bytes	0	2	22	494	root
gamecontrollerd	3.6	24:50.17	2.7 MB	0 bytes	0 bytes	19	7	103	767	_gamecontrolle
Notification Center	3.1	14:02.19	106.4 MB	0 bytes	0 bytes	0	4	3,546	597	jamie.smith
esensor	3.0	14:43:53.71	239.7 MB	575.8 MB	958 KB	51	31	131	79	root
System Events	3.0	5:42:04.89	1,007.8 MB	0 bytes	0 bytes	0	6	250	50637	jamie.smith
loginwindow	2.8	6:00:40.89	30.6 MB	0 bytes	0 bytes	0	4	621	172	jamie.smith
Camtasia 2021	2.6	2:09.37	262.2 MB	0 bytes	0 bytes	285	32	726	91269	jamie.smith
com.apple.AppleUserHIDDrivers	2.4	1.98	1.1 MB	0 bytes	0 bytes	0	3	77	94876	_driverkit
coreaudiod	2.0	22:58.73	27.9 MB	0 bytes	0 bytes	103	8	1,546	50476	_coreaudiod
tccd	1.9	5:16:56.50	3.1 MB	0 bytes	0 bytes	1	3	54	145	root
AXVisualSupportAgent	1.8	24:24.64	39.7 MB	0 bytes	0 bytes	11	5	211	587	jamie.smith
Stream Deck	1.5	3:04:15.01	131.5 MB	980 KB	67.9 MB	31	40	801	727	jamie.smith
com.apple.AppleUserHIDDrivers	1.4	1.10	1.0 MB	0 bytes	0 bytes	0	2	77	94878	_driverkit
backupd	1.2	5:34:38.31	162.3 MB	0 bytes	0 bytes	9	7	3,162	262	root
com.endgame.systemextension	1.2	4:09:42.84	5.7 MB	0 bytes	0 bytes	2	4	204	288	root
ATEM Software Control	1.2	34:20.37	243.3 MB	0 bytes	0 bytes	238	24	449	68495	jamie.smith
zoom.us	1.1	27:25.38	214.5 MB	140 KB	128 KB	259	26	3,396	78641	jamie.smith
trustd	0.8	6:07:20.14	4.8 MB	3 KB	25 KB	0	2	133	180	_trustd
tccd	0.7	1:25:59.45	3.1 MB	0 bytes	0 bytes	0	3	69	611	jamie.smith

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%

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 are counting things, and are incremented by one each time something happens. Some common examples are things like `page_faults`, which essentially tracks 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 the number of people in line.

Counters can be cumulative in that they keep track of the value since the beginning of a process — some examples would be 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 on a periodic basis: things like 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.

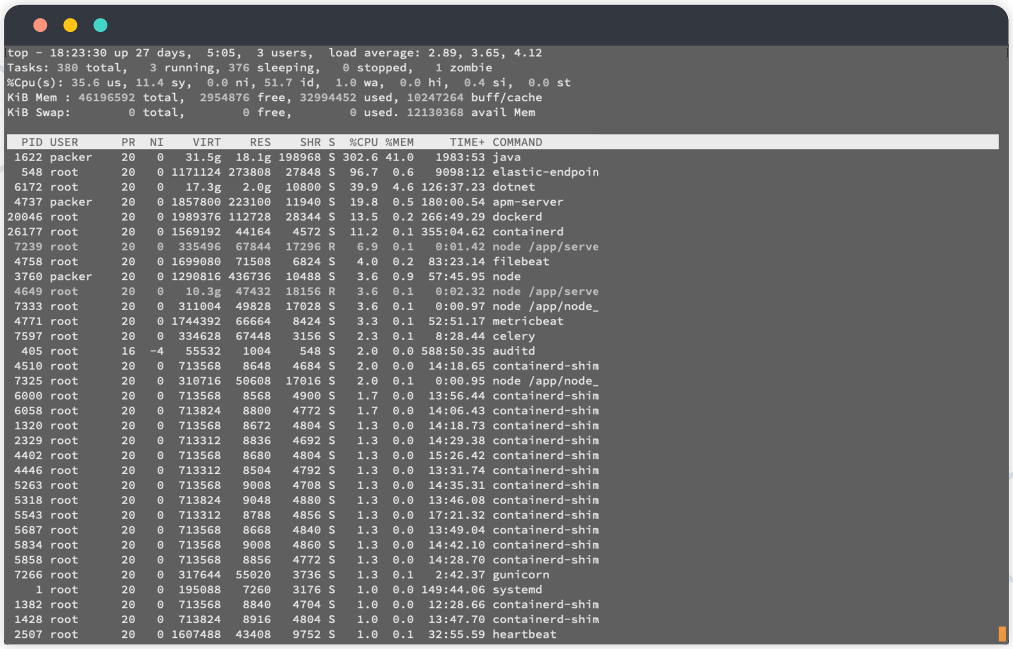
Utilization or ratio metrics

Utilization metrics, such as the `percentage_of_cpu` from above, are a comparison between the amount of a resource that is used versus what is available. In this case, `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. We haven't yet talked about different ways to *gather* metrics.

Of course, you can open up `top` and see a snapshot your system, but then you'd be back in the same situation of structured vs. 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
 1320 root      20   0  713568   8672   4804 S   1.3  0.0  14:18.73 containerd-shim
 2329 root      20   0  713312   8836   4692 S   1.3  0.0  14:29.38 containerd-shim
 4402 root      20   0  713568   8680   4804 S   1.3  0.0  15:26.42 containerd-shim
 4446 root      20   0  713312   8504   4792 S   1.3  0.0  13:31.74 containerd-shim
 5263 root      20   0  713568   9008   4708 S   1.3  0.0  14:35.31 containerd-shim
 5318 root      20   0  713824   9048   4880 S   1.3  0.0  13:46.08 containerd-shim
 5543 root      20   0  713312   8788   4856 S   1.3  0.0  17:21.32 containerd-shim
 5687 root      20   0  713568   8668   4840 S   1.3  0.0  13:49.04 containerd-shim
 5834 root      20   0  713568   9008   4860 S   1.3  0.0  14:42.10 containerd-shim
 5858 root      20   0  713568   8856   4772 S   1.3  0.0  14:28.70 containerd-shim
 7266 root      20   0  317644  55020   3736 S   1.3  0.1   2:42.37 gunicorn
    1 root      20   0  195088   7260   3176 S   1.0  0.0 149:44.06 systemd
 1382 root      20   0  713568   8840   4704 S   1.0  0.0  12:28.66 containerd-shim
 1428 root      20   0  713824   8916   4804 S   1.0  0.0  13:47.70 containerd-shim
 2507 root      20   0 1607488  43408   9752 S   1.0  0.1  32:55.59 heartbeat
```

`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 to the other types of metrics we discussed earlier, there's also a hybrid type of metrics: aggregation metrics.

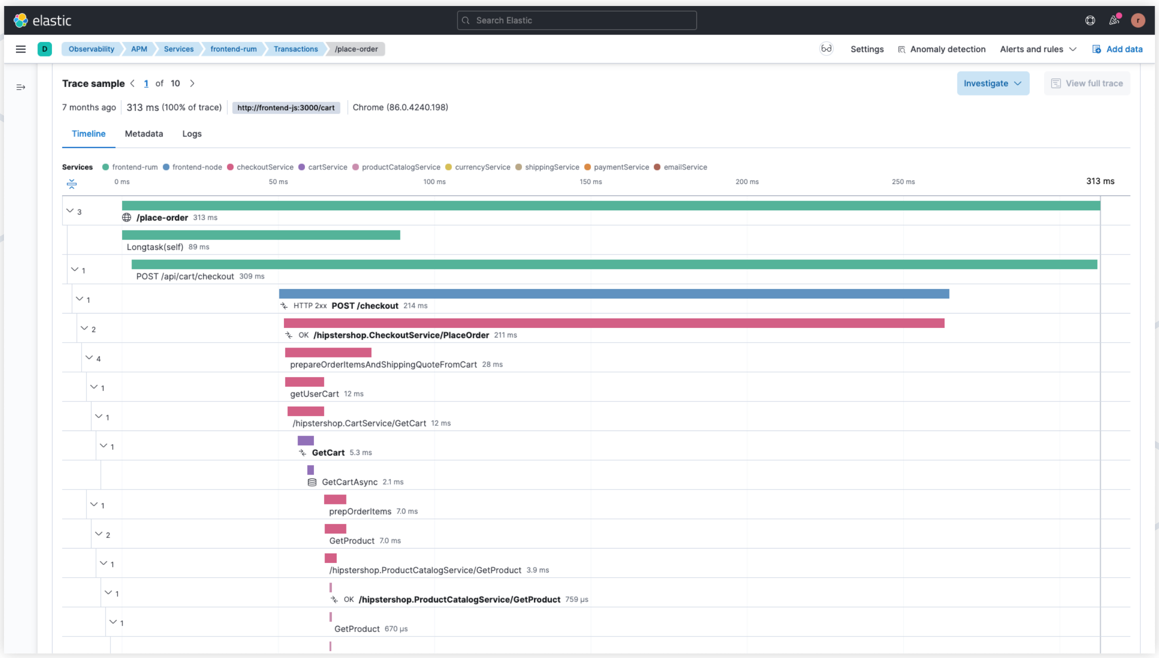
Aggregation metrics are good when you don't want to know the exact metric value at a single point in time (for example, CPU utilization); that might require you to grab the value(s) more often than you'd like. Instead, services provide *aggregated* metrics. Instead of getting the CPU usage every ten seconds, you'd rather get the *average* CPU usage for the last ten minutes. Aggregated metrics have the benefit of taking up less storage space and overhead than discrete methods. In this case, rather than one metric every ten seconds, we have one every 600 seconds. The drawback is, the longer the time is between measurements, the more likely it is that you'll miss a significant event. A short CPU spike in a ten-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. For example, what was actually happening when the CPU was spiking?

Traces

For logs and metrics to get generated by an application, they need to be explicitly implemented. Logs are emitted when a particular line of code is executed. Metrics need to be calculated, aggregated or summed up, incremented, and published based on the retrieval or dissemination mechanism of choice. Of course, many common services provide logs and metrics out of the box. Things like databases, message stores, and even operating systems, tend to include robust logs and relevant metrics. So all your organization will need to do is add logs and metrics from the applications and services that you write.

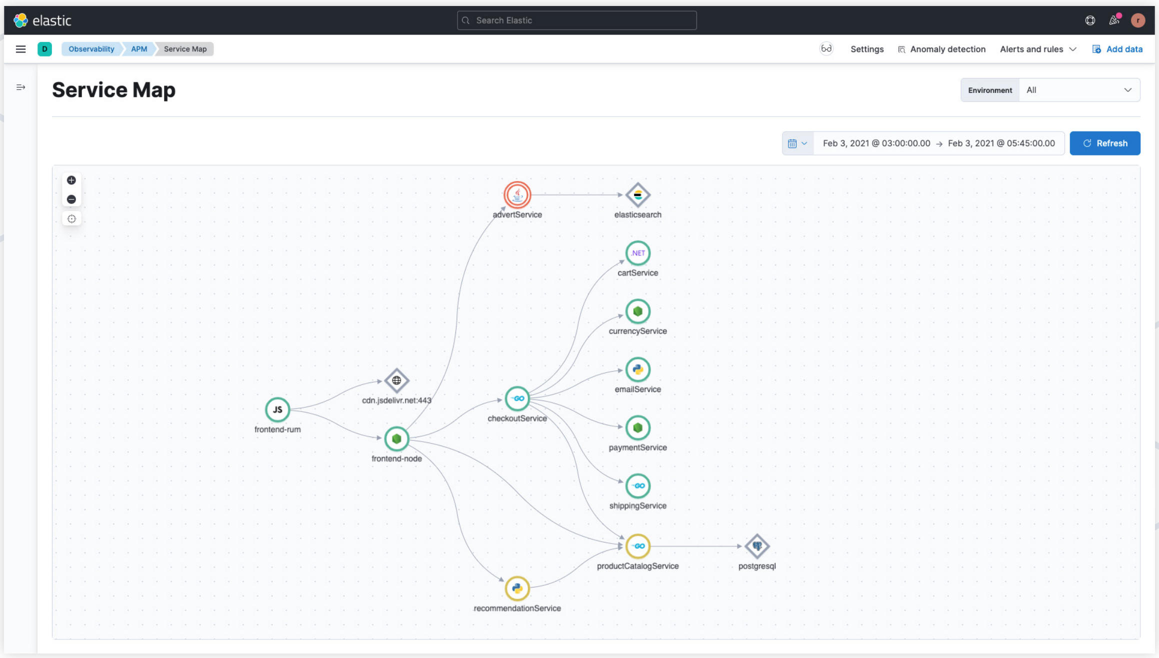
Logs and metrics show interesting events that have happened in your systems, and key performance indicators of resource utilization. However, they don't show where your applications are spending their time. This is where application performance monitoring (APM) comes in. *APM traces* show what your applications 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. Additionally, APM often includes information about the instrumented services such as garbage collection, memory, exceptions, and errors.


elastic
Search Elastic
Observability
APM
Services
frontend-rum
Transactions
/place-order
Settings
Anomaly detection
Alerts and rules
Add data
Trace sample 1 of 10
Investigate
View full trace
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 ms
/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

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.



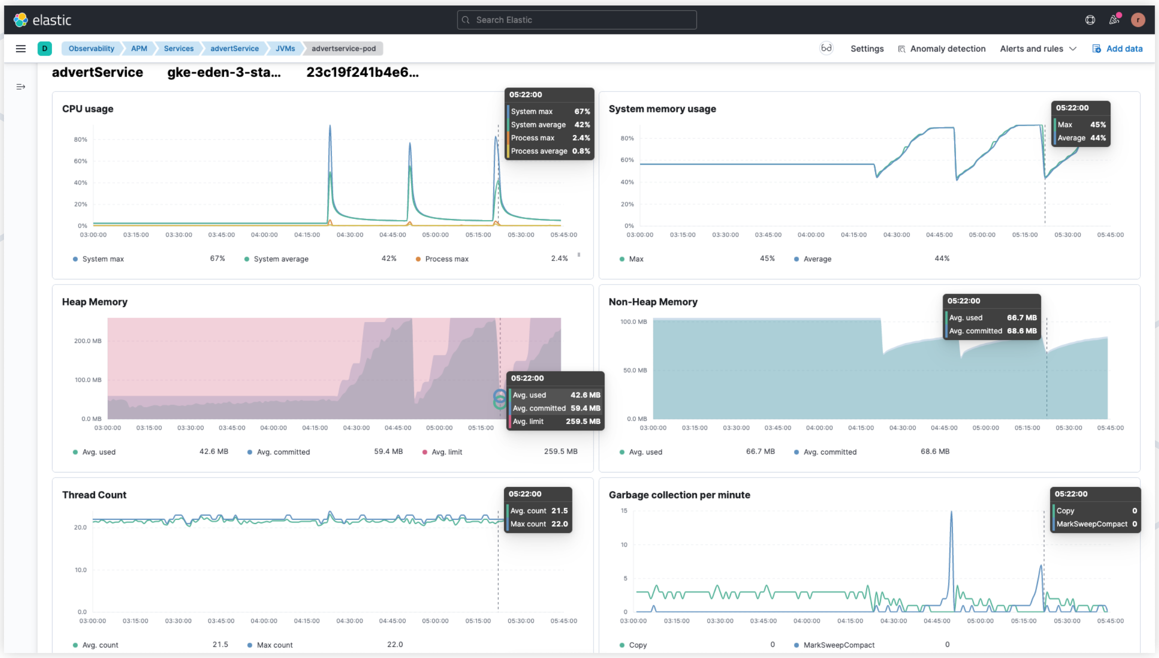
Connections and health indicators

Application trace data not only captures where your applications are spending their time and what they're talking to, but it 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. The request isn't done just because a service sent a response; it still has to be rendered in the browser. If this final step is done inefficiently, you may end up with unhappy users (or worse, *former* users).

Earlier we talked about infrastructure metrics: data points that provide information about how your servers, hosts, virtual machines, or containers are performing. Application performance metrics show you what is going on with your application from the inside-out, rather than the outside-in approach of infrastructure metrics. Infrastructure metrics might show you CPU or memory usage for the pod running your Java application. They may even include a per-process breakdown, but as mentioned above, application performance metrics can also include things like heap usage or garbage collection, both of which are extremely important to know when things go wrong.

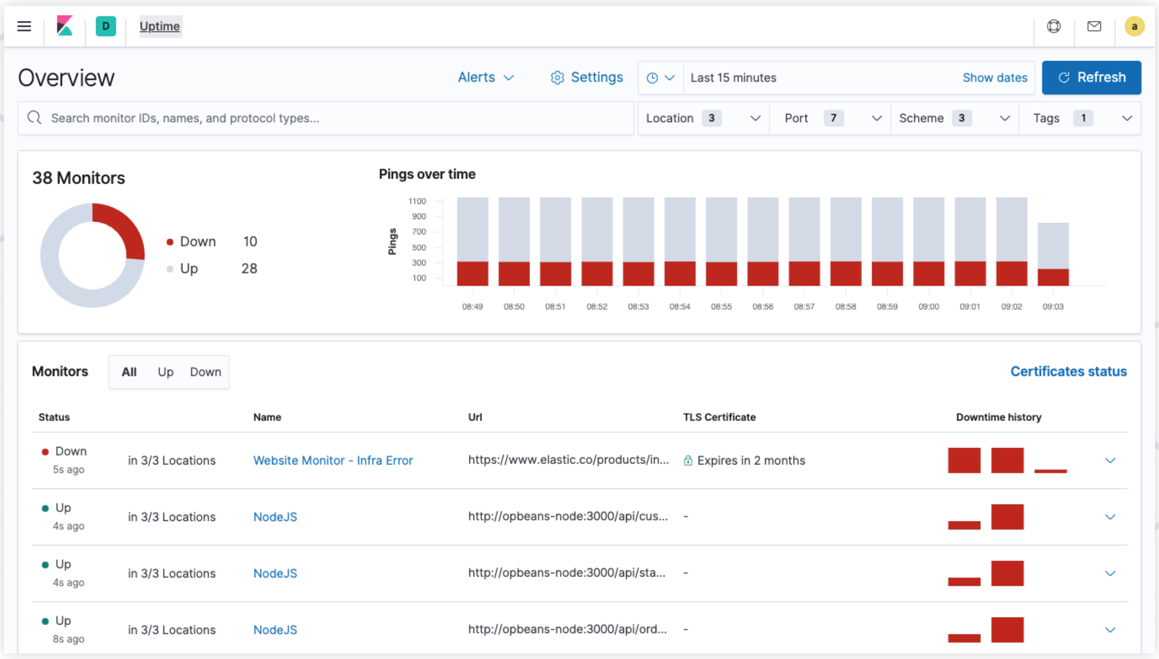


VM metrics via application performance monitoring

Proactive availability testing

Logs, metrics, and APM are generally "after-the-fact" tools, which are helpful when trying to resolve issues, solve problems, or to 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. Proactive testing can be as simple as `ping` or a response test, or it can be as complex as multi-step transaction checks. It's important to proactively and continually test key user journeys such as the checkout process, a product search, or even the action of logging on. Proactively testing these journeys gives you the chance to uncover issues before they impact your users. You identify problems *before* your users ever encounter them.

These proactive tests generate additional latency metrics, response codes, and success (or failure) metrics, and provide the ability to continually verify behavior and response. They can be run from multiple geographic locations (for example, latency as seen from Europe vs. the United States). Also these types of tests can also be used to validate and verify SLAs for any outside services that you rely on, allowing you to correlate downstream telemetry with internal data.



38 Monitors

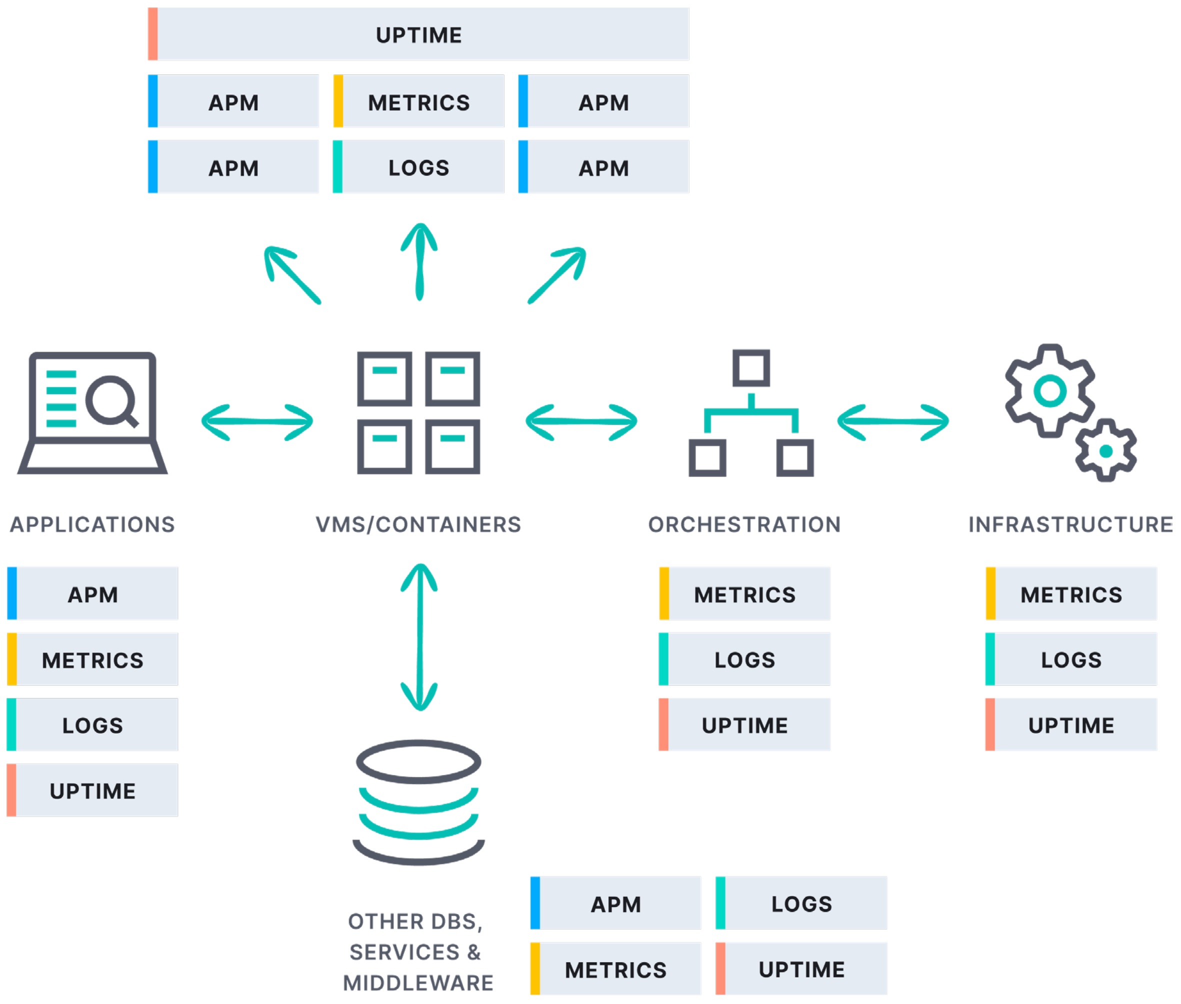
● Down	10
● Up	28

Status		Name	Url	TLS Certificate	Downtime history
● Down 5s ago	in 3/3 Locations	Website Monitor - Infra Error	https://www.elastic.co/products/in...	Expires in 2 months	
● Up 4s ago	in 3/3 Locations	NodeJS	http://opbeans-node:3000/api/cus...	-	
● Up 4s ago	in 3/3 Locations	NodeJS	http://opbeans-node:3000/api/sta...	-	
● Up 8s ago	in 3/3 Locations	NodeJS	http://opbeans-node:3000/api/ord...	-	

Ping test to check for host availability

Putting all your metrics, logs, and traces to use in an observability platform

As seen in the diagram below, there are different types of telemetry to gather at different layers in your infrastructure and your application stack. At the very least, capture logs and metrics from everything, including the infrastructure and orchestration layers. Next, add in traces for any applications and services possible, and leverage proactive tests for internal and external services that you rely on.



Elastic Observability is the ideal solution to get started collecting and exploring the telemetry data from your applications and your entire ecosystem. Multiple deployment options allow you to begin your journey towards observability quickly and easily, with full control of your data. You can start out with a free trial of Elastic Observability, start gathering and visualizing your telemetry data, and begin to improve your users' experience.

Try Elastic Observability

elastic

Search. Observe. Protect.

AMERICAS HQ
800 West El Camino Real, Suite 350, Mountain View, California 94040
General +1 650 458 2620, Sales +1 650 458 2625

info@elastic.co