elastic

Search. Observe. Protect.

Leveraging observability to build better applications

What is observability?

elastic.co →

Table of contents

Introduction to observability

The concept of ***observability*** dates back to the mid-20th century, where it was originally used in control theory to describe how the internal states of a system can be inferred from knowledge of the system's external outputs. While the term is still used in the mathematical context, today it is commonly used in the context of infrastructure, service, and software application stacks.

Software systems are usually designed and described with certain **functional requirements** that specify what the software should do. These might include specific interaction or user stories, like "when a user enters a valid user ID and password pair, they should get logged in." Or they might be a little more specific, such as "if an item has zero stock, it should not be able to be put in a cart." These requirements might not all be written down in the same manner that they would have been when software development followed a strictly waterfall approach, but rather implemented as tests in test-driven development or agile approaches; however, the requirements are still there.

Adjacent to functional requirements are **non-functional requirements (NFRs).** NFRs tend to be the kind of things you can't quite put your finger on to specify, but rather have more of a basic understanding — an "I don't know what art is, but I know it when I see it" kind of thing. Non-functional requirements are also often called **-ilities:* usability, availability, scalability, and maintainability are some examples. If the "checkout" button moves all around the page, or the dropdown menus make it impossible to click the sub-menus because they disappear, then that's a strike against the *usability* NFR. If your site crashes frequently, that goes against availability. You've probably noticed that observability is also an **-ility*, and it can actually help gauge some of the other NFRs.

For many organizations, observability is becoming a critical initiative as companies increasingly digitize and adopt cloud technologies, which dramatically increases the runtime complexity of applications. In this ebook, we'll walk you through some concepts and considerations for observability:

- ☑ **How software architecture and cloud technologies have evolved**
- ☑ **Who can benefit from observability and why it's even more important today**
- ☑ **What an observability solution should include**
- ☑ **What a unified observability platform can do for you**

And finally, we'll wrap up with important things to consider when researching an observability solution and how to get started.

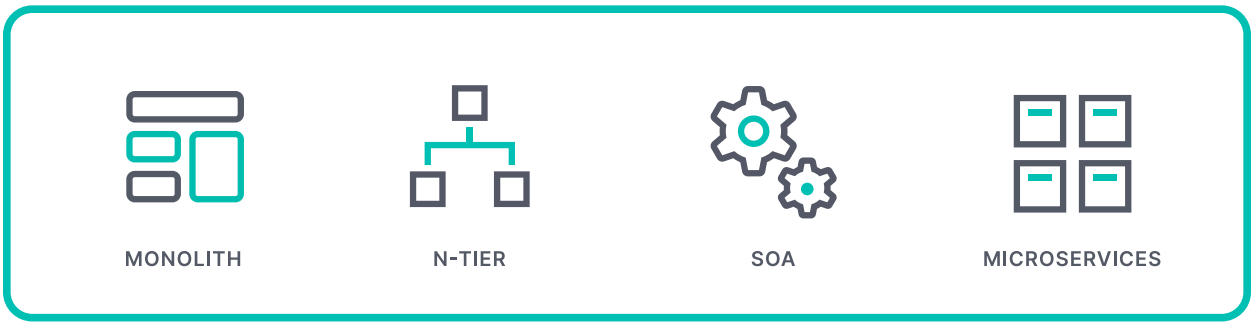
Why observability matters in modern DevOps

During the era of data centers and monolithic applications, production changes were infrequent and planned. Dependencies were easily understood and overall application health could be determined by monitoring the variance of a few known metrics: **known unknowns.**

With the distributed nature of modern applications today, no single team or individual has a complete picture of all the dependencies. Telemetry data (metrics, logs, and traces) is often siloed in different tools. Developer and operations teams are spending too much time triaging problems due to swivel-chair investigations, resulting in a higher mean time to resolution (MTTR). To address this increasing application complexity, you'll need more data to truly understand your environment and your users' experience.

The first step towards observability is monitoring: gathering the logs, metrics, and application traces from your infrastructure, services, and applications. Basic monitoring data allows you to answer key questions about your application ecosystem, such as "which of my servers are over utilized?" or "which applications have high response times?" An observability solution can help you extract even more information from your monitoring data — allowing you not only to see the internal state of your systems, but also help you uncover **unknown unknowns:** the things that may be going wrong that you didn't even know to look for.

The evolution of software architecture and cloud technologies



Software and hardware deployment models have evolved over the last few decades. What started out as monolithic applications with lengthy, brittle code, migrated to client-server and then N-tier architectures. This was the rise of modularity in software development and application architecture. And with the advent of C++ and more programmer-friendly third-generation programming languages, the individual programs in these topologies became smaller and less complex than their monolithic predecessors. The next step, service-oriented architecture (SOA), was more evolutionary than revolutionary. SOA added definition around how to break up applications, and introduced the concept of pooling small, discrete units of functionality (services) to deliver applications.

The software deployment model evolution really couldn't have happened without parallel advancements in hardware utilization. Monolithic and even early client-server and N-Tier applications ran on bare metal. A server had one operating system and ran one application — an architecture that was not really scalable when talking about SOAs, which could have dozens of different services. Hypervisors enabled virtualization on those same machines and made it possible to run several virtual machines (VMs) on a single host, paving the way for SOAs.

We're still in the middle of a software evolution. The current phase includes microservices and serverless technologies, both on premises and in the cloud. This transition to microservice architectures removes the brittle interfaces of SOA deployments that often impact software iteration and evolution.

In the lock-step evolution of software architectures and deployment models, the popularity of microservices and the concept of cloud-native computing have grown in parallel. Cloud-native doesn't mean taking your application and throwing it up on a cloud provider, but rather an overall evolution of the philosophy of software development and deployment. The goal is software that can scale user capacity up or down based on demand. Cloud-native also enables an organization to be more agile, allowing a continuous integration and deployment cycle.

While virtualization paved the way for N-Tier and SOAs, it wasn't quite right for microservices: VMs still need to be provisioned, complete with an OS. The next step in the hardware deployment model was containerization and orchestration (usually with Kubernetes or a derivative), which was ideal for microservices and cloud-native models. Containerization and Kubernetes allows for dynamic deployment of workloads and was designed for cloud-native architectures (load balancing, scaling, upgrades, and more).

Serverless or Function as a Service (FaaS) technologies take things even a step further, relying on short-lived, ephemeral operations run in response to events. There's no need for the complexities of a microservice environment. Serverless technologies have sprung up at an increasing rate, providing an on-demand, lower-cost alternative to the more complex infrastructure needed by full-blown microservice ecosystems.

Monitoring and logging also need to evolve

The smaller footprint of monolithic and N-Tier applications, combined with infrequent changes, makes them inherently easier to monitor. While the applications themselves were bigger and more complex than discrete microservice components, we only had to worry about log files and metrics from a few machines and services.

Virtualization and DevOps shifting to the right is really changing the software development landscape. The higher resource utilization made possible by virtualization and containerization increases monitoring complexity. Not only do we need to monitor the base machines that are running the hypervisor or orchestrator, but we also need to monitor the individual "hosts" (containers and VMs) as well as the hypervisor and orchestrator themselves.

We should also monitor the applications running in them. Given the higher resource utilization in containerized systems, we're much more likely to run into "noisy neighbors": a situation where one container consumes too much CPU or memory to the detriment of others. When running managed servers or serverless, we can't monitor the underlying infrastructure, but we can monitor service calls.

It's no longer just client-server or a simple LAMP (Linux, Apache, MySQL, PHP/Perl/Python) stack. Modern software applications are living, breathing things, made up of hundreds or even thousands of different services. Any one of these services can break your system — which leads us to the need for observability.

Observability brings insights from your application telemetry data

As an example, imagine a process with an order processing system: transactions are failing, but not every one. Monitoring would be able to show that something is wrong and even alert you. But observability would let you correlate issues and find that globally, only people checking out using British pounds to pay are having trouble. This operational insight then leads you directly to issues with the currency API. By investing in an observability initiative, you can intelligently troubleshoot and correlate application issues, no matter how complex your application environment.

Considerations for an enterprise observability solution

When a solution that aggregates, correlates, and inspects the telemetry data from our applications and infrastructure can tell us **what is actually going on within our application environment, that's when we have observability.** Observability is what the solution you use to evaluate that monitoring data provides. Without telemetry data from your applications, services, and infrastructure, there is no observability. And you will need **all** your data, for reasons we will discuss later.

Let's walk through some essential capabilities for an observability solution.

Efficient ingest and storage of all your observability data

Complex software systems mean that there's even more application data to monitor and store. Microservices, databases, message buses, key-value stores, abstraction layers — we should capture the telemetry data from all of these components and store them for future troubleshooting. Data ingestion isn't just about proprietary parsers and importers: we also have emerging open standards, such as OpenTelemetry, Zipkin, and Jaeger, to lean on. When reviewing observability solutions, understand how pricing models can impact your ability to instrument, ingest, and store data for your entire application system across dev, QA, and production.

Typical types of application telemetry data

It's important to gather anything and everything that you can in your application ecosystem: infrastructure logs and metrics from the hosts, servers, VMs, containers, pods, and the components that orchestrate them (hypervisors and container orchestration layers). Then you will need the logs, metrics, and availability data from the services that you produce (applications) and those that you consume (message buses, key/value stores, and databases). And when possible, trace data from the applications that you manage.

We go into more detail on telemetry data in a dedicated ebook, but here's a quick breakdown of the different types of telemetry data.

Logs

Log messages (or logs) are text that is output by an application or service when the execution reaches a certain point in the code. Logs can be structured or unstructured and usually indicate that something has happened — the database started, a query took too long, an error occurred, or the ubiquitous `got here` when you're scratching your head debugging code.

Logs can be one line:

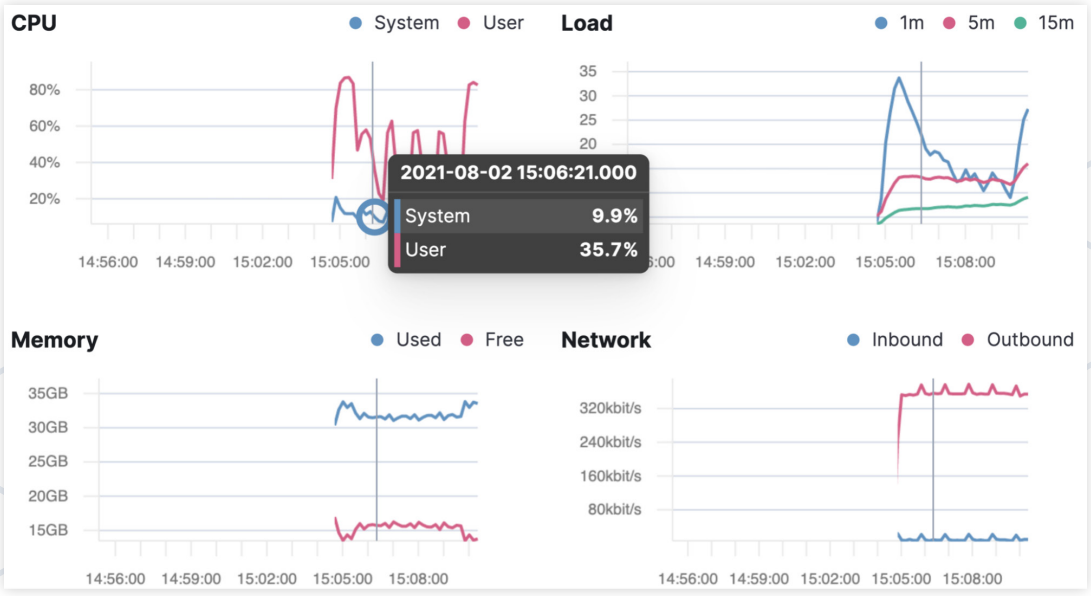
```
2017-07-31 13:36:42.585 CEST [4974] LOG:  database system was shut down
at 2017-06-17 16:58:04 CEST
```

Or they can span multiple lines:

```
2017-07-31 13:36:43.557 CEST [4983] postgres@postgres LOG:  duration:
37.118 ms  statement: SELECT d.datname as "Name",
               pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
               pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
               d.datcollate as "Collate",
               d.datctype as "Ctype",
               pg_catalog.array_to_string(d.datacl, E'\n') AS "Access
privileges"
        FROM pg_catalog.pg_database d
        ORDER BY 1;
```

Metrics

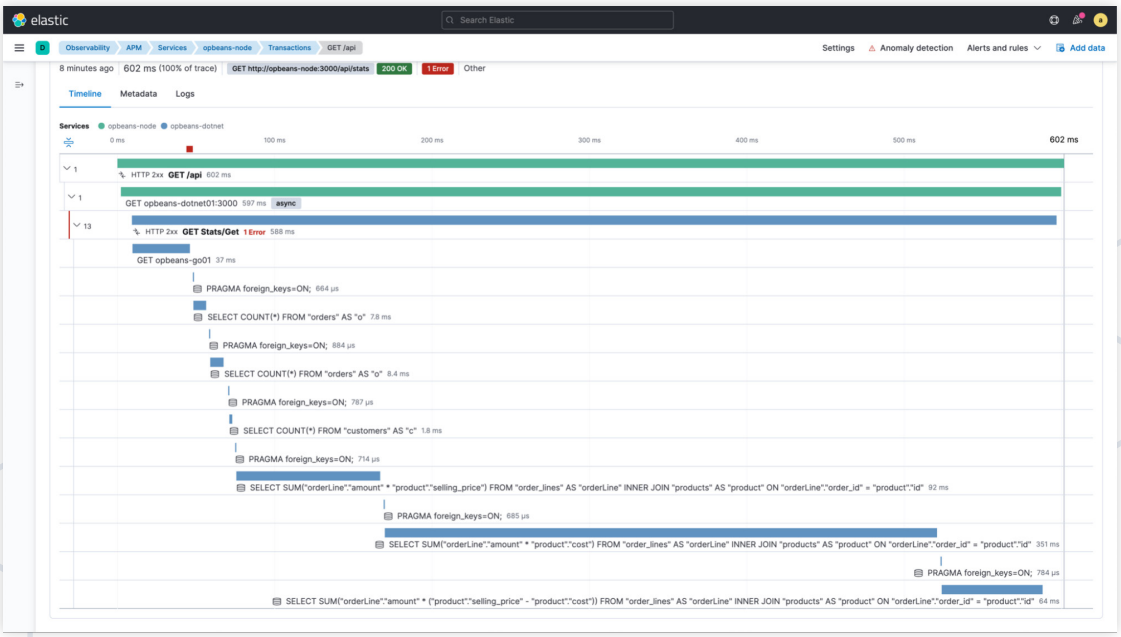
Metrics are numeric values that reflect some characteristic of a system. They can be counters that increment each time something happens (for example, when a page loads), they can be accumulators (`Sent Bytes` and `Received Bytes`), or they can be aggregated or calculated over a period of time (system load).

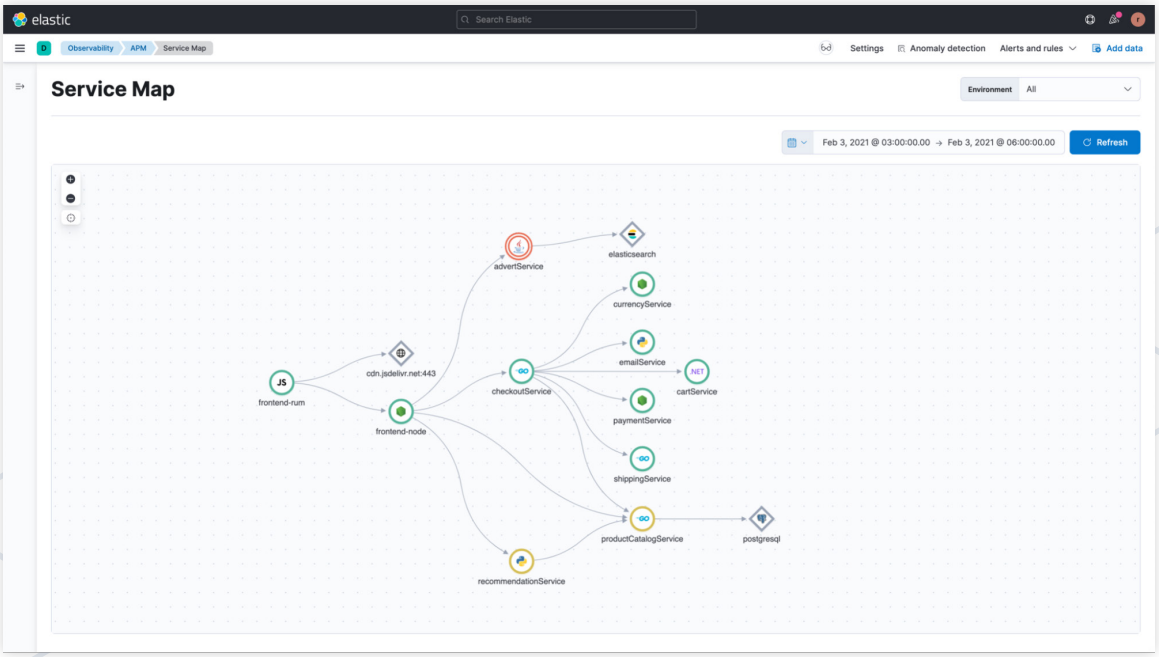


It's important to note that log data can also include metrics, so make sure that your monitoring and observability solution can readily extract metrics from your log data without losing contextual information.

Traces

Traces show the activity and the path(s) that requests take through an application, along with which components are calling other components or services. They are usually represented by a waterfall view and can show a trace for a single operation or a distributed trace, which can represent an entire transaction across multiple services.





A trace can also be represented as a service map showing service connectivity. Traces are often the first stop when debugging issues or outages. They help point out where in the logs or which metrics might be best to review.

Security data

The lines between observability and security are blurring, with similar data being used in both disciplines. Security data should be considered a part of a unified observability platform. After all, security data is simply additional telemetry data analyzed from a different perspective. The ability to triage and identify security issues should be part of your observability solution, as well as the ability to act upon and isolate problem infrastructure.

Simplifying ingestion of your observability data

Systems and services generate telemetry data in many different ways, so it's important for an observability solution to guide you through the process of onboarding all types of data. Using wizards, step-by-step instructions, or even the central management of collectors and agents will make data ingestion more intuitive. This level of support also helps abstract out the process; you shouldn't have to become an expert in a database's metrics API to get value from its key performance indicators (KPIs).

Logs and metrics are helpful for looking at your application from the outside, in. They let you capture data that someone has accounted for: system metrics like CPU or memory utilization, application-specific KPIs such as the number of hits for a page, or logs, like "item shipped" or "file downloaded: 758 kbytes." But they don't give you any information that you didn't know to ask for or didn't collect.

Traces can help. If logs and metrics give you a view of your application from the outside, traces give you a view from the inside. And gathering application trace data shouldn't be a heavy lift. Make sure that the tools that you use to instrument your code provide a quick getting started path, but also provide the ability to customize what gets instrumented. You will also need to enrich your traces with your own metadata, in addition to metadata that enhances the context and integration information as described below.

The collection of telemetry data should be as dynamic as your application ecosystem. If you're running VMs, you should be able to configure the monitoring in your base images so any VMs that spin up automatically will get included. If you're running in containers, make sure that the solution you choose can automatically detect and monitor new pods or containers as they spin up, and that it can provide the context needed to determine whether problems are programmatic or environmental.

The challenge of high-dimension, high-cardinality, and high-volume data

The easy ingestion of data isn't the only part of the equation. Systems, services, and infrastructure each generate telemetry data that may have a unique shape. If you look at the two log examples mentioned earlier, the only datagram that's really in both is the timestamp. If you think about where this data is stored and pretend it's going into a relational database, we'd be talking about a table with a very large number of columns and mostly sparse rows. Each metric is a dimension — CPU, memory, disk — and each of these dimensions has values. Each dimension then outputs its own time-series set of data.

Perpendicular to the high-dimension data, we have high-cardinality data. High-cardinality means having many instances for each of the above metric dimensions (for each host, server, pod, or container). If we continue the relational database analogy, not only would we have a wide table, we'd also be looking at many, many rows. The ability to search, filter, and aggregate across (and within) a large number of fields with an even larger number of events becomes paramount to the success of our observability goals.

Observability data growth tends to be exponential; we're constantly adding new hosts, endpoints, and services. Systems should be able to handle this complexity of growth and provide users the ability to access and analyze their data. Alongside the ability to leverage high-dimension, high-cardinality data is the need to manage telemetry data. Like many other types of information, observability data loses value over time. But data that's a year or two old can still have value. The ability to manage how long we can keep data, and control how we can access and analyze it, has a direct impact on how we can compare trends to the previous week, month, or year. Since observability is all about finding the **unknown unknowns**, having as much of your telemetry data available as possible is key to finding insights and improving application performance.

Other additional considerations around your observability platform include how you get charged for it and who controls the data. Is it agent-based, based on data ingestion or maybe based upon resource usage? As you instrument more of your application environment or if it grows substantially, your operating expenses will grow in kind. Do you choose to store more or less of it to control expenses in an agent-based model? Do you ingest less data to minimize costs?

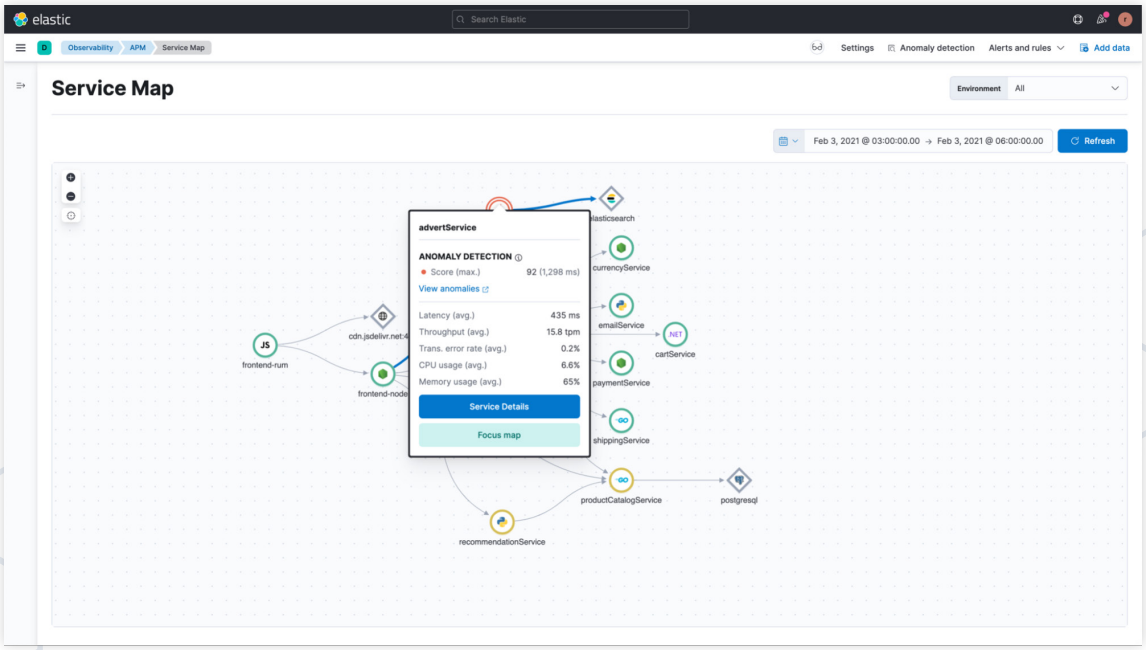
For your observability solution to work effectively, you need to collect as much data as possible at the detail level needed. And you need to have the ability to access your data in the ways that you want. In other words, **make sure that you own the data that you collect** and that it's available for you to analyze in whatever way you choose. You can always remove unneeded data, but it's much harder to add it back after the fact.

Analyzing all your observability data for actionable insights

Logs, metrics, and traces are, obviously, completely different kinds of telemetry data. But a capable observability solution will put them in context and help you act on that information. To get the most value out of an observability platform, your solution should provide intuitive data visualization and navigation: purpose-built interfaces that let you interact with your data in an easy and flexible way, and tools that allow you to filter and find the logs for a certain application on a specific day. In other words, the ability to quickly build custom metric aggregations without needing to be a data scientist — because when a performance issue strikes, you need to research and resolve the issue immediately.

When you're wearing your operations hat, you're likely more interested in infrastructure and services when exploring the overall health of your ecosystem. In this case, you'd likely be using infrastructure metrics as your launch point. This stance necessitates a high-level overview of your systems, whether it be from an actual server or cloud instance point of view. Or you may need to look at them through a container or pod lens. We absolutely need to be able to pivot and look at the infrastructure from different perspectives. If you see high utilization of your infrastructure, you need to be able to jump directly to a log viewer without losing context so you can see the logs of exactly what you're investigating.

With the developer hat on, you might instead start out at the service map in application performance monitoring (APM), which shows you which services could use some attention. This view allows you to triage any problems while keeping the context of what you're looking at (like unravelling a ball of yarn but keeping the end in your grasp).



The ultimate goal? Full end-to-end observability across your application environments with all of your telemetry data. The ability to see across your infrastructure metrics, through the application, and all the way to the end-user experience. And the ability to group and correlate telemetry data to your specific needs at any point and time. A comprehensive observability platform will allow you to monitor day-to-day application performance, troubleshoot known issues, and even uncover problems that you were totally unaware of.

Workflow context and correlation for troubleshooting

We've mentioned context a couple of times. Observability is all about using logs, metrics, and traces together. And the metadata that events are enriched with can make or break the ability of the solution to detect and find the root cause — basically, a way to manage the **known unknowns**, or problems that you are aware of in your application environment.

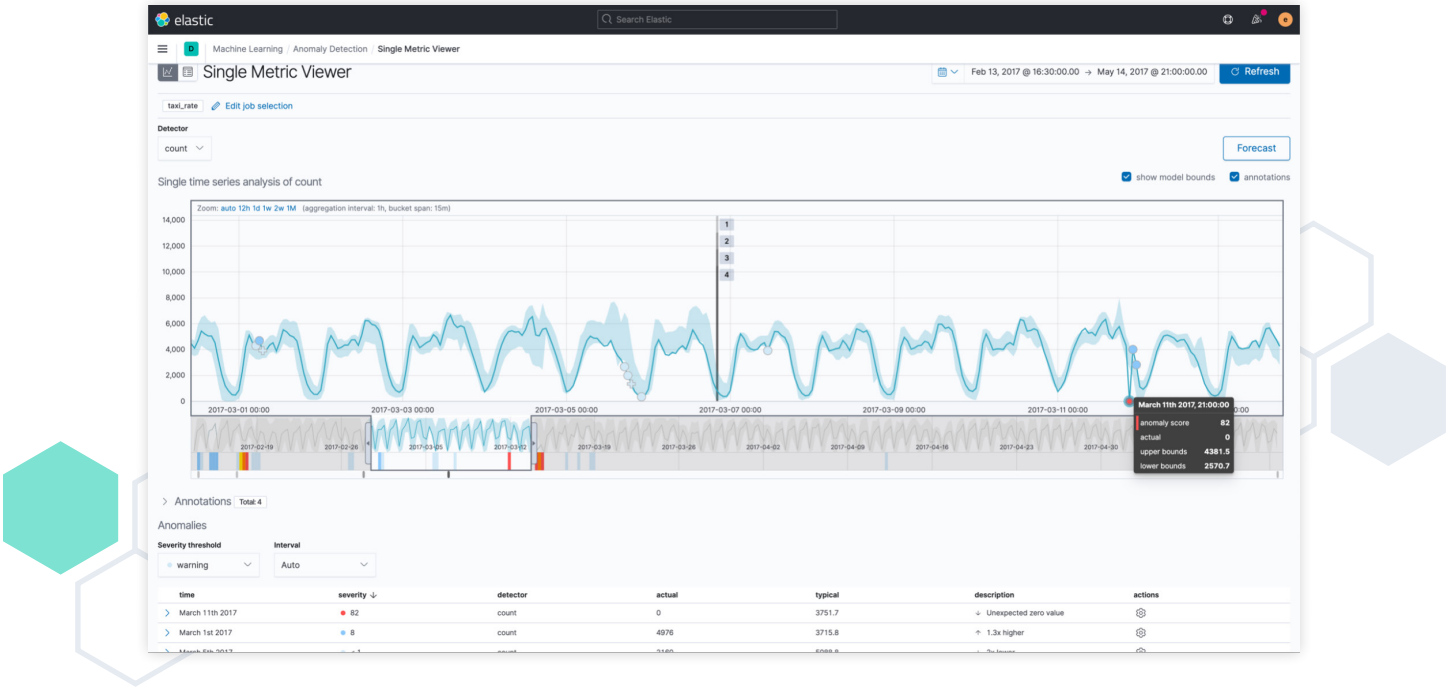
It's not very often that you find the details of a problem in the first place that you look. You'll need the ability to see the logs from an application context and then to jump directly to the relevant metrics and see the history for the host or container that the problem application is running on. Carrying that context along the way as you investigate can save a good deal of time, clicks, and money as you unravel the problem.

Addressing potential performance issues with anomaly detection, AIOps, and alerting

Earlier we mentioned that observability helps you understand the overall health of your system. Viewers and dashboards help triage problems and track down errors. But where it really adds value is letting you know proactively where there might be a problem. There are several buckets of other capabilities that an observability solution should include to enhance your troubleshooting capabilities and assist with **unknown unknowns** — the blind spots in your application performance monitoring.

Detect outliers and errors relative to baseline performance

Applications and services don't get a constant amount of traffic. At the very least, it's probably cyclical, based on the time of day or even the day of the week. Your observability solution should be able to detect the trends and patterns and let you know when something falls outside the normal window.



AIOps, powered by machine learning, helps to reduce alert fatigue by correlating differences to help reduce mean time to detection (MTTD) and mean time to resolution (MTTR). It even takes into consideration the cyclical nature of modern business as it runs. Being able to automatically identify things that are out of normal range, rather than specific thresholds, reduces the time spent on manually reviewing dashboard reports.

Alerting and notifications

Alongside anomaly detection is alerting and notifications. When anomalies occur you need to know about them in the manner that you choose. It's not just anomalies that you need to be alerted on, either. Often, cloud-native software is deployed using a DevOps model: teams aren't broken up into operations and development, but instead wear multiple hats with code continually being built and deployed.

Proactive monitoring to meet SLOs and SLAs during development

Logs, metrics, and traces are great for finding problems, slowdowns, or errors in your applications and services. Unfortunately, when relying on them, it usually means that your users have experienced a problem. It's important to proactively and continually test key user journeys — the checkout process, the product search, or even the action of logging on. Proactive monitoring during development uncovers issues before they impact your users.

It's not just your internal applications and services that you should monitor. Use proactive observability and testing to keep an eye on any external services that your system relies on. The services on the backend may be breaching their service-level agreements (SLAs), impacting your service-level objectives (SLOs).

Compare, contrast, and correlate with ad hoc queries to improve performance

A robust observability solution should be able to compare and contrast data: How is the performance today compared to yesterday? Is the new version encountering more errors than the previous version? Are mobile users seeing more slowdowns than desktop users? The ability to ask ad hoc questions of your data, then pivot and refine your investigation, can help you understand trends and patterns. This is one additional reason for a unified data store and storing your high-cardinality data. Your observability solution should not only help you compare and contrast, but it should guide you along the way and let you know what differences may be contributing to a bad user experience.

Observability helps to find unknown unknowns

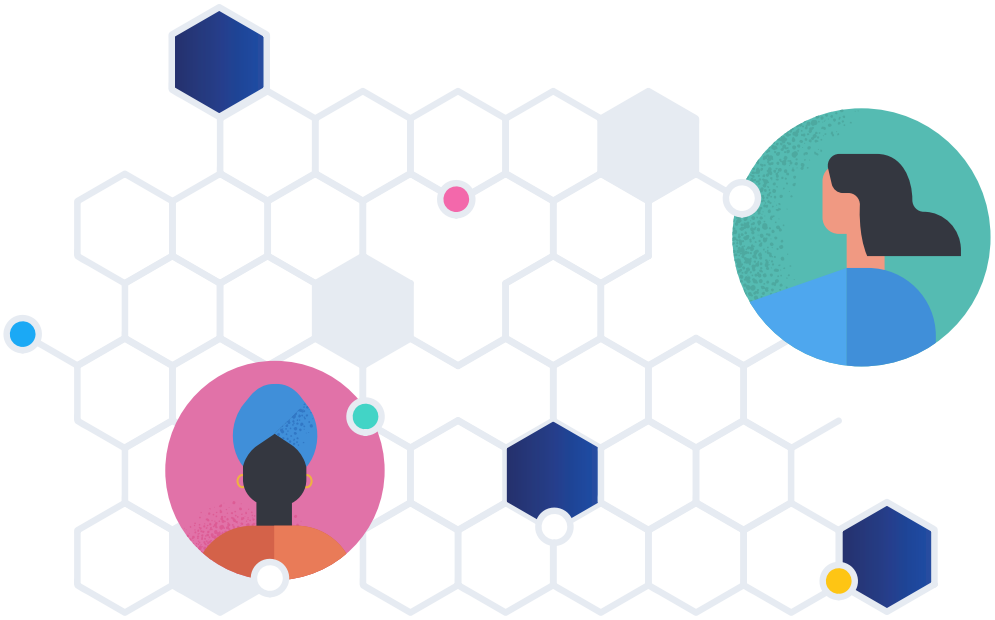
Related to the detection of outliers and errors is the ability to find and resolve problems in areas that you don't even know to watch. Often the hardest problems to solve are the ones that are transient and infrequent. They might arise due to changes in environment, the application, or a combination of factors. These are the types of problems that solidify the need to capture anything and everything, datawise. A unified data store allows you not only to determine what was happening at the time of the issue, but also helps you understand when unexpected events occur by putting different signals into context and at your fingertips when you need to explore further.

A case for a unified observability solution and avoiding tool silos

Quite often, organizations start out with a need for log aggregation. They get that up and running, then decide that they should also start gathering infrastructure metrics, so they add another tool. This process repeats again with APM. Before long, they've got three or more tools to monitor their application ecosystem, along with already siloed data stores.

While it's definitely possible to triage and resolve issues and errors with siloed tools, the swivel chair approach to diagnosing issues across multiple, single-purpose tools definitely makes it harder. You lose the context and integration aspects of a good solution. Also remember that multiple tools means that all aspects of the overhead are multiplied as well: administration, training, and storage, to name a few.

Having all telemetry data in a single data store with the right observability platform that scales and provides coverage across dev, stage, and production is ideal. With the right research and investigation, finding an observability solution that allows you to instrument and observe everything in your IT environment in a cost-effective manner is an important goal. And gaining a true understanding of the performance characteristics of your systems, applications, and services is critical to increasing developer productivity, accelerating innovation, and ensuring a great customer experience.



What can unified observability do for me?

The benefits of observability vary based on the hat you're wearing: if you're looking at it from the perspective of a developer you're going to be looking at things differently than if you were approaching it from an operations perspective, and different again if you're the business owner.

While not an exhaustive list, here are some sample IT operations, developer, and DevOps questions that you would be able to answer with a unified observability solution.

- Which servers are overloaded?
- What is our average response time for a given operation?
- Which services cost me the most on my AWS bill?
- What is causing some users to experience longer load times than others?
- Which services that we consume are nearing their SLAs?
- What would be a good SLO for this service?

- What services does service Z call?
- Which of my services is unhealthy?
- Which service should I try to tune first?
- What is the overall user experience for my site?
- How has the latest change impacted performance?
- How has the latest change impacted stability?
- Should I roll back the latest change?
- Which version has the best performance?
- What logs would be interesting to look at?
- Are any services behaving abnormally?

In addition to these day-to-day questions, a unified observability solution can help platform owners consolidate costs with one single platform to purchase, learn, and maintain. Choose an observability solution that will grow and scale with your organization over the long term.

Observability solution checklist

As a bonus, we've put together a checklist of things discussed above along with some spots for you to drop in some common services, while leaving room for you to add your own technologies, needs, and concerns. Customize the list based on your environment and the tools that you use. We hope that the checklist will help you gain a better understanding of what you should consider when planning an observability initiative. For more details, please download our Observability solutions: An interactive checklist available online.

Getting started with Elastic Observability

As you embark on your observability journey, there are definitely many considerations to think through about your application environment and needs. While features and functions are important for your observability platform, the openness and extensibility of your observability platform, access to your telemetry data, and the total cost of ownership will impact your long-term success. And be prepared for the fact that the lines between observability and security are blurring. The same telemetry information you are collecting to observe your environment can also be used for security insights and detecting intrusions.

Now that you've got a good idea of some key considerations for your observability solution, it's time to get started. Elastic is the only observability solution built on a search platform that ingests all telemetry, adds context, and correlates for faster root cause analysis, significantly reducing MTTR and increasing developer productivity. You can start out with a free trial of Elastic Observability, start monitoring, and begin to improve your users' experience today!

Try Elastic Observability

Search. Observe. Protect.

Elastic makes data usable in real time and at scale for enterprise search, observability, and security. Elastic solutions are built on a single free and open technology stack that can be deployed anywhere to instantly find actionable insights from any type of data — from finding documents, to monitoring infrastructure, to hunting for threats. Thousands of organizations worldwide, including Cisco, Goldman Sachs, Microsoft, The Mayo Clinic, NASA, The New York Times, Wikipedia, and Verizon, use Elastic to power mission-critical systems. Founded in 2012, Elastic is publicly traded on the NYSE under the symbol ESTC. Learn more at elastic.co.

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