Giant Swarm

A Guide to Your Cloud-Native Stack

Table of Contents

Cloud computing has been with us for well over a decade now, and although its adoption has been steadily rising over that time, there are many reasons why the rate of adoption hasn't been swifter. Chief among them, might be that organizations don't necessarily believe that a 'lift and shift' approach to workload migration provides much benefit.

Organizations want to transform the way they operate; to be more agile and responsive, and simply replicating a data center in the cloud is not going to bring about that transformation.

Forward thinking organizations are beginning to make this transformation, by embracing the cloud-native approach. This is not just about moving software applications to run on cloud infrastructure, but it is also about building applications using the microservices architecture. This makes those applications more resilient and scalable, and more capable of benefiting from the on-demand, elastic nature of the cloud.

The cloud-native paradigm is not all sweetness and light, however; in exchange for the considerable benefits it affords, there is a price to pay in the form of increased complexity.

Application workloads tend to be highly distributed and often transient in nature, which makes them all the harder to manage in a production setting. This hasn't deterred the proponents of the cloud-native approach though, and a lot of effort has been put into the creation of tools, methods, and standards to facilitate running these workloads, and to lower the bar to cloud-native adoption. We call this the 'cloud-native stack'.

Defining the cloud-native stack is a bit like trying to nail jelly to the wall! New companies, projects, tools, and approaches are appearing almost on a monthly basis. Trying to keep pace with the change is both challenging and time consuming. But, some things are settled, and have become a fact of life. This guide discusses the nature of the cloud-native stack, and the choices that make up its composition.

This guide discusses the nature of the cloud-native stack, and the choices that make up its composition.

Orchestration Systems

As the cloud-native approach has transitioned from a hype, to a level of maturity that has encouraged organizations to trust their business with, so too has the technology that characterizes it. Not least, the orchestration element of the stack that hosts cloud-native applications.

In the beginning, with the advent of the Docker container format, there wasn't an easy way to run containerized microservices at scale. The Docker Engine was perfect for local development but trying to wrestle with the intricacies of managing distributed workloads, was beyond its capabilities. As a basis for hosting cloud-native applications as distributed workloads, a system was required that allowed applications to scale easily and quickly. It also needed to provide resilience in the event of inevitable failures, support advanced deployment patterns, and facilitate automation for the delivery of frequent application updates.

Though there weren't any open source, purpose-built orchestration systems to provide these requirements early on, it wasn't long before a number of new projects materialized to fill the void.

This list is not exhaustive, but does represent the contenders that have received the most interest and largest contributions by the open source community. They are also the most widely adopted.

Docker Swarm

The Docker project introduced 'Swarm Classic' toward the end of 2015, but quickly replaced it with a more functional 'in-Engine' version called 'Swarm Mode' in the middle of 2016.

Apache Mesos

A platform for pooling the physical or virtual resources of a datacenter, Apache Mesos can be used in conjunction with 'frameworks' (e.g. Aurora, Marathon) for providing the mechanisms for running distributed workloads.

Kubernetes

Borrowing and building on the concepts inherent in Google's proprietary Borg cluster manager, Kubernetes is a very popular open source project supported by many influential organizations, such as Google, Microsoft and IBM.

Orchestration Systems

In a relatively short time, Kubernetes has emerged as the de-facto choice for running cloud-native applications. There may be many reasons for this, but prime among them is as a result of the many different organizations that have coalesced around Kubernetes, turning it into a well respected, open, community-driven project.

That's not to say that Swarm doesn't continue to play a part in the ecosystem; Swarm has a very low entry bar into the orchestration of cloud-native applications, and is still a foundational component of Docker Enterprise Edition. You'll also find that DC/OS, a distributed operating system based on Mesos, can itself host Kubernetes clusters - Kubernetes on Mesos, if you will.

If Kubernetes has won the battle for the hearts and minds of engineering teams as the de facto standard choice for hosting cloud-native applications, then we might be excused for thinking that we're limited for choice. Whilst it's true that adopting and investing in the Kubernetes platform funnels you into a particular technology, it doesn't lock you into a particular vendor, unless you rely heavily on the value-add extras that particular vendors provide.

In some ways, it's like choosing Linux as your preferred host operating system (OS) kernel, and then electing to adopt Ubuntu as your preferred OS distribution. In the same way, because Kubernetes is open source, there is a wide choice available. Currently, there are over 50 different Kubernetes distributions from a myriad of vendors, each of which has been independently certified by the Cloud-native Computing Foundation (CNCF) as part of its Certified Kubernetes Conformance Program.

One other factor to consider when determining orchestration system choice, is complexity. There is no doubt that Kubernetes (and the other competing platforms) is technically complex, and requires significant knowledge and skill to operate. Many organizations prefer to focus on developing the value that characterizes their business or cause, and to leave the task of running infrastructure and operations, to third-parties (at least in part).

> If DIY is your preferred approach, be prepared to make a considerable investment in terms of engineering skills and know-how.

Enhancing the Stack

Systems like Kubernetes provide a significant level of automation that makes orchestrating cloud-native applications possible. These systems can't be everything to everybody though, and at some point there has to be some delineation in terms of purpose. Otherwise, we end up creating a functional monster and we start to become more opinionated, with choice becoming the inevitable casualty.

To successfully operate a cloud-native stack, it's necessary to augment the orchestration system with capabilities that enable us to gain insight into the health of the stack. We need to know when individual services are ailing, or whether there is unacceptable latency between services, or a predetermined threshold of failed requests has been breached, and so on.

And we need to know about this health at all levels in the stack; right from the host machines, up to the application services themselves. Tools that enable us to observe what's going on at every level of the stack, help us to maintain service levels, as well as gain valuable insight into application behavior. The insight we gain, subsequently informs future development decisions.

Taking a peek under the hood of our applications running in production is no easy task, but there are some techniques and tools that can enhance our cloud-native stack. This allows us to begin to crack this difficult problem.

Monitoring

Monitoring is an operational practice that has been an essential ingredient for managing the availability of software applications for a number of years. It's a tried and tested technique for observing the state of health of a complete system and has a mature set of tools that pre-date the cloud-native era. A lot of these tools are proprietary but some have been developed using the open source software model.

Generally, these tools are agent-based and collect metrics from the host operating system and the monolithic applications that run on top of them, using 'probes'. Their architecture is based on a different era in computing, and is not entirely suited to the elastic, on-demand nature of cloud computing environments.

In fact, their inadequacies are exposed further when we toss cloud-native applications into the mix, which are composed of multiple loosely-coupled, replicated services that have an ephemeral existence. Pinning down what to monitor and where, is a significant problem. Consequently, as the cloud-native movement has gathered steam, different approaches have been sought to reinvent monitoring for the cloud-native era.

A lot of the legacy monitoring tools rely on a 'black box' approach to monitoring by probing an application to determine whether it responds as expected. This provides limited insight regarding the state of an application, however, and in cloud-native stacks it's more common to see monitoring tools that provide introspection of applications.

This is often termed 'white box' monitoring and is characterized by the collection of metrics (including those obtained from instrumented applications), which are stored in a time-series database for subsequent analysis and alerting purposes.

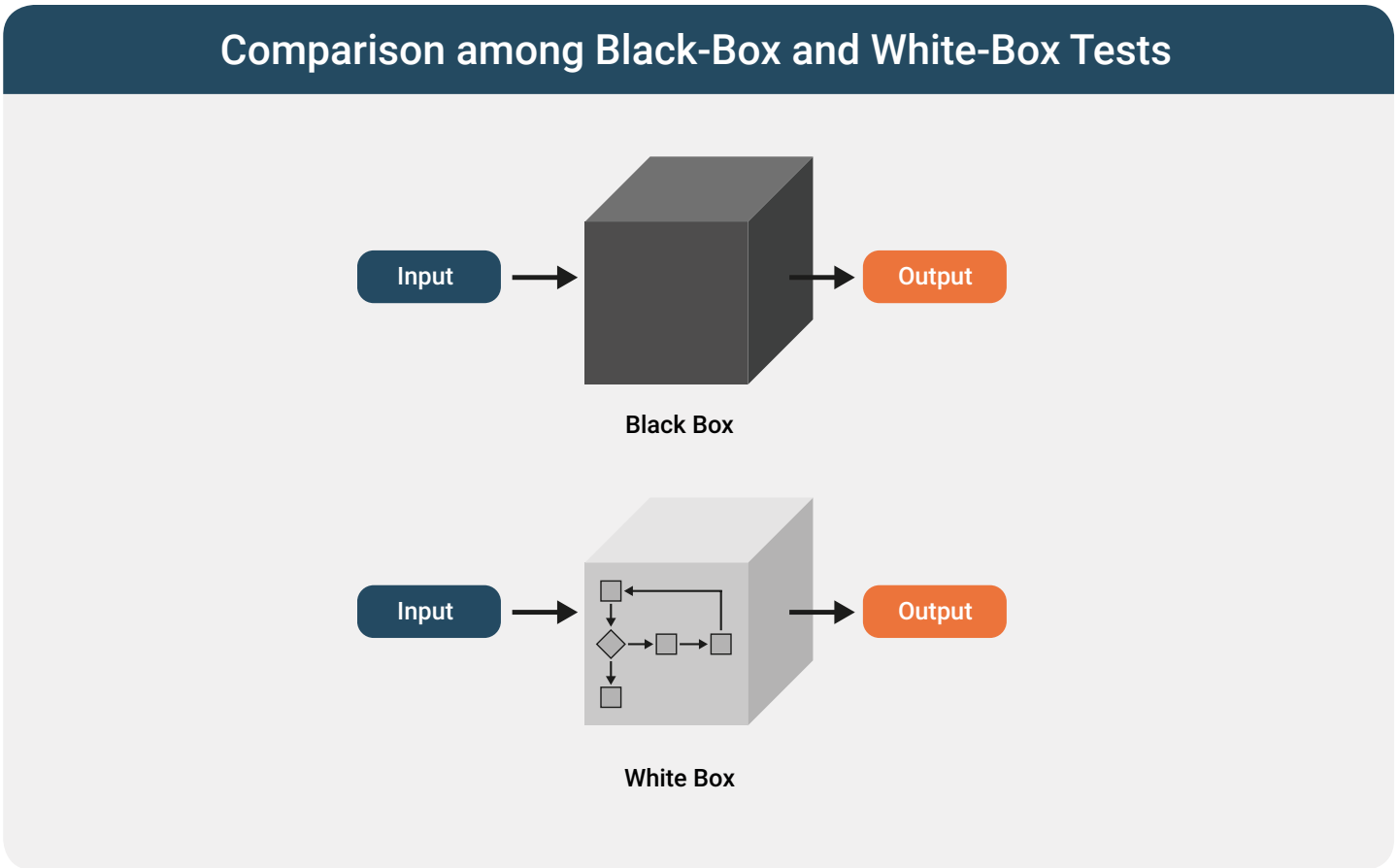

Fig 1. Black Box Monitoring vs. White Box Monitoring

Rather than a point-in-time assessment like that provided by a probe, this approach to monitoring provides a rich, multi-faceted, continuous insight into the state of the application's health. Tools that provide white box monitoring usually provide a visualization feature, or the means for exporting data to be visualized in other tools.

So, what's on offer to us? There are some very capable commercial products that provide comprehensive monitoring for infrastructure as well as applications such as New Relic, AppDynamics, Instana, and Datadog. These commercial solutions tend to be SaaS-based and come at a price. The price, however, may well be worth the investment for the value that can be gained from a comprehensive, hosted monitoring solution.

We should also mention that the major public cloud providers also provide monitoring solutions for applications that run on their infrastructure; Google Stackdriver, AWS CloudWatch and Azure Monitor. Using a tool native to the platform in question, may make a lot of sense if your stack is based entirely on a specific cloud platform. But bear in mind that these tools are not as feature rich as they could be at the present time.

By far the most popular monitoring tool that is used for cloud-native applications, is the open source Prometheus. Prometheus is a graduated project under the stewardship of the CNCF, and has followed closely behind Kubernetes in its journey to maturity. It's incredibly well-respected in the cloud-native community, and is frequently used to monitor production cloud-native stacks.

With a multi-dimensional data model, Prometheus applies key-value pairs to metrics in order to generate time series data that can be queried in a fine-grained manner. It also provides a set of client libraries for instrumenting application services, with most popular programming languages supported. And if there's a need to translate metrics from other systems that don't support the Prometheus format, there are a large number of 'exporters' available that enable you to gather metrics from these systems. The Prometheus metrics format forms the basis of an open standard for metrics exposition, called OpenMetrics.

Perhaps one of its biggest selling points for the cloud-native stack, is its ability to interact with service discovery mechanisms. This enables it to automatically discover what to collect metrics from. Additionally, when

cloud-native applications are deployed to a Kubernetes cluster, because Kubernetes itself exposes metrics in the Prometheus format, it's possible to start to construct a complete picture of the health of the entire stack.

Whilst Prometheus has an in-built visualization capability, it's fairly limited in nature, and most organizations that use Prometheus seriously, make use of Grafana instead. Grafana can use Prometheus as a data source, and provides a comprehensive, general-purpose graphing and dashboard capability. Prometheus also has an alerting mechanism for issuing notifications to a number of different supported backends (e.g. Slack, OpsGenie, PagerDvuty, etc.) or via a webhook receiver.

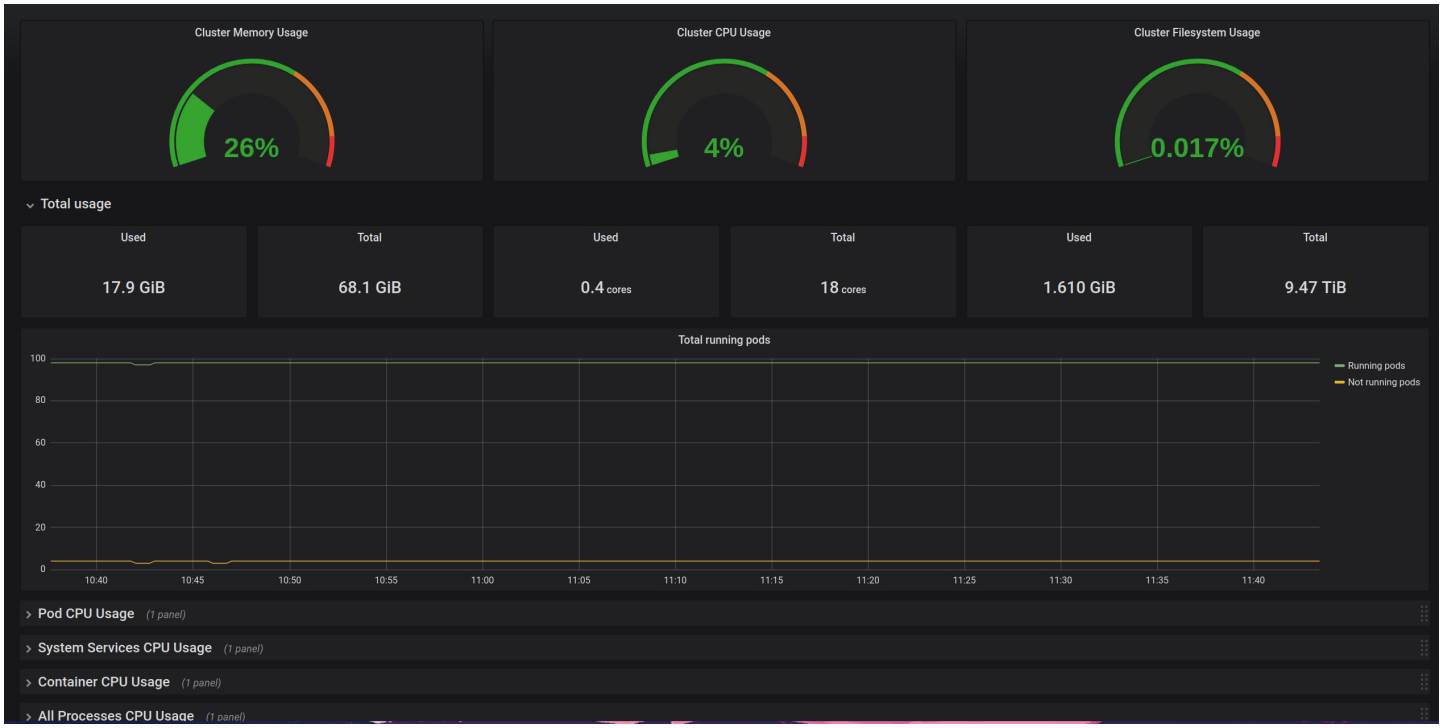

Fig 2. Screenshot of the Grafana Dashboard

Logging

Monitoring is an essential ingredient in the determination of the state of health of cloud-native applications, but it relies on us knowing which metrics to collect. This isn't always known for every application from the outset, and there is a degree of learning about the behavior of the applications we develop over time. Once we're entirely intimate with how our application behaves, then we can fine tune our metrics to monitor for known problem circumstances in production.

As a complement to monitoring, we can and should log events during the execution of our applications. Logs are often described as a 'set of time-ordered events' that are emitted by an application or system, which are stored for subsequent analysis. Event logging has long been a staple component that supports deep introspection of application behavior, and the wider system within which the application resides. When things go wrong with applications or systems, logging onto a host server and poring over copious amounts of logged events, is an obligatory problem solving technique.

But, just as cloud-native applications cause problems for traditional monitoring practices, they also pose a challenge for the collection and analysis of event logs. Once again, it's the ephemeral, distributed nature of these applications that are the crux of the problem. How can we examine the logs of an application service instance, if it's been and gone before we've even found out where it's running? What happens to the logs when a host node fails and is replaced by a healthy alternative? How do we make sense of the myriad of logs generated by multiple sources across the application environment?

To answer that question, it's worth first looking at what the '12-factor app' methodology has to say about application logs. In short, it states logs should be treated as unbuffered event streams, and that their management shouldn't be the concern of the developer, but that of the environment in which the application runs.
In other words, developers insert code to output events at key points in their application, but this is simply written to the stdout and stderr streams. For cloud-native environments, this means extending the stack to ensure this valuable information is collected, stored, and made availablefor subsequent analysis.

Enhancing the Stack

The tools that you choose to use for managing event logs will depend on many different factors.

The gorilla of the logging world is Splunk, which has served large enterprises for well over a decade. As an enterprise-grade solution, it's scope extends beyond traditional logging, and can be used for eliciting business insights, too. If you're a large organization with deep pockets, looking for a comprehensive solution that goes beyond standard logging for operational purposes, it might be the right choice for you. But if your requirements are less expansive, then there are plenty of other choices to be had.

Just like the monitoring domain, SaaS-based solutions exist for logging too. Sumo Logic's capabilities and market proposition is very similar to that of Splunk, whilst Loggly was a pioneer in the SaaS market for logging, and was acquired by Solarwinds in 2018. Bear in mind that if you elect to go the SaaS route for log management, you may end up shifting gigabytes of data over the wire on a daily basis.

The ELK (Elasticsearch, Logstash, Kibana) stack has dominated the open source approach to log management for a number of years. Elasticsearch provides the indexing and query capabilities, Logstash performs the log collection and parsing function, and Kibana enables visualization of the information collected. In more recent times, an increasingly popular alternative to the Logstash component of the logging stack, is a CNCF-hosted project called Fluentd, which is a little more efficient on memory consumption. Just as the different Beats can be configured to ship logs to Logstash, Fluentd can be the forwarding target for Fluent Bit, a lightweight collector of logs with input plugins for a number of different data sources. Fluent Bit can be deployed to ship logs without Fluentd running alongside it. In either case the logging stack is referred to as EFK then.

Standing up and managing the infrastructure to host an ELK or EFK stack, however, is not a trivial exercise, and it quickly becomes an end in itself. If you can't or don't want to make the investment in skills and infrastructure, then there are managed service providers who will take care of running the stack on your behalf.

Before we leave the topic of logging, it's worth pointing out that variations of the ELK stack don't have a monopoly when it comes to open source solutions, and

there are other reputable tools available, such as Graylog and Grafana Loki. Loki, whilst very new, is particularly interesting as it is inspired by Prometheus, and can use the very same Kubernetes labels that Prometheus uses, to analyze the collected logs in Grafana. Switching between metrics and logs using the exact same labels in one visualization tool, is incredibly powerful.

Distributed Tracing

The recent innovations in monitoring and logging get us a long way in observing what's happening in our cloud-native applications; but only so far. Cloud-native applications are comprised of a set of distributed microservices. As a client request hits one of those services, it might spark a complex transaction pattern across numerous other services. If we suffer performance or latency issues during the course of the request transaction, it's going to be very hard to stitch the metrics and logs together to build a picture of what's going on. To help build that picture, and to give us a fighting chance of pinpointing the cause of what has manifested as a performance issue, there needs to be a way of tracing the request or transaction as it traverses the mesh of services that make up the application. The techniques which provide this kind of insight, belong in the domain of distributed tracing.

Distributed tracing is not a new concept, and has been used by the likes of Google and Twitter for a number of years. But it's been the lack of a standard that has prevented its widespread adoption in cloud-native environments.

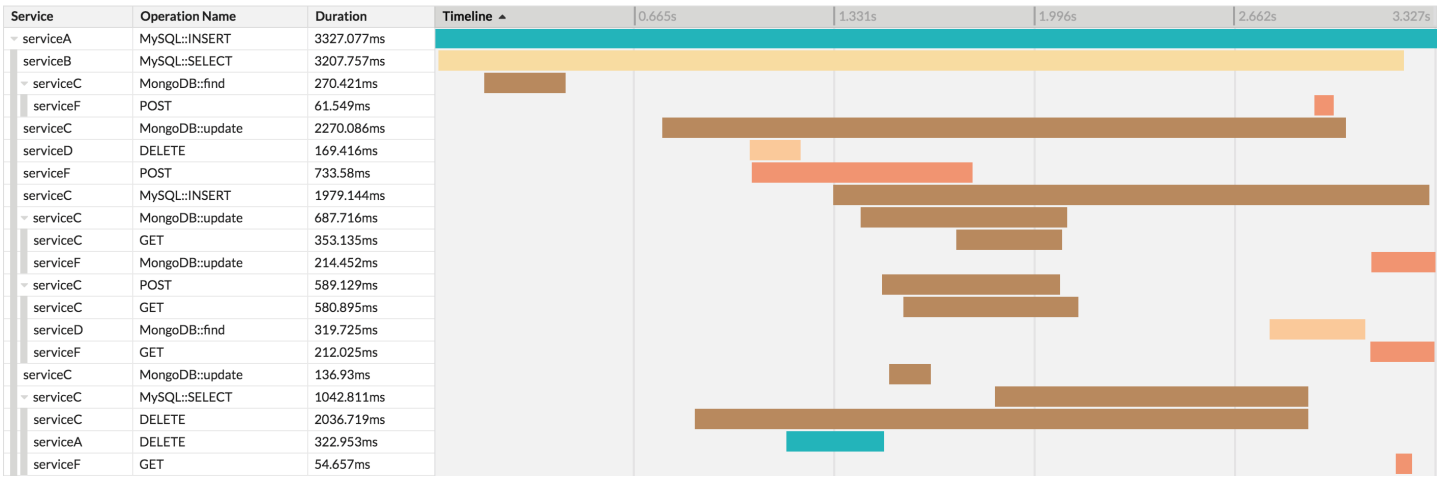

Fig 3. Screenshot of Tracing with Jaeger

If your services rely on libraries, frameworks or services, which don't all provide the means to instrument in the same manner, then there will be breaks in the trace which will degrade the insights that can be learned.

Open Tracing is an incubating project of the CNCF that aims to derive a vendor neutral API specification for instrumenting cloud-native applications. A trace refers to the picture that we want to build of the transaction as it winds its way through the distributed services, with each step in the journey referred to as a span. A context is passed along from span to span, and contains the ID of the trace and span, as well as any other data that could be useful for carrying across each span boundary. Developers use the same instrumentation in their code, irrespective of the tracing system they adopt, provided that system supports the Open Tracing API specification. In theory, this means it's possible to instrument once, but swap the tracing system that provides the visualization and insights, without having to re-instrument your code.

In terms of tracing systems, there is a growing list available to choose from. The most mature is Zipkin, which has a number of libraries for different programming languages (e.g. C#, Java, Go, JavaScript), which are supported by its open source project. Additionally, it has a large number of community s upported libraries. Jaeger is a more recent and very popular alternative, and is an open source project hosted by the CNCF. It makes use of the language support provided by the Open Tracing project, which includes Go, JavaScript, Java, Python and so on. When researching which tracing system you intend to adopt, it will pay to observe (no pun intended) the languages and frameworks that are supported.

There are, of course, commercial solutions available, including SaaS offerings like LightStep, Instana, and Datadog.

Observability

Before we leave this topic, it's worth pointing out that monitoring, logging, and tracing enable us to detect anomalies in our applications and systems. Though, in isolation they don't really help us understand the behavior of the application itself. Combined, however, they provide us with a rich context, which enables us to gain insights that we would otherwise not be able to glean from our applications. There is a trend to refer to this imbued competence, as 'observability'.

Earlier on we suggested that systems like Kubernetes do a particular job and do it well. We also mentioned that they need to be augmented with additional tooling to be the rounded solution required for cloud-native applications. In that sense, Kubernetes has been likened to a kernel, that manages and schedules its guest workloads. To make the job of running cloud-native workloads more seamless in a Kubernetes environment, the community produces helpful in-cluster operators, such as the TLS certificate management operator, cert-manager.

They also build abstractions on top of Kubernetes, and one of the more important of these is the 'service mesh'.

The term service mesh comes from the notion that cloud-native applications implemented as microservices, don't necessarily conform to the tiered nature of previous generations of application architectures; instead they present a loosely-connected mesh of interacting services. Suddenly, client traffic traverses east-west, as well as north-south. Managing and controlling the traffic between these ephemeral services is non-trivial. The need to do this has spawned a new management layer on top of the orchestration system, called service mesh technology.

The goal of service mesh technology or tooling is to provide the means for cluster administrators to apply traffic management policy to service meshes, whilst securing the communication, and enabling introspection of traffic behavior. Policy and configuration is defined using a control plane component, and it's enforced in the service mesh courtesy of sidecar proxies that constitute a data plane. Using service mesh technology, we can define client request routing, configure ingress and egress and deal with timeouts and retries. We do this, while relying on the data plane to handle service discovery and load balance traffic to service instances.

The capabilities you find in service mesh technology are not new concepts, and it's possible to trace early implementations of service meshes back to libraries like Google's Stubby, Netflix's Hystrix, and Twitter's Finagle. But libraries have limited appeal, particularly as updates require a complete update and rollout of the entire service mesh. The sidecar proxy approach removes the need for this disruptive course of action.

The first proxy-based service mesh technology was Linkerd, a CNCF hosted open source project initiated by Buoyant. It found its way into production at a number of early adopters, including

Monzo Bank. Its first incarnation (version 1), was based on Finagle. It was daemon-based, and was followed by a lighter, sidecar proxy version originally called Conduit.

Conduit, with a control plane written in Golang and a data plane provided by a proxy technology written in Rust, was targeted specifically at Kubernetes. It has subsequently been subsumed into Linkerd as version 2, and the heavier JVM-based version 1 is likely to be retired in favor of version 2 at some point in the future.

Hot on the heels of Linkerd, a series of companies announced an open source collaboration around another service mesh technology called Istio. The project started with Google, IBM and Lyft but it has subsequently gained rapid support from a variety of other companies.

It is generally accepted as the leading technology in service mesh technology. It uses Lyft's popular open source Envoy project as its sidecar proxy, and has a number of supported adapters for interfacing with third-party infrastructure backends. For example, there are adapters for Fluentd, Prometheus, and Jaeger, amongst a host of others.

With Linkerd 2 playing catch up with Istio in terms of features and popularity, Istio certainly gets the most attention in the community. This may be as a result of the corporate weight behind the project, but it's too early to say which technology will win as the de-facto standard for service mesh technology. Istio is certainly complex, whilst Linkerd and the recent Connect feature of Hashicorp's Consul, provide a simpler approach to the management of service meshes. There's plenty of road left to run in the service mesh race, so it would be wise to take a carefully considered decision, when it comes to defining your approach.

"Kubernetes is the new kernel. We can refer to it as a "cluster kernel" versus the typical operating system kernel." -- **Jessie Frazelle**

Conclusion

Embarking on the journey to cloud native adoption may seem daunting, especially when you consider that the cloud-native paradigm is still relatively new, and that it seems like there is so much to learn and to assimilate. And, for sure we haven't even touched on other aspects of the cloud-native stack, such as continuous integration/deployment, application registries and distribution, networking and so on.

If we are realistic and pragmatic in our approach, we can look forward to realizing significant benefits for the organizations we serve, by taking advantage of the availability of excellent open source platforms, tools, and methods that make cloud-native stacks viable, and the body of knowledge and experience that is willingly and collaboratively shared within the community.

About Giant Swarm

Giant Swarm provides an API Driven Open Source Platform that enables businesses to easily provision and scale Kubernetes clusters together with a wide set of managed services. Giant Swarm continuously updates the platform with all its tenant clusters and takes full responsibility that the cloud-native infrastructure is operational at all times. Giant Swarm provides its customers hands-on expert support via a private Slack channel to further accelerate their cloud-native success.

Get in touch with us now

Let's have a first call where we can discuss your goals, solutions, setup and timeline to come up with the perfect plan for scaling and delivering your solutions with ease.

 hello@giantswarm.io

 www.giantswarm.io

 twitter.com/giantswarm