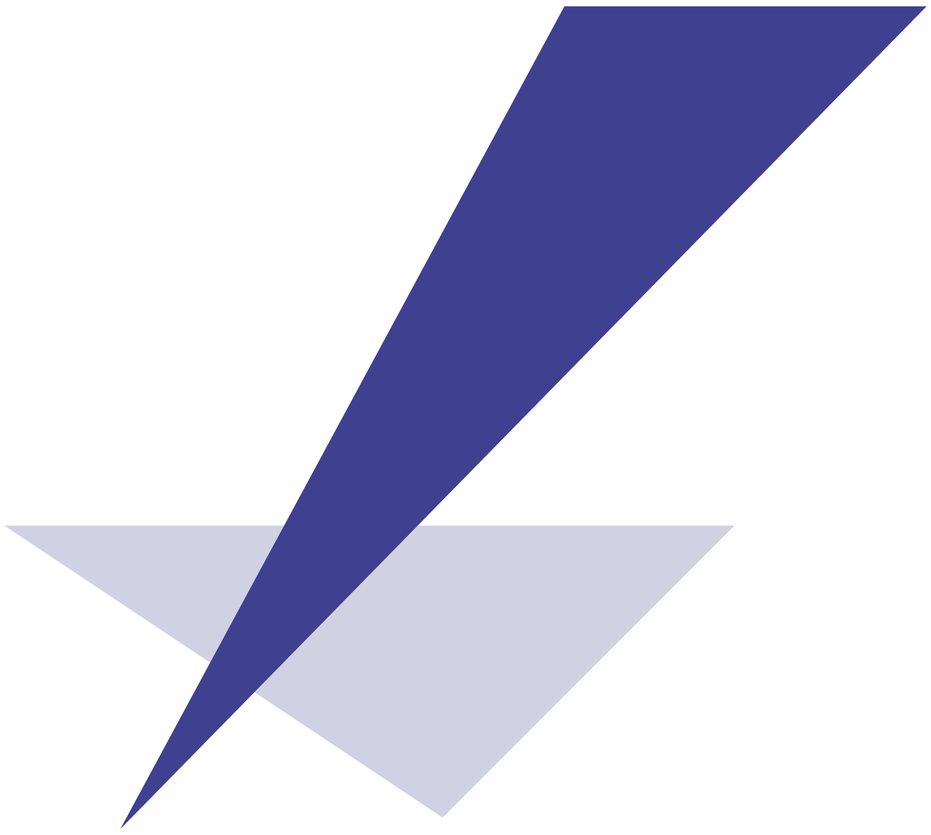
THUNDRA™

Monitoring vs. Observability



Contents

Abstract

Apps in production fail for various reasons. No matter how much effort you expend, there will always be something that goes wrong. If you don't effectively instrument your application's components to be observable, you'll have a hard time debugging production issues. On the other hand, even an observable system does not have the answers to all issues.

You should always examine the data you have to determine its usefulness. Observability involves having the right data to help you get answers to known and unknown problems in production. You have to continuously adapt your system's instrumentation until it's appropriately observable to a point where you can get answers to any questions needed to support your application at the level of quality you want.

Monitoring vs. Observability: What's the Difference?

The complexity of modern applications has increased, as the various components of the distributed app must be monitored. The need for a more inclusive debugging and diagnostics method for modern applications has never been more apparent, especially in distributed systems. Fortunately, cloud and orchestration tools make it possible to easily provision new environments and deploy and manage modern applications.

Historically, a combination of monitoring and testing has been used to handle predictable failures, but they are less effective with unpredictable failures. This is where observability comes into play. Observability has its roots in control theory, which addresses how well you can infer the internal state of a system by looking at its output.

Since observability is still pretty new, the line between observability and monitoring seems blurry for development teams. In this white paper, we shed light on observability and monitoring and provide insights into some of the tools you can use to achieve observability.

THE MOVE TO MICROSERVICE ARCHITECTURE

Microservice architecture is now the de facto standard for web applications. In a new survey, 61% of respondents stated that they had used microservice architecture for a year or more, and 55% stated that microservice architecture is critical to their success. Microservices are a collection of loosely connected services. One of the advantages of microservice architecture is that the system's different components can be individually created and implemented on different platforms.

Businesses can structure their teams around business problems, with each team working independently of others to become more agile and to scale. With the monolithic way of building applications, it wasn't possible for organizations wanting to adopt the DevOps mindset to do so. Put differently, if your organization wants to scale up, you need to split things up.

Although microservice architecture has advantages, as the services grow, its complexity becomes apparent, and debugging and diagnosis of the various interconnected parts become more difficult.

With the advance of DevOps paradigm and improvements in cloud technologies, migrating to cloud has become an inevitable move for companies at any scale from any industry.

AWS, the most popular vendor for cloud operations, provides various different options for compute services such as Amazon EC2, Amazon ECS, AWS Fargate, and AWS Lambda. In this whitepaper, we'll compare the compute platforms from different perspectives such as operability, cost optimization, technology enablement and more.

Since migrating to a microservice architecture is no longer an option but a must for organizations wanting to adopt the DevOps mindset, businesses must figure out how they can avoid losing the ability to know about system failures and react to them as the system becomes more complex.

Could monitoring provide this capability?

MONITORING

According to the SRE book, a monitoring system should address two questions: What is broken and why? “What’s broken” is about the symptom, while “why” is about a (possibly intermediate) cause. A good understanding of the distinctions between the “what” and “why” enables you to effectively monitor a distributed system with minimum noise and maximum signal. Monitoring allows you to watch and understand your system’s state using a predefined set of metrics and logs. We monitor applications to detect known failures.

Monitoring is crucial for analyzing long-term trends, for building dashboards, and for alerting. It lets you know how your apps function, how they’re growing, and how they’re being utilized.

When monitoring is combined with alerting, your system is able to tell what is broken or what is about to break. With this data, you can easily understand your app’s behavior, detect problems, and rapidly resolve those issues before your users are impacted.

For instance, you may monitor a system for known or predicted problems like storage running out of disk space or failure from an I/O bound service, which can be very effective, but when it comes to unknown unknowns or unpredicted failures, monitoring is not enough.

As Cindy Sridharan beautifully puts it in a post:

> “For monitoring to be effective, you need to be able to accurately identify a core set of metrics that indicates the health of a system or a set of failure modes of a system.

Monitoring's limitation comes from the fact that in order to be effective, it requires you to know what's normal. You have to know which metrics to track, but production failures are not linear. You can't predict failures all the time.

Since monitoring can become so complex that it's complicated to change and burdensome to maintain, it's always good to design your monitoring system to be as simple as possible. The following points are important to remember when you're choosing what to monitor:

- The rules that catch real incidents should be as reliable, predictable, and straightforward as possible.
- Your monitoring data needs to be actionable.
- You should avoid using any data collection, alerting, and aggregation configuration that's rarely exercised.
- You should avoid using signals that are collected but not used by any alerts and not exposed in any prebaked dashboards.

Although monitoring doesn't make systems wholly immune to failure, it should provide a reasonably good view of the system's health. Even if your monitoring data is not directly used to drive alerts, it should at least give you a panoramic view of a system's behavior and performance in the wild. The data should also provide visibility into the effects of any fix deployed. If there is a failure, the data should help you understand its impact.

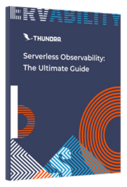
The bottom line is that monitoring is an indispensable tool for building, operating, and running systems.

OBSERVABILITY

Observability, which originated from control theory, measures how well you can understand a system's internal states from its external outputs. You can think of observability as a superset of monitoring in the sense that if a system is observable, it can be monitored. Observability provides insights that aid monitoring. Monitoring is what you do after a system is observable. Without some level of observability, monitoring is impossible in the first place.

An observable system allows you to navigate from effects to cause in a production system. It's in a state that enables you to understand and measure the internals of the system. Observability tells you what, where, and why.

Observability will also help your team own the
system and better understand how it behaves in a live environment, especially if you're adopting cloud-based distributed architectures, such as those commonly found with microservices and serverless architectures.



This paper walks through the problem of serverless observability. It starts by defining the observability problem space, explaining how debugging and tracing are only part of a larger picture of application behavior. We'll explore AWS CloudWatch and X-Ray as a first attempt at observability. Then we will detail several open source serverless availability tools, before discussing Thundra's automated approach to observability. Finally, We'll compare some alternatives and try to pave the path to ultimate serverless observability by distributed tracing, metrics and logs.

Observability is uniquely positioned to answer the questions that arise when you troubleshoot or operate modern distributed systems. It helps you find answers to questions like:

- What services did a request go through, and what were the performance bottlenecks?
- How was the execution of the request different from the expected system behavior?
- Why did the request fail?
- How did each microservice process the request?

Observability can be divided into three primary pillars: logs, metrics, and traces.

LOGS

A log is a timestamped, immutable record of discrete events that can provide you with a comprehensive account of what happened in an application at a specific time. Event logs may come in plaintext, structured, or binary format, but all formats contain a payload.

With proper logs, you can identify unpredictable behavior in a system and understand what changed in the system's behavior when things went wrong. Logs can either be structured or unstructured. When generating logs, it's important you ensure that they're readable by both humans and machines. Therefore, it's highly recommended to ingest logs in a structured way, preferably in JSON format, so that log visualization systems can auto-index and make logs easily queryable.

METRICS

Metrics are the foundation of monitoring. They are values that express some of a system's internal state. Typically, metrics are defined as counts or measurements that are aggregated over a period of time.

Metrics will tell you how much of the total amount of memory is used by a method, or how many requests a service handles per second. As stated earlier, unless a system is made observable, it’s impossible to collate metrics.

TRACES

A trace describes the whole journey of a request as it moves from one node to another in a distributed system. It uncovers the request’s unintentional effects and provides visibility into its structure.

For an individual transaction or request, a single trace displays the operation as it passes through an application. Traces allow you to g et into the details of particular requests to determine which components cause system errors; to monitor flow through modules; and to find performance bottlenecks.

The Relationship between Observability and Monitoring

It’s not helpful to conceptualize the relationship between observability and monitoring as “observability versus monitoring.” This is because observability isn’t a substitute for monitoring and doesn't eliminate the need for monitoring. In fact, observability and monitoring complement each other.

The two are symbiotic, not mutually exclusive, and they serve different purposes.

While observability is about being able to understand the internal states of a system by interrogating or inspecting its output, monitoring refers to the actions that take place in observing the quality of a system's performance over a specified period.

Observability is a state, while monitoring is something you do. Monitoring tells you when something is wrong, while observability enables you to understand why. Monitoring is a subset of and key action for observability. You can only monitor a system that's observable.

Monitoring tracks how applications are performing in terms of access speeds, connectivity, downtime, and bottlenecks. Observability, on the other hand, drills down into the "what" and "why" of application operations by providing a high-level overview of a system's health and granular insight into its specific failure modes.

Monitoring tells you about the overall functionality and health of your system. Monitoring is best limited to key system and business metrics obtained from blackbox tests, known failure modes, and time-series-based instrumentation. On the other hand, observability is perfect for debugging purposes, and in the event of an error, it enables you to zoom in and understand what happened.

No system is immune to failure, and it's impossible to predict how production systems could misbehave or the failure modes that systems could potentially run into. That's why it's important to be armed with evidence and to build systems that can be easily debugged.

Debuggable software should answer questions about itself, and the evidence shouldn't have to be inferred from percentiles, averages, aggregates, or other forms of data primarily meant for monitoring purposes. The evidence needs to be reported by the system in the form of highly contextual and aggregated observability pillars, and it should let developers ask new questions to troubleshoot the problem.

The table below summarizes the differences between monitoring and observability:

Observability	Monitoring
Designed for debugging, granular insight, and context	Suitable for the overall health status of an application
Conveys the "why" of what is happening and offers data about the system's state through instrumentation	Shows what is happening in the system
Asks questions based on hypotheses	Asks questions based on dashboards
Allows you to make queries about an occurrence or the general state of the system	Provides answers only for known problems or occurrences
A superset of monitoring, both a culture and an outcome	Remains an essential task for DevOps, SREs, and IT operations
Built to tame dynamic environments with changing complexities and unknown permutations	Built to maintain static environments with little variation and known permutations

Building an Observable System

Making a system observable simply means taking note of the pillars of observability mentioned earlier: metrics, logs, and traces.

Anything that occurs within the system or outside of it will provide useful information when analyzed. Compiling all kinds of metrics surrounding the application is a simple way to start making your systems observable. CPU, network, and memory metrics are good places to start.

The logs in a system are also very important. Although logs can quickly grow and become expensive to store, they are essential in ensuring a system's observability. Luckily, some tools increase the effectiveness of logging. An example is OpenTelemetry, which is used not only for logging, but for metric collation and tracing. OpenTelemetry also integrates with popular frameworks and libraries, such as Spring, ASP.NET Core, and Express.

Tracing improves the overall effectiveness of the observable system and allows you to identify the root cause of an issue in a distributed system. Tracing can be seen as the most important part of observability implementation: understanding the causal relationship in your microservice and being able to follow the issue from the effect to the cause, and vice versa.

CI/CD in Observability

CI/CD means continuous integration and continuous delivery. It's a development practice that automates building, testing, and deployment of applications. Having a reliable and consistent CI/CD pipeline enables you to build, test, and deploy changes to production more frequently. Since automation is an essential element of an efficient CI/CD pipeline, it makes a great deal of sense to automate observability.

Continuous observability enables you to stay on top of any risks or problems with your pipeline. Throughout the software development lifecycle, it also provides visibility across the entire CI/CD pipeline and your infrastructure, giving you information on the health of the environment at any time.

One of the most important shifts in DevOps is the focus on actively getting feedback as fast as possible. Traditional application development may shift observability to the time an app is actually running in a production environment, but it doesn't have to. Having observability at build-time ensures that you get fast feedback.

Oscar Medina puts it beautifully in this blog post:

> "Modern applications require modern instrumentation, not only once they are deployed; even at build-time having proper instrumentation can help you gain insights into what is happening at various stages of the build and release process which may include spotting any latency issues, performance and dependency download times.

That's why observability should be baked into your deployment pipeline.

Tools for Observability

To manage distributed system infrastructures, you'll need a dedicated set of tools to visualize the operational states of the system and notify you when a failure occurs. These tools allow you to understand system behaviors and prevent future system problems. Below are some tools you can use to instrument your apps:

1 OpenTelemetry

OpenTelemetry is a project of the Cloud Native Computing Foundation that provides a set of agents, libraries, collector services, and APIs to capture distributed metrics and traces from your application.

OpenTelemetry was formed by merging the OpenCensus and OpenTracing projects to provide a unified set of vendor-agnostic APIs/libraries to send and collect data to compatible backends.

OpenTelemetry provides a standard data format for metrics and distributed trace data and makes vendor-specific integrations unnecessary. Auto-instrumentation and language-specific SDKs for common frameworks and languages also allow you to instrument your code and start collecting observability data.

OpenTelemetry provides the following benefits:

Cross-platform: OpenTelemetry supports several backends and languages. It's a vendor-neutral path for capturing and sending telemetry to backends without affecting the existing instrumentation.

Simplified choice: OpenTelemetry offers backward compatibility for OpenCensus and OpenTracing.

2 Jaeger

Jaeger is a distributed tracing platform used to monitor and troubleshoot complex microservices environments. Jaeger was developed by ride-sharing company Uber in 2015 and adopted as a Cloud Native Computing Foundation incubation project in 2017.

Jaeger includes tools to optimize latency and performance, monitor distributed transactions, and perform root cause analysis. It's open-source, and it benefits from a community of contributors. Jaeger backend is designed to scale with business needs and to have no single point of failure.

Jaeger also has several repositories that provide instrumentation for various frameworks, such as Dropwizard, Django, and the Go standard library. Jaeger supports structured logs and strongly typed span tags, and it represents traces as directed acyclic graphs via span references.

3 Zipkin

Zipkin is a distributed tracing system, based on Google's Dapper, which gathers information about performance indicators/measurements and user request flows. It's a Java-based app used for identifying latency issues and distributed tracing.

Zipkin enables you to avoid vendor lock-in when collecting and identifying data for distributed applications. Initially developed by Twitter, Zipkin has an OpenTracing-compatible API, and it supports nearly all of today's programming languages and development platforms.

Zipkin supports both Elasticsearch and Cassandra, and you can use either of them as a scalable storage backend. It also supports major cloud providers: Google Cloud, Azure, and AWS.

Observe and Improve

While we've seen some open-source tools for observability, there are also a number of tools that can help in instrumenting your application. A good example is Thundra, which provides an automated plug-play solution for observability but keeps the door open for manual instrumentation compatible with OpenTracing–and soon OpenTelemetry.

About Thundra

Thundra is an enterprise SaaS company providing the industry's first Application Observability and Security Platform™ for serverless-centric, container, and virtual machine workloads. Application teams spanning software development, DevOps/SRE, IT operations, and IT security rely on Thundra to run fast safely, troubleshooting and debugging with improved MTTR while ensuring security and compliance policies are enforced. Thundra is committed to making the lives of enterprise IT professionals better by reducing the complexity, costs, and bottlenecks slowing teams down, leveraging Thundra's unique technology footprint to replace numerous existing enterprise tools while improving productivity and efficiency.

Want to see Thundra in action?

demo.thundra.io

Any questions or inquiries?

Contact us at support@thundra.io

© 2020 Thundra, Inc. All Rights Reserved.

THUNDRA™