To gain insight into cloud-native systems means knowing what to observe, more than anything else. It’s not solely about monitoring, alerting, tagging and metrics. Rather, these four capabilities combined allow DevOps engineers to observe scaled-out applications running on containers in an orchestrated environment such as Kubernetes.

Some early adopters of cloud-native technologies refer to observability as the new monitoring. The rising demand for observability is real and it comes from the legitimate need to understand raw data produced by the complex infrastructure and multitude of components that run and comprise cloud-native applications. It will gain prominence as more organizations deploy cloud-native applications. A granular understanding of the underlying microservices architectures will lead organizations to accept and embrace concepts pertaining to observability. The interest will drive demand for deeper capabilities to collect data using time-series databases, which in turn will lead to better observability.
Monitoring has a different meaning today. In the past, developers built the applications. Deployments were done by operations teams who managed the applications in production. The health of services was mainly determined based on customer feedback and hardware metrics such as disk, memory or central processing unit (CPU) usage. In a cloud-native environment, developers are increasingly involved in monitoring and operational tasks. Monitoring tools have emerged for developers who use them, to set up their markers and fine tune application-level metrics to suit their interests. This, in turn, allows them to detect potential performance bottlenecks sooner.

Developers are applying techniques like continuous integration/continuous delivery (CI/CD) to optimize programmable and immutable infrastructure. As new infrastructure increasingly becomes available, so does the demand for DevOps professionals and people with site reliability engineering (SRE) experience. Observability gives engineers the information they need to adapt systems and application architectures to be more stable and resilient. This, in turn, provides a feedback loop to developers which allows for fast iteration and adaptation to changing market conditions and customer needs. Without this data and feedback, developers are flying blind and are more likely to break things. With data in their hands, developers can move faster and with more confidence.

The integration of a graph database in a cloud-native monitoring tool, such as Prometheus, lends such tools some considerable staying power. By capturing data that can be viewed as a graph, in relation to time, such tools allow developers to observe applications with more granular detail. Graph databases will increasingly serve as a way to gain deeper visibility, and that enables any number of monitoring use cases. The outcome is deeper efficiencies in the application architecture. Organizations can begin to construct self-remedying application architectures on diverse infrastructure environments.
In this new cloud-native era, continuous understanding about an infrastructure’s state of health defines how applications are built, deployed and managed. It determines how components are modified automatically in an elastic manner — up or down — depending on the load. It tells the operator how to make a decision about a failover or the rollback of a service. Cloud-native systems, by their very nature, are ephemeral and short-lived. They may fail at any time, triggering new events. They scale fast, requiring new monitoring capabilities to cover a range of situations. Cloud-native monitoring must not treat any specific component independently, but rather focus on the aggregate functions that these components together are supposed to perform.

The topic of observability is fairly new, but highly pertinent. Our authors are software engineers who have studied the new monitoring approaches that are emerging with cloud-native architectures. Ian Crosby, Maarten Hoogendoorn, Thijs Schnitger and Etienne Tremel are experts in application deployment on Kubernetes for Container Solutions, a consulting organization that provides support for clients who are doing cloud migrations. These engineers have deep experience with monitoring using Prometheus, which has become the most popular monitoring tool for Kubernetes, along with Grafana as a visualization dashboard.

Monitoring in a cloud-native environment needs to move beyond checks on the state of a resource; it must consider other factors besides “my HTTP service is responding,” or “my disk is at 60 percent capacity.” The cloud-native monitoring environment must provide insight into how a service’s state is related to the state of other resources. This, in turn, must point to the overall state of the system, which is reflected in error messages that encompass multiple considerations, with statements such as “HTTP service response times are increasing beyond threshold, and we can’t scale up because we have hit CPU resource limits.” It’s these types
of insights that enable the system to change based on informed decisions. These insights define observability and move DevOps teams further along the path toward a true CI/CD feedback loop. Without deeper insights, problems can go unseen. In this cloud-native era with scalability in mind, monitoring is one of many factors that come with the broader practice of observability.

**Observability is About Context**

Cloud native means a lot more than just hosting services on the cloud with unlimited computing capacity. A cloud-native system is composed of applications assembled as microservices. These loosely coupled services run across cloud providers in data centers around the globe. There may be millions of these services running at the same time, producing a hive of interactions. At such scale it becomes impossible to monitor each one individually, let alone their interdependencies and communications.

At scale, context is important. It shows how separate events in a system relate to each other. The understanding of this interrelationship serves as the foundation for building a model that helps determine how and why a system is behaving in a particular manner. It is not just a matter of gathering as much data as possible, but collecting meaningful data that adds to an understanding of the behavior. Visualizing data and metrics, and tracing events as they flow from one component to the next, is now a reality of monitoring microservices environments. If monitoring is about watching the state of the system over time, then observability is more broadly about gaining insight into why a system behaves in a certain way.

Observability stems from control theory, where it serves as a measure of how well internal states of a system can be inferred from knowledge of its
external outputs. For a modern SRE, this means the ability to understand how a system is behaving by looking at the parameters it exposes through metrics and logs. It can be seen as a superset of monitoring.

According to Twitter, one of the pioneering companies in web-scale computing and microservices, there are:

1. Monitoring and metrics.
2. Alerting and visualization.
3. Tracing.
4. Logging.

Collecting, storing and analyzing these new types of application performance data raises new challenges.

**Application Performance Management**

The dynamic nature of cloud-native systems poses new challenges for application performance management (APM). First of all, cloud-native systems are by definition more transient and complex than traditional systems. The components making up the system are no longer static, but ephemeral — appearing on demand and disappearing when they are no longer needed. A sudden increase in demand might lead to certain components being scaled in large numbers. Any APM solution needs to be able to accommodate these rapid and numerous changes.

Components in a cloud-native system also tend to change more often with the increased use of continuous deployment (CD) techniques. This creates the necessity for logging and metrics to be linked not only to the state of the system at a certain point in time, but also the changes in the software leading up to that state. Also, the increased number of
components vastly increases the amount of data and metrics being logged. This increases demand for storage and processing capacity when analyzing these data and metrics. Both of these challenges lead to the use of time-series databases, which are especially equipped to store data that is indexed by timestamps. The use of these databases decreases processing times and this leads to quicker results.

These large amounts of data also allow for gaining insights by applying principles of artificial intelligence and machine learning. These techniques can lead to increased performance, because they allow the system to adapt the way it changes in response to the data it’s collecting by learning from the effect of previous changes. This in turn leads to the rise of predictive analytics, which uses data of past events to make predictions for the future, thereby preventing errors and downtime.

**FIG 4.1:** In cloud-native systems, observability is the new monitoring.
The Four Pillars

Observability, and its resulting insights, comes from logging, metrics, tracing and alerting. The value that comes from these pillars of observability derives from using well-defined terms and clearly identifying the purpose of each pillar. Data is captured from each pillar and used for later evaluation. Let’s take a simple example, a 500 error, and see how DevOps engineers would gain insight through each lens.

Logging

Logging in the simplest sense is about recording discrete events. This is the first form of monitoring which any new developer gets exposed to, usually in the form of print statements. In a modern system, each application or service will log events as they occur, be it to standard out, syslog or a file. A log aggregation system will then centralize all logs to be viewed or searched as needed. In our example of a 500 error occurring, this would be visible by a service, or possibly multiple services, logging an error which resulted in the 500 status code. This error can be deciphered through an evaluation of the other three pillars.

Metrics

By contrast, metrics are a combination of data from measuring multiple events. Cloud-native monitoring tools cater to different types of measurements by having various metrics such as counters, gauges, histograms and meters.

- **Counter:** A counter is a cumulative metric that can only ever increase; for example, requests served, tasks completed and errors occurred. It should not be used for metrics that can also go down, such as number of threads.

- **Gauge:** A gauge is a metric that can arbitrarily go up and down; for
example, temperature, memory usage, and live number of users.

- **Histogram:** Histograms measure the statistical distribution of a set of events; for example, request duration and response size. Histograms track the number of observations and the sum of the observed values, allowing a user to view the average of the observed values.

- **Meter:** Measures the rate at which an event occurs. The rate can be measured over different time intervals. The mean rate spans the lifetime of your application, while one-, five- and fifteen-minute rates are generally more useful.

The idea is to aggregate similar events to gain a higher level of insight. Metrics are generally time based, therefore we usually collect metrics periodically, such as once per second. In our 500 error example, we can see the rate of 500 errors which a particular service is omitting. If we have a consistent rate of 500 errors, this would point to a different problem than a sudden spike of 500s would.

**Tracing**

Tracing is about recording and ordering connected events. All data transactions, or events, are tied together by injecting a unique ID into an initial request, and passing that ID to all further events through the system. In a distributed system, a single call will end up passing through multiple services. Tracing provides a complete picture at the application level. Again, coming back to our example of a 500 error response, we can see the entire flow of the specific request which resulted in a 500. By seeing which services the request passed through we gain valuable context, which will allow us to find the root cause.

**Alerting**

Alerting uses pattern detection mechanisms to discover anomalies that may be potentially problematic. Alerts are made by creating events from
data collected through logging, metrics and tracing. Once engineers have identified an event, or group of events, they can create and modify the alerts according to how potentially problematic they may be. Returning to our example: How do we start the process of debugging the 500 error? Establish thresholds to define what constitutes an alert. In this case, the threshold may be defined by the number of 500 errors over a certain period of time. Ten errors in five minutes means an alert for operations managed by Container Solutions. Alerts are sent to the appropriate team, marking the start of the debugging and resolution process. Take into consideration that what constitutes an alert also depends on what the normal state of the system is intended to be.

By establishing the four data pillars, observability is gained into the system and the cloud-native applications it runs. Complexity will only increase as the system is developed, requiring more observability that comes from collecting more data in a manner that can be stored and analyzed, providing a feedback loop for deeper optimizations and the proper insights into applications.

**Monitoring Patterns**

Of the four pillars, metrics provide the most insight into how an application performs. Without metrics, it is impossible to tell if an application behaves the way it should in order to meet service-level objectives. There are different strategies used to collect and analyze metrics in order to report the health of cloud-native systems, which is the foremost concern.

Blackbox and whitebox monitoring are two different strategies used to report the health of a system. Both rely on different techniques which, when combined, strengthen the reliability of the report.
Blackbox monitoring is a method to determine the state of a system without having access to the application internals. The type of metrics collected provide information about the hardware such as disk, memory and CPU usage or probes — Transmission Control Protocol (TCP), Internet Control Message Protocol (ICMP), Hypertext Transfer Protocol (HTTP), etc. A health check is a typical example of blackbox monitoring. It determines the status of a system by probing different endpoints using a particular protocol such as TCP or ICMP. If a probe is successful then the application is alive, otherwise we can assume that the system is down without knowing the exact cause.

In contrast to blackbox monitoring, whitebox monitoring is more sophisticated and relies on telemetry to collect application behavior metrics, such as the total number of HTTP requests and latencies, or the number of errors or runtimes specific via interfaces, like Java Virtual Machine Profiling Interface (JVMPRI). In order to monitor an application properly, this information must be specified and it’s up to developers to instrument it with the right metrics.

Blackbox and whitebox monitoring are two patterns that complement each other to report the overall health of systems. They play an important role in cloud-native systems where modern SREs interpret these metrics to identify server performance degradation and spot performance bottlenecks early on.

**Performance Metrics and Methodology**

In a cloud-native environment and with complex distributed systems, it takes time and effort to discover what caused a failure. Only a handful of methodologies exist, which are intended to be simple and fast in order to help SREs come to a conclusion. Each method relies on one of the following key metrics:
• **Error**: rate of error events produced.

• **Latency**: duration of a request.

• **Utilization**: how busy the system is.

• **Saturation**: the threshold at which a service cannot process extra work.

• **Throughput**: rate or quantity at which the system is being requested.

From these metrics you can apply one of the following four methodologies to determine how performant the system is:

• **USE** (utilization, saturation and errors): This technique, developed by Brendan Gregg, is a resource-oriented method which is intended to detect resource bottlenecks in a system under load. It relies on three metrics: utilization, saturation and errors.

• **TSA** (thread state analysis): This method is complementary to the USE method. Also developed by Brendan Gregg, it focuses on threads instead of resources and tries to find which state takes the most time. It relies on six key sources of performances issues: executing, runnable, anonymous paging, sleeping, lock and idle.

• **RED** (rate errors duration): This method is aimed at request-driven services. Like TSA, the RED method is complementary to the USE method and relies on three key metrics: rate, errors and duration.

• **Golden signals**: This method was promoted by the Google SRE team and relies on four key metrics to determine the state of a system: latency, throughput, errors and saturation.

In any given system, if the right metrics are collected, engineers — even if they’re not aware of the entire architecture of the system they use — can
apply one of these methodologies to quickly find out which part of a system has the potential to become a performance bottleneck and cause failure.

**Anomaly Detection**

In modern production systems, observability is a core feature which is needed to detect and troubleshoot any kind of failure. It helps teams make decisions on actionable items in order to return the system to its normal state. All the steps taken to resolve a failure should be meticulously recorded and shared through a post-mortem, which can be used later on to speed up the resolution time of recurrent incidents.

Recovery procedures that used to be handled by an operations team are now handled by container orchestrators. When the recovery procedure is more complex, additional tooling can be developed to automate the recovery steps, which bring the system back to its normal state. The decision to recover can be triggered based on metrics and threshold, or some other predictive mechanism such as machine learning.

Implementing a self-healing capability based on recurrent problems is a first step toward making a system resilient. Observations and the resolution described in a post-mortem can be translated into actionable items for future decision-making.

Analytics can tell a lot about the behavior of a system. Based on historical data it is possible to predict a potential trend before it becomes a problem. That’s where machine learning comes into play. Machine learning is a set of algorithms which progressively improves performance on a specific task. It is useful to interpret the characteristics of a system from observed behavior. With enough data, finding patterns that do not conform to a model of “normal” behavior is an advantage, which can be used to reduce false positive alerts and help decide on actions that will
attempt to bring the system back to its normal state.

Supervised and unsupervised are two types of anomaly detection. A model labelled either normal or abnormal is called supervised. It gets its name when a dataset is used to train a model and then subsequently gets labelled. It comes with limitations, since labelling can be difficult and expensive. In contrast, unsupervised detection doesn’t identify faulty datasets. Based on events that rarely occur or don’t repeat, it is possible to classify them as abnormal by using standard inference in Bayesian networks and compute a rank using probability.

Since this can be complex to integrate into a monitoring system, a simpler approach is to make use of triple exponential smoothing, also known as the Holt-Winters method. This has the potential to deliver accurate predictions since it incorporates seasonal fluctuations to predict data points in a series over time.

**FIG 4.2:** The Holt-Winters method has the potential to deliver accurate predictions since it incorporates seasonal fluctuations to predict data points in a series over time.
predictions since it incorporates seasonal fluctuations into the model. It is one of the many methods that can be used to predict data points in a series over time. The following graph provides an overview of what the Holt-Winters method evaluates.

**Push vs. Pull Data Collection**

We can distinguish two models when it comes to gathering metrics from an application: push and pull. Many monitoring solutions expect to be handed data, which is known as the push model. Others reach out to services and scrape data, which is known as the pull model. In both cases, developers need to instrument a specific part of their application in order to measure its performance — time to execute a task, time an external request takes, etc. — and optimize it later on. Depending on the use case, one method may be a better fit than the other.

The push model works best for event-driven, time-series datasets. It’s more accurate, as each event is sent when it’s triggered at the source. With the push model, it takes some time to tell if a service is unhealthy, as the

---

**FIG 4.3:** Many monitoring solutions expect to be handed data, which is known as the push model. Others reach out to services and scrape data, which is known as the pull model.

---

**Two Methods to Collect Data**

- **Push (receive data)**: Multiple requests. Can be triggered in parallel. More accurate, but server needs resource capacity to handle load.

- **Pull (scrape data)**: One request every few seconds.

---

Source: [https://www.slideshare.net/OliverMoser5/prometheus-introduction-infracoders-vienna](https://www.slideshare.net/OliverMoser5/prometheus-introduction-infracoders-vienna)
instance health is based on the event it receives. The push model assumes an instance is unhealthy when it cannot be reached to pull the metrics.

Each serves a different purpose. A pull model is a good fit for most use cases, as it enforces convention by using a standard language, but it does have some limitations. Pulling metrics from internet of things (IoT) devices or browser events requires a lot of effort. Instead, the push model is a better fit for this use case, but requires a fixed configuration to tell the application where to send the data.

In a cloud-native environment, companies tend to favor the pull model over the push model for its simplicity and scalability.

**Monitoring at Scale**

Observability plays an important role in any large distributed system. With the rise of containers and microservices, what happens when you start scraping so many containers that you need to scale out? How can you make it highly available?

There are two ways to solve this problem of monitoring at scale. The first is a technical solution to use a federated monitoring infrastructure. Federation allows a monitoring instance to gather selected metrics from other monitoring instances. The other option is an organizational approach to improve monitoring by adopting a DevOps culture and empowering teams by providing them with their own monitoring tools. This reorganization could be further split into domains — frontend, backend, database, etc. — or product. Splitting can help with isolation and coupling issues that can arise when teams are split by role. By deciding on roles ahead of time, you can prevent scenarios like, “I’m going to ignore that frontend alert because I’m working on the backend at the
moment.” A third option, and the best yet, is a hybrid of both approaches: adopt DevOps and federate some metrics to pull some top-level service level indicators out of the various monitoring instances.

**Federation**

A common approach when having a set of applications running on multiple data centers or air-gapped clusters is to run a single monitoring instance for each data center. Having multiple servers requires a “global” monitoring instance to aggregate all the metrics. This is called hierarchical federation.

Much later, you might grow to the point where your scrapes are too slow because the load on the system is too high. When this happens you can enable sharding. Sharding consists of distributing data across multiple servers in order to spread the load. This is only required when a monitoring instance is handling thousands of instances. In general, it is recommended to avoid this as it adds complication to the monitoring system.

**High Availability**

High availability (HA) is a distributed setup which allows for the failure of one or more services while keeping the service up and running at all times. Some monitoring systems, like Prometheus, can be made highly available by running two monitoring instances simultaneously. It scrapes targets and stores metrics in a database. If one goes down, the other is still available to scrape.

Alerting can be difficult on a highly available system, however. DevOps engineers must provide some logic to prevent an alert from being fired twice. Displaying a dashboard can also be tricky since you need a load balancer to send traffic to the appropriate instance if one goes down. Then there is a risk of showing slightly different data due to the fact that each
instance might collect data at a different time. Enabling “sticky session” on the load balancer can prevent such flickering of unsynchronised time series to be displayed on a dashboard.

**Prometheus for Cloud-Native Monitoring**

Businesses are increasingly turning to microservices-based systems to optimize application infrastructure. When done at scale, this means having a granular understanding of the data to improve observability. Applications running microservices are complex due to the interconnected nature of Kubernetes architectures. Microservices require monitoring, tracing and logging to better measure overall infrastructure performance, and require a deeper understanding of the raw data. Traditional monitoring tools are better suited to legacy applications that are monitored through instrumentation of configured nodes. Applications running on microservices are built with components that run on containers in immutable infrastructure. It requires translating complicated software into complex systems. The complexity in the service-level domain means that traditional monitoring systems are no longer capable of ensuring reliable operations.

Prometheus is a simple, but effective, open source solution to that problem. At its heart, it is a time-series database, but the key feature lies in its use of a pull model. It scrapes and pulls metrics from services. This alone makes it robust, simple and scalable, which fits perfectly with a microservices architecture. Originally developed by SoundCloud for internal use, Prometheus is a distributed monitoring tool based on the ideas around Google’s Borgmon, which uses time-series data and metrics to give administrators insights into how their operations are performing. It became the second project adopted by the Cloud Native Computing Foundation (CNCF) after Kubernetes, which allows for some beneficial
Monitor as a Service, Not as a Machine

**FIG 4.4:** Representation of Prometheus in a hierarchical, federated architecture.

coordination between the projects’ communities.

Key features of Prometheus are:

- **Simplicity.**
- **Pulls data** from services, services don’t push to Prometheus.
- **No reliance** on distributed storage.
- **No complex scalability** problems.
- **Discovers targets** via service discovery or static configuration.
- **Powerful query language** called PromQL.

Prometheus works well in a microservices architecture. It handles multi-dimensional data simply and efficiently. It is also a good fit for mission-
critical systems. When other parts of your system are down, Prometheus will still be running.

Prometheus also has some drawbacks: Accuracy is one of them. Prometheus scrapes data and such scrapes are not guaranteed to occur. If you have services that require accuracy, such as per-usage billing, then Prometheus is not a good fit. It also doesn’t work well for non-HTTP systems. HTTP is the dominant encoding for Prometheus, so if you don’t use HTTP, and instead use Google remote protocol procedure (gRPC), for example, you will need to add some code to expose the metrics (see go-grpc-prometheus).

**Alternatives to Prometheus**

Grafana and Prometheus are the preferred monitoring tools among Kubernetes users, according to the [CNCF’s fall 2017 community survey](https://kubernetes.io). The open source data visualization tool Grafana is used by 64 percent of organizations that manage containers with Kubernetes, and Prometheus follows closely behind at 59 percent. The two tools are complementary and the user data shows that they are most often employed together: Some 67 percent of Grafana users also use Prometheus, and 75 percent of Prometheus users also use Grafana.

Kubernetes users often use more than one monitoring tool simultaneously, due to varying degrees of overlapping functionality, according to the CNCF survey. Grafana and Graphite are primarily visualization tools, for example. And Prometheus can be set up to provide functionality similar to a time-series database, but it doesn’t necessarily replace the need for one. Among Prometheus-using Kubernetes shops, InfluxDB’s adoption rate increases slightly, at the same time OpenTSDB’s use drops several percentage points. CNCF did not ask about many monitoring vendors’ offerings, such as Nagios and New Relic. However, 20 percent of all the respondents providing an “other” answer mentioned
**Grafana and Prometheus Are the Most Widely Used Tools for Monitoring Among Kubernetes Users**

![Bar chart showing monitoring tool usage percentages](chart.png)

- **Grafana**: 64%
- **Prometheus**: 59%
- **InfluxDB**: 29%
- **Datadog**: 22%
- **Graphite**: 17%
- **Other**: 14%
- **Sysdig**: 12%
- **OpenTSDB**: 10%
- **Stackdriver**: 8%
- **Weaveworks**: 5%
- **Hawkular**: 5%

**Source**: The New Stack Analysis of Cloud Native Computing Foundation survey conducted in Fall 2017.

Q. What monitoring tools are you currently using? Please select all that apply. English n=489; Mandarin, n=187.

Note, only respondents managing containers with Kubernetes were included in the chart.

**FIG 4.5**: Grafana and Prometheus are the most commonly used monitoring tools, with InfluxDB coming in third.

New Relic. (See the second ebook in this series, Kubernetes Deployment & Security Patterns, for a more detailed analysis.)

Based on our experience at Container Solutions, here’s our take on some of the Prometheus alternatives:

- **Graphite** is a time-series database, not an out-of-the-box monitoring solution. It is common to only store aggregates, not raw time-series data, and has expectations for time of arrival that don’t fit well in a microservices environment.

- **InfluxDB** is quite similar to Prometheus, but it comes with a commercial option for scaling and clustering. It is better at event logging and more complex than Prometheus.

- **Nagios** is a host-based, out-of-the-box monitoring solution. Each host
can have one or more services and each service can perform one check. It has no notion of labels or query language. Unfortunately, it’s not really suited towards microservices since it uses a form of blackbox monitoring which can be expensive when used at scale.

- **New Relic** is focused on the business side and has probably better features than Nagios. Most features can be replicated with open source equivalents, but New Relic is a paid product and has more functionality than Prometheus alone can offer.

- **OpenTSDB** is based on Hadoop and HBase, which means it gains complexity on distributed systems, but can be an option if the infrastructure used for monitoring already runs on an Hadoop-based system. Like Graphite, it is limited to a time-series database. It’s not an out-of-the-box monitoring solution.

- **Stackdriver** is Google’s logging and monitoring solution, integrated with Google Cloud. It provides a similar feature set to Prometheus, but provided as a managed service. It is a paid product — although Google does offer a basic, free tier.

**Components and Architecture Overview**

The Prometheus ecosystem consists of multiple components, some of which are optional. At its core, the server reaches out to services and scrapes data through a telemetry endpoint, using the aforementioned pull model.

Basic features offered by Prometheus itself include:

- **Scrapes metrics** from instrumented applications, either directly or via an intermediary push gateway.

- **Stores data**.
Prometheus Ecosystem Components

FIG 4.6: Components outside of the Prometheus core provide complementary features to scrape, aggregate and visualize data, or generate an alert.

- **Aggregates data** and runs rules to generate a new time series or generate an alert.

- **Visualizes and acts upon the data** via application programming interface (API) consumers.

Other components provide complementary features. These include:

- **Pushgateway**: Supports short-lived jobs. This is used as a workaround to have applications push metrics instead of being pulled for metrics. Some examples are events from IOT devices, frontend applications sending browser metrics, etc.

- **Alertmanager**: Handles alerts.

- **Exporters**: Translate non-compatible Prometheus metrics into
compatible format. Some examples are Nginx, RabbitMQ, system metrics, etc.

- **Grafana:** Analytics dashboards to complement the Prometheus expression browser, which is limited.

**Prometheus Concepts**
Prometheus is a service especially well designed for containers, and it provides perspective about the data intensiveness of this new, cloud-native age. Even internet-scale companies have had to adapt their monitoring tools and practices to handle the vast amounts of data generated and processed by these systems. Running at such scale creates the need to understand the dimensions of the data, scale the data, have a query language and make it all manageable to prevent servers from becoming overloaded and allow for increased observability and continuous improvement.

**Data Model**
Prometheus stores all of the data it collects as a time series which represents a discrete measurement, or metric, with a timestamp. Each time series is uniquely identified by a metric name and a set of key-value pairs, aka labels.

By identifying streams of data as key-value pairs, Prometheus aggregates and filters specified metrics, while allowing for finely-grained querying to take place. Its functional expression language, called PromQL, allows users to select and aggregate time-series data in real time using the Prometheus user interface (UI). Other services, such as Grafana, use the Prometheus HTTP API to fetch data to be displayed in dashboards.

Its mature, extensible data model allows users to attach arbitrary key-value dimensions to each time series, and the associated query language allows you to do aggregation and slicing and dicing. This
support for multi-dimensional data collection and querying is billed as a particular strength, though not the best choice for uses such as per-request billing.

One common use case of Prometheus is to broadcast an alert when certain queries pass a threshold. SREs can achieve this by defining alerting rules, which are then evaluated at regular intervals. By default, Prometheus processes these alerts every minute, but this can be adjusted by changing the Prometheus configuration key to: `global.evaluation_interval`.

Whenever the alert expression results in one or more vector elements at a given point in time, Prometheus notifies a tool called Alertmanager.

Alertmanager is a small project that has three main responsibilities:

- Storing, aggregating and de-duplicating alerts.
- Inhibiting and silencing alerts.
- Pushing and routing alerts out to external sources.

With Alertmanager, notifications can be grouped — by team, tier, etc. — and dispatched amongst receivers: Slack, email, PagerDuty, WebHook, etc.

**Prometheus Optimization**

If used intensively, a Prometheus server can quickly be overloaded depending on the amount of rules to evaluate or queries run against the server. This happens when running it at scale, when many teams make use of query-heavy dashboards. There are a few ways to leverage the load on the server, however. The first step is to set up recording rules.

Recording rules pre-compute frequently needed or computationally expensive expressions and save the result as a new set of time series, which is useful for dashboards.
Instead of running a single big Prometheus server which requires a lot of memory and CPU, a common setup adopted by companies running e-commerce websites is to provide one Prometheus server with little memory and CPU per product team — search, checkout, payment, etc. — where each instance scrapes its own set of applications. Such a setup can easily be transformed into a hierarchical federation architecture, where a global Prometheus instance is used to scrape all the other Prometheus instances and absorb the load of query-heavy dashboards used by the business, without impacting the performance of the primary scrapers.

**Installing Prometheus**

Installing Prometheus and its components is really simple. Each component is a binary which can be installed on any popular operating system, such as Unix and Windows. The most common way to install Prometheus is to use Docker. The official image can be pulled from Docker Hub prom/prometheus. A step-by-step guide to install Prometheus is available on the Prometheus website.

In a cloud-native infrastructure there is a concept called Operators which was introduced by CoreOS in 2016. An Operator is an application which has the capability to set up, upgrade and recover applications in order to reduce the heavy scripting or manual repetitive tasks — usually defined by site reliability engineers — to make it work. In Kubernetes, Operators extend the Kubernetes API through a CustomResourceDefinition, which lets users easily create, configure and manage complex applications.

The Prometheus Operator — also developed by the CoreOS team — makes the Prometheus configuration Kubernetes native. It manages and operates Prometheus and the AlertManager cluster. A complementary tool, called Kube Prometheus, is used on top of the Prometheus Operator to help get started with monitoring Kubernetes. It contains a collection of
manifests — Node Exporter, Kube State Metrics, Grafana, etc. — and scripts to deploy the entire stack with a single command. Instructions to install the Prometheus Operator are available on the project repository.

Conclusion

Cloud-native systems are composed of small, independent services intended to maximize resilience through predictable behaviors. Running containers in a public cloud infrastructure and taking advantage of a container orchestrator to automate some of the operational routine is just the first step toward becoming cloud native.

Systems have evolved, and bring new challenges that are more complex than decades ago. Observability — which implies monitoring, logging, tracing and alerting — plays an important role in overcoming the challenges that arise with new cloud-native architectures, and shouldn't be ignored. Regardless of the monitoring solution you ultimately invest in, it needs to have the characteristics of a cloud-native monitoring system which enables observability and scalability, as well as standard monitoring practices.

Adopting the cloud-native attitude is a cultural change which involves a lot of effort and engineering challenges. By using the right tools and methodology to tackle these challenges, your organization will achieve its business goals with improved efficiency, faster release cycles and continuous improvement through feedback and monitoring.