**19. Observability**

Shivakumar R Goniwada[**1**](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml#Aff2)

(1)

Bangalore, India

Observability has many names, such as monitoring, tracing, logging, telemetry, and instrumenting. All of these are required to create observability. It includes measuring your infrastructure, security, and application to understand how they are doing and then acting on the findings, with either predictive or reactive solutions. Why do you need to do observability? What do you measure? To get an answer to these questions, you need to ask yourself the following:

* Is my user happy?
* Are my applications behaving as expected?
* Are my servers performing well?
* Is my deployment safe from vulnerabilities and fraud?

You have been doing monitoring, tracing, etc. for your system, but without any insight. Current cloud native modern architecture requires more than just monitoring. For example, your system might require processing in real time to enhance the experience, including not just monitoring but examining its internal state based on the outputs.

According to Gartner, by 2024, 30 percent of enterprises implementing distributed cloud native systems will have adopted observability techniques to improve digital business service performance.

Observability is evolving. There are no full-on tools or software available yet, but a few tools such as Splunk and New Relic incorporate some observability features.

Observability is a must for every system; you should not deploy any systems whether small or complex without observability.

In this chapter, I will cover the following:

* Observability in cloud native systems
* Best practices and principles of observability
* What to measure and what not to measure
* Observability use cases

**Introduction**

Organizations have deployed monitoring tools for a decade to track the performance of their infrastructure, network, security, and applications. As the IT landscape evolves, monitoring tools have some limitations in their ability to adapt to the disruption of business and technologies in the cloud native age.

*Observability* is a measure of how well the internal state of the services can be observable from knowledge of external outputs. The concept of observability was introduced by Rudolf E. Kalman for linear dynamic systems. A dynamic system designed to estimate the state of a system from measurements of the outputs *observes* the system.

Once you have successfully deployed your services into production, you have completed half the work. You just need to build observability into your system. Your primary goal is to build systems that are designed to discover problems early and often so you can learn and improve.

As your system grows with more services, each part can start grinding together like poorly fit gears. Intentions conflict, assumptions unravel, and the system begins to operate in increasingly unexpected ways. Rather than spending intellectual capital trying to predict all the possible failure modes, you have learned to use practices that allow you to see deeply into the system, detect anomalies, run experiments, and respond to failures. Observability is required when it becomes difficult to predict the behavior of a system and how your users will be impacted by these changes and to ensure that your system behavior aligns with customer experiences.

Making the system observable involves the practice of combining context, information, and specific knowledge about the system to create the conditions for understanding. You need to integrate all the output generated by the systems such as logs, metrics, events, traces, audits, etc., and correlate them with semantics and intent.

Observability is most important in today’s world when considering the pace and characteristics of cloud native systems and how they are developed, delivered, and deployed. As mentioned in earlier chapters, the adoption of cloud native is increasing every day, so the old practices of bolting on monitoring after the fact are no longer effective and do not scale. It’s critical therefore to have a modern way to observe the behavior of the system to better understand the characteristics of a system. Many practices contribute to observability, and some of the practices are embedded within products and tools.

Observability allows teams to monitor cloud native systems more effectively and helps them to find and connect effects in a complex chain and trace them back to their cause. Further, it gives the operations team more visibility into their entire end-to-end architecture.

*“You can’t perform any operation without proper end-to-end visibility.”*

Observability is important in current scenarios because it gives you greater control over complex systems. Distributed cloud native systems have a higher number of interconnections across services and systems, so the number of failures that can occur is too high, and distributed systems constantly are updating due to business and technical disruptions. Therefore, every change can create a new type of failure. In a distributed environment, the understanding current problem is very challenging, because it produces unknown unknowns. The monitoring is able to find only known unknowns, so how do you find unknown unknowns? You can find them only through observability.

**Difference Between Monitoring and Observability**

Monitoring and observability are different concepts, but they work for the same cause, and both depend on each other. Monitoring is an action to perform to increase the observability of your system. Observability is part of the system like one end-to-end functionality.

Monitoring tools collect and analyze the system behavior as data and translate it into actionable insights such as presenting details to the dashboard, alerting, notifying all stakeholders, etc. For example, monitoring technologies, such as application performance monitoring, is able to provide information about whether the system is able to perform against the service level agreement (SLA).

On the other hand, observability is a measure of how well the internal system state and behavior can be inferred from knowledge of its external outputs. It uses the data that monitoring tools produce. The observability of your system depends on how well your monitoring tools generate and correlate data.

Monitoring requires you to know what’s important to monitor in advance, but observability lets you determine what is important by observing how the system behaves over time and by asking relevant details about it.

Let’s look at one example of a large enterprise where they deployed it in a large data center with a high number of VMs and containers with hundreds of systems that are monitored using log analysis and monitoring with ITSM tools. Analyzing hundreds of systems continually will generate a huge volume of data with unnecessary alerts and false flags. The infrastructure may present with low observability characteristics unless the correct metrics are evaluated. On the other hand, a small system with few containers and servers can be easily monitored using metrics and parameters like health, CPU, etc. These parameters are highly correlated to the health of the system, so the system demonstrates high observability.

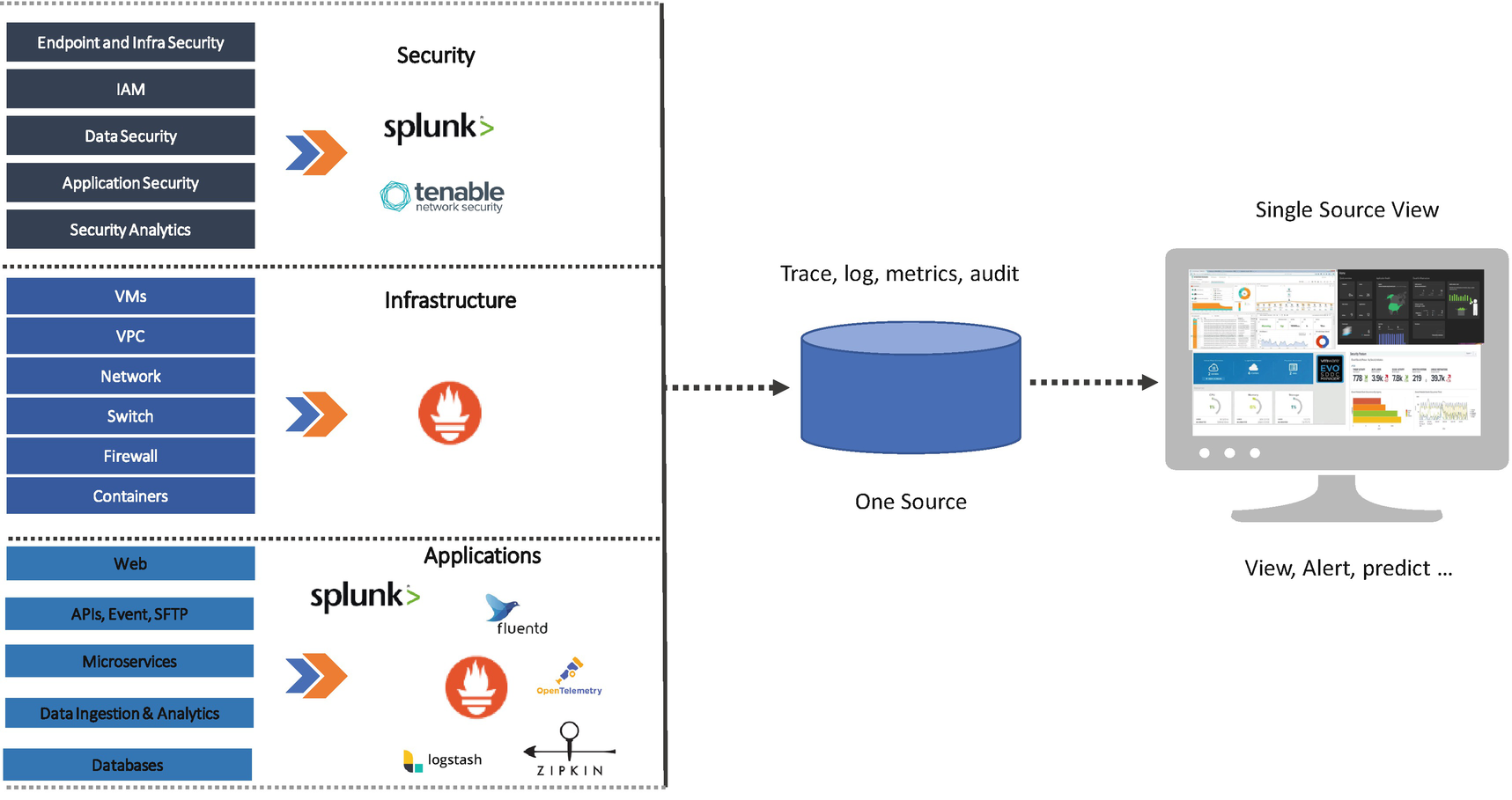
**Full-Stack Observability**

Observability has become an important practice for cloud native modern enterprises. The observability is one of the main strategy that you need to adopt in your architecture that allow you to enable various designs for the “-ilities. ” .

From web and mobile applications to polyglot persistence with varied container resources and integrated systems, there are multiple technologies in a varied infrastructure. To observe the diverse system, you should apply the principles of full-stack observability in your enterprise estate.

With the evolution of unicorn companies, peer competition, technology, and business disruption, you have pressure to innovate quickly and push new features to market faster to capture the market and meet the end customer expectations. Customers are impatient. They want more and do not tolerate slow, error-prone, or poorly designed user experiences. Once you have lost customers, they never come back to you. According to a survey, 62 percent of customers want more user-friendly apps.

To achieve this, as shown in Figure [19-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml#Fig1), you need to observe the end-to-end real estate with an eye on capturing data in a single source. This helps you to troubleshoot, analyze, debug, and optimize performance across your systems. Implementing observability provides you with a connected context and surfaces meaningful analytics from logs and can provide a single view to the administrators and management.



***Figure 19-1***

Full-stack observability

To achieve full-stack observability, you need the following core elements in your observability architecture.

**Connected Across Capabilities**

Putting telemetry data across security, infrastructure, and applications in one place is important. Your data needs to connect the capabilities with the services within the capabilities, and these relationships must be correlated with the metadata so you can understand its relationship. Such connections give your data context and meaning. When all of your telemetry data and connections are stored in one place, you can apply intelligence to your large data set, anomalies, surface pattern, and the correlations that are not easily identifiable by humans watching dashboards.

You need to see how all capabilities and services in your system are related to one another at any moment. It is difficult to maintain the mapping due to changes every day like adding features to existing services, adding new users, or adding a new network or new infrastructure. The context of your data relies on metadata and dimensions. Depending on the type of your system, the volume of data varies.

**One Source of Truth**

For many years, until recently, teams collected telemetry data for observability through agents. Application teams, infrastructure teams, and security teams deploy the agents inside applications; use hosts to collect metrics, event traces, and log data; and aggregate this data and show it in the user interface.

But in the cloud native age, there are many sources of telemetry, and many open systems have their built-in metrics, events, logs, and traces. For full-stack observability, you need to collect data from all capabilities and services and store it in one source and apply instrumentation wherever it requires based on the visibility requirement. The metrics, logs, events, traces, and audits are the essential subtypes of observability.

* *Metrics*: Metrics are the most important subtype telemetry of observability, and they are easy to collect and store quickly by using various tools. These types provide clear visibility of the overall health of your system.
* *Events*: Events are a critical subtype of telemetry of observability. The events are detailed records of every action of your system including integration points, Kubernetes clusters, and security integrations.
* *Logs*: Logs are the detailed subtype telemetry of observability; they provide high fidelity data and detailed context around an event. There are various tools available in the industry for collecting, filtering, and exporting logs; you need to hook these into your capabilities and services.
* *Traces*: Traces are valuable for showing end-to-end latency and detailed subtype telemetry of observability. They provide detailed insight into the myriad customer journeys through a system. This enables you to understand the end-to-end journeys with a unit of work and find bottlenecks with errors. There are various industry tools available to collect traces.
* *Audit*: An audit is a detailed view of the transactions subtype telemetry of observability. These details are important in identifying how and where transactions happen and in providing the various compliance issues.

The following are the characteristics of one source of truth:

* Gathers all your telemetry data in one place and generates a connected view of all the data points of your system. This helps you to understand and resolve the issues that impact your business.
* You should build on a flexible schema so you can quickly get an answer to questions; we recommend using either NoSQL or search databases.
* It scales as your business grows, so you must able to support unpredictable demands.

**Visualization**

Visualization of your connected and well-defined data from system components is very important to view what is going on in your system. When you provide a visualization with insight without requiring configuration, you are better prepared to break down silos and enable stakeholders to observe the entire system as a whole. This helps to identify bottlenecks and resolve faster, and you are able to communicate better across teams and stakeholders.

As part of the full-stack observability, you must provide intuitive, real-time visualizations that focus your attention where it is needed most and communicate the severity and scale of recent changes of your system. This allows you to discover an unknown relationship with blind spots. These views should be customizable so you can accommodate any type of anomaly.

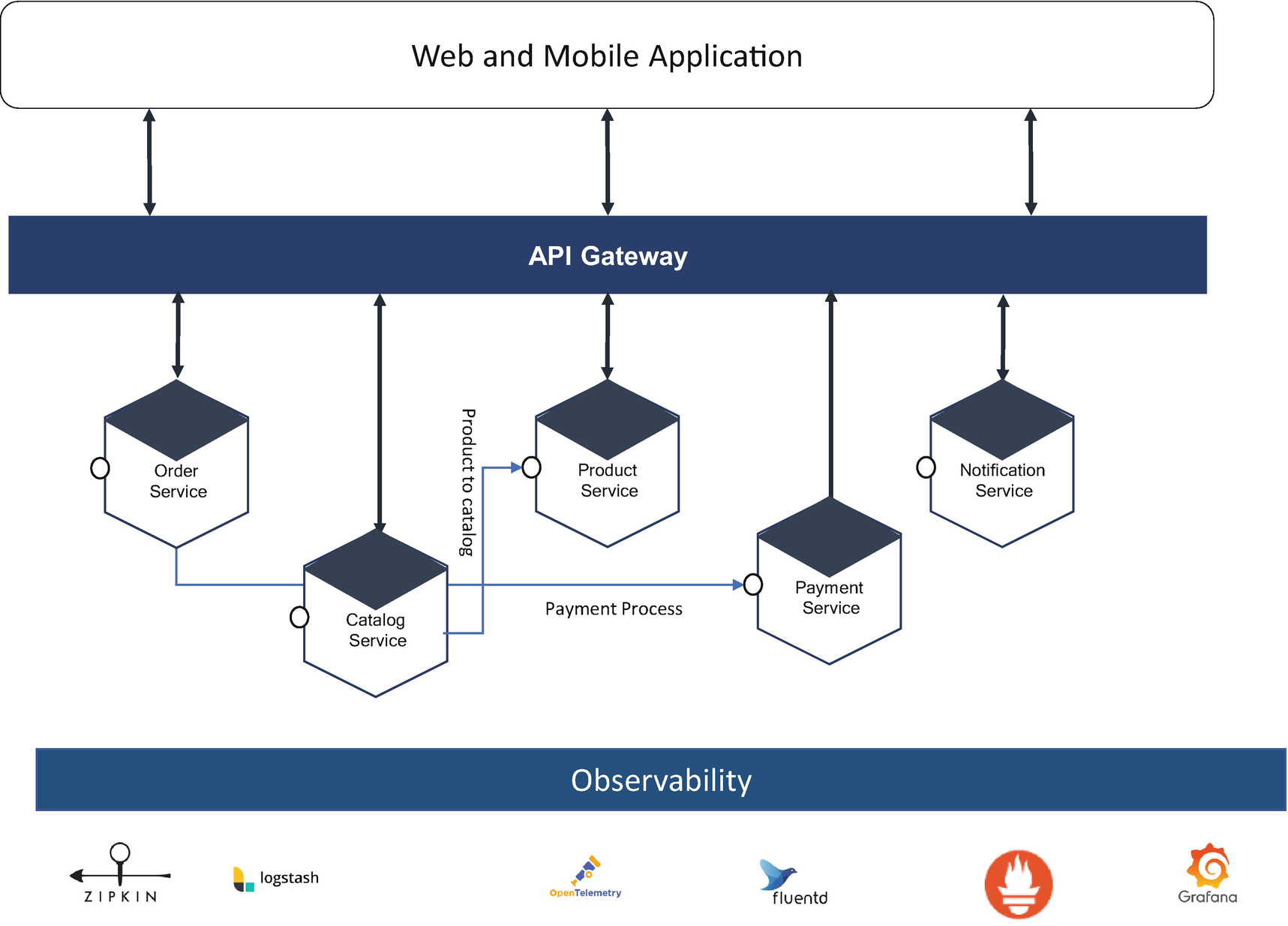
The full-stack observability helps you to reduce the meantime to resolution (MTTR), detect end-to-end issues, and predict the issues and execute fitness functions across your system.

**Observability and Cloud Native Services**

Cloud native services in a distributed system are fundamentally changing the way the systems are developed and deployed. Traditional monitoring capabilities such as metrics, instrumentation, and alerting are not enough to observe systems; you need much more to supplement these capabilities such as tracing, auditing, etc.

Cloud native services with containers address the increased risk of downtime and other issues related to monolithic applications; you can find more details in Chapter [5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_5_Chapter.xhtml). However, container-based services add more complexity due to being loosely coupled, independently deployed, and being scattered across multiple hosts. This makes it difficult for engineers to know the behavior of what is running in production. Observability addresses these challenges, providing visibility into a distributed system. It helps engineers understand the behavior and then predict it so they can make the services self-healing.

As shown in Figure [19-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml#Fig2), for cloud native services, you need to adopt five observability patterns that help to achieve observability in a distributed system.



***Figure 19-2***

Observability with microservices

**Note**

The architecture in Figure [19-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml#Fig2) is just an example of observability. I am not endorsing any specific tools.

In the sample ecommerce architecture shown in Figure [19-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml#Fig2), the customer browses the catalog, places an order, and confirms the order with a payment. You need to collect the following telemetry data for observability in these ways:

* Instrument code verbosely. Everything flows from this. Wrap every network or service call with a lot of context and as much details as possible.
* Persist all the requests and identify throughout the system lifecycle for every service and every request.
* Emit events including cardinality such as user\_id, order\_id, catalog\_id, and Payment\_id.
* Structure your data and store it in one source.
* Don’t neglect any events and touchpoints.
* Create visualization and alerting mechanism and build analytics.

**Observability in Kubernetes**

Kubernetes is one of the main cloud native tools. The Cloud Native Computing Foundation (CNCF) defined observability/analysis as one of the main elements of the cloud native journey. Caleb Hailey defined the top seven Kubernetes APIs for cloud native observability. These APIs help you to achieve a holistic view of your Kubernetes cluster’s health.

* *Kubernetes Metrics API*: All the Kubernetes metrics are exposed as Prometheus endpoints, so anything that can consume Prometheus metrics collects these metrics. This API provides built-in Prometheus exporters, Kubelet metrics, and Kube-state-metrics.
* *Service APIs*: These APIs are important. Without proper visibility into your services, you are able to get the proper errors. These APIs provide networking configuration, including ingress, endpoint, and service resources; service metadata, spec, and status; service ports; internal and external IP addresses; load balancing and label selector configuration details; and Kubectl describe services.
* *Container API*: The containers will run within pods. Kubernetes APIs are able to provide details of both pods and containers. If you want more drilled-down details of the container, you can use these metrics to find the behavior of your system. These APIs provide pod API resources, information about running containers, and container status and details.
* *Pod API*: Pods are the building blocks of all Kubernetes workloads. Pods are managed by Kubernetes controllers. These APIs provide primary workload API resources; pod metadata, spec, and status; controller references; and read log API.
* *Kubernetes downward API*: These APIs enable pods to expose information about themselves to a container running in the pod. These APIs provide pod configuration directives, an alternative to the service account, etc.
* *Kubernetes events API*: In Kubernetes, events are most important. They will give you information about what is happening inside a cluster or a given namespace. These APIs provide resource state changes, errors, and other system messages.
* *Kubernetes API watchers*: These APIs return lists of pods. These APIs provide change notification, return change management notifications, etc.

**Observability and DevOps**

Observability is more important in DevOps-based software development and the deployment lifecycle. DevOps unites all development stakeholders like developers, QA, infrastructure, and operations into one. The monitoring is not just collecting log data, metrics, and event traces; now the monitoring becomes more observable. The scope of observability encompasses the development process, technologies, and people. This allows the team to understand services’ internal state at any given time and has access to more accurate information about the system. The following are a few key benefits:

* There is better visibility of the services catalog in production.
* Predictive alerting helps to identify issues up front and make services self-healing.
* Engineers can see end-to-end workflows about a particular issue.
* There is better collaboration across teams and services deployed in production.
* With DevOps, observability provides a common data model between development engineers and operation engineers to interpret system state and behavior.

**Common Use Cases for Observability with AIOps**

As described in the previous chapter, AIOps have been the driving force in helping you to adapt to continually disrupting environments and enhance your operational capabilities. AIOps consists of the AI technologies used in IT operations and helps the DevOps and IT Ops teams to enhance your organization’s agility and detect anomalies. The following are the few use cases for observability with AIOps:

* *Cloud native systems*: The cloud native services are required to update regularly. They are required to leverage the AIOps in observability. It helps speed up analytics and predictability.
* *Cloud native transformation*: AIOps features help you to collect data from many resources and give a collective cross-domain overview in observability.
* *Predictability*: AIOps helps you to predict from one source of truth to identify the anomalies quickly.
* *The volume of data*: The volume of data aggregated by tools can be immense, and it will be difficult to understand the data without the aid of AIOps.

**Guidance to Choose Observation Tools**

Regardless of the type of tools you use, whether open source or commercial or in-house, all observability tool should provide the following features:

* *Integrate with existing tools*: IT is not new, and monitoring is not new. Every organization might use some kind of monitoring tool. You can’t just throw away old and create new. This is the desired approach. You need to embrace the reusability principle, so pick the observability tool that integrates or collaborates with existing toolsets.
* *Better usability*: If tools provide excellent observability but will fail if you do not have a proper dashboards, configuration management, etc. so you need to make sure you design better usability for an operation team.
* *Able to provide real-time data*: Your observability tool should be able to provide data in both real time and batch mode and be able to integrate streaming technologies like Kafka.
* *Visualize aggregated data*: The observability tool should surface insights in easily configurable formats and be able to integrate with other tools for dashboards, summaries, etc.
* *Context details*: The observability tool must provide the detailed context of incident such as how the system behavior changed over time, etc.
* *Support machine learning*: The observability toolset must support or have a built-in capability for machine learning models for predictions.
* *Business value*: Observability tools must support metrics important to your business, such as speed, customer experience, stability, etc.
* *Open standards*: Observability tools must support emerging open standards for collection such as Open Telemetry and Open Metrics.

**Benefits of Observability**

Observability enables you to reduce the time it takes to identify the root-cause analysis of anomalies. The following are the few benefits:

* *Improved coverage of distributed cloud native architecture*: Observability emphasis on a collection and analysis of telemetry across all elements of your system.
* *Improving the time to market*: Observability helps you to do analysis of anomalies. This helps with shorter resolution times.
* *Infrastructure and storage optimization*: Observability generates less data compared to monitoring. This helps to optimize your infrastructure and storage requirements.
* *Shift-left observability*: Observability can be integrated into the DevSecOps cycle to identify anomalies in the early lifecycle of your system development.

The main drawback of observability is adoption because of the lack tools maturity. The adoption rate is still 5 percent, but there is an indication of a growing interest because organizations are frustrated with the limitations of monitoring. As observability is evolving, you will be able to find more tools that provide observability features.

The observability tools must include features such as arbitrarily wide structured events, high-cardinality dimensions without the need for indexes or schemas, and shared context propagation between contexts.

**Observability, Monitoring, and Machine Learning Models**

You need to always think of how to avoid failure in the system when you configure observability and monitoring tools. You might have to assume that something happened to one service such as a request by the customer, an internal request, a bad experience, etc. As I mentioned in earlier chapters, you can’t avoid failures, but you need to think of how to optimize the responses to the failures. To achieve this, you are required to follow these methods.

Data collection is important for a machine learning model. The following are the data classes required to capture for the model:

* *Logs*: A log is a text record of an event in your application that happened at a particular time that tells when it occurs and a payload that provides the context. Usually, logs come in three formats. They are plain text, structured, and binary. For the model, you select a structured log, which is easier to query.
* *Metrics*: A metric is a numeric value measured over an interval of time and includes specific attributes such as timestamp, name, KPIs, etc. These are structured by default, which makes them easier to query.
* *Traces*: Traces provide an end-to-end journey of a request through a distributed system. The traces provide important data of requests moving across services.

**Algorithms Help in Observability**

Once you collect the data and store it in one source truth, as shown in Figure [19-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml#Fig1), you need to automate the following:

* Applies AI and machine learning models to data
* Detects anomalies and eliminates noise
* Correlates relevant metrics to anomalies, traces, and log events
* Surfaces incidents with contextual data
* Identifies probable root causes

Clustering and correlating are crucial steps for models, and they require multiple different approaches. A combination of historical pattern matching and real-time identification helps you to identify both recurring and new issues.

**Workflow Steps for ML**

* Aggregate event data, logs, metrics, traces, and changes across your environment including your services and infrastructure.
* Integrate configuration management data to discover the system. This provides context within the monitoring data, which helps to understand interdependencies and relationships.
* Enrich data including parsing, aggregating of data, or combining values in fields to equate to a value in another field. The enrichment optimizes several processes including clustering, diagnostics, etc.
* Entropy is an algorithmically determined numerical value that rates the importance of an event. The higher an alert’s entropy, the more important it is.
* Correlate data across your systems. An alert correlation allows you to see patterns across the systems to ensure your services are behaving well. Correlation algorithms analyze alerts to identify clusters of similarity across services. The correlation helps you to enable faster incident management, problem management, MTTD, and MTTR.
* With root-cause analysis, you can apply supervised ML techniques. It uses alert attributes in combination with feedback to analyze real-time data sets and predict which alerts are most casual.
* Collaboration is the process of operation teams to quickly triage and remediate incidents.
* Create a visual representation to illustrate all these steps.

**Summary**

In a nutshell, observability is an important and useful approach to understand the state of your cloud native systems such as microservices, containers, Kubernetes, and other technologies that have made systems complex. Identifying anomalies and troubleshooting is difficult, but these systems produce a wealth of telemetry data that provide a clear understanding of their behavior. Effective observability provides all the instrumentation and analytics to you.

In this chapter, I covered a brief note about observability. It is a vast topic, but I tried my best to cover the relevant details for you to understand and implement observability in your cloud native architecture.